

Объектно ориентированное программирование на Java

Тимур Машнин

Тимур Машнин

**Объектно-ориентированное
программирование на Java.
Платформа Java SE**

«Издательские решения»

Машнин Т.

Объектно-ориентированное программирование на Java. Платформа Java SE / Т. Машнин — «Издательские решения»,

ISBN 978-5-00-503960-6

Эта книга предназначена для тех, кто хочет научиться программировать на языке Java. С этой книгой вы обучитесь объектно-ориентированному программированию на платформе Java SE и научитесь применять принципы ООП на практике. Эта книга охватывает важные аспекты программирования на языке Java, начиная с основ и заканчивая объектно-ориентированным подходом и командной разработкой кода.

ISBN 978-5-00-503960-6

© Машнин Т.
© Издательские решения

Содержание

Введение	7
Выражения	13
Основные операторы	19
Переменные	22
Строки и печать	25
Условия if и else	29
Выражение switch	33
Тернарный оператор	36
Циклы while и for	41
Массивы	46
Представление данных и типы данных	51
Методы	57
Область видимости переменных	62
Комментарии. Javadoc	71
Исключения	76
Рекурсия	84
Инкапсуляция. Объекты и классы	94
Классы и типы	98
Область видимости	102
Наследование	106
Приведение типов	110
Полиморфизм	113
Переопределение и перегрузка	115
Примитивы и объекты	123
Абстракция	133
Интерфейсы. Абстрактные методы и классы	135
Пакеты	143
Абстрактные классы vs Интерфейсы	148
Интерфейсы программирования API. Стандартная библиотека Java	151
Вложенные классы	162
Перечисления	166
Компиляция и выполнение программ	170
Модульность	178
Моделирование с UML	185
Синтаксические ошибки	188
Выявление ошибок	194
Отладка кода	200
Тестирование кода	203
Модульное тестирование	210
Интеграционное тестирование	218
Рефакторинг кода	228
Java Collections Framework	237
Общие понятия	239
Структурированные данные	241
ArrayList	251
HashMap	258

Дженерики	265
Потоки коллекций и фильтры	271
Коллекции в Java 9	276
Java Reflection	278
Лямбда-выражения. Синтаксис лямбда	287
Функциональные интерфейсы	296
Потоки Stream	303
Параллельные и бесконечные потоки	313
Потоки Stream в Java 9	319
Java Date/Time API	322
Основы ввода-вывода	328
Сериализация	341
Символьные потоки	348
Java NIO	356
File NIO	371
Ввод-вывод в Java 9	380
Хранение данных	382
JDBC	384
Пример	387
PreparedStatement	396
Транзакции	397
DataSource	399
Локализация и интернационализация. Введение	401
Наборы ресурсов	403
Интернационализация чисел, валюты, даты и времени	408
Проверка вводимых данных	414
Основы сетевого взаимодействия	419
Сокеты	424
Серверный сокет	427
Клиентский сокет	429
Использование URL	431
Обмен Java объектами	434
UDP, широковещательные сообщения, многоадресная рассылка	437
Remote Method Invocation	441
HTTP/2 клиент в Java 9	445
Разработка ПО	451
Системы контроля версий	457
CVS	459
Subversion	461
Subversion в IntelliJ IDEA	463
Git	470
Git в IntelliJ IDEA	473
Основы системы безопасности Java. Введение	482
Менеджер безопасности	491
Привилегированные блоки	501
Защищенные объекты	506
Введение в Java криптографию	508
Целостность и конфиденциальность данных	515
Аутентификация и авторизация	526

Объектно-ориентированное программирование на Java Платформа Java SE

Тимур Машнин

© Тимур Машнин, 2019

ISBN 978-5-0050-3960-6

Создано в интеллектуальной издательской системе Ridero

Введение



Объектно-ориентированное программирование на Java

Введение

Лекция

Общий обзор технологии Java

На этом курсе мы будем изучать технологию Java.

Итак, что такое технология Java?

Начнем с самого понятия технологии программирования.

Технология программирования – совокупность методов и инструментов, позволяющих создавать программное обеспечение

Технология Java – это язык программирования Java и платформа Java.

Можно сказать, что *технология программирования* – это совокупность методов и инструментов, позволяющих создавать программное обеспечение.

Технологии программирования могут иметь различный уровень применения. В процессе разработки программного обеспечения могут применяться технологии, решающие как конкретные задачи, так и технологии, являющиеся платформой для создания частей приложения или всего приложения.

Поэтому, как правило, для создания программного обеспечения применяется целый набор различных технологий.

Применительно к Java, *технология Java* – это язык программирования Java и платформа Java.

Язык программирования Java представляет собой объектно-ориентированный язык программирования, имеющий синтаксис, близкий к синтаксису языка C++.

Отличия языка Java от языка C++ обусловлены самим происхождением этих языков программирования.

Язык C++ является расширением языка C, который создавался как язык системного программирования.

Java vs C++

Ограничения работы с памятью: нет доступа к указателям и отсутствует возможность динамического выделения памяти - устраняется ненадежность кода

Более строгая типизация: дескрипторы объектов в Java включают в себя полную информацию о классе, представителем которого является объект, так что Java может выполнять проверку совместимости типов

Автоматическая сборка мусора: периодическое освобождение памяти с удалением объектов, которые уже не будут востребованы приложениями

Язык Java, в свою очередь, создавался для решения задач сетевого программирования и является самостоятельным языком программирования.

Главные отличия языка Java от языка C++ – это более строгая типизация, ограничения работы с памятью, автоматическая сборка мусора.

Понятно, что для создания программного обеспечения наличие одного языка программирования недостаточно.

Для компилируемых языков нужны инструменты, компилирующие исходный код в машинный, исполняемый операционной системой компьютера.

Для интерпретируемых языков программирования необходимы интерпретаторы, выполняющие исходный код в операционной системе.

В случае языка Java, реализация платформы Java как раз и обеспечивает выполнение Java-кода в операционной системе компьютера.

Таким образом, для того чтобы Java-приложение могло быть запущено, необходима реализация платформы Java.

Мы упомянули реализацию платформы Java.

Что это такое?

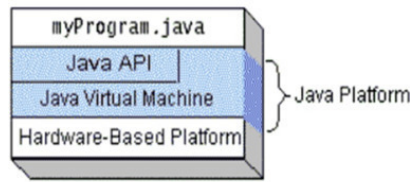
Платформа Java состоит из виртуальной машины Java Virtual Machine (JVM) и библиотек интерфейса программирования Java Application Programming Interface (API).

Реализация платформы Java – это конкретная реализация JVM для конкретной операционной системы плюс библиотеки Java API

Для всех распространенных операционных систем существуют свои виртуальные машины JVM, тем самым реализуется принцип «Write Once, Run Anywhere» – написанное однажды, работает везде.

Реализация платформы Java – это конкретная реализация JVM для конкретной операционной системы плюс библиотеки Java API.

На самом деле компанией Oracle для выполнения Java-приложений предоставляется набор сред выполнения *Java Runtime Environment (JRE)*, охватывающий все распространенные операционные системы.



Виртуальная машина JVM составляет основную часть среды выполнения Java Runtime Environment (JRE).

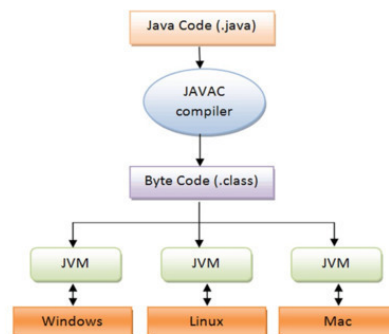
Помимо JVM JRE содержит базовые библиотеки API, необходимые для выполнения Java-приложений, а также дополнительные инструменты, включая Java Plug-in – для запуска апплетов в браузере и Java Web Start – для развертывания Java-приложений через Интернет.

Компанией Oracle также предоставляется минимальный комплект разработки Java-приложений *Java Development Kit (JDK)*, состоящий из набора инструментов, включая компилятор в байт-код `javac`, документации, примеров и среды выполнения JRE.

Язык программирования Java является одновременно и интерпретируемым, и компилируемым. Причина этого кроется в устройстве виртуальной машины JVM.

Виртуальная машина JVM – это набор специальных программ, созданных для конкретной операционной системы.

Точкой входа в виртуальную машину JVM является программа `java`, запускающая Java-приложение.



Приложения, написанные на языке Java, представляют собой текстовые файлы с расширением. `java`.

Чтобы JVM выполнила Java-приложение, приложение должно быть откомпилировано в специальный двоичный формат – *байт-код*.

Откомпилированное Java-приложение состоит из файлов с расширением. `class`, которые могут быть упакованы в архивный исполняемый файл с расширением. `jar`.

При запуске Java-приложения на вход JVM подается байт-код Java-приложения, а также байт-код используемых приложением библиотек Java API.

Виртуальная машина JVM может выполнять приложения, написанные и на других языках программирования – Scala, Groovy, Ruby, PHP, JavaScript, Python и др., при этом приложения также должны быть откомпилированы в байт-код.

В процессе обработки байт-кода виртуальная машина JVM производит его интерпретацию, т.е. выполняет команды, содержащиеся в байт-коде, или использует компилятор Just-in-time compilation (JIT), который транслирует байт-код в машинный код непосредственно во время выполнения Java-приложения, и тем самым увеличивает скорость обработки байт-кода.

Таким образом, язык Java является компилируемым, потому что необходима компиляция исходного кода в промежуточный по отношению к машинному байт-коду, и интерпретируемым, потому что байт-код не может быть исполнен самой операционной системой компьютера, а должен интерпретироваться.

Платформа Java содержит два типа JVM:

Java HotSpot Client VM (Client VM). Вызывается опцией – client инструмента java и обеспечивает быстрый запуск и потребление небольшого объема оперативной памяти.

JVM:

- Java HotSpot Client VM (Client VM): вызывается опцией – client инструмента java и обеспечивает быстрый запуск и потребление небольшого объема оперативной памяти.
- Java HotSpot Server VM (Server VM): вызывается опцией – server инструмента java и обеспечивает максимальную скорость выполнения приложения.

Java HotSpot Server VM (Server VM). Вызывается опцией —server инструмента java и обеспечивает максимальную скорость выполнения приложения.

Для обеих JVM технология Java HotSpot оптимизирует обработку байт-кода, распределение памяти, сборку мусора и управление потоками.

Технология Java – это общее понятие, на самом деле обозначающее широкий спектр Java-технологий.

Среда выполнения JRE и комплект разработки JDK являются основными продуктами платформы Java Platform, Standard Edition (Java SE).

Как уже было сказано, платформа Java содержит библиотеки интерфейса программирования Java API. Для чего они предназначены и какую роль они выполняют?

Библиотеки Java API – это готовые классы и интерфейсы, обеспечивающие для создаваемых Java-приложений общую функциональность.

Библиотеки Java API – это готовые классы и интерфейсы, обеспечивающие для создаваемых Java-приложений общую функциональность.

С библиотеками Java API программисту не нужно самому реализовывать ввод-вывод, сетевое соединение, создавать стандартные графические компоненты для интерфейса пользователя и многое-многое другое.

Все это уже предоставлено технологией Java.

Платформа Java SE является основой для всех остальных платформ технологии Java. Все вместе Java-платформы обеспечивают применение технологии Java к широкому диапазону устройств – от смарт-карт, встроенных и мобильных устройств до серверов и суперкомпьютеров.

Технология Java представлена следующими платформами:

Java Platform, Standard Edition (Java SE) – предоставляет среду выполнения и набор технологий и библиотек API для создания и запуска серверных и настольных приложений, апплетов и является основой для остальных платформ.

Java Platform, Standard Edition (Java SE) – предоставляет среду выполнения и набор технологий и библиотек API для создания и запуска серверных и настольных приложений, апплетов и является основой для остальных платформ.

Java SE Embedded – предназначена для встроенных систем, таких как интеллектуальные маршрутизаторы и коммутаторы, профессиональные принтеры и др.

Java Platform, Micro Edition (Java ME) – содержит набор сред выполнения и библиотек API, предназначенных для встроенных и мобильных устройств.

Java Card – позволяет создавать и запускать небольшие приложения (Java Card-апплеты) в смарт-картах и других устройствах.

Java Platform, Enterprise Edition (Java EE) – является расширением платформы Java SE и добавляет библиотеки, позволяющие создавать распределенные, многоуровневые серверные Java-приложения.

Кроссплатформенность обеспечивается наличием сред выполнения для различных операционных систем.

Платформа Java SE включает в себя следующие компоненты – среду выполнения Java Runtime Environment (JRE) и комплект разработчика приложений Java Development Kit (JDK).

Java SE Embedded – предназначена для встроенных систем, таких как интеллектуальные маршрутизаторы и коммутаторы, профессиональные принтеры и др.

Платформа Java SE for Embedded обеспечивает ту же функциональность, что и платформа Java SE, дополнительно добавляя поддержку для платформ, специфических для встроенных систем, оптимизацию использования памяти, а также предоставляя уменьшенную среду выполнения и опцию Headless для устройств, не имеющих дисплея, мышки или клавиатуры.

Java Platform, Micro Edition (Java ME) – содержит набор сред выполнения и библиотек API, предназначенных для встроенных и мобильных устройств. В настоящее время активно применяется для Интернет вещей.

Java Card – позволяет создавать и запускать небольшие приложения (Java Card-апплеты) в смарт-картах и других устройствах с очень ограниченными ресурсами, таких как SIM-карты мобильных телефонов, банковские карточки, карты доступа цифрового телевидения и др.

Java Platform, Enterprise Edition (Java EE) – является расширением платформы Java SE и добавляет библиотеки, позволяющие создавать распределенные, многоуровневые серверные Java-приложения.

Если сравнивать язык Java с такими распространенными языками как C#, JavaScript, Python и PHP,

То сравнивая Java с C#, который работает на платформе NET, с точки зрения разработчика языки Java и C# очень похожи.

Но у них есть некоторые синтаксические различия, и язык Java считается более простым языком.

Кроме того, С# все таки больше привязан к платформе Windows.

Так как эти два языка очень похожи, при их сравнении возникают большие дискуссии, в которые мы сейчас углубляться не будем.

Если сравнивать Java и JavaScript, язык JavaScript является только интерпретируемым и выполняется только в веб-браузерах.

Если сравнивать Java и Python, то Python также является компилируемым и интерпретируемым языком, но с полной динамической типизацией, он проще в изучении, но проигрывает в скорости Java, хотя для него есть альтернативные реализации интерпретаторов: Jython, Cython и другие.

По поводу сравнения Java и Python также ведутся жаркие дискуссии.

Если сравнивать Java и PHP, то PHP это скриптовый серверный язык для разработки веб приложений, он проще в изучении и является языком с динамической типизацией. PHP не предназначен для крупных проектов, однако, PHP хостинг более распространен, чем для Java.

Как видно у каждого языка есть свои плюсы и минусы, но «Вы должны писать на языке, который делает вас счастливее», как сказал Пэт Аллан.

Выражения



Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 1

Выражения

Теперь каким способом можно хорошо представить, что такое компьютер?

И какие есть концепции языка программирования?

Вместо того, чтобы начинать с нуля, мы начнем с устройства, которое вам очень хорошо известно, а именно, калькулятора.

И мы постепенно преобразуем простой калькулятор в компьютер.

Сделав это, мы также перейдем от ввода последовательности клавиш калькулятора к компьютерной программе.

Таким образом, вы гораздо лучше поймете концепции, на которых основывается компьютер и языки программирования, такие как Java.

Вы использовали калькулятор много раз.

И вы знаете из чего он состоит.



Здесь есть клавиши с цифрами, которые помогут вам составить число.

Составленные числа отображаются на дисплее.

И тогда вы можете выполнять операции с этими числами,

Для которых вы используете другие клавиши, которые представляют эти операции.

Мы начнем с рассмотрения базового калькулятора.

Таким образом, эти операции могут быть сложение, вычитание, умножение, и деление.

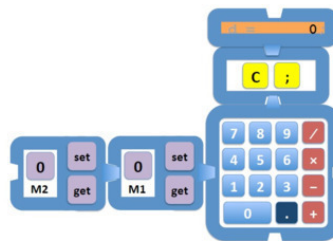
И калькулятор состоит из трех основных частей – дисплея, контрольной части, где есть завершающая вычисления клавиша равно и клавиша сброса, и клавиатура с цифрами и операциями.

Теперь, используя цифры, вы можете писать выражения и запрашивать их вычисления.

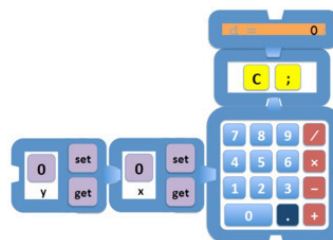
Выражение содержит числа и арифметические операции.

Эти выражения называются числовыми выражениями.

Теперь давайте улучшим этот калькулятор.
 Но сначала давайте поговорим о выражениях.
 Как правило, мы думаем о выражениях математически.
 Это выражение равно другому выражению или какому-либо значению.
 И это очень хорошая абстракция в большинстве случаев.
 Но на самом деле мы знаем, что вычисление выражения требует усилий и времени.
 И если у нас есть более сложное выражение, в нем может быть порядок, согласно которому вычисляются разные части этого выражения.
 И вычисление более сложного выражения может занять больше времени.
 Но что более важно, представьте, что у нас есть сложное выражение, и мы вычислили его один раз.
 И мы должны снова вычислить его, если мы хотим позже получить значение выражения.
 Хотя было бы неплохо иметь некий способ запомнить значение выражения для будущего использования?
 Поэтому, рассмотрим такой калькулятор, где у нас есть запоминание.
 Здесь у нас есть несколько клавиш для хранения или получения значений из этой памяти.



Функция запоминания позволяет нам сохранить значение для будущего использования.
 Память может содержать значение, и могут быть связанные с ней операции, такие как MS, чтобы сохранить значение, и MR, чтобы восстановить его или вызвать его.
 Иногда есть третья клавиша, MC для очистки памяти,
 Назовем эти две клавиши для работы с памятью set и get.
 Сейчас ячейки памяти названы предопределенными именами, M1, M2 и т. д.
 Но мы хотели бы назвать их x и y, как мы привыкли в математике.
 И мы будем присваивать этим ячейкам памяти имена переменных.



Теперь мы обсудим, что такое начальное значение переменной, которое сохраняется до того, как мы установим переменную в другое значение.

Мы можем сказать, что значение переменной неопределенно.

Поэтому, если мы попытаемся получить это значение, мы получим ошибку.

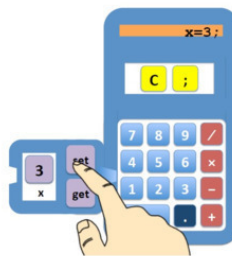
В калькуляторах, где есть числовые переменные, эта переменная обычно устанавливается равной 0, чтобы избежать ошибки.

Теперь мы хотим, чтобы дисплей показывал что-то, когда мы нажимаем кнопки Set или Get.

Давайте сначала поговорим о Set.

Предположим, что дисплей показывает число 3, и что мы нажимаем кнопку set переменной x.

Теперь значение 3 будет храниться в переменной x.



И дисплей может показать что-то вроде x равно 3 точка с запятой,

Чтобы записать то, что мы только что сделали.

Мы говорим, что мы назначили значение 3 переменной X, и записали это как x равно 3 в инструкции присваивания.

Как только мы установили значение переменной, мы можем использовать это значение в выражениях.

Например, представьте, что у нас есть 5 на дисплее,

И мы хотим добавить значение x.

Мы нажимаем символ плюса, а затем кнопку Get x.

Таким образом, мы увидим на дисплее 5 плюс x.



Но это выражение, и до того, как мы используем оператор присваивания, что дисплей действительно отображает, выражение или законченную операцию?

Мы можем рассматривать выражения в калькуляторе как законченные операции, считая, что дисплей также может считаться переменной, переменной с прямым вводом.

Поэтому на дисплее написано d равно перед выражением.

Таким образом мы преобразуем выражение в операцию.

На слайде показаны различные выражения присваивания.

```
x = 3;
x = 3+4;
x = 3+y;
x = 3+x;

<Variable> = <Expression> ;
```

Здесь показано, что выражения могут также иметь переменные.

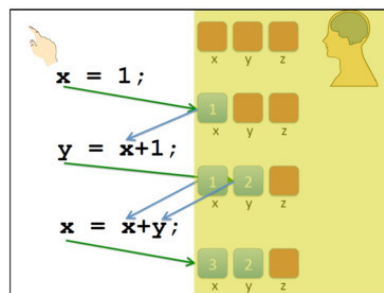
И для вычисления выражения, нам нужно найти сохраненное значение в соответствующих переменных.

Теперь может оказаться, что одна и та же переменная появляется как слева, так и справа от присваивания.

Давайте проанализируем это более подробно.

Но сначала, давайте вспомним, что выражение присваивания состоит из переменной, за которой следует символ равенства, за которым следует выражение для вычисления, которое завершается точкой с запятой.

Представьте, что мы имеем три переменные x, y и z.



Мы не знаем их начальных значений.

У нас есть первая операция, которая присваивает 1 переменной x.

Поэтому после выполнения содержимое переменной x равно 1.

Следующая операция присваивания y равно x плюс 1.

Сначала мы должны оценить выражение справа, x плюс 1.

Для этого нам нужно получить значение, сохраненное в x.

Поэтому мы получаем 2 и 2 сохраняем в y.

Мы всегда работаем справа налево.

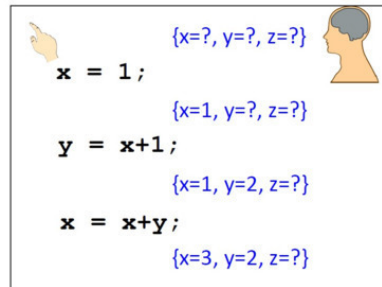
Сначала вычисляем выражение, а затем сохраняем результат в переменной.

Теперь мы сначала получаем значения x и y , складываем их вместе, получаем 3 и сохраняем 3 в x .

Переменные вместе со значениями – это то, что мы называем состоянием.

Таким образом, оператор присваивания преобразует одно состояние в другое состояние.

Здесь состояния обозначены фигурными скобками.



Коллекция значений переменных – это состояние.

Поэтому присваивание приводит к изменению состояния.

Теперь представьте, что вы сегодня делаете расчеты, и вы хотите повторить те же самые вычисления завтра.

Для этого вам нужно будет ввести все выражения снова.

Поэтому мы хотели бы иметь возможность записывать вычисления.

Точно так же, как мы хотим использовать память переменных для хранения значений, мы хотели бы теперь сохранить всю программу.

Некоторые калькуляторы печатают вычисления на бумаге.

Таким образом, у нас может быть запись наших вычислений.

Мы называем эту запись программой.

На данный момент программа является последовательностью простых вычислений.

Теперь было бы здорово, если бы мы могли повторно использовать программу, чтобы программа была не только результатом записи калькулятора, чтобы мы имели возможность подавать эту программу в калькулятор, как инструкции для повторного расчета.

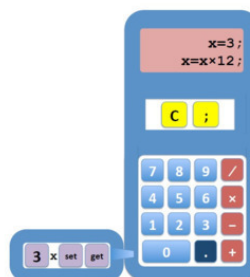
Теперь наш калькулятор становится все больше похож на компьютер.

Таким образом, последовательность инструкций является программой.

Этот набор инструкций должен быть четко определен, и каждая из инструкций должна эффективно исполняться нашим компьютером.

Теперь эти инструкции обычно представляют собой текст.

Расширенный калькулятор выглядит следующим образом.



На дисплее теперь отображается история операций.

Это то, что мы называем программой.

Помимо записи истории операций, мы также хотим возможность ввода этой истории в калькулятор.

Таким образом, мы получим простой компьютер.

В общем и целом, программа является не чем иным, как записанным вычислением.

Ее можно записать на листе бумаги или сохранить другим способом, например, в памяти компьютера.

И компьютер будет интерпретировать эту программу и выполнять вычисление каждый раз, когда это потребуется.

Таким образом, мы прошли путь от значения и выражения до программы.

Value	<input type="text" value="7"/>
Expression	<input type="text" value="3 + 4"/>
Statement	<input type="text" value="d = 3 + 4;"/>
Program	<input type="text" value="d = 2 - 1;
x = d;
d = x * d;
d = 3 + 4;"/>

Основные операторы



Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 2

Основные операторы

Калькулятор, которые мы рассматривали, работал с числами.

Мы использовали числа и операции с числами для получения чисел.

Теперь, что делать, если вы хотите сравнить два числа?

Если мы хотим проверить, например, 5 меньше 6 или нет.

Ответ может быть положительным или отрицательным, – да или нет.

Это будет утверждение истинное или ложное.

В этом случае true и false также являются значениями, но они не являются числовыми значениями.

Их называют булевыми значениями в честь математика Джорджа Була.

Существует шесть операций сравнения – меньше чем, больше чем, меньше или равно, больше или равно.

$n < m$	(less than, $n < m$)
$n > m$	(greater than, $n > m$)
$n \leq m$	(less than or equal to, $n \leq m$)
$n \geq m$	(greater than or equal to, $n \geq m$)
$n == m$	(equal than, $n = m$)
$n != m$	(not equal than, $n \neq m$)

TRUE
FALSE

И наконец, мы должны проверить, являются ли два значения равными или разными.

Результатом проверки будет булево значение true или false.

Булевы значения представляют собой тип данных с двумя значениями true и false.

Мы могли бы назвать их да или нет, или один и ноль, но мы будем называть их true и false, как это делает Java.

И так же, как у нас были арифметические операции, теперь мы имеем несколько булевых операций.

Давайте посмотрим на некоторые из них.

– Negation (not):	<code>! a</code>
– Conjunction (and):	<code>a && b</code>
– Disjunction (or):	<code>a b</code>

Отрицание, которое также называется «нет» и представлено восклицательным знаком.

Эта операция принимает одно логическое значение, один аргумент, и возвращает другое логическое значение.

Конъюнкция – это еще одна операция, также называемая «и», и она представлена двумя амперсандами.

Эта операция принимает два значения, два аргумента.

И еще одна операция – дизъюнкция, также называемая «или», и она представлена двумя вертикальными полосами.

Эта операция также принимает два аргумента.

Операция отрицания принимает одно логическое значение и возвращает также логическое значение, а именно другое.

Таким образом, отрицание true, это false и наоборот.

Операция «и» принимает два boolean значения в качестве аргумента и возвращает boolean значение.

И результат true, если оба аргумента true, и false в противном случае.

Операция или также принимает два аргумента, два булевых значения и возвращает булево значение.

Теперь результат true, если какой-либо аргумент true, и false, если оба аргумента являются false.

Мы могли бы добавить все эти операции в наш калькулятор, который бы исполнял их также успешно, как и операции с числами.

Таким образом, суммируя, в Java мы имеем следующие основные операторы.



А также оператор присваивания = равно.

Арифметические операторы

A = 10, B = 20

Оператор	Описание	Пример
+	Складывает значения по обе стороны от оператора	A + B даст 30
-	Вычитает правый операнд из левого операнда	A - B даст -10
*	Умножает значения по обе стороны от оператора	A * B даст 200
/	Оператор деления делит левый операнд на правый операнд	A / B даст 2
%	Делит левый операнд на правый операнд и возвращает остаток	A % B даст 0
++	Инкремент - увеличивает значение операнда на 1	B++ даст 21
--	Декремент - уменьшает значение операнда на 1	B-- даст 19

Операторы сравнения

A = 10, B = 20

Оператор	Описание	Пример
==	Проверяет, равны или нет значения двух операндов, если да, то условие становится истинным	(A == B) — не верно
!=	Проверяет, равны или нет значения двух операндов, если значения не равны, то условие становится истинным	(A != B) — значение истинно
>	Проверяет, является ли значение левого операнда больше, чем значение правого операнда, если да, то условие становится истинным	(A > B) — не верно
<	Проверяет, является ли значение левого операнда меньше, чем значение правого операнда, если да, то условие становится истинным	(A < B) — значение истинно
>=	Проверяет, является ли значение левого операнда больше или равно значению правого операнда, если да, то условие становится истинным	(A >= B) — значение не верно
<=	Проверяет, если значение левого операнда меньше или равно значению правого операнда, если да, то условие становится истинным	(A <= B) — значение истинно

Логические операторы

A = true, B = false

Оператор	Описание	Пример
&&	Называется логический оператор «И». Если оба операнда являются не равны нулю, то условие становится истинным	(A && B) — значение false
	Называется логический оператор «ИЛИ». Если любой из двух операндов не равен нулю, то условие становится истинным	(A B) — значение true
!	Называется логический оператор «НЕ». Использование меняет логическое состояние своего операнда. Если условие имеет значение true, то оператор логического «НЕ» будет делать false	!(A && B) — значение true

Переменные

Теперь, когда мы добавляли переменные в калькулятор, мы хотели называть их своими именами.

В Java, когда мы хотим использовать переменную, мы должны сначала представить ее с помощью объявления.

Объявление должно включать сначала тип данных переменной.



Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 3

Переменные

В Java существует несколько типов данных, предусмотренных для чисел.

На данный момент для упрощения, представьте себе, что у нас есть один тип данных, называемый «int».

«int» включает в себя как положительные, так и отрицательные целые числа, в некоторых пределах.

```
boolean b;  
int number;  
int veryLongName;
```

Таким образом, объявление переменной состоит из имени типа, затем имени переменной и точки с запятой.

Имя переменной можно выбирать с некоторыми ограничениями.

В некоторых случаях мы также называем имя переменной идентификатором переменной.

Теперь, как мы можем создавать имена для переменных?

По сути, имена – это слова, которые должны следовать некоторым правилам.

И вот некоторые правила.

Correct	Incorrect
<code>int n;</code>	<code>int int;</code>
<code>int _n;</code>	<code>int n?;</code>
<code>int n1;</code>	<code>int 1n;</code>
<code>int noMore;</code>	<code>int no More;</code>
<code>int no; int more;</code>	<code>int no; int no;</code>

Имена должны начинаться с буквы или символа подчеркивания.

И они могут содержать буквы – маленькие или заглавные буквы, цифры, и символ подчеркивания.

Другие специальные символы не допускаются.

Исключением является знак доллара, который используется в начале для автоматически генерируемых переменных.

Итак, «n» и «_n» являются правильными именами, тогда как «n?» не может использоваться.

И вы не можете использовать цифру в начале имени.

«n1» является правильным именем, а «1n» – нет.

Кроме того, есть некоторые слова, которые запрещены.

Такие как зарезервированные ключевые слова, например, «int» или «boolean», или литералы, такие как «true» и «false».


Таким образом, вы не можете иметь «int» или «true» как имя переменной.

Кроме того, в имени не должно быть пробелов.

И, наконец, будет ошибкой объявление одного и того же имени в одной и той же области видимости.

Теперь есть рекомендации по выбору имен переменных.

Recommended	Not Recommended
<code>int average;</code>	<code>int sjdfjsJDJF;</code>
<code>int finalBalance;</code>	<code>int finalbalance;</code>
<code>int ONE;</code>	<code>int one;</code>



Во-первых, имена должны иметь смысл.

Это поможет вам и другим людям понять, как использовать переменные.

Теперь, если вы хотите объединить несколько слов в одно имя, хорошей практикой является начинать каждое следующее слово с большой буквы.

И, наконец, если у нас будет переменная, значение которой не должно изменяться в программе, хорошей практикой будет написать его заглавными буквами.

И мы поставим также что-то перед «int», чтобы сигнализировать о постоянстве переменной.

После того, как мы объявили переменные, мы готовы использовать их и назначить им значения.

```
boolean b;  
int n;  
b=true;  
n=1;  
boolean b=true;  
int n=1;  
int n=true;
```

Также мы можем объявить и присвоить значения одновременно.

Строки и печать



Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 4

Строки и печать

Мы заинтересованы не только в работе с числами.

Нам также нужно работать с текстом.

Поэтому мы будем расширять теперь наш калькулятор значениями и операциями для текста.

Текст состоит из последовательности символов.

Один символ – это символ, который вы можете найти на клавиатуре.

```
• Characters • Strings
'a'          "Java"
'A'          "Hello, World!"
'1'          "2015"
'. '         " "
' '          ""
```

Строка представляет собой последовательность символов.

Строка может состоять из нескольких символов, но она может также иметь только один символ, как в этом примере строки с пробелом.

Строка также может не содержать никаких символов.

В этом случае мы говорим о пустой строке.

Обратите внимание, что мы помещаем одиночные символы в одинарные кавычки и строки в двойные кавычки.

Это позволяет нам чётко различать литералы строк и символов. Если бы и строки, и символы можно было задавать с помощью одного и того же типа кавычек, то пришлось бы при операциях проверять, символ ли это, или строка.

Теперь, что, если мы хотим иметь двойную кавычку в строке?

Метод, который мы используем, заключается в том, чтобы поставить escape-символ, обратную косую черту.

Escaping

```
"\"Hello\""
```

```
"Hello\n"
```

```
"\\ and \\"
```

Здесь внешние двойные кавычки не являются частью строки.

Они просто указывают, что у нас есть строка.

Но теперь, если обратная косая черта является символом со специальным свойством, что делать, если мы хотим иметь обратную косую черту в строке?

Тогда мы тоже ставим перед ней обратную косую черту.

Теперь это объявление переменной для строки с именем `s`, которой мы присваиваем строку, состоящую из просто символа `s`.

Variables and Strings

```
String s = "s";
```

Так что не путайте имя переменной со строкой.

Вот почему мы используем двойные кавычки.

Теперь, какие есть основные операции для строк?

Очень важной операцией является конкатенация или соединение строк.

Обратите внимание, что символ для операции конкатенации – тот же самый, что и для сложения.

Concatenation

```
String s = "s";  
String t = "top";  
String p = s + t;    "stop"  
String q = t + s;    "tops"
```

Это знак плюса.

Вы должны быть осторожны, чтобы не путать число один со строкой «1» в кавычках. В этом примере n является целым числом и s строкой.

```
int n = 1;
String s = "1";

int m = n + n;      2
String p = s + s;   "11"
String q = s + n;   "11"
```

Поэтому, если говорить n плюс n, мы складываем числа и в результате получим целое число 2.

Если, смотреть на s плюс s, мы объединяем две строки и получаем строку 11.

Интересно отметить, что разрешено писать s плюс n – строка плюс число.

Если один из операндов является строкой, другой также преобразуется в строку.

Поэтому в последнем примере целое число 1 преобразуется в строку «1»

И в результате получим строку 11.

length – это операция, которая применяется к строке и возвращает число, соответствующее количеству символов в строке.

Length

```
String s = "s";
String t = "top";
String p = "";
String q = " ";
int n = s.length();    1
int m = t.length();    3
int j = p.length();    0
int k = q.length();    1
```

Интересно отметить, что длина конкатенации двух строк – это сумма их длин.

С операцией substring мы можем извлечь часть данной строки.

Substring

```
String s = "Hello!";
String t = s.substring(2,4);  "ll"
String p = s.substring(0,2);  "He"
String q = s.substring(2,6);  "llo!"
String r = s.substring(2);    "llo!"
```

```
"Hello!"
0 1 2 3 4 5
```

Предположим, что у нас есть строка с этими 6 символами, Hello восклицательный знак. Первый символ, H находится в нулевой позиции.

Второй E в позиции 1 и так далее, до позиции 5.

Таким образом, `substring (2,4)` означает, что мы извлекаем подстроку, которая начинается в позиции 2, L, и заканчивается в позиции до 4.

Таким образом, позиция 4 не включена.

Мы включаем символы в позициях 2 и 3, два L.

`substring (0,2)` выбирает два первых символа, а `substring (2,6)` остальные.

Также возможно написание одного аргумента в `substring`.

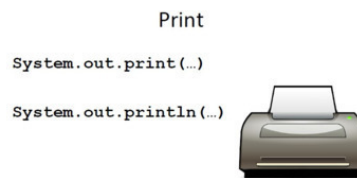
Это означает, что подстрока выбрана до конца строки.

Теперь есть много других операций для строк, таких как `indexOf`, `compareTo` и т. д.

Которые мы увидим позже.

Если вы хотите напечатать строку в Java, вы можете использовать оператор `System.out.print`.

И этот оператор принимает аргумент, который нужно напечатать.



Это может быть строка или другой тип.

`System.out.println`, в отличие от `System.out.print`, переводит печать на новую строку после печати.

Теперь надо отметить, что фактически, `String` не является примитивным типом данных как `boolean` или `int`.

Вот почему вы пишете `String` с заглавной буквы S.

Но мы поговорим об этом в позже.

Условия if и else



Tech Solutions

Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 5

Условный оператор if-else

Теперь поговорим об условии if.

Мы принимаем все время решения.

Если мы думаем, что пойдет дождь, мы берем зонт, прежде чем уйти из дома.

Но если мы думаем, что погода прояснится, мы оставим зонтик дома.

Мы видели, как мы можем составлять выражения последовательно, чтобы сделать программу.

Выражения выполнялись по порядку одно за другим.

Представьте себе, что мы хотим выполнить одну последовательность выражений, если выполнено какое-либо условие, и некоторую другую последовательность, если это условие не выполнено.

Давайте посмотрим пример.

Предположим, что мы хотим вычислить квадратный корень из числа.

```
...  
...  
if (n<0) n=-n;  
...  
...
```

И мы знаем, что число должно быть положительным, чтобы квадратный корень был реальным числом.

Поэтому, если нам дано отрицательное число, нам нужно сделать его положительным.

Если число положительное, нам не нужно ничего делать.

Как мы сделаем это на Java?

Ключевое слово if вводит условное выражение.

В этом примере выражение присваивания n равно минус n, выполняется только в том случае, если выполняется условие n меньше 0.

Если это условие ложно, ничего не делается.

Теперь, что, если мы хотим выполнить более одного выражения в зависимости от условия.

Мы просто помещаем выражения между фигурными скобками, делая их блоком.

```

...
n=-3; x=1;
if (n<0) {
    n=-n;
    x=0;
}
...

```

n	x
-3	1
3	1
3	0
3	0

Если условие ложно, ни одно из выражений этого блока не выполняется.

В общем, рекомендуется писать фигурные скобки, даже если при этом условии должно быть только одно выражение.

Логическое выражение для условия должно всегда находиться между круглыми скобками.


И следите, чтобы не поставить точку с запятой после логического выражения.

Выражение при этом условии – это пустое выражение, которое представлено точкой с запятой, и следующее выражение в фигурных скобках всегда будет выполняться независимо от значения логического выражения.

```

...
if (n<0) ; {
    n=-n;
}
...

```



Таким образом, условное выражение позволяет нам выполнить выражение или блок выражений, в зависимости от значения логического выражения.

Это одна из структур, контролирующая поток выполнения программы.

Иногда мы сталкиваемся с альтернативой на своем пути.

В зависимости от некоторых условий мы идем так или иначе.

Как мы это выразим в Java?

Сейчас мы знаем, как выполнить выражение в зависимости от одного условия.

Если условие не выполняется, ничего не делается.

Теперь мы хотим выполнить альтернативное выражение в этом случае.

Здесь мы видим простой пример.

```

...
if (n<0) {
    x=-n;
}
else {
    x=n;
}
...

```

n	x
-3	
	3
-3	3

n	x
3	
	3
3	3

x присваивается минус n, если n отрицательно.

Если это не так, x присваивается n .

Таким образом, существует два альтернативных блока выражений.

Тот, который выполняется, если условие истинно.

И тот, который выполняется, если условие ложно.

Этот блок записывается после ключевого слова `else`.

Конечно, в каждой из двух альтернатив, у нас может быть блок выражений вместо одного выражения.

Что теперь, если мы хотим разделить не только два случая, но и больше, например, три случая.

Поскольку условное утверждение является выражением, мы можем поместить его в любую из ветвей.

Например, давайте напишем условное выражение внутри другой ветви.

Новое условие проверяет, равно ли n 0.

```
if (n<0) {
    x=-n; s="n<0";
} else if (n==0) {
    x=0; s="n==0";
} else {
    x=n; s="n>0";
}
```

n<0

!(n<0) && (n==0)

!(n<0) && !(n==0)

Если это так, мы что-то делаем.

Иначе мы делаем что-то еще.

В целом, теперь у нас есть три случая, из которых только один выполняется.

Здесь показан пример с 4 случаями.

```
if (n<0) {  
    x=-n; s="n<0";  
} else if (n==0) {  
    x=0; s="n==0";  
} else if (n<5) {  
    x=n; s="0<n<5";  
} else {  
    x=n; s="n>=5";  
}
```

n<0

n==0

(0<n) && (n<5)

n>=5

Выражение switch



Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 6

Оператор switch

Для исследования проблемы else, давайте взглянем на эти два блока кода.
Единственное различие между двумя блоками является идентификация принадлежности else.

```
int a=10, b=5, c=10;
if (a>b)
    if (b>c)
        a = 20;
else
    a = 30;

int a=10, b=5, c=10;
if (a>b)
    if (b>c)
        a = 20;
else
    a = 30;
```

Question: The only difference between the two code blocks is the indentation on the else-clause.

- Which if-statement does the else-clause belong to?
- What would be the value of a?

И здесь могут быть два вопроса.

Первый, к какому выражению if выражение else принадлежит?

Второй вопрос, это то, каким будет значение после оценки if выражения?

Идентификация фактически не влияет на то, как компилятор будет интерпретировать блоки кодов.

В Java, else выражение соотносится с ближайшим возможным if выражением.

В этом случае, это проверка значения b.

Таким образом, здесь блок кода слева такой же, как код блока справа, с парой вставленных фигурных скобок.

```
int a=10, b=5, c=10;
if (a>b)
    if (b>c)
        a = 20;
else
    a = 30;

int a=10, b=5, c=10;
if (a>b) {
    if (b>c)
        a = 20;
    else
        a = 30;
}
```

the same as

- Indentation takes no effect.
- The else-clause is paired with the closest possible if-clause.
- Use braces to make your intention clear!
- a = 30 is the correct result.

Результат оценки блока кода приведет к установке значения $a = 30$ в конце выполнения. Мы можем также использовать комбинацию if-else if.

Пример здесь показывает, как эта комбинация может быть использована для определения уровня знаний в зависимости от оценки.

```
if (score >=90)
    grade = 'A';
else if (score >=80)
    grade = 'B';
else if (score >=70)
    grade = 'C';
else if (score >=50)
    grade = 'D';
else
    grade = 'F';
```

Обратите внимание, что это будет иметь большое значение, если ключевое слово else остается перед if.

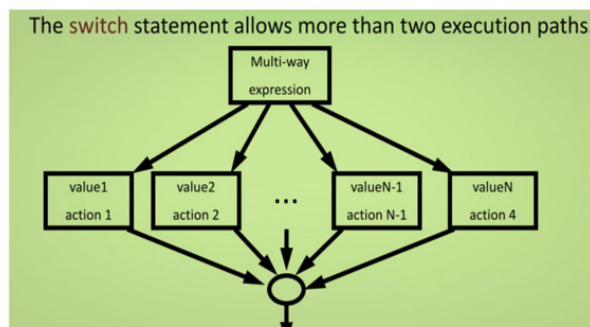
Сравните со случаем, когда else убрано.

```
if (score >=90)
    grade = 'A';
if (score >=80)
    grade = 'B';
if (score >=70)
    grade = 'C';
if (score >=50)
    grade = 'D';
else
    grade = 'F';
```

В этом случае поток выполнения прерываться не будет.

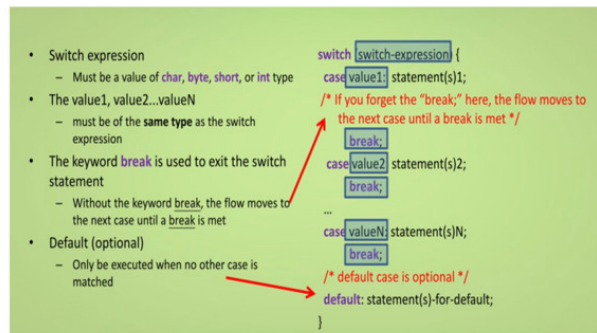
В то время как if выражение позволяет выбрать из двух возможных путей, switch выражение позволяет более двух путей выполнения.

Вот диаграмма для иллюстрации потока управления switch выражения.



И вот синтаксис switch выражения.

Синтаксис switch выражения начинается с ключевого слова switch.



Выражение switch может иметь тип char, byte, short или int, и String.

Значения case value1, value2 и т.д., должны быть того же типа, что и выражение switch.

Ключевое слово break используется для выполнения switch выражения.

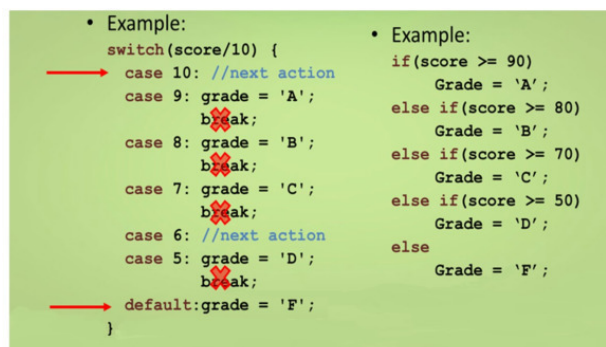
Важно помнить, что без break, поток будет продолжать двигаться к следующему case, пока break не будет найден.

Наконец, есть опция по умолчанию.

С ключевым словом default, эта часть кода будет выполняться только, когда никакие другие случаи не соответствуют.

Теперь посмотрим пример с использованием switch выражения.

Угадайте, что произойдет, если убрать все ключевые слова break?



Это будет то же самое, как если в примере if-else if убрать ключевое слово else.

На самом деле, все, что может быть сделано с помощью switch выражения, также может быть сделано с помощью if-else выражения.

Таким образом, в отличие от операторов if и else оператор switch может иметь несколько возможных путей выполнения.

И switch работает с примитивными типами данных char, byte, short или int и строками.

Решение о том, следует ли использовать операторы if и else или оператор switch, зависит от выражения, которое тестирует оператор.

Операторы if и else могут тестировать выражения на основе диапазонов значений или условий, тогда как оператор switch проверяет выражения, основанные только на одном перечисляемом значении.

Тернарный оператор



Объектно-ориентированное программирование на Java

Основные конструкции языка Java

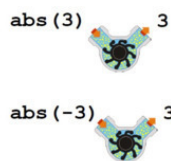
Лекция 7

Тернарный оператор

Представьте, что мы хотим вычислить абсолютное значение числа.

Это число без знака.

Предположим, что `abs`, является функцией, которая вычисляет абсолютное значение.



Таким образом, `abs 3` равна 3, а `abs -3` также равно 3.

Давайте определим проблему более формально.

Если условие `x` больше 0 вычисляется как `true`, тогда вычисление `abs x` совпадает с вычислением `x`.

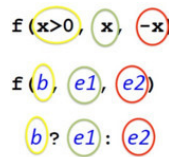


Если условие `x` больше 0 вычисляется как `false`, тогда вычисление `abs x` – это то же самое, что и вычисление значения минус `x`.

Теперь мы хотели бы написать выражение, которое вычисляет абсолютное значение.

Мы бы решили проблему, если бы у нас была функция `f` с тремя аргументами.

Первый аргумент – это условие.



Второй аргумент – это выражение для вычисления в случае true.

И третий аргумент – это выражение для вычисления в случае false.

В Java эта функция существует, называется она тернарный оператор, и имеет определенный синтаксис.

Здесь используется знак вопроса между условием и выражением для случая true и двоеточие между выражением для случая true и выражением для случая false.

В этом примере, если условие истинно, оператор выдает 1.

```
true ? 1 : 2 → 1
```

```
false ? 1 : 2 → 2
```

Если условие ложно, оператор выдает 2.

Основным типом данных в условных выражениях является тип `boolean`, который имеет два значения: `true` и `false`.

Но существуют ли в наших условных выражениях `if else` только два возможных случая?

Представьте, что вы плохо запрограммировали логическое выражение, тогда это приведет к вычислению, которое не может завершиться.

В этом случае, если вычисление логического выражения не завершается, вся программа не будет завершена.

```

if (non-terminating) {S1;}
    else {S2;}
↓
non-terminating

```

Поэтому, на самом деле, у нас есть три случая, это true, false и undefined.

В дальнейшем, анализируя сегменты кода, мы также должны учитывать это неопределенное значение.

Для логических выражений это означает, что у нас есть три возможных случая – true, false и undefined.

И это отличается от традиционной математики, где мы обычно имеем только истину и ложь.

Теперь, давайте немного вспомним о возможностях, которые мы видели.

Здесь, слева, у нас есть условное утверждение, где, в зависимости от значения булевой переменной b, мы присваиваем m или n переменной x.

<pre> if (b) {x=m;} else {x=n;} </pre>	<pre> x= (b) ? m : n ; </pre>
Conditional Statement	Conditional Expression

С другой стороны, у нас есть тройной оператор, который позволяет писать логические выражения.

Оба сегмента кода эквивалентны.

Теперь рассмотрим этот пример.

Представьте, что у нас есть булево значение b и что выражение сравнивает b с true.

```

if (b==true) {S1;} else {S2;}

if (b) {S1;} else {S2;}

```

Это может быть явно упрощено до `b`, так как если `b` истинно, `b == true`, вычисляется как `true`.

И если `b` является ложным, `b == true`, вычисляется как `false`.

И если `b` не определено, выражение `b == true` также не определено.

Так почему бы не написать более простую версию, просто `b` как условие?

Аналогично вы можете поступить, если мы имеем выражение `b == false`.

```
if (b==false) {S1;} else {S2;}

if (!b) {S1;} else {S2;}

if (b) {S2;} else {S1;}
```

Вы можете выбрать более простую версию, не `b`.

И еще вы можете написать `b` как условие, и поменять операторы `S1` и `S2`.

Здесь у нас есть другое выражение.

```
b ? true : false
```

Давайте проанализируем его.

Здесь, если `b` не определено, результат не определен.

Если `b` истинно, результат будет истинным.

И если `b` является ложным, результат будет ложным.

Мы рассмотрели все возможные значения `b` и всего выражения

И мы видим, что они имеют одинаковые значения, что они эквивалентны.

Поэтому вместо всего этого выражения мы можем написать только `b`.

Та же самая ситуация будет с выражением не `b`.

Теперь, давайте посмотрим выражение `b ? c : false`.

```
b ? false : true  
  
!b
```

Если `b` не определено, все выражение не определено.

```
b ? c : false  
  
b && c
```

Если `b` истинно, результат равен `c`.

Однако, если `b` является ложным, результат будет ложным.

Результат будет истина, только если `b` и `c` истина, во всех других случаях результат будет ложным.

Это эквивалентно логическому оператору `и`.

И наоборот, выражение `b ? true : c` эквивалентно логическому оператору `или`.

```
b ? true : c  
  
b || c
```

Циклы while и for



Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 8

Циклы while, for и do-while

Давайте представим, что мы хотим разделить целое число m на другое целое число n .

И мы хотим получить результат целочисленного деления, то есть самое большое количество раз, которое n вписывается в m .

$$m = (y * n) + x$$

$$m=7$$
$$n=2$$

Integer remainder

$$x = 7 \% 2 = 1$$

Integer division

$$y = 7 / 2 = 3$$

Например, целочисленное деление 7 на 2, равно 3, потому что 2 по 3 раза, это 6. Остаток равен 1.

И представьте себе, что у нас нет встроенной операции, которая выполняет эту операцию для нас.

Поэтому нам нужно сделать повторяемые вычитания.

И если нам удастся вычесть 2 из 7 три раза, это означает, что целочисленное деление равно 3.

Целочисленное деление y и целочисленный остаток x соответствуют формуле, m равно y умножить на n плюс x .

Предположим, что нам даны целые числа m и n .

А в x сохраняется оставшееся значение после вычитаний.

	x	y
int x=m; int y=0;	7	0
if (x>=n){x=x-n; y=y+1;}	5	1
if (x>=n){x=x-n; y=y+1;}	3	2
if (x>=n){x=x-n; y=y+1;}	1	3
if (x>=n){x=x-n; y=y+1;}	1	3
if (x>=n){x=x-n; y=y+1;}	1	3

Итак, давайте начнем с x равно m .

y содержит результат целочисленного деления.

Мы инициализируем y 0 и прирачиваем y на 1 каждый раз, когда мы вычитаем n из x .

И мы продолжаем вычитать n из x , пока x не меньше n .

Если x больше или равно n , мы вычитаем n из x и увеличим y на 1.

Таким образом, эта программа делает то, что мы хотим, но тут есть проблема.

Мы не знаем, сколько операторов `if` мы должны добавить.

Потому что это зависит от фактических значений m и n .

Например, с 7 и 2, это будет три выражения `if`.

При других входных данных это должно быть другое число `if` выражений.

В Java эту проблему решает оператор `while`.

Теперь эта программа делает то же самое, что и прежде, повторяет выражение, пока выполняется условие.

	x	y	(y*n)+x
int x=m; int y=0;	7	0	7
while (x>=n){	5	1	7
x=x-n; y=y+1;	3	2	7
}	1	3	7

Но теперь у нас есть одно большое преимущество.

Выражения повторяются столько раз, сколько это необходимо, автоматически.

Но теперь вы должны быть очень осторожны при написании условия `while`.

Потому что есть опасность войти в бесконечный цикл, если условие `while` никогда не прекратится.

Преимущество цикла `while` заключается в том, что нам не нужно заранее знать, сколько раз мы должны что-либо повторять.

```
while (Boolean Exp.) {
    Statements;
}
```

Мы повторяем, пока не будет достигнута цель, выраженная логическим условием.

Иногда, однако, мы знаем, сколько раз нам нужно что-либо повторить.

Это легко реализовать подсчетом.

Хитрость заключается в том, чтобы ввести счетчик.

Это целочисленная переменная, которую мы обновляем на каждой итерации.

```
int i=0;
while (i<4){
    i=i+1;
}
```

i	i<4
0	
0	true
1	true
2	true
3	true
4	false

Здесь существует три важных элемента: величина, с которой мы хотим начать, значение в конце и шаг между значениями.

Здесь мы начинаем с 0 и заканчиваем 3. И шаг 1.

Поэтому мы выполняем четыре итерации для *i* равного 0, 1, 2 и 3.

Теперь, помимо подсчета, мы можем захотеть что-то сделать в теле цикла.

В этом случае предположим, что у нас есть другая переменная, *n*, которую мы хотим умножать на 2 при каждой итерации.

```
int n=1;
int i=0;
while (i<4){
    n=n*2;
    i=i+1;
}
```

i	i<4	n
0		1
0	true	2
1	true	4
2	true	8
3	true	16
4	false	

Так как такого рода подсчет используется часто, в Java для этого есть специальная конструкция.

А именно, цикл `for`.

Этот цикл объединяет три важных момента для переменной счетчика:

```
int n=1;
int i=0;
while (i<4) {
    n=n*2;
    i=i+1;
}

int n=1;
for (int i=0;
    i<4;
    i=i+1) {
    n=n*2;
}
```

Ее инициализацию, условие остановки, и выражение обновления.

Имейте в виду, что обновление выполняется после того, как выполняется тело цикла, а не раньше.

Для цикла `for` скобки являются обязательными, а также две точки с запятой внутри.

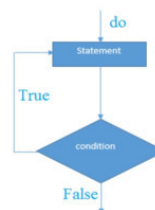
Фигурные скобки необходимы только в том случае, если есть более одного выражения в теле цикла.

Но это хорошая практика, чтобы всегда писать их.

Как вы видели, если в начальный момент условное выражение, управляющее циклом `while`, ложно, тело цикла вообще не будет выполняться.

```
do {
    // тело цикла
} while (условие);

int n = 5;
do {
    System.out.println("do-while " + n);
} while (--n>0);
```



Однако иногда желательно выполнить тело цикла хотя бы один раз, даже если в начальный момент условное выражение ложно.

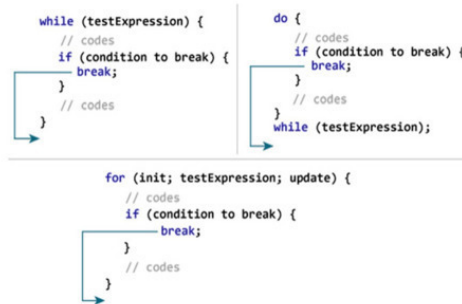
Иначе говоря, существуют ситуации, когда проверку условия прерывания цикла желательно выполнять в конце цикла, а не в его начале.

И в Java есть именно такой цикл: `do-while`.

Этот цикл всегда выполняет тело цикла хотя бы один раз, так как его условное выражение проверяется в конце цикла.

В приведенном примере тело цикла выполняется до первой проверки условия завершения.

Мы уже видели оператор `break` в выражении `switch`.



Но оператор `break` также может прерывать любой цикл.

Предположим, у вас есть цикл.

И иногда желательно немедленно завершить цикл, не проверяя условие.

В таких случаях используется оператор `break`.

Оператор `break` немедленно завершает цикл, и управление программой переходит к следующему выражению, следующему за циклом.

Оператор `break` почти всегда используется вместе с выражением `if else`.

Также иногда желательно не прервать цикл, а пропустить код тела цикла и перейти к следующей итерации.



Оператор `continue` пропускает текущую итерацию цикла и когда выполняется оператор `continue`, управление программой переходит к концу цикла.

Затем проверяется условие, которое управляет циклом.

Оператор `continue` также почти всегда используется вместе с выражением `if else`.

Массивы



Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 9

Массивы

На почте мы могли арендовать ячейку, чтобы получать письма.



Есть также много других мест, где мы можем арендовать ячейку: в банке, для хранения ценностей, на вокзале, чтобы оставить багаж.

Ячейки обычно обозначаются последовательными номерами.

Ячейки или шкафчики могут быть разного размера.

В программировании мы видели переменные, которые позволяют хранить значения.

Здесь размеры также могут отличаться в зависимости от того, хотим ли мы сохранить логическое значение или число с плавающей запятой.

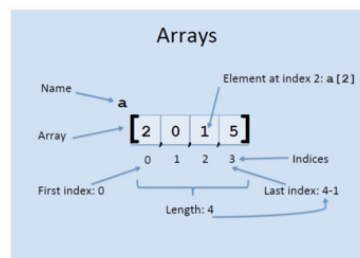
Однако в некоторых случаях нам может понадобиться упорядоченный набор значений одного и того же типа.

Например, когда мы хотим хранить оценки учеников класса, или температуры каждого дня месяца.

Точно так же, как мы могли бы арендовать ряд ячеек, нам может потребоваться зарезервировать набор или массив переменных одного и того же типа.

Как нам к этим переменным обращаться?

Мы привыкли свободно выбирать имена переменных.



И таким же образом мы можем дать имя массиву переменных. Для обозначения местоположения одной переменной используется индекс. Так, например, мы могли бы назвать массив `a`. Предположим, что у него четыре элемента в четырех позициях. Мы будем ссылаться на каждую позицию, добавляя индекс в квадратные скобки. Обратите внимание, что мы начинаем с индекса 0 и увеличиваем его на единицу. Здесь мы видим примеры массивов.

Declaration

```
byte[] arrayOfBytes;
short[] arrayOfShorts;
int[] arrayOfInts;
long[] arrayOfLongs;
float[] arrayOfFloats;
double[] arrayOfDoubles;
boolean[] arrayOfBooleans;
char[] arrayOfChars;
String[] arrayOfStrings;
```

Массивы могут содержать разные типы значений, но в каждом массиве, тип является одинаковым для всех значений.

Массив может иметь любую длину, но после определения длины при создании массива его длина остается фиксированной.

Надо помнить, что есть два шага при работе с массивами. Объявление массива и его создание.

Creation

```
a = new int[4];
```

Declaration and Creation

```
int[] a = new int[4];
```

Элементы в массиве можно получить с помощью индекса.

Мы не должны путать значение элемента с его индексом.

Еще одна вещь, которую следует помнить, это то, что первым элементом массива является элемент с индексом 0.

Таким образом, индексы начинаются с 0 и до длины массива минус 1.

Мы объявляем массив, указывая тип элементов, затем открываем и закрываем квадратные скобки, и затем указываем имя, которое мы выбрали для нашего массива.

После объявления, создавая массив с помощью ключевого слова `new`, мы физически резервируем для него место в памяти, как в почтовом отделении.

Мы также можем сделать это вместе: объявить и создать массив в одной строке.

Теперь мы можем хранить значения в разных позициях.

Как мы сохраняем значения?

Мы используем оператор присваивания, как раньше мы использовали его для переменных.

Initialization

- Assignment one by one


```
a[0] = 2;
a[1] = 0;
a[2] = 1;
a[3] = 2*a[0]+a[2];
```
- Declaration, creation and initialization


```
int[] a = {2, 0, 1, 5};
```

Имя массива с индексом используется, как мы раньше использовали идентификаторы переменных.

Мы также можем объявить, создать и инициализировать массив сразу, как мы видим здесь, в последней строке, используя фигурные скобки.

Обратите внимание, что в этом случае нам не нужно писать ключевое слово «`new`».

Теперь, если строки – это упорядоченные последовательности символов, вопрос, является ли строка и массив символов одним и тем же.

Это не так, хотя можно конвертировать одно в другое.

```
['2', '0', '1', '5']

"2015"
```

Другой вопрос заключается в том, может ли элемент массива быть массивом.

Здесь ответ да.

Таким образом, мы получаем то, что мы называем двумерными массивами.

2-Dimensional Arrays

- `[[1], [3], [5,6]]`
- `[[1,2], [3,4], [5,6]]`
- `[[1,2],
[3,4],
[5,6]]`

Но возможны и многомерные массивы.

Таким образом, массивы – это упорядоченные последовательности элементов одного и того же типа.

И длина фиксируется при создании массива.

И элементы массива могут быть массивами.

Массивы и циклы `for` имеют нечто общее.

Массив состоит из последовательности данных, а цикл `for` выполняет выражения последовательно несколько раз подряд.

Здесь мы видим массив с четырьмя целыми числами от 0 до 3.

```
[0, 1, 2, 3]
```

```
for (int i=0;  
    i<4;  
    i=i+1)  
{...}
```

И ниже приведена структура цикла `for`, которая повторяет выражения четыре раза.

Теперь, если мы хотим сделать одно и то же преобразование для всех значений в массиве, цикл `for` является хорошим для этого способом.

Например, если применить операцию возведения в степень 2 к целому числу 3, получим 9.

```
y = square(x);  
  
for (int i=0; i<4; i=i+1){  
    y[i] = square(x[i]);  
}
```

Теперь представьте, что мы хотим применить эту операцию ко всем целым числам в массиве.

Цикл `for` поможет нам последовательно брать все значения в массиве и возводить их в степень 2, начиная с индекса 0 до индекса 3.

Другой пример – сложить все числа в массиве.

`[0, 1, 2, 3] → 6`

```
int z = 0;
for (int i=0;
    i<4;
    i=i+1){
    z += x[i];
}
```

Если вы хотите сделать это для любой длины массива, используйте `x.length` вместо 4.

`[0, 1, 2, 3] → 6`

```
int z = 0;
for (int i=0;
    i<x.length;
    i=i+1){
    z += x[i];
}
```

Перебор элементов массива в цикле `for`, начиная с индекса 0 до длины массива, настолько распространен, что для этого существует специальный цикл `for`.

`[0, 1, 2, 3] → 6`

```
int z = 0;
for (int elem: x){
    z += elem;
}
```

В этом цикле `for` мы можем проинструктировать переменную `elem` последовательно использовать все элементы массива.

Представление данных и типы данных



Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 10

Представление данных и типы данных

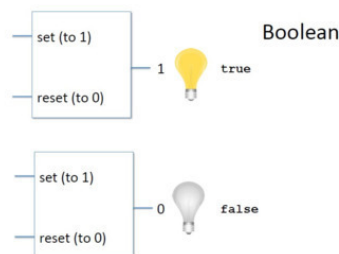
Давайте посмотрим под капот калькулятора или компьютера, и посмотрим, как мы можем представлять данные.

И начнем с простого.

Давайте посмотрим на логические значения, потому что там есть только два значения, true и false.

Цифровые компьютеры состоят из электроники, которая может находиться только в одном из двух состояний.

Триггер – это базовая единица, которая может оставаться либо в одном положении, либо в другом.



Выход триггера остается в одном из этих двух состояний и будет оставаться там до тех пор, пока не появится сигнал для его изменения.

В действительности 1 может иметь нулевое напряжение, а другое состояние – пять вольт. Но мы можем произвольно интерпретировать их как 0 и 1.

Поэтому мы можем сказать, что триггер может хранить один бит информации.

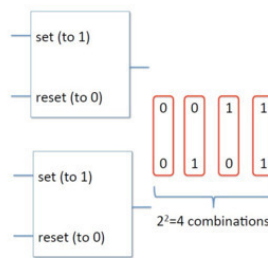
Теперь это именно то, что нам нужно, чтобы сохранить логическое значение, потому что логических значений также два, ложь и истина.

И мы, опять же, можем произвольно присвоить 0 false и 1 true.

Итак, мы говорим, что нам нужен бит, чтобы сохранить логическое значение.

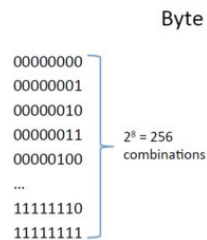
Теперь, если у вас есть два триггера, мы можем сохранить два бита.

Если мы соберем их вместе, у нас будет четыре возможных комбинации: 0—0, 0—1, 1—0 и 1—1, поскольку каждый из них может иметь состояние 0 или 1 независимо друг от друга.



И если мы возьмем восемь триггеров, чтобы сохранить восемь бит, у нас будет 2 в степени 8 различных комбинаций.

То есть 256 комбинаций в целом.



Что мы можем с ними делать?

Восемь бит называется байт.

Итак, что мы можем сделать с байтом?

Мы можем представить 256 различных чисел.

Например, натуральные числа от 0 до 255.

Мы также можем отображать 256 уровней красного, от черного до ярко-красного.

И мы можем получить любой цвет, составляя уровни красного, зеленого и синего.

Для каждого из этих компонентов мы используем один байт.

Таким образом, это всего три байта или 24 бита, что означает 2 в степени 24 , что почти 17 миллионов цветовых комбинаций.

Звуки, фильмы, все представлено битами 0 и 1.

Это позволяет нам иметь богатую информацию, но в тоже время иметь единый способ обработки этой информации.

Наконец, мы можем также представлять отдельные символы, как те, которые есть у вас на клавиатуре, а также некоторые другие специальные символы.

Для этого существует множество кодировок.

Java использует кодировку юникода, использующую 16 бит.

Другие кодировки используют только восемь бит.

Таким образом, все в компьютере представлено битами.

Все сводится к нулям и единицам.

Давайте сосредоточимся на том, как мы представляем числа в двоичной форме битами.

С 1 байтом – 8 бит – мы можем сформировать 256 различных комбинаций, или 2 в степени 8 .

1 Byte = 256 Numbers

00000000	0
00000001	1
...	...
01111111	127
10000000	128
...	...
11111110	254
11111111	255

$2^8 = 256$ combinations

Поэтому мы можем представить 256 различных чисел.

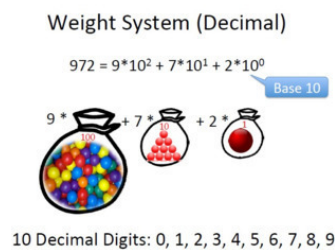
Это могут быть, например, натуральные числа от 0 до 255.

Но какая комбинация байт соответствует какому числу?

Давайте проанализируем систему, которую мы используем для представления чисел в нашей десятичной системе, которая использует 10 цифр, от 0 до 9.

Используем систему, основанную на весах.

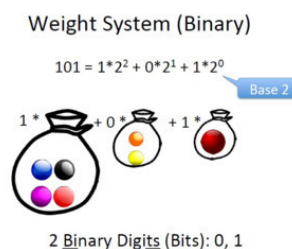
Чем больше мы двигаемся влево, тем выше вес.



Когда мы пишем 972, мы имеем в виду 9 умножить на 100 плюс 7 умножить на 10 плюс 2.

Так как здесь основание 10, система исчисления называется десятичной.

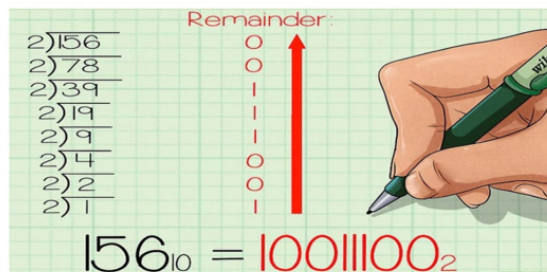
Для двоичной системы исчисления тот же принцип, только основанием будет 2.



Соответственно, перевести число из двоичной системы в десятичную очень просто, нужно сложить получившийся ряд.

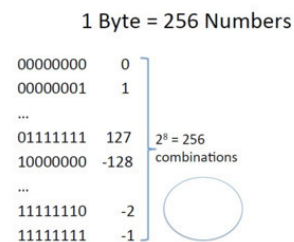
$$\begin{aligned}
 &10110110 = \\
 &\quad \text{Base 2} \\
 &1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = \\
 &128 + \quad \quad 32 + \quad 16 + \quad \quad 4 + \quad 2 = \\
 &182 \\
 &\quad \text{Base 10}
 \end{aligned}$$

Перевести число из десятичной системы в двоичную тоже просто, нужно делить на 2 и записывать остаток.



Но как насчет отрицательных чисел? Нам тоже нужно работать с ними.

Неотрицательные числа, т. е. 0 и положительные числа – закодированы по-прежнему, где самый левый бит установлен в 0.



И у нас осталось семь бит.

Таким образом, мы можем иметь 2 в степени 7 различных неотрицательных чисел, а именно от 0 до 127 .

Для отрицательных чисел они кодируются таким образом, что сумма отрицательного числа и его положительного аналога равна 2 в степени числа бит, т. е. восемь, или 256 , или 1 , а затем восемь 0 .

Таким образом, с этим кодированием мы можем представлять, как положительные, так и отрицательные числа.

Теперь давайте сосредоточимся на Java.

Какие типы данных мы используем для целых чисел?

На самом деле это не один тип данных, а доступно несколько типов данных.

```
Java Data Types for Integers

byte
8 bits: -128..127
short
16 bits: -32,768..32,767
int
32 bits: -231..231-1
long
64 bits: -263..263-1
```

У нас есть тип данных, называемый «байт», который использует точно восемь бит – это и есть, один байт.

Мы можем представить цифры от -128 до 127, как мы только что видели.

Есть тип данных «short», который использует 16 бит и находится в диапазоне от -32 000 до плюс 32000.

Но основным типом данных, которым мы будем пользоваться, будет «int».

Здесь максимальное положительное число составляет более 2 миллиардов.

Если вам потребуются большие цифры, можно использовать «long» с 64 битами.

Для чисел с плавающей запятой есть два типа данных в Java: «float», который использует 32 бита, и «double», который использует 64 бита.

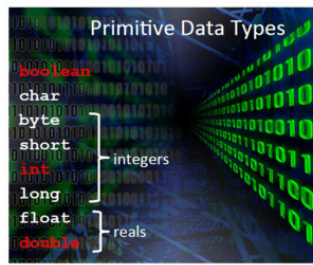
```
Java Data Types for Reals

float
single-precision 32-bit
IEEE 754 floating point
double
double-precision 64-bit
IEEE 754 floating point
```

Рекомендуется использовать double, когда нужны числа с плавающей запятой.

Подводя итог, существует восемь примитивных типов данных в Java.

Два для представления нечисловых данных: boolean для булевых значений, true и false, и char – для представления одного символа.



И числовые типы данных.

int – это основной тип данных, который нужно запомнить для представления целых чисел.

И остальные байт, short, и long.

И double – это основной тип данных для чисел с плавающей запятой.

Другой тип – float.

Таким образом мы не можем работать с бесконечно большими числами или числами с бесконечной точностью.

Методы



Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 11

Методы

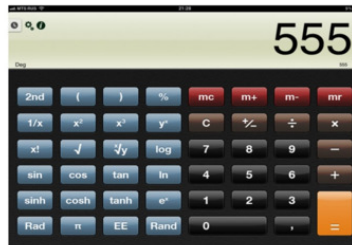
Представьте себе, что вам приходится многократно вычислять квадрат числа.

Для каждого числа, вы вводите одно число, затем оператор умножения, а затем снова число.

И то же самое для другого числа и т. д.

Поэтому в калькуляторе было бы неплохо иметь программируемую кнопку, которая выполняет любую операцию, которую мы определим.

Это может быть квадрат или квадратный корень, или любые вычисления, которые нам понадобятся.



В Java также возможно определять пользовательские операции.

Но вместо того, чтобы называть их операциями, мы называем их методами.

Это является терминологией Java.

В других языках программирования они называются функциями или процедурами.

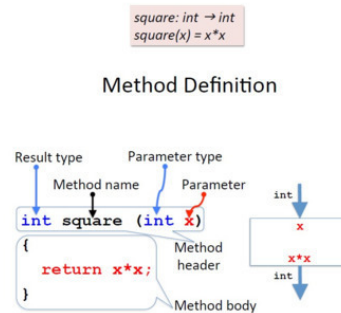
Метод – это вычисление, которому мы даем имя, так что мы можем вызывать его в любое время, когда нам нужно выполнить это вычисление.

Метод может зависеть от одного или любого числа параметров.

И метод может привести к какому-то результату или какому-то эффекту.

Рассмотрим метод вычисления квадрата числа.

Можно представить этот метод как черный ящик, который получает целое число в качестве входных данных и выводит другое целое число.



В математических терминах мы можем определить его как функцию следующим образом.

Мы дадим функции имя, например, `square`.

И эта функция принимает целое число как параметр и возвращает целое число.

Функция определяется следующим образом.

Если мы назовем аргумент или параметр как `x`, результат получается умножением `x` на `x`.

Теперь, как мы определим это в Java?

Сначала мы напишем что-то похожее на первую строку в математическом определении.

Но порядок немного другой.

Во-первых, мы пишем тип результата, затем имя метода, а затем в круглых скобках тип параметра и далее идентификатор параметра.

При этом у нас может быть несколько параметров.

Все это называется заголовком метода.

Затем мы напишем в фигурных скобках то, что мы должны сделать, чтобы вычислить результат.

И мы указываем, что это результат возврата, поместив ключевое слово `return` перед выражением.

Затем в фигурных скобках мы пишем вычисление, которое хотим выполнить.

И мы называем это телом метода.

Имя метода может быть любым допустимым идентификатором.

Но мы будем следовать соглашению, и напишем его с маленькой буквы.

И обычно это глагол.

Если нам нужно больше одного слова, мы будем писать каждое следующее слово с заглавной буквы.

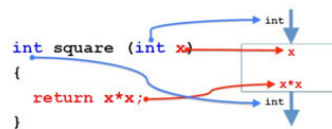
isEmpty

Как мы видим здесь в `isEmpty`.

И рекомендуется, чтобы имя метода имело значение, чтобы другие могли легко понять, что здесь вычисляется.

Имена параметров мы также можем свободно выбирать.

Нам нужно дать имя параметру, потому что нам нужно обращаться к параметру в теле метода.



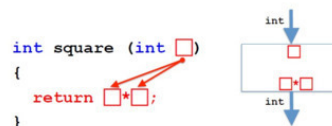
Но этот идентификатор является внутренним.

Если мы заменим его на другой идентификатор, мы не изменим метод.

Вместо `x` мы можем указать `y` в качестве идентификатора параметра.

Так как, по существу, `x` или `y` являются просто заполнителями для фактического параметра, который мы указываем при вызове метода.

Сколько входных параметров может иметь метод?

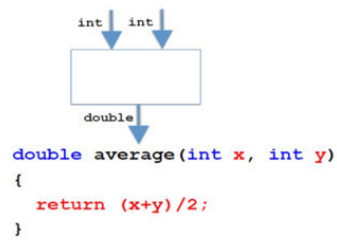


И что насчет результата?

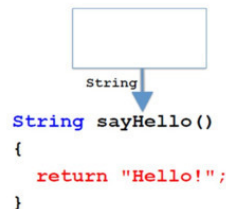
Мы видели, как определить метод с одним параметром и одним результатом.

Можем ли мы также иметь больше параметров?

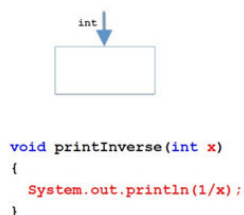
У нас может быть несколько параметров.



Здесь мы видим метод с двумя параметрами.
Обратите внимание, что они разделяются запятыми.
И у нас может быть еще больше параметров, разделенных запятыми.
Также у нас может не быть никаких параметров.



Теперь круглые скобки пустые.
В этом случае этот метод всегда возвращает одно и то же значение.
Или у нас может не быть никакого возвращаемого результата.



В этом случае мы пишем void как тип результата.
Это имеет смысл, например, если мы хотим что-то напечатать.
В других языках программирования говорят о процедурах, если нет возвращаемого значения.
И о функциях, если возвращается результат.
Но в Java мы просто говорим о методах.
Наконец, мы можем иметь метод без параметров и без результатов.



```
void printHello()  
{  
    System.out.println("Hello!");  
}
```

Теперь мы рассмотрели все возможные случаи.

Область видимости переменных



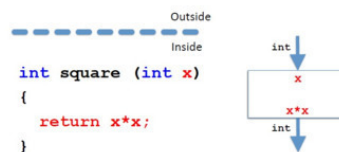
Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 12

Область видимости переменных

В предыдущей лекции мы узнали, как определить метод.
И мы хотим знать, что происходит, когда мы его вызываем.
Возьмем снова метод, вычисляющий квадрат числа.



Он называется square и принимает одно значение и возвращает другое значение – квадрат числа.

Важно отметить, что определение метода идентифицирует два контекста – внутри и снаружи.

Внутри мы можем использовать параметры *x* или *y* или что угодно.

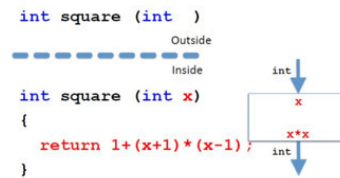
Но не снаружи.

Извне мы просто знаем название метода, параметры, и тип результата.

Как вычисляется метод, это вопрос внутреннего контекста.

В какой-то момент мы могли бы изменить тело метода.

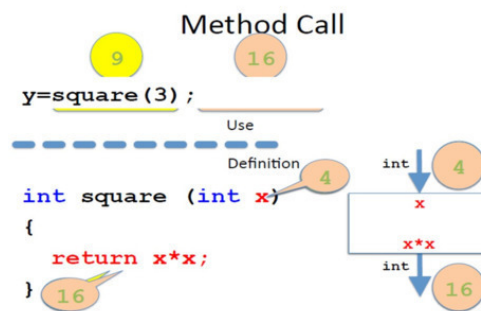
Здесь мы видим альтернативный способ вычисления квадрата числа.



Но мы не знали бы этого извне, из контекста вызова.

Теперь давайте посмотрим, что происходит, когда мы вызываем метод с заданным значением.

Мы могли бы проанализировать, что происходит, когда мы вызываем square (3).



Но давайте сделаем немного интереснее.

Попробуем оценить выражение square (3) + square (4).

Чтобы получить результат суммы, сначала мы должны вычислить первый операнд, square (3).

И для этого мы перейдем к определению метода, где x теперь равно 3.

Это означает, что мы должны заменить все x на 3.

Таким образом, мы вычисляем 3 умножить на 3.

Результат будет – 9, и это то, что возвращает вызов метода.

9 теперь является значением первого операнда суммы.

Затем нам нужно вычислить значение для square (4).

Перейдем к определению метода, но теперь x равно 4.

3 больше не существует.

Поэтому мы заменяем все x на 4, и поэтому умножаем 4 на 4.

Этот вызов метода возвращает 16 вызывающему выражению.

Теперь у нас есть оба операнда, и мы можем сложить 9 и 16.

Во всех этих вычислениях важно отметить, что два вызова одного и того же метода полностью независимы.

Мы использовали x с двумя независимыми значениями.

Сначала 3, а затем 4.

И когда мы использовали 4, 3 уже не существовало.

Каждый раз, когда мы делаем новый вызов, параметры создаются со значениями вызова.

Значения, которые мы имели от предыдущих вызовов, просто забываются.

Мы использовали идентификаторы или имена в разных целях: для переменных, для методов, для параметров метода и т. д.

Теперь возникает вопрос: если у нас есть переменная с именем «x», а затем у нас есть метод с параметром.

Можно ли назвать этот параметр как «x»?

Или будет какая-то несовместимость?

Можем ли мы использовать одно и то же имя в разных контекстах?

Давайте рассмотрим пример.

Представьте, что у нас есть программа, где есть целочисленная переменная с именем x,

```

int x=1;
int f(int x)
{
    return x+x;
}
int z=f(x+1);

```

Которую мы инициализируем в значение 1.

И у нас также есть метод «f», который имеет целочисленный параметр.

И мы просто решили назвать его «x».

Вопрос, можем ли мы это сделать?

И если да, то что этот метод вернет в качестве результата?

Ответ на этот вопрос при написании кода на Java – да, мы можем это сделать.

Каким образом, мы управляем двумя x?

Каждый x действителен в определенном контексте, при выполнении определенного сегмента кода.

У нас есть черный x, который действителен, и который существует, и для которого мы сохраняем пространство в памяти, когда объявляем переменную.

Мы также зарезервировали пространство в памяти для z.

И когда мы вызываем f с x плюс 1, значение x равно 1.

1 плюс 1 равно 2, и мы вызываем f с 2.

Далее мы переходим к определению метода.

Вызываем f с 2.

Таким образом, красный x равен 2.

Итак, мы выполняем x плюс x со значением 2.

2 плюс 2 равно 4.

И это то, что этот метод возвращает и что хранится в z.

Теперь помните, что параметр x метода f является просто заполнителем.

Поэтому, если f вызывается с переменной x, а значение x равно 2, f с x возвращает 4.

И с этим нет никаких проблем.

Мы говорим, что первое x является глобальной переменной, тогда как параметр x является локальным для метода.

В этом примере мы видим, что эта локальная переменная – этот параметр – создается дважды: во-первых, для внутреннего вызова f с x плюс 1, со значением 2, – и второй раз для внешнего вызова со значением 4.

```

int x=1;
int f(int x)
{
    return x+x;
}
int z=f(f(x+1));

```

После выхода из каждого определения метода, созданная переменная будет уничтожена. И выделенное пространство в памяти компьютера будет освобождено.

Поэтому, если вы хотите использовать внешнюю переменную в теле метода, вы должны выбрать другое имя.

Здесь мы видим, как использовать глобальную переменную `x` и параметр `y` в теле метода.

```

int x=1;
int f(int y)
{
    return x+y;
}
int z=f(x+1);

```

В этом случае у нас есть переменная `x`, которая видна во всем теле метода и за его пределами, и у нас есть переменная `y`, которая существует и видна только в теле метода.

Вне этого метода `y` не существует.

В этом примере, все, что мы только что сказали для параметров метода, применяется к переменным, объявленным внутри тела метода.

```

int x=1;
int f()
{
    int y=2;
    return x+y;
}
int z=f();

```

Здесь переменная `y` является локальной переменной в методе `f`.

В этом методе мы используем глобальную переменную `x` и локальную переменную `y`. Этот пример аналогичен предыдущему.

```

int x=1;
int f()
{
    int x=2;
    return x+x;
}
int z=f();

```

Но в этом случае мы решили назвать локальную переменную внутри метода `x`, так же, как и глобальную переменную.

Таким образом, в этом случае у нас нет доступа к глобальной переменной.

Когда мы вызываем `f` для вычисления `z`, мы вызываем `f`, где внутри определяется `x` со значением 2.

Таким образом, мы возвращаем 2 плюс 2, равно 4.

Метод `f` всегда возвращает 4.

И это то, что мы сохраним в переменной `z`.

Таким образом, мы видели, что у нас есть глобальные и локальные переменные.

Глобальные переменные существуют, начиная с объявления и для остальной части программы.

Но они могут временно затеняться другими локальными переменными с тем же именем.

В этом примере показан цикл.

```

int x=1;
int y=0;
for (int x=1; x<3; x++)
{
    y=y+x;
}
int z=y;

```

Для циклов также объявляются локальные переменные.

Здесь переменная `x` цикла `for` не позволяет нам видеть глобальную переменную при выполнении цикла.

Здесь у нас есть глобальная переменная `x` и глобальная переменная `y`.

Они инициализируются 1 и 0 соответственно.

Затем у нас есть глобальная переменная `z`, которая сохраняет значение `y`, но после выполнения этого цикла `for`.

Этот цикл `for` выполняется дважды.

Один раз для `x` равного 1 и один раз для `x` равного 2.

В каждом цикле `for`, `y` накапливает значение `x`.

Таким образом, при первом запуске `y` получает значение 1, а во втором `y` получает значение 1 плюс 2, равно 3.

Когда мы выходим из цикла `for`, локальная переменная `x` исчезает, остается только глобальная.

`y` имеет значение 3, и это значение, которое мы сохраняем в `z`.

Таким образом, мы видим точно такое же поведение для этих переменных в цикле `for`, как мы видели с локальными переменными в методах и с параметрами в методах.

В этом примере у нас есть глобальная переменная `x`.

```
int x=1;
int f(int x){
    int y=0;
    for (int x=1;x<3;x++){
        {y=y+x;}
    }
    return y+x;
}
int z=f(x+2)+x;
```



И у нас есть метод с параметром `x`.

И внутри этого метода у нас есть цикл `for` с другой переменной `x`.

Таким образом, в этом случае у нас есть 3 переменных `x`.

Поэтому, когда мы вызываем `f` с `x` плюс 2, в последней строке, где `x` равно 1, мы вызываем `f` с 3, чтобы вычислить `z`.

В методе, параметр `x` равен 3.

Внутри метода мы объявляем переменную `y`, инициализированную 0, и затем мы определяем цикл `for`.

Этот цикл `for` выполняется два раза, как в предыдущем примере.

Здесь, мы объявляем другую переменную `x`, которая делает невидимыми предыдущие две переменные `x`, пока мы не выполним цикл `for`.

Здесь мы увеличиваем значение `y`.

`y` в конце получает 3 и возвращает `y` плюс `x`.

Но что это за `x`?

Это не та переменная `x` в цикле `for`, потому что мы вышли из цикла `for`.


Эта `x` равна 3 и это параметр метода.

Поэтому возвращается 3 плюс 3.

Это то, что мы возвращаем `z`, и что добавляется к `x`, но в этом случае это глобальная переменная `x`, поэтому мы получаем 7 и присваиваем 7 в `z`.

Этот пример легко проанализировать.

```
int x=1;
int f()
{
    return x;
}
int z=f();
```



Метод `f` определяется в контексте, где `x` равно 1.

Таким образом, этот метод всегда возвращает 1 независимо откуда он был вызван. `x` равно 1 и `z` также присваивается 1.

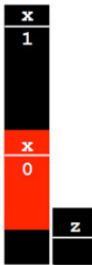
Важно отметить, что `f` получает свое определение в том месте, где он определен.

Если он определен в том месте, где `x` равно 1, метод `f` определяется, чтобы вернуть 1.

И это видно в этом примере.

В этом примере у нас есть два метода: `f` и `g`.

```
int x=1;
int f()
{
    return x;
}
int g()
{
    int x=0;
    return f();
}
int z=g();
```



`g` вызывает `f`, и он вызывает его в контексте, где `x` равно 0.

И здесь нужно учитывать, что метод `f` был определен в контексте, где `x` равно 1.

И мы уже сказали, что метод `f` всегда возвращает 1 независимо от того, где он вызывается.

Так как здесь `x` равно 1.


Это называется лексической областью действия или статической областью действия в отличие от динамической области действия.

Большинство языков программирования имеют статическую область действия, в том числе и Java.

Поэтому, как только метод определен, его значение и его поведение, зафиксированы.

Теперь, если мы удалим самое верхнее объявление `x`, переменная `x` не определяется при объявлении `f`.

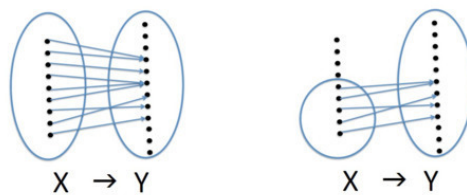
```
int f()
{
    return x;
}
int g()
{
    int x=0;
    return f();
}
int z=g();
```



Следовательно, этот сегмент кода выдаст ошибку во время компиляции.

Далее мы проанализируем взаимосвязь между частично определенными функциями в математике, и методами в Java, которые не определены для некоторых входных значений.

В математике мы изучаем функции, т. е. отображения между множествами значений, где значения области определения X сопоставляются значениям множества Y .



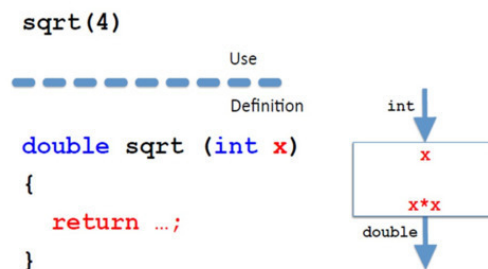
Обычно для всех значений из множества X существуют значения во множестве Y .

Однако может быть случай, когда для некоторых значений X нет отображения, определенного в Y .

В этом случае мы говорим о частично определенной функции.

Если вы хотите избежать частично определенных функций и всегда работать с полными функциями, вы можете выделить в X меньшее множество, где все значения имеют отображение.

Теперь вернемся к Java.



Предположим, мы хотим вычислить квадратный корень из 4.

Здесь есть два результата, плюс 2 и минус 2.

Предположим, что наш метод просто возвращает положительное значение, плюс 2.

Мы всегда можем получить другое решение, добавив знак минус.

Теперь, что произойдет, если мы вызовем метод `square` с аргументом минус 4?

Мы знаем, что решением в этом случае являются не действительные числа, а мнимые числа.

Таким образом, не существует реального числа, которое может быть предложено в качестве результата метода.

Метод не определен для отрицательных чисел.

В математике мы можем определить функции более подробно.

Мы можем настроить область определения в соответствии с тем, что нам нужно.

Например, мы могли бы сказать, что область определения этой функции не множество целых чисел, а множество натуральных чисел, то есть 0 и положительные целые числа.

Таким образом, функция будет определена для всех значений в этой области определения натуральных чисел.

Но в программировании мы имеем дело с существующими типами.

Теперь, как мы определяем в Java частично определенные функции или частично определенные методы?

Что мы можем сделать в случае метода, который не определен для всех возможных входных значений.

Во-первых, так как возникает ошибка при вызове метода `square` с отрицательным числом, мы будем ожидать ошибку, и программа должна завершиться с ошибкой.

Это, конечно, не самый удобный способ для решения этой проблемы.

Во-вторых, мы можем проверять значения параметров метода в самом методе или при вызове метода.

Или мы можем перехватить и обработать возникшую ошибку в самом методе или после его вызова, и об этом мы поговорим, когда будем обсуждать исключения Java.

Комментарии. Javadoc



Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 13

Комментарии и Javadoc

В прошлой лекции мы говорили о том, что мы можем сделать в случае метода, который не определен для всех возможных входных значений.

Программы могут содержать сотни тысяч строк кода.

И очень сложно отслеживать все возможности.

Поэтому мы также можем использовать языковые конструкции, чтобы избежать ошибки, как для себя, так и для других программистов, которые могут использовать ваш код.

Для этого можно использовать комментарии.

Комментарии представляют собой текст, чередующийся с кодом, и этот текст не должен выполняться компьютером, а должен читаться людьми.

Comments

```
double sqrt (int x){  
    /* x has to be  
       non-negative */  
    return ...;  
}
```

Еще одна возможность – это изготовить сопроводительную документацию к программе.

Javadoc – это инструмент, который является генератором документации на основе специальных комментариев.

Javadoc

```
double sqrt (int x){  
    /**  
     * @param x an int value  
     *   Precondition: x>=0  
     * @return the non-neg. sq. root  
     */  
    return ...;  
}
```

Если вы используете эти специальные комментарии, вы можете автоматически создать хорошую документацию.

Таким образом, комментарий представляет собой текст в программе, бесполезный с точки зрения компьютера, но который может быть полезен для программиста.

Здесь мы видим один из возможных способов написания комментария.

Comments

```
double sqrt (int x){
    /* x has to be
       non-negative */
    return ...;
}
```

Комментарий начинается с косой черты и звездочки и заканчивается звездочкой и косой чертой.

Комментарий может включать в себя несколько строк.

Здесь у нас есть еще один комментарий.

Javadoc

```
double sqrt (int x){
    /**
     * @param x an int value
     *   Precondition: x>=0
     * @return the non-neg. sq. root
     */
    return ...;
}
```

Это комментарий, так как он начинается с косой черты и звездочкой и заканчивается несколькими строками позже звездочкой и косой чертой.

Но на разных строках есть еще несколько звездочек.

И это указание для специальной программы под названием Javadoc.

Javadoc принимает в качестве входа Java-код с этими специальными комментариями и выдает документацию для ее использования программистами.

Специальные команды, такие как @param и @return, имеют смысл, который Javadoc понимает при подготовке итоговой документации.

Операционная система компьютера, веб-браузер, приложения мобильного телефона, все они – состоят из очень сложных частей программного обеспечения.

Например, смартфон с операционной системой Android имеет более 12 миллионов строк кода.

Из них более 2 миллионов написано на языке Java.

Представьте себе, что вы кодируете все эти строки самостоятельно.

Вам понадобится много времени.

Как программистам, нам нужно работать с другими программистами для достижения цели.

Нам также необходимо расширять или изменять предыдущие программы, написанные другими людьми, которых мы не знаем.

Также и другие программисты вполне вероятно будут работать с нашим кодом.

Попытка понять все строки кода, которые нам нужно использовать, требует огромных усилий.

Поэтому очень полезно писать заметки в наших программах, чтобы помочь другим и нам самим понять код, используя человеческий язык в этих заметках, но при этом не подвергая опасности выполнение нашей программы.

Эти примечания в программе – это то, что мы называем комментариями.

Это дополнительный текст, который мы добавляем в наш код, чтобы улучшить его читаемость и повторное использование.

Эти комментарии прозрачны для компьютера, поскольку они служат только для людей, но не имеют вычислительного смысла.

Как и во всем, что есть в жизни, существуют разные подходы в том, как мы можем писать эти комментарии.

Комментарии полезны для разных целей.

Например, описание кода, то есть резюмирование целей сегмента кода.

Описание алгоритма, который вы создаете.

Комментирование сегмента кода, который не работает должным образом.

Или автоматическое создание документации.

В Java существуют разные способы написания комментариев.

Сначала, мы сосредоточимся на тех типах комментариев, которые направлены на предоставление сведений о вашем коде вам и другим программистам.

Если для нашего комментария нужна только одна строка, мы будем писать две косые черты перед текстом комментария.

```
// This is a single line comment in Java
```

```
/* This is a multiple  
line comment in Java */
```

И комментарий будет идти до конца строки.

Если мы хотим включить комментарий из нескольких строк, мы будем писать косую черту, за которой следует звездочка.

И мы закончим комментарий звездочкой, а затем косой чертой.

При этом начало и конец комментария могут быть в одной строке или в разных строках.

Будьте осторожны и избегайте вложения друг в друга этих типов комментариев.

Существуют рекомендации по написанию кода на языке Java.

Советуют использовать комментарии с несколькими строками только при комментировании блока кода.

И использовать однострочные комментарии для всего остального.

Вы можете задаться вопросом, сколько комментариев вы можете вставить в свой код. Для этого нет однозначного ответа.

Убедитесь, что ваши комментарии соответствуют вашему коду.

Не забывайте обновлять свои комментарии при изменении кода.

Хороший программист создает не только хороший код, но также предоставляет другим возможность использовать свой код.

То есть, дает хорошие комментарии.

Есть еще один полезный и почти обязательный тип комментариев, который предназначен для создания подробной документации о нашем коде.

Существует программа под названием Javadoc, которая генерирует документацию из кода Java в HTML-файлы, чтобы мы могли легко их прочитать в нашем браузере.

Документация в Java-коде должна начинаться с косой черты, а затем идут две звездочки, и заканчивается одной звездочкой, а затем косой чертой.

```
/**
 * Description: Separate paragraphs with <p>
 * Blank comment line
 * @ tags
 */
```

```
/**
 * Returns the number of pixels of an image
 *
 * @param x the width of the image, measured in pixels
 * @param y the length of the image, measured in pixels
 * @return the number of pixels of the image
 */
int numberPixels(int x, int y);
```

Javadoc просматривает вашу программу, ищет строки, начинающиеся с косой черты и двух звездочек, и создает HTML-документацию.

Но почему мы должны использовать этот комментарий?

Вместо поиска комментариев в миллионах строк кода, вы можете открыть веб-страницу и найти всю важную информацию о программе.

Когда мы говорим в Java об автоматической генерации документации, мы используем термин Javadoc.

Какую информацию мы должны включить в Javadoc?

На сайте Oracle вы можете найти руководство по эффективной практике написания комментариев для инструмента Javadoc.

Мы попытаемся обобщить наиболее важные из них, используя пример.

Мы начнем с определения Javadoc-комментария.

Комментарий Javadoc написан в формате HTML и должен предшествовать коду.

Он состоит из двух частей: описания и блока тегов.

Рассмотрим теги, которые вы должны использовать и как их использовать.

Давайте посмотрим на метод, который здесь указан, и вид информации, которая должна быть предоставлена для него в Javadoc.

Вы должны начать свой комментарий Javadoc с краткого и полного описания того, что делает этот метод.

Если в вашем Javadoc-комментарии есть несколько абзацев, разделите их тэгом `p`.

Затем вставьте пустую строку комментария, между описанием и блоком тегов.

Обратите внимание, что каждый комментарий Javadoc имеет только одно описание.

И как только инструмент Javadoc найдет пустую строку, он решит, что описание закончено.

Затем вы используете теги для добавления информации о вашем методе.

Наконец, вы должны поместить в конце строку со звездочкой и косой чертой, чтобы отметить конец комментария Javadoc.

Какая информация должна быть включена в блок тегов?

Для описания метода нам понадобятся, в основном, два типа тегов – @param и @return.

@param описывает аргумент метода.

И его необходимо указать для всех аргументов метода.

За тегом всегда следует имя аргумента.

Это имя всегда указывается в нижнем регистре.

Затем идет описание аргумента.

Далее вы должны всегда указывать тип данных аргумента.

Единственным исключением является тип данных, int, который вы можете опустить.

Чтобы разделить имя, описание и тип данных аргумента, вы можете добавить один или несколько пробелов.

Теги @param должны быть перечислены в порядке объявления аргумента.

Что касается описания, если это фраза без глагола, начните его с маленькой буквы.

Если это предложение с глаголом, начните его с заглавной буквы.

Таким образом, Javadoc – это полезный инструмент, который позволяет программистам автоматически генерировать HTML-страницы с документацией из их кода.

Исключения



Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 14

Исключения

Когда мы говорили о том, что мы можем сделать в случае метода, который не определен для всех возможных входных значений, мы сказали, что мы можем запрограммировать, что нужно сделать в исключительной ситуации.

То есть мы можем программировать, что делать в обычных случаях, и что делать в исключительных случаях.

Exceptions

```
double sqrt (int x) throws NegEx{  
    if (x<0) {throw new NegEx();}  
    return ...;  
}
```

Это делает нашу программу более надежной.

Таким образом, у нас есть исключения.

В этом случае мы используем не комментарии, а используем конструкции программирования языка Java.

Мы программируем, что делать для значений, которые не желательны.

Часто бывает, что наши программы хорошо написаны, их синтаксис и последовательность инструкций верны.

И они прекрасно компилируются.

Но когда мы их запускаем, возникают ошибки.

Java обрабатывает ошибки, возникающие в наших программах, во время выполнения с использованием исключений.

Огасле определяет исключения как события, которые происходят во время выполнения программы, и которые нарушают нормальный поток выполнения инструкций.

Однако важно учитывать, что совсем не плохо иметь программы, которые выбрасывают исключения.

Исключения позволяют отделить основную логику программы от действий, которые нужно предпринять, когда происходит что-то необычное.

Кроме того, исключения позволяют классифицировать и дифференцировать типы ошибок систематическим образом.

Таким образом, исключение – это ошибка, возникающая во время выполнения программы. Исключения могут возникать во многих случаях, например:

Пользователь ввел некорректные данные.

Или файл, к которому обращается программа, не найден.

Или сетевое соединение с сервером было утеряно во время передачи данных.

Рассмотрим некоторые из наиболее распространенных исключений в программах Java, а также механизм их обработки, чтобы обеспечить нормальный поток программы, даже если во время выполнения происходят ошибки.

Первое исключение, которое мы здесь видим, это `ArithmeticException`.

```
ArithmeticException

int a = 1;
int b = 0;
System.out.println(a/b);

int a = 1;
int b = 0;
System.out.println(a*b);

void printDivision (int a, int b){
    if (b != 0){
        System.out.println(a/b);
    } else {
        System.out.println("NaN");
    }
}
```

Это исключение выбрасывается при возникновении арифметических ошибок, например, при делении целого на ноль.

Эти сегменты кода вызывают исключение `ArithmeticException`.

Если вы программируете метод, в котором вы используете математическое деление, вы всегда должны проверять, что делитель отличается от нуля, так как вы не знаете, какие значения будут иметь аргументы, которые вы принимаете в этом методе.

Другим распространенным исключением является `ArrayIndexOutOfBoundsException`.

```
ArrayIndexOutOfBoundsException

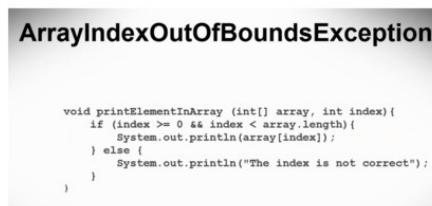
0 1 2 3
0 1 2 3

int[] array = {0, 1, 2, 3};
System.out.println(array[4]);
```

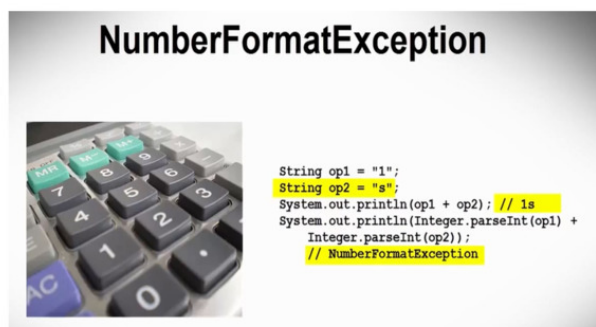
Это исключение вызывается, когда пытаются получить доступ к элементу массива с недействительным индексом.

Если мы определим, например, четырехэлементный массив, то это исключение будет выбрасываться при попытке доступа к элементу массива с индексом четыре или выше, а также при попытке доступа к элементу массиву с отрицательным индексом.

Поэтому, когда метод должен получить доступ к элементу в массиве, важно сначала проверить, что нужная позиция находится в пределах массива.

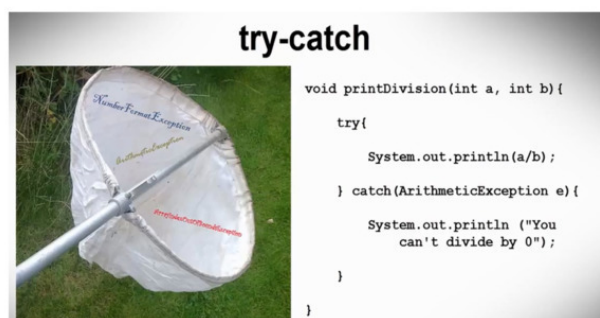


Еще одно исключение, это NumberFormatException.



Это исключение вызывается, когда пытаются преобразовать строку в числовой тип, например, double или integer, но при этом строка не содержит числа.

Стандартный способ управления этими инструкциями, которые могут выбросить исключение, это заключить их в оператор try-catch.



Здесь вы видите, что код метода printDivision заключен в выражение try-catch.

В этом случае нет необходимости проверять значение b, делителя.

Если b отличен от нуля, будет выполнен метод System.out.println (a / b);

В противном случае Java выбросит исключение ArithmeticException с сообщением, что вы не можете делить на ноль.

Поток программы будет продолжен, как обычно, после обнаружения этого исключения.

Выражение try-catch также может быть применено к примерам ArrayIndexOutOfBoundsException и NumberFormatException, которые мы видели ранее.

try-catch

```

void printElementInArray (int[] array, int index){
    try {
        System.out.println(array[index]);
    } catch (ArrayIndexOutOfBoundsException e){
        System.out.println ("The index is out of the bounds of the array");
    }
}


void printInteger (String s){
    try{
        System.out.println(Integer.parseInt(s));
    } catch (NumberFormatException e){
        System.out.println("The argument received is not an Integer");
    }
}

```

В обоих случаях, и благодаря выражению try-catch, мы можем гарантировать, что, если возникает исключение, пользователь получит соответствующую информацию, и поток выполнения программы продолжится далее нормально.

Вы также можете не обрабатывать исключения блоком try-catch в методе, а передать это право вызывающему этот метод.

Responsibility of handling exception



```

тип имя_метода(список_параметров) throws
список_исключений {
    // код внутри метода
}

public void method(int i) throws IllegalArgumentException {

}

```

В этом случае вы используете ключевое слово throws, которое прописывается в сигнатуре метода, и обозначающее что этот метод потенциально может выбросить исключение с указанным типом.

И уже в вызывающем коде обработать вызов этого метода блоком try-catch.

Также вы можете сами, специально, не виртуальная машина, а вы – выбросить исключение с помощью ключевого слова throw, указав тип исключения.

```

public void method(int i) throws IllegalArgumentException {
    if (i < 0)
        throw new IllegalArgumentException();
    // ...
}

```

Says "this method throws the checked exception IllegalArgumentException".

Throws an IllegalArgumentException.

Например, чтобы предотвратить выполнение кода, когда параметр метода не является возможным входным значением.

Таким образом, ключевое слово `throw` – служит для генерации исключений.

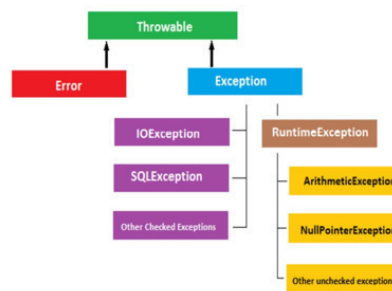
Блок `try` может иметь несколько блоков `catch`, каждый из которых имеет дело с конкретным исключением.

try-catch

```
void printElementInArray(int[] array, String index){
    try {
        System.out.println(array[Integer.parseInt(index)]);
    } catch (NumberFormatException e) {
        System.out.println("The index is not an integer");
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("The index is out of the bounds of the array");
    }
}
```

Если блок `try` генерирует исключение, то соответствующий блок `catch` обработает исключение, и программа будет продолжена.

Встроенные исключения Java имеют определенную иерархию.



Все классы, представляющие ошибки являются наследниками класса `java.lang.Throwable`.

Только объекты этого класса или его наследников могут быть «выброшены» JVM при возникновении какой-нибудь исключительной ситуации, а также только эти исключения могут быть «выброшены» во время выполнения программы с помощью ключевого слова `throw`.

Поэтому, если вы хотите создать свой класс исключения, он должен происходить от класса `Throwable`, или более точнее от класса `Exception`.

Также нужно учитывать, что все исключения делятся на «проверяемые» (`checked`) и «непроверяемые» (`unchecked`).

`checked exception` – проверяемое исключение, которое проверяется компилятором.

`Throwable` и `Exception` и все их наследники, за исключением наследников `Error`-а и `RuntimeException` – проверяемые.

`Error` и `RuntimeException` и все их наследники – не проверяемые компилятором исключения.

Компилятор при компиляции проверяет код на возможность выброса при выполнении кода проверяемого исключения.

И так как проверяемое исключение проверяется во время компиляции, возникнет ошибка компиляции, если проверяемое исключение не обработано блоком `try-catch`, или оно не объявлено в заголовке или сигнатуре метода с помощью ключевого слова `throws`.

```

public class App {
    public static void main(String[] args) {
        f(); // тут ошибка компиляции
    }

    public static void f() throws Exception {
    }
}

```

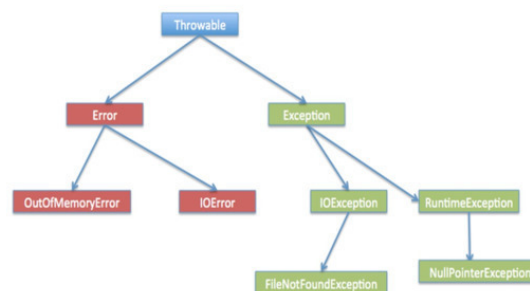
Так почему не все исключения являются проверяемыми?

Дело в том, что если проверять каждое место, где теоретически может быть ошибка, то ваш код сильно разрастется, и станет плохо читаемым.

И язык Java будет полностью непригодным для использования в качестве языка программирования.

Например, в любом месте, где происходит деление чисел, нужно было бы проверять на исключение `ArithmeticException`, потому что возможно деление на ноль.

Эту проверку создатели языка оставили программисту на его усмотрение.



Таким образом, исключение `RuntimeException` является не проверяемым и выбрасывается во время выполнения Java кода, и его дочерние исключения также являются не проверяемыми.

Это исключение `IndexOutOfBoundsException` – выбрасывается, когда индекс некоторого элемента в структуре данных не попадает в диапазон имеющихся индексов.

Исключение `NullPointerException` – выбрасывается, когда ссылка на объект, к которому вы обращаетесь, хранит `null`.

Исключение `ClassCastException` – это ошибка приведения типов.

И исключение `ArithmeticException` – выбрасывается, когда выполняются недопустимые арифметические операции, например, деление на ноль.

Исключение `Error` также является не проверяемым, которое показывает серьезные проблемы возникающие во время выполнения приложения. Исключение `Error` сигнализирует о ненормальном ходе выполнения программы, т.е. о каких-то критических проблемах.

И его дочерние исключения, также не проверяемые, `ThreadDeath` – вызывается при неожиданной остановке потока.

Исключение `StackOverflowError` – ошибка переполнение стека. Часто возникает в рекурсивных функциях из-за неправильного условия выхода.

И исключение `OutOfMemoryError` – ошибка переполнения памяти.

Из описания этих не проверяемых исключений видно, что обработать все эти возможные ситуации в коде невозможно, иначе весь код – это будет сплошной `try-catch`.

Теперь, при использовании множественных операторов `catch` обработчики подклассов исключений должны находиться выше, чем обработчики их суперклассов.

Иначе, суперкласс будет перехватывать все исключения, имея большую область перехвата.

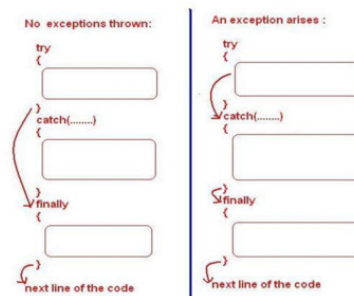
Иными словами, `Exception` не должен находиться выше `ArithmeticException` и `ArrayIndexOutOfBoundsException`.

И еще, операторы `try` могут быть вложенными.

Если вложенный оператор `try` не имеет своего обработчика `catch` для определения исключения, то идёт поиск обработчика `catch` у внешнего блока `try` и т. д.

Если подходящий `catch` не будет найден, то исключение обработает сама система завершением программы.

Таким образом, проверка на проверяемые исключения происходит в момент компиляции, а перехват исключений блоком `catch` происходит в момент выполнения кода.



Теперь, есть еще одна конструкция в обработке исключений, это блок `finally`.

Когда исключение передано, выполнение метода направляется по нелинейному пути.

Это может стать источником проблем.

Например, при входе метод открывает файл и закрывает при выходе.

Чтобы закрытие файла не было пропущено из-за обработки исключения, используется блок `finally`.

Ключевое слово `finally` создаёт блок кода, который будет выполнен после завершения блока `try/catch`, но перед кодом, следующим за ним.

Блок будет выполнен, независимо от того, передано исключение или нет.

Оператор `finally` не обязателен, однако каждый оператор `try` требует наличия либо `catch`, либо `finally`.

Таким образом, блок `finally` всегда выполняется, когда блок `try` завершается.

Это гарантирует, что блок `finally` будет выполнен, даже если произойдет непредвиденное исключение.

Также блок `finally` позволяет программисту избежать случайного обхода нужного кода.

Включение необходимого для выполнения кода в блок `finally` всегда является хорошей практикой, даже если не ожидается никаких исключений.

Однако блок `finally` не всегда может выполняться.

Если виртуальная машина JVM завершает работу во время выполнения кода `try` или `catch`, блок `finally` может не выполняться.

Аналогично, если поток, выполняющий код `try` или `catch`, прерывается или убивается, блок `finally` может не выполняться, даже если программа в целом продолжается.

Блок `finally` – это ключевой инструмент для предотвращения утечек ресурсов.

Закрывая файл или восстанавливая ресурсы, поместите код в блок `finally`, чтобы гарантировать, выполнение необходимых операций.

Рассмотрим этот пример.

```
public static void main(String[] args) {
    System.out.println("sout");
    throw new Error();
}
```

Каким здесь может быть вывод в консоль?

Здесь вполне возможна ситуация, когда в консоль сначала будет выведено сообщение об ошибке, а только потом вывод `System.out.println`.

Так как вывод `System.out` является буферизированным, то есть сообщения сначала помещаются в буфер, прежде они будут выведены в консоль.

А сообщение необработанного исключения выводится через не буферизированный вывод `System.err`.

Как уже было сказано, каждый оператор `try` требует наличия либо `catch`, либо `finally`.

```
public static void main(String[] args) {
    try {
        System.err.println("try");
    } finally {
        System.err.println("finally");
    }
}

public static void main(String[] args) {
    try {
        return;
    } finally {
        System.err.println("finally");
    }
}

public static void main(String[] args) {
    try {
        System.exit(42);
    } finally {
        System.err.println("finally");
    }
}
```

Поэтому возможна конструкция `try – finally`.

И блок `finally` получит управление, даже если `try`-блок завершится исключением.

И блок `finally` получит управление, даже если `try`-блок завершится директивой выхода из метода.

Однако блок `finally` НЕ будет вызываться, если мы убьем виртуальную машину JVM.

При всем при этом, надо отметить, что блок `finally` не перехватывает исключение, и программа завершится ошибкой при возникновении в блоке `try` исключения.

Исключение перехватывает только блок `catch`.

Таким образом мы разобрали почти все случаи работы операторов `try`, `catch`, `throws`, `throw`, и `finally`.

Рекурсия



Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 15

Рекурсия

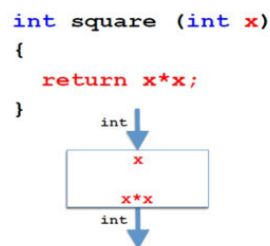
В некоторых случаях нам нужно выполнять повторные вычисления.

И мы видели циклы `for` и `while`, которые выполняют повторные вычисления.

Теперь мы увидим гораздо более мощный механизм повторных вычислений, который называется рекурсией.

Ранее мы определили метод `square`, который, принимая целое число, возвращает квадрат числа.

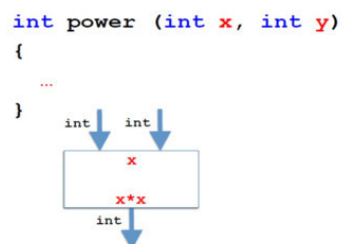
Square: x^2



Теперь мы хотели бы определить метод, который возводит в степень.

Мы хотим определить метод, который, учитывая базу `x` и показатель `y`, вычисляет `x` в степени `y`.

Power: x^y



Поэтому, если `y` равно 2, мы вычисляем квадрат числа, как и раньше.

Вы видите, что в этом методе мы имеем два аргумента, целые числа `x` и `y`.

Давайте сначала попытаемся определить этот метод.

Давайте проанализируем несколько случаев.

Если y равно 0, то результат x равен степени 0, т. е. 1.

Если y равно 1, результат будет сразу x .

Если y равно 2, результатом является квадрат x .

Мы можем вызвать метод `square`, который мы определили ранее.

Если y равно 3, мы имеем x в кубе, предполагая, что у нас есть метод, называемый `cube`, определенный заранее.

И далее нам понадобятся другие методы для всех различных значений y , которые могут быть приняты.

Power: x^y

```
int power (int x, int y)
{
    if (y==0)      {return 1;}
    else if (y==1) {return x;}
    else if (y==2) {return square(x);}
    else if (y==3) {return cube(x);}
    ... // until when?
}
```

Теперь мы можем заменить вызовы методов `square`, `cube`, и т. д. следующим кодом.

Таким образом, мы будем иметь x умножить на x , x умножить на x умножить на x и т. д.

Power: x^y

```
int power (int x, int y)
{
    if (y==0)      {return 1;}
    else if (y==1) {return x;}
    else if (y==2) {return x*x;}
    else if (y==3) {return x*x*x;}
    ... // not very intelligent!
}
```

Сейчас это немного лучше, но все же очень плохо, потому что порождает бесконечный код.

Но мы все же кое-чему научились.

Чтобы вычислить x в степени y , мы должны умножить x y раз.

Но мы должны учитывать, является ли эта процедура применима для всех целых чисел y ?

Нет.

Только для y больше или равно 0.

Для отрицательного y нам понадобится другой способ умножения.

Если у нас есть повторное умножение, мы можем использовать цикл.

Power: x^y

```
int power (int x, int y)
{ // y>=0
  int z=1;
  for (int i=1; i<=y; i++)
    {z=x*z;}
  return z;
}
```

Вот пример того, как мы можем это сделать.

Мы инициализируем целочисленную переменную z в 1, а затем вводим цикл. Счетчик i инициализируется 1 и увеличивается на 1 при каждом прогоне цикла. Этот счетчик отслеживает, сколько x мы умножаем и накапливаем с помощью z . И мы должны выполнять тело цикла ровно y раз, пока i не станет равен y . Затем мы выходим и возвращаем накопленное значение в z . Давайте проанализируем это снова.

- Precondition: $y \geq 0$
 - $x^y = 1$ if $(y == 0)$
 - $x^y = x * x^{y-1}$ if $(y > 0)$
 - $\text{power}(x, y) = 1$ if $(y == 0)$
 - $\text{power}(x, y) = x * \text{power}(x, y-1)$ if $(y > 0)$
- Precondition satisfied $\leftarrow y \geq 0 \leftarrow y > 0 \leftarrow y > 0$

x в степени y равно 1, если y равно 0.

А если y строго больше 0, то x в степени y равно x умножить на x в степени y минус 1. Это то, что в математике называется рекуррентным уравнением.

И мы можем написать это на Java в виде вызова функции `power`.

Если y равно 0, возвращаем 1.

Recursive Method

```
int power (int x, int y)
{ // y>=0
  if (y==0)
    return 1;
  else
    return x*power(x,y-1);
}
```



Иначе, возвращаем x умножить на вызов этой же функции с x и y минус 1.

Таким образом, тот же метод, который мы определили с помощью цикла, может быть определен с помощью рекурсии.

Оба эти способа эквивалентны.

Но рекурсия позволяет записать сложное поведение простым способом, который требует довольно сложного программирования при использовании циклов.

Рекурсию можно сравнить с матрешкой.



Чтобы понять это вернемся к рекурсивному методу, который мы определили.

И давайте упростим последовательно вызов этого метода для небольшой степени, чтобы увидеть, что происходит.

Начнем с x в 3 степени.

Мы можем заменить вызов метода, используя определение метода.

power (x, 3)

```
• power(x,3)
• if (3==0){return 1}
  else return x*power(x,3-1)
• if (false){return 1}
  else return x*power(x,2)
• return x*power(x,2)

• power(x,3) → x*power(x,2)
```

Таким образом, мы пишем весь код метода, подставляя вместо y 3.

И в этой последовательности выражений мы переходим от вызова метода с параметрами $(x, 3)$ к вызову метода с параметрами $(x, 2)$.

Пишем весь код метода, подставляя вместо y 2.

power (x, 2)

```

• power(x, 2)
• if (2==0){return 1}
  else return x*power(x, 2-1)
• if (false){return 1}
  else return x*power(x, 1)
• return x*power(x, 1)

• power(x, 2) → x*power(x, 1)

```

И в этой последовательности выражений, мы перешли от вызова метода с параметрами (x, 2) к вызову метода с параметрами (x, 1).

И переходим к вызову метода с параметрами (x, 0).

power (x, 1)

```

• power(x, 1)
• if (1==0){return 1}
  else return x*power(x, 1-1)
• if (false){return 1}
  else return x*power(x, 0)
• return x*power(x, 0)

• power(x, 1) → x*power(x, 0)

```

x в степени 0 равно 1.

power (x, 0)

```

• power(x, 0)
• if (0==0){return 1}
  else return x*power(x, 0-1)
• if (true){return 1}
  else return x*power(x, -1)
• return 1

• power(x, 0) → 1

```

Теперь нам нужно собрать все вместе.

power (x, 3) равно x умножить на power (x, 2).

- `power(x, 3)`
- `x * power(x, 2)`
- `x * (x * power(x, 1))`
- `x * (x * (x * power(x, 0)))`
- `x * (x * (x * 1))`
- `x * (x * x)`

А `power(x, 2)` равно `x` умножить на `power(x, 1)`.

А `power(x, 1)` равна `x` умножить на `power(x, 0)`, что равно 1.

Таким образом, мы получаем `x` умножить на `x` умножить на `x` умножить на 1.

Так работает рекурсия – сначала мы спускаемся как по лестнице вниз, а затем поднимаемся опять вверх.

Это изображение коробки с медсестрой, держащей меньшую коробку с тем же изображением.



Так что в теории, могут быть бесконечные медсестры и бесконечные коробки.

Но на практике нет бесконечных коробок, потому что изображение имеет некоторое разрешение, и мы не можем опуститься ниже 1 пикселя.

Таким образом, существует конечное число коробок.

Когда мы что-то вычисляем, мы должны заботиться о том, чтобы не создавать нежелательные бесконечные вычисления, которые нарушают нормальный поток вычислений.

Давайте посмотрим, что произойдет, когда мы что-то неправильно программируем.

Давайте рассмотрим, опять наш рекурсивный метод вычисления степени числа.

И давайте вызовем `power(x, -2)` для некоторого заданного `x`.

power (x,y)

```
int power (int x, int y){ // y>=0
    if (y==0)
        return 1;
    else
        return x*power(x,y-1);
}
```

Для этого мы можем заменить вызов метода кодом.

power (x, -2)

- power(x,-2)
- if (-2==0){return 1}
- else return x*power(x,-2-1)
- if (false){return 1}
- else return x*power(x,-3)
- return x*power(x,-3)
- power(x,-2) → x*power(x,-3)

В результате мы перейдем к вызову метода power (x, -3).

В методе power (x, -3) мы перейдем к вызову метода power (x, -4).

power (x, -3)

- power(x,-3)
- if (-3==0){return 1}
- else return x*power(x,-3-1)
- if (false){return 1}
- else return x*power(x,-4)
- return x*power(x,-4)
- power(x,-3) → x*power(x,-4)

И так далее. Без конца.

- power(x,-2) → x*power(x,-3)
- power(x,-3) → x*power(x,-4)
- power(x,-4) → x*power(x,-5)
- power(x,-5) → x*power(x,-6)
- ...

Мы получим бесконечные вычисления в теории.

На практике мы получим переполнение в какой-то момент и ошибку.

Что же мы сделали не так?

В этом случае мы не соблюдали комментарий, что у должно быть больше или равно 0.

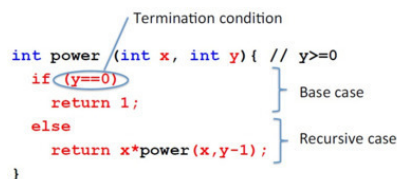
Поэтому мы должны учитывать две важные вещи.

Во-первых, рекурсия хороша, но мы можем перейти к бесконечным вычислениям.

И во-вторых, чтобы избежать этого, мы должны понять условия, при которых рекурсивный метод фактически завершается.

Может быть определенное количество рекурсивных вызовов, но в какой-то момент, нам нужно достичь не рекурсивного случая.

Поэтому при определении рекурсивного метода, всегда должны быть некоторые значения, для которых метод не вызывается рекурсивно.



```

int power (int x, int y){ // y>=0
    if (y==0)
        return 1;
    else
        return x*power(x,y-1);
}
  
```

Termination condition

Base case

Recursive case

Существует два способа чтения и понимания рекурсивных методов.

Один из них – это тот способ, который мы видели.

Другой, математический или нотационный способ, которые мы рассмотрим.

Предположим, нам дана задача написать рекурсивный метод.

Начнем с относительно простой задачи – написать метод на Java для вычисления факториала натурального числа.

В общем случае факториал натурального числа n вычисляется умножением всех натуральных чисел, начиная с 1 до n .

```

fac(n) = n! = n*(n-1)*...*2*1

fac(5) = 5! = 5*4*3*2*1
fac(4) = 4! = 4*3*2*1
  
```

Чтобы решить эту задачу, мы будем использовать следующую стратегию.

Первая часть состоит в том, что мы предполагаем, что задача решена для более простой задачи того же рода.

Предположим, что нам нужно вычислить факториал натурального числа n , но мы уже знаем, как вычислить факториал n минус 1.

$$\begin{aligned} \text{fac}(n) &= n! = n * (n-1) * \dots * 2 * 1 \\ \text{fac}(n-1) &= (n-1)! = (n-1) * \dots * 2 * 1 \\ n! &= n * (n-1)! \\ \text{fac}(n) &= n * \text{fac}(n-1) \end{aligned}$$

Если бы у нас был факториал n минус 1, мы просто бы умножили это число на n , чтобы получить факториал n .

Вторая часть стратегии – выявить случай, когда предыдущее рассуждение не выполняется.

Факториал 0 нельзя свести к более простому случаю, как мы это делали ранее.

$$\text{fac}(0) = 0! = 1$$

Так что это базовый случай.

Мы просто говорим, что факториал 0 равен 1.

Таким образом, факториал n равен 1, если n равно 0, и факториал n равен n умножить на факториал n минус 1, если n больше 0.

Теперь у нас есть основа для записи рекурсивного метода.

Из математического уравнения легко написать рекурсивный метод.

$$\text{fac}(n) = \begin{cases} 1 & (\text{if } n \leq 1) \\ n * \text{fac}(n-1) & (\text{if } n > 1) \end{cases}$$

```

long fac (int n) {
    if (n <= 1) return 1;
    else return n * fac(n-1);
}

```

} Base case
 } Recursive case

Там мы видим базовый случай, в котором нет рекурсивного вызова.

Базовый случай получается из пограничного случая.

И мы также видим рекурсивный случай, вытекающий из приведения общего случая к более простому.

Инкапсуляция. Объекты и классы



Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 1

Инкапсуляция. Объекты и классы

Давайте посмотрим на вычислительные возможности калькулятора.

Как правило, калькулятор может делать две вещи: запомнить значения и вычислить новые значения.



Запомнить значения можно с помощью переменных.

И затем мы можем вычислять новые значения с помощью методов.

Например, мы можем сложить два значения, вычесть или умножить.

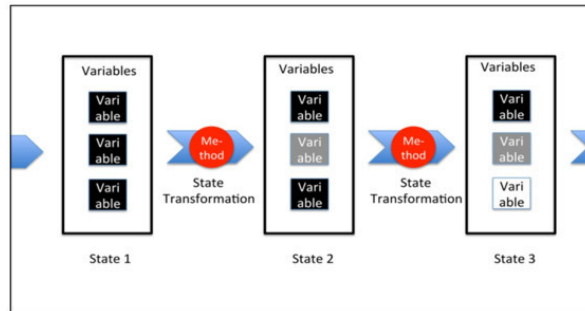
Таким образом, у нас есть методы, соответствующие арифметике, а также методы, чтобы получить или установить переменную *x*.

Когда мы пишем программу для моделирования этого калькулятора, и мы определяем для него переменные и методы, мы поместим, с одной стороны, все переменные вместе, а с другой стороны – все методы вместе.

Значения всех переменных в конкретный момент времени будут составлять состояние калькулятора.

И набор методов будет определять поведение калькулятора.

Наша модель будет меняться от одного состояния в другое со временем.



При этом состояние будет определяться значениями переменных.

А методы будут отвечать за изменение состояния.

На самом деле, определение переменных и методов – это общий способ моделирования объектов.

Эти объекты могут соответствовать физическим объектам, например, калькулятору.

Или эти объекты могут быть концептуальными, когда ваш код должен моделировать что-то новое.

Таким образом, это разделение состояния и поведения очень важно.

Представьте себе автомобиль, который моделируется в программе, которую вы пишете для игры.

Состоянием этого объекта может быть местоположение, цвет, включены ли фары или нет.

И методы могут быть изменением положения, включить свет фар и т. д.

Помните, что методы часто связаны с глаголами, потому что они подразумевают действие.

Теперь мы собираемся инкапсулировать переменные и методы в новую для нас конструкцию программирования, называемую объектом.

Эта концепция инкапсуляции является одной из ключевых концепций в так называемой объектно-ориентированной парадигме программирования.

Поэтому помните, что объекты имеют состояние, представленное отдельными переменными, которые также называются полями или атрибутами.

И поведение, то, что может делать наш объект, представлено методами.

Эти два компонента: состояние и поведение, не разбросаны по программе, а собраны и инкапсулированы в объекты.

Разные объекты могут иметь одинаковую структуру, и отличаться друг от друга только значениями переменных.

Поэтому мы можем сказать, что такие объекты принадлежат одному и тому же классу.

И наоборот, чтобы создать объект, сначала нам нужно сначала определить класс, который является шаблоном для создания объектов.

Рассмотрим пример с различными автомобилями, которые представлены различными объектами.

Все эти объекты принадлежат классу автомобилей Car, который имеет ряд атрибутов или переменных, или полей и ряд методов.

Давайте посмотрим на возможное определение, как мы можем записать этот класс на Java.

```

class Car {
    boolean lights;           Attributes
    String color;

    ...

    void turnHeadlightsOn() {lights = true;}
    void turnHeadlightsOff() {lights = false;}
    void moveForward() {...;}
    void moveBackward() {...;}
    void turnRight() {...;}   Methods
    void turnLeft () {...;}
}

```

Здесь вы можете увидеть определение класса Car.

Вы можете увидеть зарезервированное ключевое слово class.

Затем имя, которое мы хотим дать классу.

Обратите внимание, что мы пишем его с заглавной буквы.

Затем мы указываем переменные с соответствующим типом.

Наконец, мы определяем методы, которые мы хотим дать всем объектам этого класса.

Но как только мы определили класс, как сконструировать объект для этого класса?

Для этого у нас есть конструкторы.

Конструкторы – это специальные методы, которые также включены в тело определения класса.

No return type Constructors

```

Car() {lights=false; color="white";}

Car(String c) {lights=false; color=c;}
Car(boolean b) {lights=b; color="white";}

Car(boolean b, String c)
{lights=b; color=c;}

```

И они имеют имя класса.

Обратите внимание, что здесь не указан тип возвращаемого результата.


Используя конструкторы, мы можем создавать разные объекты класса.

Constructors

```

Car c1 = new Car();
Car c2 = new Car("green");
Car c3 = new Car(true);
Car c4 = new Car(true, "red");

```



The diagram illustrates the state of four car objects created using different constructors. Each object is represented by a light bulb icon (indicating the 'lights' attribute) and a colored square icon (indicating the 'color' attribute).
 - c1: lights are off (grey bulb), color is white (white square).
 - c2: lights are off (grey bulb), color is green (green square).
 - c3: lights are on (yellow bulb), color is white (white square).
 - c4: lights are on (yellow bulb), color is red (red square).

Заметьте, что может быть не один, а несколько конструкторов.

Эти конструкторы отличаются списком параметров.

Здесь вы можете увидеть несколько возможных конструкторов для класса `Car`.

Также мы можем вообще не определять конструктор, и в этом случае при вызове конструктора объект создается со значениями по умолчанию.

Здесь мы видим несколько вызовов конструкторов, определенных ранее.

Посмотрите на объявление.

Сначала мы определяем объект с именем и обратите внимание, что классы работают как типы.

Сначала мы указываем `Car`, чтобы указать, что объект имеет тип или класс `Car`.

Затем знак равенства, зарезервированное ключевое слово `new` и вызов конструктора.

Таким образом, в итоге, чтобы определить объект, мы должны сначала определить класс, предоставляя набор полей и набор методов.

После определения класса мы можем создать объект как экземпляр класса, используя конструктор, предоставляемый классом.

Мы можем создать много объектов одного класса, каждый из которых будет со своим собственным состоянием.

Классы и типы



Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 2

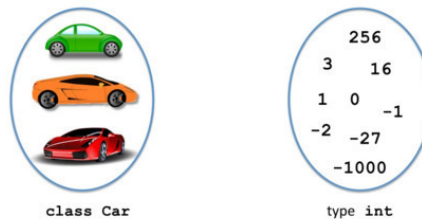
Классы и типы

Классы – это шаблоны, из которых мы строим объекты.

И все объекты имеют одинаковую структуру, определенную классом.

Давайте сравним класс, который мы определили, со встроенным Java типом.

Так, например, с одной стороны, у нас есть класс «Car», который мы определили с такими методами, как «двигаться вперед» или «включать фары» и поля, такие как «свет» и «местоположение».



И, с другой стороны, у нас есть целые числа типа «int».

И для этих целых чисел у нас есть ряд определенных операций или методов, таких как «сложение» или «умножение».

Давайте сосредоточимся на методах.

В обоих случаях методы связаны с объектами в классе или значениями данного типа.

Таким образом, классы похожи на типы, и объекты похожи на сложные значения.

Фактически, вы можете рассматривать классы как типы.

Типы, которые не являются встроенными Java типами, а типы, которые вы определили для решения какой-либо конкретной задачи.

При определении методов и конструкторов классы принимают роль типов.

Действительно, мы использовали строки так же как целые числа, для определения методов и переменных.

```

int a = 3;
String b = "abc";

String repeat(String s, int n){
    String t="";
    for (int i=0; i<n; ++i){t=t+s;}
}

```

И String- это класс, а «int» – это примитивный тип данных.

Здесь мы видим объявление переменной целого числа и переменной строки.

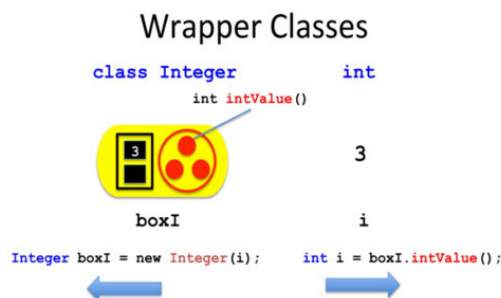
Иногда мы говорим о «ссылках», в случае объектов.

В нижней части мы видим объявление метода со String и «int» в качестве параметров.

Таким образом, вы можете рассматривать классы как типы – типы, определенных вами в соответствии с вашими потребностями.

На самом деле для каждого примитивного типа существует соответствующий класс, называемый «классом-оболочкой».

Например, у нас есть тип «int» и класс «Integer».



И этот класс Integer является классом-оболочкой.

Объект класса «Integer» содержит поле с числом «int» и метод, который возвращает число «int», сохраненное в этом объекте.

Кроме того, там есть другие поля и методы, которые используются для разных целей.

Как вы можете видеть, для преобразования числа «int» в объект «Integer», мы можем использовать конструктор «Integer».

И для преобразования объекта Integer в значение «int», мы используем метод «intValue» класса «Integer».

Представление просто «int» в компьютере намного эффективнее, чем соответствующего объекта, так как существует много вещей, которые нужно хранить в объекте.

Класс – это не просто тип.

Во-первых, потому что он может содержать более одного поля.

Это можно понимать, как составное значение – значение с несколькими компонентами – например, тремя целыми числами.

Поэтому, классы – это хороший способ собрать несколько значений вместе в полях объекта.

И эти компоненты могут быть нескольких типов или классов.

В частности, они могут быть довольно сложными объектами, определенными как части данного объекта.

Представьте, что вы определили класс «Двигатель» с набором полей, которые определяют состояние двигателя и набором методов, которые определяют то, что вы можете делать с двигателем.

```
class Car {
    boolean lights;
    String color;
    Engine eng;

    ...
    void turnHeadlightsOn() {lights = true;}
    void moveForward() {...}
    void tuneEngine(int n) {...}
    ...
}
```

У нас может быть объект класса «Двигатель» как атрибут или поле класса «Автомобиль». Это дает большие возможности, так как концепция класса позволяет структурировать вашу программу или систему, определяя различные подсистемы.

Вы можете создавать объекты, используя другие объекты.

Но концепция объекта класса еще богаче.

Мы могли бы рассматривать тип данных как набор значений, вместе с некоторыми методами для них.

И у нас есть переменные, которые могут хранить эти значения.

Но значение само по себе не имеет состояния, и сам по себе тип данных не имеет состояния.

Напротив, объект имеет состояние.

Так как он имеет внутри переменные, и он может запоминать значения.

Классы можно рассматривать как типы, классы определяют типы.

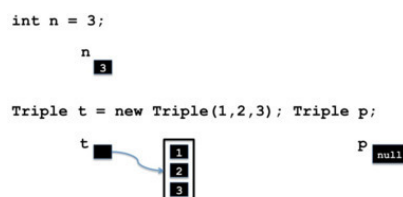
Но, кроме того, объекты имеют состояние.

Когда мы создаем новый объект – и мы делаем это с помощью ключевого слова «new» и конструктором – строки здесь являются исключением – и мы резервируем пространство в памяти для хранения значений полей.

После создания мы можем ссылаться на этот объект, создавая ссылку с именем.

Переменная с именем объекта будет хранить ссылку на объект.

Но в этих объяснениях мы не будем акцентировать внимание на идеи ссылки или указателя, поскольку Java как-то не поддерживает эту идею.



Другие языки программирования делают это.

Мы также не будем говорить об уничтожении объектов и освобождении памяти.

Потому что Java делает это автоматически с помощью так называемого «сборщика мусора», который автоматически освобождает память для объектов без какой-либо ссылки.

Когда ссылка указывает на отсутствие объекта, мы говорим, что его содержимое равно нулю.

Когда вы создали новый объект и дали ему имя – или, вернее, определили ссылку для него – как вы получаете доступ к полю или атрибуту?

Ответ заключается в использовании точечной нотации.

Вы хотите сослаться на поле «n1» объекта «t».

```
Triple t = new Triple(1,2,3);
```

```
t.n1
```

```
t.get1()
```

```
get1(t)
```

Вы пишете «t.n1.»

И для методов мы делаем что-то подобное.

Чтобы вызвать метод «get1 ()» для объекта «t», мы пишем «t. get1 ()».

Область видимости



Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования Лекция 3 Область видимости

Классы имеют двойную цель.

Они могут использоваться для определения новых типов данных, которые связаны с решением задачи.

И также они служат для структурирования кода.

У нас есть инкапсулированные переменные и методы в классах.

Однако любая другая часть программы может изменять переменные и вызывать методы.

И иногда важно скрыть доступ к некоторым переменным или методам, чтобы контролировать возникновение возможных проблем.

```
class Car {  
    int gas; // from 0 to 100  
    ...  
    void move(int n){...; gas-=n; ...;}  
    void fill(int n){...; gas+=n; ...;}  
}
```



Представьте себе, что в нашей модели автомобиля у нас есть поле `gas`, которое служит индикатором оставшегося топлива в машине.

Представьте, что эта переменная должна содержать значение от 0 до 100, 0 означает пустой бак, а 100 – полный.

Теперь, когда автомобиль тратит при движении `n` единиц топлива, эти `n` единиц вычитаются из переменной `gas`.

Мы также можем заполнить бак на АЗС, и в этом случае переменная `gas` увеличивается.

Таким образом, топливо уменьшается с помощью метода перемещения и увеличивается с помощью метода заполнения.

Однако у нас может быть проблема, поскольку любая часть программы имеет доступ к этой переменной `gas`.

Кто-то может даже изменить переменную на отрицательное число, что не имеет смысла.

Таким образом, два метода, которые должны изменять и должны иметь доступ к этой переменной, не являются единственным контролем этой переменной, и мы должны каким-то образом ограничить этот доступ.



```
class Car {
    private int gas; // from 0 to 100
    ...
    public void move(int n) {...; gas-=n; ...;}
    public void fill(int n) {...; gas+=n; ...;}
}
```

Давайте посмотрим на эти два модификатора доступа, `public` и `private`.

На данный момент мы будем использовать их только для переменных и методов класса.

Здесь мы пишем `private` до объявления переменной `gas`.

Это означает, что мы можем получить доступ к ней только в классе, а не вне класса.

Два метода, `move` и `fill`, определяются как `public`, и поэтому могут быть вызваны вне класса.

Это типичная ситуация, чтобы иметь приватные переменные и публичные методы.

У нас также могут быть приватные методы, которые определены, например, как вспомогательные методы для других публичных методов.



```
public class Car {
    private int gas; // from 0 to 100
    ...
    public Car () {gas=0;}
    public void move(int n) {...; gas-=n; check();}
    public void fill(int n) {...; gas+=n; check();}
    private void check()
    {if (gas<0) gas=0; if (gas>100) gas=100;}
}
```

Здесь мы видим метод `check`, который вызывается из `move` и `fill`, но нам не нужно вызывать этот метод вне класса.

Наконец, мы также ставим ключевое слово `public` перед классом.

Его смысл станет понятным позже.

Таким образом, извне класса, как правило, мы имеем доступ только к методам, а не к переменным.

Доступ к переменным имеют только методы.

Здесь мы разделили понятия инкапсуляции и сокрытие информации.


Хотя для некоторых эти две концепции идут вместе, то есть инкапсуляция всегда подразумевает сокрытие информации.

Как правило, мы хотим иметь приватные переменные экземпляра и публичные методы, которые получают доступ к этим переменным.

Но мы должны запрограммировать это явно с помощью ключевых слов «`private`» и «`public`».


Всегда рекомендуется делать переменные приватными.

А затем определять публичные методы для установки значений переменных и получения значений переменных.



```
public class Car {
    private int gas; // from 0 to 100
    ...
    public Car () {gas=0;}
    public int getGas() {return gas;}
    public void setGas(int g) {gas=g;}
    ...
}
```

Как правило, название этих двух типов методов соответствует одному и тому же шаблону: Как правило, имена этих методов начинаются со слова «set» и начинаются со слова «get». Поэтому эти методы иногда называют сеттеры и геттеры. Заметим, что в методе setGas мы имеем параметр g, который присваивается полю gas. Иногда, мы хотим назвать параметр setGas тем же именем, что и переменную экземпляра. И с этим не возникает никаких проблем.




```
public class Car {
    private int gas; // from 0 to 100
    ...
    public Car () {gas=0;}
    public int getGas() {return gas;}
    public void setGas(int gas) {this.gas=gas;}
    ...
}
```

Однако, если мы хотим отличить визуально параметр от поля, мы можем использовать ключевое слово this и точку перед именем.

Это означает, что это имя относится к полю класса.

В некоторых случаях нам нужно иметь поля класса, которые имеют общее значение для всех объектов в классе.



```
public class Car {
    static int noWheels=4;
    private int gas; // from 0 to 100
    ...
    public Car () {gas=0;}
    public int getGas() {return gas;}
    public void setGas(int gas) {this.gas=gas;}
    ...
}
```

Эти переменные называются переменными класса, а не переменными экземпляра класса, и они объявляются с помощью ключевого слова «static».


Эти переменные не создаются для каждого созданного объекта класса.

Они создаются только один раз для всех объектов класса.

И если мы изменим это значение, оно будет изменено для всех объектов.

Если мы не хотим, чтобы эта переменная менялась,

Мы можем сделать ее константой, добавив ключевое слово «final».



```
public class Car {
    static final int NOWHEELS=4;
    private int gas; // from 0 to 100
    ...
    public Car () {gas=0;}
    public int getGas() {return gas;}
    public void setGas(int gas) {this.gas=gas;}
    ...
}
```

Мы можем также сделать это и для переменных экземпляра.

По соглашению, имена таких переменных пишутся в верхнем регистре, заглавными буквами.

Как показано здесь.

Значения финальных переменных могут быть установлены только один раз.

Таким образом, теперь у нас есть разные виды переменных.

С одной стороны, у нас есть локальные переменные.

Затем у нас есть переменные экземпляра, которые создаются для каждого объекта или экземпляра класса.

Каждый объект может иметь свое значение, хранящееся в этой переменной.

Мы можем использовать ключевое слово «this» для обозначения этих переменных.

И у нас есть переменные класса, которые создаются только один раз для всех объектов одного класса.

Они объявляются с ключевым словом «static».

Статические переменные инициализируются только один раз, при запуске выполнения кода, при загрузке класса.

Эти переменные будут инициализированы первыми, прежде чем будут инициализированы любые переменные экземпляра.

И если вы хотите сделать переменную экземпляра или переменную класса неизменной, вы добавляете ключевое слово «final».

Наследование



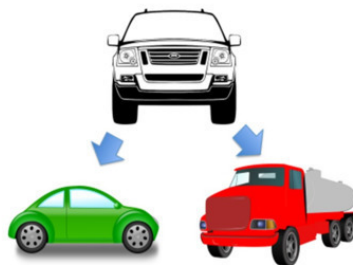
Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 4

Наследование

Рассмотрим две машины, принадлежащие к одному классу.

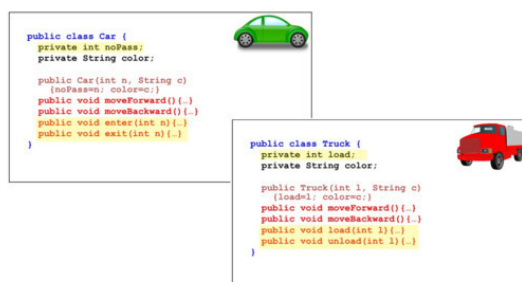


У них есть общие методы и поля, но в тоже время есть отличающиеся особенности.

И вместо того, чтобы создавать два разных объекта одного класса, а потом пытаться учесть их отличающиеся особенности с помощью отдельного кода, или создавать два разных несвязанных между собой класса, давайте сначала смоделируем общий класс автомобилей, а затем создадим класс легковых автомобилей и класс грузовиков, которые унаследуют общие поля и общие методы от общего класса автомобилей, но у них также будут и свои собственные поля, и методы.

Давайте посмотрим, как мы это делаем на Java.

Представьте, что у нас есть класс Car с этими полями и методами.



В частности, есть приватное поле количество пассажиров, `noPass`, которое содержит количество пассажиров в данный момент времени.

`enter` и `exit` – это методы, которые изменяют это число пассажиров.

Другой класс грузовиков имеет переменную загрузки, которая может быть изменена с помощью методов `load` и `unload`.

Имейте в виду, что не стоит называть переменную и метод одним и тем же именем.

Затем оба класса используют переменную `цвет`, а также методы для движения вперед и назад.

Что мы можем сделать для упрощения кода, так это сначала определить универсальный класс для транспортных средств.

Этот класс будет иметь поля и методы, общие для всех автомобилей – в нашем случае – для легковых автомобилей и грузовиков.

```
public class Vehicle{
    private String color;

    public Vehicle(String c) {color=c;}
    public void moveForward(){...}
    public void moveBackward(){...}
}
```



Затем мы можем определить классы `car` и `truck`, которые наследуют поля и методы от этого общего для них класса.

`Vehicle` будет называться суперклассом классов `car` и `truck`, и классы `car` и `truck` являются подклассами класса `Vehicle`.

Теперь мы можем определить класс `car`, расширив класс `Vehicle`, и добавить дополнительные поля и методы, которые может иметь легковой автомобиль.

```
public class Car extends Vehicle{
    private int noPass;

    public Car(int n, String c){...}
    public void enter(int n){...}
    public void exit(int n){...}
}
```



```
public class Truck extends Vehicle{
    private int load;

    public Truck(int n, String c){...}
    public void load(int n){...}
    public void unload(int n){...}
}
```



А для грузовых автомобилей мы делаем то же самое: расширяем класс `Vehicle` такими полями и методами, которые необходимы.

Все остальные поля и методы унаследованы от класса `Vehicle`.

Обратите внимание, что мы не раскрыли тело конструктора.

Это требует дальнейшего объяснения и новых концепций.

Но вы должны знать, что класс может иметь несколько подклассов, тогда как класс не может быть подклассом более чем одного класса.

У одного класса не может быть двух суперклассов, не может быть двух родителей. Таким образом, мы знаем, что один класс может расширить другой класс.

Например, если класс В расширяет класс А, это означает, что он наследует его поля и методы.

И это можно сделать многократно.

То есть класс В может быть расширен, например, классом С.

Теперь мы хотим проанализировать вопрос о том, как определить конструктор класса А, который расширяет другой класс.

В нашем определении класса vehicle и класса car, где класс car расширяет класс vehicle, мы определяем конструктор для класса vehicle, который инициализирует приватное поле color.

```
public class Vehicle{
    private String color;
    public Vehicle(String c) {color=c;}
    public void moveForward(){...}
    public void moveBackward(){...}
}
public class Car extends Vehicle{
    private int noPass;
    public Car(int n, String c)
    {color=c; noPass=n;}
    public void enter(int n){...}
    public void exit(int n){...}
}
```



И с этим не никаких проблем.

Но как мы можем определить тело конструктора car, с учетом двух аргументов, целого числа для количества пассажиров и строки для цвета?

Класс car наследует все методы от класса vehicle – перемещение вперед и назад, и все его поля, в данном случае, только color.

Но поле color является приватным полем и не может быть доступно извне класса vehicle. Это относится также и к подклассам, и это очень важно.

Поэтому неправильно присваивать значение «с» полю color в классе car.

Мы не можем получить к этому полю доступ, потому что оно является приватным.

Мы можем использовать только публичный метод, например, конструктор.

Теперь, если мы хотим вызвать конструктор суперкласса, мы используем ключевое слово super.

```
public class Vehicle{
    private String color;
    public Vehicle(String c) {color=c;}
    public void moveForward(){...}
    public void moveBackward(){...}
}
public class Car extends Vehicle{
    private int noPass;
    public Car(int n, String c)
    {super(c); noPass=n;}
    public void enter(int n){...}
    public void exit(int n){...}
}
```



Здесь вы это видите.

super (c) – вызов конструктора vehicle (c).

Таким образом, мы сможем инициализировать поле color из подкласса.


Вызов конструктора суперкласса должен быть перед любым другим кодом в теле конструктора подкласса.

Например, сначала установить количество пассажиров, а затем вызвать супер будет неправильным.

Вы должны сначала вызвать супер, а затем включить любой другой вызов, который вам может понадобиться.

```
public class Vehicle{
    private String color;
    public Vehicle(String c) {color=c;}
    public void moveForward() {...}
    public void moveBackward() {...}
}

public class Car extends Vehicle{
    private int noPass;
    public Car(int n, String c)
    {
        noPass=n; super(c);
    }
    public void enter(int n){...}
    public void exit(int n){...}
}
```

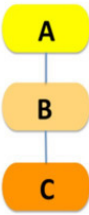


Здесь мы видим другой пример.


```
public class A {
    public A() {System.out.print("A, ");}
}

public class B extends A {
    public B() {super(); System.out.print("B, ");}
}

public class C extends B {
    public C() {super(); System.out.print("C.");}
}
```



```
C c = new C();
```



У нас есть класс А с подклассом В, а класс В с подклассом С.

Диаграмма справа от вас показывает отношения наследования.

Класс А имеет конструктор без аргументов, который печатает строку А, пробел.

В классе В мы видим, что есть также конструктор без аргументов, который правильно вызывает сначала конструктор суперкласса А, затем печатает строку В, пробел.

В классе С конструктор без аргументов сначала вызывает конструктор его суперкласса В, а затем печатает строку С точка.

Теперь, что происходит, когда мы создаем новый объект класса С?

Конструктор С вызывает конструктор В, который в свою очередь, вызывает конструктор А.

Таким образом, печатается: А, пробел, В, пробел, С точка.

Подводя итог, первое, что нам нужно сделать в конструкторе подкласса, это вызвать конструктор суперкласса.

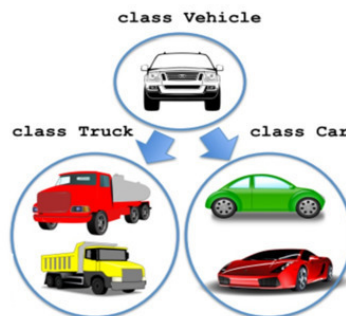
Приведение типов



Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования Лекция 5 Приведение типов

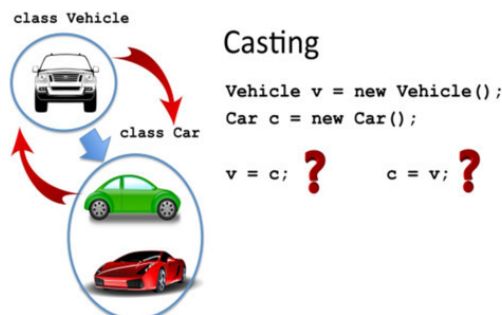
Давайте посмотрим снова на эту иерархию классов.



Легковой автомобиль и грузовик являются подклассами или производными классами класса vehicle.

Вопрос в том, если ли у нас есть объект класса car, мы можем использовать его там, где должны быть объекты класса vehicle?

Например, в переменной vehicle?

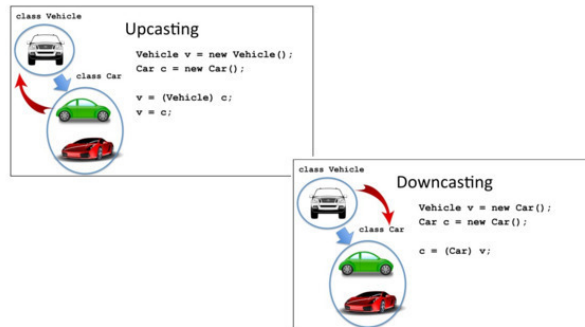


И наоборот, можем ли мы поместить объекты суперкласса там, где должны быть объекты подкласса?

И если да, то при каких обстоятельствах?

Мы говорим о кастинге или приведении при преобразовании объекта из одного класса к другому связанному классу.

Представьте себе, что у нас есть переменная `vehicle`, которая хранит объект `vehicle`, и переменная `car`, с сохраненным в нем объектом `car`.



Можем ли мы присвоить объект `car` переменной `vehicle` и наоборот?

Мы говорим о приведении к базовому типу при преобразовании объекта из класса в суперкласс.

И переход от подкласса к суперклассу всегда возможен.

Объекты подкласса наследуют все от суперкласса.

Поэтому все, что вы хотите сделать с переменной суперкласса, применимо к объекту подкласса.

Чтобы привести к базовому типу объект, вы можете указать суперкласс в круглых скобках, как вы здесь видите.

Но вы также можете не делать это, как вы видите в последней строке.

Мы говорим о понижающем приведении при конвертации объекта от класса к его подклассу.

Теперь мы хотим заставить `vehicle` стать `car`.

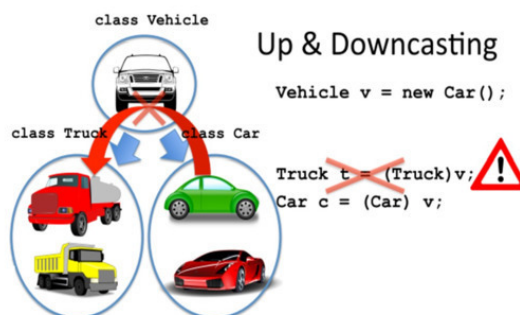
Мы переходим от общего класса к более конкретному классу, и это должно быть сделано явно.

В этом примере мы объявляем переменную типа `vehicle`, но храним в ней `car`.

Таким образом, мы можем явно понизить эту переменную для хранения `car`, который находится в переменной `v`.

Вы должны быть очень осторожны при кастинге вверх и вниз.

Мы объявляем переменную `v`, и мы храним в ней `car`.

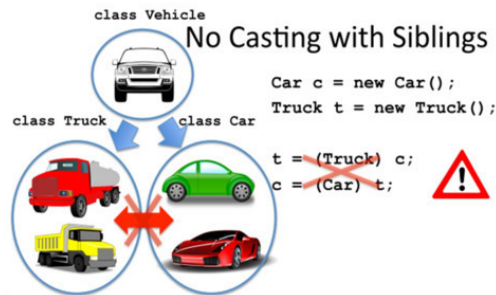


Мы можем это сделать, поскольку `car` является `vehicle`.

Однако вы не можете привести `v` в переменную `truck`.

Вы не можете сделать приведение между классами, полученными из одного класса.

Вы не можете превратить car в truck или truck в car.



У них разные поля и методы.

Преобразование применимо не только для классов.

Это также возможно с примитивными типами и между примитивными типами.

```
double d = (double) 3;      3.0
double e = 1.0 * 3;        3.0

int n = (int) 3.6;          3
int m = 1234567890; float f = m;
int dif = m - (int)f;       -46
```

Мы видели несколько примеров со строками и целыми числами.

Это особый случай, когда нет необходимости явного преобразования числа в строку, а можно сделать это используя оператор плюс.

При кастинге вверх мы не теряем информацию о числовом значении.

Поэтому мы можем делать это преобразование неявно.

Кастинг вниз более опасен, поскольку мы можем потерять информацию о числовом значении.

При преобразовании double в int мы получаем усеченное целочисленное значение, поэтому это преобразование нужно указывать явно.

Полиморфизм



Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 6

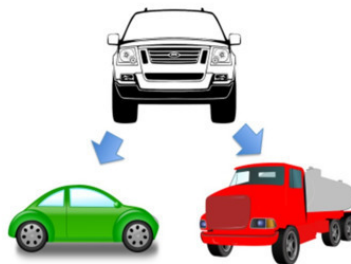
Полиморфизм

В объектно-ориентированном программировании мы организуем объекты в классы.

Объекты в одном классе имеют одинаковые поля, и одни и те же методы.

Можно сказать, что объекты в классе имеют одну и ту же форму, они могут просто отличаться значениями полей в определенном состоянии.

Когда мы ввели наследование, мы ввели семейства связанных классов.



Класс может наследовать поля и методы из базового класса и добавить дополнительные свои поля и методы.

Теперь мы хотим настроить возможности в классах такой иерархии.

Представьте, что мы хотим иметь одни и те же методы в базовом классе и в производном классе, но мы хотим сделать что-то другое в зависимости от класса, к которому принадлежит объект.

```
public class Vehicle{
    private String color;
    public Vehicle(String c) {color=c;}
    public String toString()
    {return "I am a vehicle";}
}
public class Car extends Vehicle{
    private int noPass;
    public Car(int n, String c)
    {super(c); noPass=n;}
    public String toString ()
    {return
    "I am a car. I am carrying "+noPass+" passengers";}
}
```



Здесь мы видим, что в методе `toString` подкласса `car` определено другое поведение, отличное от того, которое определено в суперклассе.

Поэтому поведение считается переопределенным.

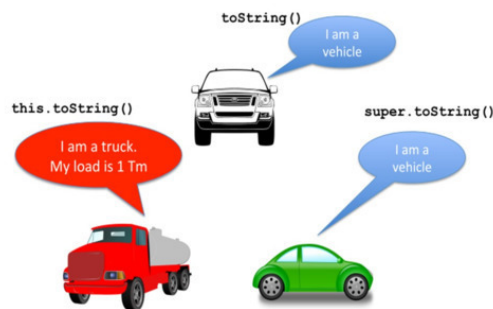
Этот же метод может делать что-то совершенно отличное от метода суперкласса, с тем же именем и теми же функциональными возможностями.

Таким образом, мы видим, что метод с тем же именем и одинаковой функциональностью может иметь разный код в разных классах иерархии.

Это называется переопределением.

Однако при необходимости можно вызвать метод суперкласса.

Для этого нам просто нужно вызвать метод с префиксом `super`.



Здесь также может использоваться ключевое слово `this`, чтобы обратиться к методу, который определен в соответствующем классе.

Это переопределение методов называется полиморфизмом.

Слово полиморфизм происходит от греческого, что означает многие формы.

И в контексте объектно-ориентированного программирования, полиморфизм позволяет нам иметь методы с одним и тем же именем, и одинаковой функциональностью, но разным поведением в группе классов, связанных отношением наследования.

Другими словами, полиморфизм позволяет использовать наследников, как родителей. При этом, если в классе-наследнике был переопределен какой-то метод, то вызовется он.

Переопределение и перегрузка



Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 7

Переопределение и перегрузка

Теперь давайте рассмотрим две концепции, которые выглядят взаимосвязанными, но на самом деле являются разными, это перегрузка и переопределение.

Обе эти концепции применяются к методам.

Ранее мы говорили о конструкторах.

Помните, что у нас был автомобиль с двумя полями, lights и color.

Constructors

No return type

```
Car() {lights=false; color="white";}

Car(String c) {lights=false; color=c;}
Car(boolean b) {lights=b; color="white";}

Car(boolean b, String c)
{lights=b; color=c;}
```

И мы определили в одном классе не один, а несколько конструкторов.

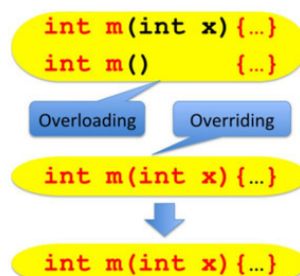
Имена этих конструкторов были одинаковыми, но параметры были разные.

И это важно, чтобы список параметров был другим.

Вы не можете определить два конструктора с одним и тем же именем, и одним и тем же списком параметров.

Фактически, Java понимает, какой конструктор вызвать, просматривая параметры.

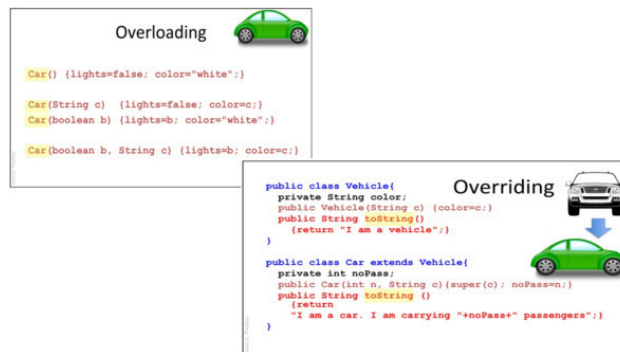
И то, что мы делали для конструкторов, также применимо для методов.



Мы говорим о перегрузке, когда у нас есть разные методы с тем же именем, но разным списком параметров.

С другой стороны, мы ввели переопределение, когда мы хотели изменить поведение метода, унаследованного от суперкласса.

В этом примере метод `toString` суперкласса переопределяется в подклассе с помощью метода с тем же именем, и теми же параметрами, и возвращаемым типом, но другим телом метода.



Важно, чтобы параметры и возвращаемый тип были одинаковыми.

Отличалось только тело метода.

И в пределах одного класса мы можем перегрузить метод.

В этом случае имя и возвращаемый тип совпадают, но список параметров будет другим.

Компилятор будет различать, какой вызывается метод, сравнивая списки параметров.

Неправильно пытаться перегрузить метод, просто изменив возвращаемый тип.

Если мы это сделаем, мы получим ошибку компилятора.

То же самое произойдет, если мы просто изменим имена параметров.

В этом случае определенный метод не изменится вообще.

И мы также получим ошибку компилятора.

Когда мы определяем метод, мы связываем идентификатор – имя метода – с некоторым кодом – телом метода.

```

public int sq(int x){
    return x*x;
}

```

sq **int → int**
 $x \rightarrow x * x$

```

sq(3);
sq(4);

```

Всякий раз, когда мы вызываем это имя метода с некоторыми значениями, мы знаем, какой код нужно выполнить.

Например, используя объявление метода, мы связываем идентификатор `sq` с методом, который отображает целые числа в целые числа, возводя число в квадрат.

Идентификатор `sq` всегда привязан к методу в соответствующей области кода.

Во многих языках, которые не являются объектно-ориентированными, эта привязка выполняется обычно во время компиляции.

Во время выполнения эта привязка зафиксирована.

И это называется «ранним» или «статическим» связыванием.

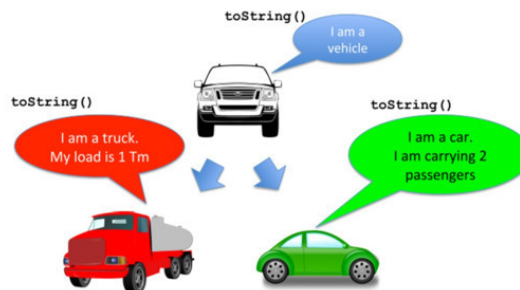
Но этот способ не соответствует концепции полиморфизма и переопределения методов в производных классах.

Здесь мы хотим точно противоположного – чтобы часть кода была не привязана статически к имени метода, а, чтобы зависела от объекта, вызванного во время выполнения.

Поведение, которое нам нужно, называется «динамическим» связыванием.

Поэтому нам нужно различать статическое или раннее связывание, которое выполняется во время компиляции, от динамического или позднего связывания, которое выполняется во время выполнения кода.

В отличие от переопределения перегруженные методы разрешаются во время компиляции.



При этом информация предоставляется классом.

Когда код программы доходит до имени метода, компилятор знает, какое тело метода выполнить – по крайней мере в случае перегруженных методов.

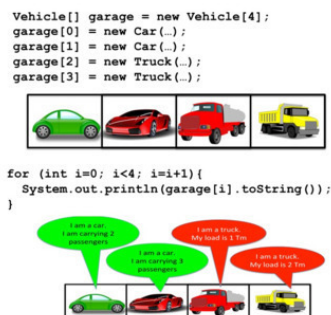
Но это не относится к переопределению.

Здесь разрешение имен выполняется во время выполнения программы.

Динамическое связывание используется для переопределенных методов.

Здесь информация задается объектом, а не классом.

Предположим, мы объявили массив транспортных средств под названием «гараж» для хранения четырех автомобилей.



И предположим также, что у нас есть автомобили и грузовики, которые стоят в разных позициях.

Теперь в цикле for мы применяем методы toString,
Которые мы определили ранее, ко всем элементам массива.
Что происходит?

Свой метод применяется к каждому из этих элементов.

Таким образом, мы можем иметь единую форму объектов, но разнообразие в том, что выполняется.

Возможно даже, в случае компиляции мы не знаем классов элементов массива.

Это будет считываться во время выполнения программы.

Поэтому динамическое связывание является необходимым поведением для переопределения метода.

Теперь посмотрим на другой пример.

Давайте теперь определим несколько перегруженных методов с именем p.

```
public void p(Vehicle v){
    System.out.println("I am a vehicle");
}
public void p(Car c){
    System.out.println("I am a car");
}
public void p(Truck t){
    System.out.println("I am a truck");
}

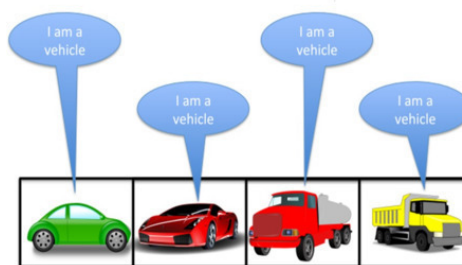
for (int i=0; i<4; i=i+1){
    p(garage[i]);
}
```



У них есть один параметр, который является объектом разных классов.

И теперь мы вызываем метод p для всех элементов этого массива.

Помните, что аргумент метода p – это vehicle в массиве vehicle.



Поскольку каждый элемент является vehicle, строка будет напечатана для vehicle, так как метод p привязывается к телу во время компиляции.

Помимо примера, который мы видели, private, final, и static методы также привязываются статически.

Кроме того, атрибуты всегда привязываются статически.

Возникает вопрос, почему все не привязывать динамически?

Имеет смысл связывать идентификаторы с данными или кодом во время компиляции по двум причинам.

Во-первых, чтобы выполнить первую проверку кода и выявить ошибки, а во-вторых, оптимизировать генерируемый код.

Вот почему эта стратегия используется чаще в языках программирования.

Однако это не работает, когда мы переопределяем метод.

Во время компиляции мы можем даже не знать, какой объект мы получим.

Тогда имеет смысл применить динамическое связывание.

Динамическое связывание также называется «поздним связыванием».

Первое приближение к классу выполняется во время компиляции, но нужный класс окончательно определяется во время выполнения.

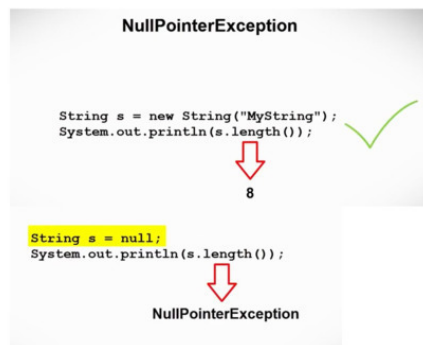
Теперь вернемся к исключениям, чтобы объяснить некоторые дополнительные исключения, которые вы должны знать и которые связаны с объектами и классами.

Небольшое напоминание, исключения – это события, которые происходят во время выполнения программы и которые нарушают нормальный поток выполнения инструкций программы.

Мы уже видели три исключения: `ArithmeticException`, `ArrayIndexOutOfBoundsException` и `NumberFormatException`.

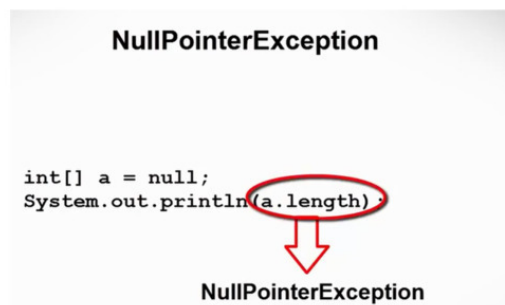
Следующее исключение, которое мы увидим, – это исключение `NullPointerException`.

Это исключение возникает при попытке программы использовать переменную, которая не имеет примитивного типа, и которая еще не была инициализирована.



Т. е. мы пытаемся использовать переменную, которая должна указывать на объект, который еще не был создан.

Представьте, что мы хотим напечатать длину массива, который мы объявили, но который мы еще не инициализировали.

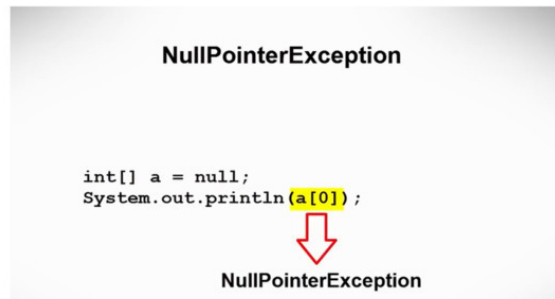


Тогда мы получим такое же исключение `NullPointerException`.

Имейте в виду, что «length» – это метод в случае класса `String`, но поле в случае массива.

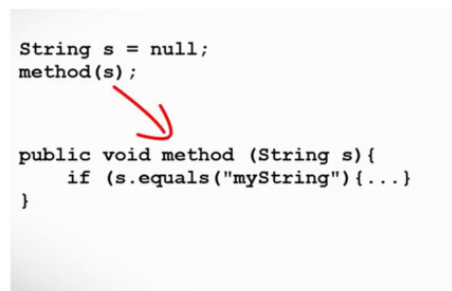
И, если мы попытаемся получить доступ к позиции в массиве, который не был инициализирован, программа будет генерировать исключение `NullPointerException`, а не исключение `ArrayIndexOutOfBoundsException`.

В этих примерах очень легко обнаружить, что мы пытаемся использовать переменную, которая не была инициализирована.



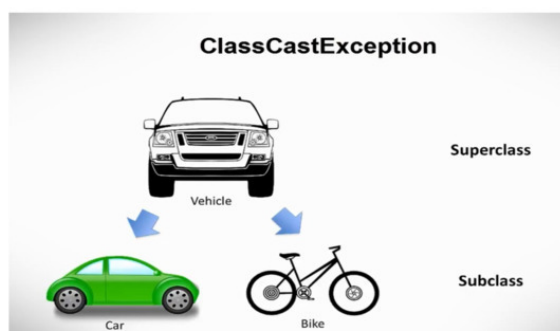
Однако, когда мы программируем, мы определяем методы, которые получают аргументы и которые вызываются из других объектов.

В этих случаях обнаружение переменных, которые не были инициализированы, не так очевидно.



Второе исключение, которое связано с объектами и классами, и которое мы увидим, это `ClassCastException`.

Чтобы проиллюстрировать это исключение, рассмотрим эту иерархию классов, где `Vehicle` является суперклассом, и `Car` и `Bike` – это подклассы.



Согласно этой иерархии, можно создать экземпляр класса Car и присвоить его переменной типа Vehicle, потому что Car также является Vehicle.

Это приведение правильное и позволяет нам воспользоваться свойством полиморфизма, сохраняя в одном массиве Vehicle набор объектов классов Car и Bike.

```
Vehicle v = new Car();

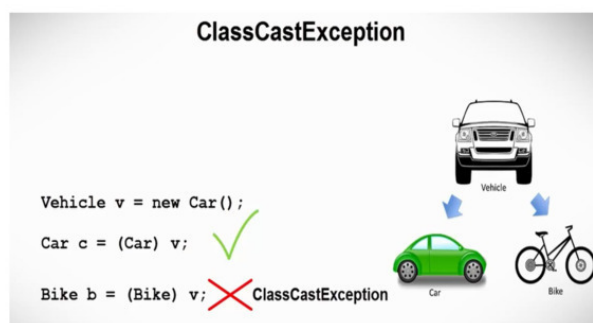
Vehicle[] arrayV = {new Car(), new Car(), new Bike()};

Vehicle v = new Car();
Car c = (Car) v; ✓
```

Позже в программе нам может понадобиться привести этот экземпляр к объекту класса Car.

Единственное условие, которое налагает Java, – это сделать это приведение явным.

Однако, если мы попытаемся применить этот экземпляр к объекту класса Bike, программа выбросит ClassCastException во время выполнения, потому что объект в переменной «v» не является байком.



Мы уже видели, как обрабатывать исключения, которые выбрасываются, когда в программах происходят определенные события, используя конструкцию «try-catch».

Однако мы также можем программировать методы, которые при определенных обстоятельствах должны выбрасывать исключения.

Чтобы явно выбросить исключение в методе, нам нужно использовать ключевое слово «throw» и создать экземпляр конкретного исключения, которое метод должен выбросить.

```
public void method (String s){  
    if (s == null){  
        throw new IllegalStateException("s is null");  
    }  
    ...  
}
```

Один и тот же метод может выбросить несколько исключений в зависимости от конкретных обстоятельств.

Примитивы и объекты



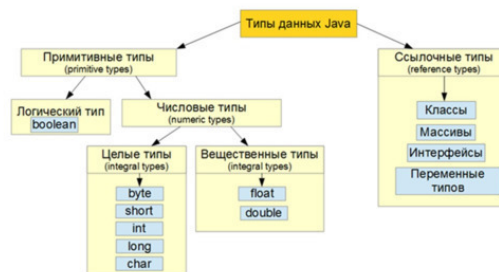
Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 8

Примитивы и объекты

Теперь в качестве обобщения.



В Java есть два общих типа данных: примитивы и объекты.

Примитив – это тип данных Java, которые считаются простейшей формой данных.

Данные этого типа хранятся непосредственно в памяти.

Это данные типа `int`, `char`, `double` и `boolean`.

И когда вы создаете новую переменную типа `int`, которая является примитивом, компьютер выделяет область в памяти с именем и значением этого `int` прямо там.

Поэтому всякий раз, когда вы передаете переменную в качестве параметра или копируете ее, вы копируете значение этой переменной.

Поэтому вы создаете совершенно новую версию этой переменной каждый раз, когда вы манипулируете ей.

Так как примитивы такие простые, мы можем выполнять с ними прямые математические операции, такие как сложение, вычитание, деление, и так далее.

Теперь, что такое объект?

Объектом является гораздо более сложный тип данных, потому что на самом деле это способ хранения нескольких фрагментов связанной информации и различных вещей, которые вы можете делать с этой информацией под одним типом данных.

Такие вещи, как `String`, `Array`, `Scanner` и `ArrayList` считаются объектами.

И все они начинаются с большой буквы в Java, чтобы обозначить их как объекты.

Когда вы создаете новую переменную типа объект, например, для массива, компьютер выделяет область памяти для ссылки на то, где этот код на самом деле собирается хранить эти данные.

Затем, когда вы передаете это значение в качестве параметра, вы передаете ссылку, а не фактические данные.

И это потому, что объекты намного больше примитивов, и постоянно копировать их очень затратно.

Поэтому вам всегда нужно понимать, когда вы копируете ссылку на объект или сами данные объекта.

Поскольку объекты сложнее примитивов, вы не можете выполнять такие вещи, как сложение и вычитание, как с простыми числами.

Но, поскольку объекты имеют свое поведение, вам просто нужно взглянуть на методы объекта, чтобы узнать, что вы можете с этим объектом сделать.

Например, если вы хотите узнать, сколько символов в строке, вы вызываете метод `length`. Каждый объект имеет свой собственный набор моделей поведения.

И есть одна вещь, о которой нужно знать.

Это специальное ключевое слово `null`.

`Null` – это просто слово, которое означает отсутствие объекта.

По сути, это значение 0 для объекта.

Точно так же, как 0 – это значение 0 для `int` или 0.0 – это значение 0 для `double`.

`Null` – это значение 0 для всех типов объектов.

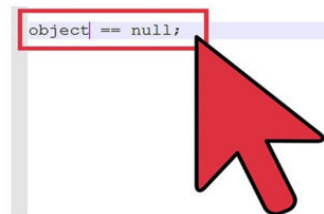
Предположим, мы создаем новый массив строк.

Если мы создадим новый массив символов, мы знаем, что он хранит значения нулей по умолчанию.

Но что он хранит в случае, когда мы создаем массив строк?

Это `Null`.

Это то, что автоматически заполняется в массив, что означает, объект может быть здесь, но его нет здесь и сейчас.



Это важно знать, потому что вы можете столкнуться с очень распространенным типом исключения `Null Pointer`.

Обычно это происходит, когда вы пытаетесь выполнить метод объекта, который является нулевым.



Например, мы хотим получить длину строки, которая хранится в этом массиве. Там нет строки, поэтому мы получаем так называемое исключение `NullPointerException`. Вы не можете назвать длину того, чего не существует.

Имейте в виду, что `null` означает объект, а не пустой объект.

Например, вы можете вызвать метод `length` для пустой `String`.

Это длина равна нулю.

Но нет такой длины, как длина того, чего не существует.

Просто важно знать, что `null` означает, что здесь нет объекта.

И нам нужно туда его поместить.

Теперь, когда мы понимаем, что такое примитив и что такое объект, важно понять, как компьютер рассматривает эти два типа переменных в своей собственной памяти.

Потому что это оказывает огромное влияние на то, как вы их программируете.

```
int x = 10;
```

	0	1	2	3
0				
1				
2				
3				

Представим себе, что это память компьютера.

На самом деле это похоже на то, как выглядит память компьютера.

Это общая сетка с адресами для каждого отдельного местоположения, очень похожая на массив.

Когда вы создаете новую переменную примитивного типа, компьютер занимает одно место в памяти и просто помещает эту информацию прямо там, имя и значение переменной.

Когда вы создаете новый объект, он является динамическим.

Он может расти и сокращаться, он может быть большим.

Поэтому компьютер должен придумать особый способ поддерживать этот объект в собственной памяти.

```
int x = 10;
int[] myArr = new int[4];
```

	0	1	2	3
0	x = 10	myArr: (0,1) - (3,1)		
1	myArr data			
2				
3				

Поэтому, при создании, например, нового массива, компьютер сначала находит место в своей памяти для хранения адреса, где он будет хранить этот массив.

И затем он занимает целую секцию памяти для какого-либо большого объекта.

И, теперь у нас есть массив, находящийся в памяти, где одна из ячеек хранит адрес, где находятся реальные данные.

Это и есть ссылка.

Таким образом существует большое различие между примитивами и объектами.

Примитивы хранятся непосредственно в памяти, как только вы создаете примитив.

Они настолько малы, что это имеет смысл.

Когда вы копируете переменную примитивного типа и меняете ее значение, первоначальное значение никак не меняется.

Между ними нет реальной связи.

Это будут две совершенно разные переменные.

Как вы можете себе представить, объекты функционируют по-другому.

Вместо того, чтобы выделять пространство для фактического значения, объекты занимают пространство в памяти для ссылки на место, где хранится информация объекта.



Поэтому, если я создаю новый массив, а затем создаю другой массив, и устанавливаю его равным первому массиву, что копируется?

Компьютер копирует ссылку.

Теперь у меня есть две переменные, которые указывают на одну и ту же информацию.

Поэтому, если я что-то изменяю в массиве z, изменится и массив y, и наоборот.

Вы просто скопировали адрес, где находится информация.

Поэтому, если я создам объект и передам его как параметр в метод, я передам ссылку или адрес.

И любые изменения, которые я сделаю в этом методе с объектом, будут отражены в первоначальном объекте.

Мне даже не нужно возвращать его в методе.

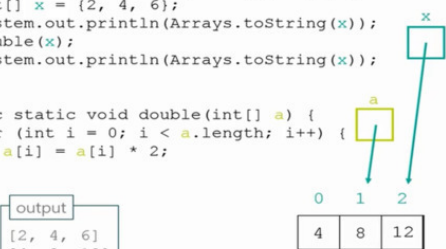
Как было сказано ранее, массивы – это объекты. Однако, у них нет полезных методов внутри объекта Array.

Для этого в Java есть класс Arrays,

который содержит набор статических вспомогательных методов для работы с числами, схожих с тем, как в классе Math есть набор статических вспомогательных методов для работы с числами.

```
public static void main(String[] args) {
    int[] x = {2, 4, 6};
    System.out.println(Arrays.toString(x));
    double(x);
    System.out.println(Arrays.toString(x));
}

public static void double(int[] a) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[i] * 2;
    }
}
```



Вот несколько популярных методов из класса Arrays.

<p>java.lang. Class Arrays</p> <p>java.lang.Object java.util.Arrays</p> <p>public class Arrays extends Object</p> <p>This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.</p> <p>The methods in this class all throw a NullPointerException if the specified array reference is null, except where noted.</p> <p>The documentation for the methods contained in this class includes brief descriptions of the implementations. Such descriptions should be regarded as implementation notes, rather than parts of the specification. Implementations should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by <code>sort(Object[])</code> does not have to be a MergeSort, but it does have to be stable.)</p> <p>This class is a member of the Java Collections Framework.</p> <p>Since: 1.2</p>							
<p>Method Summary</p> <table> <thead> <tr> <th>Modifier and Type</th><th>Method and Description</th></tr> </thead> <tbody> <tr> <td>static <T> List<T></td><td><code>asList(T... a)</code> Returns a list-view backed by the specified array.</td></tr> <tr> <td>static int</td><td><code>binarySearch(Object[] a, Object key)</code> Searches the specified array of objects for the specified value using the binary search algorithm.</td></tr> </tbody> </table>		Modifier and Type	Method and Description	static <T> List<T>	<code>asList(T... a)</code> Returns a list-view backed by the specified array.	static int	<code>binarySearch(Object[] a, Object key)</code> Searches the specified array of objects for the specified value using the binary search algorithm.
Modifier and Type	Method and Description						
static <T> List<T>	<code>asList(T... a)</code> Returns a list-view backed by the specified array.						
static int	<code>binarySearch(Object[] a, Object key)</code> Searches the specified array of objects for the specified value using the binary search algorithm.						

Метод `toString` возвращает строковое представление массива.

Метод `equals` определяет, одинаковы ли два массива.

Метод `fill` присваивает новое значение всем элементам массива.

Метод `sort` сортирует элементы.

Метод `binarySearch` выполняет поиск элемента по значению и возвращает индекс элемента в случае успеха, или отрицательное целое в случае, если такого элемента нет.

Для работы метода `binarySearch` необходимо, чтобы массив был уже отсортирован.

Method name	Description
<code>toString(arr)</code>	returns a string representing the array inside [], e.g. "[7, 33, 51, 14]"
<code>equals(arr1, arr2)</code>	returns <code>true</code> if the two arrays are equal, that is they contain the same elements in the same order
<code>fill(arr, val)</code>	sets every element in the array to have the specified value
<code>sort(arr)</code>	sorts the elements in the array into ascending order
<code>binarySearch(arr, val)</code>	returns the index of the given value in an array (< 0 if not found); the array must be sorted

```
import java.util.*;
```

Класс `Arrays` находится в пакете `java.util`, и если вы хотите его использовать, вы должны добавить строку `import java.util.*` в начало Java файла.

Давайте рассмотрим пример использования пары методов из класса `Arrays`.

```
//Return the median value of an array of numbers
//without changing the parameter array
public static double median(int[] numbers) {
    int[] tmp = Arrays.copyOf(numbers, numbers.length);
    Arrays.sort(tmp);
    int mid = tmp.length/2; // Note: int division
    if (tmp.length%2 == 0) { // even length?
        return (tmp[mid-1]+tmp[mid])/2.0; //float division
    } else {
        return tmp[mid];
    }
}
```

В этой задаче мы хотим вернуть медианное значение для множества чисел, где медиана – это среднее значение, когда числа отсортированы.

Для решения задачи, сначала мы создадим копию массива, т.о. мы не изменим оригинальный массив.

После создания копии, мы отсортируем массив. Потом мы просто сможем получить медианное значение, которое представляет собой просто средний элемент массива нечетной длины, или арифметическое среднее двух средних элементов массива четной длины.

Вот наш метод `median`, который принимает массив целых чисел в качестве аргумента, и возвращает значение типа `double`.

Мы возвращаем тип `double`, т.к. у нас может быть усреднение двух целых чисел.

Метод начинается с создания копии массива-аргумента вызовом метода `copyOf` класса `Arrays`.

Этот метод создаст копию массива с количеством элементов, которое указано вторым аргументом.

В данном случае, мы создаем полную копию массива `numbers`.

После того, как копия сделана, мы сортируем ее, вызывая метод `Arrays.sort`.

Мы находим средний элемент массива, используя целочисленное деление, и затем определяем, четная ли длина у массива или нечетная.

Если длина четная, мы возвращаем среднее значение двух центральных элементов.

В этом случае, мы делим на 2.0, чтобы получить число с плавающей запятой.

Если длина нечетная, мы просто возвращаем центральный элемент отсортированного массива.

В заключение, давайте коротко обсудим массивы объектов.

Как упоминалось ранее, когда массив создан, его элементы инициализируются нулем 0 такого же типа, что и базовый тип массива.

Для массивов объектных типов, значение при инициализации – это специальное значение `null`.

Значение `null` просто означает, что там не пока объекта.

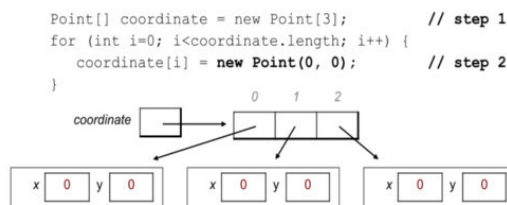
```
Point[] coordinate = new Point[3];
```



Например, если мы создадим массив `coordinate`, это массив трех элементов типа `Point`. Все три элемента будут проинициализированы значением `null`.

Перед тем, как пользоваться этим массивом, нам нужно заменить все значения `null` реальными объектами `Point`.

Т.о. массивы объектного типа требуют инициализации в два этапа.



На первом этапе, вы создаете объект массива, а на втором этапе, вы создаете объект базового типа для каждого элемента массива.

В образце кода, первым шагом является создание массива `coordinate`.

Затем, мы выполняем второй шаг с помощью цикла, в котором создается реальный объект класса `Point` для элемента 0, 1 и 2.

Массивы – полезный инструмент.

Однако они имеют некоторые ограничения.

Когда вы сначала создаете массив, вам нужно выбрать его размер.

И как только вы выберете размер массива, его нельзя изменить.

Это усложняет ситуацию, если у вас есть динамический набор информации, входящий и выходящий из вашей структуры данных.

Что, если вы не знаете, сколько всего будет элементов в конце концов?

Кроме того, если вы захотите, скажем, вставить что-то в середину массива, вы должны освободить место для этого.

Это означает, что вы должны сдвинуть все остальные элементы дальше по массиву.

Было бы неплохо, если бы существовала структура данных, которая обеспечивала бы легкий доступ и организацию массива, но при этом предоставляла бы всю гибкость, которая вам нужна.

Такая структура данных в Java есть и это список.

```

java.lang
Interface List<E>

Type Parameters:
  E - the type of elements in this list

All Superinterfaces:
  Collection<E>, Iterable<E>

All Known Implementing Classes:
  AbstractList, AbstractSequentialList, ArrayList, Arrlist, AttributeList, CopyOnWriteArrayList, LinkedList, MutableList, ReadOnlyList, Stack, Vector

public interface List<E>
  extends Collection<E>

  An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

  Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements e1 and e2 such that e1.equals(e2), and they typically allow multiple null elements if they allow null elements at all. It is not uncommon that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

  The List interface places additional stipulations, beyond those specified in the Collection interface, on the contracts of the iterator, add, remove, equals, and hashCode methods. Declarations for other inherited methods are also included here for convenience.

```

Список представляет собой упорядоченную последовательность элементов, как и массив. При этом, он добавляет функциональность, позволяющую ему расти и уменьшаться и вставлять элемент в середину без необходимости делать какие-либо изменения.

Также вы можете удалить элемент внутри списка.

Первый тип списка, так как в Java существует много типов списков, это ArrayList.

ArrayList хранит информацию в массиве, но при этом предоставляет дополнительную функциональность списка.

Вот несколько сравнений использования ArrayList и простого массива.

```

java.lang
Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:
  Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:
  Arrlist, MutableList, ReadOnlyList

public class ArrayList<E>
  extends AbstractList<E>
  implements List<E>, RandomAccess, Cloneable, Serializable

  Reusable array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

  The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding an element requires only time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementations.

  Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added,

```

```

String[] names = new String[5];
ArrayList<String> names = new ArrayList<String>();

names[0] = "Jessica";
names.add("Jessica");

String s = names[0];
String s = names.get(0);

```

С массивом вы начнете с типа и затем набор скобок, а затем его размер.

С ArrayList, вам просто нужно знать, какой тип информации вы собираетесь хранить в нем, а затем вы создаете новый ArrayList.

И он будет расти и сокращаться по мере необходимости.

Не нужно передавать его длину.

Чтобы добавить значение в массив вы должны найти в нем место и добавить в это место значение.

В ArrayList вы можете просто сказать add и затем добавить все, что захотите, в ArrayList. Он сам знает, где находится свободное пространство.

Вы также можете получить элемент, как и массив, используя индекс.

ArrayList поддерживает индексы для каждого из элементов, как и массив.

В ArrayList вы должны передать тип информации, которую он собирается хранить, в качестве параметра.

И это отлично подходит для объектов.

Но как насчет примитивов?

К сожалению, вы не можете просто создать ArrayList из, например, int.

Поэтому вам нужно использовать так называемый класс-оболочку, который является простым классом, хранящим только int внутри него.

Это класс Integer.

То же самое существует для double и char.

ArrayList поставляется с огромным набором методов, чтобы сделать жизнь проще.

```
ArrayList<DataType> name = new ArrayList<DataType>();
```

```
int => Integer    char => Character
double => Double  boolean => Boolean
```

add(value)	myList.add("hello")
add(index, value)	myList.add(0, "hello")
clear()	myList.clear()
indexOf(value)	myList.indexOf("hello")
get(index)	myList.get(0)
remove(index)	myList.remove(0)
set(index, value)	myList.set(0, "goodbye")
size()	myList.size()
toString()	myList.toString()

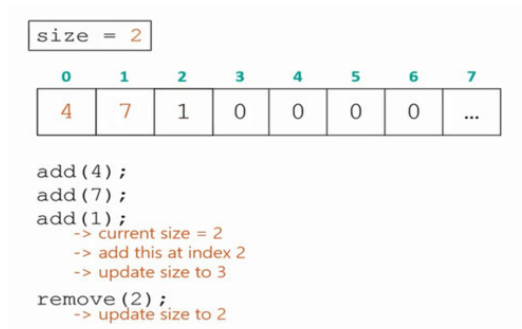
Вы не только можете добавить элемент в самом конце, но вы также можете добавить элемент по определенному индексу.

Вы можете очистить массив, вы можете выполнить поиск по массиву.

Например, вы ищете конкретное слово, но вы не знаете, в каком индексе оно находится. Это метод indexOf.

Вы также можете удалить и установить определенный индекс.

По сути, ArrayList это массив внутри класса, который имеет большой размер $2^{32}-1$, так что вы не сможете использовать всю длину массива.



ArrayList имеет переменную размера, которую он всегда поддерживает. Вы добавляете элемент в массив и удаляете, при этом изменяется переменная размера.

Абстракция



Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 9

Абстракция

Абстракция в объектно-ориентированном программировании помогает скрыть сложные детали объекта.

Абстракция является одним из ключевых принципов OOAD (объектно-ориентированный анализ и дизайн).

И абстракция достигается композицией (разделением) и агрегацией (объединением). Например, автомобиль оснащен двигателем, колесами и многими другими деталями.

```
public class Car {  
    int price;  
    String name;  
    String color;  
  
    int engineCapacity;  
    int engineHorsePower;  
  
    String wheelName;  
    int wheelPrice;  
  
    void move(){  
        //move forward  
    }  
    void rotate(){  
        //Wheels method  
    }  
    void internalCombustion(){  
        //Engine Method  
    }  
}
```

И мы можем записать все свойства автомобиля, двигателя и колеса в одном классе.

В этом примере атрибуты колеса и двигателя добавляются к типу автомобиля.

При программировании это не вызовет каких-либо проблем.

Но когда дело доходит до поддержки приложения, это становится более сложным.

Теперь, используя абстракцию, мы можем разделить вещи, которые можно сгруппировать в другом типе.

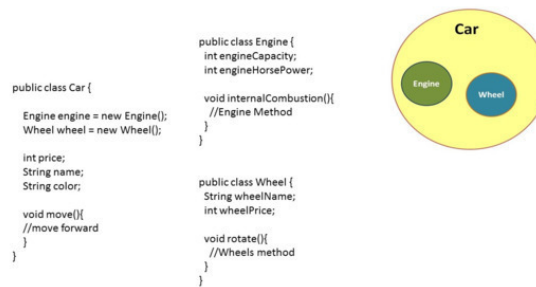
Часто изменяющиеся свойства и методы можно сгруппировать в отдельный тип, чтобы основной тип не нуждался в изменении.

Это добавляет силы принципу OOAD – «Код должен быть открыт для расширения, но закрыт для модификации».

И это упрощает представление модели.

Применяя абстракцию с композицией (разделением) и агрегацией (объединением), приведенный пример может быть изменен следующим образом.

Вы можете видеть, что атрибуты и методы, связанные с двигателем и колесом, перемещаются в соответствующие классы.



Двигатель и колесо относятся к типу автомобиля.

Когда создается экземпляр автомобиля, оба – двигатель и колесо будут доступны для автомобиля, а когда будут изменения этих типов (двигателя и колеса), изменения будут ограничиваться только этими классами и не будут влиять на класс Car.

Абстракция известна как отношение Has-A, например, у студента есть имя, у ученика есть ручка, у машины есть двигатель, то есть у каждого есть отношение Has-A.

И используя это отношение, мы разделяем на части, а затем одна часть может использовать другие части в виде объектов.

Абстракция является одним из основополагающих принципов языков объектно-ориентированного программирования.

И абстракция помогает снизить сложность, а также улучшает поддерживаемость системы.

В сочетании с концепциями инкапсуляции и полиморфизма абстракция дает больше возможностей объектно-ориентированным языкам программирования.

Абстракция – это принцип обобщения.

Абстракция принимает множество конкретных экземпляров объектов и извлекает их общую информацию и функции для создания единой обобщенной концепции, которая может использоваться для описания всех конкретных экземпляров как одно.

При этом мы переходим от конкретного экземпляра к более обобщенному понятию, думая о самой базовой информации и функции объекта.

Тем самым абстракция помогает скрыть сложные детали объекта.

Например, когда вы используете электронную почту, сложные детали того, что происходит, когда вы отправляете электронное письмо, например, протокол, используемый вашим почтовым сервером, скрыт от пользователя.

Поэтому, чтобы отправить электронное письмо, вам просто нужно ввести текст, указать адрес получателя и нажать «Отправить».

Аналогично в объектно-ориентированном программировании абстракция – это процесс скрытия деталей реализации от пользователя, и пользователю предоставляется только функциональность.

Другими словами, пользователь будет иметь информацию о том, что делает объект, а не о том, как он это делает.

И в Java это свойство абстракции реализуется с использованием абстрактных классов и интерфейсов, которые мы рассмотрим на следующей лекции.

Интерфейсы. Абстрактные методы и классы



Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

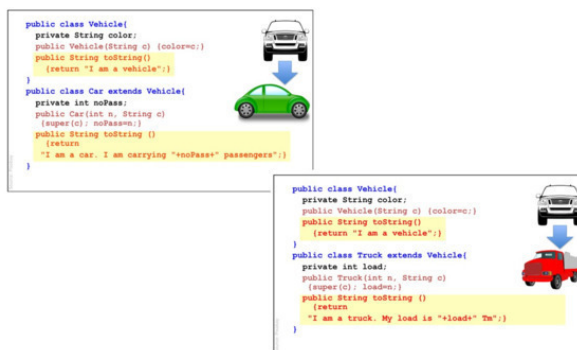
Лекция 10

Интерфейсы. Абстрактные методы и классы

Ранее мы определяли метод в одном классе и переопределяли этот метод в производных классах.

Таким способом можно делать разные вещи в зависимости от класса.

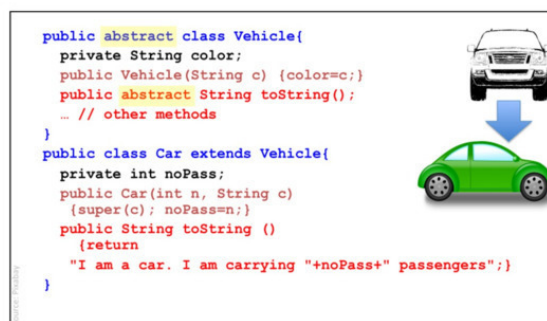
Здесь мы видим разные строки, возвращаемые методом toString.



Так как класс Vehicle является общим для этих классов, нам может вообще не понадобится строка, которая возвращается его методом toString.

В этом случае мы можем вообще не определять тело для метода toString в классе Vehicle.

Мы можем это сделать, и метод без тела называется «абстрактным».



Абстрактные методы обозначаются с помощью ключевого слова «абстрактный» в определении.

И класс с хотя бы одним абстрактным методом называется «абстрактным классом».

Здесь мы видим ключевое слово **абстрактный** в определении абстрактного класса.

Таким образом, абстрактный метод – это метод без тела.

Конструкторы, статические методы и финальные методы не могут быть абстрактными.

Абстрактный класс – это класс, в котором некоторые методы абстрактные, а некоторые – нет.

Абстрактный класс – это незавершенный класс, и мы не можем создать его объекты.

Чтобы получить объект, мы сначала должны определить производный класс, где тела определены для всех абстрактных методов.

Абстрактный класс может быть расширен до класса, или до другого абстрактного класса.

Кроме того, вы можете определить класс как абстрактный даже без абстрактного метода.

Это допускается, и вы можете это сделать для предотвращения возможности создания экземпляра класса.

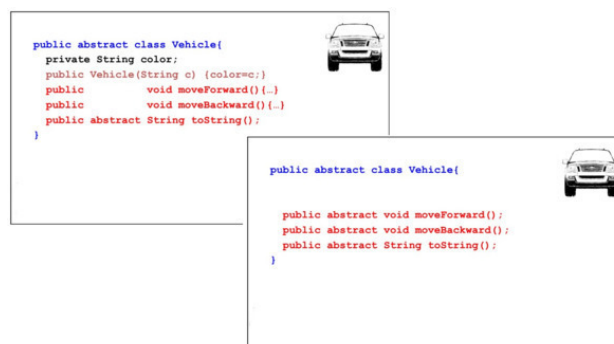
Однако, если есть абстрактный метод, вы получите ошибку, если вы не добавите ключевое слово **«abstract»** в определение класса.

Таким образом, абстрактные методы помогают разделить определение метода от поведения метода.

А абстрактные классы – это незавершенные классы, которые содержат абстрактные методы.

В абстрактном классе могут быть и абстрактные методы, а также обычные методы.

Теперь вопрос в том, что, если мы сделаем все методы абстрактными.



Но класс со всеми абстрактными методами уже не является абстрактным классом.

Это нечто другое.

Если все методы абстрактны, мы называем это интерфейсом.

Обратите внимание, что во всех методах нет тел.

```

public interface VehicleIF {

    public void moveForward();
    public void moveBackward();
    public String toString();
}

```

Здесь мы не пишем ключевое слово `abstract` для методов, но здесь нет путаницы, поскольку все методы в интерфейсе абстрактные.

Кроме того, мы объявляем методы публичными, но нам не нужно писать это явно.

```
public interface VehicleIF {  
  
    void moveForward();  
    void moveBackward();  
    String toString();  
}
```

Таким образом, все методы автоматически объявляются публичными, даже если мы не укажем ключевое слово `public`.

На самом деле не совсем верно, что все методы в интерфейсе должны быть абстрактными.

В Java 8 могут быть методы с телом, но они должны быть статическими методами или методами по умолчанию.

Но мы не будем усложнять, чтобы подчеркнуть концепцию интерфейса в его самой чистой форме.

Итак, для вас, в интерфейсе, все методы абстрактны.

Но что насчет полей?

В интерфейсе могут быть поля.

Но все они автоматически статические и финальные.

То есть, они являются константами.

Они также автоматически публичные, поэтому ключевое слово `public` не требуется явно указывать.

Таким образом, концепция интерфейса – это полезная абстракция.

Тогда как абстрактный класс реализует абстракцию, показывая некоторую общую функциональность для семейства классов без ее конкретной реализации, интерфейс, например, как физический интерфейс в радио или музыкальном проигрывателе, демонстрирует сервис снаружи и скрывает реализацию, которую мы можем определить.

То есть, в случае интерфейса, абстракция скрывает реализацию объекта от пользователя и предоставляет только интерфейс.

На самом деле мы можем изменить эту реализацию без изменения интерфейса, и, следовательно, предоставляемых нами услуг.

Интерфейсы обеспечивают уровень абстракции.

Можно использовать предоставленные методы без знания того, как они реализованы.

Но в какой-то момент эти методы должны быть реализованы.

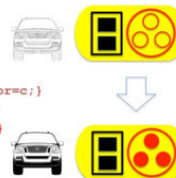
Представьте, что у нас есть интерфейс под названием `VehicleIF`.

```
public interface VehicleIF {
    int SOS_PHONE = 112;

    void moveForward();
    void moveBackward();
    String toString();
}
```

И в этом интерфейсе указан ряд методов.
Помните, что они публичные.

```
public class Vehicle
implements VehicleIF {
    private String color;
    public Vehicle(String c) {color=c;}
    public void moveForward() {...}
    public void moveBackward() {...}
    public String toString() {...}
}
```



Мы могли бы реализовать класс для этого интерфейса и назвать его Vehicle.

Обратите внимание, что теперь используется ключевое слово `implements` вместо ключевого слова `extends`.

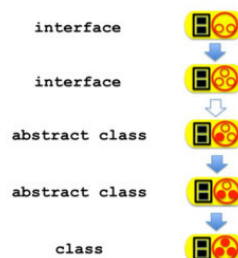
Для реализации интерфейса подразумевается определение класса, где для всех методов, дается реализация.

Мы можем также дополнительно определить поля и конструкторы, а также другие методы, возможно приватные.

Класс следует спецификации, заданной интерфейсом, и добавляет конкретные детали о том, как эти методы могут работать.

Однако нам не нужно идти от спецификации интерфейса к реализации класса за один шаг.

Мы могли бы также действовать поэтапно.



Путь от интерфейса к классу, который может быть создан, может быть короче или длиннее.

Во-первых, мы можем расширить один интерфейс от другого интерфейса, например, путем добавления абстрактных методов.

Мы можем частично реализовать интерфейс, путем реализации некоторых методов.

В этом случае мы получаем в результате не класс, а абстрактный класс, потому что не все методы реализованы.

И, наконец, мы можем перейти непосредственно от интерфейса к классу, путем реализации всех методов.

Если у нас есть абстрактный класс, мы можем расширить его другим абстрактным классом, например, если мы реализуем некоторые абстрактные методы, но не все из них.

Интерфейсы помогают нам моделировать системы, которые позволяют нам повторно использовать не только просто код, но и целиком концепции.

Теперь важно то, что класс может реализовать не только один, но и несколько интерфейсов.

В этом случае класс должен реализовать все методы от всех интерфейсов.

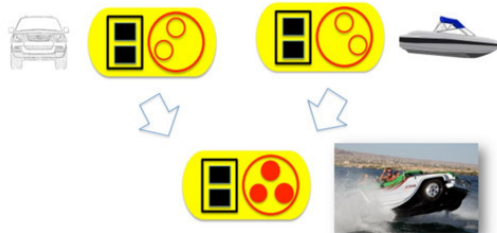
Помните, что класс не может расширять несколько классов.

В Java нет множественного наследования, как в других языках программирования, таких как C++.

Класс не может расширять два класса.

Однако в Java класс может реализовать два интерфейса.

Это способ сказать, что А является В и С.



Например, мы можем определить амфибию как реализацию интерфейса автомобиля и интерфейса лодки.

Этот амфибия-автомобиль будет реализовывать методы обоих интерфейсов.

Теперь, при этом, могут возникнуть конфликты имен.

Что произойдет, если у нас есть одно и тоже имя метода в обоих интерфейсах?

Если у нас есть одинаковое имя метода в обоих интерфейсах, но разные возвращаемые типы, тогда возникает ошибка.

Если имя метода и тип возвращаемого значения совпадают, тогда все в порядке.

Если, кроме того, совпадают и параметры методов, тогда класс должен реализовать этот метод один раз.

Они неразличимы.

Если типы параметров методов не совпадают, мы имеем случай перегрузки, который обрабатывается таким же образом, как будто эти методы принадлежат одному интерфейсу.

Оба эти метода должны быть реализованы.

Мы можем также определить класс путем реализации интерфейса и наследования от класса одновременно.

Также нужно сказать, что помимо абстрактных классов и интерфейсов для структурирования кода используются классы-утилиты, в которых определены только статические члены.

Как правило, такие классы используются для объединения родственных алгоритмов.

Примером такого класса может служить класс `Math`, предоставляющий реализации различных математических функций.

Таким образом, отношения реализации и наследования помогают нам определить структуры связанных интерфейсов и классов, которые помогают нам масштабировать и упорядочить код.

Проектирование интерфейсов всегда было непростой задачей, потому что, если мы хотим добавить дополнительные методы в интерфейсы, это потребует изменений во всех классах реализации.

По мере старения интерфейса количество реализующих его классов может увеличиться настолько, что будет невозможно расширить интерфейс, изменяя все классы реализации.

Вот почему большинство библиотек сначала обеспечивают базовый класс реализации, а затем они расширяют его и переопределяют его методы, чтобы при изменении интерфейса, изменить только базовый класс реализации.

В Java 8 вводится понятие метода по умолчанию интерфейса.

```
public interface Interface1 {
    void method1(String str);
    default void log(String str){
        System.out.println("I1 logging:"+str);
    }
}

public interface Interface2 {
    void method2();
    default void log(String str){
        System.out.println("I2 logging:"+str);
    }
}

public class MyClass implements Interface1, Interface2 {
    @Override
    public void method2() {
    }
    @Override
    public void method1(String str) {
    }
    @Override
    public void log(String str){
        System.out.println("MyClass logging:"+str);
        Interface1.print("abc");
    }
}
```

Для создания метода по умолчанию в интерфейсе нам нужно использовать ключевое слово «`default`» в сигнатуре метода.

Теперь, когда класс реализует интерфейс, необязательно предоставлять реализацию для методов по умолчанию интерфейса.

Эта функция помогает, помимо подхода с базовым классом реализации, с расширением интерфейсов с помощью дополнительных методов, и все, что нам здесь нужно, это обеспечить реализацию по умолчанию.

Мы знаем, что Java не позволяет нам расширять несколько классов, потому что это приведет к проблеме, когда компилятор не может решить, какой метод суперкласса использовать.

При использовании методов по умолчанию эта же проблема возникает и для интерфейсов.

Так как, если класс реализует интерфейс 1 и интерфейс 2 и не реализует общий метод по умолчанию, компилятор не может решить, какой из них выбрать.

Поэтому, обязательным является обеспечение реализации общих методов интерфейсов по умолчанию.

И поэтому, если класс реализует оба вышеупомянутых интерфейса, он должен будет обеспечить реализацию для метода `log`, иначе компилятор будет выбрасывать ошибку времени компиляции.

Таким образом, методы по умолчанию интерфейса Java помогают расширить интерфейсы, не опасаясь сломать классы реализации.

И методы по умолчанию интерфейса стирают различия между интерфейсами и абстрактными классами.

Если какой-либо класс в иерархии имеет метод с той же сигнатурой, метод по умолчанию теряет смысл для внедрения, чтобы избежать путаницы.

Теперь, статический метод интерфейса похож на метод по умолчанию, за исключением того, что мы не можем переопределить его в классах реализации.

```
public interface MyData {
    default void print(String str) {
        if (!isNull(str))
            System.out.println("MyData Print::" + str);
    }
    static boolean isNull(String str) {
        System.out.println("Interface Null Check");
        return str == null ? true : "".equals(str) ? true : false;
    }
}

boolean result = MyData.isNull("abc");
```

Эта функция помогает избежать нежелательных результатов, связанных с плохой реализацией в классах реализации.

Статический метод интерфейса виден только для методов интерфейса, однако, как и другие статические методы, мы можем использовать статические методы интерфейса, используя его имя.

Статические методы интерфейса хороши для предоставления вспомогательных методов, не требующих реализации в классах.

И можно использовать статические методы интерфейса Java для удаления классов-утилит, и переместить все статические методы классов-утилит в соответствующий интерфейс, который будет легко найти и использовать.

Однако мы не можем определить статический метод интерфейса для методов класса Object, мы получим ошибку компилятора.

Это связано с тем, что Object является базовым классом для всех классов.

В Java 9 вводятся приватные методы и приватные статические методы в интерфейсах.

```
public interface MyInterface {
    default void defaultMethod 1() {
        privateMethod("Hello from the default method 1!");
    }
    default void defaultMethod 2() {
        privateMethod("Hello from the default method 2!");
    }
    private void privateMethod(final String string) {
        System.out.println(string);
    }
    void normalMethod();
}

interface Example {
    static void doJob1(String arg) {
        verifyArg(arg);
    }
    static void doJob2(String arg) {
        verifyArg(arg);
    }
    private static void verifyArg(String arg) {
    }
}
```

Чтобы был выбор, какие выставлять клиентам методы реализации.

Соответственно приватные методы интерфейса предназначены для использования в методах по умолчанию интерфейса, а приватные статические методы интерфейса предназначены для использования в статических методах интерфейса.

И приватные методы должны иметь реализацию, они не могут быть абстрактными.

И мы не можем получить доступ или наследовать приватные методы от интерфейса к другому интерфейсу или классу.

Пакеты



Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 11

Пакеты

Давайте рассмотрим концепцию пакета, которая позволяет программистам лучше структурировать свои программы, что облегчает их понимание и управление.

В большом приложении классов создается тысячи и десятки тысяч.

Поэтому возникает вопрос: Если классов много, их все в одном каталоге держать? И как потом с ними разбираться?

Конечно же необходим некий механизм упорядочивания.

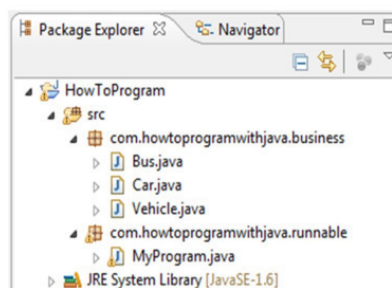
И такой механизм создан.

Причем достаточно простой и очевидный – каталоги.

Мы уже привыкли, что на диске наши файлы лежат в разных каталогах, которые мы сами организовываем в удобном порядке.

В Java сделано тоже самое – физически класс кладется в определенный каталог файловой системы, представляющий собой пакет.

Существует даже некоторые правила именования этих каталогов или пакетов.



Например, для коммерческих проектов каталог должен начинаться сначала с префикса «com» а за ним следует название компании или доменное имя компании – например «myscompany».

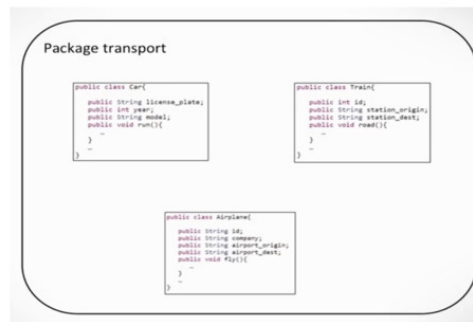
Далее следует название проекта.

Потом уже идет более точное разделение по какому-либо признаку – чаще всего функциональному.

Пакет в Java представляет собой группу связанных классов и интерфейсов, которые имеют схожие свойства.

Это абстрактная концепция, и это ответственность программиста, чтобы правильно организовать различные классы в пакеты.

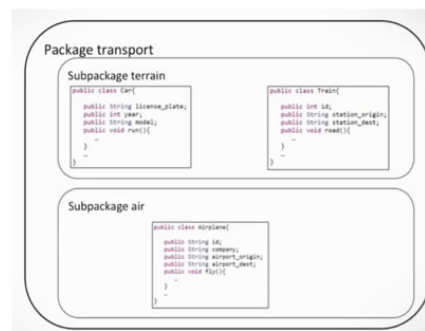
Обычно классы, сгруппированные в один и тот же пакет, имеют сходную семантику. Например, представьте, что у вас есть класс Car.



И он может быть частью пакета с именем transport, который также может содержать другие классы, такие как самолет или поезд.

Пакеты можно подразделить на подпакеты в зависимости от степени абстракции, которую хочет программист, формирующий иерархию пакетов.

Таким образом, наш предыдущий пакет транспорт мог быть разделен на подпакеты наземного транспорта и воздушного транспорта для представления физической среды транспорта.



Подумайте о том, какие классы будут храниться в каждом подпакете.

Когда вы создаете новую программу, очень полезно организовать различные классы и интерфейсы в пакеты, чтобы упростить ваш проект.

Таким образом, вы упрощаете использование классов и интерфейсов.

Далее, мы рассмотрим стандартную библиотеку Java, которая хорошо структурирована на пакеты и подпакеты.

Хорошо, но когда мы пишем новый класс, как мы можем определить, какой класс принадлежит к какому пакету?

Это очень просто.

В верхней части исходного кода класса вы добавляете слово package, за которым следует имя пакета.

Помните, что определение пакета должно быть первым выражением в исходном файле.

Имя пакета определяется программистом.

Это имя должно быть в нижнем регистре, чтобы избежать конфликтов с именами классов и интерфейсов, и не может быть одним из слов, зарезервированных Java, таким как main, for или string.

И подпакеты задаются с использованием символа точки.

Таким образом, для создания пакета, нам нужно в файловой системе создать каталоги и подкаталоги, например, каталог `transport` и подкаталог `air`.

```
package transport.air;

public class Airplane{
    ...
}
```

Затем поместить в них файлы классов и интерфейсов.

И затем указать в каждом классе и интерфейсе вверху директиву `package` с именами каталогов и подкаталогов, разделенными точками, то есть путем где находится файл класса или интерфейса.

При написании новых программ вам может потребоваться доступ к некоторым классам и интерфейсам из определенного пакета.

В этом случае вы должны заранее импортировать такие классы.

Этот импорт должен быть размещен сверху исходного кода класса, сразу после объявления пакета класса, используя слово импорт.

```
import transport.air.Airplain;

import transport.Car;

import transport.air.*;
```

За оператором импорта должен следовать весь путь пакета, вместе с классом, который вы хотите импортировать.

И это будет полное квалифицированное имя класса – имя класса вместе с именем его пакета.

Если вы хотите импортировать все классы в пакете, вы можете использовать символ звездочки.

Таким образом, вы сможете получить доступ ко всем публичным полям и методам этих классов.

Полное имя класса – весьма важный момент.

Разделение классов по пакетам служит не только для удобства, но решает еще одну важную задачу – уникальность имен классов.

Наверняка в большом проекте будет участвовать много людей и каждый будет писать свои классы.

И наверняка имена этих классов нередко будут одинаковые.

И скорее всего вы будете подключать сторонние библиотеки и в них будут классы, которые будут называться так же как ваши.

Единственным спасением различать их – это поместить в разные пакеты.

Таким образом, как правило, программа состоит из нескольких пакетов.

И каждый пакет имеет собственное пространство имен для классов и интерфейсов, объявленных в пакете.

И пакеты образуют иерархическую структуру имен.

При этом полные имена классов и интерфейсов, то есть их имена с учетом пакетов, должны быть уникальными.

И для доступа из одного пакета к другим пакетам используется ключевое слово `import`.

Также пакеты могут быть безымянными.

Классы и интерфейсы безымянного пакета не содержат объявления пакета.

И безымянные пакеты следует использовать только в небольших тестовых программах.

Также в Java можно использовать статический импорт для доступа к статическим методам и полям класса.

```
Math.sqrt(Math.pow(side1, 2) + Math.pow(side2, 2));
```

```
import static java.lang.Math.sqrt;  
import static java.lang.Math.pow;
```

```
sqrt(Math.pow(side1, 2) + pow(side2, 2));
```

```
import static java.lang.Math.*;
```

Например, в этом выражении, методы `pow` и `sqrt` являются статическими, поэтому они должны быть вызваны с указанием имени их класса – `Math`.

И это приводит к достаточно громоздкому коду.

Этих неудобств можно избежать, если воспользоваться статическим импортом.

При этом имена методов `sqrt` и `pow` становятся видимыми благодаря оператору статического импорта.

Также, с помощью звездочки, можно импортировать все остальные статические члены класса `Math`, не указывая их по одному.

Каким бы удобным ни был статический импорт, очень важно не злоупотреблять им, чтобы избежать конфликта имен, например, если вы определите в своем классе свой метод `pow`.

Теперь, когда мы узнали, что классы группируются в пакеты, и мы знаем, что методы и поля класса могут быть публичными и приватными, пора сказать, что методы и поля класса также могут быть защищенными, это ключевое слово `protected`, и приватными в пакете, это отсутствие всякого ключевого слова.

Доступ к членам класса				
	Private	Модификатор отсутствует	Protected	Public
Один и тот же класс	Да	Да	Да	Да
Подкласс класса этого же пакета	Нет	Да	Да	Да
Класс этого же пакета, не являющийся подклассом	Нет	Да	Да	Да
Подкласс класса другого пакета	Нет	Нет	Да	Да
Класс другого пакета, не являющийся подклассом класса данного пакета	Нет	Нет	Нет	Да

Публичный член класса виден везде без ограничений.

Защищенный член класса, `protected`, виден в пределах своего пакета, а также подклассом класса, даже если подкласс принадлежит другому пакету.

Если нет никакого ключевого слова, член класса виден только в пределах своего пакета.

И приватный член класса виден только в пределах своего класса.

Абстрактные классы vs Интерфейсы



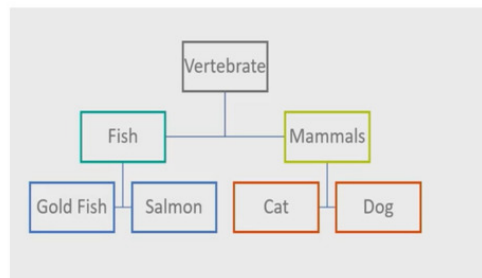
Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 12

Абстрактные классы vs Интерфейсы

Теперь давайте вернемся немного назад и рассмотрим наследование разных объектов. Предположим у нас есть эта иерархия.



Вверху иерархии у вас есть что-то более общее, например, позвоночные, а затем, когда вы спускаетесь вниз, все становится более конкретным.

Таким образом, в самом низу у вас есть наиболее специфический уровень.

В некоторых деревьях наследования, поскольку все становится более общим, объекты как бы перестают восприниматься как реальные экземпляры.

Как что такое объект рыбы?

Существует много разных видов рыб, и это слишком общее, чтобы действительно существовало в природе.

Поэтому, для представления таких объектов и вводится понятие абстрактный.

Абстрактные методы – это определение метода в суперклассе, но они не имеют реальной реализации.

Это только заголовок метода.

Что это такое, так это контракт между суперклассом и подклассом.

Как суперкласс, я диктую, что должен реализовывать подкласс.

Чтобы сделать это, вы помещаете ключевое слово `abstract` и затем заголовок метода с типом возврата и параметрами.

Вам также не нужно устанавливать видимость метода – публичный он или приватный, потому что вы решите это в подклассе.

Как только у вас появятся абстрактные методы в определении класса, значит вы создали абстрактный класс.

Теперь, если у вас есть ключевое слово `abstract` где угодно, внутри этого класса, вы должны добавить слово `abstract` в заголовок класса.

Теперь, когда вы используете абстрактный класс и когда вы используете интерфейс?

```
public abstract class Mammal {  
    public String furType;  
    public String covering() {  
        return furType;  
    }  
    abstract boolean bornLive();  
}
```

В конечном счете, абстрактный класс – это просто обычный суперкласс, к которому вы добавляете некоторое поведение.

Это похоже на добавление интерфейса к существующему суперклассу.

Итак, вы используете абстрактный класс, когда вам нужно совместно использовать код и гарантировать поведение в близко связанных классах.

Так как абстрактный класс объявляет общую функциональность для семейства связанных классов.

Вы также используете абстрактный класс, если у вас есть специфические подклассы, которые должны расширять общие классы.

Вы будете использовать интерфейс, когда вы хотите гарантировать поведение несвязанных классов.

С помощью интерфейсов вы также можете использовать дополнительное наследование, если у вас уже есть расширение класса и вы уже использовали ваше основное наследование.

Тогда вы можете использовать интерфейсы для добавления в отношения между классами, потому что вы знаете, что всегда можете добавить столько интерфейсов, сколько вам нужно, но при этом вы можете расширить только один родительский класс.

Наследование позволяет делиться кодом с классами и связывать их в иерархию.

Но, скорее всего, вы также заметили, что есть некоторые ограничения наследования.

Например, вы можете создавать иерархические отношения с помощью расширения только одного класса.

Поэтому, как создать отношения, которые будут более сложными, чем просто класс и суперкласс?

Или если у вас есть определенный метод, который отличается своей реализацией на каждом уровне наследования, так что теряется смысл его наследования?

Вот где выходят на сцену интерфейсы.

Интерфейс – это контракт, это не реализация, а это всего лишь список определений методов, которые гарантируют поведение вашего объекта.

Он позволяет связывать ваши объекты, не основываясь на иерархии, а скорее на поведении.

Объекты будут иметь похожее поведение, даже если они тесно не связаны.

Например, все живые существа, растения или животные, нуждаются в воспроизводстве, но способ, которым они это делают, настолько отличается.

Чтобы установить такого рода связь в дереве наследования, понадобится много дополнительных классов.

Интерфейсы устанавливают стандарт поведения, как ваш объект будет связан с внешними объектами.

Для определения интерфейса вы используете ключевое слово `interface`.

```
public interface name {  
    public returnType methodName(paramType paramName, ...);  
    public returnType methodName(paramType paramName, ...);  
    ...  
}
```

```
public class objectName implements interfaceName {
```

А для его реализации используете ключевое слово `implements`.

Интерфейсы программирования API. Стандартная библиотека Java



Объектно-ориентированное
программирование на Java

Концепции объектно-ориентированного
программирования

Лекция 13

Интерфейсы программирования API
Стандартная библиотека Java

Теперь, при создании программ используется интерфейс программирования приложений API Application Programming Interface.

Хорошо, что такое API?

Чтобы ответить на этот вопрос, давайте проведем аналогию.

Представьте, что вы хотите сделать пиццу дома, и что вам нужно приготовить тесто.

Если у вас есть время, и если вы умеете готовить, вы можете сделать это из ингредиентов, таких как мука, вода, масло и дрожжи.

Вам нужно будет смешать все ингредиенты, и затем замесить тесто.

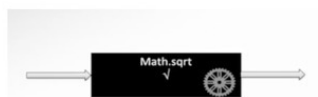
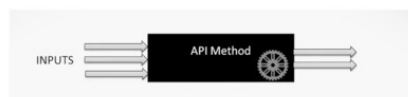
Таким образом, основа для вашей пиццы может быть готова через несколько минут.

Однако, если у вас нет времени или вы не знаете точно, как приготовить тесто, вы можете купить готовую основу для пиццы в супермаркете.

Эта основа уже была сделана другим шеф-поваром, и она предлагается вам как отдельный элемент, который поможет вам в решении вашей задачи.

В случае программирования, API – это набор инструментов и методов, который инкапсулирует некоторую функциональность и который доступен для программистов в виде библиотеки для упрощения программирования задачи.

Таким образом, вам нужно только предоставить входные данные для соответствующего метода API, и он обеспечит вам выходные данные, как волшебный черный ящик.



В частности, стандартная библиотека Java предоставляет множество классов с соответствующими наборами методов и параметров, и вместе с большим количеством документа-

ции, где вы можете найти, как использовать классы и методы с соответствующими входными и выходными данными.

Эта документация была создана с использованием инструмента Javadoc.

Например, стандартная библиотека Java содержит класс `Math`, который предоставляет методы для выполнения математических операций, таких как извлечение квадратного корня.

Таким образом, если вы хотите, например, вычислить квадратный корень числа, вам не нужно для этого программировать весь алгоритм.

Потому что это уже кто-то сделал.

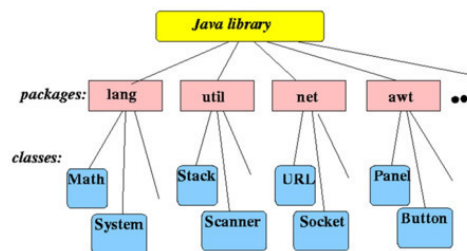
Вы можете просто использовать готовый код, используя метод класса.

Вам потребуется указать число в качестве входных данных, и метод вернет квадратный корень из числа в качестве вывода.

Существует много различных API, которые предоставляются различными разработчиками программного обеспечения и которые позволяют вам взаимодействовать, например, с различными приложениями, такими как Twitter, Google Maps и другие.

Самый важный API, который вы должны знать, это стандартная библиотека Java, которая поставляется вместе со JRE средой выполнения Java и JDK набором разработчика Java.

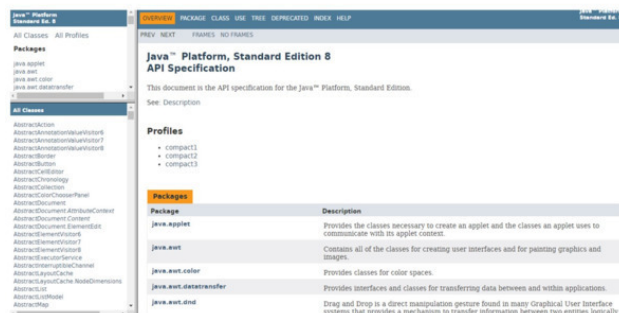
В любом API очень важно научиться, как перемещаться по библиотеке, чтобы узнать, какие элементы она предоставляет, и как их использовать.



Конкретно, Java API организует классы в пакеты, где каждый пакет содержит классы, которые предоставляют общую функциональность.

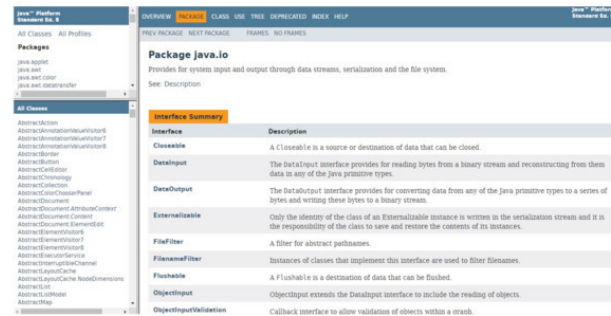
Каждый пакет может содержать классы и интерфейсы, и пакеты можно разделить на подпакеты.

Таким образом, библиотека имеет древовидную структуру, которая похожа на структуру каталогов в файловой системе.



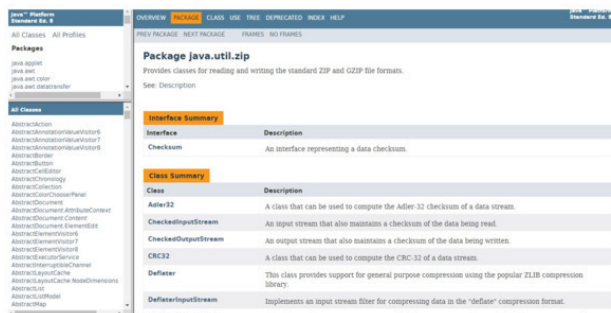
В библиотеке Java, все пакеты являются подпакетами пакетов `java`, `javax` и `org`.

Например, пакет `io` содержит классы для управления входом и выходом системы, таким как управление файлами, и многим другим.



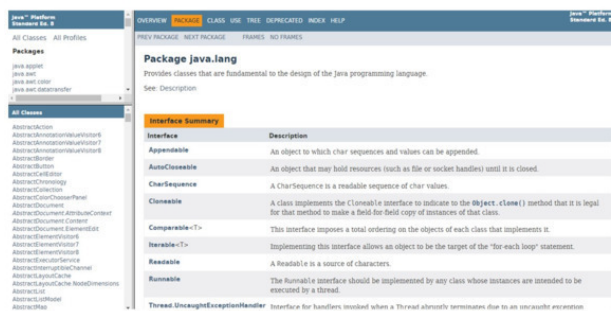
Другой пример.

Внутри пакета `util` мы можем найти подпакет `ZIP`, который содержит несколько классов для сжатия и распаковки данных с помощью известных форматов `ZIP` и `GZIP`.



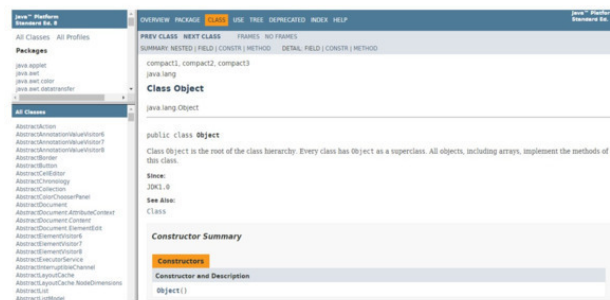
Помните, что, когда вы хотите использовать любой элемент из пакета, вы должны импортировать его в начале исходного кода Java, используя слово `import`, за которым следует имя пакета.

Основным пакетом Java API является `lang`, который импортируется по умолчанию в каждом новом классе и содержит базовые классы, такие как `String` или `Math`.



Этот пакет содержит класс `Object`.

Java – это объектно-ориентированная иерархия с одним корневым элементом, в которой все классы унаследованы прямо или косвенно от этого единственного корневого класса `Object`.



Другими словами, все классы имеют Object как суперкласс.

Это означает, что все классы наследуют заданную функциональность, предоставляемую методами этого класса.

Среди прочих других методов класс Object содержит метод, называемый equals, который указывает, является ли какой-либо данный объект, равным другому объекту.

Метод clone, который создает и возвращает копию данного объекта.

Метод toString, который возвращает текстовое представление объекта.

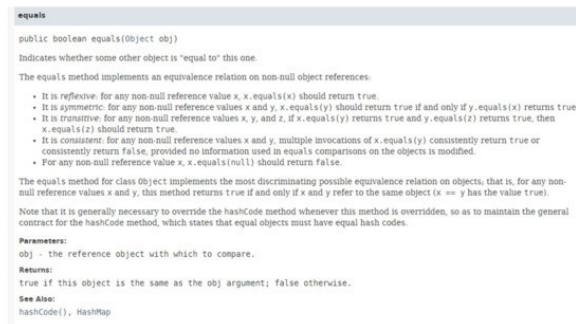
Все эти методы хорошо определены и реализованы в классе Object.

И соответственно, они наследуются всеми остальными классами Java.

Помните, что все классы по умолчанию являются подклассами класса Object.

Эти методы могут быть переопределены подклассами, чтобы лучше реализовать их функциональность.

Метод equals сравнивает два объекта и возвращает true, если вызывающий метод объект равен другому объекту, указанному в качестве параметра.



Что значит быть равным?

Опять же, это будет зависеть от конкретного объекта.

И этот метод следует переопределить для получения желаемого свойства.

Например, мы можем считать, что две машины равны, если они имеют одну и ту же модель, и цвет, хотя владелец может быть другим.

Если метод equals не переопределен, он будет указывать на то, что два объекта, x и y, одинаковы, возвращая true, если они ссылаются на один и тот же объект, что означает, что они размещены в одном и том же месте в физической памяти системы.

Метод clone создает точную копию объекта в памяти.



Это означает, что один и тот же объект копируется в новое место в памяти.

Таким образом, `x.equals(x.clone())` вернет `false`, поскольку эти два объекта не имеют одного и того же адреса в памяти.

`x.equals(x.clone())` → **FALSE**

Описание Javadoc метода `toString` класса `Object` говорит, что он возвращает строковое представление объекта.

```

toString

public String toString()

Returns a string representation of the object. In general, the toString method returns a string that “textually represents” this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

The toString method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

getClass().getName() + '@' + Integer.toHexString(hashCode())

Returns:
a string representation of the object.

```

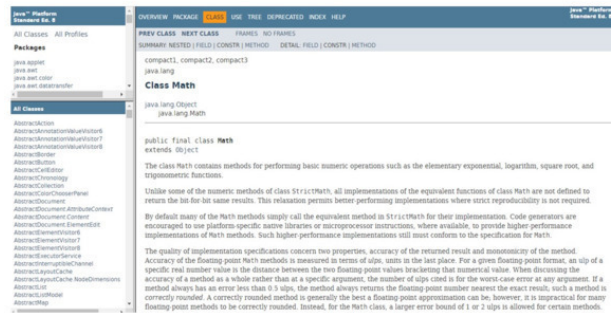
`NameOfClass@MemoryAddressHexadecimal`

Как выглядит это представление String?

По умолчанию метод возвращает имя класса и шестнадцатеричное представление физического местоположения объекта в памяти.

Чтобы переопределить этот метод, вы можете вернуть строку этого метода в удобном для вас виде.

Теперь рассмотрим класс `Math`, который предоставляет методы для выполнения основных математических операций.



Например, он предлагает две важные константы, которые представляют собой числа π и e , а также тригонометрические функции.

Field Summary	
Fields	
Modifier and Type	Field and Description
static double	E The double value that is closer than any other to e , the base of the natural logarithms.
static double	PI The double value that is closer than any other to π , the ratio of the circumference of a circle to its diameter.
Method Summary	
All Methods	Static Methods
Modifier and Type	Method and Description
static double	abs(double a) Returns the absolute value of a double value.
static float	abs(float a) Returns the absolute value of a float value.
static int	abs(int a) Returns the absolute value of an int value.
static long	abs(long a) Returns the absolute value of a long value.

Класс Math принадлежит к пакету java.lang, и, следовательно, он автоматически импортируется в каждый Java-код.

Важно знать, что этот класс определяется как статический.

Это означает, что вам не нужно использовать оператор new для создания экземпляра этого класса для доступа к его публичным методам и полям.

Напротив, вы можете просто написать имя класса, Math, затем точка, и его метод или поле, к которому вы хотите получить доступ.

```
Math.abs(-3) → 3
Math.abs(-3.5) → 3.5
```

Метод abs возвращает абсолютное значение числа.

static double	abs (double a)	Returns the absolute value of a double value.
static float	abs (float a)	Returns the absolute value of a float value.
static int	abs (int a)	Returns the absolute value of an int value.
static long	abs (long a)	Returns the absolute value of a long value.
static double	acos (double a)	Returns the arc cosine of a value, the returned angle is in the range 0.0 through π .
static int	addExact (int x, int y)	Returns the sum of its arguments, throwing an exception if the result overflows an int.
static long	addExact (long x, long y)	Returns the sum of its arguments, throwing an exception if the result overflows a long.
static double	asin (double a)	Returns the arc sine of a value, the returned angle is in the range $-\pi/2$ through $\pi/2$.
static double	atan (double a)	Returns the arc tangent of a value, the returned angle is in the range $-\pi/2$ through $\pi/2$.
static double	atan2 (double y, double x)	Returns the angle theta from the conversion of rectangular coordinates (x, y) to polar coordinates (r, theta).
static double	cbrt (double a)	Returns the cube root of a double value.

Как вы можете видеть, здесь есть четыре разные версии, каждая из которых получает свой тип параметров.

Как следует из описания, если аргумент не отрицателен, возвращается тот же самый аргумент.

И если аргумент отрицательный, возвращается аргумент с минусом.

```

pow

public static double pow(double a,
                        double b)

Returns the value of the first argument raised to the power of the second argument. Special cases:
• If the second argument is positive or negative zero, then the result is 1.0.
• If the second argument is 1.0, then the result is the same as the first argument.
• If the second argument is NaN, then the result is NaN.
• If the first argument is NaN and the second argument is nonzero, then the result is NaN.
• If
  • the absolute value of the first argument is greater than 1 and the second argument is positive infinity, or
  • the absolute value of the first argument is less than 1 and the second argument is negative infinity,
  then the result is positive infinity.
• If
  • the absolute value of the first argument is greater than 1 and the second argument is negative infinity, or
  • the absolute value of the first argument is less than 1 and the second argument is positive infinity,
  then the result is positive zero.
• If the absolute value of the first argument equals 1 and the second argument is infinite, then the result is NaN.
• If
  • the first argument is positive zero and the second argument is greater than zero, or
  • the first argument is positive infinity and the second argument is less than zero,
  then the result is positive zero.
• If
  • the first argument is positive zero and the second argument is less than zero, or
  • the first argument is positive infinity and the second argument is greater than zero,
  then the result is positive infinity.
• If
  • the first argument is negative zero and the second argument is greater than zero but not a finite odd integer, or
  • the first argument is negative infinity and the second argument is less than zero but not a finite odd integer,
  then the result is positive zero.
• If
  • the first argument is negative zero and the second argument is a positive finite odd integer, or

```

Метод `pow` возвращает степень числа.

Как вы можете видеть, этот метод получает в качестве аргументов два значения: `a` и `b` типа `double`.

Значение `a` является базой, а значение `b` является показателем.

И таким образом, метод `pow` возвращает значение `a` в степени `b`.

```

random

public static double random()

Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. Returned values are chosen pseudorandomly with (approximately) uniform distribution from that range.

When this method is first called, it creates a single new pseudorandom-number generator, exactly as if by the expression
    new java.util.Random()

This new pseudorandom-number generator is used thereafter for all calls to this method and is used nowhere else.

This method is properly synchronized to allow correct use by more than one thread. However, if many threads need to generate pseudorandom numbers at a great rate, it may reduce contention for each thread to have its own pseudorandom-number generator.

Returns:
a pseudorandom double greater than or equal to 0.0 and less than 1.0.

See Also:
Random, nextDouble()

```

Метод `random` возвращает псевдослучайное значение, полученное из приблизительно равномерного распределения.

Фактически, случайность очень тяжело получить в информатике, и, таким образом, возможны только псевдослучайные аппроксимации, которые создаются с использованием генератора псевдослучайного числа.

Как указано в документации, в первый раз, когда программист вызывает метод `random`, создается новый генератор, который затем используется.

Метод возвращает псевдослучайные десятичные числа от 0 до 1.

Таким образом, если вы хотите, например, получить случайное число от 0 до 10, вы должны реорганизовать вывод, в этом случае, умножив на 10.

В общем случае, если вы хотите генерировать случайные числа между минимальным значением x и максимальным значением i , вы должны умножить вывод на $(i - x)$, а затем прибавить x .

Метод `sqrt` вычисляет квадратный корень числа.

```

sqrt

public static double sqrt(double a)

Returns the correctly rounded positive square root of a double value. Special cases:
• If the argument is NaN or less than zero, then the result is NaN.
• If the argument is positive infinity, then the result is positive infinity.
• If the argument is positive zero or negative zero, then the result is the same as the argument.
Otherwise, the result is the double value closest to the true mathematical square root of the argument value.

Parameters:
a - a value.

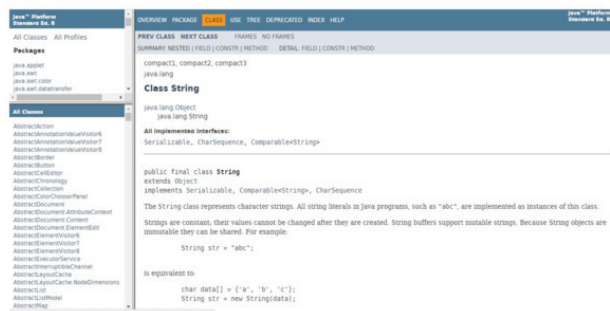
Returns:
the positive square root of a. If the argument is NaN or less than zero, the result is NaN.

```

Это число должно быть значением типа `double`.

Если аргумент не определен или меньше нуля, метод возвращает не определенный результат.

Класс `String` также является классом стандартной библиотеки Java.



Фактически, мы уже использовали переменные этого класса для представления слов и последовательностей символов.

Теперь пришло время лучше понять, как этот класс определен в Java API.

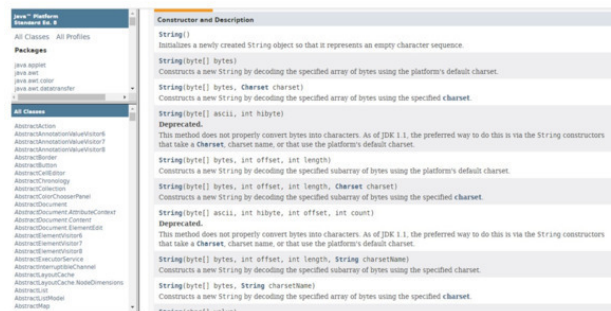
Как вы можете заметить, в документации Java говорится, что `String` хранит постоянные значения, которые не могут быть изменены после того, как они были созданы.

Когда вы создаете переменную типа `String` и назначаете ей последовательность символов, например, `a`, `b` и `c`, это похоже на создание объекта типа `String`, указывая последовательность символов для этого объекта в качестве аргумента.

```
String a= new String ("abc");
```

На самом деле существует несколько конструкторов для объектов String, как вы можете видеть в документации.

Конструктор, который вы будете использовать чаще всего, получает строку в качестве аргумента.



Также существует множество публичных методов в API для класса String.

Среди них есть четыре метода, которые требуют особого внимания.

Это методы compareTo, indexOf, length и substring.

Метод compareTo сравнивает строку, вызывающую этот метод, с другой строкой, заданной как параметр.

```
compareTo

public int compareTo(String anotherString)

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this String object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this String object lexicographically precedes the argument string. The result is a positive integer if this String object lexicographically follows the argument string. The result is zero if the strings are equal; compareTo returns 0 exactly when the equals(Object) method would return true.

This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let k be the smallest such index, then the string whose character at position k has the smaller value, as determined by using the < operator, lexicographically precedes the other string. In this case, compareTo returns the difference of the two character values at position k in the two strings - that is, the value:

    this.charAt(k) - anotherString.charAt(k)

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, compareTo returns the difference of the lengths of the strings - that is, the value:

    this.length() - anotherString.length()

Specified by:
    compareTo in interface Comparable<String>
Parameters:
    anotherString - the String to be compared.
Returns:
    the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.
```

Сравнение проводится с помощью лексикографического порядка (алфавитный порядок).

И в документации по API String вы можете найти дополнительное описание о том, как выполняется сравнение.

Учитывая этот порядок, метод compareTo возвращает отрицательное целое число, если вызывающая строка меньше данного аргумента, возвращает 0, если они одинаковы, или поло-

жительное целое число если строка больше лексикографически (в алфавитном порядке) строки аргумента.

```
String str1 = "Я буду хорошим программистом!";
String str2 = "Я буду хорошим программистом!";
String str3 = "Я буду хорошим дворником!";

int result = str1.compareTo(str2);
System.out.println(result);

result = str2.compareTo(str3);
System.out.println(result);

result = str3.compareTo(str1);
System.out.println(result);
```

0
11
-11

Например, учитывая следующий код, вызов `a.compareTo(b)` вернет отрицательное число.

```
indexOf
public int indexOf(String str)
Returns the index within this string of the first occurrence of the specified substring.
The returned index is the smallest value k for which
    this.startsWith(str, k)
If no such value of k exists, then -1 is returned.
Parameters:
str - the substring to search for.
Returns:
the index of the first occurrence of the specified substring, or -1 if there is no such occurrence.

indexOf
public int indexOf(String str,
                  int fromIndex)
Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
The returned index is the smallest value k for which
    k >= fromIndex && this.startsWith(str, k)
If no such value of k exists, then -1 is returned.
Parameters:
```

Метод `indexOf` имеет разные реализации в зависимости от полученного аргумента.

Наиболее распространенный вариант получает другую строку и возвращает индекс первого вхождения в вызывающую строку подстроки, указанной как аргумент.

Если строка не содержит такой подстроки, тогда метод вернет – 1.

Например, учитывая следующий код, вызов `a.indexOf(bc)` вернет 1.

```
String a= new String ("abc");
↓
a.indexOf("bc");
```

А вызов `a.indexOf(bd)` вернет минус 1,

Поскольку подстрока `bd` не содержится в вызывающей строке.

Метод `length` очень простой.

```
length

public int length()

Returns the length of this string. The length is equal to the number of Unicode code units in the string.

Specified by:
length in interface CharSequence

Returns:
the length of the sequence of characters represented by this object.
```

Как видно из названия, он возвращает длину вызывающей строки.

Например, `length` строки `abc` вернет 3.

Метод `substring` возвращает новую строку, состоящую из последовательности символов, содержащихся в вызывающей строке.

```
substring

public String substring(int beginIndex)

Returns a string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

Examples:
"unhappy".substring(2) returns "happy"
"harbison".substring(3) returns "bison"
"empire".substring(5) returns "" (an empty string)

Parameters:
beginIndex - the beginning index, inclusive.

Returns:
the specified substring.

Throws:
IndexOutOfBoundsException - if beginIndex is negative or larger than the length of this String object.

substring

public String substring(int beginIndex,
                        int endIndex)

Returns a string that is a substring of this string. The substring begins at the specified beginIndex and extends to the character at index endIndex - 1. Thus the length of the substring is endIndex - beginIndex.

Examples:
"hamburger".substring(4, 8) returns "urge"
```

Этот метод имеет две реализации в API.

Первая реализация получает начальную позицию подстроки и возвращает все символы от этой позиции и до конца строки.

Например, учитывая подстроку `abc`, вызов `substring (1)` вернет `bc`.

Другой вариант метода получает два целых числа, представляющих начальную и конечную позиции возвращаемой последовательности.

Метод возвращает последовательность символов, начиная с начального индекса и до конечного индекса минус 1.

Следуя предыдущему примеру, учитывая строку `abcdef`, вызов `substring (2, 5)` вернет `c, d, и e`, которые являются символами в позиции два, три и четыре.

```
"abcdef".substring(2,5);

↓

cde
```

Вложенные классы



Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 14

Вложенные классы

Мы рассмотрели суперклассы и их подклассы, абстрактные классы и интерфейсы.

Также Java позволяет вам определить класс внутри другого класса. Такой класс называется вложенным классом.

Вложенные классы могут быть статическими и нестатическими.

```
class OuterClass {  
    ...  
    class NestedClass {  
        ...  
    }  
}
```

Вложенные классы, объявленные статическими, называются статическими вложенными классами.

```
class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
    class InnerClass {  
        ...  
    }  
}
```

Нестатические вложенные классы называются внутренними классами.

Нестатические вложенные классы (внутренние классы) имеют доступ к другим членам охватывающего класса, методам и полям, даже если они объявлены приватными.

Статические вложенные классы не имеют доступа к другим членам охватывающего класса, только если они не являются статическими.

Вложенный класс может быть объявлен приватным, публичным, защищенным или приватным в пакете, т.е. без ключевого слова `private`, `public` или `protected`.

При этом внешние классы могут быть объявлены только `public` или без ключевого слова, т.е. приватными в пакете.

Зачем использовать вложенные классы?

Для того чтобы группировать классы, которые используются только в одном месте: если класс полезен только для одного другого класса, тогда логично вставлять его в этот класс.

Использование вложенных классов увеличивает инкапсуляцию.

Рассмотрим два класса А и В, где В нуждается в доступе к членам А, которые объявлены приватными.



Скрывая класс В в классе А, члены А могут быть объявлены приватными, а класс В может получить к ним доступ. Кроме того, сам класс В можно скрыть от внешнего мира, объявив его приватным.

Как и в случае с методами и переменными экземпляра класса, вложенный нестатический класс связан с экземпляром его охватывающего класса и имеет прямой доступ к методам и полям этого объекта. Кроме того, так как внутренний класс связан с экземпляром, он не может определять какие-либо статические члены.

Т.е. внутренний класс не может иметь статических членов, включая объявление интерфейсов, так как интерфейсы статические, за исключением статических констант `static final` примитивного типа или `String`, так как такие константы оцениваются во время компиляции.

Объекты, являющиеся экземплярами внутреннего класса, существуют только в экземпляре внешнего класса.

Переменные, объявленные во вложенном классе, являются локальными для этого класса.

Поэтому можно объявлять переменные с одними и теми же именами во внешнем классе и во внутреннем классе.

Вы также можете объявить вложенный нестатический класс, или внутренний класс, внутри метода, цикла или `if` блока.

```
public class Outer{  
    private int count;  
  
    public void methodName(String name){  
  
        final int currentNumber=10;  
  
        class Local{  
            ...  
        }  
        ...  
    }  
}
```

В этом случае такой класс будет называться локальным классом.

Локальный класс имеет доступ к членам охватывающего класса. В данном примере к переменной count.

Также локальный класс имеет доступ к финальным локальным переменным блока кода, где он объявлен. В данном примере к переменной currentNumber.

А также, в Java 8, локальный класс имеет доступ к параметрам метода, где он объявлен.

В данном примере к параметру name.

Локальный класс не может иметь статических членов, включая объявление интерфейсов, так как интерфейсы статические, за исключением статических констант static final примитивного типа или String.

Существует другой вид локальных классов – это анонимные классы.

Предположим, что у нас есть класс Hello и мы хотим по быстрому его расширить в другом классе.

```
public class Hello{  
    public void methodHello(){...}  
}  
  
public class Outer{  
  
    Hello hello=new Hello{  
        @Override  
        public void methodHello(){...}  
    }  
    ...  
}
```

Тогда мы его объявляем без имени и сразу создаем его экземпляр, переопределяя в теле анонимного класса метод суперкласса.

Или другая ситуация.


```
interface Hello{
    void methodHello();
}

public class Outer{

    Hello hello=new Hello{
        public void methodHello(){...}
    }
    ...
}
```

У нас есть интерфейс Hello, и мы хотим по-быстрому реализовать его в классе.

Тогда мы объявляем класс реализации интерфейса без имени и сразу создаем его экземпляр, реализуя в теле анонимного класса метод интерфейса.

Также, как и локальные классы, анонимные классы имеют доступ к членам охватывающего класса.

Также анонимный класс имеет доступ к финальным локальным переменным блока кода, где он объявлен.

И в Java 8, анонимный класс имеет доступ к параметрам метода, где он объявлен.

Но в анонимном классе нельзя объявлять конструкторы.

И анонимный класс, также, как и локальный класс, не может иметь статических членов, включая объявление интерфейсов, так как интерфейсы статические, за исключением статических констант `static final` примитивного типа или `String`.

В Java 8, лямбда-выражение (лямбда) – это короткая замена для анонимного класса.

Лямбда упрощает использование интерфейсов, которые объявляют отдельные абстрактные методы.

Здесь мы объявляем интерфейс с одним методом.

```
interface GreetingService {
    void sayMessage(String message);
}

public class Outer{

    GreetingService greetService = (message) ->
    System.out.println("Hello " + message);

    greetService.sayMessage("Mahesh");

    ...
}
```

А затем реализуем его в классе с помощью синтаксиса лямбда.

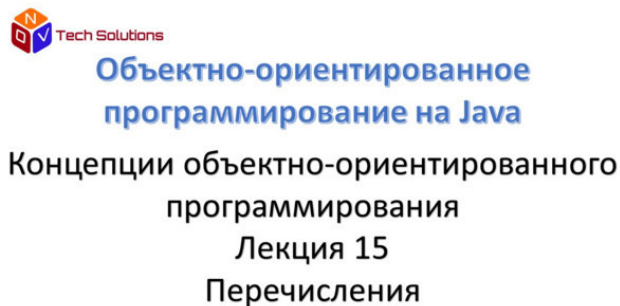
Аргументы метода интерфейса в скобках, но, если аргумент один – скобки можно не указывать.

Затем стрелка и реализация интерфейса.

Все очень лаконично и просто.

Перечисления

Программируя мы часто сталкиваемся с необходимостью ограничить множество допустимых значений для некоторого типа данных.



```
public enum Season {  
    WINTER,  
    SPRING,  
    SUMMER,  
    AUTUMN  
}  
  
Season season = Season.SPRING;  
if (season == Season.SPRING) season = Season.SUMMER;  
System.out.println(season);  
  
import static Season.* ;  
Season season = SPRING;
```

Так, например, день недели может иметь 7 разных значений, месяц в году – 12 значений, а время года – 4 значения.

Для решения подобных задач во многих языках программирования со статической типизацией предусмотрен специальный тип данных – перечисление или `enum`.

Не исключением является и Java.

В отличие от статических констант, перечисления предоставляют типизированный, безопасный способ задания фиксированного набора значений.

Перечисления в Java являются классами специального вида, они не могут иметь наследников, и сами в свою очередь наследуются от класса `Enum` пакета `java.lang`.

То есть, объявляя перечисление с помощью ключевого слова `enum`, мы неявно создаем класс производный от `java.lang.Enum`.

И наследование за нас автоматически выполняет компилятор Java.

Перечисления не могут быть абстрактными и содержать абстрактные методы, но могут реализовывать интерфейсы.

Для того чтобы иметь возможность обращаться к элементам перечислений без использования квалифицированного имени, также можно воспользоваться статической декларацией импорта.

Экземпляры объектов перечисления нельзя создать с помощью `new`, каждый объект перечисления уникален, создается при загрузке перечисления в виртуальную машину, поэтому допустимо сравнение ссылок для объектов перечислений, и для перечислений можно использовать оператор `switch`.

Таким образом, элементы перечисления `Season` – `WINTER`, `SPRING` и т. д. – это статически доступные экземпляры класса перечисления `Season`.

```
Season season = Season.SUMMER;  
  
if (season == Season.AUTUMN) season = Season.WINTER;
```

И их статическая доступность позволяет нам выполнять сравнение с помощью оператора сравнения ссылок `==`.

Довольно часто возникает задача получить элемент перечисления по его строковому представлению.

```
String name = "WINTER";  
  
Season season = Season.valueOf(name);  
  
Arrays.toString(Season.values());  
  
season.ordinal();
```

Для этих целей в каждом классе перечисления компилятор автоматически создает специальный статический метод `valueOf`, который возвращает элемент перечисления по его строковому представлению.

Обратите внимание, что если элемент не будет найден, то будет выброшено исключение `IllegalArgumentException`, а в случае, если `name` равен `null` – будет выброшено исключение `NullPointerException`.

Иногда необходимо получить список всех элементов перечисления во время выполнения.

Для этих целей в каждом классе перечисления компилятор создает метод `values`.

И обратите внимание, что ни метод `valueOf`, ни метод `values` не определен в классе `java.lang.Enum`.

Вместо этого они автоматически добавляются компилятором на этапе компиляции класса перечисления.

Метод `ordinal` возвращает порядковый номер элемента перечисления.

Как и обычные классы, перечисления могут реализовывать поведение и содержать вложенные классы-члены.

```
enum Direction {
    UP, DOWN;
    public Direction opposite() { return this == UP ? DOWN : UP; }
}

enum Direction {
    UP {
        public Direction opposite() { return DOWN; }
    },
    DOWN {
        public Direction opposite() { return UP; }
    };
    public abstract Direction opposite();
}
```

Можно добавлять собственные методы как в перечисление, так и в его элементы.

То есть отдельные элементы перечисления могут реализовывать свое собственное поведение.

Класс перечисления может иметь конструктор `private` либо `private-package`, который вызывается для каждого элемента при его декларации.

```
public enum Direction {
    FORWARD(1.0) {
        public Direction opposite() { return BACKWARD; }
    },
    BACKWARD(2.0) {
        public Direction opposite() { return FORWARD; }
    };
    private double ratio;

    Direction(double r) { ratio = r; }

    public double getRatio() { return ratio; }

    public static Direction byRatio(double r) {
        if (r == 1.0) return FORWARD;
        else if (r == 2.0) return BACKWARD;
        else throw new IllegalArgumentException();
    }
}
```

С помощью перечисления в Java можно реализовать иерархию классов, объекты которой создаются в единственном экземпляре и доступны статически.

```
enum Type {
    INT(true) {
        public Object parse(String string) { return Integer.valueOf(string); }
    },
    INTEGER(false) {
        public Object parse(String string) { return Integer.valueOf(string); }
    },
    STRING(false) {
        public Object parse(String string) { return string; }
    };
    boolean primitive;
    Type(boolean primitive) { this.primitive = primitive; }
    public boolean isPrimitive() { return primitive; }
    public abstract Object parse(String string);
}
```

При этом элементы перечисления могут содержать собственные конструкторы.

Здесь объявляется перечисление `Type` с тремя элементами `INT`, `INTEGER` и `STRING`.

И компилятор создаст следующие классы и объекты:

`Type` – класс производный от `java.lang.Enum`

`INT` – объект класса производного от `Type`

`INTEGER` – объект другого класса производного от `Type`

`STRING` – объект еще одного класса производного от `Type`.

При этом объекты классов `INT`, `INTEGER` и `STRING` будут существовать в единственном экземпляре и будут доступны статически.

Компиляция и выполнение программ



Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 16

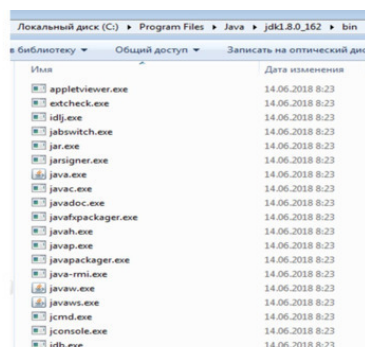
Компиляция и выполнение программ

Набор разработчика JDK, помимо среды выполнения JRE и стандартной библиотеки, содержит разнообразные инструменты командной строки.

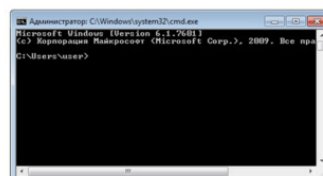
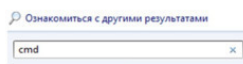
Интегрированная среда разработки, такая как IntelliJ IDEA, использует набор разработчика JDK за сценой.

Это вы можете увидеть в меню Project Structure Java проекта.

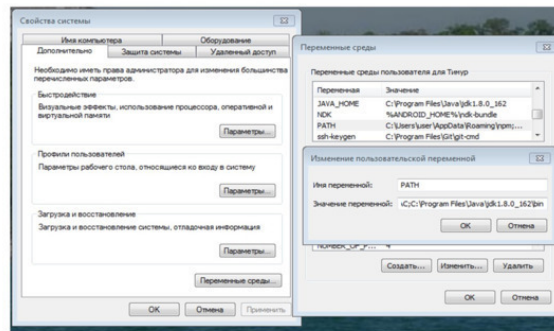
В частности, JDK содержит в папке bin компилятор `javac` и инструмент `java` для запуска Java программы.



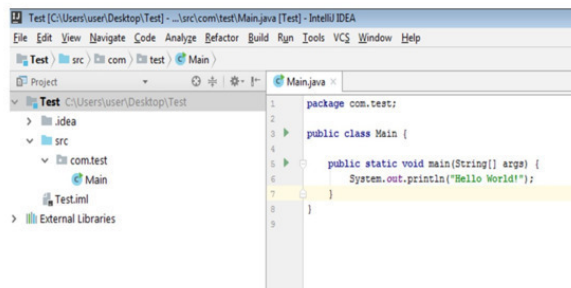
Эти инструменты вызываются из консоли, которая открывается в Windows в меню Пуск с помощью программы `cmd.exe`.



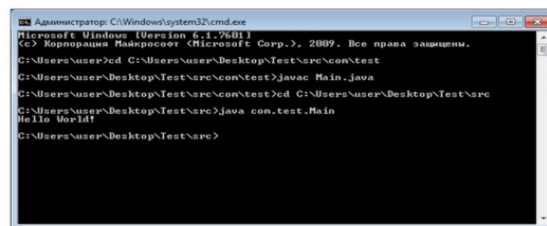
Для того чтобы эти инструменты запускались из любого каталога, а не только из папки bin JDK, нужно прописать путь к папке bin в переменной среды PATH.



Предположим, у нас есть Java класс Main и соответственно файл Main. java.



Теперь, чтобы откомпилировать java файл в консоли перейдем в папку с файлом с помощью команды cd.



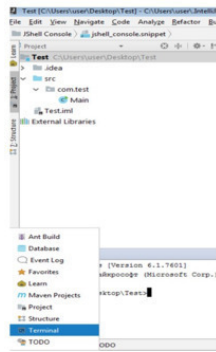
И наберем команду javac затем имя файла Main. java.

В результате, в папке будет создан откомпилированный файл Main.class.

Затем с помощью команды cd перейдем в папку до пакета класса и наберем команду java а затем полное квалифицированное имя класса.

В результате будет запущен метод main класса.

В среде IntelliJ IDEA консоль можно запустить, открыв меню в нижнем левом углу.



Далее можно вводить все те же самые команды.

```
Terminal
+ Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\Users\user\Desktop\Test>javac src\com\test\Main.java

C:\Users\user\Desktop\Test>od src

C:\Users\user\Desktop\Test\src>java com.test.Main
Hello World!

C:\Users\user\Desktop\Test\src>
```

Чтобы откомпилировать все классы пакета, можно использовать шаблон *.java.

В главном классе нашей программы есть метод main, который в качестве аргумента принимает массив строк.



Этот массив строк в качестве аргумента можно передать в программу при запуске из командной строки.

Для этого изменим класс Main, чтобы обработать передаваемые аргументы.

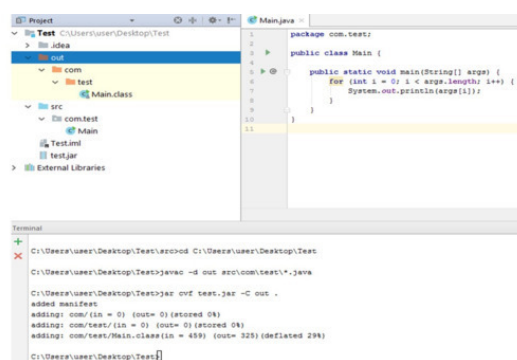
Заново откомпилируем класс инструментом javac и запустим код, набрав команду java, затем полное имя класса, и затем аргументы метода main.

Чтобы разделить аргументы по строкам, нужно использовать двойные кавычки.


```
C:\Users\user\Desktop\Test\src>java com.test.Main "arg0 arg1" arg2
arg0 arg1
arg2

C:\Users\user\Desktop\Test\src>
```

Для создания архивного файла `jar`, создадим папку `out` и наберем команду `javac`, затем опцию `d` и `out`, чтобы указать, что откомпилированные файлы нужно поместить в папку `out`.



Затем наберем команду `jar`, затем опцию `c`, которая показывает, что вы хотите создать JAR-файл, затем опцию `v`, которая выводит подробный отчет в консоль, и опцию `f`, которая показывает, что вы хотите направить вывод в файл.

Далее идет имя создаваемого JAR-файла, и затем опция `C` и `out`, чтобы указать, где нужно брать файлы для архивации.

В результате будет создан JAR-файл test.jar.

JAR-архив предназначен для удобства распространения программ, написанных на Java, так как программа может содержать сотни, тысячи, а иногда и миллионы файлов.

JAR-файл содержит файл манифеста, class-файлы и необходимые ресурсные файлы, также может содержать исходный код и цифровую подпись, которая позволяет защитить программу от модификации.

Манифест – это текстовый файл формата ключ-значение, и он содержит описание jar-файла.

И в нем может быть указан главный класс, содержащий метод `main`, тогда `jar`-файл будет исполняемым файлом, а также манифест может содержать другие ключи, например, относящиеся к цифровой подписи `jar`-файла.

Без указания главного класса, jar-файл будет не исполняемым файлом, а библиотекой классов.

И мы можем его запустить командой `java` с опцией `—jar`.

```
java -jar test.jar
no main manifest attribute, in test.jar

jar cfe test.jar com.test.Main -C out .
```

Однако в нашем случае выскочит ошибка – не найден главный класс, так как в нашем манифесте он не указан.

Поэтому мы должны собрать jar-файл с опцией `e` – entry point – точка входа, указав главный класс.

После этого мы сможем запустить его командой `java`.

Теперь, когда мы разобрали простой случай, обсудим переменную `CLASSPATH`.

`CLASSPATH` или путь к классу – это путь, который указывает инструментам JDK и приложениям, где можно найти сторонние и пользовательские классы, то есть классы, которые не являются расширениями платформы Java или частью платформы Java.

```
sdkTool -classpath classpath1;classpath2...

set CLASSPATH=classpath1;classpath2...

java -classpath C:\java\MyClasses utility.myapp.Cool

java -classpath C:\Users\user\Desktop\Test\out com.test.Main

java -classpath C:\java\MyClasses\myclasses.jar utility.myapp.Cool
```

Здесь `sdkTool` – инструменты командной строки `java`, `javac`, `javadoc`, и др.

`CLASSPATH` или путь к классу может быть задан с помощью параметра `-classpath` при вызове инструмента JDK (и это предпочтительный метод) или путем установки переменной среды `CLASSPATH`.

Также `CLASSPATH` может быть расширен в файле манифеста jar-файла.

Опция `-classpath` предпочтительнее, потому что вы можете установить ее отдельно для каждого случая.

После опции `classpath` указываются пути к файлам `jar`, `zip` или `class`.

Каждый путь должен заканчиваться именем файла или каталогом в зависимости от того, для чего вы устанавливаете путь к классу.

Для файла `jar` или `zip`, содержащего файлы `class`, путь заканчивается именем файла `zip` или `jar`.

Для файлов `class` в неименованном пакете путь заканчивается каталогом, который содержит файлы `class`.

Для файлов `class` в именovanном пакете путь заканчивается каталогом, который содержит «корневой» каталог или первый каталог в полном имени пакета.

Путь по умолчанию – это текущий каталог.

Путь к классу должен найти любые классы, которые вы скомпилировали с помощью javac-компилятора.

JDK и JVM ищут классы, сначала просматривая классы платформы Java в библиотеке rt.jar папки lib JRE или JDK, затем просматривая классы расширения платформы Java папки ext JRE или JDK, и затем используя путь класса CLASSPATH в конце.

Записи пути класса могут содержать символ подстановочного имени *, который считается эквивалентным заданию списка всех файлов в каталоге, за исключением class-файлов.

Для class-файлов просто указывается каталог.

И подкаталоги не ищут рекурсивно.

То есть указание каталога не означает, что поиск будет производиться в его под-каталогах.

Java классы организованы в пакеты, которые сопоставляются с каталогами в файловой системе.

Но, в отличие от файловой системы, всякий раз, когда вы указываете имя пакета, вы указываете имя всего пакета – и никогда его часть.

Так как имя пакета является частью класса и не может быть изменено, за исключением перекомпиляции класса.

Например, предположим, что вы хотите, чтобы среда выполнения Java находила класс с именем Cool.class в пакете utility.myapp.

Тогда вы указываете путь к классу до каталога, содержащего пакет.

Когда программа запускается, JVM использует параметры пути класса, чтобы найти любые другие классы, определенные в пакете utility.myapp, которые используются классом Cool.

Когда классы хранятся в каталоге, тогда запись пути к классу указывает на каталог, содержащий первый элемент имени пакета.

Но когда классы хранятся в файле архива, файле zip или jar, путь к пути класса – это путь к файлу zip или jar и включая его.

Если есть несколько путей к классу, они разделяются точкой с запятой.

Понимание пути к классу очень важно для всех разработчиков Java.

Хотя использование интегрированных средств разработки скрывает технические аспекты пути к классу, проблема использования CLASSPATH особенно остро стоит при разработке распределенных приложений, так как система, которая будет запускать приложение, скорее всего, будет отличаться от той, в которой происходит разработка.

Предположим, что у нас есть два класса в одном пакете.

```
package com.web;
public class Test1
{
    public static void main(String[] args)
    {
        System.out.println("Run Test1.main()");
    }
}

package com.web;
public class Test2
{
    public static void main(String[] args)
    {
        System.out.println("Run Test2.main()");
        Test1 t1 = new Test1();
    }
}
```

```
[root]
com
web
  Test1.java
  Test2.java

cd [root]/com/web
javac -classpath "" Test1.java
javac -classpath "" Test2.java

cd [root]
javac -classpath "." com/web/Test2.java
```

И мы пытаемся их откомпилировать.

Вторая команда с Test2 не пройдет, так как класс Test2 содержит ссылку на класс Test1. Что здесь происходит?

Когда компилятор встречает ссылку на Test1 здесь, он предполагает, что это класс в том же пакете, что и Test2, который в настоящее время компилируется.

Это правильное предположение.

Поэтому компилятору необходимо найти com. web. Test1.

Но искать нечего, так как мы явно задали путь к классу как пустой с помощью двойных кавычек.

То же самое будет, если мы укажем путь с помощью точки, или текущий каталог, так как путь должен быть именно com/web, а не текущий каталог.

Одним из решений будет перейти в корневой каталог, и уже оттуда компилировать Test2.

Этот пример демонстрирует сложность работы с CLASSPATH.

Classpath представляет собой связующее звено между исполняемым модулем Java и файловой системой.

Он описывает, где компилятор и интерпретатор ищут пакет файлов class для загрузки.

Основная идея заключается в том, что иерархия файловой системы отражает иерархию пакета Java, а classpath указывает, какие директории в файловой системе являются корневыми для иерархии пакета Java.

К сожалению, файловые системы очень сложны, зависят от платформы, и не очень хорошо согласуются с пакетами Java.

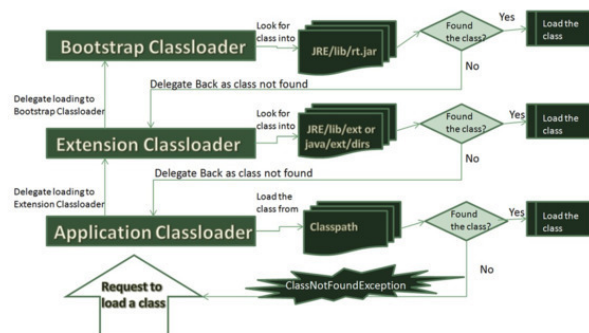
Поэтому classpath является источником постоянного раздражения, как у новых пользователей, так и у опытных Java-программистов.

Теперь, что значит, что JVM и JDK ищут классы.

Надо заметить, что инструменты JDK, такие как компилятор, могут работать без запуска виртуальной машины, виртуальную машину запускает инструмент java.

Так вот, поиск классов осуществляется либо самими инструментами JDK, либо в случае запуска виртуальной машины, загрузчиком классов.

И он оперирует classpath.



Классы загружаются по мере надобности, то есть динамически, за небольшим исключением.

Некоторые базовые классы из библиотеки rt. jar (например, пакет java.lang) загружаются при старте приложения.

Классы расширений папки ext, пользовательские и большинство системных классов загружаются по мере их использования.

Соответственно, различают 3-и вида загрузчиков в Java.

Это – базовый загрузчик, загрузчик расширений и загрузчик приложения.

Базовый загрузчик реализован на уровне JVM и не имеет обратной связи со средой исполнения.

Данным загрузчиком загружаются классы из каталога lib.

Управлять загрузкой базовых классов можно с помощью опции `-Xbootclasspath`, которая позволяет переопределять наборы базовых классов.

Загрузчик приложения реализован уже на уровне JRE.

Это класс `AppClassLoader`. Этим загрузчиком загружаются классы, пути к которым указаны в `CLASSPATH`.

Управлять загрузкой пользовательских классов можно с помощью опции `—classpath`.

Загрузчик расширений загружает классы из каталога `ext`.

Это класс `ExtClassLoader`.

Загрузчики классов образуют иерархию.

Корневым является базовый загрузчик.

Такая иерархия необходима для модели делегирования загрузки.

То есть право загрузки класса рекурсивно делегируется от самого нижнего загрузчика в иерархии к самому верхнему.

Такой подход позволяет загружать классы тем загрузчиком, который максимально близко находится к базовому.

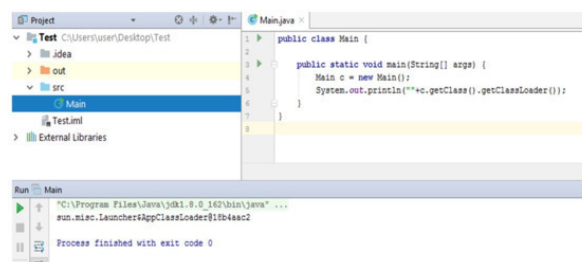
Так достигается максимальная область видимости классов.

Под областью видимости подразумевается следующее.

Каждый загрузчик ведет учет классов, которые были им загружены.

Множество этих классов и называется областью видимости.

Получить загрузчик класса можно методом `getClassLoader`.



Если вы хотите создать свои пользовательские загрузчики, они должны расширять класс `java.lang.ClassLoader`;

И поддерживать модель динамической загрузки.

Модульность



Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 17

Модульность

Еще одним основным принципом объектно-ориентированного программирования, наряду с абстракцией и инкапсуляцией, является принцип модульности.

Современные программные системы состоят, как правило, из нескольких различных компонентов, правильное взаимодействие которых обуславливает работу всей системы в целом.



Для обеспечения такого взаимодействия необходима соответствующая организация компонентов системы.

Объектно-ориентированный подход связывает возможность такой организации с концепцией модульности.

Эта концепция состоит в необходимости разбиения различных компонентов программной системы на отдельные функциональные блоки.

Например, дом может рассматриваться как совокупность нескольких взаимосвязанных систем: система электропитания, системы отопления и охлаждения, водопровод и собственно дом как структура.

Вместо того чтобы рассматривать все эти системы совместно, как огромный клубок проводов, клапанов, труб и щитов, квалифицированный архитектор, занимающийся проектированием дома или квартиры, рассматривает их как отдельные модули, взаимодействующие известным образом.

Таким образом, принцип модульности применяется для более четкого представления об объектах и организации работы отдельных блоков.

Подобным образом модульность в программной системе позволяет создать удобную схему реализации программного продукта.

Полученная таким образом модульная структура обеспечивает возможность многократного применения программного обеспечения.

Если отдельные модули программы созданы с помощью абстрактного представления и предназначены для решения общих задач, то они могут использоваться и в тех случаях, когда подобные задачи возникают в другом контексте.

Таким образом, модульность – это процесс разложения задачи на набор модулей, чтобы уменьшить общую сложность задачи.

Модульность – это свойство системы, которая была разложена на множество когезионных, то есть внутренне связанных, и слабо связанных между собой модулей.

Модульность - это свойство системы, которая была разложена на множество когезионных и слабо связанных модулей.

Модульность связана с инкапсуляцией.

И модульность может быть представлена как способ отображения инкапсулированных абстракций в реальные физические модули, имеющие высокую степень связанности внутри модулей, а их межмодульное взаимодействие или связь является слабой.

Модульность – это метод, позволяющий упростить как проектирование, так и обслуживание программного продукта.

Если разработка программного обеспечения следует модульному подходу и, конечно, построена правильно, она имеет ряд преимуществ как в длительном, так и в краткосрочном плане.

Вот некоторые из этих преимуществ:

Так как программный комплекс состоит из отдельных модулей, их можно использовать повторно. Некоторые (или все) компоненты могут быть повторно использованы в любой другой программе.

Модульный код более читабелен, чем монолитный код.

Его легко поддерживать и обновлять, так как отдельные компоненты решают отдельные задачи. Легко выбрать один модуль и внести необходимые изменения, как можно минимально вызывая изменения в других модулях.

Модульные программы сравнительно легко отлаживаются, так как их модульность изолирует отдельные компоненты для быстрого модульного тестирования. Кроме того, при интеграционном тестировании проблема может быть локализована и исправлена эффективным образом.

До Java 9 модульное программирование было сосредоточено на использовании пакетов и JAR-файлов.

И, возможно, самым простым примером являются библиотеки, которые являются частью Java.

Модульность в Java

Уровни модульности в Java

- класс
- пакет
- программа

Библиотечные модули создаются путем сборки нескольких частей, и каждая часть выполняет дискретную функцию.

Таким образом, модуль в Java – это всего лишь набор классов и интерфейсов, представляющих общую функциональность, которые обычно группируются в пакеты и распространяются как JAR файл.

Теперь каждый модуль имеет публичное представление, то есть набор программных интерфейсов API, предоставляющих средства, с помощью которых внешний мир обменивается данными с модулем.

Поэтому те элементы классов или интерфейсов, которые предназначены для внешнего интерфейса модуля, обозначаются с помощью модификатора `public`.

Основная функция этих публичных элементов классов и интерфейсов – служить в качестве доступного API для взаимодействия с другими модулями.

Приватные элементы, с другой стороны, недоступны извне и предназначены для внутренней работы.

Надо понимать, что Java не строилась, чтобы быть модульной с нуля, но модульность можно достичь с помощью пакетов и JAR файлов.

При этом большинство разработчиков Java сталкиваются с проблемой, называемой JAR-ад или classpath-ад.

Существует несколько проблем с модульным программированием на Java.

Как только вы начинаете использовать какую-либо библиотеку или JAR файл, вам, как правило, становятся доступны все классы в ней.

Дело в том, что разработчики библиотек не имеют возможности спрятать классы, которые используются для реализации их внутренней логики, так как класс может иметь модификаторы доступа только `public`, `final` или `abstract`.

Правда без указания конкретного модификатора классы будут иметь видимость только внутри пакета, и также можно использовать вложенные классы.

Но далеко не всегда можно все спрятать, сильно не усложняя архитектуру библиотеки, и, если вы используете код, который не предполагался для применения вне библиотеки, вы можете столкнуться с несовместимостью при использовании новой версии библиотеки или нарушить ее корректное функционирование.

Также, как только вы начинаете использовать разные версии одной библиотеки, а такое случается в больших проектах, которые эволюционируют со временем, вы можете столкнуться с тем, что один и тот же класс имеет разные методы в разных версиях библиотеки.

Java же устроена так, что будет использована первая версия библиотеки, которую найдет загрузчик классов.

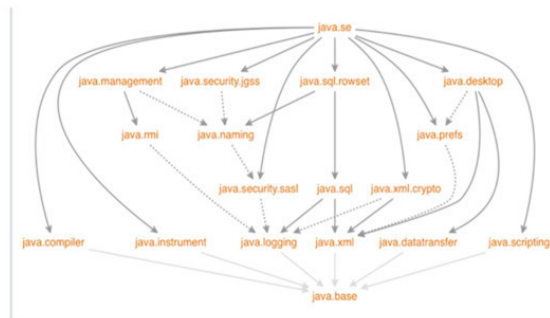
Тем самым, обратившись в коде к какому-либо классу во время выполнения программы, вы получите ошибку, что метод, к которому вы обращаетесь, не существует.

Связано это с тем, что на этапе выполнения Java ничего не знает о версии библиотеки, которая должна использоваться в том или ином случае.

Еще одна неприятная проблема – несоответствие версии.

Библиотека JAR, которая зависит от другой библиотеки JAR, может не работать, потому что версию библиотеки нужно обновить или понизить, чтобы она была совместима для работы.

Модульная система Java 9 разработана с учетом следующих основных идей.



Определение явных зависимостей.

Модульность предоставляет механизмы для явного объявления зависимостей между модулями таким образом, который распознается как во время компиляции, так и во время выполнения.

Система может пройти через эти зависимости, чтобы определить подмножество всех модулей, необходимых для поддержки вашего приложения.

Это устраняет CLASSPATH-ад и предотвращает загрузку лишнего кода.

И вместо classpath в Java 9 вводится module-path.

Хотя в Java 9 module-path работает вместе с classpath.

Используя модули в module-path, JVM может проверять, как во время компиляции, так и во время выполнения, что все необходимые модули присутствуют.

И все просто JAR-файлы в classpath, как члены неименованного модуля, доступны для модулей в module-path и наоборот.

Далее модульная система обеспечивает сильную инкапсуляцию.

Пакеты в модуле доступны для других модулей только в том случае, если модуль явно экспортирует их.

И даже тогда другой модуль не может использовать эти пакеты, если он явно не объявляет, что он требует этих возможностей другого модуля. Это улучшает безопасность платформы.

Перед Java 9 было возможно использовать многие классы платформы, которые не были предназначены для использования классами приложений.

Теперь, из-за сильной инкапсуляции эти внутренние API действительно инкапсулируются и скрываются от приложений.

И это может привести к необходимости перенастройке устаревшего кода в модульную Java 9, если ваш код зависит от внутренних API.

Далее модульная система обеспечивает масштабируемость платформы Java.

Раньше платформа Java была монолитом, состоящим из огромного количества пакетов, что затрудняет разработку, поддержку и развитие.

Теперь платформа состоит из модулей.

И вы можете создавать пользовательские среды выполнения, состоящие только из модулей, которые необходимы для ваших приложений или устройств.

Например, если устройство не поддерживает графические интерфейсы, вы можете создать среду выполнения, которая не включает модули графического интерфейса, что значительно сокращает размер среды выполнения.

Для того чтобы понять модули Java 9, рассмотрим создание модуля в среде IntelliJ IDEA.

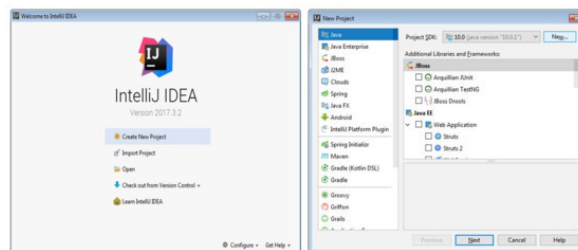
Для начала установим JDK версии 9 или более позднюю версию.

IntelliJ IDEA уже имеет концепцию модулей для проекта.

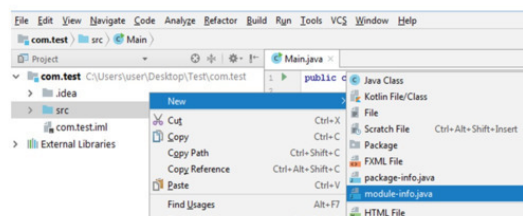
И каждый модуль IntelliJ IDEA создает свой собственный путь к классу classpath.

С внедрением новой модульной системы платформы Java, модули IntelliJ IDEA расширяются, поддерживая module-path платформы Java, если он используется вместо classpath.

Для создания Java-модуля сначала создадим проект IntelliJ IDEA как модуль IntelliJ IDEA, указав JDK 9 или выше.



Затем нажав правой кнопкой мышки на папке src вы выбираете создание module-info.java.



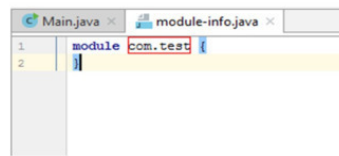
При создании файла объявления модуля module-info.java, IntelliJ IDEA выберет имя модуля IntelliJ IDEA в качестве имени по умолчанию для модуля платформы Java.

Это можно изменить.

Хотя лучше сразу называть проект именем, соответствующим имени создаваемого модуля.

Таким образом происходит именование модуля Java.

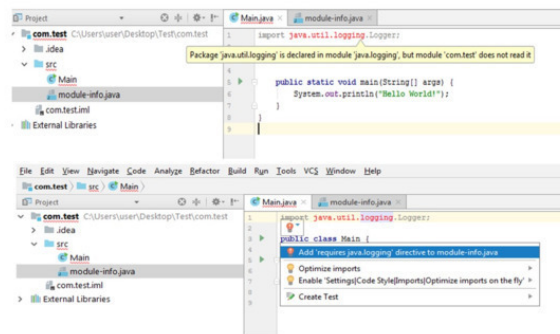
Модуль должен предоставить дескриптор модуля module-info.class, метаданные которого определяют зависимости модуля, пакеты, которые модуль предоставляет другим модулям, и многое другое.



Дескриптор модуля – это скомпилированная версия объявления модуля module-info.java.

Каждое объявление модуля начинается с ключевого слова module, за которым следуют уникальное имя модуля и тело модуля, заключенное в фигурные скобки.

Теперь вы начинаете писать код.



И импортируете какой-нибудь пакет.

И получаете ошибку.

Нажимаете левой кнопкой мышки на ошибке и добавляете зависимость от модуля, содержащего нужный пакет.

Директива модуля requires указывает, что этот модуль зависит от другого модуля – и это отношение называется зависимостью модуля.



Каждый модуль должен явно указывать свои зависимости.

Когда модуль А требует модуля В, говорят, что модуль А читает модуль В, а модуль В считывается модулем А.

Чтобы указать зависимость от другого модуля, используется директива `requires`.

Также есть директива `requires static`, указывающая, что модуль требуется во время компиляции, но является необязательным во время выполнения.

Это используется как необязательная зависимость.

Директива `requires transitive` используется, чтобы указать зависимость от другого модуля и гарантировать, что другие модули, читающие ваш модуль, также читают и эту зависимость.

То есть это сквозная зависимость.

Существуют и другие директивы.

Директива модуля `exports` указывает пакет модуля, публичные типы которого, и их вложенные публичные и защищенные типы, должны быть доступны для кода во всех других модулях.

Директива `exports... to` позволяет указать в списке, разделенном запятыми, точно, какой код модуля может получить доступ к экспортированному пакету – это называется квалифицированным экспортом.

Директива модуля `uses` определяет службу, используемую этим модулем, делая модуль потребителем службы.

Служба – это объект класса, который реализует интерфейс или расширяет абстрактный класс, указанный в директиве `uses`.

Директива модуля `provides... with` указывает, что модуль предоставляет реализацию службы, делая модуль поставщиком службы.

Часть директивы `provides` определяет интерфейс или абстрактный класс, указываемый в директиве модуля `uses`, частью директивы `with` указывается имя класса, реализующего интерфейс или расширяющий абстрактный класс.

Теперь, после создания модуля Java, IntelliJ IDEA будет запускать JVM с помощью `module-path`, а не `classpath`.

Это обеспечит более сильную инкапсуляцию, и решит любые проблемы с зависимостями.

Компиляция `module-info` и классов производится, как и раньше.

```
javac -d out src/module-info.java src>HelloModules.java
java --module-path out --module HelloModules
java --module-path target/hello-modules.jar --module HelloModules
```

А вот выполнение запускается с использованием опции `module-path`, а не `classpath`.

Эта опция устанавливает каталоги, в которых могут быть найдены модули.

А опция `module` устанавливает основной класс, который должен быть вызван.

Jar-файл создается с помощью инструмента `jar` как и раньше.

А его запуск производится с использованием опции `module-path`.

Моделирование с UML



Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 18

Моделирование с UML

Для разработки больших программ требуются не только инструменты программирования, но и концептуальные инструменты, чтобы помочь управлять сложностью программ, состоящих из многих компонентов, и облегчить работу в команде разработчиков.

Мы познакомились с моделированием систем с использованием интерфейсов, абстрактных классов и классов.

И для визуализации переменных, методов, и классов – существует стандартное представление UML, что означает Unified Modeling Language, который является стандартом группы Object Management Group (OMG) и Международной организации по стандартизации ISO.

UML содержит множество различных диаграмм, полезных для моделирования систем.

Мы рассмотрим здесь подмножество так называемых диаграмм классов.

Чтобы представить класс, например, `Vehicle`, мы можем использовать следующую диаграмму.

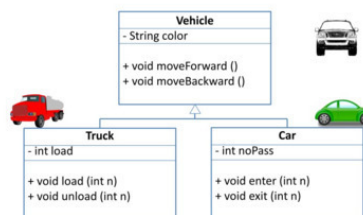


Диаграмма состоит из прямоугольника с тремя разделами.

Первая секция содержит название класса, жирным шрифтом и в центре.

В следующем разделе указываются поля класса.

И в третьем разделе конструкторы и методы с аргументами и возвращаемыми типами, без тела метода.

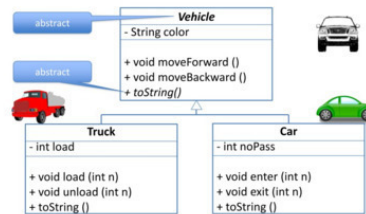
Статические поля и методы класса подчеркиваются.

Наследование выражается с помощью стрелки, идущей от подкласса к суперклассу.

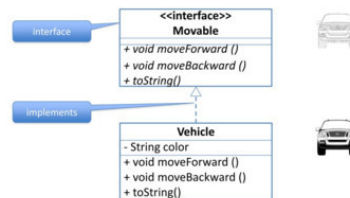
Видимость полей и методов также может быть указана.

Мы используем «плюс», чтобы указать «публичные» поля и методы, и «минус», чтобы указать приватные поля и методы.

Если класс является абстрактным, его название пишется курсивом и абстрактный метод также пишется курсивом.



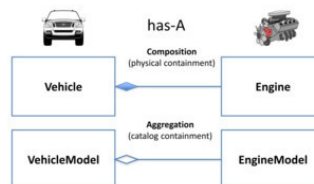
Также мы можем моделировать интерфейсы в UML.



Мы пишем слово интерфейс в двойных скобках.

И обозначаем реализацию пунктиром.

UML также позволяет моделировать различные отношения.



Например, отношения композиции, которые представляют собой физическую связь.

Например, когда автомобиль содержит двигатель, двигатель является частью автомобиля.

Когда автомобиль уничтожается, двигатель также уничтожается.

Другая взаимосвязь – это агрегация.

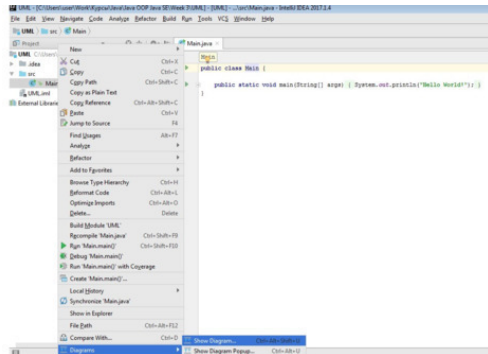
Например, когда модель двигателя автомобиля является частью модели автомобиля в каталоге в базе данных.

Когда модель автомобиля убирается, модель двигателя может остаться.

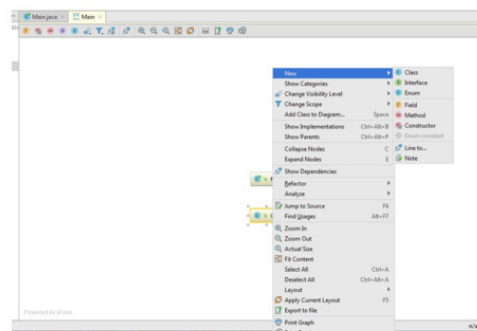
Модели важны в процессе проектирования.

Диаграммы классов помогают нам проектировать наши вычислительные системы, и преимущество состоит в том, что они могут быть транслированы в код Java.

В IDEA вы можете нажать правой кнопкой мышки на классе и выбрать Diagrams -> Show diagrams.



Затем вы можете добавлять классы, интерфейсы, поля и методы.



При этом будет автоматически генерироваться Java код.

Синтаксические ошибки



Объектно-ориентированное программирование на Java

Отладка и тестирование кода

Лекция 1

Синтаксические ошибки

Первый тип ошибок в программах – это синтаксические ошибки.

Синтаксис определяет структуру программ согласно определенного языка программирования.

И это похоже на человеческий язык, где вы строите предложения по некоторым грамматическим правилам.

В первом примере мы написали оператор присваивания вместо логического выражения.



```
if (x=0) {x=1;}

System.out.println("hi,
buddy");
```

Потому что мы написали только один знак равенства, а не два.

Вы можете заметить ошибку во втором примере?

Здесь, мы написали строку на новой линии.

В строке не может быть никаких новых встроенных строк.

Теперь, как вы можете себе представить, существует множество возможных синтаксических ошибок; и их слишком много, чтобы здесь перечислить.

К счастью, синтаксические ошибки улавливаются компилятором, который выдает сообщение об ошибке.

Как правило, сообщение об ошибке содержит указание строки, где была замечена ошибка.

Сама ошибка может фактически находиться на другой строке.

Затем сообщение об ошибке содержит некоторые указания на возможную причину ошибки.

Этот признак может быть более или менее полезным для исправления ошибки.

Обычная ошибка – это забыть о закрывающейся скобке или точке с запятой.

Использование некоторой схемы отступа (либо автоматически, либо вручную) помогает избежать таких ошибок.


В наших интересах, чтобы компилятор мог уловить как можно больше ошибок, так как лучше иметь ошибку компилятора, чем получить ошибку во время выполнения.


Java – это типизированный язык, в отличие от других языков программирования, которые являются не типизированными.

Это означает, что с помощью типизации мы можем поймать некоторые ошибки, которые не являются строго синтаксическими ошибками, но могут быть распознаны компилятором.

Это, конечно, относится не только к примитивным типам, определенным в Java, но и к классам, которые вы определили.


В некоторых языках программирования, если у вас есть несколько аргументов одного типа в методе, вам не нужно повторять тип в списке параметров, как здесь, где в качестве параметров мы имеем две целые переменные *x* и *y*.


```
double average(int x, y)
{return (x+y)/2;} 
```

```
double average(int x, int y)
{return (x+y)/2;} 
```

Однако в Java вам нужно повторить тип параметра.

Кроме того, в других языках вы могли бы привыкнуть не писать ключевое слово `return`.

```
double average(int x, int y)
{ (x+y)/2;} 
```

```
double average(int x, int y)
{return (x+y)/2;} 
```

А просто записывать выражение для вычисления.

Однако в Java необходимо указывать ключевое слово `return`, если что-то возвращается.

Теперь, если ваш метод не имеет параметров, вам все равно придется писать открывающие и закрывающие скобки.

```
String sayHello  
{return "Hello!";}
```



```
String sayHello()  
{return "Hello!";}
```



Аналогично для возвращаемого типа, вам все равно придется написать тип возврата, а именно void.

```
public static void main  
(String []args) {...}
```



В любой программе Java вы должны написать основной метод main, который является точкой входа в программу.

Основной метод main имеет определенный синтаксис.

Он должен быть обязательно публичным и статическим.

Теперь давайте продолжим работу с объектами.

```
meth1(t); meth2(x,y,z);
```



```
t.meth1(); x.meth2(y,z);
```



Если вы привыкли к функциональным или императивным языкам, вы используете для записи сначала метод, а затем в круглых скобках аргумент или аргументы.

В объектно-ориентированном языке, таком как Java, вы должны думать наоборот:

Вы отправляете метод объекту, и вы используете точечную нотацию.

Однако, если у вас есть статический метод, вы не можете отправить его объекту.

`x.abs()``Math.abs(x)`

Он должен быть отправлен классу, а не объекту.

Рассмотрим другие общие ошибки.

Java чувствительна к регистру.

Если вы определяете переменную с именем `myVar` и используете ее как `myvar`, вы имеете в виду другую переменную.

Имена типов (примитивные типы) начинаются с строчных букв, а имена классов с заглавной буквы.

Вы должны быть осторожны, когда используете `int` и `Integer`.

```

int n=1;
Integer x = n;
Integer x = new Integer(n);

```



`String` пишете с заглавной буквы, потому что `String` на самом деле является классом.

Обратите внимание, что имена классов начинаются с заглавной буквы, но ключевое слово – `class` – в нижнем регистре.

Существует много видов скобок: круглые `()`, квадратные `[]`, фигурные `{ }`.

Они должны использоваться правильно.

Это означает, что вы не должны забывать закрыть то, что вы открыли, и что вы должны закрыть скобки в правильном порядке.

Возможно, наиболее распространенной ошибкой является забыть закрыть скобку, например, закрытие `}` в конце определения класса.

правильное закрытие Простых кавычек и двойных кавычек имеет такое же значение, что и закрытие скобок, с той разницей, что символы открытия и закрытия здесь одинаковы.

```
String s = "A loooooooooooooooooong  
string"  
Это не правильно  
  
Правильно:  
String s = "A loooooooooooooooooong string"  
  
или  
  
String s = "A loooooooooooooooooong " +  
"string"
```

В кавычках вам нужно не включать новую строку в строку.

Также вы должны хорошо понимать форму и функцию различных операторов на Java.

Например, вам нужно отличать присваивание = от сравнения ==.

Поскольку строка является объектом, с помощью двойного знака равенства, мы всего лишь проверяем, являются ли адреса внутреннего представления этих объектов одинаковыми, но не проверяем, являются ли эти объекты одинаковыми.

Для сравнения объектов *a* и *b*, и соответственно строк, вы можете использовать метод `equals()`.

Одной из распространенных ошибок является попытка использования класса в программах без импорта требуемого пакета.

Например, если мы хотим использовать экземпляр класса `Vector`, мы всегда должны импортировать пакет `java.util`.

Но почему мы можем использовать класс `String` без импорта какого-либо пакета?

Потому что класс `String` принадлежит пакету `java.lang`, и этот пакет является единственным пакетом, который импортируется автоматически (поэтому мы можем использовать классы, такие как `String` или `Integer`, без импорта пакета `java.lang`.)

Что касается использования двумерных массивов, начинающие программисты предполагают, что двумерные массивы непосредственно реализуются на Java.

Таким образом, общая ошибка заключается в том, чтобы написать:

```
int [,] array = new int [2,5];
```

В Java мы должны сначала создать строки, а затем столбцы.

```
int[,] array = new int[2,5];
```

Двумерные массивы – это одномерные массивы, каждый элемент которых содержит еще один размерный массив.

Поэтому, предыдущее выражение должно быть написано следующим образом.

```
int[][] array = new int[2][];  
array[0] = new int[5];  
array[1] = new int[5];
```

Или:

```
int[][] array = new int[2][5];
```

Выявление ошибок



Объектно-ориентированное программирование на Java

Отладка и тестирование кода

Лекция 2

Выявление ошибок

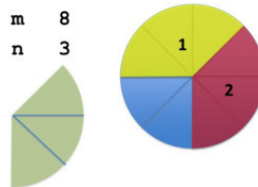
Одним из способов найти ошибки в коде, является запуск кода в своей голове, и, возможно, с помощью ручки и бумаги, для отслеживания значений, которые последовательно принимают переменные.

Это называется «ручной трассировкой», поскольку вы отслеживаете код вручную.

Давайте рассмотрим следующий пример.

Мы хотим написать метод, который вычисляет целочисленное деление двух целых чисел.

У нас есть делимое, m , скажем, восемь, и делитель, n , скажем три.



И мы хотим знать, сколько раз n вписывается в m .

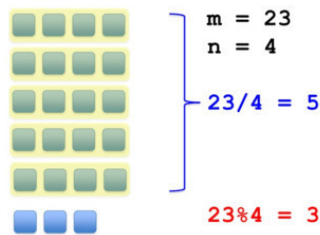
Если у нас есть восемь единиц, мы можем сделать первую группу из трех единиц, и вторую группу из трех единиц.

И тогда у нас остается только две единицы, поэтому мы не можем сделать дополнительную группу из трех единиц.

Поскольку мы смогли сделать только две группы, результат целочисленного деления равен двум.

И так как у нас есть еще две оставшиеся единицы, мы говорим, что остаток равен двум.

Предположим, что делимое m , теперь двадцать три, и n , делитель, равен четырем.



Сколько раз четыре вписывается в двадцать три?

Мы можем составить не более пяти групп из четырех единиц.

Поэтому, результат целочисленного деления – пять.

И остаток равен трем.

Теперь, как мы можем вычислить целочисленное деление?

Пойдем поэтапно.

Мы можем взять делимое, m , и вычитать из него делитель, n , пока это будет возможно.

always $y*n + x == m$

```

int intDiv
(int m, int n) {
    int x=m;
    int y=0;
    while (x>n){
        x=x-n;
        y=y+1;
    }
    return y;
}

```

m	n	x	y
23	4	23	0
		19	1
		15	2
		11	3
		7	4
		3	5

$23/4 = 5$ $23\%4 = 3$

Каждый раз, когда мы вычитаем n из m , мы увеличиваем счетчик y на единицу.

Этот алгоритм можно перевести в Java код с помощью цикла `while`.

Здесь y – это число n -групп, которые мы собрали на каждом шаге, и x – это количество оставшихся единиц, которые будут сгруппированы.

Следовательно, x изначально равно m , поскольку ни одна единица не была сгруппирована, и y равна нулю по этой же причине.

В цикле `while` мы видим, что x уменьшается на n , и y увеличивается на единицу, потому что на каждой итерации мы строим одну группу.

Мы сделаем это, пока есть количество единиц для сбора группы.

Теперь, можно проверить правильность этого алгоритма, запустив алгоритм вручную.

Для этого, мы можем составить таблицу, где мы будем записывать значения наших переменных в разные моменты выполнения.

Здесь мы будем использовать тот же пример, что и раньше, где m – двадцать три, а n – четыре.

Идя по циклу, мы получаем результат деления 5, а остаток 3.

Вроде бы все верно.

Но давайте проверим еще раз с другими входными значениями.

Теперь предположим, что m – двадцать четыре, а n снова четыре.

Здесь сделаем то же самое отслеживание вручную.

always $y \cdot n + x == m$

```
int intDiv
(int m, int n) {
    int x=m;
    int y=0;
    while (x>n){
        x=x-n;
        y=y+1;
    }
    return y;
}
```

m	n	x	y
24	4	24	0
		20	1
		16	2
		12	3
		8	4
		4	5

$24/4 = 5$ $24\%4 = 4?$

Но теперь, когда мы доходим до x равно четырех, мы выходим из цикла, поскольку четыре не больше четырех, и мы возвращаем пять в результате целочисленного деления.

Это верно?

Нет!!!

Если осталось четыре единицы, мы можем сделать еще одну группу из четырех единиц.

Таким образом, нам нужно сделать еще одну итерацию в цикле.

Проблема в том, что условие цикла здесь неправильное.

Оно должно быть x больше или равно n .

Таким образом, используя таблицу для записи значений переменных во время выполнения, можно выявить ошибки кода.

always $y \cdot n + x == m$ $x < n$ at the end

```
int intDiv
(int m, int n) {
    int x=m;
    int y=0;
    while (x>=n){
        x=x-n;
        y=y+1;
    }
    return y;
}
```

m	n	x	y
24	4	24	0
		20	1
		16	2
		12	3
		8	4
		4	5
		0	6

$24/4 = 6$ $24\%4 = 0!$

Теперь, вместо ручного отслеживания, мы можем использовать дополнительный код, чтобы помочь нам отладить программу.

Здесь мы видим метод вычисления целочисленного деления.

```
int intDiv (int m, int n) {
    System.out.println("m: "+m+" n: "+n);
    int x=m; int y=0;
    while (x>=n){
        x=x-n; y=y+1;
        System.out.println("x: "+x+" y: "+y);
    }
    return y;
}
```


И, вместо того, чтобы вычислять вручную значения переменных и записывать их, мы можем позволить компьютеру сделать это для нас.

Здесь мы включили метод печати, который выводит значение переменных *x* и *y* в каждой итерации цикла.

И мы также предварительно напечатали два параметра метода, *m* и *n*.

Также, помимо печати значений переменных, мы можем напечатать при каждом прогоне цикла, выполняется ли условие *y* умножить на *n* плюс *x* равно *m*.

```
int intDiv (int m, int n) {
    System.out.println("m: "+m+" n: "+n);
    int x=m; int y=0;
    while (x>=n){
        x=x-n; y=y+1;
        System.out.println("x: "+x+" y: "+y);
        System.out.println
            ((y*n + x == m)+" in loop "+y);
    }
    System.out.println((x < n)+" out of loop");
    return y;
}
```

И мы также можем напечатать условие *x* меньше *n* после выхода из цикла.

После проверки правильности нашего алгоритма, мы можем удалить лишний код.

Также мы можем ввести специальную переменную, чтобы указывать, ведем ли мы сейчас режим отладки, или нет.

```
int intDiv (int m, int n){

    int x=m; int y=0;
    while (x>=n){
        x=x-n; y=y+1;
        if (tm){System.out.println(y*n + x == m);}
    }
    if (tm){System.out.println(x < n)};
    return y;
}
```

Теперь вопрос: можем ли мы автоматизировать обнаружение ошибок еще больше?

Иметь тестовый режим – это хорошая идея, которая позволяет нам повысить нашу уверенность в правильности программы.

Если мы хотим протестировать программу, мы установим переменную *tm* в *true*, а для нормальной работы мы устанавливаем значение переменной *false*.

Но тестирование настолько важно, что Java имеет специальное утверждение *assert*, которое позволяет нам проверять условия при работе в специальном режиме, который используется для тестирования.

Предположим, мы хотим проверить условие цикла и условие после выхода из цикла.

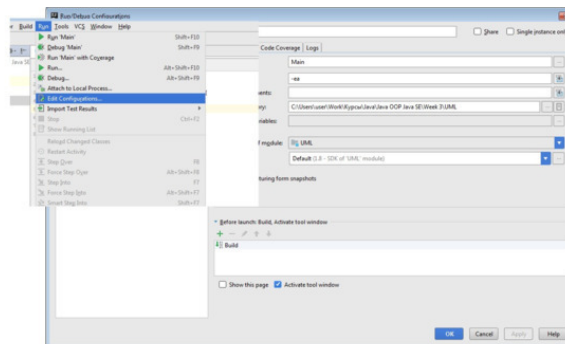
```
int intDiv (int m, int n){
    int x=m; int y=0;
    while (x>=n){
        x=x-n; y=y+1;
        assert y*n + x == m;
    }
    assert x < n;
    return y;
}
```

Но вместо добавления печати мы добавим утверждение `assert`.

В обычных условиях эти утверждения `assert` не выполняются: как будто их там нет.

И если вы хотите включить утверждения во время выполнения, вы должны соответствующим образом настроить среду выполнения.

В IDEA – это настройка конфигурации запуска, где вы должны указать опцию `—ea`.



Если утверждения включены и утверждение не выполняется, возникает ошибка утверждения.

Здесь мы видим другой пример, в котором мы использовали оператор `assert`.

```
if (a&&b) {...}
else if (!b) {...}
else assert !a&&b; {...}
```

`a` и `b` являются булевыми переменными, и у нас есть вложенные операторы `if`.

Сначала проверяем условие `a` и `b` и выполняем что-то, если это правда.

В противном случае, если `a`, либо `b` являются ложными, у нас есть другой оператор `if`, где мы определяем оператор для выполнения, если `b` является ложным.

Но теперь, что осталось для последнего выражения?

По какому условию выполняется это выражение?

Оно выполняется, когда *a* является ложным и *b* истинным.

Все остальные случаи рассматриваются ранее.

Это легко проверить, но мы, возможно, хотим включить на всякий случай утверждение `assert`.

Здесь у нас есть рекурсивно определенный метод, который вычисляет *x* в степени *y*.

```
int power (int x, int y) {
    assert y>=0;
    if (y==0) return 1;
    else     return x*power(x,y-1);
}

int power (int x, int y) {
    if (y<=0) return 1;
    else     return x*power(x,y-1);
}
```

И мы знаем, что это работает только тогда, когда *y* равно нулю или больше нуля.

Для отрицательного *y*, мы получим бесконечный цикл.

Вместо того, чтобы писать комментарий, мы могли бы написать утверждение `assert`, чтобы проверить, выполнили ли мы условие не отрицательности во время выполнения.

Но это плохое решение, так как эта проверка выполняется только в специальном тестовом режиме.

Если мы хотим действительно убедиться, что мы не сталкиваемся с неокончательным случаем, мы лучше изменим условие в коде.

Обратите внимание, что вы можете писать утверждения `assert` во многих случаях, когда вы ранее писали комментарии с булевыми выражениями.

• Comment	<code>/* n>=0 */</code>
– Human understanding	
• Javadoc comment	<code>/** n>=0 */</code>
– Generation of documentation	
• Assertion	<code>assert n>=0</code>
– Check during program development	
• Code	<code>if (n>=0)</code>
– Actual program	

Но вы должны хорошо понимать разницу.

Утверждение обрабатывается средой выполнения только в специальном режиме во время разработки.

Отладка кода



Объектно-ориентированное программирование на Java

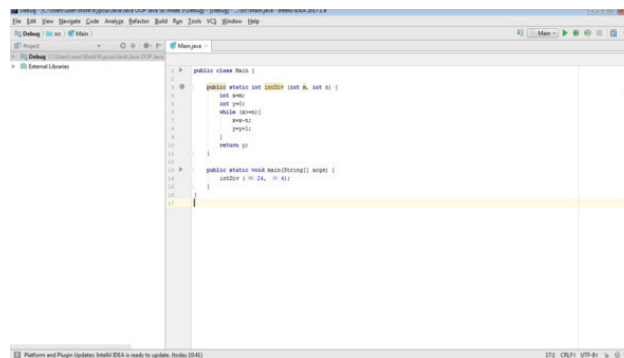
Отладка и тестирование кода

Лекция 3

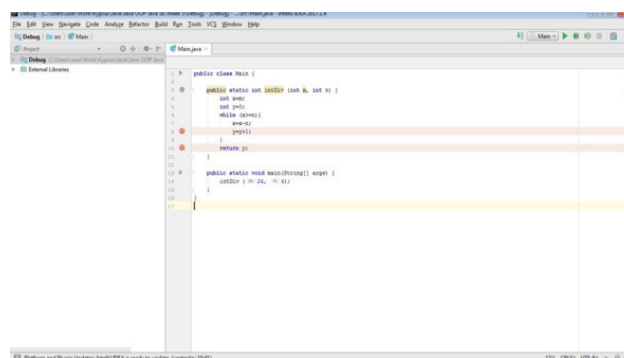
Отладка кода

Чтобы продемонстрировать отладку программы в IDEA, создадим проект приложения.

Чтобы начать отладку программы, в первую очередь вам нужно разместить точку останова в выражении, где вы хотите приостановить выполнение вашего приложения.



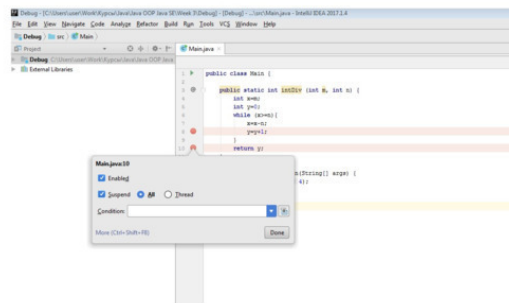
Это делается простым нажатием левой кнопки мыши в левом поле редактора кода, где проставлена нумерация строк кода.



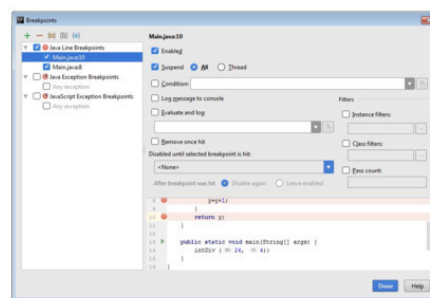
Если вы наведете указатель мыши на точку останова, вы увидите ее свойства в подсказке.

Если вы хотите изменить свойства точки останова, щелкните на ней правой кнопкой мыши и откроется диалоговое окно свойств точки останова.

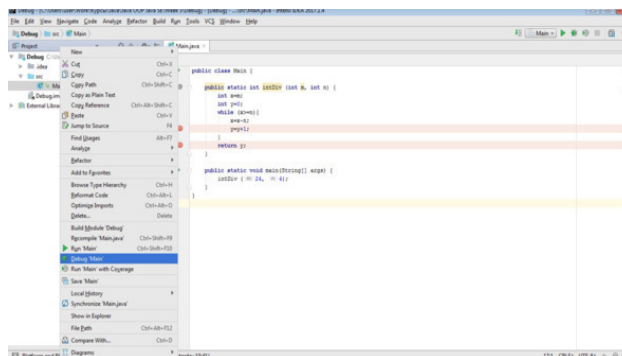
Если вы хотите посмотреть все доступные свойства точки останова и увидеть ее местоположение среди других точек останова, нажмите **Ctrl + Shift + F8**.



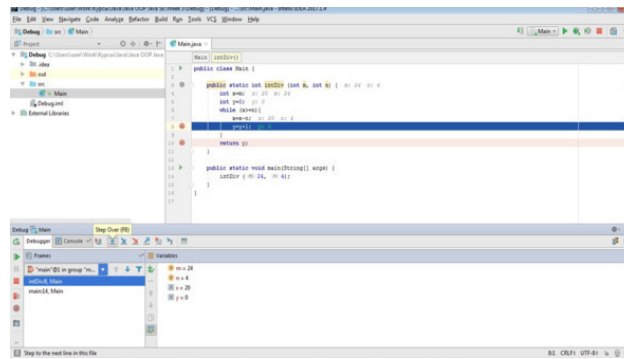
После добавления точек остановки, можно запустить отладку.



IntelliJ IDEA откомпилирует ваше приложение, а затем приостановит приложение в первой точке остановки.



При этом будут видны все значения переменных и используя кнопку Step Over можно продвигаться шаг за шагом по выполнению приложения и наблюдать изменение значений переменных.



Команда Force Step Into позволяет вам войти непосредственно в метод класса, например, в стандартный класс Java.

Команда Force Step Over позволяет вам перепрыгнуть через вызов метода, игнорируя точки останова.

Команда Force Run to Cursor позволяет вам перейти к позиции курсора, игнорируя существующие точки останова.

Тестирование кода



Объектно-ориентированное программирование на Java

Отладка и тестирование кода

Лекция 4

Тестирование кода

Предположим, вы написали программу, но уверены ли вы, что она работает правильно? Там могут быть ошибки кодирования.

И причиной ошибок может быть неправильное понимание задачи для решения, или плохая практика кодирования, или плохое сотрудничество с другими разработчиками, работающими над одним и тем же проектом, и так далее.

Одним из способов найти ошибки в коде, заключается в проверке результата для определенных входных данных против ожидаемого результата.

Предположим, у вас есть спецификация для программы, которую вы хотите написать, скажем, вы хотите написать функцию квадрата любого заданного числа.



Если у вас на входе 2, результат должен быть 4.

Если у вас на входе 3, результат должен быть равен 9.

Затем вы пишете программу, которая следует этой спецификации.

Итак, ваша программа должна вернуть 4, если у вас на входе 2.

Как вы знаете, что ваша программа верна?

Просто проверьте программу, чтобы увидеть, является ли результат ожидаемым.

Дайте ей 2 и посмотрите, вернет ли она 4.

Хотя этого будет недостаточно, вы должны попробовать также числа 3, 4, 5 и ноль и так далее, а также 1 и отрицательное число.

И если числа могут быть числами с плавающей запятой, вам придется продолжить проверку с числами с плавающей запятой в качестве входных данных.

Таким образом, это будет довольно большой набор тестовых примеров.

И даже с самым быстрым компьютером у нас не хватит всей нашей жизни, чтобы проверить эту простую программу на всех возможных входных данных.

И ситуация станет намного хуже, если у нас есть несколько входных данных.

Таким образом, возникает вопрос как мы можем создать сокращенный набор тестовых примеров, который достаточно мал для выполнения в разумные сроки, но достаточно большой, чтобы дать нашу уверенность в нашем коде?

Возможно, смотреть на поведение ввода-вывода недостаточно.

Мы должны посмотреть на код более подробно, и, в частности, посмотреть на все возможные пути выполнения программы.

Мы должны понять все возможные последовательности выражений при выполнении при разных значениях условий в условных выражениях и циклах.

Эти условия могут принимать разные значения, истинные или ложные, и поэтому мы можем получить различные последовательности выражений.

Также, мы должны попробовать тщательно подобранный набор возможных входных значений, чтобы проверить, соответствуют ли результаты нашим ожиданиям.

Таким образом, в принципе есть два подхода к тестированию.

Одним из них является функциональный подход, который проверяет поведение ввода-вывода, не глядя на код.

Этот подход также называется тестированием черного ящика, потому что мы не заглядываем внутрь.

Другой подход – это структурный подход, где смотрят на код.

Кроме того, мы различаем unit тестирование или модульное тестирование от интеграционного тестирования.

Unit тестирование относится к тестированию одного блока исходного кода.

Эта единица может быть просто методом или полным классом.

Интеграционное тестирование относится к тестированию нескольких таких блоков как группы.

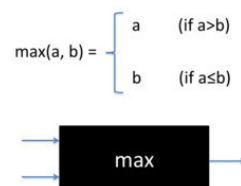
Единица может показывать правильное поведение, если она была протестирована независимо, но показать некорректное поведение в сочетании с группой.

Таким образом, наша цель – найти хороший набор тестовых примеров, достаточно небольшой, чтобы быть приемлемым, но достаточно большой, чтобы убедить нас в том, что наша программа верна.

Сначала, давайте рассмотрим функциональное тестирование, или тестирование черного ящика, где мы не рассматриваем код.

Поскольку мы не можем проверить все возможные входные значения, давайте классифицируем возможные входные значения на группы, для которых можно ожидать аналогичное поведение.

Здесь у нас программа для вычисления максимума двух целых значений.



Результат должен быть `a`, если `a` больше `b`, и `b`, если `a` меньше `b`.

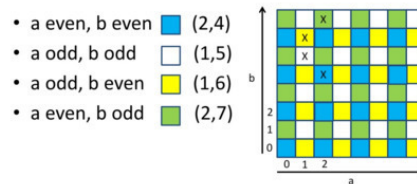
Заметим, что во втором случае, в условии есть равенство обоих значений.

Мы могли бы включить его в первом случае.

Теперь, если a и b являются целыми значениями, мы не можем протестировать все целые числа, и как мы могли бы сгруппировать эти числа для проверки нашего черного ящика?

Мы можем сгруппировать эти числа a и b на четные или нечетные.

Таким образом, мы получим четыре набора значений.



И давайте определим четыре пары (a, b) значений, как представителей этих четырех наборов.

Получим результаты для этих четырех пар и сравним с ожидаемыми результатами.

Можем ли мы сделать вывод, что наша программа правильная?

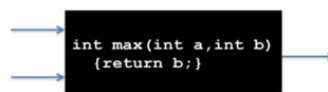
(a,b)	Result obtained	Result expected
(2,4)	4	4
(1,5)	5	5
(1,6)	6	6
(2,7)	7	7



Но давайте посмотрим под капот этой программы.

Здесь нас ждет сюрприз! Мы всегда возвращаем второй аргумент, b .

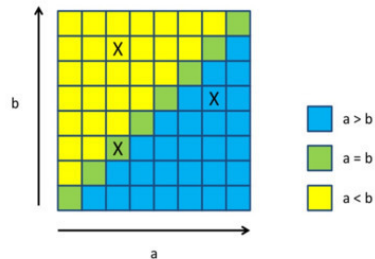
(a,b)	Result obtained	Result expected
(2,4)	4	4
(1,5)	5	5
(1,6)	6	6
(2,7)	7	7



Просто случается, что для выбранных входных значений, b совпадает с максимумом, поэтому такая группировка входных значений недостаточная.

Давайте посмотрим на другую группировку входных значений.

Теперь мы разделим эти значения на три группы – a меньше чем b , a равно b , и a больше, чем b .



И мы выбираем три тестовых значения, по одному из каждой группы.

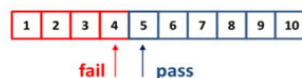
Теперь наша предыдущая программа не справляется с выполнением этого теста, и мы обнаружили, что программа работает неверно.

Таким образом, разбивка набора возможных входных значений является способом для тестирования черного ящика, но мы должны выбирать классификацию входных значений осторожно.

Программисты часто делают ошибки на границах возможных значений, в местах, где есть разрывы значений.

Если, например, у вас есть оценки от одной до десяти, и нужно решить, выполнил ли студент задание, предполагая, что при значении менее пяти, студент не выполнил задание, а при пяти или выше студент выполнил задание.

$0 < \text{grade} < 5 \rightarrow \text{fail}$
 $5 \leq \text{grade} \leq 10 \rightarrow \text{pass}$

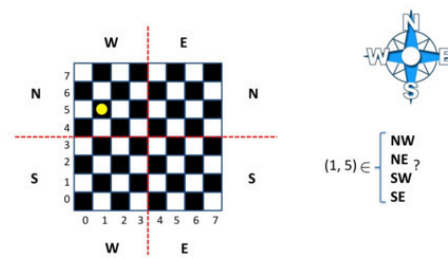


Что в этом случае будет хорошим тестом?

В этом случае тестировать нужно значения, которые находятся на границах; в этом случае четыре и пять.

Но давайте сделаем задачу немного сложнее.

Рассмотрим пример, где есть две переменные.



Представьте, что вы смоделировали шахматную доску, и что в вашей программе важно знать, в каком из четырех квадрантов находится значение.

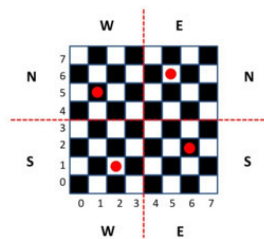
Мы знаем, что, если y больше трех, мы находимся в одном из северных квадрантов, и, если y меньше или равно двум, мы находимся в одном из южных квадрантов.

Аналогично, если x больше трех, мы находимся на востоке, и, если x не более трех, мы находимся на западе.

Какие значения нужно протестировать для этой программы.

Один из подходов – это проверить каждую из четырех зон.

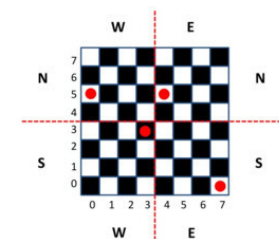
Например, мы могли бы взять четыре значения, показанные здесь, где каждая точка находится в другой зоне.



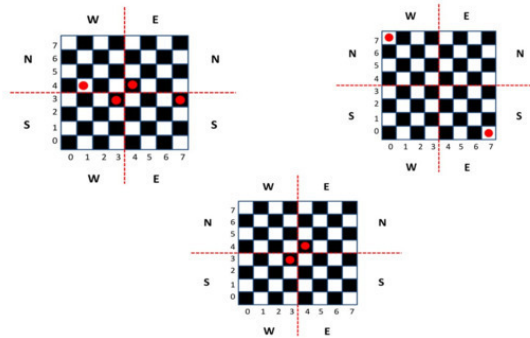
Но мы говорили, что ошибки кодирования часто происходят на границах.

Таким образом, вместо этого мы могли бы взять четыре значения, которые расположены на вертикальных границах.

Или, дополнительно, мы могли бы взять четыре значения на горизонтальных границах и по краям.



Таким образом, этот подход к тестированию заключается в том, чтобы установить граничные случаи и написать тестовую программу именно для проверки этих случаев.



Рассмотрим теперь структурный подход к тестированию.

В этом случае мы подробно смотрим на код.

Рассмотрим эту простую программу.

```
int max(int a,int b){
    int m=0;
    if (a<b) m=b;
    else    m=a;
    return m;
}
```

Она вычисляет максимум двух чисел.

И она имеет два целочисленных аргумента a и b.

Если a меньше b, программа возвращает b, в противном случае программа возвращает a.

Есть два пути, которым может следовать программа.

Для заданных a и b, либо a меньше, чем b и программа присваивает b значению m и возвращает m.

Или, условие, a меньше b, не выполняется, и в этом случае мы просто выполняем случай else.

Как только мы определили возможные пути выполнения программы, мы хотим иметь входные значения для этих двух путей, вместе с ожидаемыми результатами.

Здесь мы их видим.

<pre>int max(int a,int b){ int m=0; if (a<b) m=b; else m=a; return m; }</pre>			<pre>int max(int a,int b){ int m=0; if (a<b) m=b; else m=a; return m; }</pre>		
a	b	Expected result	a	b	Expected result
1	2	2	1	1	1

Давайте посмотрим на более сложный случай.
Здесь у нас есть цикл while и внутри него условие.

```
double my_sqrt(double n, double epsilon){
    double low, high, mid, oldmid;
    low=0; high=mid=n; oldmid=-1;
    while (Math.abs(oldmid-mid)>=epsilon){
        oldmid=mid; mid=(high+low)/2;
        if (mid*mid>n) {high=mid;}
        else {low=mid;}
    }
    return mid;
}
```

В этом примере, какие случаи мы должны проверить?

Первой возможностью было бы совсем не входить в цикл.

Поэтому мы должны найти тестовый пример, который делает это условие while ложным.

Затем предположим, что вы входите один раз в цикл.

Здесь у нас есть две возможности, если выполняется условие if, или выполняется условие else.

Еще одной возможностью было бы войти дважды в цикл, после выполнения условия if и после выполнения условия else.

Теперь, есть еще один интересный случай, когда у нас есть несколько вложенных циклов.

Идея здесь состоит в том, чтобы двигаться из самого внутреннего цикла наружу.

Проведите тесты для самого внутреннего цикла, удерживая внешние циклы при минимальных значениях параметров итерации.

Добавьте тесты для значений вне диапазона или исключенных значений.

После этого двигайтесь в следующий наружный цикл, и так, пока не пройдете все циклы.

Это был структурный подход к тестированию, в котором мы просматриваем код и пытаемся следовать как можно большему количеству путей выполнения программы, чтобы повысить нашу уверенность в правильности кода.

И для установленных путей необходимо определить соответствующие входные значения для тестирования.

Модульное тестирование



Объектно-ориентированное программирование на Java

Отладка и тестирование кода

Лекция 5

Модульное тестирование

При тестировании программы сначала тестируются отдельные единицы или unit.

Unit могут быть методами или целыми классами.

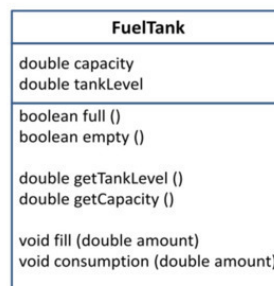
И это называется модульным или unit тестированием.

Затем мы тестируем эти единицы в более широком контексте, группами

И это называется интеграционным тестированием.

Но давайте сначала сосредоточимся на модульном тестировании.

Пусть этот класс является единицей тестирования.

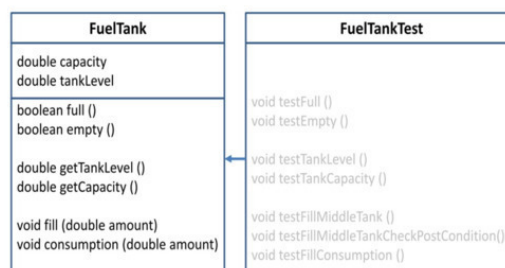


Здесь мы не даем полной реализации Java.

Мы просто показываем UML диаграмму класса.

Чтобы протестировать этот класс, мы определяем другой класс.

Обычно название этого другого класса получается путем добавления слова «test» в конце имени тестируемого класса.



В этом классе мы добавляем ряд методов, чтобы протестировать методы тестируемого класса.

Давайте начнем с метода `getTankLevel` и давайте напишем метод класса `FuelTankTest`, который называется `testTankLevel`.

```
public void testTankLevel(){
    FuelTank tank=new FuelTank(60,10);
    System.out.println("Tank level expected:
    10 liters. Value obtained: " +
    tank.getTankLevel());
}
```

Это метод для теста уровня бака.

Мы тестируем ожидаемый уровень бака относительно фактического уровня бака.

Следующий метод для тестирования метода `full ()`.

```
public void testFullTank(){
    FuelTank tank=new FuelTank(60, 60);
    System.out.print("The tank should be full,
    the test says that the tank is: ");
    System.out.println(tank.full());
}
```

Здесь мы устанавливаем полный бак и печатаем, что он должен быть полным, и какой результат метода `full` для сравнения.

Давайте теперь напишем тестовый метод для метода `fill ()` заполнения бака.

```
/*
 * fill: add petrol to the fuel tank
 *
 * @param amount: amount of fuel to add in liters
 *
 * precondition 0.0<amount<=getCapacity()-getTankLevel()
 * postcondition not empty
 * postcondition tankLevel>tankLevel_initial
 */
public void fill(double amount){
    tankLevel = tankLevel+amount;
}
```

Здесь такая же идея – мы устанавливаем бак емкостью шестьдесят литров и десять литров в баке, и используем метод заполнения, чтобы добавить двадцать литров.

```

public void
testFillMiddleTankCheckPostCondition(){
    int initTank=10;
    FuelTank tank=new FuelTank(60,initTank);
    tank.fill(20.0);
    boolean postCondition=tank.getTankLevel()>initTank;
    System.out.println("Expecting
    tankLevel>tankLevel_initial: true. Obtained: " +
    postCondition);
}

```

Затем мы печатаем ожидаемое и полученное значения.

Вспомним, что мы можем только заполнить бак до определенной емкости.

```

public void testFillConsumption (){
    FuelTank tank=new FuelTank(60,10);
    tank.fill(50.0);
    System.out.println("Tank level expected: 60
    liters. Obtained: " + tank.getTankLevel());
    System.out.println("Tank expected to be full.
    Obtained: " + tank.full());
    tank.consumption(20.0);
    System.out.println("Tank Level expected: 40
    liters. Obtained: " + tank.getTankLevel());
    System.out.println("Tank expected not to be
    full. Obtained: " + !tank.full());
}

```

Это согласуется с предварительным условием, величина, которую можно добавить, меньше или равна емкости бака за вычетом уровня бака.

Здесь мы могли бы написать несколько тестов.

Тестирование выполняется не только после завершения кодирования.

Тестирование также выполняется во время разработки программы.

Вы немного кодируете, затем вы немного тестируете, затем вы кодируете дальше, после этого тестируете и т. д.

Поэтому хорошо было бы иметь некоторую автоматизированную поддержку для такого тестирования.

И такой инструмент для автоматического тестирования у нас имеется.

Давайте посмотрим, как мы можем автоматизировать модульное тестирование.

JUnit – это Java-фреймворк, который помогает нам автоматизировать модульное тестирование.

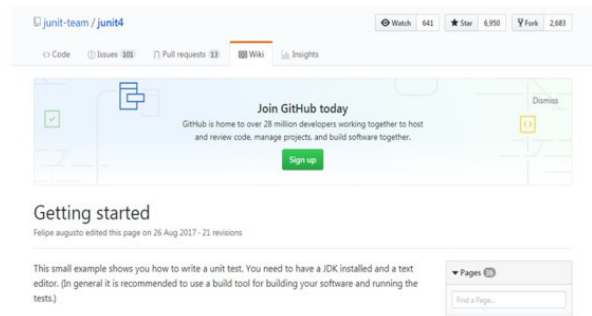
Фреймворк отличается от просто библиотеки тем, что он определяет каркас программы.

А платформа отличается от фреймворка тем, что она еще и предоставляет среду выполнения для каркаса.

JUnit помогает нам писать тесты, а также запускать их.

Мы будем использовать фреймворк JUnit 4.

Здесь вы видите вверху класс, который мы хотим проверить.



Это класс, моделирующий простой калькулятор со статическим целочисленным результатом и четырьмя методами.

Ниже, вы видите тестовый класс для класса Calculator, который использует фреймворк JUnit.

Здесь показаны только два метода, один для проверки метода сложения и один для проверки метода вычитания.

Во-первых, перед каждым из этих двух методов имеется аннотация JUnit @Test, которая помечает методы как методы тестирования.

Во-вторых, в каждом из методов мы создаем контролируруемую среду.

Это означает, что мы переходим в состояние, в котором мы знаем, чего ожидать.

Это называется испытательный стенд.

В нашем примере мы создаем новый объект класса Calculator.

Затем мы вызываем метод add или subtract.

После этого, мы создаем утверждение assert.

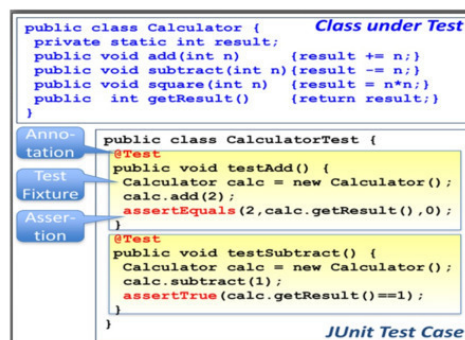
Это утверждения JUnit assert, в которых мы проверяем, совпадают или нет полученные значения с ожидаемыми.

Теперь давайте рассмотрим эти концепции, эти понятия, одно за другим.

Ранее мы познакомились с утверждением Java assert для проверки условий.

Фреймворк JUnit вводит дополнительные утверждения.

Вот некоторые из утверждений, доступных в JUnit.



Например, assertTrue проверяет, истинно ли заданное логическое условие.

Аналогично, assertFalse проверяет, является ли оно ложным.

В противном случае, выдается сообщение об ошибке.

Утверждение assertEquals проверяет, совпадает ли фактическое значение с ожидаемым.

И это равенство понимается в контексте определенной точности, дельты, которая предварительно задается.

Утверждение `assertNull` проверяет, является ли объект нулевым.

В этих утверждениях можно указать первый необязательный аргумент с выдаваемым в случае ошибки сообщением.

В JUnit также много других утверждений, но идея такая же.

Испытательный стенд – это то, что позволяет нам последовательно тестировать нечто в контролируемых условиях.

Например, при тестировании какого-либо физического устройства, стенд удерживает это устройство в контролируемом положении.

Аналогично, при тестировании программного обеспечения, тестовый стенд ставит проверяемый код в известное состояние, для его проверки воспроизводимым способом.

Таким образом, последовательность событий в тесте обычно всегда одна и та же:

Сначала вы получаете эту контролируемую среду, установив тестовый стенд.

В примере с калькулятором – это просто создать новый объект калькулятора.

Затем мы взаимодействуем с методом, который мы хотим протестировать.

Затем мы проверяем, получено ли ожидаемое значение.

Это делается с помощью утверждения.

И, наконец, может потребоваться убрать тестовый стенд, если это необходимо для восстановления некоторого начального состояния.

Теперь эта установка и удаление тестового стенда может быть общим для нескольких методов.

Фреймворк JUnit помогает нам в этом.

Здесь у нас есть метод `setUp`, которому предшествует аннотация `@Before`.

Assertion	Description
<code>assertTrue([message,] condition)</code>	Tests whether condition is true
<code>assertFalse([message,] condition)</code>	Tests whether condition is false
<code>assertEquals([message,] expected, actual, delta)</code>	Tests whether values are equal with a given precision
<code>assertNull([message,] object)</code>	Tests whether object is null

Этот метод выполняется до каждого из методов тестирования этого класса.

Мы могли бы также включить метод с аннотацией `@After`, чтобы убрать тестовый стенд после выполнения каждого из методов тестирования.

Теперь одно важное предостережение.

Мы не можем предполагать какого-либо порядка при выполнении тестовых методов.

Они могут быть выполнены в любом порядке.

Поэтому, мы должны позаботиться о создании правильного состояния для выполнения каждого теста.

Таким образом, у нас есть аннотация `@Test`, указывающая, что следующий за аннотацией метод является методом тестирования.

```
public class CalculatorTest {
    Calculator calc = null;
    @Before
    public void setUp() throws Exception {
        calc = new Calculator();
    }
    @Test
    public void testAdd() {
        calc.add(2); assertEquals(2, calc.getResult(), 0);
    }
    @Test
    public void testSubtract() {
        calc.subtract(1); assertTrue(calc.getResult()==1);
    }
}
```

Аннотация `@Before` маркирует метод, который должен быть запущен перед каждым из методов тестирования.

И существует также аннотация `@After`, которая маркирует метод, выполняемый после каждого из методов тестирования.

Она служит для освобождения тестового стенда.

Аннотации `@BeforeClass` и `@AfterClass` маркируют методы, которые выполняются только при входе и выходе, соответственно, из тестового класса.

Теперь, нужно ли писать тестовый метод для каждого возможного значения?

Annotations

```
@Test
@Before
@After
@BeforeClass
@AfterClass
```

Мы можем заранее определить наборы входных значений с ожидаемыми выходными значениями в параметризованных тестах.

Здесь мы задаем входные значения вместе с ожидаемыми выходными значениями для метода вычисления квадрата числа.

Чтобы вызвать ошибку, давайте напишем число семнадцать для квадрата четырех.

В результате мы получим, что общий тест потерпит неудачу.

Для тестового примера со входом четыре мы увидим, что обнаружено несоответствие.

Что, если мы хотим сделать не только один такой, но и набор тестов?

Мы можем объединить тесты в тестовый набор.

Parametrized Tests

```
public class ParameterizedSquareTest {
    private int param; private int result;
    @Parameters
    public static Collection<Object[]> squareNumbers() {
        Object[][] numbers=new Object[][]{{2,4},{3,9},{4,17}};
        return Arrays.asList(numbers);
    }
    public ParameterizedSquareTest(int param, int result) {
        this.param = param; this.result = result;
    }
    @Test
    public void testSquare() {
        calc.square(param); assertEquals(result, calc.getResult(), 0);
    }
}
```

Test	Input	Expected output
0	2	4
1	3	9
2	4	17

Тестовый набор представляет собой набор тестов для нескольких классов.

Каким образом запускаются JUnit тесты.

Фреймворк JUnit предоставляет классы Java, которые запускают тесты.

Они называются runners.

Тест обычно запускается в JUnit с классом по умолчанию.

Test Suites

```
@SuiteClasses({
    CalculatorTest.class,
    ParameterizedSquareTest.class
})
```

Однако вы можете изменить это поведение по умолчанию.

Здесь один runners – класс Suite, который позволяет нам запускать несколько тестовых классов, собранных вместе в тестовом наборе.

А другой – это Parameterized параметрайзд runner, который позволяет нам запускать параметризованные тесты.

Для создания тестов в IDEA в первую очередь нужно добавить библиотеку JUnit в путь приложения.

Для этого есть два способа.

Первый способ навести курсор на имя класса в коде, который нужно тестировать, и нажать Alt+Enter, после этого выбрать Create Test.

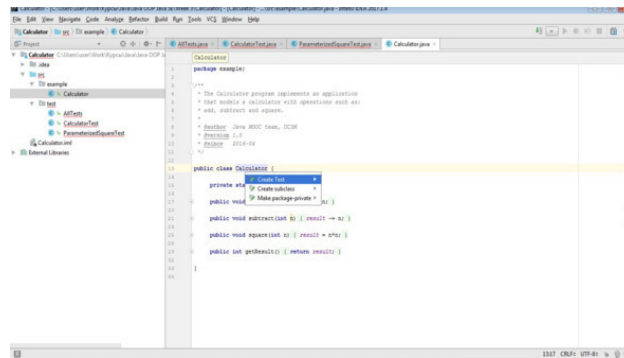
Runner

```
@RunWith(JUnit4.class) (default)

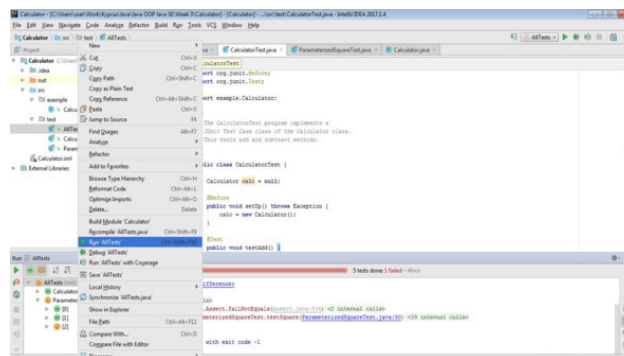
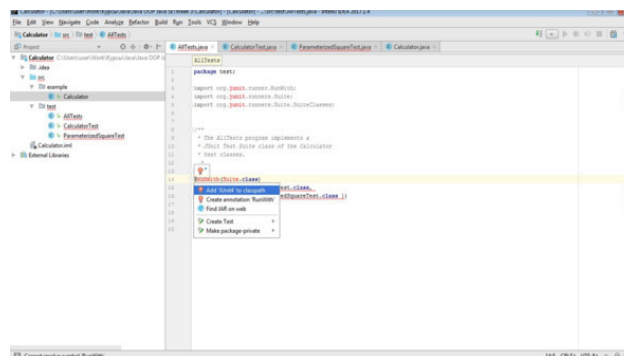
@RunWith(Suite.class)

@RunWith(Parameterized.class)
```

Или вручную создать тестовый класс с аннотациями.
И исправить ошибку, добавив библиотеку JUnit в путь программы.



Для запуска теста нажать правой кнопкой мышки на тестовом классе и выбрать Run.



В результате будет запущен тест и откроется окно, в котором вы можете посмотреть результаты прохождения теста.

Интеграционное тестирование



Объектно-ориентированное программирование на Java

Отладка и тестирование кода

Лекция 6

Интеграционное тестирование

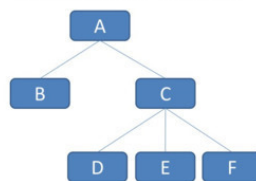
Теперь, мы протестировали все units и теперь мы хотим знать, хорошо ли они работают все вместе.

Это называется интеграционным тестированием.

Давайте посмотрим, какие у нас есть варианты.

Обычно у нас есть иерархия, подобная той, которая здесь показана.

Integration Testing



Мы имеем самый верхний класс, который использует классы B и C, а затем класс C использует классы D, E и F, например.

Теперь есть несколько подходов к интеграционному тестированию.

Два основных способа тестирования – это тестировать сверху вниз и снизу вверх по иерархии.

Или можно тестировать комбинацией сверху вниз и снизу вверх.

Или еще один способ – выбрать сначала класс, который является наиболее рискованным, или который труднее всего протестировать.

Когда мы тестируем сверху вниз, мы сначала тестируем класс A, самый верхний класс.

Учтите, что остальные классы, возможно, еще не были закодированы.

Поэтому, для тестирования класса A, мы используем упрощенные версии классов B и C, которые выступают в качестве их представителей.

Их иногда называют заглушками.

Затем мы тестируем класс B с заглушкой для C.

И затем мы тестируем класс C с заглушками для классов D, E и F;

И затем мы тестируем классы D, E и F в любом порядке.

И, наконец, мы тестируем всю систему вместе.

Тестирование «снизу вверх» проще.

Сначала мы тестируем классы D, E и F независимо, в любом порядке.

Затем мы тестируем класс C, зная, что классы D, E и F уже были протестированы.

Затем мы тестируем класс B.

А затем мы тестируем класс A, и тестируем всю систему.

Интеграционное тестирование важно, поскольку при взаимодействии разных частей системы могут возникать сбои.

Теперь, если мы напишем несколько тестовых примеров, мы можем спросить себя: все ли возможности мы проверили?

Или мы забыли рассмотреть часть кода? или какой-то путь кода?

Покрытие кода – это показатель того, сколько частей кода было протестировано.

Здесь существует несколько возможных критериев.

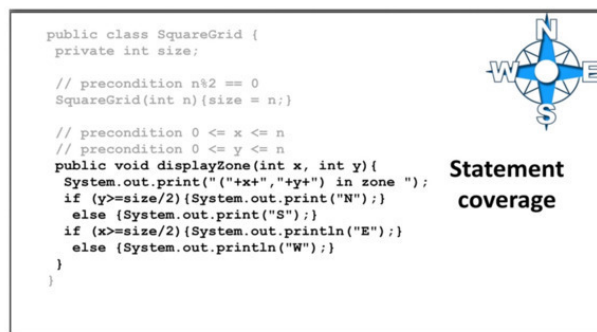
Покрытие методов – это были ли все методы протестированы.

Покрытие выражений – это охватываются ли все выражения тестом, или только их часть.

Покрытие ветвей относится к ветвям, которые могут выполняться программой.

И покрытие условий аналогично покрытию ветвей, но оно относится ко всем логическим условиям, как истинным, так и ложным значениям, которые могут быть протестированы.

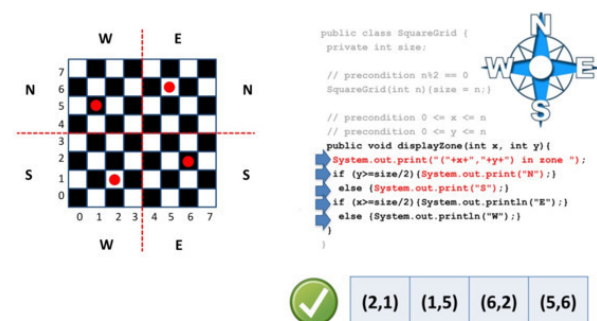
Давайте рассмотрим этот пример, где мы печатаем в котором из четырех квадрантов шахматной доски, находилась шахматная фигура.



Здесь у нас есть метод, и мы хотим организовать покрытие выражений.

У нас есть пять выражений печати.

И давайте проверим эти четыре позиции, и посмотрим, выполняются ли все выражения хотя бы один раз.



Теперь, для позиции (2,1) выполняется первое выражение печати и выражения условия else.

Т.е. три выражения.

Смотрим для других позиций и видим, что у нас есть стопроцентное покрытие выражений.

С четырьмя тестами мы также имеем стопроцентное покрытие методов, а также ветвей и условий.

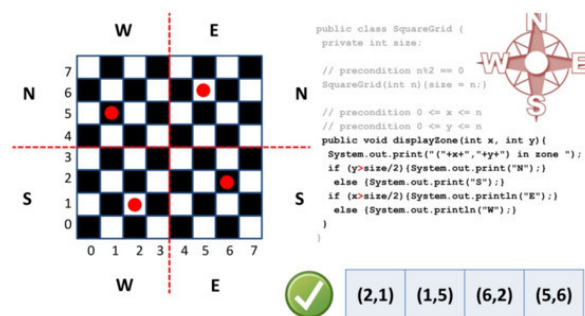
Но что это значит?

Является ли это гарантией правильности кода?

Нет!

Мы можем иметь стопроцентное покрытие кода и, тем не менее, получить ошибку.

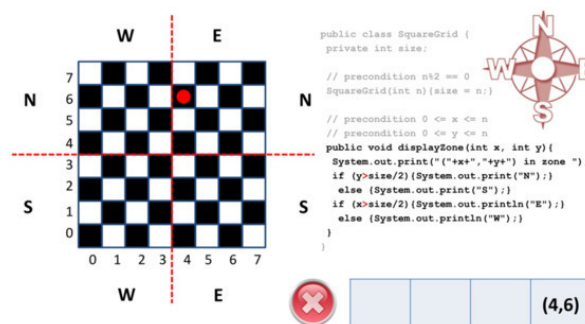
Представьте себе, что мы заменяем два оператора больше или равно, просто оператором больше, что делает код неправильным.



И с теми же четырьмя значениями мы получаем стопроцентное покрытие кода, и ожидаемые результаты также верны во всех четырех случаях.

Таким образом, стопроцентное покрытие кода и правильные результаты не дают гарантию правильности.

Мы могли бы обнаружить ошибку, если бы мы проверили позицию (4,6).

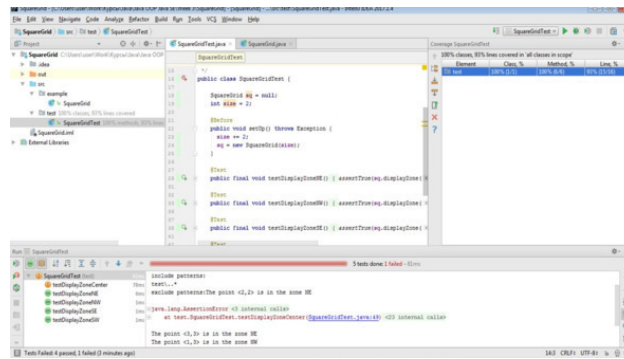


Результат был бы запад, а не правильный восток.

Таким образом, если у нас нет стопроцентного покрытия кода, мы знаем, что некоторые части кода остались непроверенными, и мы должны попробовать дальше.

Однако, со стопроцентным покрытием, у нас все равно нет гарантии правильности кода.

Для запуска тестирования с покрытием кода в IDEA откроем проект SquareGrid и нажмем правой кнопкой мышки на тестовом классе.



Выберем Run With Coverage.

В результате выполнения теста будет открыто окно, где будет указан процент покрытия кода на разных уровнях – класса, методов и строк кода.

При написании кода, может возникнуть ситуация, когда вам нужно сделать одно и то же вычисление в двух или более разных местах.

Таким образом, ваш код может иметь повторяющиеся части.

Посмотрите на этот код.

```
public static void main(String[] args){
    float arrayA[] = {1,2,3,4}; float arrayB[] = {1,3,5};
    float sum, average;

    sum = 0;
    for (int i=0; i<arrayA.length; i++){sum+=arrayA[i];}
    average = sum/arrayA.length;
    System.out.println("The average of arrayA is "+average);

    sum = 0;
    for (int i=0; i<arrayB.length; i++){sum+=arrayB[i];}
    average = sum/arrayB.length;
    System.out.println("The average of arrayB is "+average);
}
```

У вас есть два массива: arrayA и arrayB.

И вы хотите вычислить среднее значение для каждого из массивов.

Для arrayA вы сначала инициализируете переменную суммы в нуль, затем накапливаете в цикле for все элементы массива A в переменной суммы и делите ее на длину массива.

Затем вы можете распечатать это значение.

Для массива arrayB мы делаем то же самое, инициализируем сумму, накапливаем в ней элементы, делим на длину массива и печатаем это значение.

Понятно, что оба фрагмента кода идентичны, кроме имени массива.

Но если вы обнаружите ошибку, вы должны исправить ее дважды.

Или, если вы хотите что-то изменить, вам нужно сделать это дважды.

Разумное преобразование кода будет – это определить метод, который вычисляет среднее значение указанного массива, и затем вызов этого метода дважды, для каждого из массивов.

Здесь вы видите внизу два оператора печати, где вызывается определенный метод average.

```

public static void main(String[] args){
    float arrayA[] = {1,2,3,4}; float arrayB[] = {1,3,5};

    System.out.println("The average of arrayA is "+average(arrayA));
    System.out.println("The average of arrayB is "+average(arrayB));
}

public static float average(float[] array){
    float sum = 0;
    for (int i=0; i<array.length; i++){sum += array[i];}
    return sum/array.length;
}

```

Но теперь, мы должны дважды написать выражение печати.
Должны ли мы включить эту часть кода в метод?
Как здесь.

```

public static void main(String[] args){
    float arrayA[] = {1,2,3,4}; float arrayB[] = {1,3,5};

    averagePrint(arrayA, "arrayA");
    averagePrint(arrayB, "arrayB");
}

public static void averagePrint(float[] array, String n){
    float sum = 0;
    for (int i=0; i<array.length; i++){sum += array[i];}
    System.out.println("The average of "+n+" is "+ sum/array.length);
}

```

Теперь у нас есть метод `averagePrint`, который вычисляет среднее значение указанного массива, а также выводит его.

Теперь у метода есть два аргумента – сам массив и имя массива.

Вопрос – что лучше? Иметь метод, который просто вычисляет среднее значение массива, или тот, который выполняет две вещи: вычисляет среднее значение и печатает его?

Если у вас есть метод, который просто вычисляет среднее значение элементов массива, мы могли бы использовать этот метод для вычисления чего-то еще, например, стандартного отклонения.

Возможно, этот метод стандартного отклонения не хочет печатать среднее значение массива, каждый раз, когда он вычисляется. Или мы хотим напечатать что-то еще.

```

public static float standardDeviation(float[] array){
    float mean = average(array);
    float squareSum = 0;
    for (int i=0; i<array.length; i++){
        squareSum += Math.pow(array[i]-mean, 2);
    }
    return Math.sqrt(squareSum/(array.length-1));
}

```

Таким образом, если определенный метод выполняет только одну задачу, его проще повторно использовать.

Это приводит нас к концепции связанности Cohesion когезии.

Когезия определяется числом и разнообразием задач, которые выполняет метод.

Метод должен отвечать за одну и только одну четко определенную задачу.

Мы называем это высокой степенью когезии.

Метод с высокой степенью когезии легче понять, изменять и поддерживать, а также повторно использовать его.

Дублирование кода не является хорошей практикой.

Если вы захотите что-то изменить, вам нужно будет сделать это дважды.

Определите метод, если вы собираетесь использовать один и тот же код как минимум дважды.

Определенные методы, кроме того, не должны делать сразу несколько вещей, а только одну четко определенную задачу.

Концепция когезии применяется не только к методам, но также и к классам.

Представьте, что мы программируем игру и что в ней мы определили класс Player.

```
class Player{  
    private String name;  
    private int numberLives;  
    private String weaponDescription;  
}
```

Мы определили некоторые переменные экземпляра, такие как имя, количество жизней и описание оружия, которое несет игрок.

Но при кодировании вашей программы в какой-то момент вам может потребоваться включить также боеприпасы для оружия, которое имеет игрок.

Когда игрок стреляет, оставшиеся боеприпасы могут уменьшиться.

```
class Player{  
    private String name;  
    private int numberLives;  
    private String weaponDescription;  
    private int weaponAmmunition;  
}
```

Но теперь мы пересекаем тонкую границу.

Что, если нам придется указывать еще и вес оружия?

Должны ли мы включить дополнительную переменную для этого тоже?

Или мы должны определить класс Weapon со всеми необходимыми полями.

И у игрока будет оружие, определенное новым классом?

```
class Player{
    private String name;
    private int numberLives;
    private Weapon myWeapon;
}

class Weapon{
    private float weight;
    private String description;
    private int ammunition;
    ...
}
```

Если мы это сделаем, мы можем предоставить игроку даже несколько видов оружия в массиве объектов класса `Weapon`.

В этом случае становится ясно, что определение класса `Weapon` здесь уместно, вместо того, чтобы помещать все атрибуты оружия в класс `Player` без какой-либо структуры.

```
class Player{
    private String name;
    private int numberLives;
    private Weapon[] myWeapons;
}

class Weapon{
    private float weight;
    private String description;
    private int ammunition;
    ...
}
```

Здесь мы видим важность когезии.

Также как когезия методов, когезия классов помогает читать и понимать код, а также повторно использовать его в разных контекстах.


Хорошим подходом является написание кода, отдельные части которого мало зависят друг от друга.

Если одна часть кода зависит от другой, любое изменение в последней будет означать изменение в первой.

И это не хорошо.

Давайте смоделируем геометрические объекты на плоскости.

Во-первых, нам нужно смоделировать точки на плоскости.



```

class Point{
    double x;
    double y;
    public Point(double x, double y){
        this.x=x;
        this.y=y;
    }
}

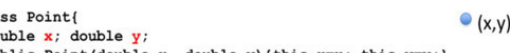
```

Точка характеризуется двумя координатами x и y.

И как только мы определили класс для точек, мы можем определить класс для отрезка линии.

Отрезок определяется двумя точками.

Таким образом, объект класса Segment имеет два объекта класса Point.



```

class Point{
    double x; double y;
    public Point(double x, double y){this.x=x; this.y=y;}
}

class Segment{
    Point a; Point b;
    public Segment(Point a, Point b){this.a=a; this.b=b;}
    double length(){
        return
            Math.sqrt(Math.pow(b.x-a.x,2)+Math.pow(b.y-a.y,2));
    }
}

```

В классе Point, у нас есть два поля или переменных экземпляра, x и y.

Мы определяем их тип как double.

Конструктор класса Point принимает два аргумента и сохраняет их в соответствующих полях.

Как только мы определили класс Point, мы можем определить класс Segment.

Отрезок линии определяется на плоскости, с помощью двух точек.

Поэтому здесь мы определяем две переменные экземпляра или два поля для двух точек, a и b.

Затем мы определяем конструктор Segment и метод length для вычисления длины отрезка.

Но с таким подходом есть проблема.

Если мы решим представить точки по-другому, нам также нужно будет изменить класс Segment.

Например, если класс Point не использует декартово представление с координатами x и y, а полярные координаты с углом и модулем, нам нужно изменить класс Segment.

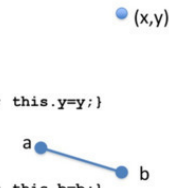
Нам нужно сделать определение класса Segment как можно более независимым от внутреннего представления точек.

Здесь у нас код уже лучше.

```

class Point{
    private double x; private double y;
    double getX(){return x;}
    double getY(){return y;}
    public Point(double x, double y){this.x=x; this.y=y;}
}
class Segment{
    private Point a; private Point b;
    public Segment(Point a, Point b){this.a=a; this.b=b;}
    double length(){
        return Math.sqrt(Math.pow(b.getX()-a.getX(),2)
            + Math.pow(b.getY()-a.getY(),2));
    }
}

```



На этот раз мы делаем оба поля *x* и *y* приватным.

Может быть, вы уже задавались вопросом, почему мы не сделали этого с самого начала.

Это было нужно, чтобы проиллюстрировать тесную связь между двумя классами.

Эти поля могут использоваться только внутри класса *Point*.

Но класс *Segment* также нуждается в этой информации.

Поэтому мы определяем два публичных метода *getX()* и *getY()*.

И они используются внутри класса *Segment*.

С первого взгляда, кажется, что мы мало что сделали.

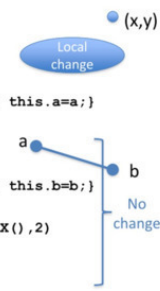
Но если мы решим определить точки в полярных координатах с модулем и углом, не будет никаких проблем.

Мы используем поле *r* для модуля и поле *a* для угла.

```

class Point{
    private double r; private double a;
    double getX(){return r*Math.cos(a);}
    double getY(){return r*Math.sin(a);}
    public Point(double r, double a){this.r=r; this.a=a;}
}
class Segment{
    private Point a; private Point b;
    public Segment(Point a, Point b){this.a=a; this.b=b;}
    double length(){
        return Math.sqrt(Math.pow(b.getX()-a.getX(),2)
            + Math.pow(b.getY()-a.getY(),2));
    }
}

```



Как насчет методов *getX* и *getY*? Они необходимы в классе *Segment*.

Здесь не будет никаких проблем, мы переопределяем тела этих методов в классе *Point*.

И обратите внимание, что это изменение остается локальным для класса *Point*.

В классе *Segment* нет никаких изменений.

Таким образом, здесь каждый класс управляет своими собственными данными и предоставляет внешнему классу все необходимое.

Это позволяет нам контролировать внутреннее представление независимо от того, как мы его используем.

Поэтому инкапсуляция, если она хорошо выполнена, помогает нам уменьшить связь между разными частями кода.

Слабое связывание подразумевает меньшую координацию между частями программы и меньшее распространение изменений.

Первым инструментом, который помогает нам достичь этого, является механизм инкапсуляции классов.

Иногда связывание является неявным.

Давайте рассмотрим простую ячейку, способную хранить значение.

```
class Cell{
    private static double value;
    public void add(double n){value += n;}
    public void sub(double n){value -= n;}
    public double getValue(){return value;}
}
```

Cell
- double value
+ void add (double n)
+ void sub (double n)
+ double getValue ()

Мы объявляем статическое поле типа double, а затем несколько методов.

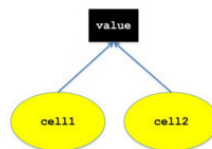
Один метод для добавления заданного числа, один метод для вычитания и один для получения значения, хранящегося в ячейке.

И мы не определяем никаких конструкторов, поэтому по умолчанию поле автоматически инициализируется равным нулю.

Теперь, что происходит, когда мы используем две таких ячейки в другом классе, скажем, калькуляторе?

Кажется, нет проблем.

```
class Calculator{
    private Cell cell1;
    private Cell cell2;
}
```



Но есть одно но.

Обе ячейки ссылаются на одно и то же поле!

Помните, что это поле статическое.

Таким образом, я могу изменить это значение из обеих.

Это было неявное связывание.

Этот пример простой, но показывает важную проблему, которая может возникнуть при неявном связывании.

Рефакторинг кода



Объектно-ориентированное программирование на Java

Отладка и тестирование кода

Лекция 7

Рефакторинг кода

Теперь давайте поговорим о рефакторинге.

Рефакторинг – это изменение, внесенное во внутреннюю структуру программного обеспечения, для того, чтобы было легче понять код, и проще его модифицировать, но без изменения внешнего поведения кода.

Поэтому рефакторинг не добавляет новых функций, он просто улучшает наш код с точки зрения понятности и ремонтпригодности.

Существует много принципов рефакторинга.

Мы рассмотрим здесь только несколько, чтобы попробовать рефакторинг.

Это такие принципы рефакторинга, как экстракция, переименование, перемещение и подъем.

Иногда лучше извлекать выражения в отдельные методы.

```
double paintNeeded(double spreadRate, int nCoats,  
double radius, int nCircles, double length, int nSquares){  
    return nCoats *  
        (nCircles*radius*radius*PI +  
         nSquares*length*length) / spreadRate;  
}
```

Такой код лучше для понимания, но также для тестирования и повторного использования.

Представьте, что вам нужно нарисовать несколько кругов и несколько квадратов.

И вы хотите знать, сколько краски вам нужно, принимая во внимание, что вы хотите сделать n слоев краски и что краска имеет расход, измеряемый в квадратных метрах на литр.

Здесь вы видите возможный метод.

Метод `paintNeeded` принимает в качестве аргумента расход краски, количество слоев, количество кругов и их радиус, а также количество квадратов и их сторона.

Далее вам нужно просто собрать выражение из этих параметров.

Что мы можем здесь сделать лучше, чтобы было гораздо яснее и проще поддерживать код.

Мы определили методы вычисления площади окружности и квадрата, а затем использовали их в методе `paintNeeded`.


```
double areaCircle (double radius){
    return radius*radius*PI;
}
double areaSquare (double length){
    return length*length;
}
double paintNeeded(double spreadRate, int nCoats,
double radius, int nCircles, double length, int nSquares){
    double a = areaCircle(radius);
    double b = areaSquare(length);
    return nCoats*(nCircles*a+nSquares*b)/spreadRate;
}
```

Теперь код проще понять, легче протестировать и методы легче использовать повторно. Рефакторинг с помощью переименования – это использование осмысленных идентификаторов для всех видов имен: переменных, методов, классов и т. д.

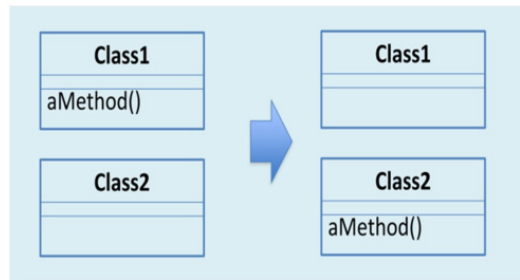
```
int add(int value){
    return value+1;
}
int addOne(int value){
    return value+1;
}
```

В предыдущем примере, мы объявили переменные `a` и `b` для хранения площади круга и квадрата.

```
double areaCircle (double radius){
    return radius*radius*PI;
}
double areaSquare (double length){
    return length*length;
}
double paintNeeded(double spreadRate, int nCoats,
double radius, int nCircles, double length, int nSquares){
    double areaC = areaCircle(radius);
    double areaS = areaSquare(length);
    return nCoats*(nCircles*areaC+nSquares*areaS)/spreadRate;
}
```

Но гораздо лучше использовать осмысленные имена, такие как `areaC` и `areaS`.

В некоторых случаях мы могли бы переместить методы в другое место, где они имеют больше смысла.



Например, в другой класс.

Также можно перемещать и поля класса.

Если вы хотите вычислить площадь круга и квадрата, гораздо лучше иметь конкретные классы, где вычисляется площадь, а не вычислять площадь в этом методе.

```

double paintNeeded(double spreadRate, int nCoats,
    double radius, int nCircles, double length, int nSquares){
    double areaC = areaCircle(radius);
    double areaS = areaSquare(length);
    return nCoats*(nCircles*areaC+nSquares*areaS)/spreadRate;
}

```

Здесь мы представили два класса Circle и Square, вместе с методом вычисления площади.

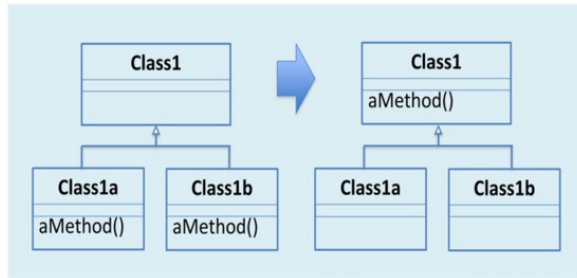
```

public class Circle{private double radius;
    public double area(){return radius*radius*PI;}
}
public class Square{private double length;
    public double area(){return length*length;}
}
...
double paintNeeded(double spreadRate, int nCoats,
    Circle c, int nCircles, Square s, int nSquares){
    return nCoats*(nCircles*c.area()+nSquares*s.area())/spreadRate;
}

```

Метод paintNeeded теперь имеет объекты классов в качестве аргументов.

Теперь о рефакторинге с помощью подъема.



Если у нас есть одни и те же методы в двух классах, которые расширяют один класс, нам лучше поднять этот метод.


То же самое относится и к полям класса.

Здесь мы видим пример с классом Figure и двумя дочерними классами Circle и Square.

```
class Figure{...}

class Circle extends Figure{
    String color;
    String getColor(){return color;}
}

class Square extends Figure{
    String color;
    String getColor(){return color;}
}
```



Метод getColor () находится в обоих классах.

И то же самое происходит с полем цвета.


В этом случае лучше поднять метод, а также поле класса.

Код стал намного яснее.

```
class Figure{...
    String color;
    String getColor(){return color;}
}

class Circle extends Figure{...
}

class Square extends Figure{...
}
```



Мы рассмотрели только несколько принципов рефакторинга. Есть много других.

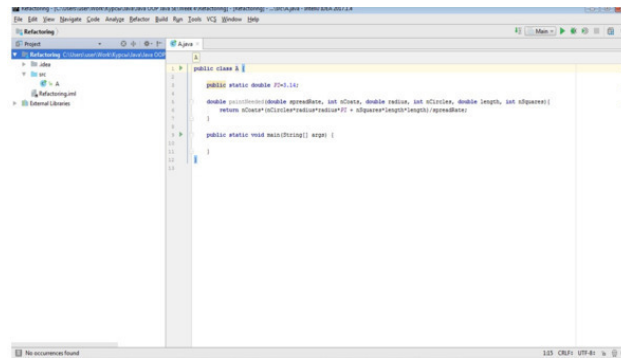
Например, помимо поднятия, есть также опускание.

Например, если у вас есть метод, который применим только к кругу, например, метод getRadius (),

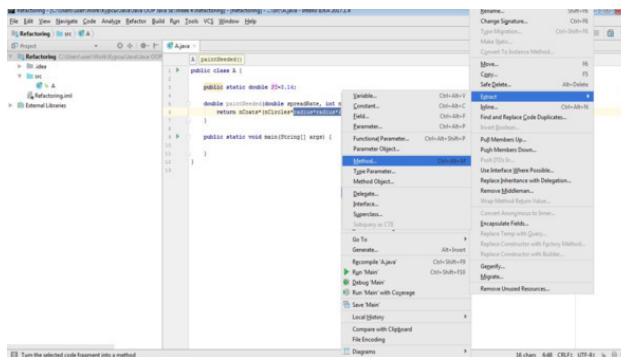
Он должен быть в классе круга, а не в его суперклассе.

Посмотрим, как делается рефакторинг в IDEA.

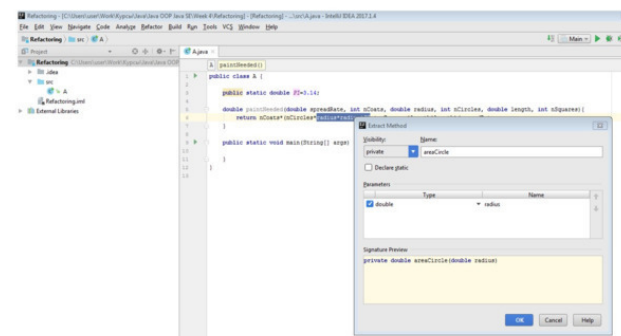
Здесь у нас есть класс A с методом paintNeeded.



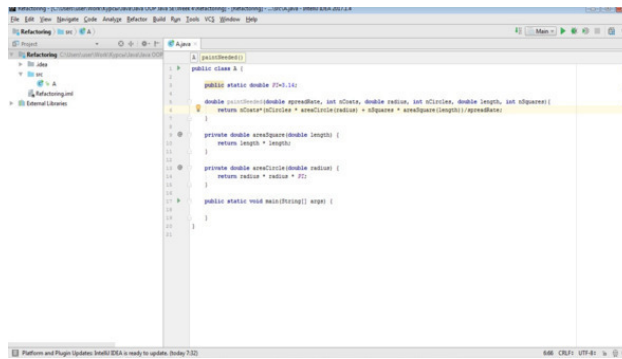
Для экстракции кода в метод нужно выделить код и нажать правой кнопкой мышки.



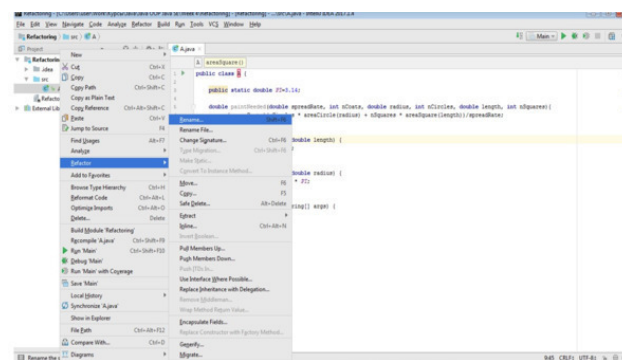
Выбрать в меню Refactor -> Extract -> Method.
Ввести имя нового метода и нажать ОК.



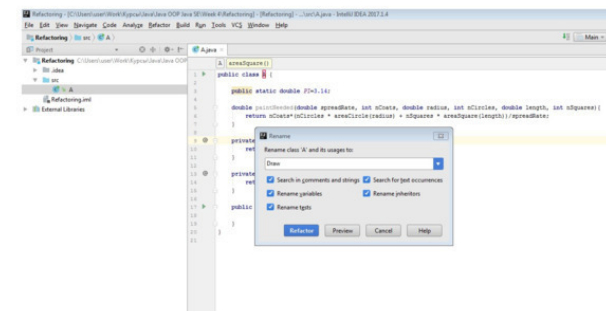
В результате получим рефакторинг кода.



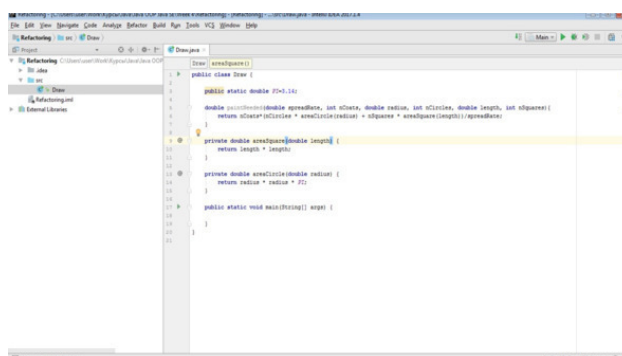
Для переименования, например, класса, нажмем правой кнопкой мышки на имени класса и выберем Refactor -> Rename.



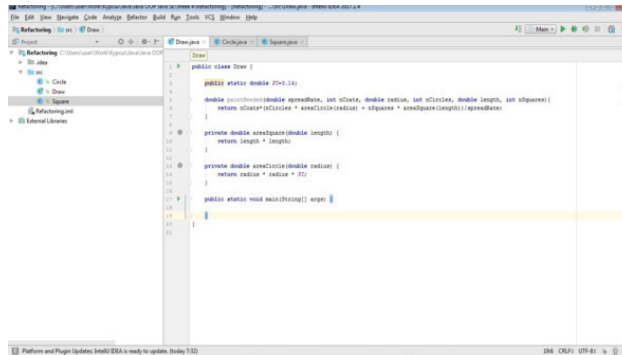
Введем новое имя класса и нажмем Refactor.



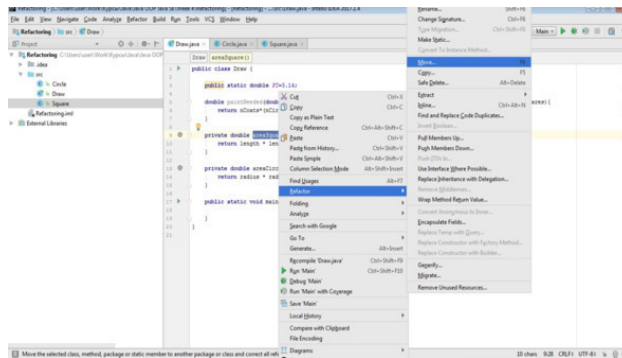
В результате получим новое имя.



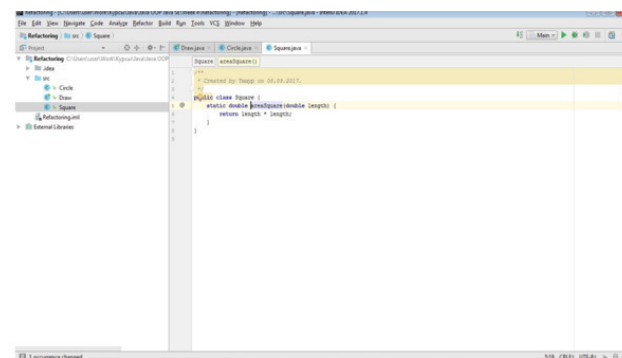
Точно так же можно переименовывать переменные и методы.
 Для перемещения метода, создадим два класса Circle и Square.
 Затем выделим имя метода, нажмем правой кнопкой мышки и выберем Refactor -> Move.



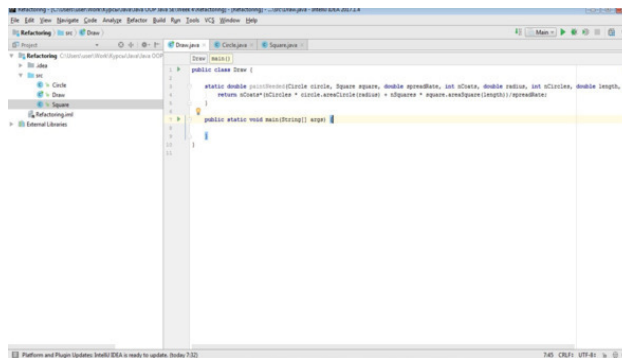
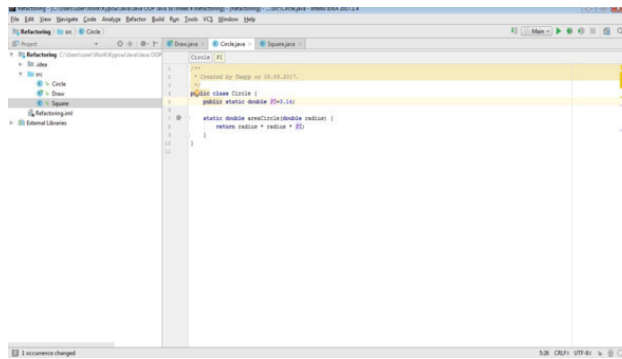
При этом метод сначала будет сделан статическим, а затем его можно будет переместить.



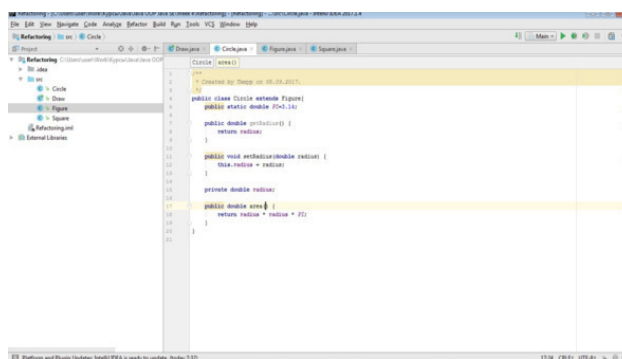
Тоже самое сделаем с другим методом.



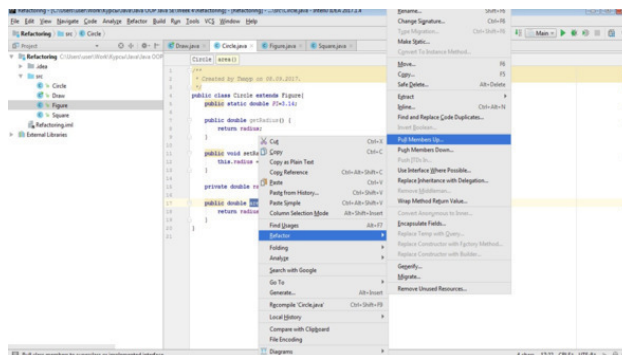
После этого можно переделать эти методы в публичные.

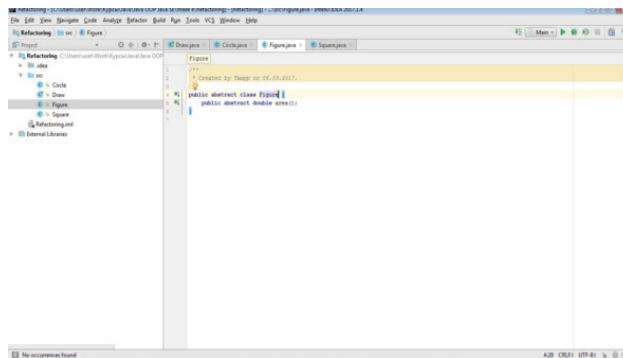
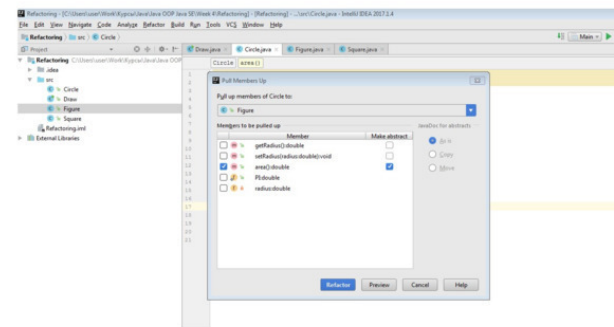


Переименуем эти методы в area и создадим суперкласс для этих двух классов.



Затем выделим метод и выберем Refactor -> Pull up. Таким образом мы получим абстрактный метод в классе Figure.





Java Collections Framework

Далее мы рассмотрим Java Collections Framework.



Объектно-ориентированное программирование на Java

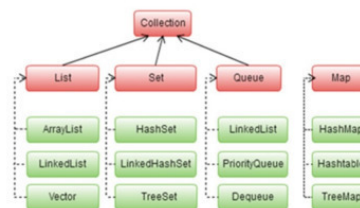
Коллекции в Java

Лекция 1

Введение

Сначала у нас будет общий обзор фреймворка, посвященный ключевым интерфейсам, которые он определяет.

Java Collections Framework



Потом мы сосредоточимся на нескольких популярных классах, которые реализуют интерфейсы этого фреймворка.

Java Collections Framework – это набор интерфейсов в классах, которые позволяют хранить данные в одном объекте, подобно тому, как массив может хранить данные.

Но Java Collections Framework дает нам гораздо больше, чем массивы.

Хотя массивы отлично подходят для хранения данных коллекции, у них есть много ограничений.

Первое ограничение заключается в том, что при создании массива, устанавливается его размер, и он не может быть изменен.

Таким образом, как только массив заполнен, мы не можем добавить данные к нему, пока мы не создадим совершенно новый массив и не скопируем в него существующие данные.

Можно было бы сказать, что, я всегда буду создавать массив, который больше, чем может когда-либо понадобиться.

Но тогда у нас будет много неиспользованного и потраченного впустую пространства.

Другое ограничение состоит в том, что мы должны точно управлять тем, как все данные будут храниться в массиве.

Выбирая индекс, куда элемент данных будет помещаться при добавлении в массив и в явном виде указывать, куда элемент данных будет перемещен, если нам придется перетасовывать элементы данных в массиве.

Массивы также не имеют многих встроенных функций, которые помогли бы нам в управлении данными, которые хранятся.

А для некоторых приложений массивы просто не обеспечивают эффективное решение. Например, при использовании массива для представления нити ДНК.

Если бы мы вырезали часть генов из середины нити, нам нужно было бы сдвинуть все последующие данные в массиве для заполнения освобожденных элементов.

Чтобы устранить эти недостатки, Java предоставляет библиотеку Java Collections Framework.

Здесь коллекция с заглавной буквы C, может просто считаться контейнером, который содержит элементы одного типа в одном объекте.

В этом смысле коллекция похожа на массив.

Но в отличие от массива, мы могли бы иметь контейнер, который содержит список студентов, упорядоченных по идентификатору студента.

Или у нас мог бы быть набор карт для игры в карты.

Или у нас может быть группа имен и телефонных номеров, которые позволяют нам сопоставить имя для соответствующего номера.

Обратите внимание, как эти примеры могут рассматриваться в терминах списка, набора или карты.

Collection Framework выполняет работу по представлению единой методологии для хранения, извлечения и обработки данных независимо от того, рассматриваем ли мы данные в виде списка, набора или карты.

Collection Framework имеет много преимуществ.

Во-первых, и в первую очередь это дает нам несколько стандартных способов хранения данных, стандартных структур данных.

Он освобождает программиста от необходимости создания кода низкого уровня для решения таких задач, и, таким образом, он позволяет программисту сосредоточиться на решении задачи и функциональности программы на высоком уровне.

Во-вторых, размер контейнера не статичен, а может расти и сокращаться по мере необходимости, при его использовании. И все это делается автоматически.

Вы можете выбрать контейнер, который лучше подходит для задачи, которую вы решаете, что приводит к более эффективному коду.

И вы можете быть уверены, что используемый вами контейнер был оптимизирован для быстрого запуска.

И, как мы увидим позже, фреймворк содержит в своей основе множество интерфейсов, которые имеют общие методы.

Это помогает в двух отношениях. Во-первых, путем повышения уровня абстракции, чтобы не беспокоиться о деталях реализации.

Во-вторых, это позволяет нам легко перейти к другой реализации тогда, когда нам это понадобится.

И, наконец, любая коллекция очень проста в изучении.

И ваши знания применимы к другой коллекции во фреймворке.

Подводя итог, некоторые программисты считают, что Collection Framework является лучшей частью языка Java, поскольку он элегантен, единообразен, и гибок.

Это позволяет Java-программисту, быть гораздо более продуктивным при написании Java приложений.

Таким образом, этот фреймворк очень важен для нас.

Общие понятия



Объектно-ориентированное программирование на Java

Коллекции в Java

Лекция 2

Общие понятия

Прежде чем мы углубимся в детали фреймворка, нам нужно определить термины, некоторые из которых мы будем обсуждать позднее.

В Java, коллекция представляет собой объект, который содержит группу других объектов.

Java Collections Framework - унифицированная архитектура для представления и управления коллекциями.

В Java, коллекция представляет собой объект, который содержит группу других объектов. Заметьте, коллекция содержит только объекты, что исключает примитивные типы. Но для примитивных типов язык Java обеспечивает соответствующие классы-оболочки. И Java определяет фреймворк как унифицированную архитектуру для представления и управления коллекциями.

Это означает, что мы можем обрабатывать большинство коллекций очень схожим образом, даже если они работают совсем по-разному.

Продолжая определения, язык Java определяет интерфейс как набор заголовков методов, которые не содержат реализаций.

Поскольку нет реализации методов, мы называем это абстрактным классом.

Интерфейс полезен для указания того, какие методы будут доступны и как их вызывать, и что они возвращают, но не как они фактически реализуются.

В Java абстрактный класс – это класс, у которого не реализован один или больше методов, а интерфейс – это абстрактный класс, у которого все методы не реализованы, все методы публичные и нет переменных класса.

И, наконец, общий тип или дженерик – это класс или интерфейс, который параметризован.

```
java.util.  
Interface Collection<E>  
  
Type Parameters:  
E – the type of elements in this collection  
  
All Superinterfaces:  
Iterable<E>  
  
All Known Subinterfaces:  
BeanContext, BeanContextServices, BlockingQueue<E>, BlockingQueue<E>, Deque<E>, List<E>, NavigableSet<E>, Queue<E>, Set<E>, SortedSet<E>, TransferQueue<E>  
  
All Known Implementing Classes:  
AbstractCollection, AbstractList, AbstractMap, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrangedQueue, ArrayList, ArrayListSubList, BeanContextServicesSupport, BeanContextSupport, ConcurrentHashMap, ConcurrentLinkedDeque, ConcurrentLinkedList, ConcurrentLinkedQueue, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, Jdk8TimeLearners, LinkedBlockingDeque, LinkedBlockingList, LinkedBlockingQueue, LinkedList, LinkedHashMap, PriorityQueue, PriorityQueue, RoleList, RoleBasedRoleList, Stack, SynchronousQueue, TreeSet, Vector  
  
public interface Collection<E>  
extends Iterable<E>  
  
The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any direct implementations of this interface; it provides implementations of more specific subclasses like Set and List. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.
```

В этом случае, параметр представляет собой информацию типа, которая определяет тип данных, которые фактически будут храниться в контейнере.

Этот параметр получит тег, при создании экземпляра фактического объекта контейнера.

Прежде чем продолжить, давайте поговорим о важности интерфейсов и дженериков в терминах collections framework.

Дженерики позволяют типам (классам и интерфейсам) быть параметрами при определении классов, интерфейсов и методов.

Также как формальные параметры, используемые в объявлениях методов, параметры типа предоставляют возможность повторного использования одного и того же кода с различными входными данными.

Разница в том, что входные данные для формальных параметров являются значениями, а входные параметры типа – это типы.

Когда различные контейнеры в collections framework используют один и тот же интерфейс, это означает, что все они поставляют один и тот же набор базовых методов.

Они могут также предоставить дополнительные методы, но мы знаем, как минимум, что существуют все методы, указанные в интерфейсе.

Это делает классы единообразными, и как только вы знаете, как использовать один класс, вы знаете, как использовать другие классы.

И что еще более важно, вы можете решить использовать другой контейнер без изменения кода, который работает на данном контейнере.

Опять же, дженерики важны, поскольку они позволяют нам определять контейнер независимо от типа, но при этом, позволяя нам обеспечить совместимость типов во время компиляции.

Мы увидим множество примеров интерфейсов и классов дженериков по мере продвижения вперед.

Структурированные данные



Tech Solutions

Объектно-ориентированное программирование на Java

Коллекции в Java

Лекция 3

Структурированные данные

Итак, еще раз, коллекция – это объект, который может хранить данные для нас. Или, это то, что мы также называем структурой данных.

Для решения задач создания сложных
структур данных применяются классы
Java Collections Framework

Мы будем ссылаться на сохраненные объекты в коллекции как элементы.

Теперь, некоторые коллекции сохраняют элементы, используя определенный порядок, в то время как другие неупорядочены.

Некоторые позволяют дублировать элементы, в то время как другие этого не делают.

Несколько типичных операций, которые мы можем выполнять в коллекции, заключаются в добавлении нового элемента в коллекцию, удалении элемента из коллекции, удалении всех элементов из коллекции, поиске в коллекции, чтобы увидеть, содержит ли она определенный элемент, и получении размера коллекции.

Примеры реализаций коллекций в Java включают ArrayLists, LinkedLists, HashMaps и TreeSets и другие.

ArrayList, LinkedList, HashMap, TreeSet,

```
import java.util.*;
```

Мы рассмотрим ArrayList и HashMaps более подробно позднее.

Java collection framework содержится в пакете java.util, и поэтому чтобы использовать его, вы должны включить импорт java.util в верхней части вашей программы.

Напомним, что наши коллекции являются общими типами или дженериками.

```
Collection<Type> name = new Collection<Type>();

ArrayList<String> words = new ArrayList<String>();
words.add("Interface");
words.add("Generic");
```

Опять же, это позволяет их определять независимо от типа данных, подлежащих хранению.

Но это также означает, что мы обязаны указать тип данных, который мы собираемся хранить при построении или создании объекта коллекций.

Синтаксис создания объекта коллекции включает в себя указание имени коллекции, за которым сразу следует тип элемента в угловых скобках.

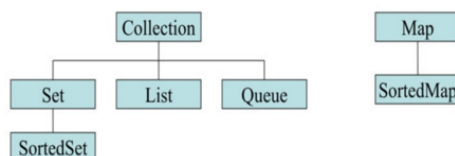
Затем вы даете объекту имя и инициализируете его, с помощью оператора new.

После создания объекта мы можем вызвать любой из методов интерфейса коллекции.

В примере кода мы видим, что мы можем добавить две строки в ArrayList.

Collection framework формирует иерархию интерфейсов.

В самой верхней части у нас есть интерфейс коллекции, представляющий собой просто набор объектов.



Set представляет собой интерфейс коллекции, который не может принимать повторяющиеся элементы.

Сортированный набор sorted set – упорядоченная версия интерфейса set.

Список list – это упорядоченный интерфейс коллекции, который позволяет дублировать элементы и обеспечивает поиск объектов на основе целочисленного индекса.

Очередь queue соответствует стандарту – первым пришел, первым ушел и предлагает множество реализаций.

Карта `map` образует собственную иерархию, поскольку вместо того, чтобы иметь дело с одним типом элемента, она поддерживает коллекцию пар ключ – значение и мы должны указать тип обоих.

Карта `map` не допускает дублирования ключей.

И, наконец, отсортированная карта `sorted map` – это карта, которая сохраняет свои ключи в порядке возрастания.

Интерфейс коллекции в верхней части иерархии обеспечивает использование этих методов.

```
– boolean add(E o)
– boolean contains(Object o)
– boolean remove(Object o)
– boolean isEmpty()
– int size()
– Iterator<E> iterator()
```

Метод `add` добавляет новый элемент коллекции.

Метод `contains` будет искать в коллекции указанный объект.

Метод `remove` удаляет объект из коллекции, если он там присутствует.

Метод `isEmpty` указывает пустая ли коллекция или нет.

Метод `size` говорит нам, сколько элементов находится в коллекции.

И метод `iterator` возвращает итератор элементов в коллекции.

Мы обсудим итераторы позже.

Наконец, если вы планируете использовать любой из классов `collections framework`, важно изучить все методы интерфейса коллекции.

Интерфейс `Set`, который расширяет интерфейс коллекции, обеспечивает неупорядоченный набор элементов, который не допускает дублирования.

```
interface Set<E> extends Collection, Iterable

interface List<E> extends Collection, Iterable
– void add(int index, E element)
– E remove(int index)
– E set(int index, E element)
– E get(int index)
– int indexOf(Object o)
– int lastIndexOf(Object o)
– ListIterator<E> listIterator()
```

Таким образом, он моделирует математическую концепцию множества.

Он предоставляет те же методы, что и интерфейс коллекции с добавленным ограничением, который предотвращает дубликаты.

Интерфейс списка `List` обеспечивает упорядоченную последовательность элементов.

Поскольку он упорядочен, пользователь может получить доступ к элементам по их целочисленному индексу.

Пользователь этого интерфейса имеет точный контроль над местоположением элемента в списке.

Списки, также, как и массивы, используют индексирование элементов, начиная с нуля.

Помимо предоставления всех методов интерфейса коллекции, интерфейс списка также предоставляет методы для добавления или удаления элемента по определенному индексу.

В этом случае другие элементы в списке будут перемещены вверх, чтобы освободить место для нового элемента или сместятся вниз, чтобы заполнить освободившееся место.

Вы можете установить или получить элемент с заданным индексом.

Вы можете получить индекс для объекта, и вы можете получить специальный итератор, который позволяет вам перемещаться по списку в обоих направлениях.

Интерфейс карты Map обеспечивает структуру данных для связанных ключей и значений.

```
Interface Map<K,V>

– V put(K key, V value)

– V get(Object key)

– Set<K> keySet()

– Collection<V> values()
```

Этот интерфейс не связан с интерфейсом коллекции, поскольку он имеет дело с парами элементов, а не с одним типом элемента.

Таким образом, карта не может содержать повторяющиеся ключи.

Каждый ключ может отображать не более одного значения.

Это похоже на то, как массив отображает индекс в значение, за исключением того, что теперь вы можете использовать любой объект как ключ, а не последовательность целых чисел.

Два важных метода карты позволяют вам добавить или изменить пару ключ-значение в карте и получить значение карты, связанное с данным ключом.

Кроме того, карты позволяют получить набор всех ключей или совокупность всех значений, содержащихся в карте.

Мы обсудили интерфейсы и то, как они соотносятся друг с другом. Но важно помнить, что интерфейс просто определяет методы, которые должны быть предоставлены.

Сами интерфейсы не являются чем-то, что может на самом деле сделать что-либо для нас.

Чтобы использовать интерфейс, нам нужен конкретный класс, который фактически предоставляет все методы, указанные в интерфейсе.

Классы могут содержать дополнительные методы, но, как минимум, они должны обеспечить методы интерфейса.

Эти классы по-прежнему дженерики, что позволяет написать их один раз, но при этом сохранять данные любого типа.

И ключевым аспектом всего этого является то, что существует много возможных путей для реализации заданного интерфейса.

Мы можем реализовать любой интерфейс, используя масштабируемый массив для управления базовым хранением объектов.

Или мы можем использовать связанный список, а не массив, или мы могли бы использовать древовидную структуру, или хеш-таблицу для хранения объектов.

Каждый из них может быть использован для реализации интерфейса с собственным набором преимуществ и недостатков.

Вот таблица, показывающая некоторые из возможных реализаций интерфейсов, которые мы обсуждали.

Interfaces	Implementations				
	Хэш-таблица	Масштабируемый массив	Дерево бинарного поиска	Связанный список	Хэш-таблица + связанный список
Set	HashSet		TreeSet (sorted)		LinkedHashSet
List		ArrayList		LinkedList	
Map	HashMap		TreeMap (sorted)		LinkedHashMap

Мы видим, что существует несколько вариантов реализовать набор, список или карту.

Давайте выделим основные отличия между масштабируемыми массивами и связанными списками, а также между хэш-таблицами и деревьями.

Важно понимать их сильные и слабые стороны и подумать о компромиссах, при выборе подходящей реализации.

Прежде чем мы сможем сравнить масштабируемый массив и связанный список, давайте коротко обсудим их основные понятия.

Ранее мы узнали, что когда массив создан, его размер установлен и не может быть изменен.

Итак, как может коллекция иметь масштабируемый массив?

Она может при необходимости создавать новый массив и копировать все данные из старого массива в новый массив.

Если массив большой, копирование всех элементов может занять некоторое время.

Теперь, связанный список хранит последовательность элементов, и вместо того, чтобы хранить их последовательно в выделенной памяти как массив, связанный список хранит каждый элемент в своем собственном узле, и каждый узел имеет ссылки на предыдущий элемент и последующий элемент.

Поэтому, когда мы сравниваем масштабируемые массивы со связанными списками, мы должны сделать это, рассматривая различные операции, которые мы можем сделать с ними.

Рассмотрим индексирование. Массивы позволяют нам получить прямой доступ к любому элементу в массиве на основе его индекса.

Это потому, что массивы хранят свои элементы в непрерывной памяти.

Если же мы хотим перейти к конкретному индексу в связанном списке, нам нужно пройти весь список с начала.

Когда мы пытаемся вставить или удалить элемент из середины массива, нам нужно сдвинуть последующие элементы чтобы освободить место или заполнить пробел.

Но такое смещение элементов не требуется со списками. Так как связать новый элемент с узлом или отвязать старый элемент от узла очень легко.

И мы должны помнить, что изменяемый размер влияет на скорость, тогда как связанные списки могут расти без необходимости делать полное копирование самих себя.

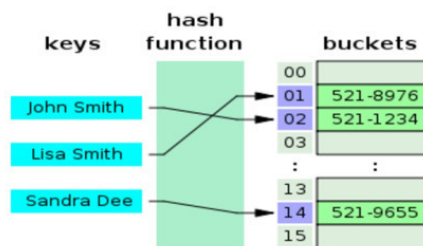
Прежде чем продолжить, следует отметить, что добавление новых элементов в конец массива выполняется быстро и эффективно, по крайней мере до тех пор, пока он не будет изменен.

Если мы сравним выделение памяти, мы увидим, что для массивов память нужна только для хранения фактических элементов, а связанный список также должен использовать память для ссылок в дополнение к элементам.

Теперь перейдем к хэш-таблицам и деревьям.

Во-первых, хэш-таблица – это структура, которая использует функцию, называемую хэш-функцией, для преобразования ключа в индекс для таблицы.

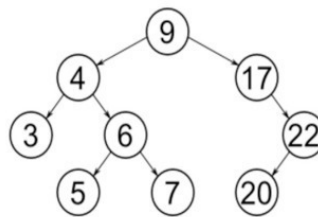
Цель состоит в том, чтобы разнести ключи случайным образом для таблицы, где мы храним элементы, связанные с ключом, таким образом, получая ключ, мы можем быстро найти и извлечь элемент, связанный с ним.



Эти таблицы должны обрабатывать ситуацию, когда два разных ключа хэшируют одно и то же место в таблице. Это называется коллизией.

Существуют разные способы обработки коллизий, и мы будем просто считать, что таблица обрабатывает их для нас, все еще поддерживая быстрый поиск.

Дерево двоичного или бинарного поиска хранит информацию в узлах дерева, которые поддерживают родительские и дочерние ссылки.



Фактически, каждый родитель может иметь две дочерние ссылки на поддеревья.

И чтобы облегчить быстрый поиск, деревья поддерживаются в упорядоченном виде.

Все элементы в левой части дерева меньше чем родительский элемент, и все элементы в правой части дерева больше чем родительский элемент.

Это называется свойством дерева двоичного поиска.

Учитывая, что хэш-таблицы и деревья бинарного поиска используются только для реализации интерфейсов `set` и `map`, мы будем сравнивать их по операциям, характерным для набора и карт.

В частности, эти два интерфейса в основном решают три вопроса.

Один вопрос, это добавление элементов или пар в коллекцию.

Второй, поиск элементов или пар в коллекции.

И третий, обработка всех элементов или пар в коллекции.

В случае выполнения вставки или поиска, хеш-таблицы дают нам прямой доступ к элементу или ключу через хеш-функцию.

В то время как для дерева, в этом случае, нам нужно пройти весь путь от корня до листа дерева.

Обратите внимание, что нам не нужно пересекать все дерево, так как мы можем использовать свойство дерева двоичного поиска.

При обработке всех элементов в наборе или карте, дерево позволяет нам обрабатывать элементы в отсортированном порядке, тогда как с хеш-таблицей, обработка не упорядочена.

Еще раз обратите внимание, что важно понимать компромисс между различными реализациями, чтобы вы могли выбрать более подходящую реализацию для задачи, которую вы решаете.

После того как вы выбрали свою реализацию класса, вы все равно должны объявлять свои переменные как интерфейс, вместо реализации класса.

Это повышает совместимость с кодом, написанным другими, и позволяет вам перейти на другой класс реализации в будущем, изменяя только одну строку кода.

```
Set<String> ss = new LinkedHashSet<String>();
```



```
LinkedHashSet<String> ss = new LinkedHashSet<String>();
```



Например,

Если вам нужен набор строк, и вы решили, что хотите использовать в качестве реализации `LinkedHashSet`, вы должны объявить свой объект коллекции как набор строк, вместо связанного набора хэшей строк.

Еще одно замечание, прежде чем двигаться дальше.

`Collections framework` также определяет несколько алгоритмов, которые могут применяться к коллекциям и картам.

Эти алгоритмы определены как статические методы внутри класса коллекций.

Это методы сортировки и поиска, перестановки и поворота элементов, для реверса и заполнения списка, а также для нахождения минимальных и максимальных элементов.

Завершая наше обсуждение Java коллекций, рассмотрим задачу итерации по всем элементам данных, содержащихся в коллекции, чтобы мы смогли обработать каждый элемент.

Существует несколько способов выполнения этого в Java, и мы рассмотрим три из них в этой лекции.

Мы рассмотрим использование индексирования, `for-each` цикла и итераторы.

Если используемая вами коллекция реализует интерфейс списка, тогда вы можете получить доступ к элементам в коллекции на основе индекса.

И возможность индексирования в коллекции позволяет нам использовать стандартный цикл `for` для доступа к каждому элементу.

Например, если вы создадите список массива строк, а затем элементы в списке массива, вы можете использовать простой цикл `for`, который перебирает от `i = 0` до размера массива `ArrayList`.

```
List<String> myFriends = new ArrayList<String>();
myFriends.add("Pete");
myFriends.add("Lisa");
myFriends.add("Gus");
for (int i=0; i<myFriends.size(); i++) {
    out.println(myFriends.get(i));
}
```

И тогда вы можете использовать метод `get` для доступа к каждому элементу по его индексу.

Это работает, но это не предпочтительный способ.

На самом деле этот стиль программирования может быть очень плохим для больших списков, так как каждая операция `get` должна пройти по связанному списку, начиная с первого элемента.

Более простой способ перебора коллекции, это использовать `for each` цикл.

Это тот же `for each` цикл, который мы видели при обходе массива, и он работает таким же образом.

```
List<String> myFriends = new
ArrayList<String>();
myFriends.add("Pete");
myFriends.add("Lisa");
myFriends.add("Gus");
for (String aFriend : myFriends) {
    out.println(aFriend);
}
```

Это очень чисто и лаконично, и это рекомендуется использовать по возможности.

Но есть ограничение на использование `for each` цикла, которое заключается в том, что он только для чтения, и он не позволяет вам изменять коллекцию при ее итерации.

Третий способ перебора, это когда коллекция использует итератор.

Итератор – это механизм итерации коллекции в обобщенном виде.

```
List<String> myFriends = new
ArrayList<String>();
myFriends.add("Pete");
myFriends.add("Lisa");
myFriends.add("Gus");

Iterator<String> itr = myFriends.iterator();
while (itr.hasNext()) {
    String aFriend = itr.next();
    out.println(aFriend);
}
```

Все коллекции содержат метод с именем `iterator`, который возвращает объект итератора для этой коллекции.

Этот итератор имеет три метода.

У него есть метод `hasNext`, который позволяет проверить, есть ли следующий элемент в коллекции.

Метод `next`, который дает вам доступ к следующему элементу коллекции.

И, дополнительно, метод `remove`, который позволяет вам удалить из коллекции последний элемент, к которому был выполнен доступ.

И вот наш пример, который теперь использует итератор.

После заполнения массива данными мы вызываем метод итератора, который создает объект итератора.

И затем мы используем `while` цикл, который выполняется до тех пор, пока итератор говорит, что есть еще один объект.

И в теле цикла, мы получаем следующий объект и обрабатываем его.

Эта стратегия работает очень хорошо и даже позволяет нам удалять элементы из коллекции, если мы этого хотим.

Хотя это дает больше строк кода, чем использование простого `for each` цикла.

Большое преимущество итераторов заключается в том, что они заботятся обо всех деталях низкого уровня для нас и позволяют нам перемещаться по коллекции независимо от того, какой это тип коллекции. Итераторы работают одинаково для массивов, связанных списков, бинарные деревья и хеш-таблиц.

Это важно, потому что иногда мы не знаем точной реализации, с которой мы можем иметь дело.

Но пока мы имеем дело с абстракцией более высокого уровня, такой как итератор, детали низкого уровня нам не важны.

Итак, как уже упоминалось ранее, мы хотим программировать скорее интерфейс, чем конкретный класс.

И, таким образом, наш код будет продолжать работать, даже если позже мы решили использовать другой конкретный класс.

Наконец, стоит отметить, что интерфейс карты не предоставляет итератор.

Но карта имеет два метода, которые возвращают нам коллекции, и мы можем легко перебирать их.

Interface Map<K,V>

Set<K> keySet()

Collection<V> values()

ArrayList



Tech Solutions

Объектно-ориентированное программирование на Java

Коллекции в Java

Лекция 4

ArrayList

Теперь мы рассмотрим подробнее класс ArrayList фреймворка Java Collection.

Этот класс предоставляет структуру, похожую на массив и обеспечивает прямой доступ к любому элементу.

```
java.util
Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:
AttributeList, RoleList, RoleUnresolvedList

public class ArrayList<E>
  extends AbstractList<E>
  implements List<E>, RandomAccess, Cloneable, Serializable
```

И в то же время, он дает возможность увеличивать свой размер в процессе выполнения приложения.

Класс ArrayList реализует интерфейс List фреймворка Java Collection, и класс ArrayList предназначен для того же что и массив, т.е. он хранит данные одного типа в близко расположенных местах памяти и дает нам прямой доступ к любому элементу на основе его индекса в последовательности.

Самая большая разница по сравнению с массивом заключается в том, что ArrayList позволяет изменять размер коллекции динамически во время выполнения программы.

Operation	Array	ArrayList
Get an element.	<code>x = data[4];</code>	<code>x = data.get(4);</code>
Replace an element.	<code>data[4] = 35;</code>	<code>data.set(4, 35);</code>
Number of elements.	<code>data.length</code>	<code>data.size();</code>
Number of filled elements.	-	<code>data.size();</code>
Remove an element.	-	<code>data.remove(4);</code>
Add an element, growing the collection.	-	<code>data.add(35);</code>
Initializing a collection	<code>int[] data = { 1, 4, 9};</code>	-

В то время как размер массива устанавливается единожды в момент его объявления.

Так как же объект класса ArrayList поддерживает прямое индексирование, что и обычный массив, и при этом может увеличивать свой размер?

Ну, во-первых, он использует массив в качестве переменной внутри объекта.

Этот массив скрыт внутри объекта ArrayList, и объект ArrayList будет хранить свои элементы в массиве.

Это позволяет классу ArrayList поддерживать прямое индексирование элементов.

Но так как массив спрятан внутри объекта ArrayList, объект может свободно заменить его на больший массив, когда это будет нужно.

Этой цели он достигает путем создания нового, большего массива.

Затем копирует все данные из старого массива в новый и заменяет его.

И мы индексируем элементы в ArrayList, начиная с 0 так же, как мы делали это с массивами и строками.

Поскольку ArrayList хранит данные в приватном массиве, ему необходимо обеспечить два свойства.

Первое – это логический размер, который является количеством элементов, хранящихся в массиве.

И второе – физический размер, который и есть реальный размер массива.

Теперь, метод size будет возвращать нам логический размер ArrayList.

Таким образом, мы можем узнать сколько элементов нужно обойти, если мы захотим написать цикл.

У ArrayList нет метода, возвращающего физический размер массива.

Но нам и не нужно знать об этом. Всё что нам нужно знать в случае, если мы заполним физический массив, объект ArrayList автоматически создаст новый, больший массив.

Когда объект ArrayList впервые создается, его логический размер равен 0, т.к. он пуст.

И объект будет отслеживать его размер по мере добавления или удаления элементов.

Когда мы сравниваем ArrayList с обычным массивом, мы уже упоминали, что его главное отличие – это динамическое изменение размера по мере того, как мы добавляем или удаляем элементы коллекции.

Но есть еще одна вещь. ArrayList сильно облегчает некоторые задачи, поскольку он делает многое за нас.

Это касается вставки и удаления элементов из любого места коллекции.

Когда мы добавляем элемент в середину, другие элементы нужно сдвинуть, чтобы освободить место для нового элемента.

Или когда мы удаляем элемент из середины, последующие элементы нужно сдвинуть назад, чтобы заполнить освободившееся пространство.

ArrayList делает это всё за нас.

Также, ArrayList предоставляет некоторые функции поиска, которых нет у массивов, что тоже облегчает нам жизнь, и мы можем выбрать тип поиска.

И обратите внимание, что, когда вам необходимо обойти все элементы, будь то массив или ArrayList, в каждом случае вы можете написать простой цикл for-each или стандартный цикл.

Итак, имея все эти преимущества ArrayList, почему мы всё же иногда используем обычный массив?

Почему бы не пользоваться ArrayList постоянно? Ну, помимо того факта, что массивы были всегда с нами и не во всех языках программирования есть ArrayList, есть пара причин, по которым кому-то захочется использовать массив.

Во-первых, ArrayList немного менее эффективен, чем массив.

Разница не настолько велика, но она может быть существенной в некоторых приложениях.

Если вы пишете некоторый код, для которого решающим фактором является время выполнения, тогда вам следует использовать обычный массив.

Во-вторых, мы теряем эти замечательные квадратные скобки, которые позволяют нам легко обращаться к элементам массива.

Для ArrayList вы обязаны вызывать методы доступа к элементам.

В-третьих, ArrayList одномерен. И хотя создать многомерный ArrayList возможно, это несколько мутно.

И, наконец, поскольку ArrayList является частью фреймворка коллекций, он содержит элементы типа class.

Хотя это и не является проблемой.

Так как в Java есть классы-оболочки Integer и Double типов, и они автоматически конвертируются из примитивных типов int и double.

Итак, подытожим.

Существует несколько причин, по которым некоторые захотят остаться с обычными массивами, а не с ArrayList, но для большинства приложений ArrayList – возможно наилучший выбор, благодаря простоте использования.

Так как кроме тех квадратных скобок и поля length, которое нам сообщает о размере массива, массивы предлагают не так много других инструментов для манипуляций с ними.

И нам нужно писать код для управления элементами массива.

Для сравнения, у ArrayList есть набор очень полезных методов, которые избавляют нас от необходимости обработки элементов самим.

Сейчас мы рассмотрим некоторые из этих методов.

add (value)	remove (index)	
add (index, value)	remove (value)	equals (list)
get (index)	indexOf (value)	iterator()
set (index, value)	lastIndexOf (value)	listIterator()
size ()		clear ()
isEmpty ()	contains (value)	
toString ()	containsAll (list)	

Сначала давайте обсудим добавление новых элементов в ArrayList.

Метод add сделает это за нас.

Есть две версии этого перегруженного метода.

В первой версии вы просто указываете элемент, который хотите добавить, и он будет добавлен в конец списка.

Во второй версии вы указываете элемент и индекс, куда вы хотите его вставить.

В этом случае все последующие элементы сдвигаются для освобождения места для нового элемента.

Заметьте, когда вы добавляете элемент в ArrayList, размер ArrayList всегда увеличивается на единицу.

Также, есть два способа удалить элемент из ArrayList.

В первом способе вы указываете, какой объект хотите удалить, и первый встретившийся такой элемент будет удален из ArrayList.

Во втором, вы указываете индекс элемента, который нужно удалить.

В обоих случаях, если есть еще элементы, следующие за удаляемым, они будут сдвинуты, чтобы заполнить освободившееся место.

И размер `ArrayList` будет уменьшен на единицу.

Если вы хотите извлечь или изменить элемент в `ArrayList`, используйте методы `get` и `set` соответственно.

Оба требуют указать индекс существующего элемента в `ArrayList`.

И как уже было упомянуто ранее, метод `size` возвращает количество элементов, содержащихся в `ArrayList`.

Резюмируем главные методы класса.

В этой таблице методов мы уже охватили методы `add`, `get`, `set`, `size`.

Метод `isEmpty` сообщает, пустой список или нет.

Метод `toString` возвращает строковое представление элементов списка.

Методы `remove` были также показаны.

Метод `indexOf` принимает искомое значение для поиска и возвращает индекс первого встретившегося искомого элемента.

В то же время метод `lastIndexOf` возвращает индекс последнего встретившегося элемента.

Оба возвращают `-1` в случае, если таковой элемент не найден.

Метод `contains` возвращает `true`, если искомый элемент присутствует где-нибудь в списке.

И метод `containsAll` принимает в качестве параметра список элементов и проверяет, если они все присутствуют в `ArrayList`.

Метод `equals` сравнивает объект `ArrayList` с другим объектом `list`, и они равны в том случае, если содержат все одинаковые элементы в одном и том же порядке.

Класс `ArrayList` содержит два метода, которые возвращающих итераторы.

Один возвращает обычный итератор, а второй возвращает итератор, позволяющий обходить список вперед и назад.

Метод `clear` удаляет все элементы из списка и устанавливает его размер равный нулю.


Это всего лишь некоторые методы, имеющиеся в классе `ArrayList`.

Как вы видите, эти методы делают `ArrayList` намного мощнее обычных массивов.

Как обсуждалось ранее, всякий раз, когда мы создаем объект `ArrayList` нам следует использовать интерфейс `List` в качестве типа переменной.

Например, если мы создаем `ArrayList` строк, нам следует присвоить его переменной тип `List` строк.

```
List<String> aList = new ArrayList<String>();
```



```
ArrayList<String> aList = new ArrayList<String>();
```



Это позволит нам легко изменять код в будущем, если мы захотим сменить `ArrayList` на `LinkedList` в случае необходимости.

Это не обязательно, но крайне рекомендуется.

И перед тем, как мы перейдем к последнему примеру, отметим, что у `ArrayList` есть альтернативный конструктор, позволяющий указать размер массива, который скрыт внутри объекта.

Это полезно, когда вам известно примерное число элементов, которые вы хотите хранить в `ArrayList`.

```
List<String> aList = new ArrayList<String>(250);
```

И это поможет избежать лишних операций по изменению размера при добавлении элементов.

Заметьте, даже если вы указали размер, `ArrayList` будет расти в случае необходимости, и указанный размер – всего лишь начальный физический размер массива.

Наконец, рассмотрим расширенный пример использования `ArrayList`.

```
public static List<String> loadWords(int len, String[]
ospd)
{
    List<String> words = new ArrayList<String>(1000);
    for (String word : ospd) {
        if (word.length()==len) {
            words.add(word);
        }
    }
    return words;
}

public static void mustHaveAt(char ch, int position, List<String>
aList)
{
    for (int i=aList.size()-1; i>=0; i--) {
        String word = aList.get(i);
        if (position >= word.length() ||
            word.charAt(position)!=ch)
        {
            aList.remove(i);
        }
    }
}
```

Рассмотрим игру Виселица.

Один человек загадывает слово и сообщает другому число букв.

Другой человек пытается отгадать слово, всякий раз угадывая буквы.

При каждой попытке, первый либо говорит угадывающему, что буква верна и где она находится в слове, либо сообщает, что такой буквы нет.

Игра завершается, когда все буквы слова отгаданы, и угадывающий выиграл.

Или при шести промахх угадывающий проиграл.

Сейчас, мы не будем разрабатывать игру Виселица целиком.

А сделаем пару методов, которые помогут угадывающему. В частности, мы создадим объект `ArrayList`, который будет содержать коллекцию возможных слов для нас и затем мы напишем другие методы, удаляющие слова, которые либо содержат неправильную букву, либо в них пропущена требуемая буква.

Начнем писать метод, который заполнит `ArrayList` коллекцией возможных слов.

Для этой игры мы используем словарь.

Поскольку считывание всех слов из файла является нетривиальной задачей, мы предположим, что кто-то уже считал все слова из файла в массив строк.

И этот массив нам предоставлен.

Заметьте, мы не загружаем весь словарь из массива в ArrayList.

А только те слова, чья длина совпадает с длиной загаданного слова.

Использование ArrayList для этой задачи – хороший выбор, так как мы не знаем заранее, сколько слов мы будем загружать.

Также важно, что ArrayList может увеличиваться при необходимости.

Начнем обсуждать код с изучения заголовка метода.

Метод называется loadWords и принимает два аргумента.

Первый – целое число, представляющее длину загаданного слова, второй – это массив строк, содержащий все слова из словаря.

Как мы видим, метод возвращает список строк.

В теле метода первое, что мы делаем – это создаем ArrayList строк и присваиваем его переменной words.

Так как словарь содержит около 80,000 слов, мы создадим ArrayList с начальным размером 1000 элементов.

Затем, у нас есть цикл for-each, обрабатывающий каждое слово в массиве.

И для каждого слова мы сверяем его длину с длиной загаданного слова.

Если длины совпадают, мы добавляем слово в ArrayList.

По завершении метода, мы возвращаем объект ArrayList вызывающему.

Далее скажем, что отгадчик сделал попытку, и она открыла одну из букв в загаданном слове.

В этот момент мы хотим убрать все слова из списка, в которых нет буквы на этой позиции.

Для этого, напишем еще один метод.

Метод будет принимать три аргумента: открытая буква, ее положение и ArrayList.

Заметьте, если попытка открывает два экземпляра с одинаковой буквой, тогда нам нужно вызвать этот метод дважды, для каждой позиции по разу.

Вот код, это выполняющий.

Метод называется mustHaveAt, поскольку мы требуем, чтобы каждое слово содержало букву на указанной позиции.

Этот метод принимает три аргумента, которые мы уже обсудили.

В этом методе мы будем удалять слова из ArrayList.

Когда мы это делаем, все последующие слова будут сдвинуты, чтобы заполнить освободившуюся запись.

Это создаст проблемы, если мы обходим список из начала в конец, так как если мы продвигаемся к следующему элементу после удаления, мы не сможем проверить элемент, который только что заполнил освободившееся место.

Простой способ решить эту проблему – обходить список в обратном направлении.

Теперь, когда мы переходим к следующему элементу, мы берем элемент, на который не повлиял сдвиг данных.

Рассматривая тело метода, мы видим цикл for, который перебирает индекс ArrayList от list size – 1 до 0.

В цикле for мы берем слово из этого индекса, и затем проверяем, пропущена ли искомая буква в данной позиции.

И если это так, то мы удаляем слово из списка. У нас есть еще маленькая дополнительная проверка, чтобы убедиться, что слово имеет достаточную длину для буквы в данной позиции.

По завершении метода, ArrayList, который был передан в качестве аргумента, изменился, и вызывающий увидит эти изменения.

Наконец, допустим, отгадчик сделал попытку при которой такая буква не найдена в загаданном слове.

На этом этапе мы хотим удалить все слова из списка, содержащие эту букву.

```
public static void mustNotHave(char ch, List<String> aList)
{
    Iterator<String> itr = aList.iterator();
    while (itr.hasNext()) {
        String word = itr.next();
        if (word.indexOf(ch)>=0) {
            itr.remove();
        }
    }
}
```

Метод, выполняющий это, принимает два аргумента: неправильную букву и ArrayList.

При написании этого кода, мы используем итератор вместо цикла for, в качестве примера альтернативного варианта.

Вот метод mustNotHave, в котором мы удаляем все слова, содержащие букву, которой нет в загаданном слове.

Начнем с создания итератора, который будет перебирать все слова в ArrayList.

И пока итератор сообщает, что есть еще слово для обработки, мы получаем следующее слово.

Затем, мы ищем слово на присутствие неправильной буквы, используя метод indexOf.

Метод indexOf вернет неотрицательное значение в случае успеха.

В таком случае, мы удаляем его из коллекции, вызывая метод remove для итератора.

Заметьте, для этого приложения ArrayList – отличный выбор.

Поскольку он может увеличивать размер, когда мы добавляем слова и может уменьшаться, когда мы удаляем слова.

И в любой момент, мы легко можем вывести все слова на экран, которые все еще в ArrayList, чтобы увидеть все оставшиеся варианты для загаданного слова.

HashMap



Объектно-ориентированное программирование на Java

Коллекции в Java

Лекция 5

HashMap

Теперь рассмотрим класс `HashMap`, который реализует интерфейс карты фреймворка коллекций Java.

Прежде чем перейти к `HashMap`, давайте вернемся к массивам и спискам массивов.

Массивы и списки массивов предоставляют нам удобный способ хранения данных.

И данные хранятся таким образом, что мы имеем прямой доступ к любому элементу, используя его индекс.

На самом деле иногда полезно подумать о массивах в терминах сопоставления целочисленного ключа со значением.

Например, рассмотрим симуляцию бросания двух кубиков несколько раз.

В этом примере, когда мы бросаем два кубика, они дают значение или ключ, который использовался для получения соответствующего счетчика.

Для решения такого рода задач массивы и списки массивов быстры и эффективны. А также очень просты в использовании.

Но есть некоторые проблемы, которые так просто не поддаются решению с помощью массивов.

Часто мы сталкиваемся с задачами, где наши ключи не являются целыми числами, которые могут использоваться как индекс в массиве.

И у нас есть данные разных типов, которые действуют как ключи.

Например, мы можем сопоставлять имена соответствующим номерам телефонов.

Или, возможно, сопоставлять идентификатор студента соответствующей записи студента.

И в этих случаях в качестве ключей мы используем уже не целые числа.

И даже если ключи являются целыми числами, как и в случае идентификаторов ученика, такие значения могут быть слишком велики, чтобы эффективно использоваться как индекс массива, потому что массив будет намного больше, чем количество записей, которые мы на самом деле хотим сохранить.

Вот где `HashMap` приходит на помощь.

Класс `HashMap` является частью фреймворка коллекций Java.

И его можно использовать для решения большого количества задач, для которых массивы и списки массивов не обеспечивают хорошее решение.

Класс `HashMap` реализует интерфейс карты, который определен во фреймворке, и который позволяет нам создавать отношения между ключами и значениями.

При создании таких отношений, важно гарантировать, чтобы ключи были уникальными. Это гарантирует, что существует только одна связь между ключом и значением.

Например, словарь – хорошая аналогия карты.

Он отображает ключ, в данном случае слово, для связанного значения, которое является соответствующим определением слова.

Здесь мы видим, что ключи и их значения образуют пару, которая хранится в карте.

Всякий раз, когда вы хотите получить значение из карты, вы даете ей ключ, а карта возвращает связанное значение.

С этой точки зрения мы видим, что массив это карта, которая содержит целые числа как ключи.

Вы предоставляете массиву целочисленный ключ или индекс, и получаете обратно связанное значение.

HashMap существует во многих языках программирования и часто используются с разными именами.

Некоторые языки, такие как Common Lisp, Perl и Ruby, называют хэш-карты как хэш-таблицы или просто хеши.

В то время как другие языки, такие как Python и Objective C называют их словарями.

Другие языки называют их ассоциативными массивами.

Итак, если вы слышите эти другие названия, вы можете связать их с классом HashMap языка Java.

HashMap – это структура, которая использует функцию, называемая хеш-функцией, для преобразования ключа в целочисленный индекс таблицы.

```

java.util
Class HashMap<K,V>

java.lang.Object
  java.util.AbstractMap<K,V>
    java.util.HashMap<K,V>

Type Parameters:
  K - the type of keys maintained by this map
  V - the type of mapped values

All Implemented Interfaces:
  Serializable, Cloneable, Map<K,V>

Direct Known Subclasses:
  LinkedHashMap, PrinterStateReasons

public class HashMap<K,V>
  extends AbstractMap<K,V>
  implements Map<K,V>, Cloneable, Serializable

```

Целью этого является разнесение ключей равномерно по таблице, где мы храним элементы, связанные с ключом.

Таким образом, имея ключ, мы можем преобразовать его в индекс, используя хеш-функцию, а затем быстро вставить или получить связанное с ним значение.

Эти таблицы должны обрабатывать ситуацию, когда два разных ключа сопоставляются с одним и тем же значением в таблице. Это называется коллизией.

И есть разные способы обработки коллизий.

Мы просто будем предполагать, что таблицы обрабатывают их для нас, обеспечивая при этом быструю вставку и быстрый поиск значений.

HashMap обеспечивает несколько ключевых свойств.

Прежде всего, возможность использования любых объектов в качестве ключей, а не только целые числа.

Кроме того, объект HashMap очень быстрый и эффективный, поскольку мы можем добавить пару ключ значение в HashMap в любое время, независимо от количества элементов, которые уже добавлены.

И имея ключ, мы можем быстро получить связанное значение, и мы можем быстро удалить пару ключ значение из HashMap.

Таким образом, `HashMap` – это мощный инструмент, который мы можем применить при решении различных вычислительных задач.

Однако чтобы иметь возможность быстрой вставки и извлечения, нам нужно пожертвовать другими возможностями.

В частности, мы больше не используем удобный целочисленный индекс для доступа к значениям, а мы можем использовать только ключи.

Кроме того, `HashMap` неупорядочены, так что если мы перебираем набор ключей, они оказываются в случайном порядке.

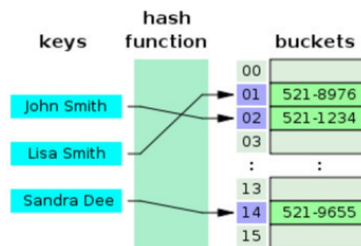
Таким образом, если мы хотим хранить ключи в определенном порядке, мы не можем использовать `HashMap`, или мы должны дополнить `HashMap` дополнительной структурой.

И, наконец, если мы хотим перебрать заданные ключи или значения в `HashMap`, требуется время, пропорциональное количеству ячеек `HashMap` плюс их размер (количество связанных пар ключ-значение) `HashMap`, так как `HashMap` можно рассматривать как совокупность связанных списков.

В то время как итерация большинства других контейнеров занимает время, пропорциональное только их размеру.

Но в основном, эти недостатки не имеют большого значения, когда мы решаем задачи, для которых `HashMap` хорошо подходят.

Основные операции в `HashMap`, это добавление, извлечение и удаление информации.



<code>put(key, value)</code>	<code>equals(map)</code>
<code>get(key)</code>	<code>clear()</code>
<code>remove(key)</code>	<code>containsKey(key)</code>
<code>size()</code>	<code>containsValue(value)</code>
<code>isEmpty()</code>	
<code>toString()</code>	

Чтобы добавить новую пару ключ значение в `HashMap`, можно использовать метод `put`.

Метод `put` принимает два параметра, ключ и связанное с ним значение.

Если у `HashMap` нет записи для указанного ключа, создается новая запись.

Если запись для ключа уже существует, новое значение заменит любое прежнее значение.

Чтобы получить значение из `HashMap`, можно использовать метод `get`.

Этот метод принимает только один параметр, ключ значения, которое мы хотим найти, и метод возвращает это связанное значение.

Это похоже на использование индекса для получения значения из `ArrayList`, но это ключ, а не индекс.

Если вы вызываете метод `get` и `HashMap` не содержит записи для указанного ключа, возвращается значение `null`.

`Null` – это особое ссылочное значение, указывающее, что не существует объекта, на который мы пытаемся сослаться.

И удаление из `HashMap` похоже на получение значения, когда вы просто указываете ключ.

В этом случае, метод называется `remove`, и он удалит записи, связанные с данным ключом.

Кроме того, метод `remove` возвращает значение, которое было связано с ключом, поэтому нет необходимости выполнять отдельный вызов метода `get`, если вы хотите узнать значение ключа, которое удаляется.

Кроме того, существует метод `size`, который сообщает, сколько пар ключ-значение было добавлено в `HashMap`.

Метод `isEmpty` сообщает, `HashMap` пуст или нет.

И метод `toString` возвращает строковое представление карты.

Все эти методы выполняются за фиксированное время, за исключением метода `toString`, который должен обязательно обрабатывать все записи в `HashMap`.

Существует также метод `equals`, который определяет, содержит ли другой объект карты такой же набор записей.

Метод `clear` удалит все записи из `HashMap`.

И методы `containsKey` и `containsValue` определяют, имеет ли `HashMap` указанный ключ или указанное значение соответственно.

Обратите внимание, что метод `containsKey` выполняется за фиксированное время, так как ключ хэшируется и это в итоге индекс.

Но выполнение метода `containsValue` зависит от объема и размера `HashMap`, так как здесь нужно перебрать все значения карте, чтобы узнать, существует ли указанное значение.

И еще есть дополнительные методы `HashMap`, которые здесь не были рассмотрены.

Теперь, если вы помните, интерфейс карты никак не связан с интерфейсом `Collection`.

В результате `HashMap` не предоставляет итератор, который позволяет обрабатывать все записи в `HashMap`.

Но `HashMap` обеспечивает представления коллекции.

Эти представления позволяют вам получить доступ к набору ключей `HashMap` или набору значений `HashMap`.

Обратите внимание: поскольку ключи должны быть уникальными, мы можем использовать набор, тогда как значения считаются коллекцией, поскольку они могут содержать дубликаты.

У `HashMap` есть два метода, которые обеспечивают эти представления.

Метод `keySet` возвращает набор ключей, в то время как метод `values` возвращает коллекцию всех значений в `HashMap`.

Обратите внимание, что поскольку оба эти метода возвращают тип коллекции, вы можете создавать итераторы, которые могут обрабатывать все данные в них.

```
// given a HashMap<String, Integer> named
myMap
Set<String> keys = myMap.keySet();
Iterator<String> itr = keys.iterator();
while (itr.hasNext()) {
    String key = itr.next();
    int value = myMap.get(key);
    Out.println(key + " => " + value);
}
```

И ключевая особенность этих представлений заключается в том, что они обеспечивают динамический доступ к HashMap.

В этом случае, если вы измените HashMap, вы увидите эти изменения в представлении и наоборот.

Например, если вы удалите ключ во время итерации по набору ключей, соответствующая пара ключ-значение также будет удалена из HashMap.

Например, скажем, у нас есть HashMap с именем myMap, которая представляет карту строковых значений для целых значений, и мы хотим обработать все записи HashMap, мы могли бы сделать это следующим образом.

Во-первых, мы вызываем метод keySet для HashMap объекта myMap и присваиваем его набору строк.

Мы используем набор строк, поскольку ключи являются строковыми объектами.

Затем мы вызываем метод iterator для набора, чтобы получить итератор типа String.

И затем, в цикле, пока итератор говорит, что у нас есть еще один элемент для обработки, мы получаем следующий элемент из набора, который будет строкой, представляющей ключ, и затем мы используем этот ключ для доступа к соответствующему значению.

Тогда мы можем делать все, что пожелаем с ключом и его значением.

И также как и со всеми реализациями фреймворка Java коллекций, рекомендуется объявить объект интерфейсом, а не реализацией.

```
Map<String,Integer> myMap =
    new HashMap<String,Integer>();
```



```
HashMap<String,Integer> myMap =
    new HashMap<String,Integer>();
```



В этом случае мы объявляем, что наш объект HashMap будет иметь тип Map, а не HashMap.

Поскольку это позволяет нам легко перейти на другую реализацию в будущем, если мы это сделаем.

Существует несколько требований, которым мы должны подчиняться при использовании HashMap.

Во-первых, как и для всех структур фреймворка коллекций, значения данных, с которыми мы имеем дело, должен быть типом класса. Это требование означает, что мы должны использовать классы-оболочки для примитивных типов.

Кроме того, любой тип, который мы используем для ключей, должен иметь два метода, определенных для этого класса объектов.

Метод `equals`, который можно использовать, чтобы сообщить – один и тот же это ключ, и метод `hashCode`, который реализует хэш-функцию для отображения объекта в индекс, который может использоваться для доступа к таблице.

Эти два метода существуют в большинстве классов, которые Java предоставляет нам, и эти классы мы могли бы использовать в качестве ключей, в частности, класс `String`.

И, наконец, настоятельно рекомендуется, чтобы тип, который вы используете для ключей, создавал неизменяемые объекты.

Теперь это не является жестким требованием, но рекомендуется, так как если вы поместите пару ключ – значение в `HashMap`, а затем измените ключ, вы больше не найдете эту пару.

Потому что пара хранится на основе старого ключа, но поиск выполняется по новому ключу.

Рассмотрим пример.

```
// Given ArrayList<String> words, an array list
// of words to be counted
Map<String, Integer> counts =
    new HashMap<String, Integer>();
for (String word : words){
    word = word.toLowerCase();
    if(!counts.containsKey(word)) {
        counts.put(word, 1);
    } else {
        counts.put(word, counts.get(word)+1);
    }
}
```

В вычислениях нам часто нужно подсчитать элементы.

Теперь, если элементы, которые мы считаем, являются числами в небольшом конечном диапазоне, тогда можно просто использовать массив счетчиков, аналогично примеру, где мы подсчитывали сколько раз выпало значение при броске двух кубиков.

В других случаях мы подсчитываем вещи, для которых мы не можем использовать массив.

В этом случае мы можем использовать `HashMap` для создания сопоставления объектов, которые мы подсчитываем, счетчику, связанному с данным объектом.

Каждый раз, когда мы сталкиваемся с другим экземпляром объекта, мы увеличиваем счетчик в `HashMap`.

Например, рассмотрим задачу подсчета вхождения слов в файл.

В этом случае мы создадим `HashMap`, которая отображает строки в целые числа, где строки – это слова, которые мы подсчитываем, а целые числа являются счетчиками для каждого слова.

Поскольку обработка файлов выходит за пределы наших нынешних навыков, будем считать, что все слова из файла уже были прочитаны и помещены в `ArrayList`, и наша задача будет состоять в том, чтобы обработать слова в `ArrayList`.

Первое, что мы делаем, это создадим `HashMap`, которая будет отображать строки в целые числа.

Затем мы хотим обработать все слова в списке массивов.

Самый простой способ сделать это – использовать `for each` цикл.

Первое, что мы сделаем, это преобразуем слово в нижний регистр, чтобы наш подсчет был нечувствительным к регистру.

Затем мы определяем, содержит ли `HashMap` текущее слово в качестве ключа или нет.

Если текущее слово отсутствует в `HashMap`, мы добавляем слово к карте со счетчиком, установленным в 1.

В противном случае мы получим текущий счетчик слова из `HashMap`, добавим 1 к нему и обновим `HashMap` с новым счетчиком.

Дженерики



Объектно-ориентированное программирование на Java

Коллекции в Java

Лекция 6

Дженерики

Теперь еще поговорим про дженерики, в том плане, что вы сами можете определять дженерики.

Общий тип или дженерик – это дженерик класс или интерфейс, параметризованный по типу.

Здесь сначала показан класс `Box`, который работает с объектами любого типа.

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}

public class Box<T> {
    // T stands for "Type"
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}

Box<Integer> integerBox = new Box<>();
```

Так как методы этого класса принимают или возвращают объект, вы можете передавать все, что захотите, при условии, что это не является одним из примитивных типов.

Во время компиляции невозможно проверить, как используется класс.

Одна часть кода может поместить `Integer` в поле класса и ожидать, что метод `get` вернет целое число, в то время как другая часть кода может ошибочно передать `String`, что приведет к ошибке выполнения.

Поэтому мы используем параметризованный вариант этого класса.

Как вы можете видеть, все вхождения объекта заменяются на `T`.

Таким образом, дженерики используются, потому что обеспечивают более строгую проверку типов во время компиляции.

Исправить ошибки компиляции проще, чем исправлять ошибки во время выполнения, которые трудно найти.

Кроме того, при использовании дженериков не требуется приведение типов, так как тип изначально задается параметром.

Дженерик может иметь несколько параметров типа.

Вы также можете заменить параметр типа (например, `K` или `V`) параметризованным типом.

```
public interface Pair<K, V> {  
    public K getKey();  
    public V getValue();  
}  
  
Pair<String, Integer> p;  
  
Pair<String, Box<Integer>> p;
```

Вы можете написать одно объявление дженерик метода, который можно вызвать с помощью аргументов разных типов.

```
public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {  
    return p1.getKey().equals(p2.getKey()) &&  
        p1.getValue().equals(p2.getValue());  
}  
  
static <T> T pick(T a1, T a2) {  
    return a2;  
}
```

Основываясь на типах аргументов, переданных дженерик методу, компилятор обрабатывает каждый вызов метода соответствующим образом.

Дженерик методы – это методы, которые имеют параметры типа.

Здесь область параметров типа ограничена методом, в котором они объявлены.

Можно объявлять статические и нестатические дженерик методы, а также дженерик конструкторы классов.

Синтаксис для дженерик метода включает параметр типа в угловых скобках перед типом возвращаемого метода.

Может возникнуть ситуация, когда вы захотите ограничить типы, которые могут использоваться как аргументы типа в параметризованном типе.

Например, метод, который работает с числами, может принимать только экземпляры класса `Number` или его подклассов.

Для этого используются параметры ограниченного типа.

Чтобы объявить параметр ограниченного типа, укажите имя параметра типа, за которым следует ключевое слово `extends`, а затем верхняя граница типа.

```
public <U extends Number> void inspect(U u){  
    System.out.println("U: " + u.getClass().getName());  
}
```

Если в такой метод передать, например, строку, возникнет ошибка компиляции. Также параметр типа может иметь несколько границ.

```
class D <T extends A & B & C> { /* ... */ }
```

Знак вопроса определяет подстановку верхней границы типа.

```
public static void process(List<? extends Foo> list)  
{ /*  
    ...  
    */ }
```

В данном случае знак вопроса указывает, что `Foo` – это любой тип, который соответствует `Foo` и любому подтипу `Foo`.

Если просто указать знак вопроса в угловых скобках как параметр типа, это будет означать тип `Object`.

Также как мы определили верхнее ограничение, можно определить нижнее ограничение типа.

Нижнее ограничение, с помощью знака вопроса и ключевого слова `супер`, ограничивает неизвестный тип конкретным типом или `супер`-типом этого типа.

```
public static void addNumbers(List<? super Integer> list) {  
    for (int i = 1; i <= 10; i++) {  
        list.add(i);  
    }  
}
```

Здесь параметр типа ограничен от класса `Integer` и выше.

Используя параметризованные типы, нужно быть осторожными с логическими операторами.

Например, нельзя применять такие простые операторы сравнения, как больше, меньше или равно, так как они применяются к примитивным типам.

В случае дженериков нужно применять специальные методы `equal` и `compareTo`.

Теперь, предположим, у нас есть класс `Box <Number>`.

`Box<Number>`

Можем ли мы использовать объявление `Box <Integer>`.

Хотя класс `Integer` является подклассом класса `Number`, мы не можем использовать объявление `Box <Integer>`, так как сам класс `Box <Integer>` не является подклассом класса `Box <Number>`.

Хотя при этом мы можем передавать в параметризованные методы класса `Box <Number>` объекты типа `Integer`.

Однако если использовать ограничение типа, это становится возможным.

Поскольку здесь одно ограничение вписывается в другое ограничение.

```
Box<? extends Integer> b = new Box<>();  
  
Box<? extends Number> a = b;
```


Рассмотрим другой пример.

Так как `List<EvenNumber>` является подтипом `List<? extends NaturalNumber>`, вы можете назначить `le` для `ln`. Но вы не можете использовать `ln` для добавления натурального числа в список четных чисел, так как это является сужением типа.

```
List<EvenNumber> le = new ArrayList<>();

List<? extends NaturalNumber> ln = le;

ln.add(new NaturalNumber(35)); // compile-time error
```

Наоборот можно.

В использовании дженериков существует ряд исключений.

Вот наиболее распространенные из них.

Нельзя создавать дженерики с примитивными типами.

```
Pair<int, char> p = new Pair<>(8, 'a'); // compile-time error

public static <E> void append(List<E> list) {
    E elem = new E(); // compile-time error
    list.add(elem);
}

public static <E> void append(List<E> list, Class<E> cls) throws Exception {
    E elem = cls.newInstance(); // OK
    list.add(elem);
}

public class MobileDevice<T> {
    private static T os;
}
```

Нельзя создавать экземпляры параметров типа с помощью оператора `new`.

Нельзя объявлять статические поля, типы которых являются параметрами типа.

Нужно быть осторожными при приведении типов с дженериками.

```
List<Integer> li = new ArrayList<>();
List<Number> ln = (List<Number>) li; // compile-time error

List<Integer>[] arrayOfLists = new List<Integer>[2]; // compile-time error

class QueueFullException<T> extends Throwable { /* ... */ // compile-time error

public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

Нельзя создавать массивы дженериков, так как массив, это данные одного и того же типа, а дженерики могут быть объектами разного типа.

Дженерик класс не может расширять класс `Throwable` прямо или косвенно.

Класс не может иметь двух перегруженных методов, которые будут иметь одну и ту же сигнатуру после стирания типа.

Потоки коллекций и фильтры



Tech Solutions

Объектно-ориентированное программирование на Java

Коллекции в Java

Лекция 7

Потоки коллекций и фильтры

Используя фреймворк коллекций в Java, программист должен использовать циклы и выполнять повторные проверки.

Еще одна проблема – эффективность; поскольку существуют многоядерные процессоры, программист Java должен писать параллельную обработку кода, которую трудно уберечь от ошибок.

Чтобы решить эти проблемы, Java 8 представила концепцию потока stream, которая позволяет программисту обрабатывать данные декларативно и использовать многоядерную архитектуру без необходимости писать для нее какой-либо конкретный код.

Поток stream представляет собой последовательность объектов из источника, которая поддерживает агрегированные операции.

```
java.util.stream
Interface Stream<T>

Type Parameters:
  T - the type of the stream elements

All Superinterfaces:
  AutoCloseable, BaseStream<T,StreamOf>

public interface Stream<T>
    extends BaseStream<T,StreamOf>

A sequence of elements supporting sequential and parallel aggregate operations. The following example illustrates an aggregate operation using Stream and IntStream.

    int sum = widgets.stream()
        .filter(w -> w.getColor() == RED)
        .mapToInt(w -> w.getWeight())
        .sum();

In this example, widgets is a Collection<Widget>. We create a stream of widget objects via Collection<Widget>.filter(). Then this stream is summed to produce a total weight.

In addition to Stream, which is a stream of object references, there are primitive specializations for IntStream, LongStream, and DoubleStream, all of which are referred to as "streams" and conform to the characteristics and restrictions described here.

To perform a computation, stream operations are composed into a stream pipeline. A stream pipeline consists of a source (which might be an array, a collection, a generator function, an I/O channel, etc.), zero or more intermediate operations (which transform a stream into another stream, such as filter(), map(), etc.), and a terminal operation (which consumes a
```

В качестве источника входных данных поток принимает коллекции, массивы или ресурсы ввода-вывода.

Stream поддерживает агрегированные операции, такие как filter, map, limit, reduce, find, match и т. д.

Большинство операций потока возвращают поток, так что их результат может быть конвейерным.

Эти операции называются промежуточными операциями, и их функция состоит в том, чтобы принимать входные данные, обрабатывать их и возвращать выходные данные целевому объекту.

Метод collect () – это завершающая операция, которая обычно присутствует в конце операции конвейерной обработки, чтобы отметить конец потока.

Операции потока выполняют итерации внутри исходных элементов, в отличие от коллекций, для которых требуется явная итерация.

В Java 8 каждый класс, реализующий интерфейс `Collection`, имеет метод `stream`, который возвращает последовательный поток, рассматривающий коллекцию как источник, и метод `parallelStream`, который возвращает параллельный поток, рассматривающий коллекцию как источник.

```
// Create an ArrayList
List<Integer> myList = new ArrayList<Integer>();
myList.add(1);
myList.add(5);
myList.add(8);

// Convert it into a Stream
Stream<Integer> myStream = myList.stream();
```

`Stream` предоставляет новый метод «`forEach`» для итерации каждого элемента потока.

```
List<String> strings = new ArrayList<>();
for (String string : strings) {
    System.out.println("Content: " + string);
}

List<String> strings = new ArrayList<>();
strings.stream().forEach((string) -> {
    System.out.println("Content: " + string);
});
```

Здесь вверху показан код для списка, который обходит список с помощью цикла `for`.

А внизу показан код, где создается поток, затем к нему применяется метод `forEach`, аргументом которого служит реализация интерфейса, представляющего операцию, которая принимает единственный входной аргумент и не возвращает результат.

При этом метод `forEach` применяет эту операцию к каждому элементу потока.

Метод `map` потока принимает в качестве аргумента реализацию интерфейса, представляющего функцию, которая принимает единственный входной аргумент и, в отличие от `forEach`, возвращает результат.

```
List<String> names = Arrays.asList("Pankaj", "amit", "DAVID");

List<String> upperCaseNames = new ArrayList<>();
for (String n : names) {
    upperCaseNames.add(n.toUpperCase());
}

upperCaseNames = names.stream().map(t -> t.toUpperCase()).collect(Collectors.toList());
```

При этом метод `map` применяет эту операцию к каждому элементу потока.

Здесь вверху показан код, в котором строки списка переводятся в верхний регистр в цикле `for`.

А внизу показан код, где создается поток, затем к нему применяется метод `map`, который применяет к каждому элементу списка метод `toUpperCase`.

Здесь также в конце к потоку применяется метод `collect`, который является завершающей операцией потока и который накапливает входные элементы в контейнер результатов.

Здесь используется класс `Collectors` и его метод `toList`, который накапливает входные элементы в новый список.

Метод `limit` потока усекает поток на длину, указанную в качестве аргумента метода.

```
List<String> list = Arrays.asList("AA", "BB", "CC", "DD", "EE");

list.stream().limit(3).forEach(s->System.out.println(s));

AA
BB
CC
```

Здесь, в примере, метод `limit` ограничивает список из 5 элементов, тремя элементами. Метод `sorted` потока сортирует элементы потока.

```
public class Employee {
    public String name;

    public Employee(String n) {
        name = n;
    }
}

Stream<Employee> emps = Stream.of(new Employee("John"), new Employee("Jane"),
    new Employee("Jack"));
List<String> sl = emps
    .map(e -> e.name)
    .sorted()
    .collect(Collectors.toList());
System.out.println(sl);

[Jack, Jane, John]
```

В этом примере поток получает на вход список из объектов.

Затем применяется метод `map`, который выделяет из каждого объекта поле `name`.

Затем поля `name` сортируются в алфавитном порядке и методом `collect` возвращается отсортированный список.

В этом примере используется другой вариант метода `sorted`, который в качестве аргумента принимает объект `Comparator`, обеспечивающий функцию сравнения элементов.

```

public class Employee {
    public String name;
    public Employee(String n) {
        name = n;
    }
    @Override
    public String toString() {
        return name;
    }
}

[Nick, Steve, Nathaniel]

Stream<Employee> emps = Stream.of(new Employee("Nathaniel"), new Employee("Steve"),
new Employee("Nick"));
List<Employee> sl = emps
    .sorted(Comparator.comparing(e -> e.name.length()))
    .collect(Collectors.toList());
System.out.println(sl);

```

Функция `comparing` принимает функцию, которая извлекает ключ сортировки. В данном случае это длина строки. Поэтому поля `name` здесь сортируются не в алфавитном порядке, а по длине. В Java 7 это выглядело бы следующим образом.

```

Collections.sort(employees, new Comparator<Employee>() {
    @Override
    public int compare(Employee o1, Employee o2) {
        return Integer.compare(o1.name.length(), o2.name.length());
    }
});

```

Здесь используется метод `sort` класса `Collections`, который получает на вход сортируемый список и реализацию интерфейса `Comparator`.

Как мы видим, с потоками код проще.

Метод `filter` потока возвращает элементы потока, соответствующие данному условию.

```

List<String> lines = Arrays.asList("spring", "node", "mkyong");

List<String> result = lines.stream()           // convert list to stream
    .filter(line -> !"mkyong".equals(line))   // we dont like mkyong
    .collect(Collectors.toList());            // collect the output and convert streams to a List

result.forEach(System.out::println);          //output : spring, node

spring
node

```

Условие – это реализация метода интерфейса, который принимает на вход объект и возвращает `true` или `false`.

В Java 7 это выглядит следующим образом.

```
List<String> lines = Arrays.asList("spring", "node", "mkyong");  
  
List<String> result = new ArrayList<>();  
for (String line : lines) {  
    if (!"mkyong".equals(line)) { // we dont like mkyong  
        result.add(line);  
    }  
}
```

Мы вручную переписываем список в новый список, фильтруя элементы в цикле.

Коллекции в Java 9



Tech Solutions

Объектно-ориентированное программирование на Java

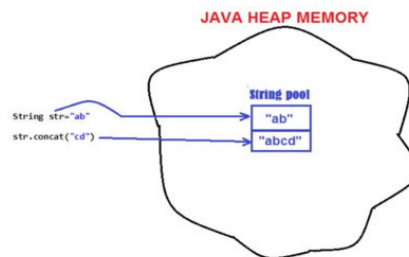
Коллекции в Java

Лекция 8

Коллекции в Java 9

Неизменяемый объект – это такой объект, чье внешнее видимое состояние не может измениться после его создания.

Immutable Class in Java



Класс String в библиотеке классов Java является примером неизменяемых объектов – они представляют отдельное значение, которое не может измениться в течение жизненного цикла объекта.

Неизменяемые классы при правильном использовании могут значительно упростить программирование.

Они могут находиться только в одном состоянии, поэтому если они правильно сконструированы, они никак не могут быть в несогласованном состоянии.

Вы можете свободно делать общими и кэшировать ссылки на неизменяемые объекты без необходимости копировать или клонировать их.

И вы можете кэшировать их поля или результаты их методов, не беспокоясь о том, что значения устареют или станут несогласованными с остальными состояниями объекта.

Из неизменяемых классов обычно получаются лучшие ключи карты.

И неизменяемые классы потоко-безопасны, поэтому вам не нужно синхронизировать доступ к ним через потоки.

В Java 9 создавать immutable или неизменяемые коллекции стало намного проще.

Ранее для этого приходилось использовать сторонние библиотеки.

Interface List<E>		
static <E> List<E>	of()	Returns an immutable list containing zero elements.
static <E> List<E>	of(E e1)	Returns an immutable list containing one element.
static <E> List<E>	of(E... elements)	Returns an immutable list containing an arbitrary number of elements.
static <E> List<E>	of(E e1, E e2)	Returns an immutable list containing two elements.
static <E> List<E>	of(E e1, E e2, E e3)	Returns an immutable list containing three elements.
static <E> List<E>	of(E e1, E e2, E e3, E e4)	Returns an immutable list containing four elements.
static <E> List<E>	of(E e1, E e2, E e3, E e4, E e5)	Returns an immutable list containing five elements.
static <E> List<E>	of(E e1, E e2, E e3, E e4, E e5, E e6)	Returns an immutable list containing six elements.
static <E> List<E>	of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)	Returns an immutable list containing seven elements.
static <E> List<E>	of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)	Returns an immutable list containing eight elements.
static <E> List<E>	of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)	Returns an immutable list containing nine elements.
static <E> List<E>	of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)	Returns an immutable list containing ten elements.

Теперь, метод of списка создает неизменяемый список из указанных элементов. Те же самые методы of есть для наборов и карт.

```
Set<String> questions = Set.of("What?", "Where?", "When?");

Set<String> questions = Set.of("What?", "What?"); // IllegalArgumentException: duplicate
element: What?

Map<String, String> params = Map.of("Name:", "John", "Surname:", "Snow", "Status:",
"Unmarried");
```

В отличие от списка, если передать дубликат в метод of набора Set, мы получим исключение.

В карте Map, последовательности идут парами ключ-значение.

Всего таких пар может быть 10, в отличие от List и Set.

И если в последовательности будут повторения ключей, мы получим исключение.

Полученные таким способом коллекции нельзя изменить, при попытке изменения мы получим UnsupportedOperationException.

И в метод of нельзя передать null, при этом во всех коллекциях мы получим NullPointerException.

Метод of не создает привычные ArrayList, HashSet или HashMap.

В Java 9 были созданы специальные ImmutableCollections которые и возвращаются;

И ImmutableCollections являются сериализуемыми, если содержимое тоже является сериализуемым.

Java Reflection



Tech Solutions

Объектно-ориентированное
программирование на Java

Лекция Java Reflection

Отражение – это одна из особенностей языка программирования Java.

Отражение позволяет исполняемой программе Java анализировать саму себя и управлять внутренними свойствами программы.

Например, класс Java может получить имена всех своих членов и отобразить их.

```
Method[] methods = MyObject.class.getMethods();  
  
for(Method method : methods){  
    System.out.println("method = " + method.getName());  
}
```

Такая возможность исследовать и манипулировать классом изнутри в других языках программирования просто не существует.

Например, в программе, написанной на Pascal, C или C ++ нет возможности получить информацию о функциях, определенных в этой программе.

Отражение применяется в JavaBeans, где программными компонентами можно визуально манипулировать с помощью инструмента builder.

Инструмент использует отражение для получения свойств компонентов Java, так как они загружаются динамически.

Java Reflection позволяет проверять классы, интерфейсы, поля и методы во время выполнения, не зная имен классов, методов и т. д. во время компиляции.

Также возможно создавать новые объекты, вызывать методы и получать / устанавливать значения полей с помощью отражения.

В этом примере мы получаем объект Class из класса MyObject.

И используя объект Class, мы получаем список методов в этом классе.

Программный интерфейс Reflection API представлен пакетом java.lang.reflect.

Package java.lang.reflect
Provides classes and interfaces for obtaining reflective information about classes and objects.

Class Summary	
Class	Description
AccessibleObject	The AccessibleObject class is the base class for Field, Method and Constructor objects.
Array	The Array class provides static methods to dynamically create and access Java arrays.
Constructor<T>	Constructor provides information about, and access to, a single constructor for a class.
Executable	A shared superclass for the common functionality of Method and Constructor.
Field	A Field provides information about, and dynamic access to, a single field of a class or an interface.
Method	A Method provides information about, and access to, a single method on a class or interface.
Modifier	The Modifier class provides static methods and constants to decode class and member access modifiers.
Parameter	Information about method parameters.
Proxy	Proxy provides static methods for creating dynamic proxy classes and instances, and it is also the superclass of all dynamic proxy classes created by those methods.
ReflectPermission	The Permission class for reflective operations.

Для использования этих классов необходимо выполнить три шага.

Первый шаг – получить объект `java.lang.Class` для класса, которым вы хотите манипулировать.

```
1. Class c = Class.forName("java.lang.String");
   Class c = int.class;
   Class c = Integer.TYPE;
   Class myObjectClass = MyObject.class

2. Method m[] = c.getDeclaredMethods();

3. System.out.println(m[0].toString());
```

Объект `java.lang.Class` используется для представления классов и интерфейсов в исполняемой программе Java.

Все типы Java, включая примитивные типы (`int`, `long`, `float` и т. д.), включая массивы, имеют связанный с ними объект `Class`.

Если вы знаете имя класса во время компиляции, вы можете получить объект класса с помощью поля `class`.

Если вы не знаете имя во время компиляции, но у вас есть имя класса в виде строки во время выполнения, вы можете получить объект `Class` с помощью метода `forName`.

При использовании метода `forName` вы должны указать полное имя класса, включая все имена пакетов.

Метод `Class.forName` может вызывать исключение `ClassNotFoundException`, если класс не может быть найден во время выполнения.

Второй шаг – это вызвать метод, такой как `getDeclaredMethods`, чтобы получить список всех методов, объявленных классом.

Как только эта информация будет получена, третий шаг – использовать API отражения для управления этой информацией.

После получения объекта `Class`, с помощью метода `instanceof` можно проверить будет ли экземпляр указанного класса экземпляром данного класса.

```
try {
    Class cls = Class.forName("A");
    boolean b = cls.isInstance(new A());
}
catch (Throwable e) {
    System.err.println(e);
}
```

Из объекта `Class` вы можете получить его имя в двух версиях.

```
Class aClass = ... //obtain Class object
String className = aClass.getName();
String simpleClassName = aClass.getSimpleName();

int modifiers = aClass.getModifiers();

Package package = aClass.getPackage();

Class superclass = aClass.getSuperclass();

Class[] interfaces = aClass.getInterfaces();

Constructor[] constructors = aClass.getConstructors();

Method[] method = aClass.getMethods();

Field[] method = aClass.getFields();

Annotation[] annotations = aClass.getAnnotations();
```

Полноценное имя класса (включая имя пакета) получается с помощью метода `getName`.

Если вы хотите имя класса без имени пакета, вы можете получить его с помощью метода `getSimpleName`.

Вы можете получить доступ к модификаторам класса через объект `Class`.

Модификаторы класса – это ключевые слова «`public`», «`private`», «`static`» и т. д.

Вы получаете модификаторы класса с помощью метода `getModifiers`.

Модификаторы упаковываются в `int`, где каждый модификатор представляет собой бит, который либо установлен, либо очищен.

Вы можете проверить модификаторы, используя эти методы класса `java.lang.reflect.Modifier`.

Также вы можете получить информацию о пакете из объекта `Class` с помощью метода `getPackage`.

Из объекта `Class` вы можете получить доступ к суперклассу класса с помощью метода `getSuperclass`.

Объект суперкласса также является объектом `Class`, как и любой другой, поэтому вы можете продолжить изучение этого класса.

С помощью метода `getInterfaces` можно получить список интерфейсов, реализованных данным классом.

Класс может реализовывать множество интерфейсов. Поэтому возвращается массив.

Интерфейсы также представлены объектами `Class`.

И возвращаются только определенные интерфейсом, реализованные определенным классом.

Если суперкласс класса реализует интерфейс, но класс не указывает, что он также реализует этот интерфейс, этот интерфейс не будет возвращен в массиве.

Чтобы получить полный список интерфейсов, реализованных данным классом, вам придется рекурсивно исследовать класс и его суперклассы.

Вы можете получить доступ к конструкторам класса с помощью метода `getConstructors`.

И вы можете получить доступ к методам класса с помощью метода `getMethods`.

С помощью метода `getFields` вы можете получить доступ к полям (переменным-членам) класса.

И вы можете получить доступ к аннотациям класса, используя метод `getAnnotations`.

Как уже было сказано, конструкторы класса можно получить методом `getConstructors`.

```
Class aClass = ...//obtain class object
Constructor[] constructors = aClass.getConstructors();
Constructor constructor = aClass.getConstructor(new Class[]{String.class});

Class[] parameterTypes = constructor.getParameterTypes();

MyObject myObject = (MyObject)constructor.newInstance("constructor-arg1");
```

Массив конструкторов будет иметь один экземпляр `Constructor` для каждого публичного конструктора, объявленного в классе.

Если вы знаете точные типы параметров конструктора, к которому хотите получить доступ, вы можете получить именно этот конструктор, а не массив всех конструкторов.

В этом примере возвращается публичный конструктор класса, который принимает параметр типа `String`.

Если нет конструктора, соответствующего заданным аргументам, генерируется исключение `NoSuchMethodException`.

Вы можете прочитать, какие параметры принимает данный конструктор, используя метод `getParameterTypes`.

Вы можете создать экземпляр объекта, используя конструктор и метод `newInstance`.

Этот метод принимает необязательное количество параметров, но вы должны указать только один параметр для каждого аргумента в вызываемом конструкторе.

В этом примере это конструктор, принимающий строку, поэтому должна быть передана одна строка.

```
Class aClass = ...//obtain class object
Field[] fields = aClass.getFields();

Field field = aClass.getField("someField");

String fieldName = field.getName();

Object fieldType = field.getType();

MyObject objectInstance = new MyObject();

Object value = field.get(objectInstance);

field.set(objectInstance, value);
```

Используя `Java Reflection`, вы можете проверять поля (переменные-члены) классов и получать / устанавливать их во время выполнения.

Это выполняется с помощью класса `Field`.

Класс `Field` получается из объекта `Class` с помощью метода `getFields`.

Массив `Field` будет иметь один экземпляр `Field` для каждого публичного поля, объявленного в классе.

Если вы знаете имя поля, к которому хотите получить доступ, вы можете получить доступ к нему, передавая в метод имя поля.

Если не существует поля с этим именем, генерируется исключение `NoSuchFieldException`.

После того как вы получили экземпляр `Field`, вы можете получить его имя поля, используя метод `Field.getName`.

И вы можете определить тип поля (`String`, `int` и т. д.), используя метод `Field.getType`.

После получения ссылки на поле вы можете получить и установить его значения с помощью методов `Field.get` и `Field.set`.

Параметр `objectInstance`, передаваемый методу `get` и `set`, должен быть экземпляром класса, которому принадлежит это поле.

Если поле является статическим полем (`public static ...`), передается `null` в качестве параметра методам `get` и `set` вместо параметра `objectInstance`.

```
Class aClass = ...//obtain class object
Method[] methods = aClass.getMethods();

Method method = aClass.getMethod("doSomething", new
Class[] {String.class});

Method method = aClass.getMethod("doSomething", null);

Class[] parameterTypes = method.getParameterTypes();

Class returnType = method.getReturnType();

Object returnValue = method.invoke(null, "parameter-value1");
```

Используя `Java Reflection`, вы можете исследовать методы классов и вызывать их во время выполнения.

Это делается с помощью класса `Java java.lang.reflect.Method`.

Класс `Method` получен из объекта `Class`, используя метод `getMethods`.

Массив `Method []` будет иметь один экземпляр `Method` для каждого публичного метода, объявленного в классе.

Если вы знаете точные типы параметров метода, к которому хотите получить доступ, вы можете получить этот метод, а не получать массив всех методов.

В этом примере возвращается публичный метод, который принимает параметр типа `String`.

Если ни один метод не соответствует указанному имени и аргументам метода, генерируется исключение `NoSuchMethodException`.

Если метод, к которому вы пытаетесь получить доступ, не принимает никаких параметров, передается `null` в качестве массива типов параметров.

Вы можете прочитать, какие параметры использует данный метод, с помощью вызова метода `getParameterTypes`.

Вы можете получить доступ к типу возвращаемого значения методом с помощью вызова метода `getReturnType`.

Также, вы можете вызывать метод, используя вызов метода `invoke`.

Первый параметр здесь – это объект, для которого вы хотите вызвать метод.

Если этот метод статический, вы указываете `null` вместо объекта.

```
public class PrivateObject {
    private String privateString = null;

    public PrivateObject(String privateString) {
        this.privateString = privateString;
    }
}

PrivateObject privateObject = new PrivateObject("The Private Value");
Field privateStringField = PrivateObject.class.getDeclaredField("privateString");
privateStringField.setAccessible(true);
String fieldValue = (String) privateStringField.get(privateObject);
System.out.println("fieldValue = " + fieldValue);
```

Несмотря на то, что модификатор `private` призван скрывать информацию, на самом деле можно получить доступ к приватным полям и методам других классов с помощью `Java Reflection`.

Это может быть очень удобно для модульного тестирования.

Однако есть возможность отключения доступа к приватным полям с помощью отражения в будущих версиях Java.

Чтобы получить доступ к приватному полю, вам нужно вызвать метод `Class.getDeclaredField` или `Class.getDeclaredFields`.

Методы `Class.getField` и `Class.getFields` возвращают только публичные поля, поэтому они не будут работать.

В этом примере кода мы получаем значение приватного поля и печатаем его.

Вызов этих методов возвращает только поля, объявленные в этом конкретном классе, а не поля, объявленные в любых суперклассах.

Вызывая метод `setAccessible(true)`, вы отключаете проверку доступа для этого конкретного экземпляра поля для отражения.

Теперь вы можете получить к нему доступ, даже если поле является приватным.

Но вы по-прежнему не можете получить доступ к полю, используя обычный код, только отражение.

```
public class PrivateObject {
    private String privateString = null;

    public PrivateObject(String privateString) {
        this.privateString = privateString;
    }

    private String getPrivateString() {
        return this.privateString;
    }
}

PrivateObject privateObject = new PrivateObject("The Private Value");
Method privateStringMethod = PrivateObject.class.getDeclaredMethod("getPrivateString", null);
privateStringMethod.setAccessible(true);
String returnValue = (String) privateStringMethod.invoke(privateObject, null);
System.out.println("returnValue = " + returnValue);
```

Чтобы получить доступ к приватному методу, вам необходимо вызвать метод `Class.getDeclaredMethod` или `Class.getDeclaredMethods`.

Методы `Class.getMethod` и `Class.getMethods` возвращают только публичные методы, поэтому они не будут работать.

Методы `getDeclaredMethod` или `getDeclaredMethods` возвращают только методы, объявленные в этом конкретном классе, а не методы, объявленные в любых суперклассах.

Вызывая метод `setAccessible (true)`, вы отключаете проверку доступа для этого конкретного метода для отражения.

Но вы все еще не можете получить доступ к этому методу с использованием обычного кода, только с помощью отражения.

```
public class MyClass {
    protected List<String> stringList = ...;

    public List<String> getStringList(){
        return this.stringList;
    }
}

Method method = MyClass.class.getMethod("getStringList", null);
Type returnType = method.getGenericReturnType();

if(returnType instanceof ParameterizedType){
    ParameterizedType type = (ParameterizedType) returnType;
    Type[] typeArguments = type.getActualTypeArguments();
    for(Type typeArgument : typeArguments){
        Class typeArgClass = (Class) typeArgument;
        System.out.println("typeArgClass = " + typeArgClass);
    }
}
```

Существует мнение, что вся информация о Java дженериках удаляется во время компиляции, чтобы вы не могли получить доступ к какой-либо из этой информации во время выполнения.

Это не совсем верно.

Во многих случаях можно получить доступ к информации о дженериках во время выполнения.

Использование Java дженериков обычно относится к одной из двух ситуаций:

Это объявление класса или интерфейса как параметризируемого.

И это использование параметризируемого класса.

Когда вы пишете класс или интерфейс, вы можете указать, что он должен быть параметризуемым.

При проверке самого параметризируемого типа во время выполнения через отражение, нет способа узнать, для какого типа была параметризация.

Сам объект не знает, для чего он был параметризован.

Однако ссылка на объект знает, на который тип идет ссылка, включая тип дженерика.

То есть, если это не локальная переменная, и, если объект ссылается на поле в объекте, вы можете посмотреть объявление поля через отражение, чтобы получить информацию о типе дженерика, объявленном этим полем.

То же самое можно сделать, если объект ссылается на параметр в методе.

Через объект `Parameter` отражения для этого метода, вы можете увидеть, какой тип дженерика этот параметр объявляет.

Наконец, вы также можете посмотреть тип возвращаемого значения метода, чтобы узнать, какой тип дженерика объявляется.

Опять же, вы не можете увидеть его из фактического объекта, который был возвращен.

Вы должны посмотреть объявление метода через отражение, чтобы увидеть, какой тип возвращаемого значения, включая тип дженерика, объявляется.

Таким образом, вы можете видеть только из объявлений ссылок (поля, параметры, типы возвращаемых данных), какой дженерик тип имеет объект, на который ссылаются эти ссылки.

Вы не можете увидеть это из самого объекта.

В приведенном примере мы получаем объект `Method`, чтобы получить информацию о его дженерик типе возвращаемого значения.

Другими словами, можно обнаружить, что метод `getStringList` возвращает `List <String>`, а не просто `List`.


```

public class MyClass {
    protected List<String> stringList = ...;

    public void setStringList(List<String> list){
        this.stringList = list;
    }

    method = MyClass.class.getMethod("setStringList", List.class);

    Type[] genericParameterTypes = method.getGenericParameterTypes();

    for(Type genericParameterType : genericParameterTypes){
        if(genericParameterType instanceof ParameterizedType){
            ParameterizedType aType = (ParameterizedType) genericParameterType;
            Type[] parameterArgTypes = aType.getActualTypeArguments();
            for(Type parameterArgType : parameterArgTypes){
                Class parameterArgClass = (Class) parameterArgType;
                System.out.println("parameterArgClass = " + parameterArgClass);
            }
        }
    }
}

```

В этом примере, мы получаем доступ к дженерик типам параметров метода во время выполнения через Java Reflection.

В этом примере, мы получаем доступ к дженерик типам публичных полей.

```

public class MyClass {
    public List<String> stringList = ...;
}

Field field = MyClass.class.getField("stringList");

Type genericFieldType = field.getGenericType();

if(genericFieldType instanceof ParameterizedType){
    ParameterizedType aType = (ParameterizedType) genericFieldType;
    Type[] fieldArgTypes = aType.getActualTypeArguments();
    for(Type fieldArgType : fieldArgTypes){
        Class fieldArgClass = (Class) fieldArgType;
        System.out.println("fieldArgClass = " + fieldArgClass);
    }
}

```

И наконец, применительно к Java 9, с помощью отражения можно получить информацию о Java модуле.

```

Module myClassModule = MyClass.class.getModule();

boolean isNamed = myClassModule.isNamed();

boolean isOpen = myClassModule.isOpen();

ModuleDescriptor descriptor = myClassModule.getDescriptor();

String moduleName = descriptor.name();

Set<ModuleDescriptor.Exports> exports = descriptor.exports();

boolean isAutomatic = descriptor.isAutomatic();

boolean isOpen = descriptor.isOpen();

Set packages = descriptor.packages();

Set<String> uses = descriptor.uses();

```

Концепция модулей Java была добавлена в Java 9.

И Java-модуль представляет собой набор пакетов Java.

Таким образом, каждый класс Java относится к пакету, а пакет принадлежит модулю.

Java-модуль представлен классом Java-отражения `java.lang.Module`.

С помощью этого класса вы можете получить информацию о данном модуле.

Вы можете получить экземпляр класса модуля через экземпляр `Class`.

Вы можете проверить, является ли экземпляр модуля именованным модулем, вызвав метод `isNamed` или метод `isOpen`.

Как только вы получите доступ к экземпляру модуля, вы можете получить доступ к его дескриптору с помощью метода `getDescriptor`.

Далее вы можете прочитать информацию в дескрипторе модуля.

Вы можете получить имя именованного модуля из его дескриптора с помощью метода `name`.

И вы можете прочитать список пакетов, экспортированных модулем Java через отражение Java, с помощью метода `exports`.

Вы можете проверить, является ли модуль Java автоматическим модулем или нет с помощью метода `isAutomatic`.

Вы можете проверить, является ли модуль Java открытым модулем или нет с помощью метода `isOpen`.

И вы можете получить список имен пакетов в данном Java-модуле с помощью метода `packages`.

Также вы можете прочитать, какие сервисы использует данный модуль Java, используя метод `uses`.



Лямбда-выражения. Синтаксис лямбда



Tech Solutions

Объектно-ориентированное программирование на Java

Лямбда-выражения

Лекция 1

Синтаксис лямбда

Зачем понадобились лямбда в Java?

Лямбда-выражения обеспечивают краткий синтаксис, являются более краткими и понятными, по сравнению с анонимными внутренними классами.

Устраняют такие недостатки анонимных внутренних классов, как громоздкость и путанность с `this` и именованием вообще, отсутствие доступа к не финальным локальным переменным и трудности с оптимизацией.

Лямбда-выражения удобны для новой библиотеки потоков.

В Java 7 код выглядит следующим образом.

```
В Java 7:
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        doSomethingWith(e);
    }
});

В Java 8:
button.addActionListener(e -> doSomethingWith(e));
```

В Java 8 этот же код выглядит гораздо компактнее.

Выражения Lambda облегчают функциональное программирование и значительно упрощают разработку.

Функциональное программирование основывается на взаимодействии с функциями, то есть некими процессами, описывающими связь между входными и выходными параметрами.

Когда используется функциональное программирование, многие проблемы легче решать и приводить к коду, который более понятен для чтения и проще поддерживается.

Функциональное программирование не заменяет объектно-ориентированное программирование в Java 8.

ООП по-прежнему является основным подходом к представлению типов. Но функциональное программирование дополняет и улучшает многие методы и алгоритмы.

Java 8 вводит понятие потоков.

Потоки – это обертки вокруг источников данных (массивов, коллекций и т. д.), которые используют лямбда, обеспечивают методы обработки элементов, используют отложенное вычисление и которые можно сделать параллельными автоматически.

Невозможно сделать параллельную обработку автоматически для цикла `for`

```
for(Employee e: employees) { e.giveRaise(1.15); }

employees.stream().parallel().forEach(e -> e.giveRaise(1.15));
```

Но этот же код будет автоматически запускаться параллельно в потоке с методом `stream`
Синтаксис лямбда выглядит как функция:

```
Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());

taskList.execute(() -> downloadSomeFile());

someButton.addActionListener(event -> handleButtonClick());

double d = MathUtils.integrate(x -> x*x, 0, 100, 1000);
```

Ожидаемый тип в качестве аргумента должен быть интерфейсом, который имеет ровно один (абстрактный) метод

Такой интерфейс называется функциональным интерфейсом или интерфейсом с одним абстрактным методом

Обозначение одного метода «абстрактный» не является избыточным, поскольку интерфейсы в Java 8 могут иметь методы, называемые «методы по умолчанию».

Интерфейсы Java 8 также могут иметь статические методы.

Рассмотрим пример сортировки строк по длине.

```
Arrays.sort(testStrings, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return(s1.length() - s2.length());
    }
});

Java 8
Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());
```

Вверху показан код Java 7, где мы переопределяем метод `compare` интерфейса `Comparator` для сравнения двух строк из массива строк по длине для сортировки массива строк.

Из программного интерфейса API, Java уже знает, что второй аргумент для `Arrays.sort` – это `Comparator`, поэтому вам не нужно говорить, что это `Comparator`.

`Comparator` имеет только один метод, поэтому вам не нужно говорить, что имя метода `compare`.

В итоге вы добавляете `"->"` стрелку между параметрами метода и телом метода.

Рассматривая первый аргумент (`testStrings`), Java может сделать вывод о том, что тип второго аргумента является `Comparator <String>`. Таким образом, параметры для сравнения – это строки. Поскольку Java это знает, вам не нужно это говорить.

```
Arrays.sort(testStrings, new Comparator<String>(){
    @Override
    public int compare(String s1, String s2) {
        return(s1.length() - s2.length());
    }
});

Arrays.sort(testStrings,
    (String s1, String s2) -> { return(s1.length() - s2.length()); });
```

Java по-прежнему делает строгую проверку типов во время компиляции.

Это похоже на то, как Java подставляет типы для оператора `<>` угловых скобок в случае дженериков.

В некоторых случаях типы являются неоднозначными, и компилятор предупреждает вас о том, что он не может подставить типы. В этом случае вы не можете отбрасывать объявления типов, как показано на слайде.

```
List <String> words = new ArrayList <> ();
```

Если тело метода может быть записано как одно выражение `return`, вы можете отбросить фигурные скобки и оператор «`return`», и просто поместить возвращаемое значение как выражение.

```
Arrays.sort(testStrings,  
    (String s1, String s2) -> { return(s1.length() - s2.length()); });
```

```
Arrays.sort(testStrings,  
    (s1, s2) -> { return(s1.length() - s2.length()); });
```

Это не всегда можно сделать, особенно если вы используете циклы или выражения. Однако, лямбда чаще всего используются, когда тело метода короткое.

```
Arrays.sort(testStrings,  
    (s1, s2) -> {return(s1.length() - s2.length());});
```

```
Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());
```

Если нет, оставляйте фигурные скобки и «return»

И, если тело лямбда длинное, вам может потребоваться использовать обычный внутренний класс.

Если метод интерфейса имеет ровно один параметр, то скобки необязательные.

Здесь показана трансформация от Java 7 к Java 8 со скобками, и мы отбрасываем скобки для одного аргумента.

```
button.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        doSomethingWith(e);  
    }  
});  
  
button.addActionListener((e) -> doSomethingWith(e));  
  
button.addActionListener(e -> doSomethingWith(e));
```

Таким образом, лямбда существенно сокращают код.

Под капотом, выполняя показанный на слайде код.

```

Java 7
taskList.execute(new Runnable(){
    @Override
    public void run() {
        processSomeImage(imageName);
    }
});
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent event) {
        doSomething(event);
    }
});
Java 8
taskList.execute(() -> processSomeImage(imageName));
button.addActionListener(event -> doSomething(event));

```

Здесь вы используете сокращение для представления экземпляра класса, который реализует интерфейс `Comparator <T>`.

```
Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());
```

Вы предоставляете тело метода `compare` после «->».

Java 8 технически не имеет типов функций, поскольку под капотом лямбда становятся экземплярами классов, которые реализуют любой интерфейс.

Тем не менее, вы обычно думаете о лямбдах как о функциях.

Где можно использовать лямбда?

Найдите любую переменную или параметр, который ожидает интерфейс, имеющий один метод.

Такие интерфейсы с одним методом называются «функциональные интерфейсы» или «SAM (Single Abstract Method)»

Здесь у нас есть интерфейс

```

public interface Blah { String foo(String someString); }

public void someMethod (Blah b) { ... b.foo (...) ...}

someMethod (s -> s.toUpperCase ()) + "!";

```

И у нас есть код, который использует этот интерфейс:

Таким образом код, вызывающий метод, будет указывать реализацию интерфейса с помощью стрелки.

Дальнейшее упрощение лямбда выражения возможно с помощью ссылки на метод.

```
(args) -> ClassName.staticMethodName(args)

ClassName::staticMethodName
(Math::cos, Arrays::sort, String::valueOf)

variable::instanceMethod
(System.out::println)

Class::instanceMethod
(String::toUpperCase)

ClassOrType::new
(String[]::new)
```

В случае, если выражение лямбда ничего не делает, кроме вызова существующего метода.

В таких случаях часто бывает проще обратиться к существующему методу по имени.

Ссылка на метод – это сокращенный синтаксис выражения лямбда, которое выполняет только один метод.

Ссылки на методы помогают указывать на методы по их именам.

Ссылка на метод указывается символом:: (двойной двоеточие).

Ссылка на метод может использоваться для указания следующих типов методов:

Статические методы

Методы экземпляра

Конструкторы, использующие оператор new.

Таким образом, вместо лямбда мы можем писать ссылку на метод.

При использовании ссылки на метод, возникает вопрос – какой тип у аргумента метода.

```
MathUtilities.integrationTest(x -> Math.sin(x), 0, Math.PI);
MathUtilities.integrationTest(x -> Math.exp(x), 2, 20);

MathUtilities.integrationTest(Math::sin, 0, Math.PI);
MathUtilities.integrationTest(Math::exp, 2, 20);
```

Мы можем ответить на этот вопрос только одним способом – нужно смотреть документацию API.

Если вы не понимаете ссылки на методы, вы всегда можете использовать явные лямбды.

Теперь разберем правила области видимости лямбда.


```
Заменить foo(Math::cos) на foo(d -> Math.cos(d))

Заменить bar(System.out::println) на bar(s -> System.out.println(s))

Заменить baz(Class::twoArgMethod) на (a, b) -> Class.twoArgMethod(a, b)
```

В случае с лямбда, «this» переменная относится к внешнему классу, а не к анонимному внутреннему классу, в который превращается лямбда.

```
public class Application() {
    public void doWork() {
        Runnable runner = () -> { ...; System.out.println(this.toString()); ... };
        ...
    }
}
```

Здесь выражение `this.toString()` вызывает метод `toString` объекта `Application`, а не экземпляра `Runnable`.

Лямбда не может вводить «новые» переменные с тем же именем, что и переменные в методе, который использует лямбда.

Здесь показаны возможные ошибки с локальными переменными.

```
Ошибка: повторяется имя переменной
double x = 1,2;
someMethod (x -> doSomethingWith (x));

Ошибка: повторяется имя переменной
double x = 1,2;
someMethod (y -> {double x = 3.4; ...});

Ошибка: lambda модифицирует внешние локальные переменные
double x = 1,2;
someMethod (y -> x = 3.4);
```

Однако лямбда может ссылаться (но не изменять) локальные переменные окружающего метода.

Здесь лямбда использует локальные переменные метода `count` и `text`.

```
public static void repeatMessage(String text, int count) {  
    Runnable r = () -> {  
        for (int i = 0; i < count; i++) {  
            System.out.println(text);  
        }  
    };  
    new Thread(r).start();  
}
```

В этом случае говорят, что лямбда захватывает значения этих переменных.

В выражении лямбда вы можете ссылаться только на переменные, значение которых не изменяется.

Здесь выражение лямбда пытается захватить значение `i`, но это не является законным, потому что `i` меняется.

```
for (int i = 0; i < n; i++) {  
    new Thread(() -> System.out.println(i)).start();  
    // Error—cannot capture i  
}
```

Для захвата нет единственного значения.

Правило заключается в том, что выражение лямбда может обращаться только к локальным переменным внешнего кода, которые фактически являются финальными.

Эффективная финальная переменная никогда не изменяется — она либо объявляется, либо может быть объявлена финальной.

Здесь в каждой итерации создается новая переменная `arg` и назначается следующее значение из массива `args`.

```
for (String arg : args) {  
    new Thread(() -> System.out.println(arg)).start();  
    // OK to capture arg  
}
```

```
String s = "...";  
doSomething(someArg -> use(s));
```

Однако при всем при этом лямбда все еще может ссылаться (и изменять) переменные экземпляра окружающего класса.

Здесь показано изменение и использование переменных экземпляра лямбда выражением.

```
private double x = 1.2;  
public void foo() { someMethod(y -> x = 3.4); }
```

```
private double x = 1.2;  
public void bar() { someMethod(x -> x + this.x); }
```

Функциональные интерфейсы



Tech Solutions

Объектно-ориентированное программирование на Java

Лямбда-выражения

Лекция 2

Функциональные интерфейсы

Далее обсудим функциональные интерфейсы.

Мы уже видели аннотацию `@Override`.

Что дает эта аннотация `@Override`?

```
button.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent event) {  
        doSomething(event);  
    }  
});
```

Правильный код будет работать с или без аннотации `@Override`, но наличие аннотации `@Override` позволяет улавливать ошибки во время компиляции.

Например, если вы допустили ошибку в имени метода.

Кроме того, аннотация сообщает другим разработчикам, что это метод, который наследуется из родительского класса, и нужно смотреть документацию API для объекта родительского класса.

Поэтому для функциональных интерфейсов, с которыми мы познакомились, полезно использовать аннотацию `@FunctionalInterface`.

Эта аннотация улавливает ошибки во время компиляции.

```
@FunctionalInterface  
public interface ShortToByteFunction {  
  
    byte applyAsByte(short s);  
  
}
```

Если позже разработчик добавляет второй абстрактный метод, интерфейс не будет компилироваться.

И эта аннотация сообщает коллегам-разработчикам, что это интерфейс, который используется в лямбда.

Но, также, как и с аннотацией `@Override`, вы можете использовать лямбда с любым интерфейсом, который имеет один абстрактный метод, независимо от того, использует ли этот интерфейс аннотацию `@FunctionalInterface`.

Также Java 8 предлагает целый пакет `java.util.function` встроенных функциональных интерфейсов.

```
public interface Op {
    void runOp();
}
```

```
@FunctionalInterface
public interface Op {
    void runOp();
}
```

Пакет `java.util.function` предоставляет набор повторно используемых общих функциональных интерфейсов (и их соответствующих лямбда) определений, которые могут быть использованы программистом в его коде вместо создания новых функциональных интерфейсов.

Package java.util.function
Functional interfaces provide target types for lambda expressions and method references.
See: [Description](#)

Interface	Description
<code>BiConsumer<T,U></code>	Represents an operation that accepts two input arguments and returns no result.
<code>BiFunction<T,U,R></code>	Represents a function that accepts two arguments and produces a result.
<code>BinaryOperator<T></code>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
<code>BiPredicate<T,U></code>	Represents a predicate (boolean-valued function) of two arguments.
<code>BooleanSupplier</code>	Represents a supplier of boolean-valued results.
<code>Consumer<T></code>	Represents an operation that accepts a single input argument and returns no result.
<code>DoubleBinaryOperator</code>	Represents an operation upon two double-valued operands and producing a double-valued result.
<code>DoubleConsumer</code>	Represents an operation that accepts a single double-valued argument and returns no result.
<code>DoubleFunction<R></code>	Represents a function that accepts a double-valued argument and produces a result.
<code>DoublePredicate</code>	Represents a predicate (boolean-valued function) of one double-valued argument.

Например, когда нам нужно проверить условие и вернуть логическое значение, лямбда будет `(T) -> boolean`, где `T` – это параметр абстрактного метода, а `boolean` – возвращаемое значение.

(T) -> boolean

`@FunctionalInterface`
`public interface Predicate<T>`
 Represents a predicate (boolean-valued function) of one argument.
 This is a functional interface whose functional method is `test(Object)`.
 Since: 1.8

Method Summary	
All Methods	Static Methods
default <code>Predicate<T></code>	<code>and(Predicate<? super T> other)</code> Returns a composed predicate that represents a short-circuiting logical AND of this predicate and the other predicate.
static <T> <code>Predicate<T></code>	<code>isEqual(Object targetObj)</code> Returns a predicate that tests if two arguments are equal.
default <code>Predicate<T></code>	<code>negate()</code> Returns a predicate that represents the logical negation of this predicate.
default <code>Predicate<T></code>	<code>or(Predicate<? super T> other)</code> Returns a composed predicate that represents a short-circuiting logical OR of this predicate and the other predicate.
boolean	<code>test(T t)</code> Evaluates this predicate on the given argument.

Теперь, когда нам нужно использовать такую лямбду, мы можем использовать встроенный функциональный интерфейс `Predicate`, потому что у него есть метод `(T) -> boolean`.

Затем в нашем коде мы можем написать метод, который принимает этот функциональный интерфейс следующим образом:

Здесь мы передаем в метод число и реализацию метода `test` интерфейса `Predicate`, который проверяет, число больше 7 или нет.

```
process(10, (i) -> i > 7);

void process(int number, Predicate<Integer> predicate) {
    if (predicate.test(number)) {
        System.out.println("Number " + number + " was accepted!");
    }
}
```

Другой распространенный функциональный интерфейс – это `Consumer`.

T -> ()

`@FunctionalInterface`
`public interface Consumer<T>`
 Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, `Consumer` is expected to operate via side-effects.
 This is a functional interface whose functional method is `accept(Object)`.
 Since: 1.8

Method Summary	
All Methods	Static Methods
void	<code>accept(T t)</code> Performs this operation on the given argument.
default <code>Consumer<T></code>	<code>andThen(Consumer<? super T> after)</code> Returns a composed <code>Consumer</code> that performs, in sequence, this operation followed by the after operation.

`Consumer` может использоваться во всех контекстах, где нужно потреблять объект, т. е. принять объект и выполнить некоторую операцию над объектом, не возвращая никакого результата.

Здесь мы определяем реализацию интерфейса `Consumer` и используем ее для распечатки элементов списка.

```

Consumer<Integer> consumer = i -> System.out.print(" "+i);

List<Integer> integerList = Arrays.asList(new Integer(1),
    new Integer(10), new Integer(200),
    new Integer(101), new Integer(-10),
    new Integer(0));
printList(integerList, consumer);

public static void printList(List<Integer> listOfIntegers, Consumer<Integer> consumer){
    for(Integer integer:listOfIntegers){
        consumer.accept(integer);
    }
}

```

Здесь также показано, что с помощью функционального интерфейса можно сохранять лямбда выражение как объект, и затем повторно использовать его.

Еще один распространенный функциональный интерфейс – это Function.

T -> R

@FunctionalInterface
public interface Function<T,R>
Represents a function that accepts one argument and produces a result.
This is a functional interface whose functional method is apply(Object).
Since:
1.8

Method Summary	
Modifier and Type	Method and Description
default <U> Function<T,U>	<code>andThen(Function<? super R,? extends U> after)</code> Returns a composed function that first applies this function to its input, and then applies the after function to the result.
R	<code>apply(T t)</code> Applies this function to the given argument.
default <U> Function<U,R>	<code>compose(Function<? super V,? extends R> before)</code> Returns a composed function that first applies the before function to its input, and then applies this function to the result.
static <U> Function<T,U>	<code>identity()</code> Returns a function that always returns its input argument.

Основной целью, для которой был создан интерфейс Function, это отображение, когда объект одного типа принимается как входной и преобразуется (или отображается) в другой тип.

Здесь с помощью реализации этого интерфейса, из объекта Employee выделяется поле name и формируется список имен из списка объектов.

```

Function<Employee, String> funcEmpToString = (Employee e) -> {return e.getName();};

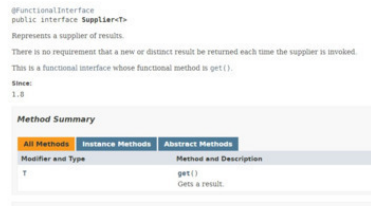
List<Employee> employeeList =
    Arrays.asList(new Employee("Tom Jones", 45),
        new Employee("Harry Major", 25),
        new Employee("Deborah Sprightly", 29));
List<String> empNameList = convertEmpListToNamesList(employeeList, funcEmpToString);
empNameList.forEach(System.out::println);

public static List<String> convertEmpListToNamesList(List<Employee> employeeList,
    Function<Employee, String> funcEmpToString){
    List<String> empNameList = new ArrayList<String>();
    for(Employee emp:employeeList){
        empNameList.add(funcEmpToString.apply(emp));
    }
    return empNameList; }

```

И еще один распространенный функциональный интерфейс – это Supplier.

() -> T



Supplier может использоваться во всех контекстах, где нет входных данных, но ожидается возврат результата.

```
Supplier<String> helloStrSupplier = () -> new String("Hello");

String helloStr = helloStrSupplier.get();

System.out.println("String in helloStr is->" + helloStr + "<-");
```

Здесь с помощью реализации этого интерфейса просто создается новая строка.

В отличие от Java 7 в Java 8, в интерфейсах можно определять статические методы и методы по умолчанию.

Для статических методов интерфейса, идея здесь заключается в том, чтобы поместить операции, связанные с данным общим типом, в интерфейс.

И это не нарушает «дух» идеи интерфейса.

При этом вызов статического метода интерфейса будет выглядеть следующим образом. Общий тип, затем вызов метода.

```
Shape.sumAreas (arrayOfShapes);

public interface Shape {

    double getArea(); // All real shapes must define a getArea

    public static double sumAreas(Shape[] shapes) {
        double sum = 0;
        for(Shape s: shapes) {
            sum = sum + s.getArea();
        }
        return(sum);
    }

}
```

При этом вы должны использовать имя интерфейса в вызове метода, даже из кода внутри класса, который реализует этот интерфейс.

Shape.sumAreas, а не просто sumAreas.

И статические методы интерфейса не могут манипулировать статическими переменными, так как интерфейсы Java 8 по-прежнему не могут содержать изменяемые поля.

Идея для методов по умолчанию интерфейсов следующая.

В Java 8 необходимо было добавить такие методы, как `stream` и `forEach` для класса `List`.

Для встроенных классов платформы Java не возникло никаких проблем, просто обновилось определение интерфейса `List` и всех встроенных классов, которые реализовали `List` (`ArrayList` и т. д.),

Большая проблема возникла для пользовательских (определяемых пользователем) классов, которые реализовали `List`: они бы дали сбой в Java 8.

И это очень серьезно нарушило бы правило, что новые версии Java не должны ломать существующий код.

Это конечно нарушает дух самой идеи интерфейсов, и теперь интерфейсы больше похожи на абстрактные классы.

Но это методы по умолчанию в интерфейсах также полезны в вашем коде.

Здесь метод по умолчанию вызывается для экземпляра класса, реализующего функциональный интерфейс.

```
Op op1 = () -> someCode(...);
Op op2 = () -> someOtherCode(...);
Op op3 = op1.combinedOp(op2);
Op.timeOp(op3);

@FunctionalInterface
public interface Op {
    void runOp();
    static void timeOp(Op operation) {
        // Unchanged from last example
    }
    default Op combinedOp(Op secondOp) {
        return () -> { runOp();
            secondOp.runOp(); };
    }
}
```

Рассмотрим следующую ситуацию.

```
public interface Int1 { int someMethod(); }
public interface Int2 { int someMethod(); }

public class ParentClass {
    public int someMethod() { return(3); }
}

public class SomeClass implements Int1, Int2 { ... }
public class ChildClass extends ParentClass implements Int1 { ... }
```

Предположим, что у нас есть два интерфейса и класс с методом, имеющим одинаковую сигнатуру.

Если какой-либо класс реализует оба интерфейса, в Java 7, не будет никакого конфликта.

Этот класс должен определить этот метод, и тем самым это будет соответствовать требованиям реализации методов интерфейсов.

Если какой-либо класс расширит класс и реализует интерфейс, в Java 7, не будет никакого конфликта.

Дочерний класс наследует этот метод из родительского класса, и тем самым это будет соответствовать требованию реализации метода интерфейса.

Теперь, предположим, что у нас в интерфейсах есть методы по умолчанию.

```
public interface Int1 { default int someMethod() { return(5); } }
public interface Int2 { default int someMethod() { return(7); } }

public class ParentClass {
    public int someMethod() { return(3); }
}

public class SomeClass implements Int1, Int2 { ... }
public class ChildClass extends ParentClass implements Int1 { ... }
```

И если какой-либо класс реализует оба интерфейса, тогда какой метод будет работать – первого интерфейса или второго.

И если какой-либо класс расширит класс и реализует интерфейс, тогда какой метод будет работать – родительского класса или интерфейса.

В случае реализации двух интерфейсов с одинаковыми методами по умолчанию, конфликт не может быть разрешен автоматически, и класс должен предоставить новое определение метода.

И при этом, в новом методе можно вызывать методы по умолчанию интерфейсов, используя ключевое слово супер.

В случае расширения класса и реализации интерфейса с одинаковыми методами по умолчанию, конфликт разрешается следующим образом, будет работать версия из родительского класса.

```
public class car implements vehicle, fourWheeler {
    default void print(){
        vehicle.super.print();
    }
}
```

Это правило также означает, что интерфейсы не могут предоставлять стандартные реализации для методов класса Object (например, toString).

Таким образом, методы по умолчанию используются для добавления поведения к существующим интерфейсам без нарушения классов, которые уже реализовали интерфейс.

Потоки Stream



Tech Solutions

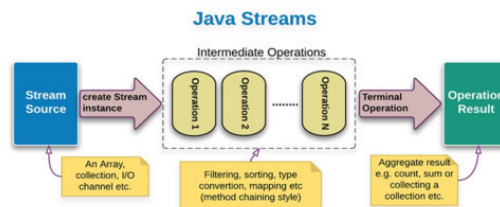
Объектно-ориентированное программирование на Java

Лямбда-выражения

Лекция 3

Потоки Stream

Мы уже познакомились с потоками в коллекциях.
Потоки имеют более удобные методы, чем списки.



Это такие методы, как `forEach`, `filter`, `map`, `reduce`, `min`, `sorted`, `distinct`, `limit`, и т. д.

Потоки более эффективно расходуют память, чем списки.

Потоки обеспечивают отложенные вычисления, поскольку промежуточные операции не вычисляются, если не задействована завершающая операция, потоки обеспечивают автоматическое распараллеливание, а также бесконечные (неограниченные) потоки.

Потоки не хранят данные.

Это всего лишь программные обертки вокруг существующих источников данных.

Это обычно списки или массивы, но также вы можете использовать функцию в качестве источника данных (`Stream.generate`), и функция вызывается каждый раз, когда вам нужен вход потока.

Потоки не хранят значения, они несут значения из источника через применяемые операции.

Потоки также никогда не изменяют базовую структуру данных (например, список или массив, для которого поток служит оберткой).

И все операции потока принимают лямбда в качестве аргумента.

Существуют три распространенных способа создать поток.

Из списка, из массива объектов и из отдельных элементов.

```
someList.stream();

Stream.of(arrayOfObjects);// [not array of primitives!]

Stream.of(val1, val2, ...);

List<String> words = ...;
words.stream().map(...).filter(...).other(...);

List<Employee> workers = ...;
workers.stream().map(...).filter(...).other(...);

Employee[] workers = ...;
Stream.of(workers).map(...).filter(...).other(...);

Employee e1 = ...;
Employee e2 = ...;
Stream.of(e1,e2,...).map(...).filter(...).other(...);
```

Остальные способы создания потока.

Из функции, из файла, из сборщика, из строки и из другого потока.

```
Stream.generate, Stream.iterate

Files.lines(somePath)

someBuilder.build()

String.chars, Stream.of(someString.split(...))

distinct, filter, limit, map, sorted, skip
```

Завершающей операцией можно перевести поток в список или массив.

```
someStream.collect(Collectors.toList())
List<SomeType> values =
someStream.map(...).filter(...).map(...).filter(...).collect(...);

someStream.toArray(EntryType[]::new)
employeeStream.toArray(Employee[]::new);
```

В основе потоков лежит следующая идея.

```
List<String> wordList =
someStream.map(someLambdaThatReturnsString)
.filter(someTestOnStrings)
.collect(Collectors.toList());

String[] wordArray =
someStream.map(someLambdaThatReturnsString)
.filter(someTestOnStrings)
.toArray(String[]::new);
```

Вы обворачиваете поток вокруг массива или списка (или даже файла).

Затем вы можете выполнять операции над каждым элементом (`forEach`), создавать новый поток, преобразовывая каждый элемент, удаляя элементы, которые не совпадают с некоторыми критериями и т. д.

И финальной операцией возвращать новый список или массив.

Основные операции потоков показаны на слайде.

```

• forEach(Consumer)
– employees.forEach(e -> e.setSalary(e.getSalary() * 11/10))
• map(Function)
– ids.map(EmployeeUtils::findEmployeeById)
• filter(Predicate)
– employees.filter(e -> e.getSalary() > 500000)
• findFirst()
– employees.filter(...).findFirst().orElse(defaultValue)
• toArray(ResultType[]::new)
– Employee[] empArray = employees.toArray(Employee[]::new);
• collect(Collectors.toList())
– List<Employee> empList =
  employees.collect(Collectors.toList());

```

Вы можете выполнять только одну цепочку операций для каждого потока.

Однако, вам ничего не стоит перестрими́ть список, потому что создание потока просто указывает на существующую структуру данных за кулисами; оно не копирует данные.

В основе метода `forEach` лежит идея создать простой способ перебора элементов в цикле.

forEach

```

Stream.of(someArray).forEach(System.out::println);

fieldList.stream().forEach(field -> field.setText(""));

```

И надо заметить, что в Java 8 метод `forEach` есть также просто для списков, для самих структур данных, а не только для потоков.

Вы предоставляете функцию (как лямбда) для каждого элемента, и эта функция вызывается для каждого элемента потока.

Точнее, вы предоставляете реализацию интерфейса `Consumer` для метода `forEach`, и каждый элемент `Stream` передается методу `accept` этого `Consumer`.

Операция `forEach` является терминальной или завершающей операцией для потока.

Поэтому эту операцию нельзя применить дважды, к одному и тому же потоку.

Обычно, `forEach` применяют для распечатки элементов потока.

Также эту операцию можно применить для создания новой структуры данных из элементов потока или модификации существующей структуры данных.

В отличие от обычного цикла, в `forEach` нельзя использовать операторы `break` или `return` для досрочного завершения цикла.

Здесь на слайде показано создание трех потоков и применение к ним операции `forEach`.

```
List<Employee> googlers = EmployeeSamples.getGooglers();  
googlers.stream().forEach(System.out::println);  
googlers.stream().forEach(e -> e.setSalary(e.getSalary() * 11/10));  
googlers.stream().forEach(System.out::println);
```

Можно объединить эти три операции в одно повторно используемое лямбда выражение.

```
Consumer<Employee> giveRaise = e -> {  
    System.out.printf("%s earned $%,d before raise.%n",  
        e.getFullName(), e.getSalary());  
  
    e.setSalary(e.getSalary() * 11/10);  
  
    System.out.printf("%s will earn $%,d after raise.%n",  
        e.getFullName(), e.getSalary());  
};  
  
googlers.stream().forEach(giveRaise);  
sampleEmployees.stream().forEach(giveRaise);
```

За операцией `map` потоков стоит следующая идея.

`map`

```
Double[] nums = { 1.0, 2.0, 3.0, 4.0, 5.0 };  
  
Double[] squares =  
    Stream.of(nums).map(n -> n * n).toArray(Double[]::new);
```

Создать новый поток, который является результатом применения функции к каждому элементу из первоначального потока.

Существует также аналогичный метод `replaceAll` непосредственно в `List`, но там входные и выходные элементы принадлежат одному списку.

Другие, относящиеся к отображению методы, это `mapToInt`, `mapToDouble` и `flatMap`.

```

• mapToInt
String[] s = {"one", "two", "three", "four"};
Stream<String> stringStream = Stream.of(s);

• mapToDouble
IntStream intStream = stringStream.mapToInt(e -> e.length());

• flatMap
IntSummaryStatistics stats =
intStream.peek(System.out::println).summaryStatistics();
System.out.println(stats);

List<String> petNames = humans.stream()
.map(human -> human.getPets()) //преобразовываем Stream<Human> в Stream<List<Pet>>
.flatMap(pets -> pets.stream()) //разворачиваем Stream<List<Pet>> в Stream<Pet>
.collect(Collectors.toList());

```

`mapToInt` применяет функцию, которая создает `Integer`, но полученный поток представляет собой `IntStream`, вместо `Stream <Integer>`. Это удобно, потому что `IntStream` имеет такие методы, такие как сумма, минимальный и максимальный элементы.

`mapToDouble` аналогичен `mapToInt`, но создает `DoubleStream`.

В случае `flatMap` – каждое применение функции создает поток, а затем все элементы потоков объединяются в один поток.

За методом `filter` потока стоит следующая идея.

filter

```

Integer[] nums = { 1, 2, 3, 4, 5, 6 };

Integer[] evens = Stream.of(nums).filter(n -> n%2 == 0)
.toArray(Integer[]::new);

```

Создать новый поток, содержащий только те элементы исходного потока, которые пройдут данный тест (предикат).

Существует аналогичный метод (`removeIf`) непосредственно в списке `List`, но фильтр потока сохраняет записи, которые проходят тест, тогда как `removeIf` удаляет те записи, которые проходят тест.

Методы `findAny ()` и `findFirst ()` потока возвращают экземпляр класса `Optional`.

Что это за класс.

```

List<String> list = Arrays.asList("A", "B", "C", "D");

Optional<String> result = list.stream().findAny();

Optional<String> result = list.stream().findFirst();

```

```

java.util
Class Optional<T>
java.lang.Object
  java.util.Optional<T>
    public final class Optional<T>
      extends Object
      A container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value.
      Additional methods that depend on the presence or absence of a contained value are provided, such as orElse() (return a default value if value not present) and ifPresent() (execute a block of code if the value is present).
      This is a value-based class; use of identity-sensitive operations (including reference equality (==), identity hash code, or synchronization) on instances of Optional may have unpredictable results and should be avoided.
      Since:
      1.8

```

Объект `Optional` либо хранит какое-либо значение, либо ничего не сохраняет. Он предназначен для методов, которые могут или не могут найти значение.

Так вот метод `findFirst` потока возвращает объект `Optional` для первой записи в потоке.

И так как потоки часто являются результатом фильтрации, первой записи может не быть, поэтому объект `Optional` может быть пустым.

Метод `orElse(other)` – возвращает значение, если оно присутствует, или возвращает `other`.

```
findFirst
```

```
someStream.filter(...).findFirst().orElse(otherValue)
```

Идея метода `findFirst` состоит в том, что `map` и `filter` должны знать, что нужно остановиться после того, как первая запись потока найдена.

Метод `findAny` позволяет вам найти любой элемент из потока `Stream`. Используйте его, когда вы ищете элемент, не обращая внимания на порядок.

Рассмотрим следующий код.

Вопрос – сколько раз будет вызван метод `getSalary` и будет ли разница, если будет 10,000,000 записей потока, вместо 10,000.

```
Integer[] ids = { 16, 8, 4, ... }; // 10,000 entries

System.out.printf("First Googler with salary over $500K: %s%n",
    Stream.of(ids).map(EmployeeSamples::findGoogler)
        .filter(e -> e != null)
        .filter(e -> e.getSalary() > 500_000)
        .findFirst()
        .orElse(null));
```

Ответ метод `getSalary` будет вызван один раз и никакой разницы нет между 10,000,000 записями потока и 10,000.

Это и есть отложенная оценка или отложенные вычисления потока.

Потоки откладывают выполнение большинства операций, пока вам не понадобятся результаты.

Операции, которые, как кажется, должны проходить по потоку несколько раз, фактически выполняются только один раз.

Например, если посмотреть на такую последовательность операций потока.


```
stream.map(someOp).filter(someTest).findFirst().get()
```

Сначала выполняется операция `map`, затем фильтр первого элемента, затем операция `map`, затем фильтр второго элемента, до тех пор, пока не обнаружится первое соответствие фильтру.

Все операции потока можно разделить на три группы.

```
Промежуточные
– map (and related mapToInt, flatMap, etc.), filter, distinct, sorted, peek,
limit, skip,
parallel, sequential, unordered

Терминальные
– forEach, forEachOrdered, toArray, reduce, collect, min, max, count,
anyMatch,
allMatch, noneMatch, findFirst, findAny, iterator

Замыкающие
– anyMatch, allMatch, noneMatch, findFirst, findAny, limit, skip
```

Промежуточные операции – это методы, которые производят другие потоки. Эти методы не обрабатываются, пока не будет вызван какой-либо терминальный метод.

Терминальные операции – после вызова одного из этих методов поток считается потребленным и для него не может быть выполнено никаких больше операций.

Замыкающие операции – эти методы позволяют обрабатывать предыдущие промежуточные методы только до тех пор, пока не будет вычислен замыкающий метод.

И замыкающие операции могут быть промежуточными или терминальными.

Методы `limit` и `skip` потока ограничивают размер потока.

```
strm.map(func1).filter(pred).map(func2).limit(10)
```

```
Первые 10 элементов
• someLongStream.limit (10)

Последние 15 элементов
• twentyElementStream.skip(5)
```

Метод `limit (n)` возвращает поток первых `n` элементов.

Метод `skip(n)` возвращает поток, начинающийся с элемента `n` (т. е. он отбрасывает первые `n` элементов).

Метод `limit` – это замыкающий метод. Например, если у вас есть поток из 1000 элементов, этот метод применит `funct1` ровно 10 раз, оценит предикат как минимум 10 раз и применит `funct2` не более 10 раз.

Операции потока, которые используют сравнения – это `sorted`, `min`, `max`, `distinct`.

Операции потока, которые используют сравнения:
`sorted`, `min`, `max`, `distinct`.

```
empStream.sorted((e1, e2) -> e1.getSalary() - e2.getSalary())
empStream.max((e1, e2) -> e1.getSalary() - e2.getSalary()).get()
stringStream.distinct()
```

Сортировка с помощью компаратора работает так же, как `Arrays.sort`.

Сортировка без аргументов работает только в том случае, если элементы потока `Stream` реализуют интерфейс `Comparable`.

Сортировка потоков более гибкая, чем сортировка массивов, потому что вы можете фильтровать и отображать до и / или после сортировки.

Быстрее использовать операции `min` и `max`, чем сортировать вперед или назад, а затем получить первый элемент.

Методы `min` и `max` принимают компаратор в качестве аргумента.

Метод `distinct` использует `equals` для сравнения, чтобы вернуть стрим без дубликатов.

Операции потока, которые проверяют на соответствие – это `allMatch`, `anyMatch`, `noneMatch`, `count`.

Операции потока, которые проверяют на соответствие:
`allMatch`, `anyMatch`, `noneMatch`, `count`.

```
List<Employee> googlers = EmployeeSamples.getGooglers();

boolean isNobodyPoor = googlers.stream().noneMatch(e -> e.getSalary() <
200_000);
Predicate<Employee> megaRich = e -> e.getSalary() > 7_000_000;
boolean isSomeoneMegaRich = googlers.stream().anyMatch(megaRich);
boolean isEveryoneMegaRich = googlers.stream().allMatch(megaRich);
long numberMegaRich = googlers.stream().filter(megaRich).count();
```

Методы `allMatch`, `anyMatch` и `noneMatch` берут предикат и возвращают логическое значение.

Они прекращают обработку, как только может быть определен результат.

Например, если первый элемент не соответствует предикату, метод `allMatch` немедленно вернет `false` и пропустит проверку других элементов.

Метод `count` просто возвращает количество элементов.

Метод `count` – это терминальная операция, поэтому вы не можете сначала подсчитать элементы, а затем сделать еще одну операцию над этим же потоком.

Помимо интерфейса `Stream`, в Java 8 есть такие интерфейсы, как `DoubleStream`, `IntStream`, `LongStream`.

Интерфейс `IntStream` представляет специализацию потока `Stream`, которая упрощает работу с целыми числами.

```
java.util.stream
Interface IntStream

All Superinterfaces:
    AutoCloseable, BaseStream<Integer,IntStream>

public interface IntStream
    extends BaseStream<Integer,IntStream>

A sequence of primitive int-valued elements supporting sequential and parallel aggregate operations. This is the int primitive specialization of Stream.
The following example illustrates an aggregate operation using Stream and IntStream, computing the sum of the weights of the red widgets.

    int sum = widgets.stream()
        .filter(w -> w.getColor() == RED)
        .mapToInt(w -> w.getWeight())
        .sum();

See the class documentation for Stream and the package documentation for java.util.stream for additional specification of streams, stream operations, stream pipelines, and
parallelism.
Since:
    1.8
```

Здесь сначала создается поток `Stream`, а затем метод `mapToInt` возвращает поток `IntStream`.

```
int population = countryList.stream()
    .filter(Utils::inRegion)
    .mapToInt(Country::getPopulation)
    .sum();
```

Также поток `IntStream` можно создать методами `of` и `range`, а также методом `ints` класса `Random`.

```
regularStream.mapToInt
personList.stream().mapToInt(Person::getAge)

IntStream.of
IntStream.of(int1, int2, int2)

IntStream.range, IntStream.rangeClosed
IntStream.range(5, 10)

Random.ints
new Random().ints()

IntStream i = IntStream.of(6,5,7,1, 2, 3, 3);
OptionalInt d = i.max();
```

Помимо обычных методов потока, к потоку `IntStream` можно применить целочисленные методы `min`, `max`, `sum`, `average` и `toArray`.

Аналогично работают потоки `DoubleStream` и `LongStream`.

```
DoubleStream
• regularStream.mapToDouble, DoubleStream.of, someRandom.doubles
• min, max, sum, average, toArray

LongStream
• regularStream.mapToLong, LongStream.of, someRandom.longs
• min, max, sum, average, toArray
```

Здесь показаны операции сокращения которые принимают поток Stream и объединяют или сравнивают записи для возврата одного значения.

```
findFirst().orElse(...)
findAny().orElse(...)

Stream
min(comparator), max(comparator)
reduce(starterValue, binaryOperator)
reduce(binaryOperator).orElse(...)

IntStream
min(), max(), sum(), average()
```

Методы, которые здесь показаны, мы уже рассмотрели, за исключением метода reduce.

В основе метода reduce лежит следующая идея.

Вы начинаете со значения, объединяете это значение с первой записью потока, объединяете результат со второй записью потока и т. д.

Этот метод особенно полезен в сочетании с методом map или filter.

Метод reduce (starter, binaryOperator) принимает стартовое значение и BinaryOperator. Возвращает результат напрямую.

Метод reduce (binaryOperator) принимает BinaryOperator, без стартового значения. Он начинается с объединения первых двух значений и возвращает объект Optional.

Здесь в первом примере этот метод перемножает все элементы потока.

```
nums.stream().reduce(1, (n1, n2) -> n1 * n2)

letters.stream().reduce("", String::concat);
```

Во втором примере этот метод объединяет все элементы потока в одну строку.

Параллельные и бесконечные потоки



Объектно-ориентированное программирование на Java

Лямбда-выражения

Лекция 4

Параллельные и бесконечные потоки

Далее рассмотрим параллельные потоки.

Параллельные вычисления подразумевают разделение задачи на подзадачи, решение этих задач одновременно (параллельно, при этом каждая подзадача выполняется в отдельном потоке), а затем объединение результатов решений.

Java SE предоставляет фреймворк `fork/join`, который позволяет легко реализовать параллельные вычисления и который мы рассмотрим позднее.

Однако в этом фреймворке вы должны определить, как задача разделяется на подзадачи.

С помощью параллельных потоков среда выполнения Java автоматически выполняет это разделение и затем объединение решений.

Указывая, что поток параллелен, операции автоматически выполняются параллельно, без необходимости явного кода фреймворка `fork/join` или `threading`.

Запускается параллельный поток с помощью операции `parallel ()` или `parallelStream ()`.

```
anyStream.parallel()
```

```
anyList.parallelStream()
```

Параллельные потоки используют внутри фреймворк `fork / join`, создают один поток на каждое ядро компьютера, поэтому не дают преимущества на одноядерных компьютерах, и могут использоваться с минимальными изменениями в коде.

Параллельное программирование с явными потоками часто полезны даже на одноядерных компьютерах, могут создавать гораздо больше потоков, чем у компьютера есть ядер, но требуют больших изменений в коде.

Рассмотрим следующий код.

```

List<Integer> list = new ArrayList<>();
for (int i = 0; i < 1000; i++) {
    list.add(i);
}
list.stream().forEach(System.out::println);

list.parallelStream().forEach(System.out::println);

```

Эта программа будет выводить числа от 0 до 999 последовательно, в том порядке, в котором они находятся в списке.

Если мы изменим `stream()` на `parallelStream()`, это уже будет не так – все числа будут выведены, но в другом порядке.

Поэтому, очевидно, что `parallelStream` действительно использует несколько потоков.

Большинство современных операционных систем могут разделять однопоточные приложения по нескольким ядрам процессора – части одного потока могут работать на нескольких ядрах, но, конечно, не в одно и то же время.

Поэтому, если вы видите, что процесс использует более одного ядра, это не означает, что программа использует несколько потоков.

Что касается параллельных потоков, производительность может не улучшиться при использовании нескольких потоков.

Стоимость синхронизации потоков может нивелировать прирост производительности при использовании нескольких потоков.

Например, в приведенном выше примере вызов `System.out` синхронизирован.

Таким образом, эффективно, только одно число может быть выведено одномоментно, хотя используются несколько потоков.

Такие операции как `sorted`, `min`, `max`, `findFirst`, `map`, `filter` не будут давать одинаковые результаты в последовательном и параллельном потоке.

Одинаковые результаты будут давать такие операции, как `allMatch`, `anyMatch`, `noneMatch`, `count`.

Там, где не важен порядок элементов.

При тестировании приложения полезно писать код, чтобы проверить одинаковость результатов при использовании параллельных потоков и понять, насколько это критично для приложения.

```

public class MathUtils {
    public static double fancySum1(double[] nums) {
        return DoubleStream.of(nums)
            .map(d -> Math.sqrt(2*d))
            .sum();
    }
    public static double fancySum2(double[] nums) {
        return DoubleStream.of(nums)
            .parallel()
            .map(d -> Math.sqrt(2*d))
            .sum();
    }

    public static double[] randomNums(int length) {
        double[] nums = new double[length];
        for(int i=0; i<length; i++) {
            nums[i] = Math.random() * 3;
        }
        return(nums);
    }

    public static void compareOutput() {
        double[] nums = MathUtils.randomNums(10_000_000);
        double result1 = MathUtils.fancySum1(nums);
        System.out.printf("Serial result = %,12f\n", result1);
        double result2 = MathUtils.fancySum2(nums);
        System.out.printf("Parallel result = %,12f\n", result2);
    }
}

Representative output
Serial result = 16,328,996.081106223000
Parallel result = 16,328,996.081106225000

```

Здесь показано сравнение результатов вычислений двух методов, один из которых выполняется в одном потоке, а другой использует параллельные потоки.

Помимо параллельных потоков, Java 8 позволяет создавать бесконечные потоки. Бесконечные потоки создаются с помощью методов `generate` и `iterate`.

```
Stream.generate(valueGenerator)

Stream.iterate(initialValue, valueTransformer)
```

При этом значения потока не вычисляются до тех пор, пока они не понадобятся.

Чтобы избежать бесконечной обработки, вы должны в конечном итоге использовать операцию ограничения размера, такую как `limit` или `findFirst`.

Дело не в том, что это «бесконечный» поток, но он является неограниченным «на лету», без фиксированного размера, где значения рассчитываются по мере необходимости.

За методом `generate` стоит следующая идея.

```
List<Employee> emps =
    Stream.generate(() -> randomEmployee())
        .limit(someRuntimeValue)
        .collect(Collectors.toList());

Supplier<Double> random = Math::random;
System.out.println("2 Random numbers:");
Stream.generate(random).limit(2).forEach(System.out::println);

public class FibonacciMaker implements Supplier<Long> {
    private long previous = 0;
    private long current = 1;
    @Override
    public Long get() {
        long next = current + previous;
        previous = current;
        current = next;
        return(previous);
    }
}
```

Вы предоставляете функцию (реализацию интерфейса `Supplier`) для `Stream.generate`.

Всякий раз, когда система требует элементы потока, она вызывает функцию для их получения.

При этом вы должны ограничить размер потока, обычно с помощью метода `limit` или `findFirst` или `findAny` для параллельных потоков.

Используя настоящий класс вместо лямбда выражения, функция может поддерживать состояние, чтобы новые значения были основаны на предыдущих значениях.

Внизу показан такой пример для генерации чисел фибоначи.

За методом `iterate` стоит следующая идея.

```

List<Integer> powersOfTwo =
Stream.iterate(1, n -> n * 2)
.limit(...)
.collect(Collectors.toList());

Stream.iterate("Big News!!", msg -> msg + "!!!!!!!!!!")
.limit(14)
.forEach(System.out::println);

Big News!!
Big News!!!!!!!!!!
Big News!!!!!!!!!!!!!!!!!!!!!!
Big News!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

```

Вы указываете начальное значение и унитарный оператор.

Начальное значение становится первым элементом потока, применение оператора к значению становится вторым элементом и т. д.

Вы должны ограничить размер потока, обычно с помощью метода `limit`.

Для параллельного потока не будет одинакового результата, потому что метод `iterate` предполагает последовательное выполнение.

Используя метод `collect` потока и методы класса `Collectors`, вы можете из потока получать различные объекты – коллекции и строки.

```

List
• anyStream.collect(Collectors.toList())

String
• stringStream.collect(Collectors.joining(delimiter)).toString()

Set
• anyStream.collect(Collectors.toSet())

Other collection
• anyStream.collect(Collectors.toCollection(CollectionType::new))

Map
• strm.collect(Collectors.partitioningBy(...)), strm.collect(Collectors.groupingBy(...))

```

Java 8 добавляет новый класс `StringJoiner`, который создает строки, разделенные разделителями, с необязательными префиксом и суффиксом.

Java 8 также добавила статический метод «`join`» к классу `String`, который использует `StringJoiner`.

```

StringJoiner joiner1 = new StringJoiner(", ");

String result1 =
joiner1.add("Java").add("Lisp").add("Ruby").toString();

String result2 = String.join(", ", "Java", "Lisp", "Ruby");

```

В потоке такие строки создаются с помощью метода `joining` класса `Collectors`.


```
List<Integer> ids = Arrays.asList(2, 4, 6, 8);
String lastNames =
ids.stream().map(EmployeeSamples::findGoogler)
.filter(e -> e != null)
.map(Employee::getLastName)
.collect(Collectors.joining(", "))
.toString();
```

Помимо потоков Java 8 вводит использование лямбда выражений непосредственно в списках и картах.

Для списков и других коллекций это методы, которые очень похожи на методы потока Stream, но они часто изменяют существующий список, в отличие от методов Stream.

```
List
- forEach
- removeIf
- replaceAll
- sort

Map
- forEach
- computeIfAbsent (compute, computeIfPresent)
- merge
- replaceAll
```

С очень большими списками новые методы могут иметь небольшие преимущества в производительности по сравнению с аналогичными методами потока Stream.

forEach метод идентичен forEach для потоков, но экономит сначала вызов метода stream.

removeIf метод работает как фильтр с отрицанием Predicate, но removeIf изменяет исходный список.

replaceAll метод работает как map, но replaceAll изменяет исходный список.

Кроме того, replaceAll функция должна отображать значения того же типа, что и в List списке.

sort метод принимает компаратор так же, как stream.sorted и Arrays.sort.

На этом слайде показаны методы списка и их аналоги для потоков.

```
employeeList.forEach(System.out::println)

employeeList.stream().forEach(System.out::println)

stringList.removeIf(s -> s.contains("q"))

stringList = stringList.stream().filter(s -> !s.contains("q")).collect(Collectors.toList())

stringList.replaceAll(String::toUpperCase)

stringList = stringList.stream().map(String::toUpperCase).collect(Collectors.toList())

employeeList.sort(Comparator.comparing(Employee::getLastName))

stringList = stringList.stream().sorted(Comparator.comparing(Employee::getLastName))
.collect(Collectors.toList())
```

Для карт Map это методы, которые значительно расширяют функциональность по сравнению с Java 7.

forEach метод для Map аналогичен методу forEach для Stream и List, кроме того, что функция принимает два аргумента – ключ и значение.

```
map.forEach((key, value) -> System.out.printf("%s,%s\n", key, value));  
  
shapeAreas.replaceAll((shape, area) -> Math.abs(area));  
  
messages.merge(key, message, (old, initial) -> old + initial);  
  
public static ReturnT memoizedMethod(arg) {  
    return (map.computeIfAbsent(arg, val -> codeForOriginalMethod(val)));  
}
```

В replaceAll методе для Map, для каждой записи карты передается ключ и значение, берется вывод и заменяется старое значение.

Этот метод аналогичен методу replaceAll для списка List, за исключением того, что функция принимает два аргумента – ключ и значение.

В методе merge для Map, если для ключа не найдено значения, сохраняется начальное значение.

В противном случае старое значение и начальное значение передается функции и производится перезапись выводом функции.

В методе computeIfAbsent для Map, если для ключа найдено значение, оно возвращается методом.

В противном случае ключ передается функции, и вывод сохраняется в карте Map.

Потоки Stream в Java 9



Tech Solutions

Объектно-ориентированное программирование на Java

Лямбда-выражения

Лекция 5

Потоки Stream в Java 9

В Java 9 программный интерфейс Stream API расширился.

Новый метод `takeWhile`, который в качестве аргумента принимает предикат, берет элементы стрима, до тех пор пока не встретится элемент, подходящий под предикат.

```
Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    .takeWhile(n -> n < 5)
    .forEach(System.out::println); // 1 2 3 4

Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    .dropWhile(n -> n < 6)
    .forEach(System.out::println); // 6 7 8 9 10
```

Метод `dropWhile`, в отличие от метода `takeWhile`, будет удалять элементы из стрима, до тех пор, пока не найдется элемент, подходящий под предикат.

Метод `iterate` позволяет создать бесконечный стрим элементов, где первый аргумент – начальное значение, а второй – это генератор нового значения стрима.

```
IntStream.iterate(0, n -> n + 1)
    .forEach(System.out::print);

IntStream.iterate(0, n -> n + 1)
    .takeWhile(n -> n < 10_000_000)
    .forEach(System.out::println);

IntStream.iterate(0, i -> i < 100_000, i -> i++)
    .forEach(System.out::println);
```

Таким образом можно создать стрим, где следующее значение будет равно значению предыдущего +1.

Если этот метод соединить с методом `takeWhile`, тогда можно получить заданную последовательность чисел.

И метод `iterate` имеет еще одну реализацию, которая заменяет классический «for цикл».

Теперь стало возможным создавать `Stream` из `null`, тем самым, уходя от проверок на `null` и уменьшая вероятность выброса исключения `NullPointerException`.

Java 8

```
users.stream()
    .flatMap(user -> {
        List<String> accounts = user.getAccounts();
        return accounts == null ? Stream.empty() :
        accounts.stream();
    })
    .collect(Collectors.toList());
```

Java 9

```
users.stream()
    .flatMap(user -> Stream.ofNullable(user.getAccounts()))
    .collect(Collectors.toList());
```

Здесь показан пример кода до Java 9, где выполняется проверка на `null`, и код в Java 9.

Видно, что в Java 9, код значительно упрощается.

В Java 9 стало возможным создавать `Stream` из `Optional`.

```
Optional<String> optional = Optional.of("Strange");
Stream<String> stream = optional.stream();
```

Если `Optional` будет пустым, то и `Stream`, соответственно, тоже.

Предположим, нам нужно найти в базе всех персонажей по именам и вывести на экран их фамилии.

```
private Map<String, String> humans = Map.of(
    "John", "Snow",
    "Aria", "Stark",
    "Daenerys", "Targaryen"
);
```

```
private Optional<String> getSurname(String name) {
    return Optional.ofNullable(humans.get(name));
}
```

```
List<String> names = List.of("John", "Aria", "Tyrion",
    "Daenerys", "Eddard");
names.stream()
    .map(this::getSurname)
    .filter(Optional::isPresent)
    .map(Optional::get)
    .forEach(System.out::println); // Snow Stark Targaryen
```

```
List<String> names = List.of("John", "Aria", "Tyrion",
    "Daenerys", "Eddard");
names.stream()
    .map(this::getSurname)
    .flatMap(Optional::stream)
    .forEach(System.out::println); // Snow Stark Targaryen
```

В Java 8 нам приходится проверять `Optional` перед тем как взять его значение.

Но в Java 9 это можно сделать проще, так как мы можем сделать `Stream` из `Optional`.

Еще один полезный метод класса `Optional` – `ifPresentOrElse`, который позволяет выполнить одно действие, если значение в `Optional` присутствует, и другое – если его нет.

```
private static Map<String, String> survivors = Map.of(
    "John", "Snow",
    "Aria", "Stark",
    "Tyrion", "Lannister",
    "Daenerys", "Targaryen"
);

List<String> names = List.of("John", "Aria", "Tyrion", "Daenerys", "Eddard");

names.stream()
    .map(this::getSurname)
    .forEach(s -> s.ifPresentOrElse(this::printAlive, this::printDead));

private static void printAlive(String surname) {
    System.out.println(surname + " is alive");
}

private static void printDead() {
    System.out.println("One more dead");
}

private static Optional<String> getSurname(String name) {
    return Optional.ofNullable(survivors.get(name));
}
```

Т.е., если значение `Optional` присутствует, выполняется одно действие, если нет – запускается другое действие.

Метод класса `Optional` – `or` пришёл в дополнение к методу `orElseGet`.

```
String username = "John";
String homeAddress = findHomeAddress(username)
    .orElseGet(() -> findWorkAddress(username));
System.out.println(homeAddress);

private static String findWorkAddress(String username) {
    return Math.random() > 0.2 ? username + "'s Work" : null;
}

private static Optional<String> findHomeAddress(String username) {
    return Math.random() > 0.5 ? Optional.of(username + "'s Home") : Optional.empty();
}

Optional<String> address = findHomeAddress(username)
    .or(() -> findWorkAddress(username));
System.out.println(address);

private static Optional<String> findWorkAddress(String username) {
    return Math.random() > 0.5 ? Optional.of(username + "'s Home") : Optional.empty();
}
```

Если метод `findHomeAddress` не «найдет» адрес пользователя, вызовется метод `findWorkAddress`, но если и он не «найдет» адрес, тогда мы получим `null`.

Новый метод `or` решает эту проблему, он принимает `Supplier`, который должен вернуть `Optional`, а не обычный объект.

Java Date/Time API



Tech Solutions

Объектно-ориентированное программирование на Java

Ввод-вывод и Date/Time API

Лекция 1

Java Date/Time API

Java предоставляет класс `Date`, доступный в пакете `java.util`, этот класс инкапсулирует текущую дату и время.

Первый конструктор инициализирует объект текущей датой и временем.

```

java.util
Class Date
java.lang.Object
  java.util.Date
All Implemented Interfaces:
  Serializable, Cloneable, Comparable<Date>
Direct Known Subclasses:
  Date, Time, Timestamp

Date( )
Date(long millisec)

boolean after(Date date)
boolean before(Date date)
Object clone( )
int compareTo(Date date)
int compareTo(Object obj)
boolean equals(Object date)
long getTime( )
int hashCode( )
void setTime(long time)
String toString( )

```

Второй конструктор принимает аргумент, равный числу миллисекунд, прошедших с полуночи, 1 января 1970 года.

Далее приведены методы класса `Date`.

Метод `after` возвращает `true`, если вызывающий объект `Date` содержит дату, которая позже указанной даты, в противном случае возвращает `false`.

Метод `before` возвращает `true`, если вызывающий объект `Date` содержит дату, которая раньше указанной даты, в противном случае возвращает `false`.

Метод `clone` дублирует вызывающий объект `Date`.

Метод `compareTo` сравнивает значение вызывающего объекта с датой. Возвращает 0, если значения равны. Возвращает отрицательное значение, если вызывающий объект раньше указанной даты. Возвращает положительное значение, если вызывающий объект позже указанной даты.

Метод `equals` возвращает `true`, если вызывающий объект `Date` содержит, то же время и дату, что и указанная дата, иначе возвращается `false`.

Метод `getTime` возвращает количество миллисекунд, прошедших с 1 января 1970 года.

Метод `hashCode` возвращает хэш-код для вызывающего объекта.

Метод `setTime` устанавливает время и дату, указанные аргументом, который представляет собой прошедшее время в миллисекундах с полуночи, 1 января 1970 года.

Метод `toString` преобразует вызывающий объект `Date` в строку и возвращает результат.

Создав объект `Date` с помощью конструктора по умолчанию, можно получить текущую дату и время.

```
Date dNow = new Date();
SimpleDateFormat ft =
    new SimpleDateFormat("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

System.out.println("Current Date: " + ft.format(dNow));

Current Date: Sun 2004.07.18 at 04:14:09 PM PDT
```

Паттерн даты и времени	Результат
"e;yyyy.MM.dd G 'at' HH:mm:ss z"e;	2012.02.07 AD at 16:55:57 EET
"e;EEE, MMM d, ''yy"e;	Tue, Feb 7, '12
"e;h:mm a"e;	4:55 PM
"e;hh 'o''clock' a, zzzz"e;	04 o'clock PM, Eastern European Time
"e;K:mm a, z"e;	4:55 PM, EET
"e;yyyyy.MMMM.dd GGG hh:mm aaa"e;	02012.February.07 AD 04:55 PM
"e;EEE, d MMM yyyy HH:mm:ss Z"e;	Tue, 7 Feb 2012 16:55:57 +0200
"e;yy@dd@mmssZ"e;	120207165557+0200

Затем, используя класс `SimpleDateFormat` и шаблон, можно вывести дату и время в удобном для вас варианте.

В Java 8 вводится новый Date-Time API для устранения следующих недостатков старого Date-Time API.

Package java.time
The main API for dates, times, instants, and durations.
See: Description

Class	Description
<code>Clock</code>	A clock providing access to the current instant, date and time using a time zone.
<code>Duration</code>	A time-based amount of time, such as '34.5 seconds'.
<code>Instant</code>	An instantaneous point on the time line.
<code>LocalDate</code>	A date without a time-zone in the ISO-8601 calendar system, such as 2007-12-03.
<code>LocalDateTime</code>	A date-time without a time-zone in the ISO-8601 calendar system, such as 2007-12-03T10:15:30.
<code>LocalTime</code>	A time without a time-zone in the ISO-8601 calendar system, such as 10:15:30.
<code>MonthDay</code>	A month-day in the ISO-8601 calendar system, such as --12-03.
<code>OffsetDateTime</code>	A date-time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 2007-12-03T10:15:30+01:00.
<code>OffsetTime</code>	A time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 10:15:30+01:00.
<code>Period</code>	A date-based amount of time in the ISO-8601 calendar system, such as '2 years, 3 months and 4 days'.
<code>Year</code>	A year in the ISO-8601 calendar system, such as 2007.
<code>YearMonth</code>	A year-month in the ISO-8601 calendar system, such as 2007-12.

Класс `java.util.Date` НЕ является потокобезопасным, поэтому разработчикам приходится иметь дело с проблемой доступа в параллельных вычислениях при использовании даты. Новый API с датой и временем неизменяемый и не имеет методов установки даты-времени.

Потокобезопасный означает, что класс потокобезопасен, если он функционирует исправно при использовании его из нескольких потоков одновременно.

В частности, он должен обеспечивать правильный доступ нескольких потоков к разделяемым данным.

Класс `java.util.Date` не является потокобезопасным, если несколько потоков попытаются получить доступ к дате и изменить ее.

Большинство классов нового Date-Time API создают объекты, которые являются неизменяемыми.

Это означает, что после создания объекта его нельзя изменить.

Чтобы изменить значение неизменяемого объекта, новый объект должен быть сконструирован как модифицированная копия оригинала.

Это также означает, что новый API-интерфейс Date-Time по определению является потокобезопасным.

Старый Date-Time API имеет плохой дизайн – дата по умолчанию начинается с 1900 года, месяц начинается с 0, а день начинается с 1, поэтому нет единообразия.

В новом API номера месяцев идут с 1.

В старом API точность представления времени составляет одну миллисекунду.

В новом API представления времени составляет одну наносекунду, что в миллион раз точнее.

В старом API обработка часовых поясов очень сложная – разработчикам приходилось писать много кода для решения вопросов часового пояса.

Новый API был разработан с учетом специфики временных зон.

Также старый API имел множество других недостатков.

Java 8 вводит новый Date-Time API в отдельном пакете `java.time`.

Существует два основных способа представления времени.

Один из способов представляет время в человеческих терминах, таких как год, месяц, день, час, минута и секунда.

Другой способ, это машинное время, которое измеряет время непрерывно по временной шкале от источника, называемой эпохой, в наносекундном разрешении.

Новый пакет Date-Time предоставляет богатый набор классов для представления даты и времени.

Некоторые классы API предназначены для представления машинного времени, а другие больше подходят для представления человеческого времени.

Первые классы, с которыми вы, вероятно, столкнетесь при использовании нового API, – это `LocalDate` и `LocalTime`.

Эти классы используются в том случае, если не требуются часовые пояса.

Существует также составной класс `LocalDateTime`, который является комбинацией `LocalDate` и `LocalTime`.

Здесь мы получаем текущее локальное время с помощью метода `now`, которое представляет собой уже форматированную строку, состоящую из даты и времени.

```
// Get the current date and time
LocalDateTime currentTime = LocalDateTime.now();
System.out.println("Current DateTime: " + currentTime);

LocalDate date1 = currentTime.toLocalDate();
System.out.println("date1: " + date1);

Month month = currentTime.getMonth();
int day = currentTime.getDayOfMonth();
int seconds = currentTime.getSecond();

System.out.println("Month: " + month + " day: " + day + " seconds: " + seconds);

Current DateTime: 2014-12-09T11:00:45.457
date1: 2014-12-09
Month: DECEMBER day: 9 seconds: 45
```

Затем мы можем извлечь из нее отдельно дату, месяц, день и время.

Также мы можем установить текущее время с нужной датой.

```
LocalDateTime date2 = currentTime.withDayOfMonth(10).withYear(2012);
System.out.println("date2: " + date2);

//12 december 2014
LocalDate date3 = LocalDate.of(2014, Month.DECEMBER, 12);
System.out.println("date3: " + date3);

//22 hour 15 minutes
LocalTime date4 = LocalTime.of(22, 15);
System.out.println("date4: " + date4);

//parse a string
LocalTime date5 = LocalTime.parse("20:15:30");
System.out.println("date5: " + date5);

date2: 2012-12-10T11:00:45.457
date3: 2014-12-12
date4: 22:15
date5: 20:15:30
```


Отдельно установить дату и время.

И конвертировать строку во время.

Локальные классы абстрагируют сложность, обусловленную часовыми поясами.

Часовой пояс представляет собой набор правил, соответствующих региону, в котором стандартное время одинаково.

Таких регионов около 40.

Часовой пояс определяется смещением от скоординированного универсального времени (UTC).

Часовые пояса могут указываться двумя типами идентификаторов: сокращенно, например, «PLT», и длиннее, например, «Азия/Карачи».

С помощью `ZonedDateTime` и `ZoneId` вы можете работать со временем, с учетом часовых поясов.

```
// Get the current date and time
ZonedDateTime date1 = ZonedDateTime.parse("2007-12-03T10:15:30+05:30[Asia/Karachi]");
System.out.println("date1: " + date1);

ZoneId id = ZoneId.of("Europe/Paris");
System.out.println("ZoneId: " + id);

ZoneId currentZone = ZoneId.systemDefault();
System.out.println("CurrentZone: " + currentZone);

// You can specify the zone id when creating a zoned date time
ZoneId id = ZoneId.of("Europe/Paris");
ZonedDateTime zoned = ZonedDateTime.of(dateTime, id);
```

Класс `ZonedDateTime`, по сути, объединяет класс `LocalDateTime` с классом `ZoneId`.

Он используется для представления полной даты с часовым поясом.

Здесь мы создаем объект времени и идентификатор зоны из строки.

Получаем текущий часовой пояс системы.

И создаем зонированное время из локального времени.

В Java 8 вводятся два специализированных класса для работы с временными отрезками — это `Period` и `Duration`.

`Period` относится к дате, основанной на времени.

```
//Get the current date
LocalDate date1 = LocalDate.now();
System.out.println("Current date: " + date1);

//add 1 month to the current date
LocalDate date2 = date1.plus(1, ChronoUnit.MONTHS);
System.out.println("Next month: " + date2);

Period period = Period.between(date2, date1);
System.out.println("Period: " + period);

Current date: 2014-12-10
Next month: 2015-01-10
Period: P-1M
```

Здесь мы берем две даты и вычисляем промежуток между ними.

`Duration` относится ко времени.

```
LocalTime time1 = LocalTime.now();
Duration twoHours = Duration.ofHours(2);

LocalTime time2 = time1.plus(twoHours);
Duration duration = Duration.between(time1, time2);

System.out.println("Duration: " + duration);

Duration: PT2H
```

Здесь мы получаем локальное время.
Создаем временной интервал в 2 часа и прибавляем его к локальному времени.
Затем обратно вычисляем разницу во времени.
Класс `OffsetDateTime` объединяет класс `LocalDateTime` с классом `ZoneOffset`.

```
LocalDateTime localDate = LocalDateTime.of(2013, Month.JULY, 20, 19, 30);
ZoneOffset offset = ZoneOffset.of("-08:00");
OffsetDateTime offsetDate = OffsetDateTime.of(localDate, offset);
```

Он используется для представления полной даты со смещением от времени Гринвича.
`Instant` класс представляет начало на временной шкале в наносекундах.

```
Instant timestamp = Instant.now();
LocalDateTime ldt = LocalDateTime.ofInstant(timestamp, ZoneId.systemDefault());
```

Этот класс полезен для создания метки времени для представления машинного времени.
Значение, возвращаемое из класса `Instant`, подсчитывает время, начинающееся с первой секунды 1 января 1970 года, также называемое эпохой.

Время до эпохи имеет отрицательное значение, и время после эпохи имеет положительное значение.

Класс `Instant` не работает с человеческими единицами времени, такими как год, месяц или день.

Если вы хотите выполнить вычисления в этих единицах, вы можете конвертировать `Instant` в другой класс, например, `LocalDateTime` или `ZonedDateTime`, путем связывания `Instant` с часовым поясом.

Класс `DateTimeFormatter` позволяет конвертировать временной объект в строковое представление с использованием указанного формата.

```

ZonedDateTime leavingZone = ...;
ZonedDateTime departure = ...;

try {
    DateTimeFormatter format = DateTimeFormatter.ofPattern("MMM d yyyy hh:mm a");
    String out = departure.format(format);
    System.out.printf("LEAVING: %s (%s)%n", out, leavingZone);
}
catch (DateTimeException exc) {
    System.out.printf("%s can't be formatted!%n", departure);
    throw exc;
}

```

Интерфейс `TemporalAdjuster` в пакете `java.time.temporal` предоставляет методы, которые принимают значение и возвращают подходящее значение.

```

LocalDate date = LocalDate.of(2000, Month.OCTOBER, 15);
DayOfWeek dotw = date.getDayOfWeek();
System.out.printf("%s is on a %s%n", date, dotw);

System.out.printf("first day of Month: %s%n",
    date.with(TemporalAdjusters.firstDayOfMonth()));

2000-10-15 is on a SUNDAY
first day of Month: 2000-10-01

```

Здесь мы берем локальную дату и пытаемся получить, исходя из нее, первый день месяца. Вы также можете создать свой собственный настраиваемый `TemporalAdjuster`.

Для этого вы создаете класс, который реализует интерфейс `TemporalAdjuster` с помощью метода `adjustInto`.

Основы ввода-вывода



Tech Solutions

Объектно-ориентированное программирование на Java

Ввод-вывод и Date/Time API

Лекция 2

Основы ввода-вывода

Теперь перейдем к рассмотрению основ ввода-вывода.

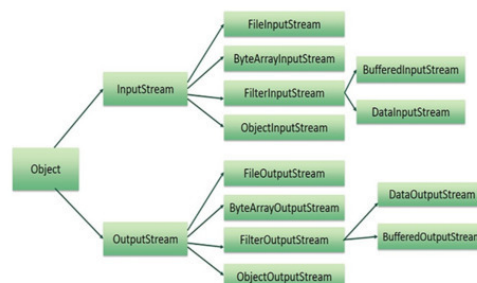
Пакет java.io содержит классы, которые используются для работы с вводом и выводом данных в Java.

Class Summary	
Class	Description
BufferedInputStream	A BufferedInputStream adds functionality to another input stream namely, the ability to buffer the input and to support the seek and reset methods.
BufferedOutputStream	The class implements a buffered output stream.
BufferedReader	Reads text from a character input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
BufferedWriter	Writes text to a character output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.
ByteArrayInputStream	A ByteArrayInputStream contains an internal buffer that contains bytes that may be read from the stream.
ByteArrayOutputStream	This class implements an output stream in which the data is written into a byte array.
CharArrayReader	This class implements a character buffer that can be used as a character input stream.
CharArrayWriter	This class implements a character buffer that can be used as a writer.
Console	Methods to access the character-based console device, if any, associated with the current Java virtual machine.
DataInputStream	A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way.
DataOutputStream	A data output stream lets an application write primitive Java data types to an output stream in a portable way.
File	An abstract representation of file and directory pathnames.
FileDescriptor	Instances of the file descriptor class serve as an opaque handle to the underlying machine-specific structure representing an open file, an open socket, or another source or sink of bytes.
FileInputStream	A FileInputStream obtains input bytes from a file in a file system.
FileOutputStream	A file output stream is an output stream for writing data to a File or to a FileDescriptor.
FileReader	This class represents access to a file or directory.
FileWriter	Convenience class for reading character files.
FilterInputStream	Convenience class for writing character files.
FilterOutputStream	A FilterOutputStream contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality.
FilterReader	This class is the superclass of all classes that filter input streams.
FilterWriter	Abstract class for reading filtered character streams.

Язык Java предоставляет простую модель ввода и вывода.

Все операции ввода-вывода выполняются путем записи и чтения из потоков данных.

Данные могут существовать в файле или массиве, быть отправленными из другого потока или даже из порта другого компьютера.



Гибкость этой модели делает ее мощной абстракцией для любых требуемых входных и выходных данных.

Все эти потоки могут представлять множество различных источников ввода и целей вывода, включая хранимые файлы, устройства, другие программы и массивы в оперативной памяти.

Потоки в пакете `java.io` поддерживают множество типов данных, таких как байты, примитивы и объекты.

Итак, поток представляет собой последовательность данных.

Программа использует входной поток для чтения данных из источника, по одному элементу за раз.

И программа использует выходной поток для записи данных в цель вывода, по одному элементу за раз.

В Java потоки представлены объектами.

Описывающие эти объекты классы и интерфейсы как раз и составляют основную часть пакета `java.io`.

Все классы разделены на две части – одни осуществляют ввод данных, другие вывод.

Java представляет два основных типа потоков – байт-ориентированные потоки ввода-вывода (упорядоченные последовательности байтов) и символьные потоки ввода-вывода (упорядоченные последовательности символов).

Файлы, созданные с использованием байт-ориентированных потоков, называются двоичными или бинарными файлами.

Файлы, созданные с использованием символьно ориентированных потоков, называются текстовыми файлами.

Минимальной «порцией» информации является, как известно, бит, принимающий значение 0 или 1.

Традиционно используется более крупная единица измерения – это байт, объединяющая 8 бит.

Таким образом, значение, представленное 1 байтом, находится в диапазоне от 0 до 255, или, если использовать знак, от -128 до +127.

Базовые классы `InputStream` и `OutputStream` позволяют считывать и записывать информацию именно в виде набора байт.

Чтобы их было удобно применять в различных задачах, пакет `java.io` обеспечивает преобразование любых данных в набор байт.

Классы `InputStream` и `OutputStream` являются абстрактными.

Их задача – определить общий интерфейс для классов, которые получают данные из различных источников.

Таковыми источниками могут быть, например, массив байт, файл, строка и т. д.

`InputStream` – это базовый класс для потоков ввода, т.е. чтения.

Соответственно, он описывает базовые методы для работы с байтовыми потоками данных.

```

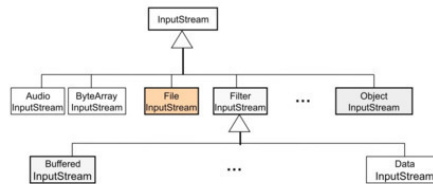
java.io
Class InputStream

java.lang.Object
  java.io.InputStream
All Implemented Interfaces:
  Closeable, AutoCloseable
Direct Known Subclasses:
  AudioInputStream, ByteArrayInputStream, FileInputStream, FilterInputStream, InputStream, ObjectInputStream, PipedInputStream, SequenceInputStream, StringBufferInputStream

public abstract class InputStream
  extends Object
  implements Closeable
This abstract class is the superclass of all classes representing an input stream of bytes.
Applications that need to define a subclass of InputStream must always provide a method that returns the next byte of input.

```

Эти методы необходимы всем классам, наследующимся от `InputStream`.



Простейшая операция класса `InputStream` представлена методом `read()` (без аргументов).

```

read
public abstract int read()
    throws IOException;

Reads the next byte of data from the input stream. The value byte is returned as an int in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value -1 is returned. This method blocks until input data is available, the end of the stream is detected, or an exception is thrown.
A subclass must provide an implementation of this method.

Returns:
    the next byte of data, or -1 if the end of the stream is reached.

Throws:
    IOException - if an I/O error occurs.
  
```

Этот метод является абстрактным, и, соответственно, должен быть определен в классах-наследниках.

Этот метод предназначен для считывания ровно одного байта из потока, однако возвращает при этом значение типа `int`.

В случае если считывание произошло успешно, то возвращаемое значение лежит в диапазоне от 0 до 255 и представляет собой полученный байт (значение `int` содержит 4 байта и получается простым дополнением нулями в двоичном представлении оставшихся байтов).

Обратите внимание, что полученный таким образом байт не обладает знаком и не находится в диапазоне от -128 до +127 как примитивный тип `byte` в Java.

В случае если достигнут конец потока, то есть в нем больше нет информации для чтения, тогда возвращаемое значение равно -1.

Если же считать из потока данные не удастся из-за каких-то ошибок или сбоев, выбрасывается исключение `IOException`.

Этот класс наследуется от класса `Exception`, и его всегда необходимо явно обрабатывать блоком `try-catch`.

Дело в том, что каналы передачи информации, будь то Интернет или, например, жесткий диск, могут давать сбой независимо от того, насколько хорошо написана программа.

А это означает, что нужно быть готовым к этим сбоям, чтобы пользователь не потерял нужные данные.

На практике обычно приходится считывать не один, а сразу несколько байт – то есть массив байт.

```

read
public int read(byte[] b)
    throws IOException

Reads some number of bytes from the input stream and stores them into the buffer array b. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.

If the length of b is zero, then no bytes are read and 0 is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at the end of the file, the value -1 is returned; otherwise, at least one byte is read and stored into b.

The first byte read is stored into element b[0], the next one into b[1], and so on. The number of bytes read is, at most, equal to the length of b. Let n be the number of bytes actually read; these bytes will be stored in elements b[0] through b[n-1], leaving elements b[n] through b[b.length-1] unaffected.

The read() method for class InputStream has the same effect as
    read(b, 0, b.length)

Parameters:
    b - the buffer into which the data is read.

Returns:
    the total number of bytes read into the buffer, or -1 if there is no more data because the end of the stream has been reached.

Throws:
    IOException - if the first byte cannot be read for any reason other than the end of the file, if the input stream has been closed, or if some other I/O error occurs.
    NullPointerException - if b is null.

See Also:
    read(byte[], int, int)

```

Для этого используется метод `read()`, где в качестве параметра передается массив `byte []`.

При выполнении этого метода в цикле производится вызов абстрактного метода `read()` (определенного без параметров), и результатами заполняется переданный массив.

Количество байт, считанное таким образом, равно длине переданного массива.

Но при этом может так получиться, что в потоке данные закончатся еще до того, как будет заполнен весь массив.

То есть, возможна ситуация, когда в потоке данных байт содержится меньше чем длина массива.

Поэтому метод возвращает значение `int`, указывающее, сколько байт было реально считано.

Понятно, что это значение может быть от 0 до величины длины переданного массива.

Если же мы изначально хотим заполнить не весь массив, а только его часть, то для этих целей используется метод `read()`, которому кроме массива `byte []`, передаются еще два `int` значения.

Первое значение – это позиция в массиве, с которой следует начать заполнение, второе значение – количество байт, которое нужно считать.

При вызове методов `read()`, возможно возникновение такой ситуации, когда запрашиваемые данные еще не готовы к считыванию.

Например, если мы считываем данные, поступающие из сети, и они еще просто не пришли.

В таком случае нельзя сказать, что данных больше нет, но и считать тоже нечего – выполнение останавливается на вызове метода `read()` и получается «зависание».

Что бы узнать, сколько байт в потоке готово к считыванию – используется метод `available()`.

Этот метод возвращает значение типа `int`, которое показывает, сколько байт в потоке готово к считыванию.

```

available
public int available()
    throws IOException

Returns an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream. The next invocation might be the same thread or another thread. A single read or skip of this many bytes will not block, but may read or skip fewer bytes.

Note that while some implementations of InputStream will return the total number of bytes in the stream, many will not. It is never correct to use the return value of this method to allocate a buffer intended to hold all data in this stream.

A subclass implementation of this method may choose to throw an IOException if this input stream has been closed by invoking the close() method.

The available method for class InputStream always returns 0.

This method should be overridden by subclasses.

Returns:
    an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking or 0 when it reaches the end of the input stream.

Throws:
    IOException - if an I/O error occurs.

```

При этом не нужно путать это количество байт, готовых к считыванию, с тем количеством байт, которые вообще можно будет считать из этого потока.

Метод `available ()` возвращает число – количество байт, именно на данный момент, готовых к считыванию.

Когда работа с входным потоком данных окончена, его следует закрыть.

Для этого вызывается метод `close ()`.

Этим вызовом освобождаются все системные ресурсы, связанные с потоком.

```
close
public void close()
    throws IOException
Closes this input stream and releases any system resources associated with the stream.
The close method of InputStream does nothing.
Specified by:
    close in interface Closeable
Specified by:
    close in interface AutoCloseable
Throws:
    IOException - if an I/O error occurs.
```

Точно так же, как `InputStream` – это базовый класс для потоков ввода, класс `OutputStream` – это базовый класс для потоков вывода.

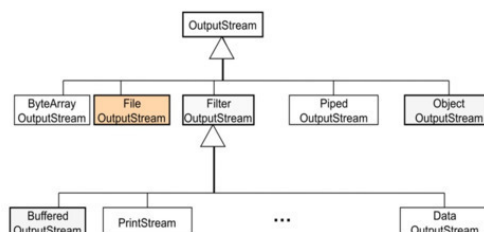
Это тоже абстрактный класс, который имеет свои конкретные реализации.

```
java.io
Class OutputStream

java.lang.Object
  java.io.OutputStream
    All Implemented Interfaces:
      Closeable, Flushable, AutoCloseable
    Direct Known Subclasses:
      ByteArrayOutputStream, FileOutputStream, FilterOutputStream, ObjectOutputStream, PipedOutputStream

public abstract class OutputStream
    extends Object
    implements Closeable, Flushable
This abstract class is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.
Applications that need to define a subclass of OutputStream must always provide at least a method that writes one byte of output.
```

Здесь показана иерархия класса `OutputStream`.



В классе `OutputStream`, аналогичным образом, определяются три метода `write ()`.

Один метод принимает в качестве параметра `int`, второй `byte []`, и третий `byte []`, плюс два `int`-числа.

Method Summary	
Methods	
Modifier and Type	Method and Description
<code>void</code>	<code>close()</code> Closes this output stream and releases any system resources associated with this stream.
<code>void</code>	<code>flush()</code> Flushes this output stream and forces any buffered output bytes to be written out.
<code>void</code>	<code>write(byte[] b)</code> Writes <code>b.length</code> bytes from the specified byte array to this output stream.
<code>void</code>	<code>write(byte[] b, int off, int len)</code> Writes <code>len</code> bytes from the specified byte array starting at offset <code>off</code> to this output stream.
<code>abstract void</code>	<code>write(int b)</code> Writes the specified byte to this output stream.

Все эти методы возвращают `void`, то есть ничего не возвращают.

Метод `write (int)` является абстрактным, и должен быть реализован в классах – наследниках.

Этот метод принимает в качестве параметра `int`, но реально записывает в поток только `byte` – младшие 8 бит в двоичном представлении.

Остальные 24 бита будут проигнорированы.

В случае возникновения ошибки этот метод бросает `IOException`, как и большинство методов, связанных с вводом-выводом.

Для записи в поток сразу некоторого количества байт, методу `write ()` передается массив байт.

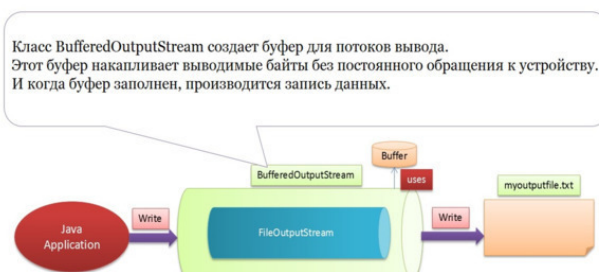
Или, если мы хотим записать только часть массива – то передаем массив `byte []`, и два `int`-числа – отступ и количество байт для записи.

Если при этом указать неверные параметры – например отрицательный отступ, отрицательное количество байт для записи, или если сумма отступ+длина будет больше длины массива – во всех этих случаях бросается исключение `IndexOutOfBoundsException`.

Реализация потока может быть таковой, что данные записываются не сразу, а хранятся некоторое время в памяти.

Это класс `BufferedOutputStream`.

Например, мы хотим записать в файл какие-то данные, которые мы получаем порциями по 10 байт, и так 200 раз подряд.



В таком случае вместо 200 обращений к файлу удобнее будет скопить все эти данные в памяти, а потом одним заходом записать все 2000 байт.

То есть класс выходного потока может использовать некоторый свой внутренний механизм для буферизации (временного хранения перед отправкой) данных.

Что бы убедиться, что данные записаны в поток, а не хранятся в буфере, вызывается метод `flush ()`, определенный в `OutputStream`.

В этом классе его реализация пустая, но если какой-либо из наследников использует буферизацию данных, то этот метод должен быть в нем переопределен.

Когда работа с потоком закончена, его следует закрыть.

Для этого вызывается метод `close ()`.

Этот метод сначала освобождает буфер, после этого поток закрывается, и освобождаются все системные ресурсы с ним связанные.

Закрытый поток не может выполнять операции вывода и не может быть открыт заново.

Для этого должен быть создан новый поток.

Для записи и считывания просто байтов есть две реализации рассмотренных абстрактных классов – это классы `ByteArrayInputStream` и `ByteArrayOutputStream`.

Класс `ByteArrayInputStream` представляет поток, считывающий данные из массива байт.

```
byte[] bytes = {1,-1,0};
ByteArrayInputStream in = new ByteArrayInputStream(bytes);

int readedInt = in.read(); // readedInt=1
System.out.println("first element read is: " + readedInt);

readedInt = in.read(); // readedInt=255. Однако (byte)readedInt даст значение -1
System.out.println("second element read is: " + readedInt);

readedInt = in.read(); // readedInt=0
System.out.println("third element read is: " + readedInt);

first element read is: 1
second element read is: 255
third element read is: 0
```

Этот класс имеет конструктор, которому в качестве параметра передается массив `byte []`.

Соответственно, при вызове методов `read ()`, возвращаемые данные будут браться именно из этого массива.

Аналогично, для записи байт в массив, используется класс `ByteArrayOutputStream`.

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
out.write(10);
out.write(11);
byte[] bytes = out.toByteArray();
```

Этот класс использует внутри себя объект `byte []`, куда записывает данные, передаваемые при вызове методов `write ()`.

Что бы получить записанные в массив данные, вызывается метод `toByteArray ()`.

Классы `ByteArrayInputStream` и `ByteArrayOutputStream` используются тогда, когда нужно проверить, что именно записывается в выходной поток.

Например, при отладке и тестировании сложных процессов записи и чтения из потоков.

Использование этих классов удобно тем, что можно сразу посмотреть результат, и не нужно создавать ни файл, ни сетевое соединение, ни что-либо еще.

Для записи байт в файл используется класс `FileOutputStream`.

```
byte[] bytesToWrite = {1, 2, 3};
byte[] bytesReaded = new byte[10];
String fileName = "d:\\test.txt";
try {
    // Создать выходной поток
    FileOutputStream outFile = new FileOutputStream(fileName);
    System.out.println("Файл открыт для записи");
    // Записать массив
    outFile.write(bytesToWrite);
    System.out.println("Записано: " + bytesToWrite.length + " байт");
    // По окончании использования должен быть закрыт
    outFile.close();
    System.out.println("Выходной поток закрыт");
} catch (FileNotFoundException e) {
    System.out.println("Невозможно произвести запись в файл: " + fileName);
} catch (IOException e) {
    System.out.println("Ошибка ввода/вывода: " + e.toString());
}
```

При создании объектов этого класса, то есть при вызовах его конструкторов, кроме указания файла, так же можно указать, будут ли данные дописываться в конец файла либо файл будет перезаписан.

При этом, если указанный файл не существует, то сразу после создания `FileOutputStream` он будет создан.

После, при вызовах методов `write ()`, передаваемые значения будут записываться в этот файл.

По окончании работы необходимо вызвать метод `close ()`, что бы сообщить системе, что работа по записи файла закончена.

Для этого в блоке `try-catch` можно использовать выражение `finally`.

```
byte[] bytesToWrite = {1, 2, 3};
byte[] bytesReaded = new byte[10];
String fileName = "d:\\test.txt";
try {
    // Создать выходной поток
    FileOutputStream outFile = new FileOutputStream(fileName);
    System.out.println("Файл открыт для записи");
    // Записать массив
    outFile.write(bytesToWrite);
    System.out.println("Записано: " + bytesToWrite.length + " байт");
    System.out.println("Выходной поток закрыт");
} catch (FileNotFoundException e) {
    System.out.println("Невозможно произвести запись в файл: " + fileName);
} catch (IOException e) {
    System.out.println("Ошибка ввода/вывода: " + e.toString());
} finally {
    // По окончании использования должен быть закрыт
    if (outFile != null) {
        outFile.close();
    }
}
```

В Java 7 и более поздних версиях вы можете объявлять переменные, которые реализуют интерфейс `AutoCloseable` в скобках после `try`.

Это так называемое выражение `try-with-resources`.

try-with-resources

<pre>BufferedReader reader; try { reader = ...; ... } catch (SomeExceptionType e) { ... } finally { reader.close(); }</pre>	<pre>try(BufferedReader reader = ...) { ... } catch (SomeExceptionType e) { ... }</pre>
---	---

Здесь область видимости переменной – область действия блока try / catch.

Метод закрытия каждой переменной вызывается в конце, независимо от того, бросается или нет исключение.

И здесь можно объявлять несколько переменных, разделенных точкой с запятой.

Таким образом выражение в скобках после try заменяет блок finally.

И так как блок try требует либо блока finally либо блока catch здесь можно использовать просто блок try со скобками без блока catch.

В Java 9, если у нас есть несколько ресурсов, и мы хотим, чтобы блок try-with-resources закрывал их все, теперь, это возможно даже без введения новой переменной в скобках после try, если наш ресурс уже имеет ключевое слово final или является эффективно финальным.

Java 7

```
try(FileInputStream input = new FileInputStream("somefile")) {
    ...
}
```

Java 9

```
final Resource resource1 = new Resource("resource1");
Resource resource2 = new Resource("resource2");

try(resource1; resource2 ...) {
    ...
}
```

Класс `FileInputStream` используется для чтения данных из файла.

```
try {
    // Создать входной поток
    FileInputStream inFile = new FileInputStream(fileName);
    System.out.println("Файл открыт для чтения");

    // Узнать, сколько байт готово к считыванию
    int bytesAvailable = inFile.available();
    System.out.println("Готово к считыванию: " + bytesAvailable + " байт");

    // Считать в массив
    int count = inFile.read(bytesReaded, 0, bytesAvailable);
    System.out.println("Считано: " + count + " байт");

    inFile.close();
    System.out.println("Входной поток закрыт");
} catch (IOException e) {
    System.out.println("Ошибка ввода/вывода: " + e.toString());
} catch (FileNotFoundException e) {
    System.out.println("Невозможно произвести чтение из файла: " + fileName);
}
```

Конструктор этого класса в качестве параметра принимает название файла, из которого будет производиться считывание.

При указании строки имени файла нужно учитывать, что именно эта строка и будет передана системе, поэтому формат имени файла и пути к нему может различаться на различных платформах.

Если при вызове этого конструктора передать строку, указывающую на директорию либо на не существующий файл, то будет брошено исключение `FileNotFoundException`.

Понятно, что если объект успешно создан, то при вызове его методов `read()`, возвращаемые значения будут считываться из указанного файла.

При работе с `FileInputStream` метод `available()` практически наверняка вернет длину файла, то есть число байт, сколько вообще из него можно считать.

Но не стоит рассчитывать на это при написании программ, которые должны устойчиво работать на различных платформах – метод `available()` возвращает число, сколько байт может быть на данный момент считано без блокирования.

И то, что в подавляющем большинстве случаев это число и будет длиной файла, является всего лишь частным случаем работы на некоторых платформах.

В данном примере для наглядности закрытие потоков производится сразу же после окончания их использования в основном блоке.

Однако считается хорошей практикой закрывать потоки в `finally` блоке.

Такой подход гарантирует, что поток будет закрыт, и будут освобождены все системные ресурсы с ним связанные.

Объекты классов `PipedInputStream` и `PipedOutputStream` всегда используются в паре – к одному объекту `PipedInputStream` привязывается точно один объект `PipedOutputStream`.

```
try {
    int countRead = 0;
    byte[] toRead = new byte[100];
    PipedInputStream pipeIn = new PipedInputStream();
    PipedOutputStream pipeOut = new PipedOutputStream(pipeIn);
    // Считать в массив, пока он полностью не будет заполнен
    while(countRead < toRead.length){
        // Записать в поток некоторое количество байт
        for(int i=0; i<(Math.random()*10); i++){
            pipeOut.write((byte)(Math.random()*127));
        }
        // Считать из потока доступные данные,
        // добавить их к уже считанным.
        int willRead = pipeIn.available();
        if(willRead < countRead + toRead.length)
            // Нужно считать только до предела массива
            willRead = toRead.length - countRead;
        countRead += pipeIn.read(toRead, countRead, willRead);
    }
} catch (IOException e) {
    pr("Impossible IOException occur: ");
    e.printStackTrace();
}
```

Эти классы могут быть полезны, если необходимо данные записать, и считать в пределах одного блока кода.

При этом создается по объекту `PipedInputStream` и `PipedOutputStream`, после чего они могут быть соединены между собой.

Один объект `PipedOutputStream` может быть соединен с ровно одним объектом `PipedInputStream` и наоборот.

Соединенный- означает, что если в объект `PipedOutputStream` записываются данные, то они могут быть считаны именно в соединенном объекте `PipedInputStream`.

Такое соединение можно обеспечить либо вызовом метода `connect()` с передачей соответствующего объекта `PipedStream`, либо передать этот объект еще при вызове конструктора.

Польза от использования этих классов в основном проявляется при разработке многопоточного приложения.

Например, если программа выполняется в нескольких потоках выполнения, организовать передачу данных между потоками удобно с помощью объектов классов `PipedStream`.

Для этого достаточно создать связанные объекты классов `PipedInputStream` и `PipedOutputStream`, после чего передать их в соответствующие потоки.

Интересный класс это `SequenceInputStream`.

```
FileInputStream inFile1 = null;
FileInputStream inFile2 = null;
SequenceInputStream sequenceStream = null;
FileOutputStream outFile = null;
try {
    inFile1 = new FileInputStream("file1.txt");
    inFile2 = new FileInputStream("file2.txt");
    sequenceStream = new SequenceInputStream(inFile1, inFile2);
    outFile = new FileOutputStream("file3.txt");
    int readedByte = sequenceStream.read();
    while(readedByte != -1){
        outFile.write(readedByte);
        readedByte = sequenceStream.read();
    }
} catch (IOException e) {
    System.out.println("IOException: " + e.toString());
} finally {
    try{sequenceStream.close();}catch(IOException e){};
    try{outFile.close();}catch(IOException e){};
}
```

Он считывает данные из других двух и более входных потоков.

При этом можно указать два потока или их список.

Данные будут считываться последовательно – сначала все данные из первого потока в списке, потом из второго, и так далее.

Конец потока `SequenceInputStream` будет достигнут только тогда, когда будет достигнут конец потока, последнего в списке.

При создании объекта этого класса в конструктор в качестве параметров передаются объекты `InputStream` – либо их экземпляры, либо перечисление из них.

Когда вызывается метод `read()`, `SequenceInputStream` пытается считать байт из текущего входного потока.

Если в нем больше нет данных (считанное из него значение равно `-1`), у этого входного потока вызывается метод `close()`, и следующий входной поток становится текущим.

Так до тех пор, пока последний входной поток не станет текущим, и из него не будут считаны все данные.

Тогда, если при считывании обнаруживается, что в текущем входном потоке нет больше данных, и больше нет входных потоков `SequenceInputStream` возвращает `-1`, то есть объявляет, что данных больше нет.

`SequenceInputStream` автоматически вызывает вызов метода `close()` у входных потоков, у которых достигнут конец.

Вызов метода `close()` класса `SequenceInputStream` закрывает этот поток, предварительно закрыв все содержащиеся в нем входные потоки.

Классы `FilterInputStream` и `FilterOutputStream` обеспечивают реализацию паттерна проектирования `Decorator`.

```
java.io
Class FilterInputStream

java.lang.Object
  java.io.InputStream
    java.io.FilterInputStream

All Implemented Interfaces:
  Closeable, Readable

Direct Known Subclasses:
  BufferedInputStream, CheckedInputStream, CipherInputStream, DataInputStream, DeflaterInputStream, DigestInputStream, InflaterInputStream, LineNumberInputStream, ProgressMonitorInputStream,
  PushbackInputStream

public class FilterInputStream
    extends InputStream

A FilterInputStream contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality. The class FilterInputStream itself
simply overrides all methods of InputStream with versions that pass all requests to the contained input stream. Subclasses of FilterInputStream may further override some of these methods and may also provide
additional methods and functionality.

java.io
Class FilterOutputStream

java.lang.Object
  java.io.OutputStream
    java.io.FilterOutputStream

All Implemented Interfaces:
  Closeable, Flushable, Writable

Direct Known Subclasses:
  BufferedOutputStream, CheckedOutputStream, CipherOutputStream, DataOutputStream, DeflaterOutputStream, DigestOutputStream, InflaterOutputStream, PrintStream

public class FilterOutputStream
    extends OutputStream

This class is the superclass of all classes that filter output streams. These streams all sit on top of an already existing output stream (the underlying output stream) which it uses as its basic sink of data, but possibly
transforming the data along the way or providing additional functionality.

The class FilterOutputStream itself simply overrides all methods of OutputStream with versions that pass all requests to the underlying output stream. Subclasses of FilterOutputStream may further override some of
these methods as well as provide additional methods and functionality.
```

Это когда вы хотите в наследниках классов `InputStream` и `OutputStream` не просто реализовывать методы чтения и записи элементов потока, но и манипулировать с самим потоком, добавляя к нему дополнительные свойства, например, обеспечить буферизацию базового входного потока для ускорения чтения данных, или вычислять криптографический хеш данных.

Для этого в классах `FilterInputStream` и `FilterOutputStream` вводится поле, содержащее объект потока, который инициализируется в конструкторе класса.

На практике, при считывании с внешних устройств, ввод данных почти всегда необходимо буферизировать.

public class BufferedInputStream
extends FilterInputStream

A BufferedInputStream adds functionality to another input stream namely, the ability to buffer the input and to support the mark and reset methods. When the BufferedInputStream is created, an internal buffer array is created. As bytes from the stream are read or skipped, the internal buffer is refilled as necessary from the contained input stream. Many bytes at a time. The mark operation remembers a point in the input stream and the reset operation causes all the bytes read since the most recent mark operation to be reread before new bytes are taken from the contained input stream.

Since:
JDK1.0

Field Summary

Modifier and Type	Field and Description
protected byte[]	buf The internal buffer array where the data is stored.
protected int	count The index one greater than the index of the last valid byte in the buffer.
protected int	markLimit The maximum read ahead allowed after a call to the mark method before subsequent calls to the reset method fail.
protected int	markPos The value of the pos field at the time the last mark method was called.
protected int	pos The current position in the buffer.

Для буферизации данных служат классы `BufferedReader` и `BufferedOutputStream`.

Класс `BufferedReader` служит оберткой вокруг входного потока, содержа в себе массив байт, который служит буфером для считываемых данных.

То есть, когда байты из потока считываются (вызов метода `read()`) либо пропускаются (метод `skip()`), сначала перезаписывается этот буферный массив, при этом считываются сразу много байт за раз.

Так же класс `BufferedReader` добавляет поддержку методов `mark()` и `reset()`.

Эти методы определены еще в классе `InputStream`, но их реализация по умолчанию просто бросает исключение `IOException`.

Метод `mark()` запоминает точку во входном потоке и метод `reset()` приводит к тому, что все байты, считанные после наиболее позднего вызова метода `mark()`, будут считаны заново, прежде чем новые байты будут считываться из содержащегося входного потока.

В этом примере мы создаем входной поток из файла и оборачиваем его `BufferedReader`.

```
InputStream inStream = null;
BufferedReader bis = null;
try {
    // open input stream test.txt for reading purpose.
    inStream = new FileInputStream("c:/test.txt");
    // input stream is converted to buffered input stream
    bis = new BufferedReader(inStream);
    // read until a single byte is available
    while(bis.available() > 0) {
        // read the byte and convert the integer to character
        char c = (char) bis.read();
        // print the characters
        System.out.println("Char: " + c);
    }
} catch (Exception e) {
    // if any I/O error occurs
    e.printStackTrace();
} finally {
    // releases any system resources associated with the stream
    if (inStream != null)
        inStream.close();
    if (bis != null)
        bis.close();
}
```

Затем считываем из потока, используя уже метод реализации класса `BufferedReader`, который декорирует входной поток буферизацией, тем самым повышая скорость считывания.

При использовании объекта класса `BufferedOutputStream`, запись производится без необходимости обращения к устройству ввода/вывода при записи каждого байта.

```
try {
    String fileName = "d:\\file1";
    InputStream inputStream = null;
    OutputStream outputStream = null;
    //Записать в файл некоторое количество байт
    long timeStart = System.currentTimeMillis();
    outputStream = new FileOutputStream(fileName);
    outputStream = new BufferedOutputStream(outputStream);
    for(int i=1000000; i>0; i--){
        outputStream.write(i);
    }
    long time = System.currentTimeMillis() - timeStart;
    System.out.println("Writing durates: " + time + " millisec");
    outputStream.close();
} catch (IOException e) {
    pr("IOException: " + e.toString());
    e.printStackTrace();
}
```

Сначала данные записываются во внутренний буфер.

Непосредственное обращение к устройству вывода и, соответственно, запись в него произойдет, когда буфер будет полностью заполнен.

Освобождение буфера с записью байт на устройство вывода можно обеспечить вызовом метода `flush ()`.

Так же буфер будет освобожден непосредственно перед закрытием потока (вызов метода `close ()`).

При вызове этого метода также будет закрыт и поток, над которым буфер настроен.

В этом примере мы записываем в файл байты, при этом измеряя скорость записи.

Классы `BufferedInputStream` и `BufferedOutputStream` добавляют только внутреннюю логику по обработке запросов, они не добавляют никаких дополнительных методов.

Сериализация

До сих пор речь шла только о считывании и записи в поток данных в виде byte.



Объектно-ориентированное программирование на Java

Ввод-вывод и Date/Time API

Лекция 3

Сериализация

```

java.io
Class DataInputStream

java.lang.Object
  java.io.InputStream
    java.io.FilterInputStream
      java.io.DataInputStream

All Implemented Interfaces:
  Closeable, DataInput, AutoCloseable

public class DataInputStream
  extends FilterInputStream
  implements DataInput

A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream.
DataInputStream is not necessarily safe for multithreaded access. Thread safety is optional and is the responsibility of users of methods in this class.

java.io
Class DataOutputStream

java.lang.Object
  java.io.OutputStream
    java.io.FilterOutputStream
      java.io.DataOutputStream

All Implemented Interfaces:
  Closeable, DataOutput, Flushable, AutoCloseable

public class DataOutputStream
  extends FilterOutputStream
  implements DataOutput

A data output stream lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in.

```

Для работы с другими примитивными типами данных java, определены интерфейсы `DataInput` и `DataOutput`, и их реализации – классы-фильтры `DataInputStream` и `DataOutputStream`.

Интерфейсы `DataInput` и `DataOutput` определяют, а классы `DataInputStream` и `DataOutputStream`, соответственно реализуют, методы считывания и записи всех примитивных типов данных.

При этом происходит конвертация этих данных в byte и обратно.

В поток будут записаны байты, а ответственность за восстановление данных лежит целиком и полностью на разработчике – нужно считывать данные в виде тех же типов, в той же последовательности, как и производилась запись.

То есть, можно, конечно, записать несколько раз `int` или `long`, а потом считывать их как `short` или что-нибудь еще – считывание произойдет корректно и никаких предупреждений о возникшей ошибке не возникнет, но результат будет в виде значений, которые никогда не записывались.

В этом примере создается байтовый поток, который обертывается `DataOutputStream`.

```

try {
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    DataOutputStream outData = new DataOutputStream(out);
    outData.writeByte(128); // it would write -128, as casted to byte
    outData.writeInt(128);
    outData.writeLong(128);
    outData.writeDouble(128);
    outData.close();
    byte[] bytes = out.toByteArray();
    InputStream in = new ByteArrayInputStream(bytes);
    DataInputStream inData = new DataInputStream(in);
    System.out.println("Read data in order it was written:");
    System.out.println("readByte: " + inData.readByte());
    System.out.println("readInt: " + inData.readInt());
    System.out.println("readLong: " + inData.readLong());
    System.out.println("readDouble: " + inData.readDouble());
    inData.close();
} catch (Exception e) {
    System.out.println("Impossible IOException occurs: " + e.toString());
    e.printStackTrace();
}

```

Далее в поток записываются байт, int, long и double.

Затем это все считывается в том же порядке, что и записывалось.

Интерфейсы DataInput и DataOutput представляют возможность записи/считывания данных примитивных типов Java.

Для аналогичной работы с объектами определены унаследованные от них интерфейсы ObjectInput и ObjectOutput соответственно.

В пакете java.io имеются реализации этих интерфейсов – классы ObjectInputStream и ObjectOutputStream.

Процесс превращения в байты и обратно для объектов сложнее чем для примитивных типов – записываться могут объекты разных классов, они могут иметь ссылки на другие объекты и т. д.

```

java.io
Class ObjectInputStream

java.lang.Object
java.io.ObjectStream
java.io.ObjectInputStream

All Implemented Interfaces:
    Closeable, DataInput, ObjectInput, ObjectStreamConstants, AutoCloseable

public class ObjectInputStream
    extends ObjectStream
    implements ObjectInput, ObjectStreamConstants

ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream.
Objects are deserialized in the order they were serialized. Other uses include passing objects between threads using a socket stream or for marshaling and unmarshaling arguments and parameters in a remote communication system.
ObjectInputStream ensures that the types of all objects in the graph created from the stream match the classes present in the Java Virtual Machine. Classes are loaded as required using the standard mechanisms.
Only objects that support the java.io.Serializable or java.io.Externalizable interface can be read from streams.
The method readObject() is used to read an object from the stream. Java's safe casting should be used to get the desired type. In Java, strings and arrays are objects and are treated as objects during serialization. When
read they need to be cast to the expected type.

java.io
Class ObjectOutputStream

java.lang.Object
java.io.ObjectStream
java.io.ObjectOutputStream

All Implemented Interfaces:
    Closeable, DataOutput, Flushable, ObjectOutput, ObjectStreamConstants, AutoCloseable

public class ObjectOutputStream
    extends ObjectStream
    implements ObjectOutput, ObjectStreamConstants

An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream. The objects can be read (reconstructed) using an ObjectInputStream. Persistent storage of objects can be accomplished
by using it for the stream. If the stream is a random access stream, the objects can be reconstructed on another host or in another process.
Only objects that support the java.io.Serializable interface can be written to streams. The class of each serializable object is encoded including the class name and signature of the class. The values of the object's fields
and arrays, and the values of any other objects referenced from the object objects.
The method writeObject() is used to write an object to the stream. Any object, including Strings and arrays, is written with writeObject(). Multiple objects or primitives can be written to the stream. The objects must be read
back from the corresponding ObjectInputStream with the same type and in the same order as they were written.
Primitive data types can also be written to the stream using the appropriate methods from DataOutput. Strings can also be written using the writeUTF method.

```

Процесс превращения объекта в набор байт носит название сериализация (Serialization) и, соответственно, для обратного процесса – конвертацию набора байт в объект – десериализация.

Для того, чтобы объект мог быть сериализован, он должен реализовать интерфейс Serializable (соответствующее объявление должно явно присутствовать в классе объекта или, по правилам наследования, неявно в родительском классе вверх по иерархии).

Интерфейс Serializable не определяет никаких методов.

Его присутствие только определяет, что объекты этого класса разрешено сериализовывать.

При попытке сериализовать объект, не реализующий этот интерфейс, будет брошено исключение NotSerializableException.

После того, как объект был сериализован, то есть превращен в последовательность байт, его по этой последовательности можно восстановить, при этом восстановление можно проводить на любой машине (вне зависимости от того, где проводилась сериализация), то есть после-

довательность байт можно передать на другую машину по сети или любым другим образом, и там провести десериализацию, независимо от типа используемой операционной системы.

В этом примере мы создаем файловый поток, обертываем его в `ObjectOutputStream` и записываем в него объекты.

```
String s = "Hello world!";
int i = 897648764;
try {
    // create a new file with an ObjectOutputStream
    FileOutputStream out = new FileOutputStream("test.txt");
    ObjectOutputStream oout = new ObjectOutputStream(out);

    // write something in the file
    oout.writeObject(s);
    oout.writeObject(i);

    // close the stream
    oout.close();

    // create an ObjectInputStream for the file we created before
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream("test.txt"));
    // read and print what we wrote before
    System.out.println("" + (String) ois.readObject());
    System.out.println("" + ois.readObject());

} catch (Exception ex) {
    ex.printStackTrace();
}
```

После этого десериализуем эти объекты из файла.

Сериализуемый объект может хранить ссылки на другие объекты, которые в свою очередь так же могут хранить ссылки на другие объекты.

И все они тоже будут восстановлены при десериализации.

При этом, важно, что если несколько ссылок указывают на один и тот же объект, то в восстановленных объектах эти ссылки так же будут указывать на один и тот же объект.

Если класс содержит в качестве полей другие объекты, то эти объекты так же будут сериализовываться и поэтому тоже должны быть сериализуемы.

В свою очередь, сериализуемы должны быть и все объекты, содержащиеся в этих сериализуемых объектах и т. д.

Полный путь ссылок объекта по всем объектным ссылкам, имеющимся у него и у всех объектов, на которые у него имеются ссылки, и т. д. – называется графом исходного объекта.

Что бы указать, что сеанс сериализации завершен, и мы хотим заново записывать объекты, у `ObjectOutputStream` нужно вызвать метод `reset()`.

При десериализации никакие из конструкторов объекта не вызываются – даже конструктор по умолчанию.

То есть при десериализации объект просто восстанавливается в том виде в каком он был, а не конструируется заново.

Однако, вопрос, на который следует обратить внимание – что происходит с состоянием объекта, унаследованным от суперкласса.

Ведь состояние объекта определяется не только значениями полей, определенными в нем самом, но также и значениями полей, унаследованными от суперкласса.

В процессе десериализации, поля НЕ сериализуемых классов (родительских классов, НЕ реализующих интерфейс `Serializable`) иницируются вызовом конструктора без параметров.

Такой конструктор должен быть доступен из сериализуемого их подкласса.

В противном случае, во время выполнения будет брошено исключение `InvalidClassException`.

Поля сериализуемого класса восстанавливаются из потока.

Возможны ситуации, когда по некоторым причинам, необходимо самостоятельно управлять ходом сериализации и восстановления объекта.

Например, записывать не все поля, а только некоторые из них.

Для такого гибкого управления процессом сериализации используется интерфейс `Externalizable`.

java.io

Interface Externalizable

All SuperInterfaces:
`Serializable`

All Known SubInterfaces:
`RemoteRef`, `ServerRef`

All Known Implementing Classes:
`ActivationDataFlavor`, `DataFlavor`, `MimeType`, `Mlet`, `PrivateMlet`

Methods	
Modifier and Type	Method and Description
void	<code>readExternal(ObjectInput in)</code> The object implements the <code>readExternal</code> method to restore its contents by calling the methods of <code>DataInput</code> for primitive types and <code>readObject</code> for objects, strings and arrays.
void	<code>writeExternal(ObjectOutput out)</code> The object implements the <code>writeExternal</code> method to save its contents by calling the methods of <code>DataOutput</code> for its primitive values or calling the <code>writeObject</code> method of <code>ObjectOutput</code> for objects, strings, and arrays.

При использовании этого интерфейса в поток автоматически записывается только идентификация класса.

Сохранить и восстановить всю информацию по состоянию экземпляра, должен обеспечить сам класс.

Для получения классом полного контроля за форматом и содержанием потока, для объекта и его суперклассов, должны быть реализованы методы `writeExternal()` и `readExternal()` интерфейса `Externalizable`.

Эти методы должны полностью координировать сохранение состояния, унаследованное от суперкласса.

При сериализации, сериализуемый объект первым делом проверяется на поддержку интерфейса `Externalizable`.

Если объект реализует `Externalizable`, вызывается его метод `writeExternal`.

Если объект не поддерживает `Externalizable`, но реализует `Serializable`, используется стандартная сериализация `ObjectOutputStream`.

При восстановлении `Externalizable` объекта, экземпляр создается путем вызова `public` конструктора без аргументов, после чего вызывается метод `readExternal`.

В противовес этому, объекты `Serializable` восстанавливаются путем считывания из потока `ObjectInputStream`.

При управлении процессом сериализации не всегда имеет смысл обращаться к реализации интерфейса `Externalizable` или реализации методов `readObject`, `writeObject`, если вопрос состоит только в том, что нежелательно сохранять и восстанавливать некоторые объекты-члены экземпляра.

Обычно подобное требование возникает, когда в объекте хранится некоторая конфиденциальная информация, например, пароль.

Даже если поле такого объекта описано как `private`, оно все равно будет сериализовано, то есть записано в поток, а значит, это значение можно будет прочитать, а при умелом обращении и изменить.

Так же может потребоваться пропустить сохранение объекта, десериализация которого все равно не будет иметь смысла в последующем – например сетевое соединение все равно нужно будет устанавливать заново.

Один способ пропустить сохранение объекта – реализовать интерфейс `Externalizable`, или определить методы `readObject` и `writeObject` – тогда вообще вся запись и чтение проходят как будет указано.

Однако в данном случае можно поступить намного проще – достаточно такое поле объявить с модификатором `transient`.

Когда объект восстанавливается, таким полям выставляется значение по умолчанию, для объектов это null.

```
class Account implements java.io.Serializable {
    private String name;
    private String login;
    private transient String password;
    /* Some accessors and mutators for fields
    ...
    */
}
```

Сериализованный объект может храниться сколь угодно долго, например, если записать его на диск.

Тогда, на момент его десериализации может возникнуть такая ситуация, что в класс этого объекта уже внесены изменения – добавлены или изменены методы, поля и т. д.

Некоторые такие изменения могут изменить класс таким образом, что десериализация станет невозможной.

В этом случае попытка десериализации приведет к возникновению исключения `InvalidClassException`.

Например, если сериализовать объект класса `User`, определенного следующим образом. После чего модифицировать класс, заменив поле `name` на два.

```
class User implements java.io.Serializable{
    String name;
}
```

То при попытке десериализовать записанный ранее объект, будет брошено исключение `InvalidClassException`.

```
class User implements java.io.Serializable{
    protected String firstName;
    protected String lastName;
}
```

Однако этого не произошло бы, если класс изменить следующим образом.

Для отслеживания таких ситуаций, каждому классу присваивается его идентификатор версии.

```
class User implements java.io.Serializable{
    private String name;
    String lastName;
}
```

Он представляет собой число long (длина 64 бита), полученное при помощи хэш-функции.

Для его вычисления используются имена классов, всех реализуемых интерфейсов, всех методов и полей класса.

При десериализации объекта, идентификаторы класса и идентификатор, взятый из потока сравниваются.

Изменения, проводимые с классом можно разбить на две группы – совместимые, то есть изменения, которые можно производить в классе, и при этом поддерживать совместимость с ранними версиями, и несовместимые – изменения, нарушающие поддержку с ранней версией класса.

Определить к какому типу относится изменение, можно, руководствуясь следующей логикой: так как предполагается, что поздние версии будут поддерживать более ранние, то поздние должны иметь такой набор полей, что в них возможно восстановить поля из старых записей.

Итак, к совместимым относятся следующие изменения:

```
Совместимые изменения:
• добавление поля к классу
• добавление или удаление суперкласса
• изменение модификаторов доступа полей
• удаление у полей модификаторов static или transient
• изменение кода методов, инициализаторов, конструкторов

Несовместимые изменения:
• удаление поля
• изменение название пакета класса
• изменение типа поля
• добавление к полю экземпляра ключевого слова static или transient
• реализация Serializable вместо Externalizable или наоборот.
```

- добавление поля к классу
- добавление или удаление суперкласса
- изменение модификаторов доступа полей
- удаление у полей модификаторов static или transient
- изменение кода методов, инициализаторов, конструкторов

К несовместимым относятся:

- Удаление поля
- изменение название пакета класса
- изменение типа поля
- добавление к полю экземпляра ключевого слова `static` или `transient`
- реализация `Serializable` вместо `Externalizable` или наоборот.

Другими словами, должна остаться возможность восстановить именно те поля, которые были записаны в поток при сериализации.

Обеспечение такой совместимости может стать неприятной задачей, если не позаботиться об этом заранее.

В таких случаях сразу следует объявить, что класс реализует интерфейс `Externalizable`, и потом просто дорабатывать методы `readExternal` и `writeExternal`.

Символьные потоки



Tech Solutions

Объектно-ориентированное программирование на Java

Ввод-вывод и Date/Time API

Лекция 4

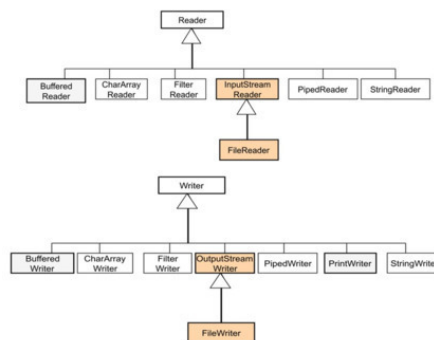
Символьные потоки

Рассмотренные нами классы – были наследниками классов `InputStream` и `OutputStream`, которые работают с байтовыми данными.

При этом, при работе со строковыми данными этими классами поддерживаются 8-ми битные символы и зачастую при интернационализации неверно происходит обработка 16-ти битных символов Unicode, на которых основан примитивный тип `char` (символ) в Java.

Для правильного использования символов в операциях ввода/вывода используются наследники классов `Reader` (чтение) и `Writer` (запись), которые представляют символьные потоки, в отличие от байтовых потоков.

Эта иерархия очень схожа с аналогичной для байтовых потоков `InputStream` и `OutputStream`.



Для символьных потоков нет потока, аналогичного потоку `SequenceInputStream` последовательного считывания, и, конечно же, отсутствует преобразование в символьное представление примитивных типов Java и объектов.

В свою очередь, в символьных потоках присутствует поток записи символьных данных в строку, чего нет для байтовых потоков.

Кроме того, в символьных потоках имеются классы-мосты, преобразующие символьные потоки в байтовые: `InputStreamReader` и `OutputStreamWriter`.

Есть также другие различия:

- Классы – основы фильтров для символьных потоков: `FilterReader` и `FilterWriter` являются абстрактными, а аналогичные для байтовых: `InputStream` и `OutputStream` – не абстрактные.

• Методы: `close` класса `Reader` и `flush` и `close` класса `Writer` являются абстрактными, в то время как в соответствующих классах `InputStream` и `OutputStream` байтовых потоков они имеют просто пустую реализацию

• `BufferedReader` наследуется не от фильтра `FilterReader`, а напрямую от `Reader`.

• `LineNumberReader` не наследуется от `FilterReader`, вместо этого он унаследован от `BufferedReader`, который в свою очередь напрямую унаследован от `Reader`

• `FileReader` и `FileWriter` унаследованы от классов-мостов `InputStreamReader` и `OutputStreamWriter`

• Метод `available ()` класса `InputStream`, отсутствует в классе `Reader`, он заменен методом `ready ()` – возвращающим вместо количества данных готовых к считыванию, булево значение – готов ли поток к считыванию (то есть будет ли считывание произведено без блокирования).

В остальном же, использование символьных потоков идентично работе с байтовыми потоками.

Этот пример показывает запись и чтение символьных данных в файл.

```
String fileName = "d:\\file.txt";
FileWriter fw = null;
BufferedWriter bw = null;
FileReader fr = null;
BufferedReader br = null;
//Строка, которая будет записана в файл
String data = "Some data to be written and readed\n";

try {
    fw = new FileWriter(fileName);
    bw = new BufferedWriter(fw);
    // Несколько раз записать строку
    for(int i=0; i<10; i++) bw.write(data);
    bw.close();
    fr = new FileReader(fileName);
    br = new BufferedReader(fr);
    String s = null;
    int count = 0;
    // Считывать данные, отображая на экран
    while((s=br.readLine())!=null)
        System.out.println("row" + ++count + " read:" + s);
    br.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Потоки символов часто являются «обертками» для потоков байтов.

Поток символов использует поток байтов для выполнения физического ввода-вывода, при этом поток символов обрабатывает перевод между символами и байтами.

Например, `FileReader` использует `FileInputStream`, а `FileWriter` использует `FileOutputStream`.

```
FileInputStream fis = null;
InputStreamReader isr = null;
char c;
int i;
try {
    // new input stream reader is created
    fis = new FileInputStream("C:/test.txt");
    isr = new InputStreamReader(fis, "UTF8");
    // read till the end of the file
    while((i = isr.read()) != -1) {
        // int to character
        c = (char) i;
        System.out.println("Character Read: " + c);
    }
} catch (Exception e) {
    // print error
    e.printStackTrace();
} finally {
    // closes the stream and releases resources associated
    if(fis != null)
        fis.close();
    if(isr != null)
        isr.close();
}
```

Существует два универсальных потока «моста» общего назначения: `InputStreamReader` и `OutputStreamWriter`.

Используйте их для создания потоков символов, когда нет готовых классов потока символов, которые отвечают вашим потребностям.

В этом примере `InputStreamReader` считывает байты и декодирует их в символы с использованием указанной кодировки.

Классы-мосты `InputStreamReader` и `OutputStreamWriter` имеют возможность производить преобразование символов, используя различные кодировки.

Кодировка задается при конструировании потока, путем передачи в конструктор в качестве параметра строки – названия кодировки.

Символьный ввод-вывод обычно встречается в виде ввода-вывода строк, а не отдельных символов.

Здесь элемент ввода-вывода – это строка символов со знаком окончания линии в конце.

Знак окончания линии зависит от платформы и может быть последовательностью возврат каретки и возврат строки (`>>\r\n`), просто возврат каретки (`>>\r`) или просто возврат строки (`>>\n`).

Поддержка всех возможных знаков окончания строки позволяет программам читать текстовые файлы, созданные в любой из широко используемых операционных систем.

Для строкового ввода-вывода используются два класса `BufferedReader` и `PrintWriter`.

Здесь мы создаем строковый входной поток на основе символьного файлового потока и строковый выходной поток.

```
BufferedReader inputStream = null;
PrintWriter outputStream = null;

try {
    inputStream = new BufferedReader(new FileReader("kanadu.txt"));
    outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

    String l;
    while ((l = inputStream.readLine()) != null) {
        outputStream.println(l);
    }
} finally {
    if (inputStream != null) {
        inputStream.close();
    }
    if (outputStream != null) {
        outputStream.close();
    }
}
```

И считываем с помощью метода `readLine`, а записываем с помощью метода `println`.

Теперь рассмотрим такую задачу, как разбивка строки или потока на значимые независимые слова.

Такая разбивка строки или потока называется токенизацией.

Java предлагает несколько инструментов для токенизации.

Для потоков это `StreamTokenizer`.

Этот класс позволяет работать с данными посредством токенов – кусков данных, выделяемых из потока по определенным свойствам.

```
FileReader reader = new FileReader("AllContent.txt");
StreamTokenizer st = new StreamTokenizer(reader);

double sum = 0;
int numWords = 0, numChars = 0;

while (st.nextToken() != st.TT_EOF) {
    if (st.ttype == StreamTokenizer.TT_NUMBER) {
        sum += st.nval;
    } else if (st.ttype == StreamTokenizer.TT_WORD) {
        numWords++;
        numChars += st.sval.length();
    }
}

System.out.println("Sum of total numbers in the file: " + sum);
System.out.println("Total words (does not include numbers) in the file: " + numWords);
System.out.println("No. of characters available in words: " + numChars);
```

`StreamTokenizer` может распознавать идентификаторы, числа, строки в кавычках и различные стили комментариев.

Обычно приложение сначала создает экземпляр этого класса, после чего многократно повторяет в цикле вызов метода `nextToken`, пока не будет возвращено значение `TT_EOF`, указывающее конец потока.

После вызова метода `nextToken` поле `ttype` содержит значение, определяющее тип последнего считанного токена.

`StreamTokenizer` поставляется с константами, используемыми для определения типа токена.

Если последний считанный токен распознан как число, его значение содержится в поле `nval`.

Если же токен распознан как слово, его значение заносится в поле `sval`.

Класс `StringTokenizer` пакета `java.util` позволяет приложению разбивать строку на токены.

Здесь метод токенизации намного проще, чем тот, который используется классом `StreamTokenizer`.

```
StringTokenizer st = new StringTokenizer("this is a test");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

Методы `StringTokenizer` не различают идентификаторы, числа и строки в кавычках, а также не распознают комментарии.

Набор разделителей (символы, разделяющие токены) могут быть указаны либо во время создания экземпляра класса, либо для каждого токена.

Токен возвращается, беря подстроку строки, которая была использована для создания объекта `StringTokenizer`.

`StringTokenizer` – это класс еще `JDK1.0`, который сохраняется по соображениям совместимости, хотя его использование не рекомендуется в новом коде.

Вместо него рекомендуется использовать метод `split` класса `String`.

Еще один способ разбиения ввода на токены и конвертацию отдельных токенов в соответствии с их типом данных, это использование класса `Scanner` пакета `java.util`.

```
String[] result = "this is a test".split("\\s");
for (int x=0; x<result.length; x++)
    System.out.println(result[x]);
```

По умолчанию сканер использует пробелы для разделения токенов.

```

Scanner s = null;
try {
    s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));
    while (s.hasNext()) {
        System.out.println(s.next());
    }
} finally {
    if (s != null) {
        s.close();
    }
}

Scanner s = null;
double sum = 0;
try {
    s = new Scanner(new BufferedReader(new FileReader("usnumbers.txt")));
    s.useLocale(Locale.US);
    while (s.hasNext()) {
        if (s.hasNextDouble()) {
            sum += s.nextDouble();
        } else {
            s.next();
        }
    }
} finally {
    s.close();
}
System.out.println(sum);

```

Здесь первый пример показывает именно это разбиение символьного потока.

Метод `next` возвращает следующий токен из потока.

Второй пример показывает извлечение из потока значений типа `double`.

Программа часто запускается из командной строки и взаимодействует с пользователем в среде командной строки.

Платформа Java поддерживает такое взаимодействие двумя способами: через стандартные потоки и с помощью консоли.

Стандартные потоки – это функция многих операционных систем.

По умолчанию они считывают ввод с клавиатуры и записывают вывод на дисплей.

Они также поддерживают ввод-вывод в файлах и между программами, но эту функцию контролирует интерпретатор командной строки, а не сама программа.

```

InputStreamReader cin = null;

try {
    cin = new InputStreamReader(System.in);
    System.out.println("Enter characters, 'q' to quit.");
    char c;
    do {
        c = (char) cin.read();
        System.out.print(c);
    } while (c != 'q');
} finally {
    if (cin != null) {
        cin.close();
    }
}

```

Платформа Java поддерживает три стандартных потока:

Стандартный ввод с помощью объекта `System.in`;

Стандартный вывод с помощью объекта `System.out`;

И стандартная ошибка с помощью объекта `System.err`.

Эти объекты определяются автоматически и их не нужно создавать.

Можно ожидать, что стандартные потоки будут символьными потоками, но по историческим причинам они являются потоками байтов.

`System.out` и `System.err` определяются как объекты `PrintStream`.

Хотя этот поток технически является байтовым потоком, `PrintStream` использует внутренний символьный поток для эмуляции функций символьных потоков.

Напротив, `System.in` является байтовым потоком без функций символьного потока.

Чтобы использовать стандартный ввод как поток символов, оберните `System.in` в `InputStreamReader`.

Стандартный вывод можно форматировать с помощью метода `format`.

Здесь `d` форматирует целочисленное значение как десятичное значение.

```
int i = 2;
double r = Math.sqrt(i);

System.out.format("The square root of %d is %f.%n", i, r);

The square root of 2 is 1.414214.
```

`f` форматирует значение с плавающей запятой как десятичное значение.

`s` форматирует любое значение в виде строки.

`n` выводит окончание линии для конкретной платформы.

Более сложной альтернативой стандартным потокам является консоль.

Этот предопределенный объект `Console` имеет большинство функций, предоставляемых стандартными потоками.

```
Console cnsl = null;
String name = null;
try {
    // creates a console object
    cnsl = System.console();
    // if console is not null
    if (cnsl != null) {
        // read line from the user input
        name = cnsl.readLine("Name: ");
        // prints
        System.out.println("Name entered : " + name);
    }
} catch (Exception ex) {
    // if any error occurs
    ex.printStackTrace();
}
```

Консоль особенно полезна для безопасного ввода пароля, так как имеет метод `readPassword`.

Консольный объект также предоставляет потоки ввода и вывода, которые являются истинными потоками символов, с помощью его методов `reader` и `writer`.

Прежде чем программа сможет использовать консоль, она должна попытаться восстановить объект консоли, вызвав `System.console`.

Если объект консоли доступен, этот метод возвращает его.

Если `System.console` возвращает `NULL`, то операции консоли не разрешены либо потому, что ОС не поддерживает их, либо потому, что программа была запущена в не интерактивной среде.

Объект `Console` поддерживает безопасный ввод пароля с помощью метода `readPassword`. Этот метод защищает ввод пароля двумя способами.

Во-первых, пароль не отображается на экране пользователя.

Во-вторых, `readPassword` возвращает массив символов, а не строку, поэтому пароль можно перезаписать, удалив его из памяти, как только он больше не понадобится.

Если классы потоков осуществляют реальную запись и чтение данных, то класс `File` – это вспомогательный инструмент, который облегчает работу с файлами и директориями.

Объект класса `File` является абстрактным представлением файла и пути к нему.

```
String dirname = "/tmp/user/java/bin";
File d = new File(dirname);

// Create directory now.
d.mkdirs();

try {
    //Whatever the file path is.
    File statText = new File("E:/Java/Reference/bin/images/statsTest.txt");
    FileOutputStream is = new FileOutputStream(statText);
    OutputStreamWriter osw = new OutputStreamWriter(is);
    Writer w = new BufferedWriter(osw);
    w.write("POTATO!!!");
    w.close();
} catch (IOException e) {
    System.err.println("Problem writing to the file statsTest.txt");
}
```

Он устанавливает только соответствие с файлом, при этом для создания объекта не важно, существует ли такой файл на диске.

После создания объекта можно сделать проверку, вызвав метод `exists`, который возвращает значение `true`, если файл существует.

Создание или удаление объекта класса `File` никаким образом не отображается на реальных файлах.

Этот класс не имеет методов для работы с содержимым файла.

Объект `File` может указывать на директорию (узнать это можно вызовом метода `isDirectory`), и тогда вызовом метода `list` можно получить список имен (массив `String`) файлов в директории (если объект `File` не указывает на директорию – будет возвращен `null`).

В этом примере мы выводим список файлов директории `tmp`.

```
File file = null;
String[] paths;

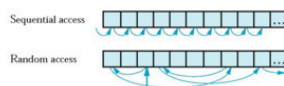
try {
    // create new file object
    file = new File("/tmp");

    // array of files and directory
    paths = file.list();

    // for each name in the path array
    for(String path:paths) {
        // prints filename and directory name
        System.out.println(path);
    }
} catch (Exception e) {
    // if any error occurs
    e.printStackTrace();
}
```

Класс `RandomAccessFile` позволяет перемещаться по файлу, читать из него или писать в него, как вам будет угодно.

Вы также сможете заменить существующие части файла, т.е. обновить фрагмент файла.



```
java.io
Class RandomAccessFile
java.lang.Object
  java.io.RandomAccessFile
All Implemented Interfaces:
  Closeable, DataInput, DataOutput, AutoCloseable
```

Это невозможно сделать с помощью `FileInputStream` или `FileOutputStream`, но `RandomAccessFile` дает эту возможность.

Этот класс реализует сразу два интерфейса: `DataInput` и `DataOutput` и, соответственно, может производить и запись, и чтение всех примитивных типов Java.

```
private static byte[] readFromFile(String filePath, int position, int size) throws IOException {
    RandomAccessFile file = new RandomAccessFile(filePath, "r");
    file.seek(position);
    byte[] bytes = new byte[size];
    file.read(bytes);
    file.close();
    return bytes;
}

private static void writeToFile(String filePath, String data, int position) throws IOException {
    RandomAccessFile file = new RandomAccessFile(filePath, "rw");
    file.seek(position);
    file.write(data.getBytes());
    file.close();
}
```

Эти операции записи и чтения производятся с файлами.

При этом эти операции можно производить поочередно, вдобавок произвольным образом перемещаясь по файлу с помощью вызова метода `seek` (узнать текущее положение указателя в файле можно вызовом метода `getFilePointer`).

При создании объекта этого класса конструктору в качестве параметров нужно передать два параметра: файл, и режим работы.

Файл, с которым будет проводиться работа указывается либо с помощью `String` – название файла, либо с помощью объекта `File`.

Режим работы (`mode`) – представляет собой объект `String`, который принимает одно из значений: «r»(чтение) либо «rw»(чтение и запись).

При этом, если передать, что `mode` равно «r» и указать несуществующий файл (либо директорию), будет брошено исключение `FileNotFoundException`.

Если указать, что `mode` равно «rw» и указать несуществующий файл, он будет создан (или же брошено исключение `FileNotFoundException` если это невозможно сделать).

После создания объекта `RandomAccessFile`, можно воспользоваться методами интерфейсов `DataInput` и `DataOutput` для проведения с файлом операций считывания и записи.

По окончании работы с файлом, объект `RandomAccessFile` нужно закрыть, вызвав метод `close`.

Java NIO



Tech Solutions

Объектно-ориентированное программирование на Java

Ввод-вывод и Date/Time API

Лекция 5

Java NIO

Помимо пакета `java.io` существует пакет `java.nio` или Java New IO, который также обеспечивает ввод/вывод в Java.

Отличия между этими двумя пакетами следующие.

Package java.nio.channels
Defines channels, which represent connections to entities that are capable of performing I/O operations, such as files and sockets; defines selectors, for multiplexed, non-blocking I/O operations.
See: Description

Interface Summary	
Interface	Description
<code>AsynchronousByteChannel</code>	An asynchronous channel that can read and write bytes.
<code>AsynchronousChannel</code>	A channel that supports asynchronous I/O operations.
<code>ByteChannel</code>	A channel that can read and write bytes.
<code>Channel</code>	A nexus for I/O operations.
<code>CompletionHandler<V,A></code>	A handler for consuming the result of an asynchronous I/O operation.
<code>GatheringByteChannel</code>	A channel that can write bytes from a sequence of buffers.
<code>InterruptibleChannel</code>	A channel that can be asynchronously closed and interrupted.
<code>MulticastChannel</code>	A network channel that supports Internet Protocol (IP) multicasting.

Основное отличие заключается в том, что Java IO является потокоориентированным, а Java NIO – буфер-ориентированным.

Потокоориентированный ввод/вывод подразумевает чтение/запись из потока/в поток одного или нескольких байт в единицу времени поочередно.

Таким образом, невозможно произвольно двигаться по потоку данных вперед или назад.

Если вы хотите произвести подобные манипуляции, вам придется сначала кэшировать все данные в буфере.

В случае Java NIO данные считываются в буфер для последующей обработки.

Вы можете двигаться по буферу вперед и назад.

Это дает немного больше гибкости при обработке данных.

В то же время, вам необходимо проверять содержит ли буфер объем данных, необходимый для корректной обработки.

Также необходимо следить, чтобы при чтении данных в буфер вы не уничтожили ещё не обработанные данные, находящиеся в буфере.

Надо отметить, что в библиотеке Java IO также есть буферизация с помощью `BufferedWriter` и `BufferedReader`, но нет доступа к самому буферу.

Кроме того Java NIO существенно быстрее Java IO.

Потоки ввода/вывода в Java IO являются синхронными.

Ввод/вывод в Java NIO являются асинхронным.

Это означает, что когда в потоке выполнения вызывается метод `read ()` или `write ()` любого класса из пакета `java.io.*`, происходит блокировка до тех пор, пока данные не будут считаны или записаны.

И поток выполнения программы в данный момент не может делать ничего другого.

Здесь надо различать поток выполнения программы и поток ввода-вывода.

В Java IO все операции ввода-вывода рассматриваются как движение одиночных байтов по одному за счет объекта, называемого поток `stream`.

Поток выполнения – это поток `thread`.

Асинхронный ввод Java NIO позволяет запрашивать считываемые данные из канала и получать только то, что доступно на данный момент, или вообще ничего, если доступных данных пока нет.

Вместо того, чтобы оставаться заблокированным пока данные не станут доступными для считывания, поток выполнения программы может выполнять что-то другое.

Тоже самое справедливо и для асинхронного вывода Java NIO.

Поток выполнения может запросить запись в канал некоторых данных, но не дожидаться при этом пока они не будут полностью записаны.

Таким образом асинхронный ввод-вывод Java NIO позволяет использовать один поток выполнения программы для решения нескольких задач вместо ожидания в заблокированном состоянии.

Наиболее частой практикой является использование сэкономленного времени работы потока выполнения на обслуживание операций ввода/вывода в другом или других каналах.

Синхронное выполнение кода часто вынуждено создавать другие потоки выполнения для работы с множеством подключений.

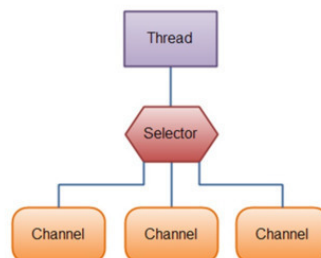
С асинхронным вводом-выводом вы можете работать в одном потоке выполнения без дополнительных потоков.

Центральным объектом в асинхронном вводе-выводе является селектор.

Селектор в Java NIO позволяет одному потоку выполнения мониторить несколько каналов ввода-вывода.

Вы можете зарегистрировать несколько каналов с селектором, а потом использовать один поток выполнения для обслуживания каналов, имеющих доступные для обработки данные, или для выбора каналов, готовых для записи.

Если вам необходимо управлять несколькими открытыми соединениями одновременно, причем каждое из них передает лишь незначительный объем данных, выбор Java NIO для вашего приложения может дать преимущество.



Если вы имеете одно соединение, по которому передается большой объем данных, то лучшим выбором станет классическая система ввода/вывода.

Здесь слева показан код с использованием Java IO, а справа код с использованием Java NIO.

```

BufferedReader br = null;
String sCurrentLine = null;
try
{
    br = new BufferedReader(
        new FileReader("test.txt"));
    while ((sCurrentLine = br.readLine()) != null)
    {
        System.out.println(sCurrentLine);
    }
}
catch (IOException e)
{
    e.printStackTrace();
}
finally
{
    try
    {
        if (br != null)
            br.close();
    }
    catch (IOException ex)
    {
        ex.printStackTrace();
    }
}
}

RandomAccessFile aFile = new RandomAccessFile(
    "test.txt", "r");
FileChannel inChannel = aFile.getChannel();
ByteBuffer buffer = ByteBuffer.allocate(1024);
while(inChannel.read(buffer) > 0)
{
    buffer.flip();
    for (int i = 0; i < buffer.limit(); i++)
    {
        System.out.print((char) buffer.get());
    }
    buffer.clear(); // do something with the data and clear/compact it.
}
inChannel.close();
aFile.close();

```

Теперь разберем пакет Java NIO более детально.

Каналы и буферы являются основными объектами в NIO и используются практически для каждой операции ввода-вывода.

Каналы аналогичны потокам в классическом пакете ввода-вывода.

Все данные, которые идут куда угодно (или происходят из любого источника), должны проходить через объект Channel канал.

```

public interface Channel
extends Closeable

A means for I/O operations.

A channel represents an open connection to an entity such as a hardware device, a file, a network socket, or a program component that is capable of performing one or more distinct I/O operations, for example reading or writing.

A channel is either open or closed. A channel is open upon creation, and once closed it remains closed. Once a channel is closed, any attempt to involve an I/O operation upon it will cause a ClosedChannelException to be thrown. Whether or not a channel is open may be tested by invoking its isOpen method.

Channels are, in general, intended to be safe for multithreaded access as described in the specifications of the interfaces and classes that extend and implement this interface.

```

Буфер является контейнером.

```

java.nio

Class Buffer

java.lang.Object
java.nio.Buffer

Direct Known Subclasses:
ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer

public abstract class Buffer
extends Object

A container for data of a specific primitive type.

A buffer is a linear, finite sequence of elements of a specific primitive type. Aside from its content, the essential properties of a buffer are its capacity, limit, and position.

A buffer's capacity is the number of elements it contains. The capacity of a buffer is never negative and never changes.

A buffer's limit is the index of the first element that should not be read or written. A buffer's limit is never negative and is never greater than its capacity.

A buffer's position is the index of the next element to be read or written. A buffer's position is never negative and is never greater than its limit.

There is one subclass of this class for each non-boolean primitive type.

```

Все данные, которые отправляются в канал, должны быть сначала помещены в буфер; аналогично, любые данные, которые считываются из канала, считываются в буфер.

Буфер – это объект, который содержит некоторые данные, которые должны быть записаны или которые только что были прочитаны.

Добавление объекта `Buffer` в NIO это одно из самых значительных различий между новой библиотекой и классической библиотекой ввода-вывода.

В поточно-ориентированном вводе-выводе вы писали и читали данные непосредственно из объектов `Stream`.

В библиотеке NIO все данные обрабатываются буферами.

Когда данные считываются, они считываются непосредственно в буфер.

Когда данные записываются, они записываются в буфер.

Каждый раз, когда вы получаете доступ к данным в NIO, вы вытаскиваете их из буфера.

Буфер по существу является массивом.

Как правило, это массив байтов, но могут использоваться другие типы массивов.

Но буфер больше, чем просто массив.

Буфер обеспечивает структурированный доступ к данным, а также отслеживает системные процессы чтения/записи.

Таким образом, в NIO у вас есть канал и буфер.

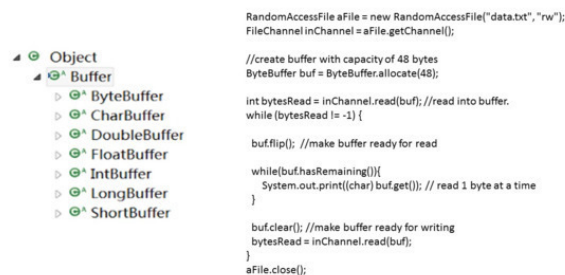
Вы считываете или записываете данные из или в канал через буфер порциями или блоками, размер которых определяется размером буфера.

И у вас получается поток блоков данных, а не непрерывный поток данных.

При этом у вас есть доступ к данным буфера непосредственно.

Наиболее часто используемым видом буфера является `ByteBuffer`.

`ByteBuffer` обеспечивает `get/set` операции в своем низлежащем массиве байтов.



`ByteBuffer` не является единственным типом буфера в NIO.

Фактически, для каждого из примитивных типов Java существует свой тип буфера – `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer`, `ShortBuffer`.

Каждый из классов `Buffer` является экземпляром интерфейса `Buffer`.

За исключением `ByteBuffer`, каждый из них имеет одни и те же самые операции, отличающиеся только типом данных, с которыми он имеет дело.

Так как `ByteBuffer` используется для большинства стандартных операций ввода-вывода, он имеет общие операции буфера, а также некоторые уникальные операции.

Канал – это объект, из которого вы можете читать данные и в который вы можете записывать данные.

Сравнивая Java NIO с Java IO, канал похож на поток.

Как я уже сказал, все данные обрабатываются через объекты `Buffer`.

Вы никогда не сможете записать байт непосредственно в канал; вместо этого вы запишете буфер, содержащий один или несколько байтов.

Аналогично, вы не сможете прочитать байт непосредственно из канала; вы считываете из канала в буфер, а затем получите байты из буфера.

Каналы отличаются от потоков тем, что они двунаправлены.

Если потоки `InputStream` или `OutputStream` идут только в одном направлении, канал может быть открыт для чтения, для записи или для чтения и записи.

Для чтения из канала мы просто создаем буфер, а затем запрашиваем канал для чтения данных.

Для записи в канал мы также создаем буфер, заполняем его данными, а затем просим канал записать из него.

Для чтения данных из файла, если бы мы использовали Java IO, мы бы просто создали `FileInputStream` и прочитали бы данные.

```
FileInputStream fin = new FileInputStream( "readandshow.txt" );
FileChannel fc = fin.getChannel();

ByteBuffer buffer = ByteBuffer.allocate( 1024 );

fc.read( buffer );
```

Однако в NIO все работает по-другому: сначала мы получаем объект `Channel` из `FileInputStream`, а затем используем этот канал для чтения данных.

Каждый раз, когда вы выполняете операцию чтения в NIO, вы читаете из канала, но вы не читаете напрямую из канала.

Так как все данные в конечном итоге находятся в буфере, вы читаете из канала в буфер. Поэтому чтение из файла включает в себя три этапа:

- (1) получение канала из `FileInputStream`;
- (2) создание буфера; и
- (3) чтение из канала в буфер.

Вы можете заметить, что нам не нужно указывать каналу, сколько данных читать в буфер.

Каждый буфер имеет сложную внутреннюю систему учета, которая отслеживает, сколько данных было прочитано и сколько есть места для большего количества данных.

Запись в файл в NIO похожа на чтение из файла.

Мы начинаем с получения канала из `FileOutputStream`.

```
FileOutputStream fout = new FileOutputStream( "writesomebytes.txt" );
FileChannel fc = fout.getChannel();

ByteBuffer buffer = ByteBuffer.allocate( 1024 );

for (int i=0; i<message.length; ++i) {
    buffer.put( message[i] );
}
buffer.flip();

fc.write( buffer );
```

Следующий шаг – создать буфер и поместить в него некоторые данные – в этом случае данные взяты из массива.

И последний шаг – записать в буфер.

Обратите внимание, что нам опять не нужно сообщать каналу, сколько данных мы хотели бы записать.

Внутренняя система учета буфера отслеживает, сколько данных он содержит и сколько осталось записать.

Здесь показан пример объединения чтения и записи.

```
FileInputStream fin = new FileInputStream( infile );
FileOutputStream fout = new FileOutputStream( outfile );

FileChannel fcin = fin.getChannel();
FileChannel fcout = fout.getChannel();

ByteBuffer buffer = ByteBuffer.allocate(1024 );

while (true) {
    buffer.clear();

    int r = fcin.read( buffer );

    if (r== -1) {
        break;
    }
    buffer.flip();

    fcout.write( buffer );
}
```

Методы `clear ()` и `flip ()` сбрасывают буфер и подготавливают его к записи считанных данных в другой канал.

Здесь также есть проверка, когда мы закончим копирование.

Мы закончили копирование, когда больше нет данных, и мы можем это сказать, когда метод `read` возвращает `-1`.

При каждой операции чтения/записи состояние буфера изменяется.

Записывая и отслеживая эти изменения, буфер может самостоятельно управлять своими ресурсами.

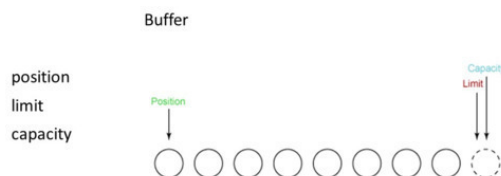
Когда вы читаете данные из канала, данные помещаются в буфер.

В некоторых случаях вы можете записать этот буфер непосредственно в другой канал, но также вы можете просмотреть данные буфера.

Это выполняется с использованием метода доступа `get ()`.

Аналогичным образом, если вы хотите поместить необработанные данные в буфер, вы используете метод доступа `put ()`.

Для определения состояния буфера в любой момент времени могут использоваться три значения:



position
limit
capacity

Вместе эти три переменные отслеживают состояние буфера и содержащиеся в нем данные.

Вспомним, что буфер – это массив.

Когда вы читаете канал, вы помещаете данные, которые вы считываете, в низлежащий массив.

Переменная `position` отслеживает, сколько данных вы записали.

Точнее, она указывает, в какой элемент массива будет идти следующий байт.

Таким образом, если вы прочитали три байта из канала в буфер, позиция этого буфера будет установлена в 3, обращаясь к четвертому элементу массива.

Потому что индексация начинается с нуля.

Аналогично, когда вы пишете в канал, вы получаете данные из буфера.

Значение позиции отслеживает, сколько данных вы получили из буфера.

Точнее, она указывает, из какого элемента массива будет поступать следующий байт.

Таким образом, если вы написали 5 байт в канал из буфера, позиция этого буфера будет установлена равной 5, ссылаясь на шестой элемент массива.

Переменная `limit` указывает, сколько данных осталось получить в случае записи из буфера в канал, или сколько осталось места для ввода данных, в случае чтения из канала в буфер.

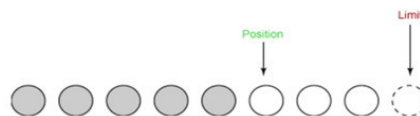
`position` всегда меньше или равна `limit`.

Переменная `capacity` буфера определяет максимальный объем данных, которые могут быть сохранены в нем.

По сути, она определяет размер низлежащего массива – или, по крайней мере, количество низлежащего массива, которое нам разрешено использовать.

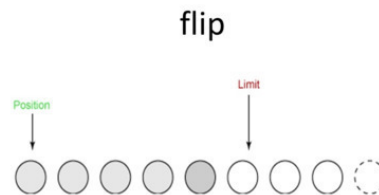
`limit` никогда не может быть больше `capacity`.

При чтении из канала в буфер у нас изменяется переменная `position`.



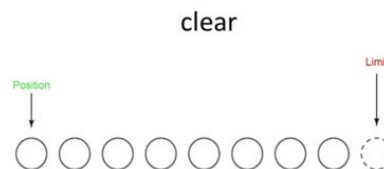
Переменная `limit` и `capacity` не изменяются.

Перед тем, записывать из буфера в канал, мы вызываем метод `flip`.



Он устанавливает переменную `limit` в текущую позицию, а переменную `position` в нуль.

Когда мы вызываем метод `clear`, он устанавливает переменную `limit` равную переменной `capacity`, а переменную `position` равную нулю.



Сейчас мы рассмотрели использование буфера для перемещения данных с одного канала в другой.

Однако часто нам нужно напрямую обращаться к данным.

Например, чтобы сохранить некие данные на диск.

В этом случае вам придется поместить эти данные непосредственно в буфер, а затем записать буфер на диск с помощью канала.

Или нужно считать данные с диска.

В этом случае вы будете считывать данные в буфер из канала, а затем анализировать данные из буфера.

В классе `ByteBuffer` существует четыре метода `get`, с помощью которых мы можем получить данные из буфера.

```
byte get();
ByteBuffer get( byte dst[] );
ByteBuffer get( byte dst[], int offset, int length );
byte get( int index );
```

Первый метод получает один байт.

Второй и третий методы считывают группу байтов в массив.

Четвертый метод получает байт из определенной позиции в буфере.

Методы, возвращающие `ByteBuffer`, просто возвращают это значение, для которого они вызываются.

Кроме того, первые три метода `get ()` являются относительными, а последний – абсолютным.

Относительный метод означает, что значения `limit` и `position` соблюдаются с помощью операции `get ()` – в частности, байт считывается из текущей позиции, и позиция увеличивается после вызова метода `get`.

Абсолютный метод игнорирует значения `limit` и `position` и не влияет на них.

Эти методы относятся к классу `ByteBuffer`.

Другие классы имеют эквивалентные методы `get ()`, которые идентичны, за исключением того, что вместо того, чтобы иметь дело с байтами, они имеют дело с типом, соответствующим этому классу буфера.

В классе `ByteBuffer` существует пять методов `put ()`.

```
ByteBuffer put( byte b );  
ByteBuffer put( byte src[] );  
ByteBuffer put( byte src[], int offset, int length );  
ByteBuffer put( ByteBuffer src );  
ByteBuffer put( int index, byte b );
```

Первый метод записывает один байт.

Второй и третий методы записывают группу байтов из массива.

Четвертый метод копирует данные из исходного `ByteBuffer` в данный `ByteBuffer`.

Пятый метод помещает байт в буфер в определенном месте.

Методы, возвращающие `ByteBuffer`, просто возвращают это значение, для которого они вызываются.

Как и в методах `get ()`, мы можем охарактеризовать методы `put ()` как относительные или абсолютные.

Первые четыре метода являются относительными, а пятый – абсолютным.

Эти методы относятся к классу `ByteBuffer`.

Другие классы имеют такие же методы `put ()`, которые идентичны, за исключением того, что вместо того, чтобы иметь дело с байтами, они имеют дело с типом, соответствующим данному классу буфера.

В дополнение к методам `get ()` и `put ()`, класс `ByteBuffer` также имеет дополнительные методы для чтения и записи значений разных типов.


```
getBytes()  
getChar()  
getShort()  
getInt()  
getLong()  
getFloat()  
getDouble()  
putByte()  
putChar()  
putShort()  
putInt()  
putLong()  
putFloat()  
putDouble()
```

Прежде чем вы сможете читать или писать данные, у вас должен быть буфер.

```
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
```

Чтобы создать буфер, вы должны выделить для него память.

Мы создаем буфер, используя статический метод `allocate`.

Метод `allocate` выделяет низлежащий массив указанного размера и обортывает его в объект-буфер – в этом случае `ByteBuffer`.

Вы также можете превратить существующий массив в буфер.

```
byte array[] = new byte[1024];  
ByteBuffer buffer = ByteBuffer.wrap( array );
```

В этом случае используется метод `wrap` для обортывания буфера вокруг массива.

Вы должны быть очень осторожны при выполнении этого типа операции.

Как только вы это сделаете, данные можно будет получить через буфер, а также напрямую.

Метод `slice()` создает вид подбуфера из существующего буфера.

```

ByteBuffer buffer = ByteBuffer.allocate( 10 );

for (int i=0; i<buffer.capacity(); ++i) {
    buffer.put( (byte)i );
}

buffer.position( 3 );
buffer.limit( 7 );
ByteBuffer slice = buffer.slice();

```

То есть он создает новый буфер, который берет данные из части исходного буфера.

Мы вырезаем часть буфера для создания подбуфера.

В некотором смысле подбуфер похож на окно на исходный буфер.

Вы указываете начало и конец окна, устанавливая положение и предельное значение, а затем вызываете метод `slice` буфера.

`slice` – это подбуфер буфера. Однако подбуфер и буфер используют один и тот же низлежащий массив данных.

Если вы возьмете подбуфер и с помощью него поменяете данные в низлежащем массиве, то эти измененные данные отобразятся и в самом буфере.

Таким образом, с помощью подбуфера вы можете ограничить свои действия в какой-либо части буфера.

Вы можете превратить любой обычный буфер в буфер только для чтения, вызвав его метод `asReadOnlyBuffer`, который возвращает новый буфер, который идентичен первому и имеет общие низлежащие данные с ним, но доступен только для чтения.

```

ByteBuffer bbuf = ByteBuffer.allocate(10);
int capacity = bbuf.capacity(); // 10
System.out.println(capacity);
bbuf.putShort(2,(short)123);

ByteBuffer buf = bbuf.asReadOnlyBuffer();

```

Буферы только для чтения полезны для защиты данных.

Когда вы передаете буфер методу некоторого объекта, вы не можете наверняка знать, попытается ли этот метод изменить данные в буфере.

Создание буфера только для чтения гарантирует, что буфер не будет изменен.

Вы не можете преобразовать буфер только для чтения в записываемый буфер.

Другим полезным видом `ByteBuffer` является прямой буфер.

Прямой буфер – это буфер, чья память выделяется особым образом для увеличения скорости ввода-вывода.

```

FileInputStream fin = new FileInputStream(infile);
FileOutputStream fout = new FileOutputStream(outfile);
FileChannel fcin = fin.getChannel();
FileChannel fcout = fout.getChannel();
ByteBuffer buffer = ByteBuffer.allocateDirect(1024);

while (true) {
    buffer.clear();
    int r = fcin.read(buffer);

    if (r== -1) {
        break;
    }
    buffer.flip();
    fcout.write(buffer);
}

```

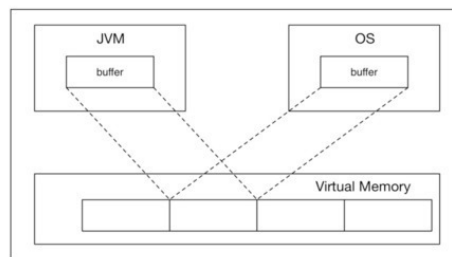
При наличии прямого байтового буфера, виртуальная машина Java будет выполнять нативные операции ввода-вывода непосредственно из него.

То есть она попытается избежать копирования содержимого буфера в (или из) промежуточного буфера до (или после) каждого вызова нативной операции ввода-вывода низлежащей операционной системы.

Вы также можете создать прямой буфер, используя файлы с отображением памяти.

Файл ввода-вывода с отображением памяти – это способ чтения и записи файлов, который может быть намного быстрее, чем обычный ввод-вывод на основе потока или канала.

Ввод-вывод с отображением памяти осуществляется путем того, что данные в файле отображаются в виде содержимого массива памяти.



Поначалу это звучит как просто чтение всего файла в память, но на самом деле это не так.

В общем случае в память выводятся или отображаются только те части файла, которые вы действительно считываете или записываете.

Отображение в памяти не является чем-то волшебным.

Современные операционные системы обычно реализуют файловые системы путем отображения частей файла в память, делая это по требованию.

Система отображения в памяти Java просто обеспечивает доступ к этому средству, если она доступна в низлежащей операционной системе.

Хотя это довольно просто создать, запись в файлы с отображением в памяти может быть опасной. Потому что в этом случае вы будете непосредственно изменять файл на диске.

В этом случае не существует разделения между изменением данных и сохранением их на диске.

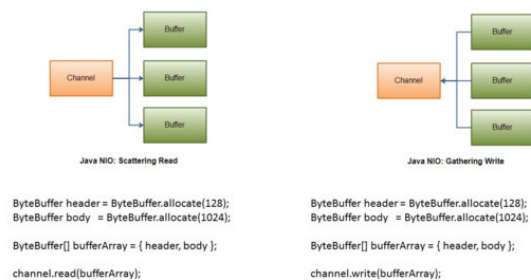
Отображение в памяти создается с помощью метода `map` канала `FileChannel`.

```
MappedByteBuffer mbb = fc.map( FileChannel.MapMode.READ_WRITE, 0, 1024 );
```

Метод `map` возвращает объект класса `MappedByteBuffer`, который является подклассом `ByteBuffer`.

Таким образом, вы можете использовать вновь отображенный буфер, как и любой другой `ByteBuffer`, и операционная система позаботится о том, чтобы сделать для вас отображение по требованию.

Scatter/gather I/O – это метод чтения и записи, который использует несколько буферов, а не один буфер, для хранения данных.



Рассеянное чтение аналогично чтению обычного канала, за исключением того, что оно считывает данные в массив буферов, а не в один буфер.

Аналогично, сборная запись записывает данные из массива буферов, а не из одного буфера.

Scatter/gather I/O полезен для разделения потока данных на отдельные разделы, что может помочь реализовать сложные форматы данных.

Для **scatter/gather I/O** есть два интерфейса `ScatteringByteChannel` и `GatheringByteChannel`. `ScatteringByteChannel` – это канал, который имеет два дополнительных метода чтения.

Эти методы `read` похожи на стандартные методы чтения, за исключением того, что вместо того, чтобы использовать один буфер, они используют массив буферов.

java.nio.channels

Interface ScatteringByteChannel

All Superinterfaces:
AutoCloseable, Channel, Closeable, ReadableByteChannel

All Known Implementing Classes:
DatagramChannel, FileChannel, Pipe.SourceChannel, SocketChannel

Method Summary

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
long	read(ByteBuffer[] dsts) Reads a sequence of bytes from this channel into the given buffers.	
long	read(ByteBuffer[] dsts, int offset, int length) Reads a sequence of bytes from this channel into a subsequence of the given buffers.	

При таком считывании канал поочередно заполняет каждый буфер.

Когда он заполняет один буфер, он начинает заполнять следующий.

В некотором смысле массив буферов рассматривается как один большой буфер.

Scatter/gather I/O полезен для разделения данных на разделы.

Например, вы можете написать сетевое приложение, которое использует объекты сообщений, и каждое сообщение делится на заголовок фиксированной длины и тело фиксированной длины.

Вы создаете один буфер, достаточно большой для заголовка, и еще один буфер, достаточно большой для тела.

Затем вы помещаете эти два буфера в массив и читаете в них, используя рассеяние, разделяя заголовок и тело между двумя буферами.

Сборная запись подобна рассеянному чтению.

java.nio.channels

Interface GatheringByteChannel

All Superinterfaces:
AutoCloseable, Channel, Closeable, WritableByteChannel

All Known Implementing Classes:
DatagramChannel, FileChannel, Pipe.SinkChannel, SocketChannel

Method Summary

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
long	write(ByteBuffer[] srcs) Writes a sequence of bytes to this channel from the given buffers.	
long	write(ByteBuffer[] srcs, int offset, int length) Writes a sequence of bytes to this channel from a subsequence of the given buffers.	

Здесь тоже есть методы, которые используют массив буферов.

Такая запись полезна для формирования единого потока данных из группы отдельных буферов.

В случае примера с сообщениями, вы можете использовать сборную запись, чтобы автоматически собирать компоненты сетевого сообщения в один поток данных для передачи по сети.

Java NIO позволяет заблокировать весь файл или часть файла.

```
RandomAccessFile raf = new RandomAccessFile( "usefilelocks.txt", "rw" );  
FileChannel fc = raf.getChannel();  
FileLock lock = fc.lock( start, end, false );  
...  
lock.release();
```

Блокировка файлов не всегда выполняется для защиты данных.

Например, вы можете временно заблокировать файл, чтобы гарантировать, что операция записи выполняется атомарно, без вмешательства других программ.

Если вы запросили эксклюзивную блокировку, никто другой не сможет получить блокировку в том же файле или части файла.

Если вы запросили общую блокировку, другие также могут получить общую блокировку, но не эксклюзивную блокировку, в том же файле или части файла.

Чтобы получить блокировку части файла, вы вызываете метод `lock` в открытом `FileChannel`.

Обратите внимание, что вы должны открыть файл для записи, если хотите получить эксклюзивную блокировку.

После того, как у вас есть блокировка, вы можете выполнить любые операции, которые вам нужны, а затем освободить блокировку.

После того, как вы отменили блокировку, у всех других программ, пытающихся получить блокировку, будет шанс сделать это.

Блокировка файлов может быть сложной задачей, особенно учитывая тот факт, что разные операционные системы реализуют блокировки по-разному.

Чтобы сохранить ваш код как можно более переносимым, используйте только эксклюзивные блокировки и рассматривайте все блокировки только как рекомендации.

File NIO



Объектно-ориентированное программирование на Java

Ввод-вывод и Date/Time API

Лекция 6

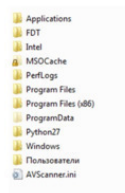
File NIO

В Java NIO также есть пакет File NIO, который обеспечивает всестороннюю поддержку файлового ввода-вывода и доступ к файловой системе.

В File NIO вводится понятие пути Path.

Объект Path – это более простая и гибкая замена для объекта File, и путь является основной отправной точкой для операций ввода-вывода файлов.

File NIO Path



IO	NIO
<code>File file = new File("D:/data");</code>	<code>Path path = Paths.get("D:/data");</code>
<code>file.createNewFile();</code>	<code>Files.createFile(path);</code>
<code>File file = new File("D:/data");</code>	<code>Path path = Paths.get("D:/data");</code>
<code>file.mkdir();</code>	<code>Files.createDirectory(path);</code>
<code>File file = new File("D:/data/java/pankaj");</code>	<code>Path path = Paths.get("D:/data/java/pankaj");</code>
<code>file.mkdirs();</code>	<code>Files.createDirectories(path);</code>
<code>File file = new File("D:/data");</code>	<code>Path path = Paths.get("D:/data");</code>
<code>file.exists();</code>	<code>Files.exists(path);</code>

Файловая система компьютера хранит и организует файлы на носителях.

Большинство используемых сегодня файловых систем хранят файлы в дереве или иерархической структуре.

В верхней части дерева находится корневой узел.

В корневом узле есть файлы и каталоги.

Каждый каталог может содержать файлы и подкаталоги, которые, в свою очередь, могут содержать файлы и подкаталоги и т. д.

Файл идентифицируется по его пути в файловой системе, начиная с корневого узла.

Путь является либо относительным, либо абсолютным.

Абсолютный путь всегда содержит корневой элемент и полный список каталогов, необходимых для поиска файла.

Относительный путь должен быть объединен с другим путем для доступа к файлу.

Интерфейс Path является центральным интерфейсом пакета `java.nio.file`.

Перейти от объекта File к объекту Path можно с помощью метода `toPath` класса File.

```

java.nio.file
Interface Path

All Superinterfaces:
ComparablePath, IterablePath, Watchable

public interface Path
extends ComparablePath, IterablePath, Watchable

An object that may be used to locate a file in a file system. It will typically represent a system-dependent file path.

A Path represents a path that is hierarchical and composed of a sequence of directory and file name elements separated by a special separator or delimiter. A root component, that identifies a file system hierarchy, may also be present. The name element that is furthest from the root of the directory hierarchy is the name of a file or directory. The other name elements are directory names. A Path can represent a root, a root and a sequence of names, or simply one or more name elements. A Path is considered to be an empty path if it consists solely of one name element that is empty. Accessing a file using an empty path is equivalent to accessing the default directory of the file system. Path defines the getFileName, getParent, getFlow, and subpath methods to access the path components or a subsequence of its name elements.

In addition to accessing the components of a path, a Path also defines the resolve and resolveSibling methods to combine paths. The relativize method that can be used to construct a relative path between two paths. Paths can be compared, and tested against each other using the startsWith and endsWith methods.

This interface extends Watchable interface so that a directory located by a path can be registered with a WatchService and entries in the directory watched.

```

Path является программным представлением пути в файловой системе.

Объект Path содержит имя файла и список каталогов, используемых для конструирования пути, и используется для проверки, поиска и обработки файлов.

Экземпляр Path отражает низлежащую платформу.

Здесь имеется в виду разделительный слэш в одну или другую сторону.

Поэтому вы не можете взять Path для файловой системы Linux и ожидать, что он будет соответствовать Path для файловой системы Windows, даже если структура каталогов идентична, и оба экземпляра обнаруживают один и тот же файл.

Файл или каталог, указанный Path, могут не существовать.

Вы можете просто создать экземпляр Path и манипулировать им различными способами.

В подходящее время вы можете использовать методы класса Files, чтобы проверить наличие файла, соответствующего Path, и создать файл, открыть его, удалить, изменить его разрешения и т. д.

Получить экземпляр Path можно с помощью метода `get` класса Paths.

```

Path p1 = Paths.get("some-file");
Path p2 = Paths.get("/usr/local/gosling/some-file");
Path p3 = Paths.get("C:\\Users\\Gosling\\Documents\\some-file");

toAbsolutePath, startsWith, endsWith, getFileName, getName,
getNameCount, subpath, getParent, getRoot, normalize, relativize

```

Далее с помощью методов уже интерфейса Path можно извлечь различную информацию о пути.

Например, конвертировать в строку, получить имя файла, получить элемент пути по определенному индексу, получить количество элементов пути, извлечь часть пути, получить путь родительского каталога, получить корень пути.


```

Path p1 = Paths.get("input-file.txt");

System.out.println("Simple Path");
System.out.printf("toString: %s\n\n", p1);

Path p2 = p1.toAbsolutePath();
System.out.println("Absolute Path");
System.out.printf("toString: %s\n", p2);

System.out.printf("getFileName: %s\n", p2.getFileName());
System.out.printf("getName(0): %s\n", p2.getName(0));
System.out.printf("getNameCount: %d\n", p2.getNameCount());
System.out.printf("subpath(0,2): %s\n", p2.subpath(0,2));
System.out.printf("getParent: %s\n", p2.getParent());
System.out.printf("getRoot: %s\n", p2.getRoot());

```

Класс `Files` является другим основным классом пакета `java.nio.file`.

```

java.nio.file
Class Files
java.lang.Object
  java.nio.file.Files

public final class Files
extends Object

This class consists exclusively of static methods that operate on files, directories, or other types of files.
In most cases, the methods defined here will delegate to the associated file system provider to perform the file operations.

Since:
1.7

```

Этот класс предлагает богатый набор статических методов для чтения, записи и управления файлами и каталогами.

Методы этого класса работают с экземплярами `Path`.

С помощью вызова метода `lines` класса `Files` можно получить поток `Stream` строк и использовать затем методы `map`, `filter`, `reduce`, `collect` и другие потока.

```
Stream lines = Files.lines(somePath);
```

Потоки помогают упростить обработку больших наборов данных, а лямбда и дженерики помогают сделать код более гибким и многообразным.

Работать с такими потоками можно несколькими способами.

Первый способ, это создать один метод, который обрабатывает поток, и второй метод, который вызывает `Files.lines` и передает поток первому методу.

Этот код многообразный, но каждая версия второго метода должна повторять много кода.

Второй способ, это определить функциональный интерфейс и статический метод, который может использовать лямбда. При этом первый метод обрабатывает поток, а второй метод передает имя файла и лямбда статическому методу.

И третий способ аналогичен предыдущему, но использует дженерики, чтобы возвращать значения.

Здесь мы определяем функциональный интерфейс, в котором определяем статический метод.

```
@FunctionalInterface
public interface StreamProcessor {
    void processStream(Stream<String> strings);

    public static void processFile(String filename, StreamProcessor processor) {
        try(Stream<String> lines = Files.lines(Paths.get(filename))) {
            processor.processStream(lines);
        } catch(IOException ioe) {
            System.err.println("Error reading file: " + ioe);
        }
    }
}
```

В этом методе мы создаем поток, исходя из пути файла, и передаем этот поток абстрактному методу интерфейса для обработки потока.

Обратите внимание, что здесь мы используем блок try-with-resources, который автоматически закрывает поток, так как поток реализует интерфейс AutoCloseable.

Затем мы можем вызвать статический метод интерфейса и передать в него имя файла и определить реализацию абстрактного метода интерфейса, как например фильтрацию по палиндромам.

```
StreamProcessor.processFile(filename, words -> words.filter(StringUtils::isPalindrome)
    .forEach(System.out::println));
```

Здесь мы создаем функциональный интерфейс, который использует дженерики.

```

@FunctionalInterface
public interface StreamAnalyzer<T> {
    T analyzeStream(Stream<String> strings);

    public static <T> T analyzeFile(String filename, StreamAnalyzer<T> analyzer) {
        try(Stream<String> lines = Files.lines(Paths.get(filename))) {
            return(analyzer.analyzeStream(lines));
        } catch(IOException ioe) {
            System.err.println("Error reading file: " + ioe);
            return(null);
        }
    }
}

StreamAnalyzer.analyzeFile(filename, stream -> letterCount(stream))

```

В статическом методе этого интерфейса создается поток, который затем передается абстрактному методу интерфейса для анализа потока.

Например, в реализации абстрактного метода интерфейса можно подсчитать длину текста или найти слово.

Методы `write` класса `Files` позволяют записать строки или байты в файл с помощью одного вызова метода.

```

List<String> lines = ...;
Files.write(somePath, lines, someCharset);

byte[] fileArray = ...;
Files.write(somePath, fileArray);

```

Обе версии `Files.write` могут принимать необязательный аргумент `OpenOption`.

```

Files.write(somePath, lines, someCharset, someOption);

Files.write(somePath, fileArray, someOption);

```

Это перечисление позволяет указать, следует ли создавать файл, если он не существует, переписывать этот файл или добавить к нему данные и т. д.

Поведение по умолчанию — это создать файл, если он отсутствует, и перезаписать, если он есть.

Здесь мы получаем кодировку по умолчанию.

```
Charset characterSet = Charset.defaultCharset();

Path path = Paths.get("output-file-1.txt");

List<String> lines =
    Arrays.asList("Line One", "Line Two", "Final Line");

Files.write(path, lines, characterSet);
```

Получаем путь файла.

Создаем список слов.

И записываем этот список слов в файл с кодировкой по умолчанию.

В этом примере мы берем файл, получаем поток и фильтруем поток по словам из четырех букв.

```
public static void main(String[] args) throws Exception {
    String inputFile = "enable1-word-list.txt";
    String outputFile = "four-letter-words.txt";
    int length = 4;
    List<String> words =
        Files.lines(Paths.get(inputFile))
            .filter(word -> word.length() == length)
            .map(String::toUpperCase)
            .distinct()
            .sorted()
            .collect(Collectors.toList());

    Files.write(Paths.get(outputFile), words, Charset.defaultCharset());

    System.out.printf("Wrote %s words to %s.%n",
        words.size(), outputFile);
}
```

Затем получившийся список записываем в другой файл.

При записи в файл часто нужно форматировать строки.

Метод `Files.write` не позволяет вам форматировать строки, когда вы вставляете их в файл.

Кроме того, для очень больших файлов требуется более высокая производительность, и вы не хотите хранить все данные в памяти в виде списка.

В этом случае можно использовать буферизированную запись Java IO, когда запись идет блоками и выполняется быстрее для очень больших файлов.

Для получения `BufferedWriter` используется метод `newBufferedWriter` класса `Files`.

```
Writer w = Files.newBufferedWriter(somePath, someCharset);
w.write(...);

PrintWriter out = новый PrintWriter(yourBufferedWriter);
out.printf(...);
```

Для форматирования вы можете обернуть `PrintWriter` вокруг `Writer`.

А затем использовать методы `print`, `println` и особенно `printf` для `PrintWriter`.
Здесь используется `BufferedWriter` для записи строк в файл.

```
public static void main(String[] args) throws IOException {
    Charset characterSet = Charset.defaultCharset();
    int numLines = 10;
    Path path = Paths.get("output-file-2.txt");
    try (BufferedWriter writer =
        Files.newBufferedWriter(path, characterSet)) {
        for(int i=0; i<numLines; i++) {
            writer.write("Number is " + 100 * Math.random());
            writer.newLine();
        }
    } catch (IOException ioe) {
        System.err.printf("IOException: %s%n", ioe);
    }
}
```

В этом примере используется `PrintWriter` для дополнительного форматирования строк при их записи в файл.

```
public static void main(String[] args) throws IOException {
    Charset characterSet = Charset.defaultCharset();
    int numLines = 10;
    Path path = Paths.get("output-file-3.txt");
    try (PrintWriter out =
        new PrintWriter(Files.newBufferedWriter(path, characterSet))) {
        for(int i=0; i<numLines; i++) {
            out.printf("Number is %5.2f%n", 100 * Math.random());
        }
    } catch (IOException ioe) {
        System.err.printf("IOException: %s%n", ioe);
    }
}
```

Помимо буферизированной записи, можно использовать и буферизированное чтение с помощью `BufferedReader` Java IO.

```
Charset charset = Charset.forName("US-ASCII");
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

В то время как потоковый ввод-вывод считывает символ за раз, каналный ввод-вывод Java NIO считывает буфер за раз.

```
try (SeekableByteChannel sbc = Files.newByteChannel(file)) {
    ByteBuffer buf = ByteBuffer.allocate(10);

    // Read the bytes with the proper encoding for this platform. If
    // you skip this step, you might see something that looks like
    // Chinese characters when you expect Latin-style characters.
    String encoding = System.getProperty("file.encoding");
    while (sbc.read(buf) > 0) {
        buf.rewind();
        System.out.print(Charset.forName(encoding).decode(buf));
        buf.flip();
    }
} catch (IOException x) {
    System.out.println("caught exception: " + x);
}
```

Интерфейс `ByteChannel` обеспечивает базовые функции чтения и записи.

`SeekableByteChannel` – это `ByteChannel`, который имеет возможность поддерживать позицию в канале и изменять эту позицию.

`SeekableByteChannel` также поддерживает усечение файла, связанного с каналом, и запрос файла по его размеру.

В этом примере используется Java NIO для чтения файла в буфер, используя канал.

`File NIO` позволяет работать с каталогами как с потоками путей.

Метод `Files.list` позволяет получить все файлы в папке.

```
Получить все файлы в папке
Files.list

Получить все файлы в папке и под ней
Files.walk

Получить соответствующие файлы в папке и под ней
Files.find
```

Метод `Files.walk` позволяет получить все файлы в папке и под ней.

Метод `Files.find` позволяет получить соответствующие файлы в папке и в ней.

Здесь мы получаем все пути в папке как поток и распечатываем его.

```
try(Stream<Path> paths = Files.list(Paths.get(folder))) {
    paths.forEach(System.out::println);
} catch (IOException ioe) {
    System.err.println("IOException: " + ioe);
}

try(Stream<Path> paths = Files.list(Paths.get(folder))) {
    Predicate<Path> test = p -> p.toString().endsWith(".txt");
    paths.filter(test).forEach(System.out::println);
} catch (IOException ioe) {
    System.err.println("IOException: " + ioe);
}
```

Ниже мы фильтруем этот поток по расширению файлов `txt`.

Метод `walk` работает аналогично, позволяя получить все дерево путей как поток.

```
try(Stream<Path> paths = Files.walk(Paths.get(rootFolder))) {  
    paths.forEach(System.out::println);  
  
    } catch(IOException ioe) {  
        System.err.println("IOException: " + ioe);  
    }  
}
```

Метод `find` позволяет получить пути, соответствующие определенному условию.

```
BiPredicate<Path, BasicFileAttributes> test = (path, attrs) ->  
    path.toString().endsWith(".java");  
  
try(Stream<Path> paths =  
    Files.find(Paths.get(rootFolder), 10, test)) {  
    paths.forEach(System.out::println);  
  
    } catch(IOException ioe) {  
        System.err.println("IOException: " + ioe);  
    }  
  
BiPredicate<Path, BasicFileAttributes> test =  
    (path, attrs) -> attrs.size() > 10000;
```

Атрибуты файла позволяют отфильтровать файлы от папок, запросить файл по его размеру и времени создания.

Ввод-вывод в Java 9



Объектно-ориентированное программирование на Java

Ввод-вывод и Date/Time API

Лекция 7

Ввод-вывод в Java 9

Сериализация Java – важная и полезная функция Java SE, которая позволяет разработчикам преобразовывать граф объектов Java в поток байтов для хранения или передачи, а затем обратно в граф объектов Java.

К сожалению, архитектура сериализации очень небезопасна и может привести к многочисленным уязвимостям, включая удаленное выполнение кода и атак типа «отказ в обслуживании».

Любая Java-программа, которая десериализует поток, подвержена таким уязвимостям, если не будут приняты соответствующие меры.

Этот код, десериализующий поток объектов, уязвим для атаки.

```
// Deserialize a string and date from a file.  
FileInputStream in = new FileInputStream("tmp");  
ObjectInputStream s = new ObjectInputStream(in);  
  
String today = (String)s.readObject();  
  
Date date = (Date)s.readObject();
```

Как показано в приведенном коде, код приложения вызывает метод `readObject` объекта `ObjectInputStream` для чтения объекта из потока.

Поток может включать любые объекты, а не только ожидаемые объекты `String` и `Date`. Если объект чтения не является объектом `String` и `Date`.

И операция кастинга приведет к исключению `ClassCastException` или исключению `ClassNotFoundException`, если не будет найдено определения для класса с указанным именем.

К сожалению, к моменту проверки типа код платформы уже выполнил значительную логику, которая может легко привести к успешной атаке.

В процессе десериализации байтового потока метод `readObject` может выполнить зловредный код до проверки типа.

Java 9 добавляет фильтрацию десериализации с помощью функционального интерфейса `ObjectInputFilter`, который позволяет определить фильтр для потока данных перед тем, как они будут десериализованы.

Interface `ObjectInputFilter`

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method	
<code>ObjectInputFilter.Status</code>	<code>checkInput</code>	<code>(ObjectInputFilter.FilterInfo filterInfo)</code>

```
public class ObjectInputStream ... {
    public final void setObjectInputFilter(ObjectInputFilter filter);
    public final ObjectInputFilter getObjectInputFilter(ObjectInputFilter filter);
}
```

Метод `checkInput` интерфейса позволяет проверить класс, длину массива, количество ссылок на объекты, глубину, размер потока и другую доступную информацию.

И этот фильтр может быть установлен для объекта `ObjectInputStream` методом `setObjectInputFilter`.

Метод `transferTo` класса `InputStream` считывает все байты из входного потока и записывает эти байты в выходной поток в том же самом порядке, в котором они считываются.

```
InputStream inputStream = new ByteArrayInputStream("test
string".getBytes());

Path path = Files.createTempFile("test-file", ".txt");

OutputStream outputStream = new FileOutputStream(path.toFile());

try (inputStream; outputStream) {
    inputStream.transferTo(outputStream);
}

Files.lines(path).forEach(System.out::println);
```

Этот метод не закрывает потоки и здесь используется выражение `try with resources` для закрытия обоих потоков.

Метод `readAllBytes` класса `InputStream` считывает все байты из входного потока.

```
try (InputStream inputStream = new ByteArrayInputStream("test string".getBytes())) {
    byte[] bytes = inputStream.readAllBytes();

    System.out.println(new String(bytes));

    byte[] bytesToRead = new byte[4];

    inputStream.readNBytes(bytesToRead, 0, bytesToRead.length);

    System.out.println(new String(bytesToRead));
}
```

Этот метод блокируется до тех пор, пока все байты не будут прочитаны, и не будет обнаружен конец потока или не будет выброшено исключение.

И этот метод не закрывает входной поток.

Метод `readNBytes` класса `InputStream` позволяет считать определенное количество байтов из входного потока в заданный массив байтов.

Хранение данных



Объектно-ориентированное программирование на Java

Взаимодействие с базами данных

Лекция 1

Хранение данных

Хранение данных в Java приложении может быть организовано с помощью переменных – в оперативной памяти компьютера, неупорядоченных данных с помощью файлов в файловой системе компьютера и упорядоченных данных с помощью баз данных.

Доступ к данным, хранящимся в переменных, организуется с помощью объявления и инициализации переменной.

Доступ к данным, хранящимся в файлах, организуется с помощью программного интерфейса I/O.

Доступ к данным, хранящимся в базах данных, также организуется с помощью программных интерфейсов, таких как JDBC, JPA и JDO.

Программный интерфейс JDBC (Java Data Base Connectivity) – это низкоуровневый API взаимодействия с реляционной базой данных, используя чистый язык запросов SQL.

JDBC (Java Data Base Connectivity) – низкоуровневый API взаимодействия с реляционной базой данных, используя чистый язык запросов SQL.

Java Persistence API (JPA) — предоставляет возможность сохранять в удобном виде Java-объекты в базе данных.

Существует несколько реализаций этого интерфейса, одна из самых популярных Hibernate.

JPA реализует концепцию ORM (Object Relational Mapping).

Java Data Objects (JDO) - способ доступа к данным в базах данных, используя простые старые объекты Java (POJO) для представления сохраняемых данных.

При использовании JDBC вам необходимо перевести набор результатов (по существу, матрицу строк/столбцов значений из одной или нескольких таблиц базы данных, возвращаемую вашим SQL-запросом) в объекты Java.

Программный интерфейс JPA (Java Persistence API) является высокоуровневым API для реляционного сопоставления объектов Object Relational Mapping.

Это технология, которая позволяет отображать объекты в таблицы баз данных.

Эта технология скрывает SQL от разработчика, так что все, с чем они имеют дело, – это классы Java.

Под капотом JPA использует JDBC для взаимодействия с реляционными базами данных.

Программный интерфейс Java Data Objects (JDO) также технология сохранения Java объектов в базах данных.

Если JPA сконцентрирована только на реляционных базах, то JDO – это более общая спецификация, которая описывает ORM Object Relational Mapping для любых возможных баз и хранилищ.

Хотя надо заметить, что и для JPA существует ряд реализаций для noSql баз данных.

Под капотом, для реляционных баз данных Java Data Objects JDO также использует JDBC.

Таким образом, в случае реляционных баз данных, JDBC является основой, и поэтому мы будем изучать JDBC.

JDBC



Tech Solutions

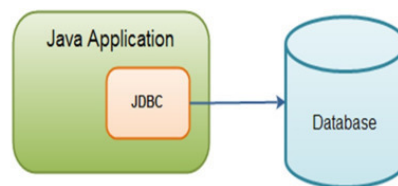
Объектно-ориентированное программирование на Java

Взаимодействие с базами данных

Лекция 2

JDBC

JDBC означает Java Database Connectivity и является стандартным Java API для связи, независимой от базы данных, между языком программирования Java и широким спектром баз данных.



JDBC API является частью платформы Java и разделен на два пакета: `java.sql` и `javax.sql`. Оба пакета включены в платформы Java SE и Java EE.

Библиотека JDBC обеспечивает подключение к базе данных, создание SQL выражений, выполнение SQL запросов в базе данных, просмотр и изменение результирующих записей.

Package java.sql
Provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java™ programming language.
See: Description

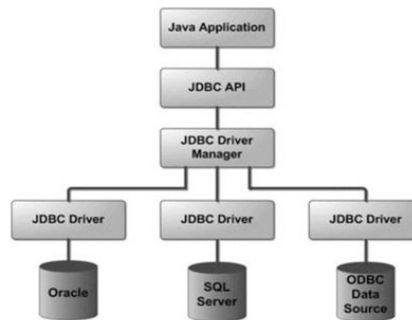
Interface Summary	Description
Array	The mapping in the Java programming language for the SQL type ARRAY.
Blob	The representation (mapping) in the Java™ programming language of an SQL BLOB value.
CallableStatement	The interface used to execute SQL stored procedures.
Clob	The mapping in the Java™ programming language for the SQL CLOB type.
Connection	A connection (session) with a specific database.
DatabaseMetaData	Comprehensive information about the database as a whole.
Driver	The interface that every driver class must implement.
DriverAction	An interface that must be implemented when a <i>Driver</i> wants to be notified by <i>DriverManager</i> .

Package javax.sql
Provides the API for server-side data source access and processing from the Java™ programming language.
See: Description

Interface Summary	Description
CommonDataSource	Interface that defines the methods which are common between <i>DataSource</i> , <i>XADataSource</i> and <i>ConnectionPoolDataSource</i> .
ConnectionEventListener	An object that registers to be notified of events generated by a <i>PooledConnection</i> object.
ConnectionPoolDataSource	A factory for <i>PooledConnection</i> objects.
DataSource	A factory for connections to the physical data source that this <i>DataSource</i> object represents.
PooledConnection	An object that provides hooks for connection pool management.
RowSet	The interface that adds support to the JDBC API for the <i>javaBeans™</i> component model.
RowSetInternal	The interface that a <i>RowSet</i> object implements in order to present itself to a <i>RowSetReader</i> or <i>RowSetWriter</i> object.

Так как же JDBC обеспечивает универсальность взаимодействия с широким спектром баз данных.

Архитектура JDBC состоит из двух уровней.



Программный интерфейс JDBC API обеспечивает соединение приложения с JDBC-менеджером.

Программный интерфейс JDBC Driver API обеспечивает соединение JDBC-менеджера со специфичным драйвером конкретной базы данных.

Таким образом универсальный программный интерфейс JDBC API использует диспетчер драйверов и драйвера, специфичные для баз данных, для обеспечения подключения к базам данных.

Менеджер драйверов JDBC гарантирует, что для доступа к каждому источнику данных используется правильный драйвер. Менеджер драйверов способен поддерживать несколько параллельных драйверов, подключенных к нескольким базам данных.

Вместо менеджера драйверов можно использовать объект DataSource, зарегистрированный в службе именования Java Naming and Directory Interface (JNDI), для установления соединения с источником данных.

Драйвер JDBC представляет собой набор классов Java, который позволяет вам подключаться к определенной базе данных. Например, база данных MySQL имеет собственный драйвер JDBC.

Драйвер JDBC, как Java классы, реализует интерфейсы JDBC API.

И ваш код использует данный JDBC-драйвер с помощью стандартных интерфейсов JDBC.

При этом конкретный драйвер JDBC скрыт за реализацией интерфейсов JDBC.

Таким образом, вы можете подключить новый драйвер JDBC, не изменяя свой код.

JDBC предназначен для получения или внесения данных из Java кода в источник данных или базу данных.

База данных – это средство хранения информации с возможностью получения или внесения в нее информации.

А реляционная база данных – это та, которая представляет информацию в виде таблиц со строками и столбцами, где строки, это объекты одного типа, а столбцы, это поля объекта.

Система управления базами данных (СУБД) – это приложение, которое обеспечивает способ хранения, внесения и получения данных.

Реляционные таблицы следуют определенным правилам целостности, чтобы гарантировать, что данные, которые они содержат, остаются точными и всегда доступны.

Во-первых, строки в реляционной таблице должны быть разными. Если есть повторяющиеся строки, могут возникнуть проблемы, какая из двух является правильной.

Второе правило целостности традиционной реляционной модели заключается в том, что столбец может содержать только отдельные значения, а не списки или массивы. Составные значения должны быть разделены на отдельные столбцы.

Третий аспект целостности данных содержит концепцию нулевого значения.

База данных заботится о ситуациях, когда данные могут быть недоступны, используя нулевое значение, чтобы указать, что значение отсутствует.

Так как каждая строка в таблице отличается, для идентификации определенной строки можно использовать один или несколько столбцов. Этот уникальный столбец или группа столбцов называются первичным ключом.

Любой столбец, который является частью первичного ключа, не может быть нулевым.

SQL – это язык, предназначенный для использования с реляционными базами данных. Существует набор базовых SQL-команд, которые считаются стандартными и используются всеми реляционными базами данных.

Когда один пользователь обращается к данным в базе данных, другой пользователь может одновременно получать одни и те же данные. Если, например, первый пользователь обновляет некоторые столбцы в таблице, и в то же время второй пользователь выбирает столбцы из этой же таблицы, второй пользователь может получить частично старые данные и частично обновленные данные. По этой причине СУБД используют транзакции для поддержания данных в согласованном состоянии, одновременно позволяя нескольким пользователям обращаться к базе данных.

Транзакция представляет собой набор из одного или нескольких операторов SQL, которые составляют логическую единицу работы. Транзакция завершается фиксацией или откатом, в зависимости от наличия проблем с согласованностью данных.

Блокировка – это механизм, который запрещает одновременную работу двух транзакций с одними и теми же данными.

Хранимая процедура представляет собой группу операторов SQL, которые могут быть вызваны по имени. Другими словами, это мини-программа, которая выполняет определенную задачу, которую можно вызвать так же, как можно вызвать функцию или метод.

После записи хранимой процедуры ее можно использовать повторно, так как СУБД, поддерживающая хранимые процедуры, будет хранить ее в базе данных.

Базы данных хранят пользовательские данные, а также хранят информацию о самой базе данных. Большинство СУБД имеют набор системных таблиц, которые перечисляют таблицы в базе данных, имена столбцов в каждой таблице, первичные ключи, внешние ключи, хранимые процедуры и т. д.

JDBC предоставляет интерфейс DatabaseMetaData, который каждый разработчик драйвера должен реализовать, чтобы его методы возвращали информацию о драйвере и / или СУБД, для которой написан драйвер.

Этот интерфейс предоставляет пользователям стандартизованный способ получения метаданных.

Пример



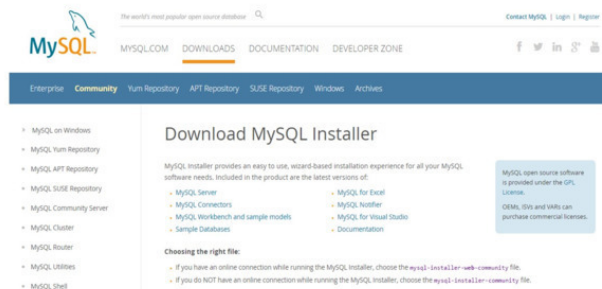
Объектно-ориентированное программирование на Java

Взаимодействие с базами данных

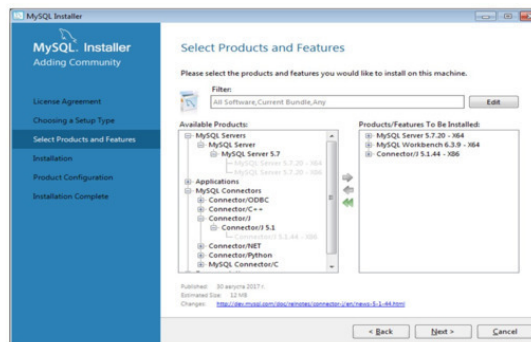
Лекция 3

Пример

Лучше всего понять работу с JDBC можно на примере.
Для начала скачаем MySQL Installer.



Определим для установки саму базу данных, это MySQL Server, приложение MySQL Workbench для администрирования базы данных, и драйвер Connector/J для подключения к MySQL кода Java.



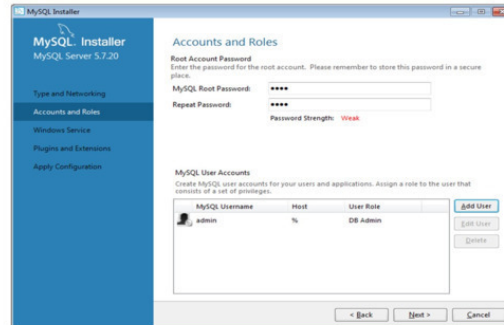
Существуют различные типы JDBC драйверов.

Connector/J это полностью Java-драйвер, который напрямую подключается к базе данных.

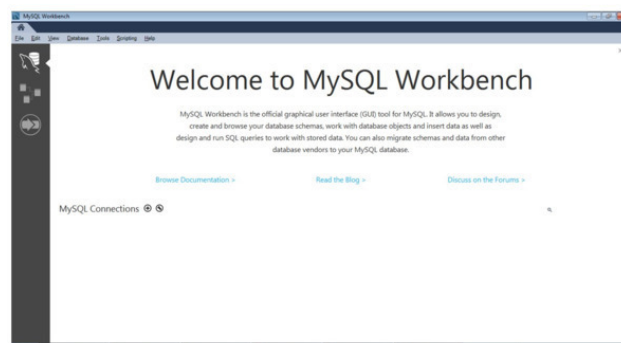
Есть также драйвер JDBC, который состоит из Java-части, которая переводит вызовы интерфейса JDBC на вызовы ODBC. Затем мост ODBC вызывает драйвер ODBC для данной базы данных.

Драйвер ODBC (Open Database Connectivity) – это стандартный драйвер Microsoft для доступа к базе данных.

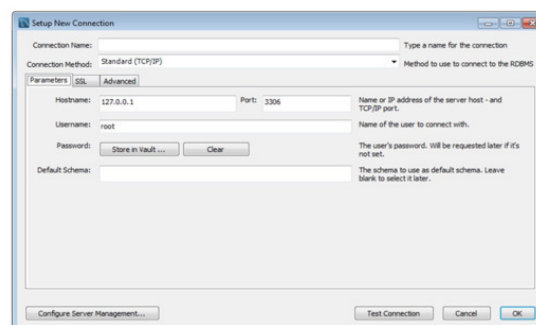
После установки базы данных, определите пароль корневого пользователя и определите администратора базы данных.



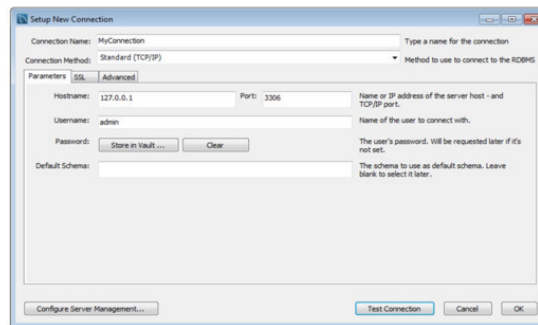
После этого запустите конфигурирование базы данных.
Запустите приложение MySQL Workbench.



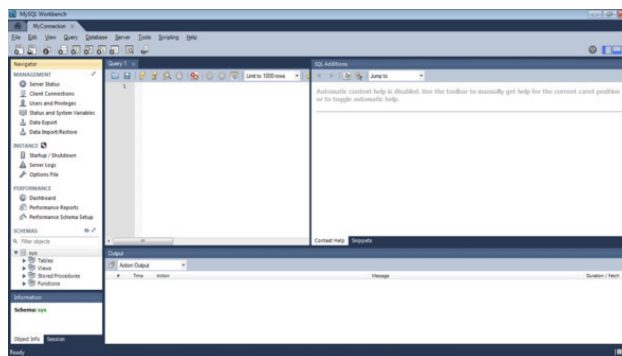
И создайте новое соединение с базой данных.



С помощью кнопки Test Connection протестируйте соединение.

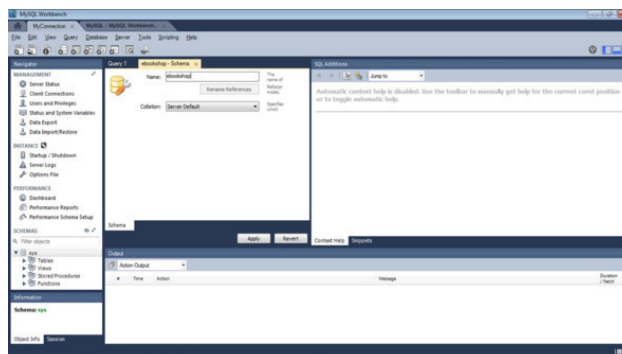


И откройте созданное соединение.

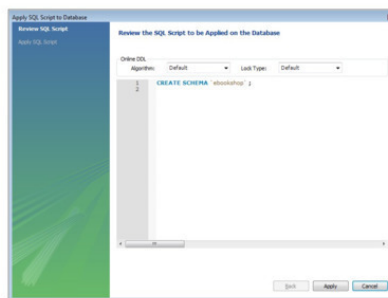


Для создания новой базы данных, в панели инструментов нажмем кнопку create a new schema.

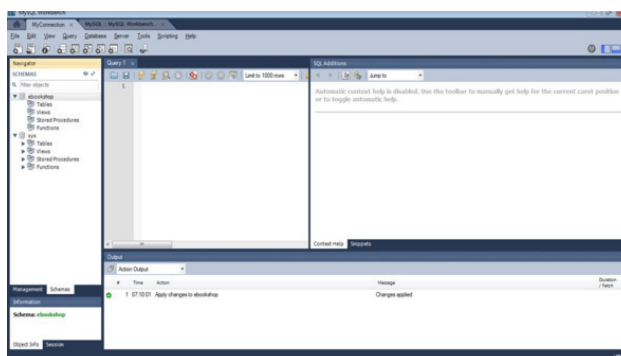
При нажатии кнопки Apply будет создан SQL запрос к базе данных.



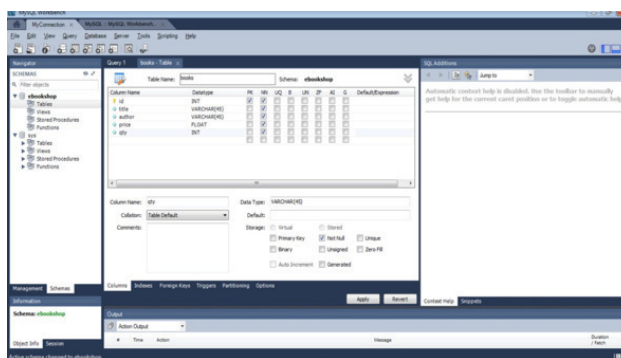
В результате выполнения SQL запроса будет создана новая база данных.



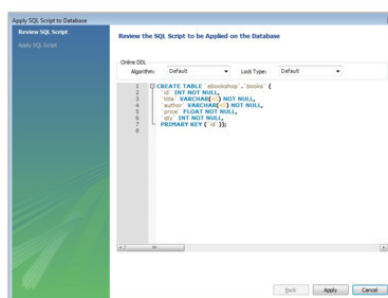
Создадим новую таблицу.



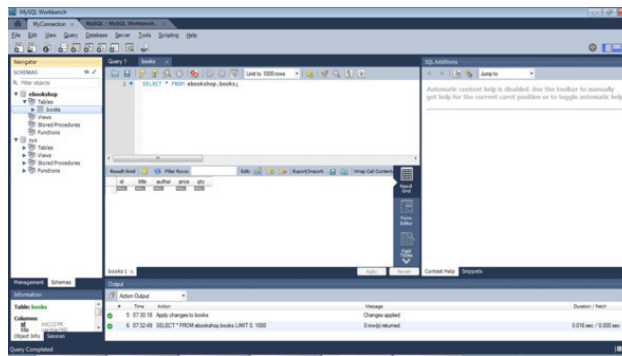
Добавим в таблицу столбцы.



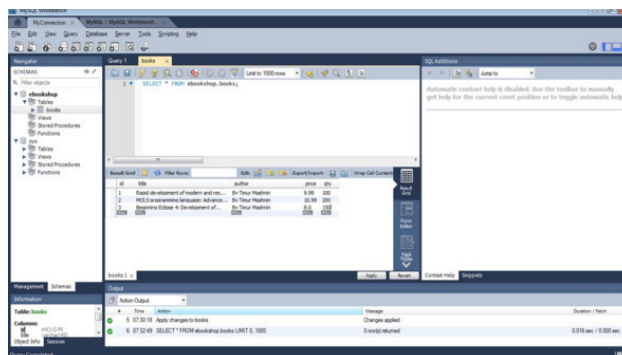
И применим соответствующий SQL запрос.



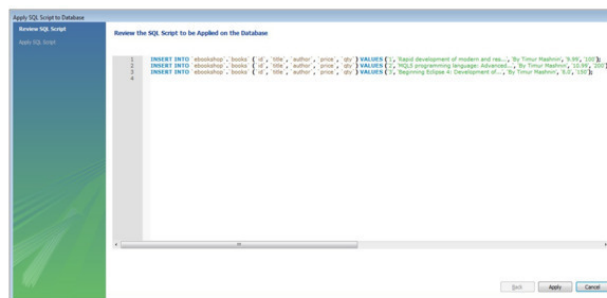
Нажмем правой кнопкой мышки на таблице и выберем Select Rows.



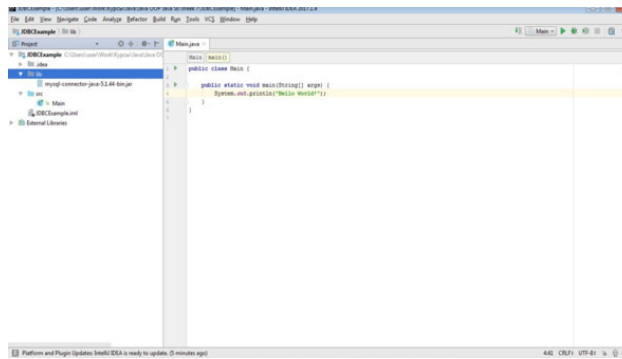
Внесем данные в таблицу.



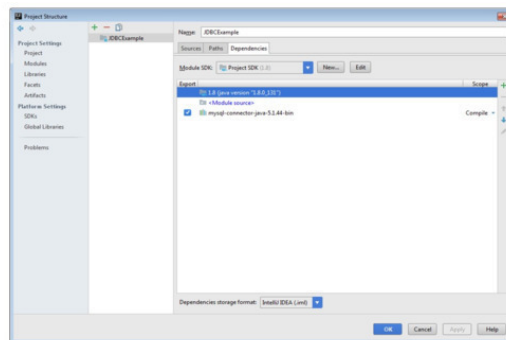
Нажмем кнопку Apply и выполним SQL скрипт.



Таким образом, мы создали базу данных с таблицей и внесли в нее данные. Дальше создадим Java проект в IntelliJ IDEA. В проекте создадим папку lib и скопируем в нее JAR файл драйвера Connector/J.



Добавим зависимость от библиотеки драйвера.



И добавим следующий Java код.

```
import java.sql.*;

public class Main {
    public static void main(String[] args) {
        try {
            Connection conn =
                DriverManager.getConnection("jdbc:mysql://localhost:3306/ebookshop?useSSL=false","admin","1234");
            Statement stmt = conn.createStatement();
        }
        {
            String strSelect = "select title, price, qty from books";
            ResultSet rset = stmt.executeQuery(strSelect);
            while(rset.next()) {
                String title = rset.getString("title");
                double price = rset.getDouble("price");
                int qty = rset.getInt("qty");
                System.out.println(title + ", " + price + ", " + qty);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Здесь Connection – это интерфейс, который обеспечивает соединение (сеанс) с конкретной базой данных. В контексте этого соединения выполняются операторы SQL.

Это соединение создается с помощью DriverManager, который под капотом использует конкретный драйвер Connector/J, добавленный нами предварительно в путь приложения.

DriverManager – это базовый сервис по управлению набором драйверов JDBC.

Когда вызывается метод getConnection, DriverManager пытается найти подходящий драйвер из числа загруженных при инициализации приложения.

По умолчанию, драйвер Connector/J использует протокол SSL, шифруя все данные (кроме начального рукопожатия) между драйвером JDBC и сервером.

Для использования протокола SSL требуется клиентский сертификат.

В данном случае мы отменили использование протокола SSL с помощью свойства useSSL=false.

После того, как мы установили соединение с базой данных, мы создаем объект `Statement` для выполнения оператора `SQL` и возврата результатов.

Далее мы создаем само `SQL` выражение и выполняем его.

В результате мы получаем объект `ResultSet`, представляющий таблицу данных, возвращаемую `SQL` запросом.

Метод `next` позволяет перемещаться по строкам этой таблицы, а `getter` методы позволяют получить значения столбцов данной строки таблицы данных.

Также здесь мы используем выражение `try-with-resources`, которое автоматически закрывает все открытые ресурсы, в нашем случае объекты `Connection` и `Statement`.

Задание

Измените Java-программу, чтобы выполнить следующие операторы `SELECT` и отобразить все найденные столбцы. Убедитесь, что вы изменили обработку `ResultSet` для обработки только извлеченных столбцов (в противном случае вы получите ошибку «Столбец не найден»).

`SELECT * FROM books`

`SELECT title, author, price, qty FROM books WHERE author = «...» OR price >= ... ORDER BY price DESC, id ASC`

Чтобы обновить данные в базе данных и выполнить `SQL` оператор `UPDATE`, вы должны вызвать метод `executeUpdate` объекта `Statement`, который возвращает число `int`, указывающее количество затронутых записей.

Задание

Измените Java-программу, чтобы выполнить следующие операторы `SELECT` и отобразить все найденные столбцы.

Убедитесь, что вы изменили обработку `ResultSet` для обработки только извлеченных столбцов (в противном случае вы получите ошибку «Столбец не найден»).

`SELECT * FROM books`
`SELECT title, author, price, qty FROM books WHERE author = '...' OR price >= ... ORDER BY price DESC, id ASC`

```
String strUpdate = "update books set price = price*0.7, qty = qty+1 where id = 1";

stmt.executeUpdate(strUpdate);

strSelect = "select * from books where id = 1";
rset = stmt.executeQuery(strSelect);

while(rset.next()) {
    System.out.println(rset.getInt("id") + ", "
        + rset.getString("author") + ", "
        + rset.getString("title") + ", "
        + rset.getDouble("price") + ", "
        + rset.getInt("qty"));
}
```

Напомним, что для оператора `SELECT` мы используем `executeQuery`, который возвращает объект `ResultSet`, представляющий возвращенную таблицу.

`SQL` операторы `UPDATE`, `INSERT` и `DELETE` не возвращают таблицу, а возвращают число `int`, которое указывает количество затронутых записей.

Задание

Задание

Измените Java-программу для выполнения следующих операторов SQL:

Увеличьте цену на 50% для книги 2.
Установите qty в 0 для книги 3.

Измените Java-программу для выполнения следующих операторов SQL:
Увеличьте цену на 50% для книги 2.
Установите qty в 0 для книги 3.

```
String sqlDelete = "delete from books where id>=3 and id<=4";

stmt.executeUpdate(sqlDelete);

String sqlInsert = "insert into books " + "values (3, 'Book', 'Author', 11.11, 11),(4, 'Book', 'Author', 33.33, 33)";

stmt.executeUpdate(sqlInsert);

strSelect = "select * from books";
rset = stmt.executeQuery(strSelect);
while(rset.next()) {
    System.out.println(rset.getInt("id") + ", "
        + rset.getString("author") + ", "
        + rset.getString("title") + ", "
        + rset.getDouble("price") + ", "
        + rset.getInt("qty"));
}
```

Для удаления или вставки строки в таблицу базы данных, используется метод `executeUpdate` для выполнения операторов «INSERT INTO» и «DELETE FROM».

Метод возвращает `int`, указывающий количество затронутых записей.

Вы не можете вставить две записи с тем же значением первичного ключа (т. е. `id`).

Поэтому, здесь мы выполняем оператор `DELETE` перед вставкой новой записи с помощью `INSERT`. Таким образом, вы можете повторно запустить программу.

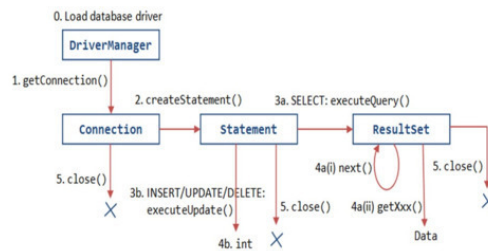
Задание**Задание**

Измените Java-программу, чтобы выполнить следующие операторы SQL:

Удалите все книги с `id=> 5`; и вставьте: (5, «Java ABC», «Kevin Jones», 15.55, 55) и (6, «Java XYZ», «Кевин Джонс», 25.55, 55);

Измените Java-программу, чтобы выполнить следующие операторы SQL:

Удалите все книги с `id=> 5`; и вставьте: (5, «Java ABC», «Kevin Jones», 15.55, 55) и (6, «Java XYZ», «Кевин Джонс», 25.55, 55);



В общем и целом, шаблон работы с JDBC подразумевает загрузку драйвера, создание соединения, создание SQL выражения, выполнение SQL выражения и обработку полученных результатов.

PreparedStatement



Объектно-ориентированное программирование на Java

Взаимодействие с базами данных

Лекция 4

PreparedStatement

Иногда, вместо Statement, удобнее использовать объект PreparedStatement для отправки оператора SQL в базу данных.

```
import java.sql.*;

public class MainPreparedStatement {
    public static void main(String[] args) {

        String strUpdate = "update books set price = ?, qty = ? where id = ?";
        String strSelect = "select title, price, qty from books";

        try {
            Connection conn =
                DriverManager.getConnection("jdbc:mysql://" +
                    "localhost:3306/" +
                    "ebookshop?useSSL=false", "admin", "1234");
            PreparedStatement update = conn.prepareStatement(strUpdate);
            PreparedStatement select = conn.prepareStatement(strSelect);

            {
                update.setFloat(1, (float) 9.0);
                update.setInt(2, 300);
                update.setInt(3, 2);
                update.executeUpdate();
                ResultSet rset = select.executeQuery();
                while(rset.next()) {
                    String title = rset.getString("title");
                    double price = rset.getDouble("price");
                    int qty = rset.getInt("qty");
                    System.out.println(title + ", " + price + ", " + qty);
                }
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Если вы хотите многократно выполнять объект Statement, лучше использовать объект PreparedStatement.

Основная особенность объекта PreparedStatement заключается в том, что, в отличие от объекта Statement, PreparedStatement получает оператор SQL при своем создании.

И этот оператор SQL сразу отправляется в СУБД, где он компилируется.

В результате объект PreparedStatement содержит не просто оператор SQL, а оператор SQL, который был предварительно скомпилирован.

Это означает, что при выполнении PreparedStatement, база данных просто запускает оператор SQL без необходимости его компиляции.

Объекты PreparedStatement могут использоваться для операторов SQL без параметров, но чаще всего используются для операторов SQL, которые принимают параметры.

Преимущество использования операторов SQL, использующих параметры, заключается в том, что вы можете использовать один и тот же оператор и предоставлять ему разные значения параметров каждый раз, когда вы его выполняете.

Параметр указывается в выражении SQL знаком вопроса, и затем подставляется значением с помощью set методов PreparedStatement.

В этом примере мы используем один объект PreparedStatement без параметров и один объект PreparedStatement с параметрами.

Транзакции

Иногда требуется изолировать несколько SQL операторов, так, чтобы при их выполнении другой клиент базы данных не смог изменить промежуточные данные.



Объектно-ориентированное программирование на Java

Взаимодействие с базами данных

Лекция 5

Транзакции

```
import java.sql.*;

public class MainTransactions {
    public static void main(String[] args) throws SQLException {
        Connection conn = null;
        PreparedStatement updateP = null;
        PreparedStatement updateQ = null;
        PreparedStatement select = null;
        String strUpdateP = "update books set price = ? where id = ?";
        String strUpdateQ = "update books set qty = ? where id = ?";
        String strSelect = "select title, price, qty from books";
        try {
            conn = DriverManager.getConnection("jdbc:mysql://" +
                "localhost:3306/" + "ebookshop?useSSL=false", "admin", "1234");
            updateP = conn.prepareStatement(strUpdateP);
            updateQ = conn.prepareStatement(strUpdateQ);
            select = conn.prepareStatement(strSelect);
            conn.setAutoCommit(false);
            updateP.setFloat(1, (float) 9.99);
            updateP.setInt(2, 2);
            updateP.executeUpdate();
            updateQ.setInt(1, 250);
            updateQ.setInt(2, 2);
            updateQ.executeUpdate();
            conn.commit();

            ResultSet rset = select.executeQuery();
            while (rset.next()) {
                String title = rset.getString("title");
                double price = rset.getDouble("price");
                int qty = rset.getInt("qty");
                System.out.println(title + ", " + price + ", " + qty);
            }
        } catch (SQLException e) {
            if (conn != null) {
                conn.rollback();
            }
        } finally {
            conn.setAutoCommit(true);
            if (updateP != null) {
                updateP.close();
            }
            if (updateQ != null) {
                updateQ.close();
            }
            if (conn != null) {
                conn.close();
            }
        }
    }
}
```

Например, в нашем примере, чтобы принять решение о скидке, обновляя количество книг, требуется также обновить их цену в одно и тоже время, иначе данные об общей стоимости книг будут непоследовательными.

Транзакция представляет собой набор из одного или нескольких операторов, которые выполняются как единое целое, поэтому либо все операторы выполняются, либо ни один из операторов не выполняется.

Когда создается соединение с базой данных, оно находится в режиме автоматической фиксации.

Это означает, что каждый отдельный оператор SQL рассматривается как транзакция и автоматически фиксируется сразу после его завершения.

Способ включения двух или более операторов в транзакцию состоит в отключении режима автоматической фиксации с помощью метода `setAutoCommit(false)`.

После того, как режим автоматической фиксации отключен, операторы SQL не фиксируются до тех пор, пока вы не вызовете явно метод `commit`.

При этом производится блокировка уровня строк или всей таблицы базы данных.

Фиксация (`commit`) транзакции закрепляет проведенные вами изменения, а откат (`rollback`) – отменяет их.

Как только вы зафиксировали транзакцию, все прочие транзакции других пользователей, которые начались после нее, смогут видеть изменения, проведенные вашими транзакциями.

Потому что после фиксации транзакции, снимаются блокировки с данных базы данных.

Откат всей транзакции, до успешного вызова метода `commit`, выполняется с помощью вызова метода `rollback`.

Вызов метода `rollback` завершает транзакцию и возвращает любые значения, которые были изменены до их предыдущих значений.

Транзакцию может прервать выброс исключения, что также снимет блокировки с данных. При этом часть операторов SQL будет выполнена, а часть нет.

Поэтому, если вы пытаетесь выполнить один или несколько операторов в транзакции и получаете `SQLException`, вызовите метод `rollback`, чтобы завершить транзакцию, откатить все данные и попытаться начать транзакцию снова.

В случае выброса исключения, если приложение продолжается и использует результаты транзакции, вызов метода `rollback` в блоке `catch` предотвращает использование возможно неправильных данных.

Задание

Задание

Смоделируйте выброс исключения и посмотрите, как работает `commit` и `rollback`.
Самостоятельно используйте точку сохранения `savepoint`.

Смоделируйте выброс исключения и посмотрите, как работает `commit` и `rollback`.
Самостоятельно используйте точку сохранения `savepoint`.

DataSource

Мы использовали класс `DriverManager` для получения объекта соединения с источником данных.



Объектно-ориентированное программирование на Java

Взаимодействие с базами данных

Лекция 6 DataSource

```
import com.mysql.jdbc.jdbc2.optional.MysqlDataSource;
import java.sql.*;

public class MainDataSource {
    public static void main(String[] args) {
        MysqlDataSource dataSource = new MysqlDataSource();
        dataSource.setServerName("localhost");
        dataSource.setPortNumber(3306);
        dataSource.setDatabaseName("ebookshop");
        dataSource.setUser("admin");
        dataSource.setPassword("1234");

        try {
            Connection conn = dataSource.getConnection();
            Statement stmt = conn.createStatement();
        }
        {
            String strSelect = "select title, price, qty from books";
            ResultSet rset = stmt.executeQuery(strSelect);
            while(rset.next()) {
                String title = rset.getString("title");
                double price = rset.getDouble("price");
                int qty = rset.getInt("qty");
                System.out.println(title + ", " + price + ", " + qty);
            }
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Вместо `DriverManager` можно использовать объект `DataSource` для получения соединения с источником данных.

Объект `DataSource` предоставляется драйвером JDBC.

Если речь идет об использовании базы данных на стороне сервера, обычно используется служба именования JNDI, с помощью которой объект `DataSource` связывается с логическим именем.

Далее в коде объект `DataSource` получается с использованием этого логического имени.

Здесь, в этом примере, мы просто используем реализацию интерфейса `DataSource`, предоставляемую драйвером `Connector/J`.

После получения объекта `DataSource` и его конфигурирования под базу данных, мы устанавливаем соединение с базой данных и дальше действуем также, как и с `DriverManager`.

В отличие от `DriverManager`, `DataSource` может обеспечивать пул соединений и распределенные транзакции, что важно при создании клиент-серверных приложений уровня предприятия.

Для этого используются реализации `ConnectionPoolDataSource` и `XADataSource`.

Пул соединений – это кеш объектов подключения к базе данных.

Во время выполнения приложение запрашивает соединение из пула.

Если пул содержит соединение, которое может удовлетворить запрос, он возвращает соединение приложению.

Если соединения не найдены, создается новое соединение и возвращается в приложение.

Приложение использует соединение для выполнения некоторой работы с базой данных, а затем возвращает объект обратно в пул.

Затем соединение доступно для следующего запроса на соединение.

Пулы соединений способствуют повторному использованию объектов соединения и уменьшают количество создаваемых объектов соединения.

Пулы соединений значительно повышают производительность приложений с интенсивным использованием базы данных, так как создание объектов соединения является дорогостоящим как с точки зрения времени, так и с точки зрения ресурсов.

Локализация и интернационализация. Введение



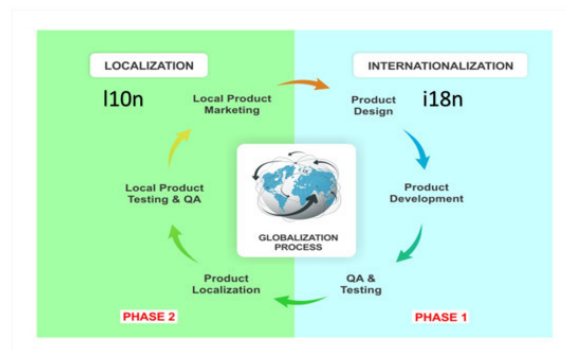
Объектно-ориентированное программирование на Java

Интернационализация и локализация

Лекция 1

Введение

При разработке программного обеспечения или веб-сайтов для глобального рынка необходимо учитывать различия каждого из локальных рынков.



Некоторые из этих различий могут включать в себя язык, пунктуацию, валюту, даты, время, цифры и часовые пояса.

Создание локализованной версии программного обеспечения может стать проблематичным и громоздким, когда изменения должны быть реализованы во множестве версиях для множества регионов.

Альтернативой индивидуальной настройке программного обеспечения для каждого региона является интернационализация.

Примером программного обеспечения, использующего интернационализацию, является Microsoft Office, который имеет один исходный код с языковыми пакетами, доступными в качестве надстройки.

Интернационализация – это процесс разработки приложения, так, чтобы оно могло быть адаптировано к различным языкам и регионам без технических изменений.

Иногда термин интернационализация сокращается как i18n, потому что между первым «i» и последним «n» в слове *Internationalization* имеется 18 букв.

С добавлением локализованных данных в такое приложение один и тот же исполняемый файл может работать по всему миру.

При этом текстовые элементы в коде программы не закодированы жестко.

Вместо этого они хранятся вне исходного кода и динамически подставляются.

Данные, такие как даты и валюты, отображаются в форматах, соответствующих региону и языку конечного пользователя.

При этом поддержка новых языков не требует перекомпиляции.

Таким образом, такое интернационализированное приложение можно быстро локализовать.

Локализация – это процесс адаптации программного обеспечения для определенного региона или языка путем добавления компонентов, специфичных для языка и региона.

Термин локализация часто сокращается как l10n, потому что между «l» и «n» в слове *Localization* имеется 10 букв.

Основная задача локализации – это языковой перевод текстовых элементов пользовательского интерфейса и документации.

Кроме того, локализация включает не только изменение языка, но и другие соответствующие изменения, такие как отображение чисел, дат, валюты и т. д.

Чем лучше интернационализировано приложение, то есть значения элементов, требующих локализации, отделены от кода, тем проще локализовать приложение.

Наборы ресурсов



Tech Solutions

Объектно-ориентированное программирование на Java

Интернационализация и локализация

Лекция 2

Наборы ресурсов

Прежде чем начать интернационализацию приложения, рекомендуется создать список атрибутов программного обеспечения, которые будут отличаться от одного региона к другому.

Java предоставляет классы для обработки интернационализации, чтобы свести к минимуму количество нового кода, который должен быть написан для обработки каждого элемента.

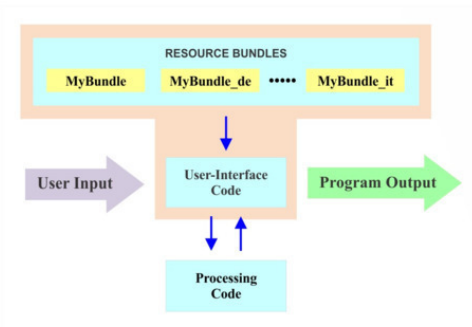
Язык
Даты
Время
Числа
Валюта
Единицы измерения
Телефонные номера
Названия
Почтовые адреса
Сообщения
Цвета
Графика

Первым шагом в интернационализации является определение значений, которые нужно отделить от кода.

Не интернационализированное приложение содержит все эти значения, закодированные в коде.

В интернационализированном приложении, для хранения текстовых сообщений, информации о форматировании и изображениях, ориентированных на определенную локализацию, могут использоваться наборы ресурсов Resource Bundles.

Resource Bundles автоматически изолируют данные, зависящие от локализации, чтобы информация, зависящая от локализации, не нуждалась в жестком кодировании в приложении.



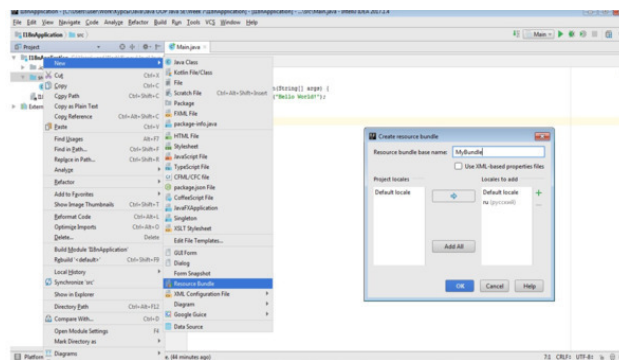
Resource Bundles позволяют хранить и извлекать всю информацию, относящуюся к локализации.

Они обеспечивают поддержку нескольких локализаций в одном приложении.

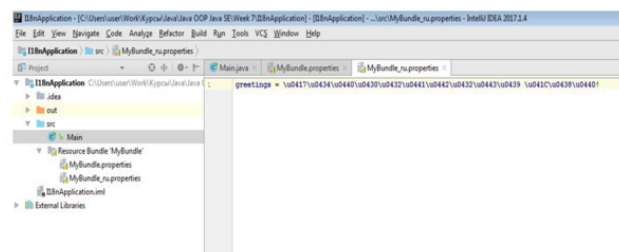
Resource Bundles обеспечивают добавление локализаций с помощью добавления дополнительных пакетов ресурсов.

Пакет ресурсов может быть представлен двумя способами.

Первый способ – это создать файл свойств, представляющий из себя простой текстовый файл с расширением. properties.



Файл свойств описывает ресурсы с помощью пар ключ-значение, где значение локализовано.



Файл свойств содержит текст в кодировке ISO 8859—1.


```

ISO-8859-1

baseName.properties
baseName_language.properties
baseName_language_country.properties
baseName_language_country_variant.properties

_ISO-639_ISO-3166

```

Символы, которые не могут быть представлены в кодировке ISO-8859-1, должны быть представлены с помощью Unicode Escapes.

При этом Java автоматически обеспечивает перевод символов в Unicode символы.

Расширение файла должно быть. properties.

Файл свойств может содержать в своем имени язык, который определен ISO-639, код страны, который определен ISO-3166, для которой производится локализация.

По умолчанию используется английский язык и файл свойств без указания кода языка и страны.

Другой способ создать пакет ресурсов – это расширить класс ListResourceBundle, в котором также определить ресурсы в виде пар ключ-значение, где значение ключа локализовано.

```

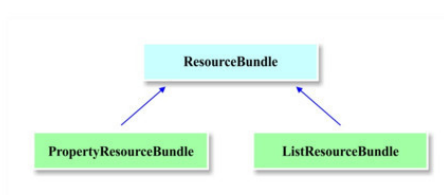
public class MyResources_fr extends ListResourceBundle {
    protected Object[][] getContents() {
        return new Object[][] {
            // LOCALIZE THIS
            {"s1", "Le disque \"{1}\" {0}."}, // MessageFormat pattern
            {"s2", "1"}, // location of {0} in pattern
            {"s3", "Mon disque"}, // sample disk name
            {"s4", "ne contient pas de fichiers"}, // first ChoiceFormat choice
            {"s5", "contient un fichier"}, // second ChoiceFormat choice
            {"s6", "contient {0,number} fichiers"}, // third ChoiceFormat choice
            {"s7", "3 mars 1996"}, // sample date
            {"s8", new Dimension(1,3)} // real object, not just string
        };
        // END OF MATERIAL TO LOCALIZE
    }
}

```

Теперь, после создания локализованных пакетов ресурсов, как передать их в код приложения?

Пакет ресурсов становится доступен в коде приложения с помощью объекта класса java.util.ResourceBundle.

Объект ResourceBundle действует как контейнер, содержащий пары ключ / значение.



При этом ключ идентифицирует конкретное значение в пакете ресурсов.

Класс `ListResourceBundle` обеспечивает доступ к ресурсам, определенным с помощью создания класса, а класс `PropertyResourceBundle` обеспечивает доступ к ресурсам, определенным с помощью файла свойств.

Если подкласс `ListResourceBundle` и файл. `properties` имеют одно и тоже имя, класс выигрывает и будет загружен первым.

После загрузки экземпляры пакета ресурсов кэшируются и этот кэш не может быть изменен.

Объект `ResourceBundle` получается с помощью статического метода `getBundle` класса `ResourceBundle`.

```
import java.util.Locale;
import java.util.ResourceBundle;

public class Main {

    public static void main(String[] args) {

        Locale currentLocale = Locale.getDefault();
        Locale locale = new Locale("ru", "RU");
        ResourceBundle resourceBundle = ResourceBundle.getBundle("MyBundle", locale);
        System.out.println(resourceBundle.getString("greetings"));

    }
}
```

При этом создается экземпляр класса `ListResourceBundle` или класса `PropertyResourceBundle`.

Метод `getBundle` требует указания локализации, представленной экземпляром класса `Locale`.

Объект `Locale` идентифицирует конкретный язык и страну.

```
Locale(String language)
Locale(String language, String country)
Locale(String language, String country, String variant)

Locale localeFromBuilder = new
Locale.Builder().setLanguage("ru").setRegion("RU").build();

Locale forLangLocale = Locale.forLanguageTag("ru-RU");

Locale localeCosnt = Locale.FRANCE;
```

И может быть создан с помощью одного из конструкторов класса.

Кроме того, с помощью метода `getDefault` класса `Locale` можно получить текущую локализацию по умолчанию для данного экземпляра виртуальной машины Java.

Варианты используются для уточнения локализации.

Они могут дополнительно определять диалекты или языковые вариации конкретного региона.

Также объект `Locale` может быть создан с помощью класса `Locale.Builder`, метода `forLanguageTag` и предустановленных констант класса `Locale`.

Класс `Locale` содержит много методов, которые возвращают информацию о локализации по умолчанию.

```
getDefault - предоставляет объект Locale по умолчанию.  
getAvailableLocales - предоставляет массив доступных локализаций.  
getDisplayCountry - предоставляет название страны.  
getDisplayLanguage - предоставляет имя языка.  
getDisplayVariant - предоставляет код варианта.  
getISO3Country - предоставляет аббревиатуру трех букв для страны текущего региона.  
getISO3Language - предоставляет аббревиатуру трех букв для текущего языка.
```

Интернационализация чисел, валюты, даты и времени



Tech Solutions

Объектно-ориентированное программирование на Java

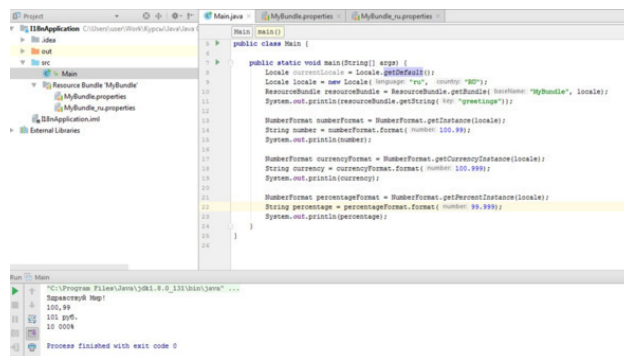
Интернационализация и локализация

Лекция 3

Интернационализация чисел, валюты, даты и времени

Как интернационализировать текст мы разобрались, теперь давайте посмотрим, как интернационализировать и локализовать числа, валюту, дату и время.

Для форматирования чисел, валюты и процентов в соответствии с локализацией может использоваться класс `NumberFormat`.



Примером различия в формате чисел в разных странах является использование «точки» в США и Англии для обозначения десятичной или дробной части и использование запятой в России.

Использование класса `NumberFormat` состоит из двух шагов:

Это применение метода `getInstance`, `getCurrencyInstance`, или `getPercentInstance`, которые возвращают форматирование чисел, валюты и процентов для указанной локализации.

И применение метода `format` для форматирования конкретного числа.

Класс `NumberFormat` также позволяет установить, как минимальное, так и максимальное количество цифр, для целочисленной части и для дробной части числа, установить режим округления числа, а также перевести число из строки.

Класс `DecimalFormat` может быть использован для форматирования чисел с помощью собственного шаблона.

```

pattern = ###,###.###

DecimalFormat decimalFormat = new
DecimalFormat(resourceBundle.getString("pattern"));

String format = decimalFormat.format(123456789.123);

System.out.println(format);

```

Здесь, в этом примере, мы определяем шаблон форматирования в файле свойств.

Затем в коде приложения, передаем этот шаблон с помощью `ResourceBundle` в конструктор класса `DecimalFormat`, и форматируем число.

Для создания пользовательских шаблонов можно использовать специальные символы.

0 - всегда отображается цифра
- цифра
. - десятичный разделитель
, - разделитель групп (например, разделитель тысяч)
E - разделение мантиссы и экспоненты для экспоненциальных форматов.
; - разделит форматы
- - префикс отрицательного числа
% - умножает на 100 и показывает число в процентах
? - умножает на 1000 и показывает число как промилле
\$ - знак валюты. при форматировании использует международные денежные символы.
X - символ, который будет использоваться в префиксе числа или суффиксе
' - цитата вокруг специальных символов в префиксе или суффиксе форматированного числа.

Pattern	Number	Formatted String
###.###	123.456	123.456
###, #	123.456	123.5
###,###.##	123456.789	123,456.79
000.###	9.95	009.95
##0.###	0.95	0.95

На этом слайде показаны специальные символы, их значение и примеры пользовательских шаблонов форматирования.

С помощью класса `DecimalFormatSymbols` вы можете настроить, какие символы использовать в качестве разделителя десятичных чисел, разделителя групп цифр и т. д.

```

DecimalFormatSymbols symbols = new DecimalFormatSymbols();

symbols.setDecimalSeparator('.');
symbols.setGroupingSeparator(',');

DecimalFormat decimalFormat = new DecimalFormat(resourceBundle.getString("pattern"), symbols);

String format = decimalFormat.format(123456789.123);

System.out.println(format);

123;456;789;123

```

В этом примере показано применение разделителя точка с запятой для дробной части числа и двоеточия для разделения групп цифр.

Для интернационализации и локализации даты и времени можно использовать класс `DateFormat`.

```
DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.LONG, locale);
String date = dateFormat.format(new Date());
System.out.println(date);
```

Style	U.S. Locale
DEFAULT	Jun 24, 2016
SHORT	6/24/16
MEDIUM	Jun 24, 2016
LONG	June 24, 2016
FULL	Friday, June 24, 2016

26 ноября 2017 г.

В разных странах существуют разные стандарты форматирования даты.

Можно указывать dd-mm-yyuu, а можно указать mm-dd-yyuu.

Для форматирования даты сначала получается формат даты с учетом локализации с помощью метода `getDateInstance` класса `DateFormat`.

Затем дата форматируется методом `format`.

Формат даты может быть определен по умолчанию, короткий, средний, длинный и полный.

Соответственно будут выведены просто цифры даты или дата с указанием месяца и дня недели.

Аналогично для форматирования времени сначала получается формат времени с учетом локализации с помощью метода `getTimeInstance` класса `DateFormat`.

```
DateFormat timeFormat = DateFormat.getTimeInstance(DateFormat.FULL, locale);
String time = timeFormat.format(new Date());
System.out.println(time);
```

Style	U.S. Locale
DEFAULT	7:03:47 AM
SHORT	7:03 AM
MEDIUM	7:03:47 AM
LONG	7:03:47 AM PDT
FULL	7:03:47 AM PDT

10:48:35 KRAT

Затем время форматируется методом `format`.

Формат времени может быть определен по умолчанию, короткий, средний, длинный и полный.

Соответственно будут выведены просто цифры времени или время с указанием часовой зоны.

С помощью метода `getDateTimeInstance` класса `DateFormat` можно отформатировать с учетом локализации и вывести дату и время одновременно.

```

DateFormat dateTimeFormat =
DateFormat.getDateInstance(DateFormat.DEFAULT, DateFormat.DEFAULT,
locale);

String datetime = dateTimeFormat.format(new Date());

System.out.println(datetime);

```

Также, как и для чисел с классом `DecimalFormat`, для даты и времени, класс `SimpleDateFormat` позволяет форматировать дату и время своим собственным форматом.

```

datepattern="EEEE MMMM yyyy HH:mm:ss.SSSZ

SimpleDateFormat simpleDateFormat = new
SimpleDateFormat(resourceBundle.getString("datepattern"));

date = simpleDateFormat.format(new Date());

System.out.println(date);

```

Для использования `SimpleDateFormat`, мы сначала определяем свой собственный шаблон форматирования, например, в файле свойств.

Затем с помощью `ResourceBundle` передаем этот шаблон форматирования в конструктор класса `SimpleDateFormat`.

И, наконец, форматируем дату и время методом `format`.

Для создания шаблона форматирования даты и времени используются специальные символы.

G - эра (до Рождества Христова).		
год - (например, 12 или 2012 год). Используйте yy или yyyy.		
M - месяц в году. Количество M определяет длину формата (например, MM, MMM или MMMMM).		
d - день в месяце. Количество d определяет длину формата (например, d или dd).		
h - час дня, 1-12 (AM / PM) (обычно hh).		
H - час дня, 0-23 (обычно HH).		
m - минута в часе, 0-59 (обычно mm).		
s - секунд в минуте, 0-59 (обычно ss).		
S - миллисекунд в секунде, 0-999 (обычно SSS).		
E - день в неделе (например, понедельник, вторник и т. д.).		
D - день в году (1-366).		
F - день недели в месяце (например, 1-й четверг декабря).		
w - неделя в году (1-53).		
W - неделя месяца (0-5).		
a - маркер AM / PM.		
k - час в дне (1-24, в отличие от HH 0-23).		
K - час в дне, AM / PM (0-11).		
z - часовой пояс		
' - эскапе для разделителя текста		
^ - одиночная цитата		

Pattern	Example
dd.MM.yy	31.01.12
dd.MM.yyyy	31-01-2012
MM-dd-yyyy	01-31-2012
yyyy-MM-dd	2012-01-31
yyyy-MM-dd HH:mm:ss	2012-01-31 23:59:59
yyyy-MM-dd HH:mm:ss SSS	2012-01-31 23:59:59.999
yyyy-MM-dd HH:mm:ss.SSSZ	2012-01-31 23:59:59.999+0100
EEEE MMMMM yyyy HH:mm:ss.SSSZ	Saturday November 2012 10:45:42.720+0100

На слайде показаны символы для создания шаблона и примеры шаблонов форматирования.

С помощью класса `DateFormatSymbols` можно настроить показ собственных символов даты, используемых для форматирования.

```

DateFormatSymbols dateFormatSymbols = new DateFormatSymbols(locale);
dateFormatSymbols.setWeekdays(new String[]{
    "Unused",
    "Sad Sunday",
    "Manic Monday",
    "Thriving Tuesday",
    "Wet Wednesday",
    "Total Thursday",
    "Fat Friday",
    "Super Saturday",
});

SimpleDateFormat simpleDateFormatSym = new
SimpleDateFormat(resourceBundle.getString("datepattern"), dateFormatSymbols);

date = simpleDateFormatSym.format(new Date());
System.out.println(date);

```

Для этого создается массив символов и передается методу `setWeekdays` класса `DateFormatSymbols`.

Такие же собственные символы можно создать и для обозначения месяца и временной зоны, применяя соответствующие методы класса `DateFormatSymbols`.

Если ваше приложение нужно запускать в разных часовых поясах, код должен отображать одно и тоже время для разных пользователей с учетом их локальной временной зоны.

```

String[] availableIDs = TimeZone.getAvailableIDs();
for(String id : availableIDs) {
    System.out.println("id = " + id);
}

Calendar calendar = new GregorianCalendar();
calendar.setTimeZone(TimeZone.getTimeZone("Europe/Samara"));
calendar.set(Calendar.HOUR_OF_DAY, 9);
System.out.println(calendar.get(Calendar.HOUR_OF_DAY));

calendar.setTimeZone(TimeZone.getDefault());
System.out.println(calendar.get(Calendar.HOUR_OF_DAY));
System.out.println(calendar.getTime());

```

Для этого сначала нужно преобразовать время в UTC (скоординированное универсальное время) перед его сохранением.

Далее время в каждом часовом поясе рассчитывается как смещение к UTC.

Для преобразования между часовыми поясами может использоваться класс `java.util.Calendar`.

Сначала вы создаете экземпляр класса `GregorianCalendar`, подкласс класса `Calendar`.

Далее вы устанавливаете часовой пояс с помощью метода `setTimeZone` класса `Calendar`.

Вы должны всегда устанавливать часовой пояс перед установкой времени.

Вы можете получить список доступных идентификаторов часовых поясов, используя метод `getAvailableIDs` класса `TimeZone`.

Далее вы устанавливаете время с помощью метода `set` класса `Calendar`.

В него вы можете передать поле `HOUR_OF_DAY`, для указания в методе часа дня, и сам час дня.

При этом время автоматически конвертируется с учетом установленной временной зоны.

Если затем изменить часовой пояс методом `setTimeZone`, время также автоматически конвертируется с учетом установленной временной зоны.

Извлечь конвертированное время можно с помощью метода `get` класса `Calendar`.

В этот метод также можно передать поле `HOUR_OF_DAY`, чтобы извлечь только час, или можно получить полную дату-время календаря.

Получить локальную временную зону пользователя можно с помощью метода `getDefault` класса `TimeZone`.

Проверка вводимых данных

При вводе данных пользователем, может возникнуть задача проверки вводимых данных.



Tech Solutions

Объектно-ориентированное программирование на Java

Интернационализация и локализация

Лекция 4

Проверка вводимых данных

```
char ch;
// ...

// check if ch is a letter
if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
// ...

// check if ch is a digit
if (ch >= '0' && ch <= '9')
// ...

// check if ch is a whitespace
if ((ch == ' ') || (ch == '\n') || (ch == '\t'))
// ...

Character.isDigit()
Character.isLetter()
Character.isLetterOrDigit()
Character.isLowerCase()
Character.isUpperCase()
Character.isSpaceChar()
```

Если приложение используется в разных регионах, применение простых операторов сравнения к символам будет некорректным, так как они работают только с английским и несколькими другими языками.

Чтобы интернационализировать операции сравнения символов, нужно использовать методы класса `Character`.

Также может возникнуть проблема при сравнении строк символов, написанных на разных языках.

```
Collator collator = Collator.getInstance(locale);

String[] words={"анпп", "типн", "джен"};

String tmp;
for (int i = 0; i < words.length; i++) {
    for (int j = i + 1; j < words.length; j++) {
        if (collator.compare(words[i], words[j]) > 0) {
            tmp = words[i];
            words[i] = words[j];
            words[j] = tmp;
        }
    }
}
System.out.println(Arrays.toString(words));
```

Каждый язык может иметь свои собственные правила для сортировки строк и букв.

И просто использование метода `compareTo` класса `String` может не работать для всех языков.

Чтобы отсортировать коллекцию строк в соответствии с правилами определенного языка, можно использовать экземпляр класса `Collator`, созданный с учетом локализации.

Метод `compare` класса `Collator` может использоваться для сравнения строк.

Метод возвращает -1, 0 или 1.

«-1» означает, что первая строка встречается раньше второй строки.

«0» означает, что строки имеют одинаковый порядок, а «1» означает, что первая строка появляется позже второй строки.

Метод `getInstance` класса `Collator` возвращает экземпляр класса `RuleBasedCollator`, который является подклассом класса `Collator`.

```
String rules = "< a < b < a";
try {
    RuleBasedCollator collatorR = new RuleBasedCollator(rules);

    String[] wordsR={"анпп","бипп","вкен"};

    String tmpR;
    for (int i = 0; i < wordsR.length; i++) {
        for (int j = i + 1; j < wordsR.length; j++) {
            if (collatorR.compare(wordsR[i], wordsR[j]) > 0) {
                tmpR = wordsR[i];
                wordsR[i] = wordsR[j];
                wordsR[j] = tmpR;
            }
        }
    }
    System.out.println(Arrays.toString(wordsR));
} catch (ParseException e) {
    e.printStackTrace();
}
```

Класс `RuleBasedCollator` содержит набор правил, которые определяют порядок сортировки строк для указанной вами локализации.

Эти правила предопределены для каждой локализации.

И поэтому, если предопределенные правила сортировки в классе `Collator` не соответствуют вашим потребностям, вы можете определить свои собственные правила и назначить их объекту `RuleBasedCollator`.

Для этого нужно определить правила, которые будут использоваться для сравнения символов в строковом объекте.

Передать строку в конструктор класса `RuleBasedCollator`.

И использовать метод `compare` для сравнения строк.

Остальные символы, не определенные вашим правилом, сортируются с использованием порядка по умолчанию экземпляра `RuleBasedCollator`.

Вы можете группировать символы, разделяя их запятой в строке правила.

```
String rules = "< c,C < b,B";
```

```
String rules = "< ch < b < a < c";
```

Или вы можете указать, что некоторые комбинации символов должны интерпретироваться как один символ при сравнении строк.

Если вам нужно сортировать и использовать одни и те же строки несколько раз, вы можете создать `CollationKey` для каждой строки и сортировать на основе этого ключа вместо использования строк.

```
String rulesK = "< c,C < b,B < a,A";

RuleBasedCollator ruleBasedCollator = null;
try {
    ruleBasedCollator = new RuleBasedCollator(rulesK);
    CollationKey[] collationKeys = new CollationKey[3];

    collationKeys[0] = ruleBasedCollator.getCollationKey("anpp");
    collationKeys[1] = ruleBasedCollator.getCollationKey("canp");
    collationKeys[2] = ruleBasedCollator.getCollationKey("босс");

    Arrays.sort(collationKeys);

    for(CollationKey collationKey : collationKeys) {
        System.out.println(collationKey.getSourceString());
    }
} catch (ParseException e) {
    e.printStackTrace();
}
```

Сортировка на основе `CollationKey` выполняется с использованием побитового сравнения.

Это существенно ускоряет сортировку, по сравнению с сортировкой строк, которую обычно использует класс `RuleBasedCollator`.

В случае использования `CollationKey` просто сортируется обычный массив ключей.

Однако создание `CollationKey` требует времени.

Если вы сортируете строки только один раз, быстрее просто использовать класс `RuleBasedCollator`.

В кодировке Unicode некоторые символы могут быть представлены несколькими способами.

```
System.out.println("Ä".equals("\u00C1"));
System.out.println("A".equals("\u0041"));
System.out.println(Normalizer.normalize("Ä", Normalizer.Form.NFD).equals("\u0041\u0301"));
System.out.println("Ä".equals("\u0041\u0301"));
System.out.println("Ä\u0041\u0301");

true
true
true
false
ÄÄ
```

Original word	NFC	NFD	NFKC	NFKD
"schön"	"schön"	"schö\u0308n"	"schön"	"schö\u0308n"

У некоторых из них есть свой собственный символ, а также комбинация других символов Юникода, которые могут их представлять.

Когда символы могут быть представлены несколькими способами, сортировка их становится сложнее.

Поэтому вы должны нормализовать текст, прежде чем сортировать его, или осуществлять в нем поиск.

Нормализация текста гарантирует, что заданная строка символов юникод всегда представляется одинаково.

Вы можете нормализовать строку, используя статический метод `normalize` класса `Normalizer`.

Первым параметром метода `normalize` является текст.

Второй параметр – это форма нормализации для нормализации текста.

Нормализатор разлагает исходные символы на комбинацию базового символа и диакритического знака (это может быть несколько знаков на разных языках).

Часто в приложении требуется разобрать текст на символы, слова, предложения или строки.

```
BreakIterator breakIterator = BreakIterator.getWordInstance(locale);

String str = "Юникод стандарт кодирования символов, позволяющий представить знаки
почти всех письменных языков ";

breakIterator.setText(str);

int boundaryIndex = breakIterator.first();
while(boundaryIndex != BreakIterator.DONE) {
    int itmp = boundaryIndex;
    boundaryIndex = breakIterator.next();
    String stmp = str.substring(itmp, boundaryIndex);
    System.out.println(stmp);
}
```

- `getCharacterInstance`
- `getWordInstance`
- `getSentenceInstance`
- `getLineInstance`

Это является задачей определения границ символов, слов, предложений и строк, которые могут подчиняться различным правилам в различных языках.

Например, при поиске границ символов необходимо провести различие между символами пользователя и символами юникода.

Пользовательский символ – это символ, который пользователь пишет и который обычно отображается на экране.

Однако для представления символа пользователя может потребоваться один или несколько символов юникода.

Поэтому задача определения границ символов пользователя, это не тоже самое как определение границ символов юникода.

Java предоставляет класс `BreakIterator`, который упрощает поиск границ на разных языках.

Методы класса позволяют создать экземпляр класса `BreakIterator` для указанной локализации, который содержит воображаемый курсор для текущей границы в строке текста, и этот курсор можно перемещать с помощью методов `previous` и `next`.

Начальная граница будет «0», а последней границей будет длина строки текста.

В этом примере мы ищем границы слов в строке текста и с помощью курсора, разбиваем текст на отдельные слова.

Общая схема использования `BreakIterator` следующая – создается экземпляр для поиска границ определенного типа, символов, слов, предложений или строк.

Далее для этого экземпляра устанавливается текст.

Затем метод `first` извлекает первую границу, и метод `next` находит все остальные границы, пока не будет возвращена константа `BreakIterator.DONE`.

Внутренне, в Java, все символы хранятся как 16 битовые или 2 байтовые в кодировке UTF-16.

```
byte[] bytes = new byte[10]; // array of bytes (0xF0, 0x9F, 0x98, 0x81)
String strNew = new String(bytes, Charset.forName("UTF-8"));
System.out.println(strNew);

String str1 = new String("hello");
byte[] strArr = str1.getBytes("UTF-8");
System.out.println(Arrays.toString(strArr));

Reader reader = new InputStreamReader(inputStream, Charset.forName("UTF-8"));
Writer writer = new OutputStreamWriter(outputStream, Charset.forName("UTF-8"));
```



Символы – это графическое сущность, которая является частью человеческой культуры. Когда компьютер должен обрабатывать текст, он использует представление этих символов в байтах.

Используемое точное представление называется кодировкой.

Так как не любой текст, получаемый от пользователей или внешних источников, находится в этой кодировке, вашему приложению, возможно, придется конвертировать текст из не-UTF-16 кодировки в UTF-16 кодировку.

Или, когда приложение выводит текст, возможно, придется преобразовать внутреннюю кодировку UTF-16 в любую кодировку, которая нужна внешним источникам.

В Java есть несколько различных способов, с помощью которых вы можете преобразовать текст в и из UTF-16 кодировки.

Для этого вы можете использовать класс String, а также классы Reader и Writer и их под-классы.

Класс String может использоваться для преобразования массива байтов в экземпляр String.

Массив байтов и конкретная байтовая кодировка используются для преобразования в качестве параметров для конструктора класса String при создания новой строки.

Затем конструктор String преобразует байты из набора символов массива байтов в UTF-16 кодировку.

Этот конструктор String принимает последовательность байтов, которая должна быть в кодировке, которую вы указали во втором аргументе, и преобразует их в представление UTF-16 символов, которые эти байты представляют в этой кодировке.

Строка также может быть преобразована в другой формат с использованием метода getBytes класса String.

Здесь мы получим массив байт символов в указанной кодировке.

Также, классы чтения и записи пакета java.io позволяют Java-приложению конвертировать между символьными потоками в кодировке UTF-16 и потоком байтов текста, в кодировке отличной от UTF-16.

Класс InputStreamReader преобразует из определенного набора символов в кодировке, например, UTF-8 в кодировку UTF-16.

OutputStreamWriter может переводить символы UTF-16 в символы кодировки, отличной от UTF-16.

Основы сетевого взаимодействия



Объектно-ориентированное программирование на Java

Введение в сетевое взаимодействие

Лекция 1

Основы сетевого взаимодействия

Общение и обмен информацией, является фундаментальной чертой современной жизни. Мы развиваемся путем обмена идеями.

На заре компьютерной эры, когда мощность процессоров была недостаточной, появилась концепция распределенных вычислений.

Не для того, чтобы делиться информацией, а просто чтобы ускорить вычисление.

С тех пор в компьютерной науке появилась область исследований распределенных вычислений.

Как должна передаваться информация? По какому протоколу?

Как обеспечить при этом безопасность?

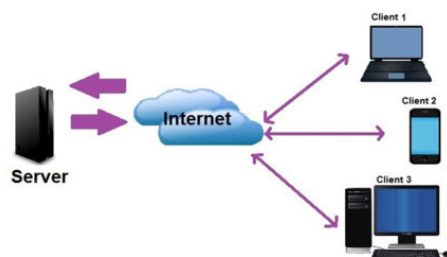
Как обеспечить пропускную способность и скорость канала?

Какие функции участников на разных концах канала?

Эти вопросы сформировали область распределенных вычислений как сложную и неотъемлемую часть информационных технологий.

Существуют два основных способа передачи информации – это система клиент/сервер и система одноранговой сети.

Клиент – это пользователь сетевых сервисов.



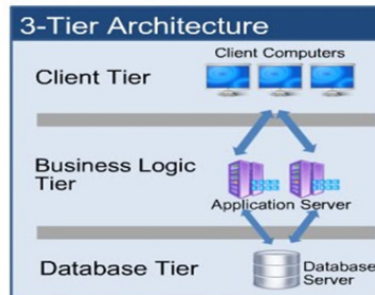
Сервер – это поставщик сетевых услуг.

Как многие из вас знают, идея системы клиент/сервер заключается в том, чтобы легкие клиенты – программы с минимальным кодом и потреблением ресурсов, работающие на пользовательских компьютерах, обменивались данными с мощными серверами.

Клиентские программы, когда они запускаются, подключаются к серверам, проходят аутентификацию, и запрашивают некоторые данные.

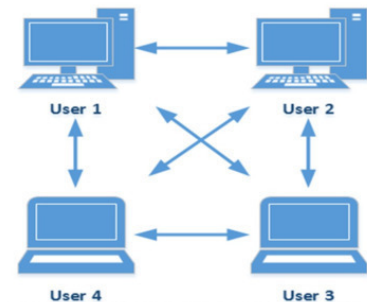
Роль сервера заключается в хранении большого количества информации, позволяя совместно использовать эту информацию, и когда это необходимо, выполнять значительные вычисления.

Как правило, работает многоуровневая клиент-серверная архитектура.



В которой хранение данных отделено от их обработки и выдачи конечного результата пользователю.

Другой тип распределенной модели – это одноранговая сеть.



В отличие от модели клиент/сервер, эта модель предполагает, что участвующие компьютеры имеют равную мощность и ресурсы.

Одноранговая сеть организована в граф.

Нет строгих правил о том, как должны подключаться компьютеры, за исключением того, что любой компьютер должен быть подключен, по крайней мере, к двум другим компьютерам.

В этой модели нет централизованного контроллера.

Когда узел хочет получить определенную информацию, он запрашивает своих пиров, инициируя рекурсивный обход графика.

В такой сети, информация хранится равномерно среди участвующих машин.

Но независимо от того, как вы решили организовать сеть, следующий вопрос, который вам нужно решить, это как ваши машины будут общаться друг с другом, с помощью какого протокола.

Существует множество различных платформ и протоколов.

Когда мы говорим о сетевых протоколах, нам нужно тщательно различать уровни коммуникации, которые мы имеем в виду.

Давайте используем упрощенное 4-уровневое представление сетевой коммуникации.



Уровень Интернета, или протокол Internet Protocol (IP), определяет, как байты должны быть организованы в пакеты, и определяет схему адресации, с помощью которой компьютеры в сети находят друг друга.

В пакетах есть заголовки, которые описывают размер содержимого пакета, его получателя и отправителя.

Транспортный уровень связан с повторной доставкой данных в случае, если некоторые из них теряются.

Он также определяет порядок, в котором получатель принимает пакеты.

Двумя основными протоколами здесь являются TCP и UDP.

В протоколе Transmission Control Protocol (TCP) пакеты доставляются по порядку, и утерянные пакеты отправляются заново.

Это один из основных протоколов передачи данных интернета, предназначенный для управления передачей данных.

Сети и подсети, в которых совместно используются протоколы TCP и IP, называются сетями TCP/IP.

Реализации протокола TCP обычно встроены в ядра ОС.

В протоколе UDP (User Datagram Protocol) пакеты доставляются в произвольном порядке и утерянные пакеты теряются.

Этот протокол не такой надежный как TCP, но значительно более быстрый.

Теперь, на прикладном уровне, вы можете общаться любым способом.

Например, веб-браузеры и веб-серверы обмениваются данными через протокол HyperText Transfer Protocol HTTP.

Java предоставляет широкий спектр возможностей для распределенных вычислений, которые варьируются от поддержки сокетов до Enterprise Java Beans.

Сокет – это абстрактный объект, представляющий конечную точку соединения для взаимодействия с удаленным процессом.

Enterprise Java Beans – серверные компоненты, реализующие бизнес-логику.

Когда вы пишете Java-программы, которые обмениваются данными по сети, вы программируете на уровне приложения.

Как правило, вам не нужно беспокоиться об уровнях TCP и UDP.

Вместо этого вы можете использовать классы пакета java.net.

Эти классы обеспечивают платформ независимое сетевое взаимодействие.

Однако, чтобы решить, какие классы Java должны использовать ваши программы, вы должны понимать, чем отличаются TCP и UDP.

Когда два приложения надежно связываются друг с другом, они устанавливают соединение и отправляют данные по этому соединению.

Протокол TCP гарантирует, что данные, отправленные с одного конца соединения, попадут на другой конец и в том же порядке, в котором данные были отправлены. В противном случае сообщается об ошибке.

TCP обеспечивает канал «точка-точка» для приложений, требующих надежной связи.

Протокол передачи гипертекста (HTTP), протокол передачи файлов (FTP) и Telnet – все это примеры приложений, требующих надежного канала связи.

Порядок, в котором данные отправляются и принимаются по сети, имеет решающее значение для работы этих приложений.

Когда HTTP используется для чтения данных с URL-адреса, данные должны быть получены в том порядке, в котором они были отправлены. В противном случае вы получите поврежденный файл.

Протокол UDP обеспечивает связь, которая не является надежной между двумя приложениями в сети.

Протокол UDP не основан на соединениях, как протокол TCP.

Скорее, UDP отправляет независимые пакеты данных, называемые дейтаграммами, из одного приложения в другое.

Отправка дейтаграмм очень похожа на отправку письма через почтовую службу – порядок доставки не важен и не гарантируется, и каждое сообщение не зависит от другого.

Протокол UDP является более быстрым по сравнению с протоколом TCP.

И для некоторых приложений скорость более важна, чем надежность.

Рассмотрим, например, сервер синхронизации, который отправляет текущее время своему клиенту, когда он этого требует.

Если клиент пропускает пакет, на самом деле не имеет смысла повторно отправлять его, потому что время будет неверным, когда клиент получит его при повторной попытке.

В этом случае надежность TCP не нужна, скорее нужна скорость получения ответа.

Реальный компьютер имеет только одно физическое соединение с сетью.

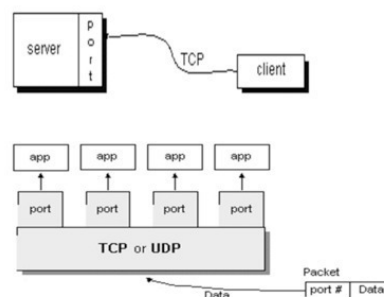
Все данные, предназначенные для конкретного компьютера, поступают через это соединение.

Однако данные могут быть предназначены для разных приложений, запущенных на этом компьютере.

Поэтому, как компьютер знает, какому приложению пересылать данные?

Он делает это с помощью использования портов.

Данные, передаваемые через Интернет, сопровождаются адресацией, которая указывает компьютер и порт, для которых эти данные предназначены.



Компьютер идентифицируется его 32-битным IP-адресом, который используется протоколом IP для доставки данных на нужный компьютер в сети.

Порт идентифицируется 16-разрядным номером, который используется протоколом TCP и UDP для доставки данных в нужное приложение.

В протоколе на основе соединения, таком как TCP, серверное приложение связывает сокет с определенным номером порта.

Это приводит к регистрации сервера в системе для получения всех данных, предназначенных для этого порта.

Затем клиент может соединиться с сервером на этом порту.

В протоколе на основе датаграмм, такой как UDP, пакет датаграммы содержит номер порта своего адресата, и UDP направляет пакет в соответствующее приложение.

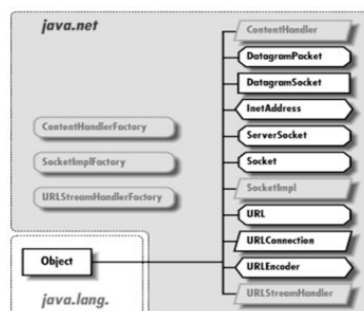
Номера портов варьируются от 0 до 65535, потому что порты представлены 16-разрядными номерами.

Номера портов в диапазоне от 0 до 1023 зарезервированы для использования системными службами, такими как HTTP и FTP.

И приложения не должны пытаться привязываться к ним.

Программы Java могут использовать TCP или UDP для связи через Интернет с помощью классов пакета `java.net`.

Классы `URL`, `URLConnection`, `Socket` и `ServerSocket` используют протокол TCP для обмена данными по сети.



Классы `DatagramPacket`, `DatagramSocket` и `MulticastSocket` предназначены для использования протокола UDP.

Сокеты

В Java вы можете думать о сокете как о двухстороннем туннеле данных.



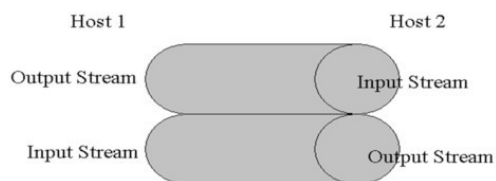
Объектно-ориентированное программирование на Java

Введение в сетевое взаимодействие

Лекция 2

Сокеты

Sockets in Java



Этот туннель имеет входные и выходные потоки.

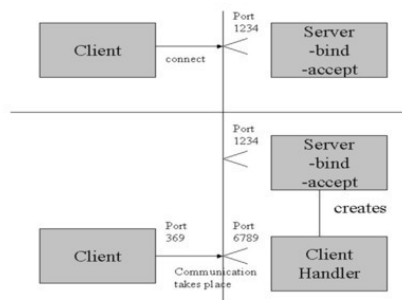
Сокет создается с сетевым адресом и портом.

Вы отправляете данные, просто записывая их в выходной поток сокета и получая данные, считывая из входного потока сокета.

Класс `java.net.Socket` обеспечивает функциональность клиентского сокета.

Однако для того, чтобы вы могли подключиться к удаленному хосту, вам нужно иметь серверный сокет на другом конце.

Серверный сокет связан с портом на хосте, где работает программа, и блокируется, пока кто-то не попытается к нему подключиться.



Как только это произойдет, он просыпается, создает новый порт, где будет происходить соединение, создает новый клиентский сокет для поддержки этой связи, а затем снова блокируется.

Таким образом, обычно сервер работает на определенном компьютере и имеет сокет, привязанный к определенному номеру порта.

Сервер просто ждет, слушая сокет для клиентского запроса на соединение.

При этом клиент знает имя хоста машины, на которой работает сервер, и номер порта, на котором сервер прослушивает.

Клиенту также необходимо идентифицировать себя серверу, чтобы привязаться к локальному номеру порта, который будет использоваться во время этого соединения.

После запроса клиента, сервер принимает соединение, и сервер получает новый сокет, который привязан к локальному порту, и который имеет удаленную конечную точку, установленную на адресе и порту клиента.

Сервер создает новый клиентский сокет, чтобы серверный сокет мог продолжать слушать клиентские запросы на соединение.

На стороне клиента, если соединение принято, сокет на стороне сервера успешно создан, тогда клиент может использовать сокет для связи с сервером.

Клиент и сервер могут теперь общаться, записывая или считывая данные с помощью сокетов.

Таким образом, сокет – это одна конечная точка двусторонней линии связи между двумя программами, запущенными в сети.

Сокет привязан к номеру порта, так протокол TCP может идентифицировать приложение, для которого предназначаются данные.

На самом деле, эта конечная точка представляет собой комбинацию IP-адреса и номера порта.

И каждое TCP-соединение может быть однозначно идентифицировано двумя его конечными точками.

Таким образом, вы можете иметь сразу несколько соединений между вашим хостом и сервером.

Пакет `java.net` предоставляет класс `Socket`, с помощью которого можно реализовать клиентскую сторону двухстороннего соединения между вашей Java-программой и другой программой в сети.

Класс `Socket` скрывает детали конкретной низлежащей платформы от вашей Java-программы, так что Java-программы могут общаться по сети не зависящим от платформы способом.

Кроме того, пакет `java.net` предоставляет класс `ServerSocket`, с помощью которого можно реализовать серверную сторону двухстороннего соединения.

Серверы могут использовать этот сокет для прослушивания и принятия подключений клиентов.

Если подключаться к Интернету, используя URL адрес веб ресурса, тогда класс `URL` и связанные с ним классы `URLConnection`, `URLEncoder` пакета `java.net` подойдут лучше, чем классы сокетов.

Фактически, URL-адреса являются более высокоуровневым подключением к Интернету и используют сокеты как часть базовой реализации.

Таким образом, мы рассматриваем систему из распределенных компьютеров, которые подключены к какой-либо сети.

А для среды Java весь код работает на виртуальной машине Java.

И каждая виртуальная машина Java является отдельным процессом.

И мы можем наладить связь между двумя конкретными JVM с помощью сокетов.

И сокет – это конечная точка.

Так что одна JVM играет роль клиента, поэтому у нас есть клиентский сокет, а другая JVM играет роль сервера, поэтому у нас есть серверный сокет.

Что именно нужно сделать для этих двух JVM, чтобы они говорили друг с другом?

Для этого, у Java есть API-интерфейсы, чтобы абстрагировать поведение системы на низком уровне.

На стороне сервера нужно создать `ServerSocket`.

И это инициализирует конечную точку на JVM сервера, связав ее с портом компьютера.

При этом процесс откроет этот порт как сокет для связи с потенциальными клиентами.

Далее вам нужно будет слушать и ждать, чтобы увидеть, хотят ли клиенты подключиться к сокету, который вы открыли.

И сервер может использовать входной поток для чтения данных от клиента.

И выходной поток для записи данных клиенту.

Вы можете общаться на уровне сокета через входные и выходные потоки.

Теперь, что нужно делать клиенту?

Клиент должен как-то подключиться к серверу.

Для этого он создает свой клиентский сокет.

И снова JVM клиента может использовать входной поток для чтения и выходной поток для записи данных.

После того, как вы все настроите, JVM могут разговаривать друг с другом, потому что они могут использовать входные и выходные потоки сокетов для чтения и записи данных.

Серверный сокет

Для создания простого однопоточного Java сервера, создадим класс, в котором определим метод с бесконечным циклом.



Объектно-ориентированное программирование на Java

Введение в сетевое взаимодействие

Лекция 3

Серверный сокет

```
public class NetworkServer {
    private int port;

    public NetworkServer(int port) {
        this.port = port;
    }

    public void listen() {
        try(ServerSocket listener = new ServerSocket(port)) {
            Socket socket;
            while(true) { // Run until killed
                socket = listener.accept();
                System.out.println("Ouyyeno coeyenne oy: " + socket.getInetAddress()+" "+socket.getPort());
                handleConnection(socket);
            }
        } catch (IOException ioe) {
            System.out.println("IOException: " + ioe);
            ioe.printStackTrace();
        }
    }

    protected void handleConnection(Socket socket) throws IOException{
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        String line = in.readLine();
        System.out.println("Ouyyeno coeyeno: " + line);
        out.println("Oy: " + socket.getLocalAddress()+" "+socket.getLocalPort()+" Ouyy: " + line);
        socket.close();
    }

    public static void main(String[] args) {
        int port = 8080;
        NetworkServer server = new NetworkServer(port);
        server.listen();
    }
}
```

В этом методе мы создадим объект `ServerSocket`.

Класс `ServerSocket` реализует интерфейс `AutoCloseable`, поэтому мы используем конструкцию `try-with-resources`, которая обеспечивает автоматический вызов метода `close` объекта `ServerSocket` при выходе из блока `try-with-resources`.

Метод `close` закрывает канал сокета.

Здесь мы создаем сокет сервера и связываем его с указанным номером локального порта.

Затем мы получаем клиентский сокет с помощью метода `accept` серверного сокета.

Метод `accept` слушает подключение к серверному сокету и принимает это соединение.

Метод блокируется до тех пор, пока не будет выполнено соединение.

При этом создается новый объект `Socket`.

Далее мы во внутреннем методе `handleConnection` создаем входящий и исходящий потоки клиентского сокета.

И читаем из входящего потока и пишем в исходящий поток.

После этого закрываем клиентский сокет и его потоки.

В методе `main` приложения мы создаем объект сервера и вызываем его метод с бесконечным циклом.

Созданный нами сервер может обрабатывать только один клиентский запрос за раз.

```

public void listen() {
    int poolSize = 50 * Runtime.getRuntime().availableProcessors();
    ExecutorService tasks = Executors.newFixedThreadPool(poolSize);
    try (ServerSocket listener = new ServerSocket(port)) {
        Socket socket;
        while (true) { // Run until killed
            socket = listener.accept();
            tasks.execute(new ConnectionHandler(socket));
        }
    } catch (IOException ioe) {
        System.err.println("IOException: " + ioe);
        ioe.printStackTrace();
    }
}

private class ConnectionHandler implements Runnable {
    private Socket connection;
    public ConnectionHandler(Socket socket) {
        this.connection = socket;
    }
    public void run() {
        try {
            handleConnection(connection);
        } catch (IOException ioe) {
            System.err.println("IOException: " + ioe);
        }
    }
}

```

Модифицируем этот сервер, что он мог обрабатывать сразу несколько клиентских запросов одновременно.

Изменим метод с бесконечным циклом, в котором создадим менеджера асинхронных задач `ExecutorService` с пулом потоков, который повторно использует фиксированное количество потоков, работающих с общей неограниченной очередью.

Здесь при каждом клиентском запросе создается своя задача `ConnectionHandler`, которая выполняется в отдельном потоке.

Таким образом, этот сервер может параллельно обрабатывать несколько клиентских запросов.

Для демонстрации работы этого сервера запустим метод `main` приложения и в браузере наберем адрес `http://localhost:8080`, таким образом попытавшись соединиться с сервером.

```

protected void handleConnection(Socket socket) throws IOException {
    PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    String line = in.readLine();
    System.out.println("Получена строка: " + line);
    // out.println("От: " + socket.getLocalAddress() + ":" + socket.getLocalPort() + " Отправ: " + line);
    out.println(
        "HTTP/1.1 200 OK\r\n" +
        "Content-Type: text/html\r\n" +
        "\r\n" +
        "<!DOCTYPE html>\r\n" +
        "<html lang='en'>\r\n" +
        "<head>\r\n" +
        "  meta charset='utf-8'\r\n" +
        "  <title> 'От: " + socket.getLocalAddress() + ":" + socket.getLocalPort() + "</title>\r\n" +
        "</head>\r\n" +
        "<body>\r\n" +
        "  <div align='center'>\r\n" +
        "    <div align='center'>> 'От: " + socket.getLocalAddress() + ":" + socket.getLocalPort() + "</div>\r\n" +
        "  </div>\r\n" +
        "</body></html>\r\n");
    socket.close();
}

```

От: /0:0:0:0:0:0:1:8080

GET /HTTP/1.1>

В нашем методе `handleConnection` мы отправляем клиенту HTML код как ответ на запрос.

Поэтому в браузере отобразится страница с указанием адреса и порта сокета, созданного сервером.

Клиентский сокет

Для создания Java клиента сервера, мы создадим класс, в котором определим метод создания объекта Socket.



Объектно-ориентированное программирование на Java

Введение в сетевое взаимодействие

Лекция 4

Клиентский сокет

```
public class NetworkClient {
    private String host;
    private int port;

    public NetworkClient(String host, int port) {
        this.host = host;
        this.port = port;
    }

    public void connect() {
        try(Socket client = new Socket(host, port)) {
            handleConnection(client);
        } catch (UnknownHostException uhe) {
            System.err.println("Unknown host: " + host);
        } catch (IOException ioe) {
            System.err.println("IOException: " + ioe);
        }
    }

    protected void handleConnection(Socket client) throws IOException {
        PrintWriter out = new PrintWriter(client.getOutputStream(), true);
        BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));
        out.println("Network client");
        String line = in.readLine();
        System.out.println("Or: " + this.host + " Получена строка: " + line);
    }

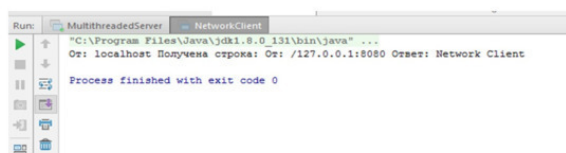
    public static void main(String[] args) {
        String host = "localhost";
        int port = 8080;
        NetworkClient client = new NetworkClient(host, port);
        client.connect();
    }
}
```

Объект Socket клиента создается на основе хоста и порта сервера.

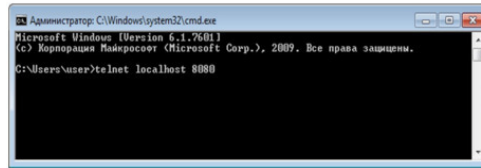
Здесь мы также используем конструкцию try-with-resources для автоматического закрытия сокета.

В методе handleConnection мы создаем исходящий и входящий потоки сокета, пишем в исходящий поток и получаем ответ от сервера во входящем потоке.

Для демонстрации работы этой системы клиент-сервер сначала запустим метод main сервера, а затем запустим метод main клиента и получим ответ от сервера.



Telnet- инструмент командной строки, предназначенный для управления удаленными серверами через командную строку.



Для активации этого инструмента, откройте панель управления.
Откройте «Программы и Компоненты».
Нажмите «Включение или отключение компонентов Windows».
Найдите «Клиент Telnet» и включите его.
Далее в командной строке набираем подключение к хосту и порту.
Подключаемся и набираем строку, которую отправляем на сервер.

Использование URL

Класс URL и связанные с ним классы пакета java.net позволяют использовать протокол HTTP для соединения с веб ресурсом по его URL адресу в Интернете.



Объектно-ориентированное программирование на Java

Введение в сетевое взаимодействие

Лекция 5

Использование URL

<https://www.google.ru>

```
protected void handleConnectionURL(Socket socket) throws IOException {
    PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush: true);
    BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    String urlStr = in.readLine();
    System.out.println("Got url: " + urlStr);
    URL url = new URL(urlStr);
    BufferedReader inURL = new BufferedReader(
        new InputStreamReader(url.openStream()));

    String inputLine;
    while ((inputLine = inURL.readLine()) != null)
        out.println(inputLine);
    in.close();

    socket.close();
}
```

Рассмотрим пример, в котором созданный нами сервер, используется в качестве прокси веб сервера для соединения клиента с веб ресурсом Интернета по URL адресу.

Вы уже наверняка слышали термин URL и использовали URL-адреса для доступа к HTML-страницам из Интернета.

Проще, хотя и не совсем точно, думать о URL-адресе как об имени файла во всемирной паутине, так как большинство URL-адресов соотносятся с файлом на какой-либо машине в сети.

Однако URL-адреса также могут указывать на другие ресурсы в сети, такие как запросы к базе данных и командам.

URL является аббревиатурой Uniform Resource Locator и является ссылкой (адресом) на ресурс в Интернете.

URL имеет два основных компонента:

Идентификатор протокола, например, http.

Имя ресурса, например, google.ru.

Обратите внимание, что идентификатор протокола и имя ресурса разделяются двоеточием и двумя косыми чертами.

Идентификатор протокола указывает имя протокола, который будет использоваться для извлечения ресурса.

В этом примере используется протокол передачи гипертекста (HTTP), который обычно используется для обработки гипертекстовых документов.

HTTP – это один из многих протоколов, используемых для доступа к различным типам ресурсов в сети.

Есть другие протоколы – это File Transfer Protocol, Network News Transfer Protocol и так далее.

Имя ресурса – это полный адрес ресурса.

Формат имени ресурса полностью зависит от используемого протокола, но для многих протоколов, включая HTTP, имя ресурса содержит один или несколько из следующих компонентов – это имя хоста, путь к файлу, номер порта для подключения (обычно необязательный) и ссылка на якорь в ресурсе, который обычно идентифицирует конкретное местоположение в файле (обычно необязательный).

В Java самый простой способ создать объект URL – это использовать конструктор класса URL и строку, представляющую URL-адрес.

Конструкторы класса URL также позволяют использовать абсолютные и относительные URL адреса ресурсов, а также указывать отдельно протокол, хост, порт и путь к файлу.

Вы можете еще сталкиваться с URI Uniform Resource Identifier.

URI – это идентификатор, а URL – это местоположение, так что URL – это всегда URI, но не наоборот.

URI не может указать местоположение ресурса, а только его идентифицирует.

После того, как вы создали URL-объект, вы можете вызвать метод `openStream` для получения потока, из которого вы можете прочитать содержимое URL-адреса.

Метод `openStream` возвращает объект `InputStream`, поэтому чтение с URL-адреса – это чтение из входного потока.

В клиенте сервера мы записываем URL адрес в исходящий поток сокета и считываем веб ресурс из входящего потока сокета.

```
public void connect() {
    try(Socket client = new Socket(host, port)) {
        handleConnection(client);
    } catch (UnknownHostException uhe) {
        System.err.println("Unknown host: " + host);
    } catch (IOException ioe) {
        System.err.println("IOException: " + ioe);
    }
}

protected void handleConnection(Socket client) throws IOException {
    PrintWriter out = new PrintWriter(client.getOutputStream(), true);
    BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));
    out.println(host);
    // String line = in.readLine();
    // System.out.println("Ver: " + this.host + " Saysame response: " + line);
    in.lines().forEach(System.out::println);
}

public static void main(String[] args) {
    String host = "localhost";
    int port = 8080;
    String url = "https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html";
    NetworkClient client = new NetworkClient(host, port, url);
    client.connect();
}
```

Вместо метода `openStream` класса URL, вы можете использовать класс `URLConnection` или один из его подклассов, например, `HttpURLConnection`, метод `connect` которого позволяет установить соединение с веб ресурсом по URL адресу.

```
protected void handleConnectionURL(Socket socket) throws IOException {
    PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush true);
    BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    String urlStr = in.readLine();
    System.out.println("Donyan url: " + urlStr);
    URL url = new URL(urlStr);
    // BufferedReader inURL = new BufferedReader(new InputStreamReader(url.openStream()));

    URLConnection urlConnection = url.openConnection();
    urlConnection.connect();

    BufferedReader inURL = new BufferedReader(new InputStreamReader(
        urlConnection.getInputStream()));

    String inputLine;
    while ((inputLine = inURL.readLine()) != null)
        out.println(inputLine);
    in.close();

    socket.close();
}
```

Объект `URLConnection` получается из объекта `URL` методом `openConnection`.

Вы можете использовать этот объект `URLConnection` для настройки параметров запроса для подключения к веб ресурсу.

После подключения к URL-адресу, вы можете использовать объект `URLConnection` для выполнения таких действий, как чтение или запись в соединение.

На самом деле, соединение по URL адресу открывается неявно, при вызове метода `getInputStream` объекта `URLConnection`.

Поэтому вызывать метод `connect` в этом случае не обязательно.

Объект `URLConnection` в основном используется для HTTP POST запросов к веб ресурсу для записи данных на сервер.

```
String url = "http://example.com";
String charset = "UTF-8";
String param1 = "value1";
String param2 = "value2";
String query = String.format("param1=%s&param2=%s",
    URLEncoder.encode(param1, charset),
    URLEncoder.encode(param2, charset));

URLConnection connection = new URL(url).openConnection();
connection.connect();
connection.setDoOutput(true); // Triggers POST.
connection.setRequestProperty("Accept-Charset", charset);
connection.setRequestProperty("Content-Type",
    "application/x-www-form-urlencoded; charset=" + charset);
try (OutputStream output = connection.getOutputStream()) {
    output.write(query.getBytes(charset));
}
```

Здесь показан пример отправки POST запроса с параметрами на сервер.

Параметры запроса должны быть в формате `name=value` и должны быть объединены `&` амперсандом.

Также используется класс `URLEncoder` для кодирования строки в URL формат с указанной кодировкой.

После создания строки параметров, метод `setDoOutput (true)` неявно устанавливает метод запроса POST для соединения.

Стандартный HTTP POST запрос в виде веб-форм имеет тип `application/x-www-form-urlencoded`, в котором строка запроса записывается в тело запроса.

Вы переводите строку запроса в байты с указанной кодировкой и записываете их в исходящий поток.

Вы можете явно запустить HTTP-запрос с помощью метода `connect` класса `URLConnection`, но запрос будет автоматически запущен по требованию, когда вы захотите получить любую информацию об ответе с использованием `URLConnection.getInputStream`.

Обмен Java объектами



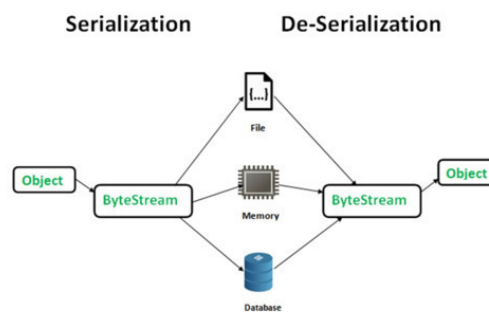
Объектно-ориентированное программирование на Java

Введение в сетевое взаимодействие

Лекция 6

Обмен Java объектами

Теперь, когда вы узнали о распределенном программировании с использованием сокетов, давайте рассмотрим очень важную концепцию, применимую к распределенному программированию, которая называется сериализацией и десериализацией.



Теперь мы знаем, как две JVM общаются друг с другом с помощью сокетов.

И мы увидели, что для обмена сообщениями используются входной и выходной потоки.

Но потоки ввода и вывода работают на уровне последовательностей байтов.

Клиентская и серверная JVM, каждая может иметь набор своих объектов.

И предположим, что мы хотим общаться между этими двумя машинами с помощью объектов.

А это означает, что нам нужно найти способ преобразования объектов в байты, когда мы хотим отправить объект из одной JVM в другую.

Преобразование объекта в байты называется сериализацией объекта.

И когда мы получаем последовательность байтов, нам нужно восстановить копию объекта в другой JVM, и это называется десериализацией.

Теперь, как мы можем это сделать?

Есть несколько вариантов.

Во-первых, мы могли бы просто использовать подход, при котором если вы знаете, что объекты простые, вы можете преобразовать их в строки, а затем строки в последовательности байтов.

Но этот подход может усложниться, если у вас есть более сложная структура объекта.

Предположим, у вас есть объект x.

И он имеет набор полей f1, f2, f3 и т. д.

И, скажем, поле f3 указывает на объект y.

Теперь нам нужно найти способ включения объекта у.

Вы можете использовать формат XML, который является стандартом обмена данными.

Существуют способы преобразования графа объектов Java в представление XML, а также много инструментов для сериализации и десериализации XML документов.

Но использование XML несет много накладных расходов.

Другой подход – это использование Java интерфейса Serializable.

В этом случае все, что нам нужно сделать, это объявить класс как реализующий интерфейс Serializable.

Теперь, как только вы это сделаете, объект класса будет автоматически преобразовываться в байты.

Если же в классе есть поле, которое не является сериализуемым, тогда вы получите исключение при попытке сериализации экземпляра этого класса.

Если вы не хотите сериализовать какое-либо поле, вы можете пометить его как transient.

Тогда это поле не будет скопировано при отправке из одной JVM в другую, что означает, что это поле будет иметь нулевое значение в восстановленном объекте.

Затем вы можете переопределить соответствующие методы чтения объекта с помощью реализации интерфейса Externalizable, чтобы поместить некоторое соответствующее значение в это поле.

Таким образом, между клиентом и сервером можно обмениваться не просто сообщениями, а объектами.

Можно создавать небольшие объекты команд, которые сериализуются и передаются между клиентом и сервером.

Как только объект команды будет прибывать в пункт назначения, он будет выполнять некий код.

Для передачи объекта между клиентом и сервером, класс объекта должен присутствовать как на стороне клиента, так и на стороне сервера.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.io.Serializable;
import java.net.Socket;

public class MessageCommand implements Serializable {

    private String message;

    public MessageCommand(String message) {
        this.message=message;
    }

    public void execute(Socket socket)throws IOException {
        PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush: true);
        out.println("To: " + socket.getLocalAddress() + ":" + socket.getLocalPort() + " Order: " + message);
    }
}
```

Кроме того, этот класс должен реализовывать интерфейс Serializable, чтобы преобразовываться в байты при записи в исходящий поток и воссоздаваться обратно в Java объект при чтении из входящего потока байтов.

Здесь мы создаем класс MessageCommand, который в своем методе execute записывает сообщение в исходящий поток сокета.

В клиенте сервера мы создаем объект MessageCommand с сообщением, затем создаем исходящий поток сокета ObjectOutputStream, в который записываем объект MessageCommand.

```
protected void handleConnectionObject(Socket client) throws IOException {  
    MessageCommand command = new MessageCommand("Hello");  
    ObjectOutputStream out = new ObjectOutputStream(client.getOutputStream());  
    BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));  
    out.writeObject(command);  
    in.lines().forEach(System.out::println);  
}
```

На стороне сервера мы создаем входящий поток сокета `ObjectInputStream` и считываем объект `MessageCommand`.

```
protected void handleConnectionObject(Socket socket) throws Exception {  
    ObjectInputStream in = new ObjectInputStream(socket.getInputStream());  
    MessageCommand command = (MessageCommand) in.readObject();  
    command.execute(socket);  
    in.close();  
    socket.close();  
}
```

После чего вызываем его метод `execute`, в котором отправляем сообщение обратно клиенту по сокет каналу.

UDP, широковещательные сообщения, многоадресная рассылка

Клиент и сервер, которые взаимодействуют через сокет TCP, имеют выделенный канал «точка-точка» между собой.



Объектно-ориентированное программирование на Java

Введение в сетевое взаимодействие

Лекция 7

UDP, Broadcast, Multicast

Они устанавливают соединение, передают данные, а затем закрывают соединение.

Все данные, отправленные по каналу, принимаются в том же порядке, в котором они были отправлены. Это гарантируется каналом.

Клиенты и серверы, которые обмениваются данными с помощью датаграмм, отправляют и получают полностью независимые пакеты информации.

У этих клиентов и серверов нет выделенного канала «точка-точка».

И доставка дейтаграмм в пункты назначения не гарантируется, также, как и их порядок доставки.

Пакет `java.net` содержит три класса, которые используют датаграммы для отправки и получения пакетов по сети – это `DatagramSocket`, `DatagramPacket` и `MulticastSocket`.

`DatagramPacket` – представляет пакет данных, который отправляется и принимается через сокет `DatagramSocket` с помощью UDP протокола.

```
public class UDPServer {
    private String address;
    private int port;

    public UDPServer(String address, int port) {
        this.address = address;
        this.port = port;
    }

    public void send(String message) throws IOException {
        DatagramSocket socket = new DatagramSocket();
        byte[] buffer = message.getBytes();
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length, InetAddress.getByName(address), port);
        socket.send(packet);
        socket.close();
    }

    public static void main(String[] args) {
        String address = "localhost";
        int port = 8080;
        UDPServer server = new UDPServer(address, port);
        try {
            server.send("Hello");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

В этом примере, у нас есть сервер, который посылает пакет `DatagramPacket` клиенту через сокет `DatagramSocket`.

Причем сокет `DatagramSocket` является потокобезопасным.

Здесь мы открываем сокет, создаем пакет данных, в котором указываем адрес назначения и порт назначения.

И методом `send` отправляем пакет.

И у нас есть клиент, который слушает на порту в бесконечном цикле, и получает пакет, отправленный на адрес машины клиента и порт сокета.

```
public class UDPClient {
    private int port;

    public UDPClient(int port) {
        this.port = port;
    }

    public void listen() {
        try {
            DatagramSocket socket = new DatagramSocket(port);
            byte[] buf = new byte[256];
            while (true) {
                DatagramPacket packet = new DatagramPacket(buf, buf.length);
                socket.receive(packet);
                String received = new String(packet.getData(), 0, packet.getLength());
                System.out.println("Получена строка: " + received);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        int port = 8080;
        UDPClient client = new UDPClient(port);
        client.listen();
    }
}
```

После получения пакета, мы можем извлечь из него сообщение.

Также, пакеты DatagramPacket могут быть отправлены всем получателям в локальной сети, которые слушают широковещательный адрес.

```
public class UDPServerBroadcast {
    private String address;
    private int port;

    public UDPServerBroadcast(String address, int port) {
        this.address = address;
        this.port = port;
    }

    public void broadcast(String message) throws IOException {
        DatagramSocket socket = new DatagramSocket();
        socket.setBroadcast(true);
        byte[] buffer = message.getBytes();
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length, InetAddress.getByName(address), port);
        socket.send(packet);
        socket.close();
    }

    public static void main(String[] args) {
        String address = "255.255.255.255";
        int port = 8080;
        UDPServerBroadcast server = new UDPServerBroadcast(address, port);
        try {
            server.broadcast("Hello");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Широковещательный адрес – условный (не присвоенный никакому устройству в сети) адрес, который используется для передачи широковещательных пакетов в компьютерных сетях.

В отличие от соединения «точка-точка», здесь нам не нужно знать IP-адрес хоста клиента.

Вместо этого используется широковещательный адрес.

В этом примере мы отправляем с сервера широковещательное сообщение на IP-адрес адрес 255.255.255.255, который является широковещательным адресом локальной сети.

Функция широкого вещания включается у сокета методом setBroadcast (true).

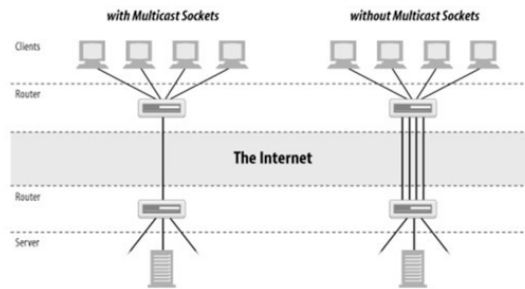
Клиент при широком вещании ничем не отличается от клиента примера соединения «точка-точка».

Он будет автоматически получать сообщения, переданные на широковещательный IP-адрес.

Широковещательный адрес в основном используется для локальных сетей.

Не существует адреса, который бы в рамках всего Интернета интерпретировался бы как широковещательный.

В одноадресной рассылке с помощью сокетов, у вас есть источник, который хочет отправить сообщение в пункт назначения.



Нам не нужно беспокоиться о деталях, но на самом деле сообщение может пройти через ряд маршрутизаторов, прежде чем оно достигнет места назначения.

И если источник должен отправить одно и то же сообщение нескольким адресатам, он должен будет повторить сообщение через другое соединение сокета, чтобы добраться до другого адресата.

Это называется одноадресной рассылкой, потому что это соединение точка-точка.

Существует также широковещательная трансляция, которая оптимизирована для локальных сетей, когда сообщение может быть отправлено от источника к маршрутизатору, для всех получателей, подключенных к этому маршрутизатору в локальной сети.

Но что делать, если адресаты находятся в другом месте в Интернете?

В этом случае вы можете использовать многоадресные сокеты, с помощью которых один источник может отправить одно и то же сообщение нескольким адресатам, которые присоединились к группе.

Это может быть очень полезно для ряда приложений: от новостных лент до видеоконференций.

Широковещательная трансляция работает только в локальной сети и не может быть использована в Интернете.

Многоадресная рассылка позволяет переслать сообщение через Интернет и затем с помощью маршрутизатора, разослать сообщение группе получателей.

Для этого в Java есть класс под названием `MulticastSocket`.

И в конструкторе этого класса указывается порт, который используется для соединения в многоадресном сокете.

Если вы не укажете порт, будет выбран анонимный порт.

Далее этот сокет присоединяется к группе, которая слушает один IP адрес.

После этого вы можете создать сообщение и отправить его в группу.

И это сообщение является дейтаграммой.

По практическим соображениям многоадресная передача не может применяться к сообщениям неограниченной длины.

Сообщения ограничены дейтаграммами, которые обычно имеют размер до 64 килобайт.

И, наконец, после того, как вы получили сообщение, если вы хотите покинуть группу, ваш сокет спокойно может уйти из группы.

Таким образом, пакеты `DatagramPacket` могут быть отправлены не всем получателям в локальной сети, а только нескольким получателям, которые слушают с помощью сокета `MulticastSocket`.

```

public class UDPServerMulticast {
    private String address;
    private int port;

    public UDPServerMulticast(String address, int port) {
        this.address = address;
        this.port = port;
    }

    public void broadcast(String message) throws IOException {
        DatagramSocket socket = new DatagramSocket();
        socket.setBroadcast(true);
        byte[] buffer = message.getBytes();
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length, InetAddress.getByLine(address), port);
        socket.send(packet);
        socket.close();
    }

    public static void main(String[] args) {
        String address = "230.0.0.0";
        int port = 8080;
        UDPServerMulticast server = new UDPServerMulticast(address, port);
        try {
            server.broadcast("Hello");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Одна вещь, которую мы должны здесь учитывать, заключается в том, что адреса, которые позволяют нам использовать MulticastSocket, ограничены в диапазоне между 224.0.0.0 и 239.255.255.255.

Поэтому здесь наш сервер использует не широковещательный адрес, а адрес 230.0.0.0 из указанной группы.

Что касается клиента, мы создаем класс, который будет принимать входящие сообщения с сервера в бесконечном цикле.

```

public class UDPClientMulticast {
    private String address;
    private int port;

    public UDPClientMulticast(String address, int port) {
        this.address = address;
        this.port = port;
    }

    public void listen() {
        try {
            MulticastSocket socket = new MulticastSocket(port);
            InetAddress group = InetAddress.getByLine(address);
            socket.joinGroup(group);
            byte[] buf = new byte[256];
            while (true) {
                DatagramPacket packet = new DatagramPacket(buf, buf.length);
                socket.receive(packet);
                String received = new String(packet.getData(), 0, packet.getLength());
                System.out.println("Получена строка: " + received);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        String address = "230.0.0.0";
        int port = 8080;
        UDPClientMulticast client = new UDPClientMulticast(address, port);
        client.listen();
    }
}

```

Многоадресная рассылка с использованием сокета MulticastSocket отправляет пакеты только тем потребителям, которые в этом заинтересованы.

И многоадресная рассылка основана на концепции членства в группах, где групповой адрес представляет каждую группу.

В IPv4 любой адрес между 224.0.0.0 и 239.255.255.255 может использоваться как групповой адрес.

Только те узлы, которые подписываются на группу, получают пакеты, которые отправляются в группу.

Здесь мы открываем сокет MulticastSocket и присоединяем его к групповому адресу методом joinGroup.

Далее мы слушаем этот адрес и получаем по нему пакет с помощью метода receive.

После получения пакета мы извлекаем из него данные.

Чтобы отсоединить сокет от группы используется метод leaveGroup.

Remote Method Invocation



Объектно-ориентированное программирование на Java

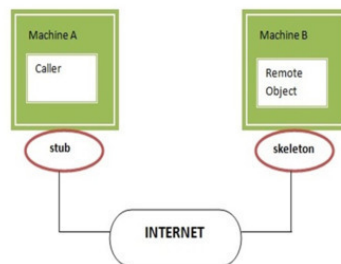
Введение в сетевое взаимодействие

Лекция 8

Remote Method Invocation

Теперь, когда мы узнали о сокетах и сериализации, давайте изучим концепцию более высокого уровня, которая называется Remote Method Invocation, или RMI.

Мы знаем, как вызовы методов работают в последовательной программе.



У вас может быть поток, T0, и в какой-то момент он вызовет, скажем, метод x.foo. Который будет выполняться в объекте x, и он может вернуть другой объект y.

И это происходит последовательно.

Теперь, что, если мы хотим попробовать это в распределенной среде?

Предположим, у нас есть, две JVM и есть поток T0, запущенный на одной JVM.

Но объект x находится на другой JVM.

И мы хотим вызвать метод x.foo и получить возвращаемое значение y.

С сокетами вы можете попытаться это имитировать, настроив сокет, передавая сообщения и настраивая потоки.

Но мы хотим просто вызвать метод foo этого удаленного объекта x.

Для этого у Java есть очень удобная концепция, которая является вызовом удаленного метода.

Идея состоит в том, что у вас есть клиент RMI, который работает на одной JVM.

И у вас есть сервер RMI, работающий на другой JVM.

И проблема состоит в том, что у вас нет прямого доступа к объекту x на другой JVM.

Эта проблема решается с помощью установки объекта-заглушки.

Который по сути является локальным прокси-сервером.

Таким образом, клиент RMI может взаимодействовать с объектом-заглушкой на своей JVM.

И этот объект-заглушка будет представителем удаленного объекта x на другой JVM.

И под капотом теперь вы можете установить сокет на обоих концах.

Однако при этом есть ограничение – объекты *x* и *y* должны быть сериализуемыми.

Потому что они должны передаваться по сети.

Поэтому их соответствующие классы должны реализовывать интерфейс *Serializable*.

Кроме того, вы должны настроить реестр RMI.

Для работы объект *x* должен быть доступен под каким-либо именем в этом реестре.

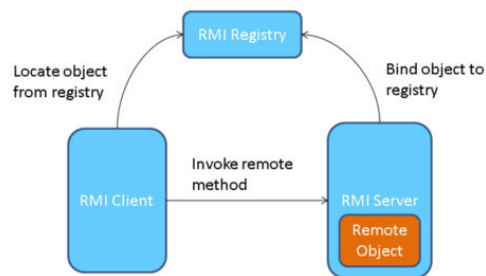
Когда клиент обращается к объекту, он использует имя, и он напрямую не ссылается на объект.

Через это имя клиент получает ссылку на локальный объект-заглушку, который позволит клиенту как передавать параметры в вызов метода, так и получать возвращаемое значение.

После того, как вы все настроили, ваш код может быть записан как обычный вызов метода объекта, который под капотом превращается в удаленный вызов метода.

Таким образом, RMI (Remote Method Invocation) – это программный интерфейс вызова удаленных методов, обеспечивающий вызов удаленных методов, работающих на другой виртуальной машине Java.

Приложение RMI включает в себя две отдельные программы, сервер и клиент.



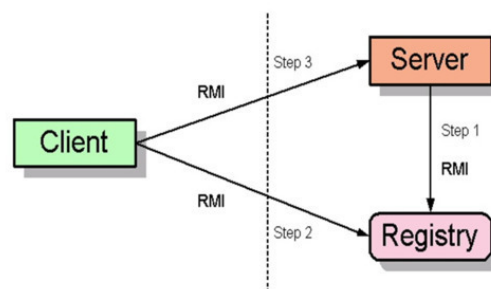
Серверная программа создает удаленный объект, делает ссылку на этот объект доступной для клиента и ждет, когда клиент вызовет методы этого объекта.

Клиентская программа получает удаленную ссылку на удаленный объект на сервере, а затем вызывает его методы.

RMI предоставляет механизм, с помощью которого сервер и клиент обмениваются информацией.

Приложения могут использовать различные механизмы для получения ссылок на удаленные объекты.

Например, приложение может регистрировать свои удаленные объекты с помощью реестра RMI.



Сервер вызывает реестр, чтобы связать имя с удаленным объектом.

После этого клиент ищет удаленный объект по его имени в реестре сервера и затем вызывает удаленный метод на сервере.

В качестве альтернативы приложение может передавать ссылки на удаленные объекты как часть других удаленных вызовов.

Детали взаимодействия с удаленным объектом скрываются механизмом RMI.

И для программиста удаленное взаимодействие похоже на обычные вызовы Java-метода.

Мы уже рассматривали обмен объектами Java между клиентом и сервером.

Но там у нас было определение класса объекта как на стороне сервера, так и на стороне клиента.

Преимуществом RMI является возможность загружать определение класса объекта, если класс не определен в виртуальной машине Java получателя.

Все типы и поведение объекта, ранее доступные только на одной виртуальной машине Java, могут быть переданы на другую, возможно удаленную виртуальную машину Java.

И поведение объектов не изменяется, когда они отправляются на другую виртуальную машину Java.

Эта возможность позволяет вводить новые типы и поведение в удаленную виртуальную машину Java, тем самым динамически расширяя поведение приложения.

Для создания удаленного объекта мы должны определить удаленный интерфейс как на стороне клиента, так и на стороне сервера.

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface RMInterface extends Remote{  
    public String hello(String name) throws RemoteException;  
}
```

Этот удаленный интерфейс должен расширять интерфейс Remote пакета java. rmi.

Интерфейс Remote служит для идентификации интерфейсов, методы которых могут быть вызваны из удаленной виртуальной машины.

И каждый метод интерфейса должен выбрасывать исключение RemoteException, которое может произойти во время выполнения удаленного вызова метода.

Далее мы создаем сервер, класс которого расширяет класс UnicastRemoteObject и реализует удаленный интерфейс.

```

public class RMIServer extends UnicastRemoteObject implements RMISInterface {

    protected RMIServer() throws RemoteException {
        super();
    }

    @Override
    public String hello(String name) throws RemoteException {
        System.out.println(name + " is trying to contact!");
        return "Server says hello to " + name;
    }

    public static void main(String[] args) {
        try {
            Registry reg = LocateRegistry.createRegistry(8080);
            reg.rebind("//localhost/MyServer", new RMIServer());
            System.out.println("Server ready");
        } catch (Exception e) {
            System.out.println("Server exception: " + e.toString());
        }
    }
}

```

Класс `UnicastRemoteObject` пакета `java.rmi.server` используется для экспорта удаленного объекта с помощью протокола Java Remote Method Protocol и получения заглушки, которая связывается с удаленным объектом.

Java Remote Method Protocol – это Java-специфический протокол для поиска и создания ссылки на удаленный объект.

Этот протокол работает поверх протоколов TCP/IP.

Заглушка – это класс, автоматически сгенерированный на основе класса, реализующего удаленный интерфейс и используемый на стороне клиента для взаимодействия с удаленным объектом на сервере.

При расширении класса `UnicastRemoteObject`, за кадром вызывается его метод `exportObject`, который экспортирует удаленный объект в виде заглушки, чтобы сделать его доступным для приема входящих вызовов.

Здесь мы также создаем конкретную реализацию удаленного интерфейса.

В методе `main` сервера мы запускаем локальный реестр на порту и связываем ссылку на удаленный объект с указанным именем в этом реестре.

После запуска сервера, он ожидает удаленных вызовов.

На стороне клиента мы получаем RMI реестр, и находим в нем ссылку на удаленный объект по его имени.

```

public class RMIClient {

    private static RMISInterface look_up;

    public static void main(String[] args)
        throws RemoteException, NotBoundException {

        Registry reg = LocateRegistry.getRegistry(8080);

        look_up = (RMISInterface) reg.lookup("//localhost/MyServer");

        String response = look_up.hello("MyName");

        System.out.println("Server response: " + response);

    }
}

```

RMI сервер регистрирует объект (на самом деле, заглушку) в реестре RMI.

Удаленный клиент ищет эту заглушку и вызывает ее методы.

За сценой вызывается метод заглушки, его аргументы сериализуются и передаются RMI-серверу.

Сервер RMI десериализует запрос, вызывает фактический метод удаленного объекта, создает результат, сериализует его и отправляет обратно клиенту (заглушке).

Заглушка десериализует результат и возвращает его обратно в код, вызывающий этот метод.

HTTP/2 клиент в Java 9



Объектно-ориентированное программирование на Java

Введение в сетевое взаимодействие

Лекция 9

HTTP/2 клиент в Java 9

До недавнего времени Java предоставлял только API `URLConnection`, который является низкоуровневым и не считается удобным для пользователя.

Поэтому широко использовались сторонние библиотеки, такие как `Apache HttpClient`, и другие.

Протокол HTTP/1.1 появился в 1997 году, и стал широко используемой версией.

В 2015 году был стандартизирован протокол HTTP/2, и в нем есть много улучшений.

Первоначальный HTTP API Java был написан с учетом версии HTTP / 1.1.

Со временем использование протокола HTTP развивалось, но Java API не поспевал за требованиями пользователей.

Таким образом, в Java 9 был введен новый API, который является более чистым и понятным в использовании, а также добавляется поддержка протокола HTTP / 2.

В HTTP/1.1 мы не можем одновременно открывать более шести подключений, поэтому каждый следующий запрос должен ждать завершения остальных.

В HTTP/1.1 мы не можем одновременно открывать более шести подключений.

В HTTP/2 можно отправлять несколько запросов данных параллельно по одному TCP-соединению.

В HTTP/1.1 каждый запрос, отправленный на сервер, имеет дополнительные данные заголовка.

В HTTP/2.0 заголовки упаковываются в один сжатый блок.

Server Push в HTTP/2 позволяет серверам самим передавать ответы клиенту, а не ждать нового запроса.

Протокол HTTP/2.0 является бинарным, а не текстовым как HTTP/1.1.

Чтобы избежать этого, выполняется различная оптимизация, такая как минимизация, сжатие и архивирование файлов.

В HTTP/2 можно отправлять несколько запросов данных параллельно по одному TCP-соединению.

В HTTP/1.1 каждый запрос, отправленный на сервер, имеет дополнительные данные заголовка, что нагружает пропускную способность.

В HTTP/2.0, заголовки упаковываются в один сжатый блок, который отправляется как единое целое, и при получении декодируется.

В среде HTTP/1.1 HTML-страница отправляется в браузер.

Браузеру необходимо разобрать ее и решить, какие дополнительные ресурсы необходимы, а затем запросить эти ресурсы с сервера.

Это можно устранить с помощью Server Push в HTTP/2, который позволяет серверам самим передавать ответы клиенту, а не ждать нового запроса.

Протокол HTTP/2.0 является бинарным, а не текстовым как HTTP/1.1.

Бинарные протоколы более эффективны для синтаксического анализа, более компактные, и, самое главное, они гораздо менее подвержены ошибкам по сравнению с текстовыми протоколами, такими как HTTP/1.1.

Java 9 API обрабатывает HTTP-соединения, используя три класса.

HttpClient обрабатывает создание и отправку запросов.

```

Class HttpClient
java.lang.Object
jdk.incubator.http.HttpClient

public abstract class HttpClient
extends Object

A container for configuration information common to multiple HTTP requests. All requests are sent through a HttpClient.
Incubating Feature. Will be removed in a future release.
HttpClients are immutable and created from a builder returned from newBuilder(). Request builders are created by calling HttpRequest.newBuilder().
See HttpRequest for examples of usage of this API.

Class HttpRequest
java.lang.Object
jdk.incubator.http.HttpRequest

public abstract class HttpRequest
extends Object

Class HttpResponse<T>
java.lang.Object
jdk.incubator.http.HttpResponse<T>

Type Parameters:
T - the response body type

public abstract class HttpResponse<T>
extends Object

```

HttpRequest используется для построения запроса для отправки через HttpClient.

HttpResponse содержит ответ от отправленного запроса.

Новый API помогает более легко поддерживать HTTP-соединения.

Чтобы использовать его, необходимо определить модуль приложения, используя файл module-info. java, в котором указать зависимость от модуля Java 9.

В отличие от HttpURLConnection, HTTP Client предоставляет синхронные и асинхронные механизмы запроса.

И HttpClient служит как контейнер для конфигурации, общей для нескольких запросов.

Первое, что мы должны сделать при создании запроса, – это предоставить URL-адрес.

```

HttpRequest.newBuilder(new URI("https://postman-echo.com/get"))
HttpRequest.newBuilder().uri(new URI("https://postman-echo.com/get"))

HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI("https://postman-echo.com/get"))
    .GET()
    .build();

HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI("https://postman-echo.com/get"))
    .version(HttpClient.Version.HTTP_2)
    .GET()
    .build();

HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI("https://postman-echo.com/get"))
    .headers("key1", "value1", "key2", "value2")
    .GET()
    .build();

HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI("https://postman-echo.com/get"))
    .timeout(Duration.of(10, SECONDS))
    .GET()
    .build();

```

Мы можем сделать это двумя способами: используя конструктор класса Builder с параметром URI или путем вызова метода uri для экземпляра класса Builder.

Далее мы должны определить метод HTTP, который наш запрос будет использовать.

Для этого мы можем вызывать один из методов из Builder – GET, POST, PUT или DELETE.

Этот запрос имеет все параметры, необходимые для HttpClient.

Однако иногда нам нужно добавить дополнительные параметры к нашему запросу, такие как версия протокола HTTP, заголовки, таймаут.

Мы можем добавить тело к запросу с помощью методов POST, PUT и DELETE, которые принимают в качестве аргумента `BodyProcessor`.

```
abstract HttpRequest.Builder POST(HttpRequest.BodyProcessor body)

abstract HttpRequest.Builder PUT(HttpRequest.BodyProcessor body)

abstract HttpRequest.Builder DELETE(HttpRequest.BodyProcessor body)

StringProcessor - читает тело из String, созданного с помощью HttpRequest.BodyProcessor.fromString.
InputStreamProcessor - читает тело из InputStream, созданного с помощью
HttpRequest.BodyProcessor.fromInputStream.
ByteArrayProcessor - читает тело из массива байтов, созданного с помощью
HttpRequest.BodyProcessor.fromByteArray.
FileProcessor - читает тело из файла по заданному пути, созданного с помощью
HttpRequest.BodyProcessor.fromFile.
```

API предоставляет ряд реализаций `BodyProcessor`, которые упрощают передачу тела запроса.

Это `StringProcessor` — читает тело из `String`, созданного с помощью `HttpRequest.BodyProcessor.fromString`.

`InputStreamProcessor` — читает тело из `InputStream`, созданного с помощью `HttpRequest.BodyProcessor.fromInputStream`.

`ByteArrayProcessor` — читает тело из массива байтов, созданного с помощью `HttpRequest.BodyProcessor.fromByteArray`.

`FileProcessor` — читает тело из файла по заданному пути, созданного с помощью `HttpRequest.BodyProcessor.fromFile`.

Если нам не нужно тело, мы можем просто передать `HttpRequest.noBody`.

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI("https://postman-echo.com/post"))
    .POST(HttpRequest.noBody())
    .build();

HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI("https://postman-echo.com/post"))
    .headers("Content-Type", "text/plain;charset=UTF-8")
    .POST(HttpRequest.BodyProcessor.fromString("Sample request body"))
    .build();

byte[] sampleData = "Sample request body".getBytes();
HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI("https://postman-echo.com/post"))
    .headers("Content-Type", "text/plain;charset=UTF-8")
    .POST(HttpRequest.BodyProcessor
        .fromInputStream(() -> new ByteArrayInputStream(sampleData)))
    .build();
```

Установка тела запроса с любой реализацией `BodyProcessor` очень проста и интуитивно понятна.

Например, если мы хотим передать простую строку как тело, мы можем использовать `StringBodyProcessor`.

Также мы можем передать входящий поток как тело.

Мы также можем использовать `ByteArrayProcessor` и передать массив байтов в качестве параметра.

```
byte[] sampleData = "Sample request body".getBytes();
HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI("https://postman-echo.com/post"))
    .headers("Content-Type", "text/plain; charset=UTF-8")
    .POST(HttpRequest.BodyProcessor.fromByteArray(sampleData))
    .build();

HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI("https://postman-echo.com/post"))
    .headers("Content-Type", "text/plain; charset=UTF-8")
    .POST(HttpRequest.BodyProcessor.fromFile(
        Paths.get("src/test/resources/sample.txt")))
    .build();
```

Для работы с файлом мы можем использовать `FileProcessor`, указывая путь к файлу в качестве параметра.

Мы рассмотрели, как создать `HttpRequest` и как установить в нем дополнительные параметры.

```
HttpResponse<String> response = HttpClient
    .newBuilder()
    .proxy(ProxySelector.getDefault())
    .build()
    .send(request, HttpResponse.BodyHandler.asString());

HttpResponse<String> response = HttpClient.newBuilder()
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build()
    .send(request, HttpResponse.BodyHandler.asString());

HttpResponse<String> response = HttpClient.newBuilder()
    .authenticator(new Authenticator() {
        @Override
        protected PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication(
                "username",
                "password".toCharArray());
        }
    }).build()
    .send(request, HttpResponse.BodyHandler.asString());
```

Теперь пришло время посмотреть на класс `HttpClient`, который отвечает за отправку запросов и получение ответов.

Все запросы отправляются с использованием `HttpClient`, который может быть создан с использованием метода `HttpClient.newBuilder` или путем вызова метода `HttpClient.newHttpClient`.

Мы можем определить прокси для соединения с метода проху в экземпляре `Builder`.

Здесь, в примере мы использовали системный прокси по умолчанию.

Иногда страница, которую мы хотим получить, переместилась на другой адрес.

В этом случае мы получим код статуса HTTP, как правило, с информацией о новом URI.

`HttpClient` может автоматически перенаправить запрос на новый URI, если мы установим соответствующую политику перенаправления.

Мы можем сделать это с помощью метода `followRedirects` в `Builder`.

Все политики определены и описаны в перечислении `HttpClient.Redirect`.

Аутентификатор – это объект, который согласовывает учетные данные для подключения. Он предоставляет различные схемы аутентификации.

В большинстве случаев для аутентификации требуется имя пользователя и пароль для подключения к серверу.

И мы можем использовать класс `PasswordAuthentication`, для передачи имени пользователя и пароля.

Новый `HttpClient` предоставляет две возможности для отправки запроса на сервер.

```

HttpResponse<String> response = HttpClient.newBuilder()
    .build()
    .send(request, HttpResponse.BodyHandler.asString());

CompletableFuture<HttpResponse<String>> response = HttpClient.newBuilder()
    .build()
    .sendAsync(request, HttpResponse.BodyHandler.asString());

List<URI> targets = Arrays.asList(
    new URI("https://postman-echo.com/get?foo1=bar1"),
    new URI("https://postman-echo.com/get?foo2=bar2"));
HttpClient client = HttpClient.newBuilder();

List<CompletableFuture<String>> futures = targets.stream()
    .map(target -> client
        .sendAsync(
            HttpRequest.newBuilder(target).GET().build(),
            HttpResponse.BodyHandler.asString())
        .thenApply(response -> response.body()))
    .collect(Collectors.toList());

```

Метод `send` – синхронно отправляет запрос, то есть блокирует до появления ответа.

И метод `sendAsync` – асинхронно отправляет запрос, то есть не дожидается ответа.

Метод `send` возвращает объект `HttpResponse`, и следующая инструкция в потоке приложения будет выполнена только тогда, когда ответ уже будет существовать.

Если же мы обрабатываем большие объемы данных, мы можем использовать метод `sendAsync`, который возвращает `CompletableFuture`, чтобы обрабатывать запрос асинхронно.

И API также позволяет обрабатывать несколько ответов, передавая запросы в потоке.

Мы также можем определить `Executor`, который предоставляет потоки, которые будут использоваться асинхронными вызовами.

```

ExecutorService executorService =
    Executors.newFixedThreadPool(2);

CompletableFuture<HttpResponse<String>> response1 =
    HttpClient.newBuilder()
        .executor(executorService)
        .build()
        .sendAsync(request, HttpResponse.BodyHandler.asString());

CompletableFuture<HttpResponse<String>> response2 =
    HttpClient.newBuilder()
        .executor(executorService)
        .build()
        .sendAsync(request, HttpResponse.BodyHandler.asString());

HttpClient.newBuilder()
    .cookieManager(new CookieManager(null,
        CookiePolicy.ACCEPT_NONE))
    .build();

httpClient.cookieManager().get().getCookieStore()

```

Таким образом, мы можем, например, ограничить количество потоков, используемых для обработки запросов.

Также можно установить `CookieManager` для соединения.

Например, мы можем определить `CookieManager`, который не позволяет принимать файлы cookie вообще.

Если наш `CookieManager` позволяет хранить файлы cookie, мы можем получить текущее хранилище файлов cookie.

Класс `HttpResponse` представляет собой ответ от сервера.

```
HttpResponse<String> response = HttpClient.newHttpClient()
    .send(request, HttpResponse.BodyHandler.asString());

HttpHeaders responseHeaders = response.headers();

HttpResponse<String> response = HttpClient.newHttpClient()
    .send(request, HttpResponse.BodyHandler.asString());

CompletableFuture<HttpHeaders> trailers = response.trailers();

HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI("https://postman-echo.com/get"))
    .version(HttpClient.Version.HTTP_2)
    .GET()
    .build();
HttpResponse<String> response = HttpClient.newHttpClient()
    .send(request, HttpResponse.BodyHandler.asString());

assertThat(response.version(), equalTo(HttpClient.Version.HTTP_1_1));
```

Он предоставляет ряд полезных методов, таких как метод `statusCode`, который возвращает код состояния для ответа.

Метод `body` – возвращает тело ответа.

И другие методы, такие как `uri`, `headers`, `trailers` и `version`.

Метод `uri` возвращает URI, с которого мы получили ответ.

Иногда он может отличаться от URI в объекте запроса, поскольку может произойти перенаправление.

Мы можем получить заголовки ответа, вызвав метод `headers`.

И HTTP ответ может содержать дополнительные заголовки, которые включаются после содержимого ответа.

Эти заголовки называются трейлерами.

Мы можем получить их, вызывая метод `trailers` для `HttpResponse`.

Метод `version` запроса определяет, какая версия протокола HTTP используется для общения с сервером.

Помните, что даже если мы определим, что хотим использовать HTTP/2, сервер может отвечать через HTTP/1.1.

Получить версию, которую использовал сервер для ответа, можно с помощью метода `version`.

Разработка ПО



Объектно-ориентированное программирование на Java

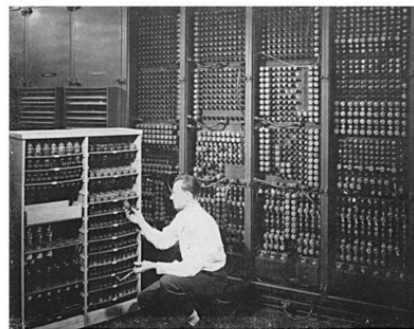
Системы контроля версий

Лекция 1

Разработка ПО

На заре компьютерной эры разработчик программ являлся оператором машины.

Потому что программирование на первом компьютере, работающем на тысячах радиолампах, осуществлялось путем вращения дисков и изменения кабельных подключений.



В те времена для обеспечения работы компьютеров вместо программирования использовалась аппаратная инженерия.

Программисты несли на себе бремя ответственности за управление и эксплуатацию компьютера.

В 1961 году президент США Джон Кеннеди провозгласил амбициозную лунную программу.

Проект NASA по разработке ПО возглавила математик из Массачусетского технологического института Маргарет Гамильтон.

Которая разработала программную инженерию.

Методы программной инженерии включали в себя отладку всех отдельных компонентов, тестирование отдельных компонентов до этапа сборки и интеграционное тестирование, при котором отдельные программные модули объединяются и тестируются в группе.

До 1964 года существовала практика создания компьютеров, которые были специфичными и соответствовали требованиям конкретного заказчика.

Оборудование и программное обеспечение были не стандартизованы и не взаимозаменяемы.

Вплоть до конца 1960-х годов компьютеры использовались на условиях аренды.

Это было связано с высокой ценой оборудования, программного обеспечения и сопутствующих услуг.

При этом обычно предоставлялся исходный код для программного обеспечения, установленного на компьютере.

После того как в 1969 году против американской компании IBM был подан антимонопольный иск, произошло разделение аппаратного и программного обеспечения и конкретных моделей компьютеров.

В результате стала возможной раздельная покупка оборудования и программного обеспечения.

Это привело к изменению подхода к программному обеспечению, которое приобрело денежную стоимость, что, в свою очередь, привело к прекращению распространения открытого программного кода.

Появление все более сложных компьютерных систем, в свою очередь, привело к неизбежности специализации навыков и к распределению ролей.

Подобные роли включали системных администраторов, специализирующихся в области управления системами, а также инженеров-программистов, которые специализировались на создании новых продуктов.

Появление языков программирования высокого уровня привело к тому, что процесс разработки ПО стал более абстрактным, все сильнее отдаляясь от оборудования и системных инженеров.

С появлением HTTP-сервера Apache началась эпоха использования решений с открытым программным кодом.

Программы с открытым исходным кодом начали конкурировать с программными решениями, имеющими закрытый программный код.

Учитывая ту простоту, с которой могли создаваться новые веб-приложения, в конце 1990-х годов отдельным программистам и коллективам приходилось работать быстрее и проявлять большую гибкость, чтобы быть конкурентоспособными.

Поэтому стали развиваться методологии разработки программного обеспечения.

Появилась гибкая разработка программного обеспечения – серия подходов к разработке программного обеспечения, с использованием итеративной разработки, динамического формирования требований и их реализации в результате постоянного взаимодействия внутри самоорганизующихся рабочих групп, состоящих из специалистов различного профиля.

И появились различные формы гибкой разработки, такие как экстремальное программирование, метод разработки динамических систем, Scrum, итеративная разработка программного обеспечения.

В 2009 году появилась методология DevOps (development и operations), способствующая быстрому выпуску релизов – набор практик, нацеленных на активное взаимодействие специалистов по разработке со специалистами по обслуживанию программного обеспечения и взаимную интеграцию их рабочих процессов друг в друга.

Эта методология нацелена на переключение внимания с результата на людей и процессы, начали поощряться сотрудничество и кооперация, исчезла конкуренция среди специалистов.

Результаты в сфере разработки программного обеспечения в значительной степени зависят от человеческого фактора.

Внимание на культуре и процессах способствует итеративному улучшению способов разработки и качества производимого продукта.

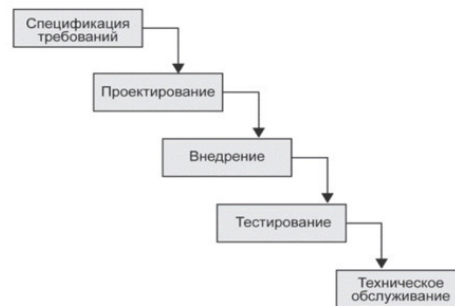
В каждой методологии разработки программного обеспечения предусматривается разбиение работы или процесса разработки программного обеспечения на фазы, каждая из которых представляет собой отдельный набор действий.

Обычно выделяются следующие фазы процесса разработки программного обеспечения – спецификация конечных результатов работы или артефактов, разработка и верификация кода

в соответствии со спецификацией и развертывание кода в финальной потребительской или производственной среде.

Процесс разработки программного обеспечения можно представить в виде моделей.

Каскадная модель (waterfall model) – модель процесса разработки программного обеспечения, жизненный цикл которой выглядит как поток, последовательно проходящий фазы анализа требований, проектирования, реализации, тестирования, интеграции и поддержки.



В этой модели много времени тратится на этапах спецификации требований и проектирования, что позволяет сократить количество ошибок, допускаемых на следующих этапах.

Во времена активного использования каскадной модели была высокая стоимость доставки программного обеспечения, распространяемого на компакт-дисках или на дискетах.

И была высокая стоимость ручной установки программ, выполняемой на рабочем месте заказчика.

В случае необходимости устранения ошибок нужно было записывать и распространять новые дискеты или компакт-диски.

Из-за подобных расходов было целесообразнее потратить больше времени и сил на стадии спецификации требований, чем пытаться устранять ошибки на следующих стадиях.

В этой модели заказчик участвует в самом начале, при разработке требований, и в конце, во время приёмочных испытаний.

Инкрементная модель подразумевает разработку программного обеспечения с линейной последовательностью стадий, но в несколько инкрементов (версий).



Разработка программного обеспечения ведется итерациями с циклами обратной связи между этапами.

Процедура разработки по инкрементной модели предполагает выпуск на первом большом этапе продукта в базовой функциональности, а затем уже последовательное добавление новых функций, так называемых «инкрементов».

Эта модель не позволяет оперативно учитывать возникающие изменения и уточнения требований к ПО.

Согласование результатов разработки с заказчиком производится только в точках, планируемых после завершения каждого этапа работ.

Спиральная модель – это модель, где на каждом витке спирали выполняется создание очередной версии продукта, уточняются требования проекта, определяется его качество и планируются работы следующего витка.

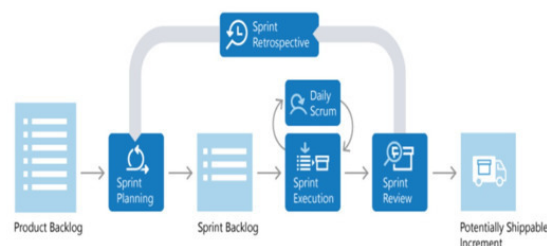


Каждый виток спирали соответствует созданию работоспособного фрагмента или версии системы.

Спиральная модель похожа на инкрементную, но с добавлением нового элемента – анализа рисков.

Эта спираль может продолжаться до бесконечности, поскольку каждая ответная реакция заказчика на созданную версию может порождать новый цикл, что отдалает окончание работы над проектом.

Термин Agile – гибкая методология разработки – относится к целой группе методологий, применяемых при разработке программного обеспечения, использующих итеративную разработку, динамическое формирование требований и их реализации в результате постоянного взаимодействия внутри самоорганизующихся рабочих групп, состоящих из специалистов различного профиля.



При итеративной разработке не требуется для начала полная спецификация требований.

Вместо этого, создание начинается с реализации части функционала, которая становится базой для определения дальнейших требований.

И этот процесс повторяется.

Версия может быть неидеальна, главное, чтобы она работала.

К гибким методологиям относится Scrum.

В методологии Scrum основной упор делается на максимизации способностей команды разработчиков к быстрому реагированию на изменение требований к самому проекту и со стороны заказчиков.

При этом используются predetermined циклы разработки, называемые спринты.

Обычно длина циклов варьируется от одной недели до одного месяца.

Процесс разработки ПО начинается с совещания по планированию спринтов, на котором определяются цели и выполняется обзор спринтов.

Одна из ключевых особенностей методологии Scrum – проведение ежедневных Scrum-встреч, на которых члены команды как можно быстрее дают ответы на следующие три вопроса:

Что из того, что я сделал, помогло команде достичь целей спринта?

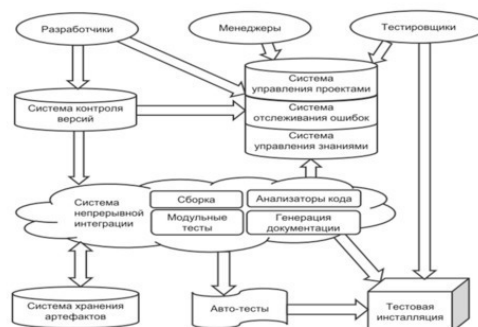
Что я планирую сделать сегодня, чтобы помочь команде достичь этих целей?

Что делать в том случае, когда какое-либо препятствие мешает мне или команде достичь целей?

При использовании методологий разработки ПО используется ряд концепций разработки, релиза и развертывания ПО.

Эти концепции описывают порядок разработки и развертывания программного обеспечения и дают представление о степени связи между ними.

Система контроля версий фиксирует изменения файлов или наборов файлов, которые хранятся в системе.



Это могут быть файлы исходного кода, ресурсы и другие документы, которые являются частью процесса разработки программного обеспечения.

Разработчики вносят изменения в форме пакетов, называемых фиксациями, или ревизиями.

Каждая ревизия, наравне с метаданными, такими как «кто внес изменения и когда», хранится в системе.

Система контроля версий своими возможностями по фиксации, сравнению, выполнению слияния и восстановлению прежних ревизий объектов в хранилище, помогает организовать кооперацию и сотрудничество внутри команды и между командами.

Она сводит к минимуму возможные риски, так как предоставляет способ вернуться к предыдущим версиям объектов.

Разработка через тестирование (test-driven development) – это техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который

позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.

Разработка через тестирование требует от разработчика создания автоматизированных модульных тестов, определяющих требования к коду непосредственно перед написанием самого кода.

Если разработчики сами разрабатывают тесты, циклы обратной связи существенно сокращаются.

К тому же разработчики принимают на себя больше ответственности за создание качественного кода.

Непрерывная интеграция – это процесс интегрирования нового кода, написанного разработчиками, в основной код, который осуществляется в течение рабочего дня.

Этот подход отличается от методики, в соответствии с которой разработчики работают с независимыми ветками неделями или месяцами, выполняя слияние кода в основную ветку только после полного завершения работы над проектом.

Длительные периоды времени между слияниями приводят к тому, что в код вносится очень много изменений, что повышает вероятность появления ошибок.

Если же используются небольшие наборы изменений, для которых часто выполняется слияние, поиск ошибок значительно упрощается.

В системах непрерывной интеграции после завершения слияния новых изменений обычно автоматически выполняется набор тестов.

Системы контроля версий



Объектно-ориентированное программирование на Java

Системы контроля версий

Лекция 2

Обзор систем контроля версий

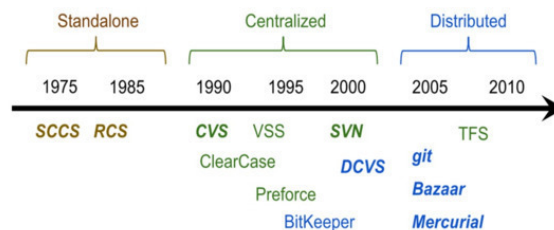
Современное программное обеспечение разрабатывается, как правило, командой программистов.

При этом командная работа над проектом может состоять как из разделения проекта между членами команды на подзадачи, так и совместной работы нескольких членов команды над отдельной подзадачей.

В случае совместного выполнения одной и той же задачи, командная разработка кода требует контроля изменений в исходном коде, обеспечивающего координацию и интеграцию изменений в исходном коде с сопровождением истории его состояния.

Такой контроль изменений в исходном коде помогает осуществлять специальное программное обеспечение системы управления версиями (Version Control System или Revision Control System).

Существует большой набор систем управления версиями, таких как CVS (Concurrent Versions System), Subversion (SVN), Git, Mercurial и др.



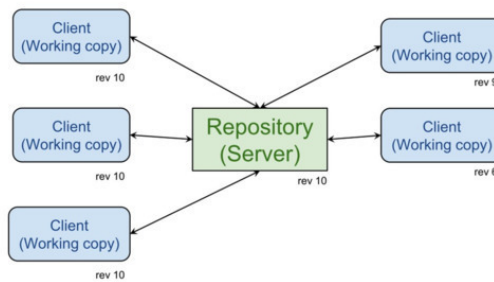
Самыми первыми появились локальные системы контроля версий, в которых все разработчики должны были использовать одну и ту же файловую систему.

Revision Control System (RCS) – это набор команд UNIX, которые позволяли нескольким пользователям разрабатывать и поддерживать программный код или документы.

Система Source Code Control System (SCCS) – также часть операционной системы UNIX.

Системы CVS и SVN являются представителями централизованных систем управления версиями, а системы Git и Mercurial – представителями распределенных систем управления версиями.

Централизованная система – это система, в которой существует единое хранилище – репозиторий, управляемый сервером, и его клиенты.



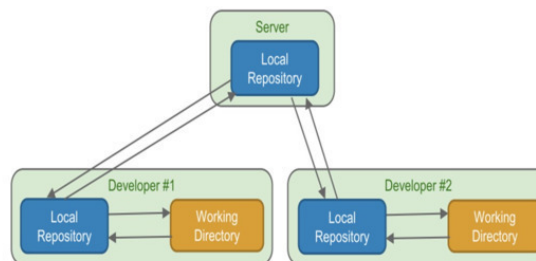
Для работы с ресурсом клиент получает нужную ему версию из репозитория, создавая рабочую копию ресурса.

После внесения изменений в рабочую копию ресурса – она помещается обратно в репозиторий, становясь новой версией ресурса.

При этом репозиторий хранит не сами ресурсы различных версий, а патчи – списки изменений между последовательными версиями, что позволяет уменьшить объем хранимых данных.

Основным недостатком систем ЦСУВ является уязвимость центрального репозитория, а также сложность с администрированием прав.

Распределенная система – это система, в которой каждый член команды или команда хранит у себя на компьютере, в локальном репозитории, собственную ветвь версий всего проекта.



При этом координатор проекта определяет из всех ветвей главную ветвь, с которой ведется синхронизация остальных ветвей.

Таким образом, в распределенных системах устраняется уязвимость центрального репозитория и решается проблема с администрированием, однако возрастает требуемый объем памяти на локальном компьютере.

CVS

Централизованная система Concurrent Versions System CVS имеет клиент-серверную архитектуру, в которой на центральном сервере хранятся индивидуальные истории файлов, а клиент имеет копии всех файлов, над которыми ведется работа.



Объектно-ориентированное программирование на Java

Системы контроля версий

Лекция 3

CVS

<http://www.nongnu.org/cvs/>

CVS - Concurrent Versions System

[Introduction](#) / [News](#) / [Documentation](#) / [Get the Software](#) / [Help and Bug Reports](#) / [Development](#)

Introduction to CVS

CVS is a version control system, an important component of Source Configuration Management (SCM). Using it, you can record the history of sources files, and documents. It fills a similar role to the free software RCS, RCS, and Delta packages.

CVS is a production quality system in wide use around the world, including many free software projects.

While CVS stores individual file history in the same format as RCS, it offers the following significant advantages over RCS:

- It can run scripts which you can supply to log CVS operations or enforce site-specific policies.
- Client server CVS enables developers scattered by geography or slow modems to function as a single team. The version history is stored on a single central server and the client machines have a copy of all the files that the developers are working on. Therefore, the network between the client and the server must be up to perform CVS operations (such as checkins or updates) but need not be up to edit or manipulate the current versions of the files. Clients can perform all the same operations which are available locally.
- It allows several developers or teams to work on the same version of the files, because of geography and/or policy. CVS's *vendor branches* can import a version from another team (even if they don't use CVS), and then CVS can merge the changes from the vendor branch with the latest files of that is what is desired.
- *Unreserved checkouts*, allowing more than one developer to work on the same files at the same time.
- CVS provides a flexible modular database that provides a symbolic mapping of names to components of a larger software distribution. It applies names to collections of directories and files. A single command can manipulate the entire collection.
- CVS servers run on most Unix variants, and clients for Windows NT/95, OS/2 and VMS are also available. CVS will also operate in what is sometimes called server mode against local repositories on Windows NT/95.

Серверная часть системы CVS позволяет организовать хранение проекта в виде модуля – набора каталогов и файлов проекта.

При этом CVS-репозиторий сервера может хранить и обслуживать несколько модулей.

Интерфейс системы CVS дает возможность извлечь модуль из репозитория и создать его рабочую копию на локальном компьютере клиента, зафиксировать сделанные клиентом изменения рабочей копии в репозитории, обновить рабочую копию из репозитория.

Каждый файл, хранящийся в репозитории, характеризуется своим уникальным номером версии (ревизией), состоящим из последовательности целых чисел, разделенных точками.

В качестве начальной версии файла CVS-сервер присваивает последовательность 1.1.

При фиксации изменений файла в репозитории последняя цифра его номера версии увеличивается на единицу.

Сохраняя изменения в репозитории, CVS-сервер сохраняет также комментарии к изменениям, дату и имя клиента.

Для хранения изменений CVS-сервер использует механизм дельта-компрессии, позволяющим хранить не сами измененные файлы, а патчи – списки изменений файла.

С помощью специальной метки (тэга) всему модулю с определенным набором ревизий может быть присвоен номер версии (релиза), после чего зафиксированный набор ревизий будет храниться в виде отдельной иерархии.

Также в любой точке иерархии ревизий можно создать отдельную ветвь, которая помечается тэгом ветви.

При этом точка разветвления помечается двумя тэгами – тэгом ветви и тэгом версии, позволяющим произвести слияние. К номеру версий ветви добавятся две цифры, последняя из которых будет увеличиваться на единицу при фиксации изменений в репозитории.

Система CVS обеспечивает разрешение конфликтных ситуаций, когда одновременно над одной и той же копией проекта работают несколько членов команды, путем автоматического слияния непересекающихся изменений или, предлагая вручную разрешить конфликт изменений.

Система CVS имеет ряд недостатков, таких как невозможность создания иерархии ревизий каталогов с отслеживанием переименования файлов каталога, отсутствие поддержки наборов изменений и др.

Так как система CVS считается устаревшей, в настоящее время бесплатного онлайн хостинга для создания удаленного репозитория не найти.

Subversion

Система управления версиями Subversion (SVN) является следующей ступенью развития системы CVS, устраняющей такие недостатки системы CVS как невозможность управления версиями каталогов, отсутствие атомарности многообъектных фиксаций, учета копирования, перемещения и переименования ресурсов в истории, отсутствие поддержки наборов изменений.



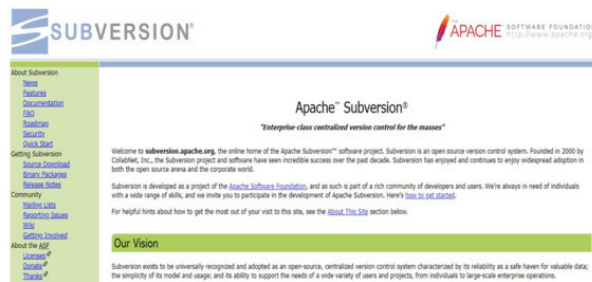
Объектно-ориентированное программирование на Java

Системы контроля версий

Лекция 4

Subversion

<http://subversion.apache.org/>



Кроме того, в системе Subversion вводятся свойства, состоящие из пар имя-значение, которые могут связываться с каталогами и файлами.

Свойства также подпадают под контроль версий.

В системе Subversion используется другой, по сравнению с системой CVS, механизм создания веток и релизов – система Subversion создает ветки и релизы путем простого копирования проекта, т.е. релиз в системе Subversion – это ветка, в которой больше не делают изменений.

В системе CVS клиенту из репозитория передается список изменений, а от клиента в репозиторий – полностью весь ресурс.

В системе Subversion – в обе стороны передается только список изменений.

В системе Subversion создается список изменений, как для текстовых файлов, так и для бинарных файлов, а в системе CVS каждая новая версия бинарного файла сохраняется в репозитории полностью.

Хотя система Subversion, так же, как и система CVS, в случае параллельной работы нескольких участников команды над одной и той же ревизией проекта, для изменения текстовых файлов предлагает механизм копирование-изменение-слияние, для изменения бинарных файлов, где слияние изменений от нескольких клиентов невозможно, она обеспечивает

механизм блокирование-изменение-разблокирование, который предотвращает параллельную работу.

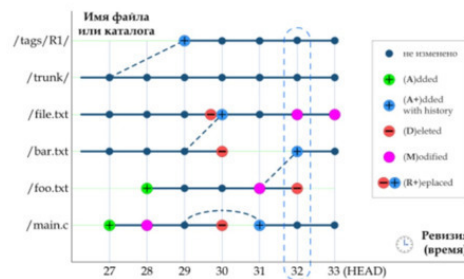
В системе Subversion номер ревизии присваивается всем файлам и каталогам, включая сам репозиторий.

Начальная ревизия Subversion-репозитория обозначается цифрой 0.

Каждое зафиксированное изменение любого ресурса в репозитории увеличивает его ревизию на единицу.

При этом в системе Subversion фиксация изменений в репозитории не приводит автоматически к обновлению рабочей копии – для получения текущей ревизии необходимо произвести операцию обновления.

Репозиторий системы Subversion представляет собой файловую систему, в которой каждый проект представлен своим каталогом, содержащим, как правило, папки trunk (главный ствол разработки), branches (ветви разработки) и tags (релизы разработки).



Рабочая копия получается путем копирования отдельного каталога (не отдельного файла) из репозитория, при этом в рабочей копии создается скрытый каталог .svn, содержащий служебную информацию, которая включает в себя ревизию, на основе которой сделана рабочая копия, и метку, указывающую, когда рабочая копия последний раз обновлялась.

Каждому файлу или каталогу репозитория может быть присвоен набор свойств, история изменений которых также отслеживается.

Помимо пользовательских свойств система Subversion может присваивать ряд служебных свойств, имена которых начинаются с префикса «svn:».

Служебные свойства хранят информацию о MIME-типе файла, ключевых словах для подстановки, правах чтения-записи, файлах и каталогах, исключенных из-под контроля версий, дате и времени создания ревизии, клиенте и др.

Subversion умеет выполнять подстановку ключевых слов – автоматическую подстановку в файле вместо шаблонных ключевых слов данных о ревизии, авторе, дате изменения, путь в репозитории.

Subversion в IntelliJ IDEA



Объектно-ориентированное программирование на Java

Системы контроля версий

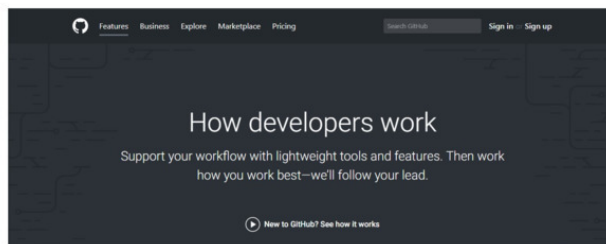
Лекция 5

Subversion в IntelliJ IDEA

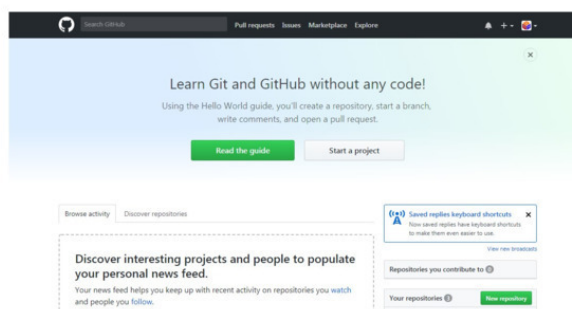
Для того чтобы начать работу с системой контроля версий нужен клиент и адрес удаленного репозитория.

В качестве удаленного репозитория возьмем GitHub – крупнейший веб-сервис для хостинга IT-проектов и их совместной разработки.

<https://github.com>



Зарегистрируем на сайте аккаунт.
И создадим репозиторий.



Обычно один репозиторий используется для организации одного проекта.

Репозиторий может содержать папки и файлы, изображения, видео, электронные таблицы и наборы данных – все, что требуется вашему проекту.

Рекомендуется также включать README файл с информацией о проекте.

GitHub позволяет добавить README файл при создании нового репозитория.

Owner: hubot / Repository name: hello-world

Great repository names are short and memorable. Need inspiration? How about [petulant-shame](#).

Description (optional): Just another repository

Public: Anyone can see this repository. You choose who can commit.

Private: You choose who can see and commit to this repository.

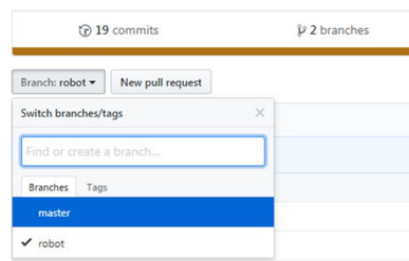
☒ Initialize this repository with a README
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: None Add a license: None

Create repository

Также можно выбрать тип лицензии для проекта.

Для параллельной разработки или разработки нового функционала без риска сломать работающий код, в проекте можно создавать ветки.



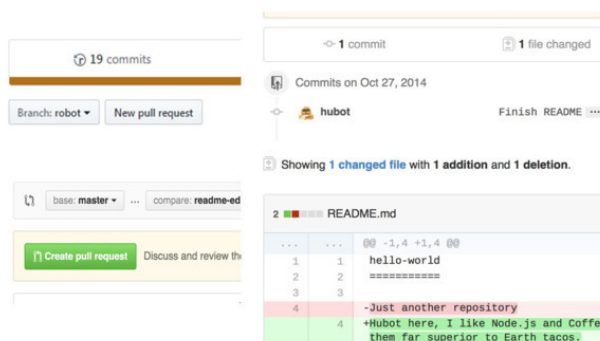
Ветвление – это способ работы с различными версиями репозитория одновременно.

По умолчанию репозиторий имеет одну ветвь с именем `master`, которая считается окончательной ветвью.

Для создания новой ветки вы раскрываете список Branch и вводите имя ветки.

Когда вы создадите ветку от главной ветки, вы создадите копию или моментальный снимок ветки `master`.

После завершения работы над веткой, вы можете объединить ее с основной веткой с помощью создания запроса pull request.



Запрос Pull Request – это основа совместной работы над проектом.

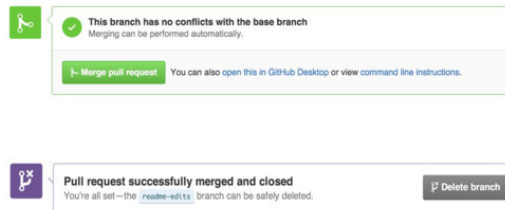
Когда вы открываете этот запрос, вы предлагаете свои изменения и запрашиваете, чтобы кто-то просмотрел их и объединил их в свою ветку.

Запросы Pull Request показывают различия между содержимым обеих ветвей.

После того, как вы посмотрели изменения на странице сравнения, если вы хотите их отправить, вы нажимаете кнопку Create Pull Request.

И даете вашему запросу заголовок и краткое описание ваших изменений.

На последнем этапе изменения объединяются с помощью объединения веток.

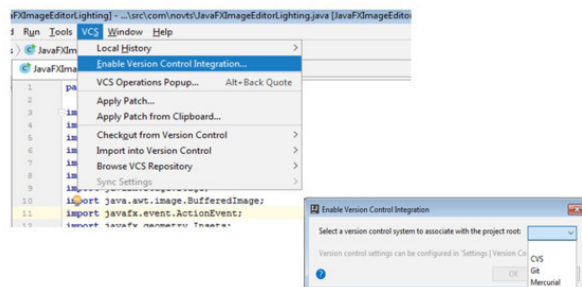


Для этого используется кнопка Merge pull request.

После объединения ветка может быть удалена с помощью кнопки Delete branch.

Таким образом, операция слияния веток требует ручной работы.

Что касается клиента системы контроля версий, вы можете использовать возможности среды разработки.

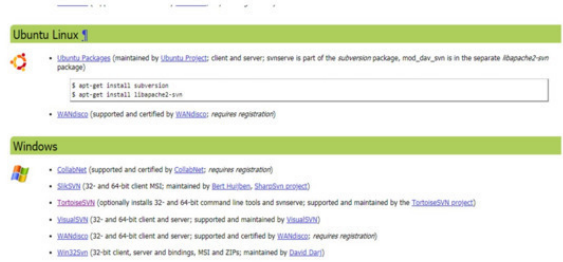


Откройте проект в IntelliJ IDEA и в меню VCS выберите Enable Version Control Integration.

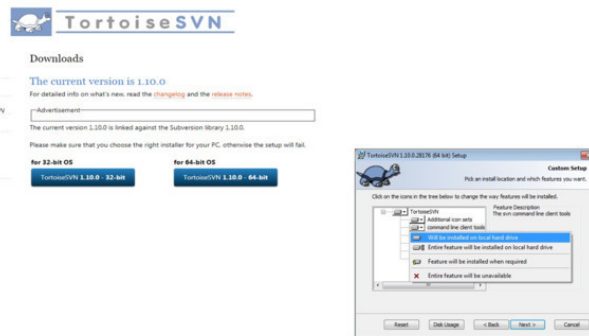
Далее выберите систему Subversion и нажмите OK.

При этом в папку idea проекта добавится файл vcs.xml.

Перед этим шагом, вам также необходимо загрузить и установить клиент командной строки с сайта Apache Subversion.

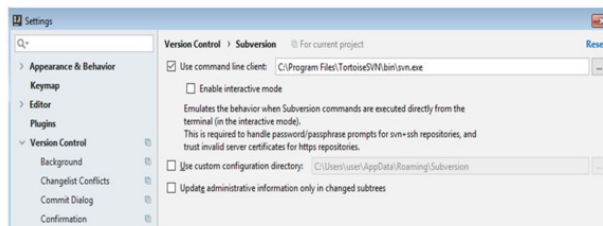


Выберем TortoiseSVN и установим клиент.

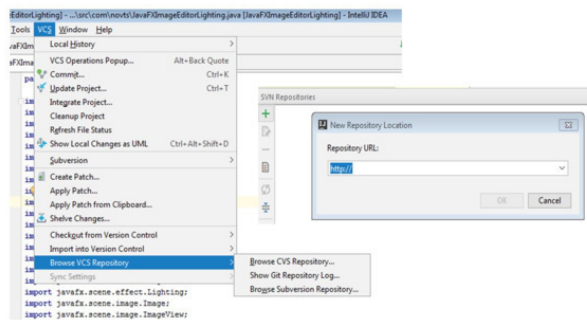


При установке не забудьте указать установку клиента командной строки.

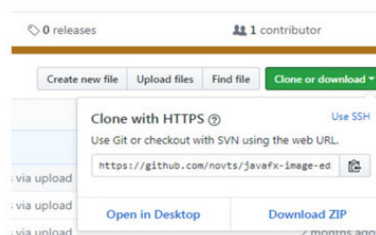
После установки, в среде IntelliJ IDEA в настройках укажите путь к клиенту командной строки svn.



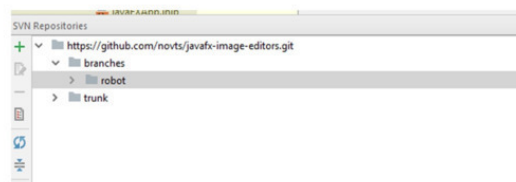
В меню VCS среды IntelliJ IDEA выберем Browse Subversion Repository и кнопкой + добавим URL адрес репозитория.



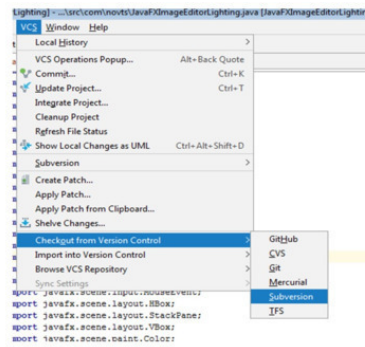
URL адрес репозитория возьмем в GitHub, нажав кнопку Clone or download проекта.



После ввода адреса репозитория и нажатия кнопки ОК, в окне SVN Repositories появится содержимое удаленного репозитория.

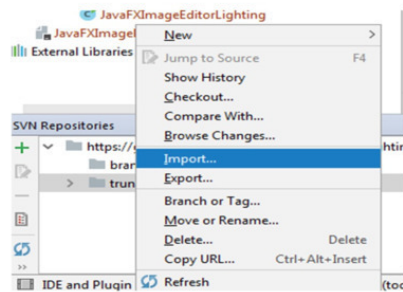


С помощью меню Check Out From Subversion вы можете получить локальную рабочую копию репозитория, которую вы можете редактировать.



После внесения необходимых изменений вы можете опубликовать результаты.

Вы можете создать пустой проект в GitHub и с помощью меню Import импортировать свой локальный проект в репозиторий.

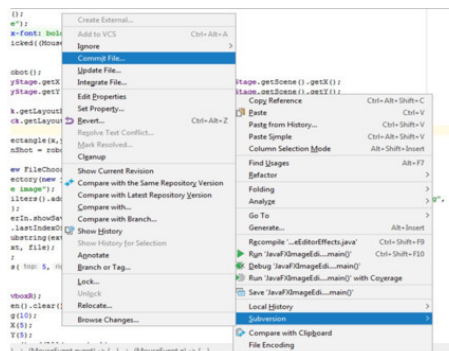


Далее вы все равно должны сделать Check Out From Subversion, чтобы создать из импортированного проекта рабочую копию репозитория.

При этом в каталог добавляется папка svn.

Команда checkout выкачивает с сервера последнюю ревизию и создает рабочую копию.

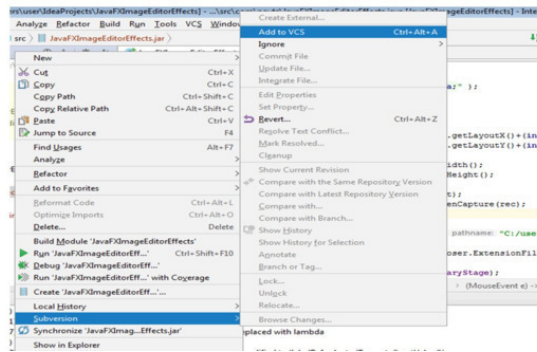
После того, как у вас есть рабочая копия репозитория, вы делаете изменения в проекте.



И эти изменения вы можете отправить в удаленный репозиторий с помощью команды commit.

Команда commit заливает изменения из рабочей копии в репозиторий.

Добавить новый файл под контроль версий вы можете с помощью команды Add to VCS.



После этого вы можете отправить новый файл в удаленный репозиторий с помощью команды `commit`.

Если вы хотите переключиться на другую ветку, вы должны сделать `Check Out From Subversion` и создать рабочую копию ветки репозитория, предварительно удалив старую папку `svn`.

Git

Система контроля версий Git – это распределенная система, поэтому работа над проектом, находящимся под контролем системы Git, подразумевает, в первую очередь, взаимодействие с локальным репозиторием – удаленный репозиторий нужен, чтобы отправить локальную версию проекта для ее использования другими членами команды.

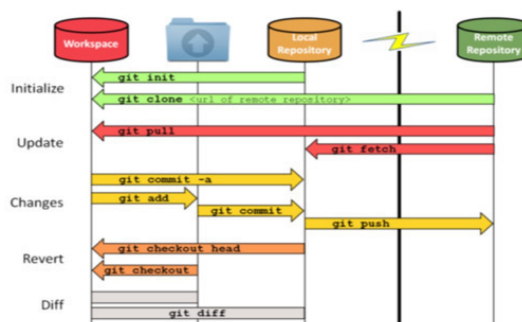


Объектно-ориентированное программирование на Java

Системы контроля версий

Лекция 6

Git



Использование локального репозитория дает существенное увеличение скорости и независимость от сети.

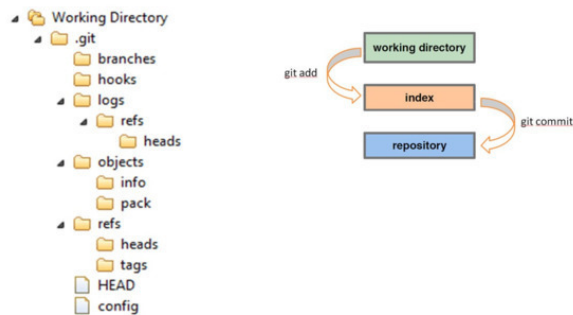
Цикл работы в системе Git с новым проектом состоит из следующих этапов:

1. Создание нового локального проекта.
2. Создание локального Git-репозитория и размещение в нем нового проекта.
3. Закачка ресурсов локального репозитория в удаленный репозиторий.
4. Редактирование ресурсов локального проекта с сохранением изменений в локальном репозитории.
5. Отправка изменений из локального репозитория в удаленный репозиторий.
6. Получение изменений из удаленного репозитория в локальный репозиторий.

Цикл работы в системе Git с уже имеющимся в удаленном репозитории проектом состоит из следующих этапов:

1. Создание локальной копии удаленного репозитория.
2. Редактирование ресурсов локального проекта с сохранением изменений в локальном репозитории.
3. Отправка изменений из локального репозитория в удаленный репозиторий.
4. Получение изменений из удаленного репозитория в локальный репозиторий.

Локальный Git-репозиторий представляет собой каталог, содержащий папку проекта – рабочий каталог с ресурсами, над которыми идет текущая работа, и папку. `.git`, включающую в себя всю историю и метаданные проекта.



Каталог. `git` содержит следующие файлы и папки:

Это файл `index` – в рассмотренных выше системах CVS и SVN, файлы проекта могут находиться в двух состояниях – измененном локальном состоянии и сохранившим изменения в репозитории.

В системе Git существует промежуточное состояние – файл был изменен и подготовлен к сохранению в репозитории.

Так вот, файл `index` хранит информацию о таких файлах, т.е. о наборе изменений, которые будут зафиксированы в репозитории.

Таким образом, файл `index` представляет область подготовленных файлов (staging area).

Это файл `HEAD` – содержит указатель на ветвь проекта, над которой идет текущая работа в рабочем каталоге.

Файл `FETCH_HEAD` – содержит информацию о полученных изменениях из удаленного репозитория в локальный репозиторий.

Файл `config` – содержит Git-конфигурацию проекта.

Папка `refs` – другое отличие системы Git от рассмотренных выше систем CVS и SVN заключается в том, что система Git не оперирует номерами версий, а использует хеши, таким образом, обеспечивая одновременно с идентификацией данных, целостность данных.

Кроме того, так как в распределенной среде нет центрального сервера, который может назначать уникальные номера ревизий, в качестве идентификаторов ревизий используются хеши.

Каталог `refs` включает в себя три папки `heads`, `remotes` и `tags`, содержащие файлы с именами локальных ветвей, удаленных ветвей и релизов проекта.

В свою очередь каждый такой файл содержит хеш последней фиксации (commit) соответствующей ветви.

Файл `HEAD` содержит путь в каталоге `refs` к файлу ветви проекта, над которой идет текущая работа в рабочем каталоге.

Файл `FETCH_HEAD` указывает хеши фиксаций, содержащиеся также в файлах папки `remotes` каталога `refs`.

Папка `objects` – содержит объекты системы Git.

Существует три типа Git-объектов – Blob-объекты – содержимое файлов, деревья (tree) – папки и имена файлов со ссылками на соответствующие объекты, и фиксации (commit).

Объект Blob (Binary Large Object) системы Git – это файл, содержащий бинарные данные заголовка плюс содержимого версии файла проекта, с именем, состоящим из хеша содержимого файла Blob-объекта, за исключением первых двух символов.

Первые два символа хеша содержимого файла Blob-объекта используются для именования папки, в которой содержится Blob-объект.

Дерево – это файл, содержащий сжатый заголовок плюс одна или несколько записей, с именем, состоящим из хеша содержимого файла дерева, за исключением первых двух символов.

Первые два символа хеша содержимого файла дерева используются для именования папки, в которой содержится дерево.

Запись содержимого дерева – это строка, состоящая из кода доступа, типа объекта (Blob-объект или дерево), хеша объекта и имени объекта (имя файла или имя директории).

Таким образом, дерево представляет содержимое каталога.

Фиксация – это файл, содержащий сжатый заголовок плюс запись, которая состоит из указателей на дерево проекта, автора, родительскую и дочернюю фиксации, ветвь проекта и комментария, с именем, состоящим из хеша содержимого файла фиксации, за исключением первых двух символов.

Первые два символа хеша содержимого файла фиксации используются для именования папки, в которой содержится фиксация.

Таким образом, фиксация представляет историю.

Помимо Blob-объектов, деревьев и фиксаций существует еще специальный объект Tag, представляющий релиз проекта.

Объект Tag – это файл, содержащий сжатый заголовок плюс запись, состоящую из указателей на фиксацию проекта, автора, имя релиза и комментария, с именем, состоящим из хеша содержимого файла релиза, за исключением первых двух символов.

Первые два символа хеша содержимого файла релиза используются для именования папки, в которой содержится релиз.

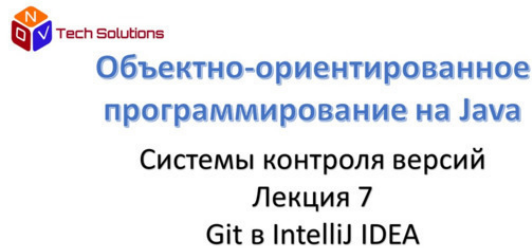
Далее есть папка logs, которая содержит историю изменений папки refs.

И папка hooks, которая предназначена для хранения скриптов, вызываемых до или после Git-команд.

Достоинства использования Git заключаются в том, что в таком распределенном репозитории нет выделенного центра, вся история дублируется в каждом локальном репозитории, нет необходимости в постоянном соединении, нет необходимости в управлении правами, поддерживает любой workflow рабочий процесс, каждый разработчик работает в своей песочнице, простое создание и слияние веток, локальные операции работают быстро, позволяет откладывать решение о коммите до момента push'a на сервер.

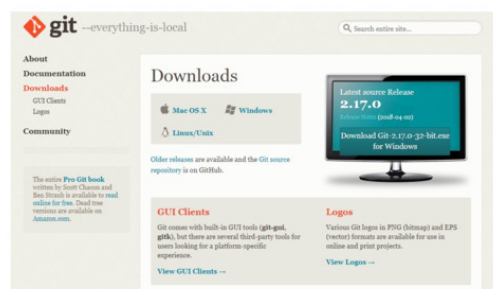
Git в IntelliJ IDEA

Для работы с Git в IntelliJ IDEA, сначала нужно установить клиента командной строки Git.

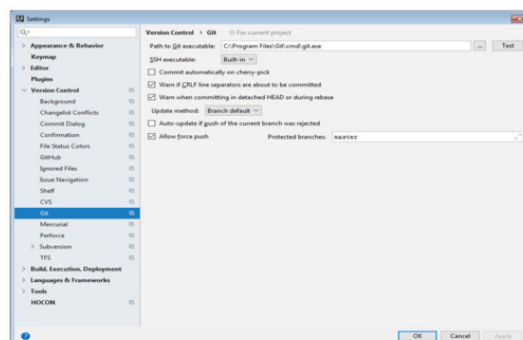


После этого нужно убедиться, что клиент добавлен в Git настройках IntelliJ IDEA.

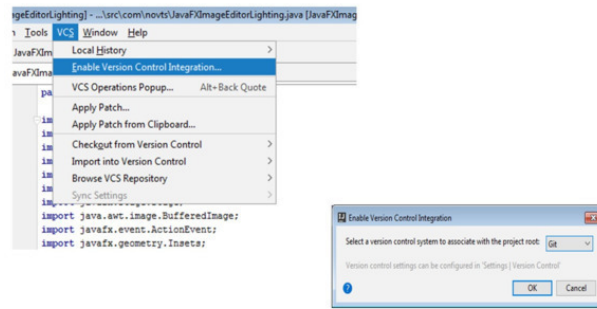
<https://git-scm.com/>



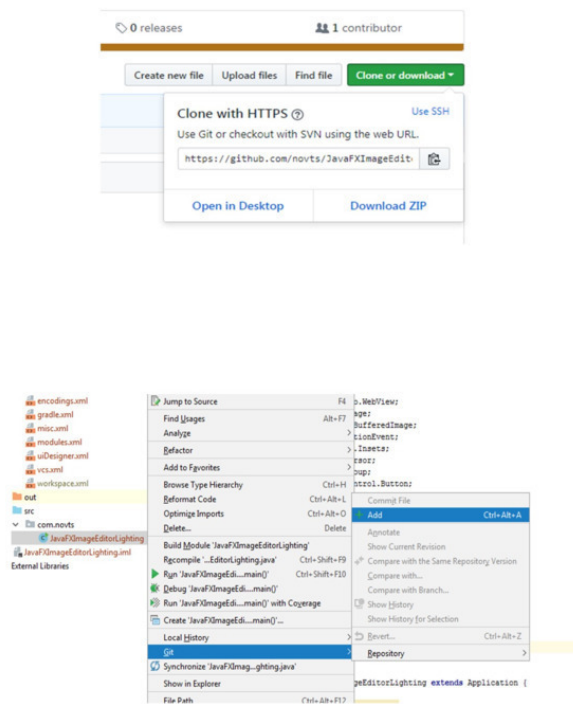
Создадим проект в IntelliJ IDEA и в меню VCS включим систему контроля версий для проекта Enable Version Control Integration.



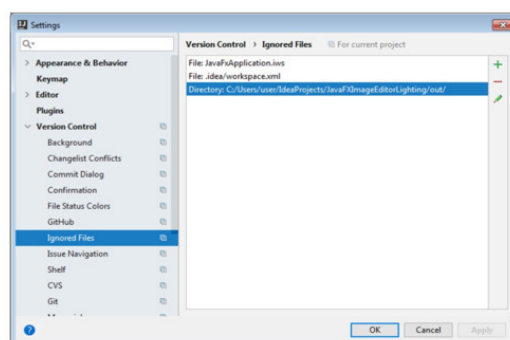
Также создадим проект в GitHub и получим его URL адрес.



С помощью меню Add добавим файлы проекта в репозиторий.

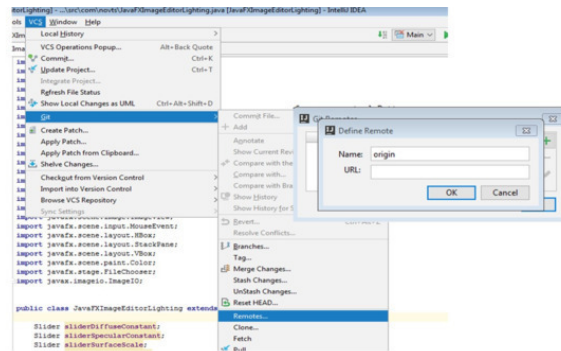


После этого отправим файлы в локальный репозиторий с помощью команды Commit. Не все файлы проекта можно добавить в репозиторий.



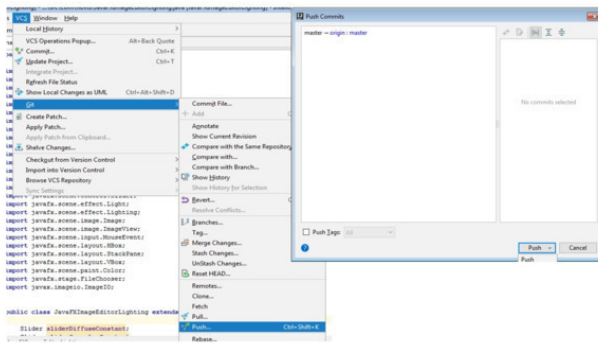
Список файлов и папок, которые игнорируются, регулируется разделом Ignored Files настроек.

Для добавления удаленного репозитория в меню VCS выберем Git и Remotes.



И введем URL адрес GitHub репозитория.

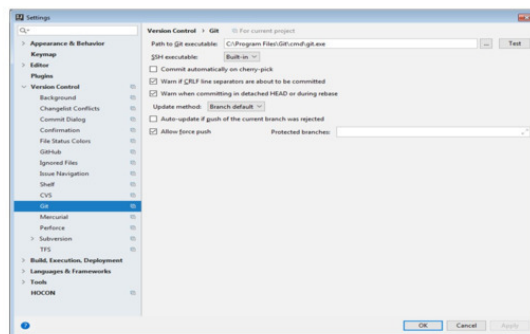
Для отправки файлов в удаленный репозиторий, выберем команду Push.



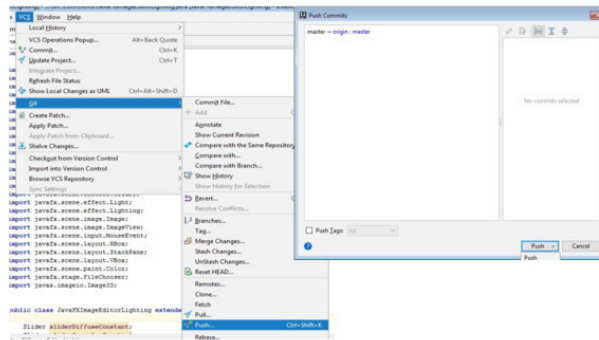
Так как в GitHub у нас создан файл README, а в локальном репозитории у нас его нет, простая команда Push не работает, так как структуры локального и удаленного репозитория разные.

Нужна команда Force push.

Для включения этой команды, в настройках нужен отмеченный флажок Allow force push без исключений, то есть с пустым полем.

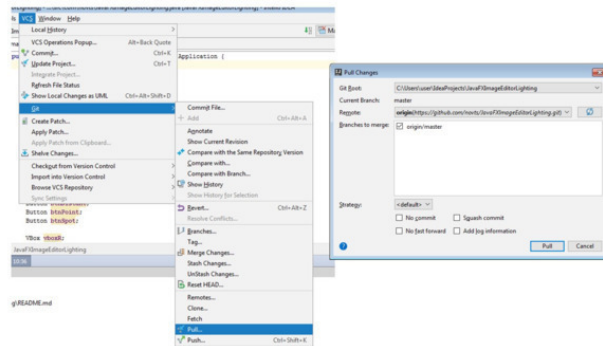


Далее выбираем команду Push и в выпадающем списке выбираем Force push.

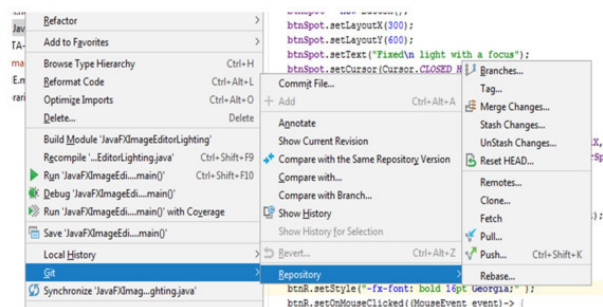


И закидываем файлы в удаленный репозиторий.

Для загрузки недостающих файлов из удаленного репозитория в локальный репозиторий выберем команду Pull.



В результате из удаленного репозитория будет закачан как минимум файл README. Теперь мы можем делать изменения в проекте.



Прежде чем вы делитесь результатами своей работы, используя команду Push, вам необходимо синхронизировать с удаленным репозиторием, чтобы убедиться, что ваша локальная копия проекта обновлена.

Вы можете сделать это с помощью команд Fetch, Pull и Update.

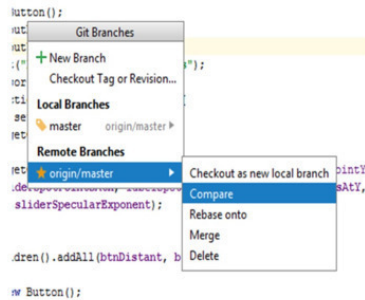
Когда вы используете команду Fetch, все новые данные от коммитов, которые были сделаны с момента последней синхронизации с удаленным репозиторием, загружаются в вашу локальную копию.

Эти новые данные не интегрируются в ваши локальные файлы, и изменения не применяются к вашему коду.

Полученные изменения сохраняются как удаленная ветвь, что дает вам возможность просмотреть их, прежде чем объединять их с вашими файлами.

Поскольку Fetch не влияет на вашу локальную среду разработки, это безопасный способ получить обновление всех изменений из удаленного репозитория.

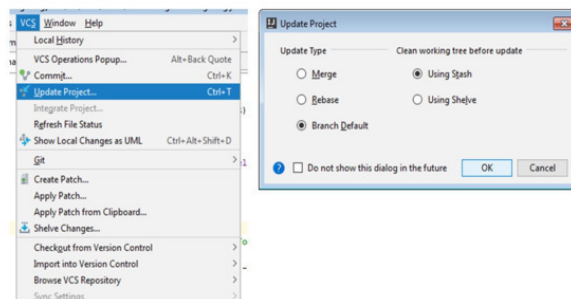
После получения изменений из удаленного репозитория с помощью команды Fetch, вы выбираете команду Branches и команду Compare удаленной ветки.



После просмотра изменений, вы можете произвести их слияние с локальным проектом, с помощью команды Merge.

При извлечении изменений из удаленного репозитория с помощью команды Pull, в отличие от команды Fetch, вы не только загружаете новые данные, но и сразу интегрируете их в свою локальную рабочую копию проекта.

Если у вас есть несколько корневых каталогов проекта или вы хотите получить изменения из всех ветвей, вы можете обновить проект с помощью команды Update Project.



Когда вы выполняете операцию обновления, IntelliJ IDEA извлекает изменения из всех корней и ветвей проекта и производит их слияние в вашу локальную рабочую копию, как если бы вы использовали команду pull.

При использовании команды Update Project, вы можете выбрать тип обновления.

Merge – выберите этот параметр, чтобы выполнить слияние во время обновления. Это эквивалентно командам fetch, затем merge.

Rebase – выберите эту опцию для выполнения rebase во время обновления. Это эквивалентно командам fetch, затем rebase.

Branch Default – применяется тип обновления, определенный в конфигурационном файле. git/config.

Также вы можете указать метод, который будет использоваться для сохранения изменений при очистке рабочей копии до обновления, чтобы ваши незафиксированные изменения можно было восстановить после завершения обновления.

Очистка рабочей копии – это фиксация коммит или сохранение всех текущих изменений.

Опции `stash` и `shelve` позволяют переключаться между различными задачами, которые остаются незавершенными, а затем к ним возвращаться без потери работы.

Они позволяют сохранять или откладывать ожидающие изменения.

Изменения `stash` очень похожи на изменения `shelve`.

Единственное различие заключается в том, как создаются и применяются патчи.

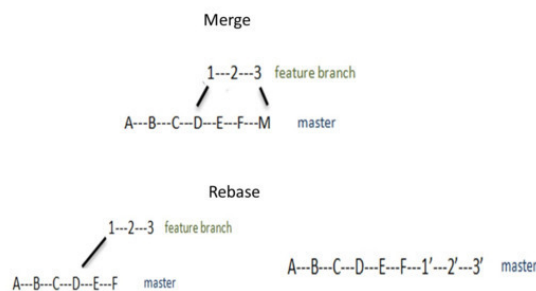
`Stash` патчи генерируются `Git` и могут применяться в `IntelliJ IDEA` или вне среды.

`Shelve` патчи генерируются `IntelliJ IDEA` и применяются в среде.

Патч – это файл изменений, который вы можете применить позже, вы также можете распространять его отдельно от проекта для выполнения изменений другими людьми.

В `Git` существует несколько способов интегрировать изменения из одной ветки в другую – это `Merge`, `Rebase` или применить отдельные фиксации из одной ветки к другой.

Когда вы запускаете `Merge`, изменения из ветки интегрируются в голову другой ветки.



`Git` создает новую фиксацию `M`, которая называется фиксацией слияния, которая возникает в результате объединения изменений одной ветки и другой ветки от точки, где две ветки разошлись.

Когда вы запускаете `Rebase`, вы применяете изменения, которые вы сделали в ветке, к основной ветке, применяя свои коммиты к голове основной ветки.

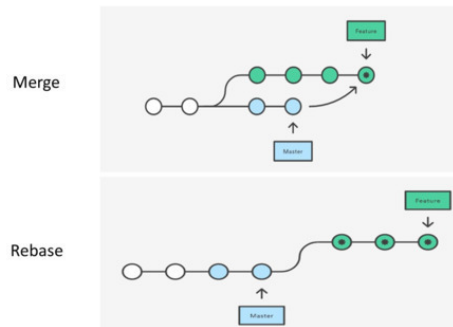
Первое, что нужно знать о `git rebase`, это то, что эта команда решает ту же проблему, что и `git merge`.

Обе эти команды предназначены для интеграции изменений из одной ветки в другую ветку – просто они делают это по-разному.

`Merge` слияние – это неразрушающая операция.

Существующие ветки не изменяются.

С другой стороны, это также означает, при разработке ветки, она будет иметь слияние каждый раз, когда вам нужно включить изменения основной ветки.



Если основная ветка очень активная, это может загрязнять историю функциональности другой ветки и затруднить понимание другими разработчиками истории проекта.

С другой стороны, rebase перезагрузка перемещает всю побочную ветку в конец ведущей ветки, эффективно включая все новые коммиты в ветку мастер.

Но вместо использования фиксации слияния, перезагрузка перезаписывает историю проекта, создавая новые коммиты для каждой фиксации в исходной ветке.

Главное преимущество перезагрузки заключается в том, что вы получаете гораздо более чистую историю проекта.

Во-первых, rebase устраняет ненужные коммиты слияния, необходимые для слияния Merge.

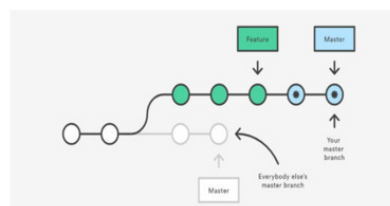
Во-вторых, перезагрузка также приводит к совершенно линейной истории проекта.

Однако такая перезапись истории проекта может быть потенциально катастрофической для вашего рабочего процесса совместной работы.

Золотое правило git rebase заключается в том, чтобы никогда не использовать эту команду для публичных веток.

Например, что произойдет, если вы выполните rebase ветки мастер на свою ветку.

Rebase переместит все коммиты из мастера в конец вашей ветки.



Проблема заключается в том, что это произошло только в вашем репозитории.

Все остальные разработчики все еще работают с мастером.

После того, как вы сделаете Rebase и получите свой мастер, Git подумает, что история вашего мастера расходится со всеми остальными мастерами.

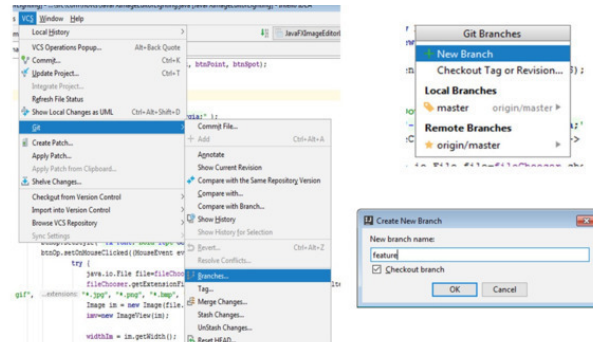
Теперь единственный способ синхронизировать разные мастера – это выполнить Merge, в результате чего добавляется дополнительное слияние и два набора коммитов, которые содержат одни и те же самые изменения.

Это очень запутывает историю проекта.

Поэтому обычно делается Rebase вашей ветки к мастеру после завершения разработки вашей ветки.

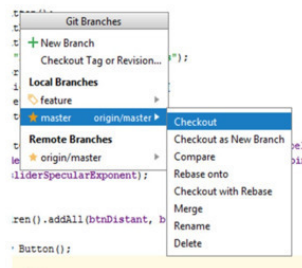
И результат посылается в удаленный репозиторий.

Для создания новой ветки выберите команду Branches и New Branch.



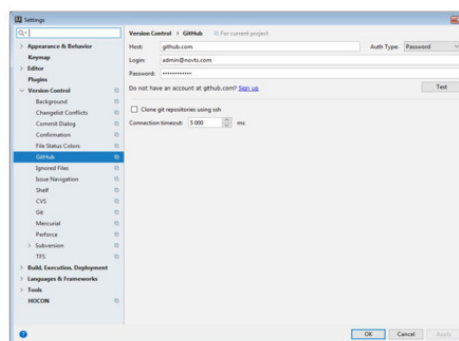
При этом команда Checkout branch автоматически переключит вас на новую ветку.

Переключиться обратно на основную ветку также можно командой Checkout.



Командой Delete можно удалить ветку.

IntelliJ IDEA позволяет управлять проектами GitHub, не выходя из среды разработки.



Чтобы получить данные из репозитория, размещенного в GitHub, вам необходимо зарегистрировать свою учетную запись GitHub в IntelliJ IDEA.

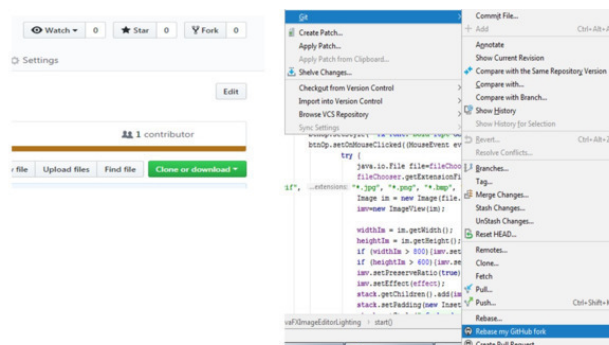
Вы также можете создать новую учетную запись в GitHub, не выходя из среды IDE.

В любом случае IntelliJ IDEA запоминает ваш логин и пароль, поэтому вам не нужно указывать свои учетные данные каждый раз, когда вы извлекаете данные из удаленного репозитория, или совершаете фиксации.

Команда Checkout from Version Control позволяет клонировать репозиторий из GitHub и создать на его основе новый проект.

Команда Import into Version Control и Share Project on GitHub создает удаленный репозиторий GitHub для вашего проекта.

Если вы хотите внести свой вклад в чужой проект, размещенный в GitHub, вы должны сначала создать вилку этого проекта, т. е. копию оригинального репозитория в GitHub в своем аккаунте с помощью команды Fork.

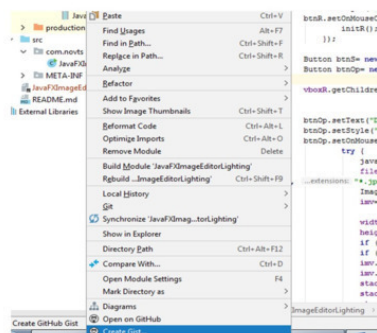


Далее вы вносите изменения в копию оригинального проекта с помощью локального репозитория.

И когда вы будете готовы поделиться результатами своей работы, вы делаете Rebase вилки с помощью команды Rebase my GitHub fork.

Далее вы создаете запрос pull request с помощью команды Create Pull Request, сообщая другим об изменениях, которые вы сделали в своей вилке, чтобы их можно было обсудить и интегрировать в базовую ветку.

Также в контекстном меню проекта, вы можете создать Гист, который позволяет делиться своим кодом.



С помощью гистов вы можете делиться фрагментами кода, целыми файлами или даже приложениями.

Вы также можете использовать гисты для сохранения и совместного использования консольного вывода при запуске, отладке или тестировании вашего кода.

Основы системы безопасности Java. Введение



Объектно-ориентированное программирование на Java

Основы системы безопасности Java

Лекция 1

Введение

Существует ли такая вещь, как абсолютная компьютерная безопасность?
Если одним словом, то нет.



Термин «безопасные системы» является неправильным, поскольку он подразумевает, что системы либо безопасны, либо нет.

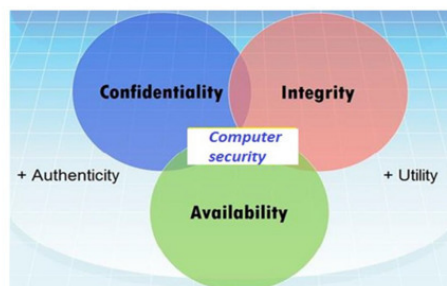
По правде говоря, безопасность – это компромисс.

Если есть в наличии неограниченные ресурсы, любая система безопасности может быть взломана.

В то время как все больше и больше ресурсов становятся доступными для злоумышленников, эти ресурсы все же остаются ограниченными.

Имея это в виду, системы должны проектироваться таким образом, чтобы затраты на их взлом значительно перевешивали потенциальную прибыль.

Атаки на компьютерную безопасность – можно в целом классифицировать как:



Атаки секретности – это попытки украсть конфиденциальную информацию либо путем использования слабых мест в криптографических алгоритмах, либо другими способами.

Это атаки целостности, когда предпринимаются попытки изменить информацию.

Это атаки доступности, когда предпринимаются попытки нарушить нормальную работу системы.

Например, атаки типа «отказ в обслуживании» или denial of service (DoS) атаки.

Для защиты от угроз безопасности существует множество механизмов.

Исторически, защитные механизмы включали в себя сооружение какой-либо стены или границы, и это обычно называется защитой периметра.

Довольно успешный пример защиты периметра – это брандмауэры, которые разделяют внутренние (приватные) и внешние (публичные) сети и обеспечивают центральную точку контроля для корпоративной политики.

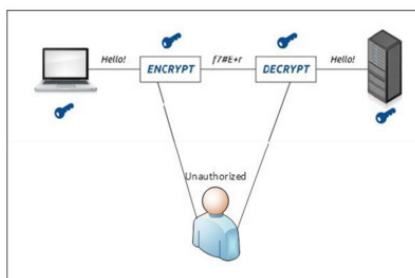
В качестве другого примера защиты служит виртуальная приватная сеть (VPN), которая обеспечивает тот же уровень безопасности, что и приватная сеть.

Задачами компьютерной безопасности является обнаружение и предотвращение атак и возможность восстановления систем.

Чтобы выполнить эти задачи, мы приходим к трем основным элементам, которые составляют компьютерную безопасность – это конфиденциальность, целостность и доступность.

Конфиденциальность – это процесс защиты данных от несанкционированного использования.

Проще говоря, это означает, что только предполагаемый получатель сообщения может получить доступ к нему.



Если вы обмениваетесь конфиденциальной информацией с кем-то еще, вы должны быть абсолютно уверены, что только предполагаемый получатель сообщения может понять смысл сообщения и, в случае, если он попадает в неправильные руки, сообщение становится бесполезным.

Конфиденциальность осуществляется с помощью какой-либо криптографической техники.

Целостность – это достоверность данных в системах или ресурсах с точки зрения предотвращения несанкционированных и неправильных изменений.

Как правило, целостность состоит из целостности данных, которая имеет отношение к содержимому данных, и аутентификации, которая связана с источником данных, так как информация должна быть правильной.

При этом должна соблюдаться неотказуемость, которая дает неопровержимые доказательства таких действий, как подтверждение происхождения данных для получателя или получение данных для отправителя.

Доступность – это возможность доступа к данным ресурса, когда это необходимо, поскольку такая информация имеет значение только в том случае, если авторизованные пользователи могут получить доступ в нужное время.

В настоящее время отказ в доступе к данным стал обычной атакой.

Политика безопасности направлена на контроль доступа к защищенным данным.

Важно, чтобы механизмы обеспечения безопасности были достаточно гибкими, чтобы обеспечить соблюдение политики.

Это называется поддержанием политики отдельно от механизма.

Хотя это решение может основываться на авторизации доступа к ресурсу на основе идентичности принципала, часто проще администрировать управление доступом на основе ролей.

Каждый принципал сопоставляется с уникальной ролью для контроля доступа.

Процесс аутентификации подтверждает личность пользователя.

И пользователь может быть программным объектом или человеком.

Принципалом является сторона, чья личность проверена.

И с принципалом связывается набор учетных данных.

Обычно аутентификация подтверждает идентичность с помощью некоторой секретной информации – например, с помощью пароля, который известен только пользователю и проверяющему.

Помимо паролей, для аутентификации используются передовые технологии, такие как смарт-карты или биометрические данные (печать пальцев, сканирование сетчатки и т. д.).

После установления подлинности доступ для принципала к ресурсам регулируется механизмами управления доступом.

Сетевой протокол аутентификации Kerberos – на основе ключей и шифрования – демонстрирует технологию ранней аутентификации.

Он использует временные метки, когда сеансы остаются действительными в течение определенного периода времени.

Для правильной работы Kerberos принципиально предполагает синхронизацию часов в распределенной системе.

Инфраструктура открытого ключа (PKI), представляет собой более общее решение для проверки идентичности.

И Java фреймворк Java Authentication and Authorization Service (JAAS) дополняет платформу Java возможностями аутентификации и контроля доступа.

JAAS является стандартным расширением Java SDK.

Java Platform, Enterprise Edition (J2EE) использует проверку подлинности на основе ролей для обеспечения соблюдения своих политик.

И в JEE разработчик бизнес-логики ограничивает доступ к определенным функциям, основываясь на ролях.

Платформа Java была разработана изначально с учетом обеспечения безопасности.

По своей сути, язык Java сам по себе безопасен по типам и обеспечивает автоматическую сборку мусора, повышая надежность кода приложения.

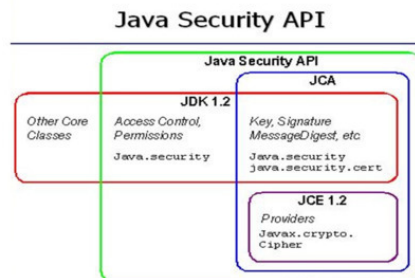
Механизм загрузки классов и механизм верификации гарантирует выполнение только легального кода Java.

Уже начальная версия платформы Java обеспечивала безопасную среду для запуска потенциально ненадежного кода, например, апплетов Java, загруженных из общедоступной сети.

По мере роста платформы и расширения ее диапазона развертывания, архитектура безопасности Java также развивалась для поддержки растущего набора сервисов.

Текущая архитектура безопасности включает в себя большой набор интерфейсов API, инструменты и реализации широко используемых алгоритмов, механизмов и протоколов безопасности.

Это обеспечивает разработчику всеобъемлющую систему безопасности для написания приложений, а также предоставляет пользователю или администратору набор инструментов для безопасного управления приложениями.



API-интерфейсы безопасности Java охватывают широкий диапазон областей.

И API-интерфейсы позволяют использовать многочисленные реализации алгоритмов и сервисов безопасности.

Сервисы реализуются в провайдерах, которые подключаются к платформе Java через стандартный интерфейс, что позволяет приложениям получать сервисы безопасности без необходимости знать что-либо об их реализации.

Это позволяет разработчикам сосредоточиться на том, как интегрировать безопасность в свои приложения, а не о том, как реально реализовать сложные механизмы безопасности.

Платформа Java включает в себя ряд провайдеров, которые реализуют основной набор сервисов безопасности.

Она также позволяет устанавливать дополнительных пользовательских провайдеров.

Это позволяет разработчикам расширять платформу новыми механизмами безопасности.

Язык Java был разработан, чтобы быть безопасным и простым в использовании.

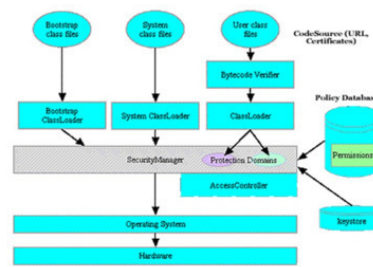
Он обеспечивает автоматическое управление памятью, сборку мусора и проверку диапазона в массивах.

Это уменьшает общую нагрузку на разработчиков, что приводит к меньшему количеству тонких ошибок программирования и более безопасному, более надежному коду.

Кроме того, язык Java определяет различные модификаторы доступа, которые могут быть назначены классам, методам и полям Java, что позволяет разработчикам ограничить доступ к их реализациям, если это необходимо.

В частности, язык определяет четыре различных уровня доступа: приватный, защищенный, публичный и, если не указано, пакетный.

Компилятор переводит Java-программы в машинное независимое представление байт-кода.



И вызывается верификатор байт-кода для обеспечения выполнения только легального байт-кода в среде выполнения Java.

Он проверяет, соответствует ли байт-код спецификации Java и не нарушает ли байт-код правил языка Java или ограничений пространства имен.

Верификатор также проверяет нарушения управления памятью, переполнение стека, а также недопустимые типы данных.

После того, как байт-код проверен, среда выполнения Java готовит его к выполнению.

Так как код Java может быть получен из любой точки сети, крайне важно просканировать код, чтобы убедиться, что он был создан надежным компилятором.

Верификатор байт-кода пытается доказать, что данный байт-код Java является легальным.

Короче говоря, верификатор подтверждает или отрицает, что файл класса соответствует спецификациям.

Хотя на слайде показано, что проверка байт-кода происходит, когда класс загружен, некоторые из этих проверок могут быть отложены до момента, когда байт-код будет выполняться.

Несмотря на то, что верификатор байт-кода выполняет важную задачу, он не очень интересен с точки зрения программирования, так как его поведение не может быть изменено программно.

Поведение может быть изменено только с помощью параметров командной строки.

ClassLoader, который загружает байт-код Java в JVM, является важным звеном в системе безопасности.

Он работает совместно с SecurityManager и контроллером доступа для обеспечения соблюдения правил безопасности.

Обратите внимание, что ClassLoader участвует в обеспечении некоторых решений по безопасности раньше, чем менеджер безопасности.

ClassLoader важен в модели безопасности Java, поскольку изначально только загрузчик классов знает определенную информацию о классах, которые были загружены в виртуальную машину. Только загрузчик классов знает, откуда был получен какой-то конкретный класс, и только загрузчик классов знает, был ли подписан какой-либо конкретный класс.

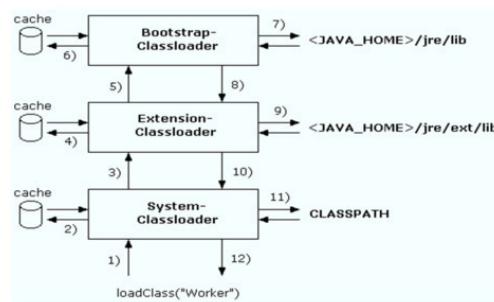
Для определения политики безопасности для Java-программы загрузчик класса должен координировать свою работу с менеджером безопасности и контроллером доступа.

ClassLoader, как указывает его имя, загружает классы в виртуальную машину.

Он также отвечает за концепцию пространств имен во время выполнения, которые создаются пакетами.

В пространствах имен одинаковые идентификаторы могут ссылаться на разные объекты.

Загрузка классов начинается с Bootstrap ClassLoader, называемого начальным загрузчиком классов и который обычно написан на нативном языке, он загружает загрузочные классы, зависящие от платформы.



Некоторые классы необходимы для JVM и среды выполнения.

Они называются системными классами, и они загружаются загрузчиком System ClassLoader.

Далее загружаются все остальные классы, найденные в переменной среды CLASSPATH.

Когда виртуальной машине Java требуется получить доступ к определенному классу, для класса требуется загрузчик класса.

Для загрузки и определения класса загрузчик классов выполняет следующие шаги:

Если загрузчик классов уже загрузил этот класс, он должен найти ранее определенный объект класса и немедленно вернуть этот объект.

Далее загрузчик должен проконсультироваться с менеджером безопасности, чтобы узнать, разрешен ли этой программе доступ к рассматриваемому классу.

Если это не так, генерируется исключение безопасности.

Файл класса считывается в массив байтов загрузчиком.

Байт-код пропускается через верификатор байт-кода.

Перед тем, как класс может быть использован, он должен быть разрешен – то есть, любые классы, на которые он ссылается, также должны быть найдены загрузчиком классов.

И данные для класса могут быть прочитаны из сети или файловой системы (или из любого другого местоположения, например, базы данных).

Существует ряд загрузчиков классов, которые используются в Java-программах.

И можно реализовать пользовательский ClassLoader как подкласс класса SecureClassLoader пакета java.security, чтобы обеспечить функции безопасности, отличные от тех, которые предлагаются стандартной моделью безопасности Java.

Поскольку код Java можно загружать по сети, исходный код и автор кода имеют решающее значение для поддержания безопасной среды.

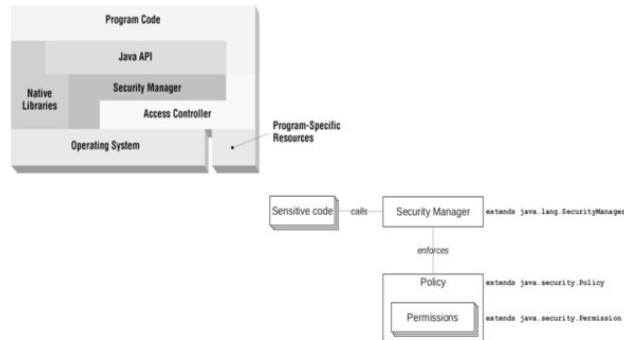
```
try{
    Class cls = Class.forName("com.Address");
    ProtectionDomain pd = cls.getProtectionDomain();
    CodeSource cs = pd.getCodeSource();
    URL url = cs.getLocation();
    System.out.println(url.getFile());           /E:/workspace/Example/target/classes/
} catch (Exception ex){
    ex.printStackTrace();
}
```

Объект java.security.CodeSource полностью описывает фрагмент кода.

CodeSource инкапсулирует происхождение кода, указанное как URL-адрес, и набор цифровых сертификатов, используемых для подписи кода.

На слайде показано как с помощью CodeSource получается происхождение кода.

Менеджер безопасности – это объект, который определяет политику безопасности для приложения.



Эта политика определяет действия, которые являются небезопасными или чувствительными.

И любые действия, не разрешенные политикой безопасности, вызывают исключение SecurityException.

Каждое приложение Java может иметь свой собственный объект менеджера безопасности.

По умолчанию приложение не имеет менеджера безопасности.

То есть среда выполнения Java не создает автоматически менеджера безопасности для каждого приложения Java.

Таким образом, по умолчанию приложение разрешает все операции, которые могут подпадать под ограничения безопасности.

Чтобы изменить это поведение по умолчанию, приложение должно создать и установить менеджер безопасности.

SecurityManager содержит несколько методов проверки.

Например, checkRead может определять доступ для чтения к файлу.

Метод checkPermission может проверить, имеет ли запрашиваемый доступ соответствующее разрешение на основе политики.

Этот метод связывается с контроллером доступа в реализации по умолчанию.

И контроллер доступа будет выбрасывать исключение, если запрашиваемое разрешение не может быть предоставлено.

Классы разрешений лежат в основе безопасности Java и представляют собой разрешения доступа к различным системным ресурсам, таким как файлы, сокеты и т. д.

Набор разрешений составляет настраиваемую политику безопасности.

Например, разрешение может быть предоставлено для чтения и записи файлов в определенном каталоге.

Классы разрешений являются аддитивными, так как они представляют собой разрешения, но не запрещения.

Можно явно разрешить чтение определенного файла, но не запретить чтение этого файла.

Классы разрешений – это подклассы абстрактного класса java.security.Permission.

Можно связывать разрешения с классами, однако более гибким решением будет группировать классы в домены безопасности и связывать разрешения с этими доменами.

Эта взаимосвязь между классами и разрешениями через домены безопасности обеспечивает гибкие механизмы реализации.

Многочисленные отображения разрешений на классы в совокупности называются политикой.

Файл политики используется для настройки политики для конкретной реализации.

Он может быть написан в простом текстовом редакторе или с использованием `policytool`, инструмента, входящего в комплект SDK.

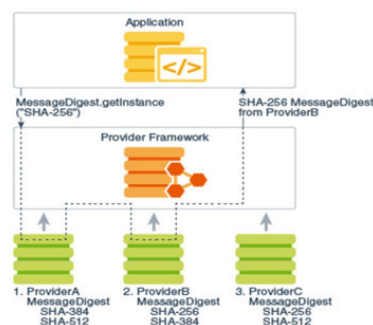
Класс `java.security.AccessController` используется для трех целей:

Чтобы решить, следует ли разрешать или запрещать доступ к критическому системному ресурсу в соответствии с действующей политикой безопасности.

Чтобы отметить код как привилегированный, что влияет на последующие разрешения доступа

Чтобы получить моментальный снимок текущего контекста вызова, так что решения по управлению доступом из другого контекста могут быть сделаны в отношении сохраненного контекста.

Платформа Java определяет набор API, охватывающих основные области безопасности, включая криптографию, инфраструктуру публичного ключа, аутентификацию, безопасную коммуникацию и контроль доступа.



Эти API-интерфейсы позволяют разработчикам легко интегрировать безопасность в свой код приложения.

Приложениям не нужно самостоятельно обеспечивать безопасность.

И приложения могут запрашивать сервисы безопасности у платформы Java.

Сервисы безопасности реализуются в провайдерах, которые подключаются к платформе Java через стандартный интерфейс.

Для обеспечения безопасности приложение может использовать различных независимых провайдеров.

И провайдеры взаимодействуют с различными приложениями.

Приложение не привязано к определенному провайдеру, и провайдер не привязан к конкретному приложению.

Платформа Java включает в себя ряд встроенных провайдеров, которые реализуют базовый набор широко используемых сервисов безопасности.

Тем не менее, приложения могут использовать новые стандарты, которые еще не реализованы в платформе, или на сторонние сервисы.

Платформа Java поддерживает установку провайдеров, которые реализуют такие сервисы.

Класс `java.security.Provider` инкапсулирует понятие поставщика безопасности на платформе Java.

```

java.security
Class Provider
java.lang.Object
  java.util.Dictionary<K,V>
    java.util.Hashtable<Object,Object>
      java.util.Properties
        java.security.Provider

All implemented interfaces:
Serializable, Cloneable, Map<Object,Object>
Direct Known Subclasses:
AuthProvider

MessageDigest md = MessageDigest.getInstance("SHA-256");
MessageDigest md =
  MessageDigest.getInstance("SHA-256", "ProviderC");

```

Provider.Service	getService(String type, String algorithm) Get the service describing this Provider's implementation of the specified type of this algorithm or alias.
Set<Provider.Service>	getServices() Get an unmodifiable Set of all services supported by this Provider.
protected void	putService(Provider.Service s) Add a service.

Он указывает провайдера и перечисляет сервисы безопасности, которые он реализует.

Когда запрашивается сервис безопасности, выбирается поставщик с наивысшим приоритетом, который реализует этот сервис.

Приложение использует метод `getInstance` для получения сервиса безопасности от базового провайдера.

Например, создание цифровой подписи сообщений представляет собой один из сервисов, который можно получить у провайдера.

И приложение вызывает метод `getInstance` класса `java.security.MessageDigest` для получения реализации конкретного алгоритма подписи сообщений, такого как SHA-256.

Приложение может по желанию запросить реализацию у конкретного провайдера, указав имя провайдера.

Менеджер безопасности



Объектно-ориентированное программирование на Java

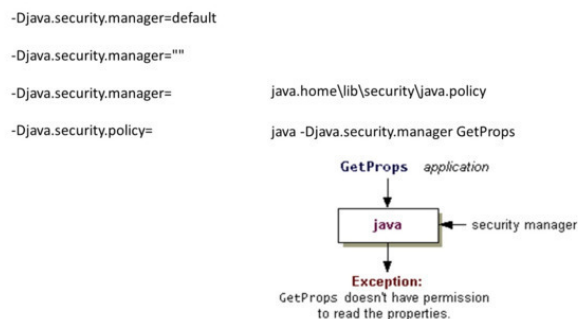
Основы системы безопасности Java Лекция 2 Менеджер безопасности, контроллер доступа и файл политики

Как уже было сказано, для защиты внешних для виртуальной машины Java ресурсов необходимо использовать менеджер безопасности.

С помощью менеджера безопасности можно определить внешние границы песочницы, в которой будет выполняться код приложения.

Среда выполнения Java автоматически не устанавливает менеджера безопасности при запуске приложения.

Чтобы включить менеджер безопасности Java для JVM и указать дополнительные файлы политик, которые вы хотите использовать в менеджере безопасности, вам необходимо настроить свойства системы безопасности для JVM.



Вы можете включить менеджер безопасности по умолчанию и указать подходящий файл политики с помощью системных свойств JVM.

Системное свойство `security.manager` указывает, что менеджер безопасности Java должен быть включен для JVM.

При этом среда выполнения Java загрузит файл политики по умолчанию.

Файл политики по умолчанию `java.policy` находится в папке `security` среды выполнения JRE.

Системное свойство `security.policy` описывает расположение дополнительных файлов политики, которые менеджер безопасности должен использовать для определения политики безопасности для JVM.

Напомним, что для каждого приложения при запуске создается свой экземпляр виртуальной машины.

Файл политики представляет собой текстовый файл в кодировке ASCII и может быть написан в текстовом редакторе или с помощью графической утилиты Policy Tool набора SDK,

который устраняет необходимость знать синтаксис файла политики, тем самым уменьшая количество ошибок.

```
keystore "some_keystore_url", "keystore_type", "keystore_provider";
keystorePasswordURL "some_password_url";

grant signedBy "signer_names", codeBase "URL",
    principal principal_class_name "principal_name",
    principal principal_class_name "principal_name",
    ... {

    permission permission_class_name "target_name", "action",
        signedBy "signer_names";
    permission permission_class_name "target_name", "action",
        signedBy "signer_names";
    ...
};
```

Политика безопасности сопоставляет `CodeSource`, который представляет происхождение кода и о котором мы говорили во введении, а также принципа, кто выполняет код, с наборами разрешений.

В Java, политика реализуется при помощи классов наследников абстрактного класса `java.security.Policy`.

И в Java есть только один наследник класса `Policy` – это `PolicyFile`.

Можно написать свой класс политики, который будет подклассом абстрактного класса `Policy` и будет реализовывать метод `getPermissions`, и, при необходимости, другие методы.

И этот пользовательский класс можно установить вместо реализации по умолчанию в файле свойств безопасности `java.security` в каталоге `security` среды выполнения, используя свойство `policy.provider`.

Файл политики может содержать одну запись `keystore` и несколько записей `grant`.

Хранилище ключей `keystore` – это база данных приватных ключей и связанных с ними цифровых сертификатов, которая может быть создана с помощью утилиты `keytool`.

Хранилище ключей, указанное в файле политики, используется для поиска публичных ключей подписантов, указанных в записях `grant` файла.

Здесь `some_keystore_url` указывает URL-адрес хранилища ключей, `some_password_url` указывает URL-адрес пароля хранилища ключей, `keystore_type` указывает тип хранилища ключей, а `keystore_provider` указывает провайдера хранилища ключей.

URL-адреса здесь могут быть абсолютными или указываться относительно местоположения файла политики.

Тип хранилища ключей определяет формат хранения и алгоритмы, используемые для защиты приватных ключей в хранилище ключей и целостности самого хранилища ключей.

По умолчанию, тип хранилища ключей – «JKS».

Источник кода, представленным объектом типа `CodeSource`, включает в себя не только местоположение (URL-адрес), из которого произошел код, но также ссылку на сертификаты, содержащие публичные ключи подписи кода.

Код также считается выполняемым принципом, подставленным объектом `Principal`, или группой принципов.

Каждая запись `grant` включает в себя одну или несколько записей разрешения, которым предшествуют необязательные значения `CodeBase`, `signedBy` и `principal`.

Значение `signedBy` указывает алиас сертификата, хранящегося в хранилище ключей.

Публичный ключ в этом сертификате используется для проверки цифровой подписи кода.

То есть вы предоставляете разрешение для кода, подписанного приватным ключом, который соответствует публичному ключу в записи хранилища ключей, указанной алиасом.

Использование принципа обеспечивается сервисом JAAS (Java Authentication and Authorization Service), который мы рассмотрим позже.

Если `principal` пара указана как одиночная строка в кавычках, она рассматривается как алиас хранилища ключей.

В этом случае запрашивается сертификат, и, если он найден, тогда имя принципа автоматически рассматривается как имя субъекта из сертификата.

Если сертификат не найден, вся запись `grant` игнорируется.

Значение `CodeBase` указывает местоположение источника кода.

То есть вы предоставляете разрешение на код из этого места.

Объект `CodeSource` является комбинацией `CodeBase` и подписантами.

Запись разрешения должна начинаться со слова `permission`.

```
grant {
    permission Foo "foobar", signedBy "FooSoft";
};

grant {
    permission java.io.FilePermission "C:\\users\\cathy\\foo.bat", "read";
};

grant signedBy "sysadmin", codeBase "file:/home/sysadmin/" {
    permission java.security.SecurityPermission "Security.insertProvider.*";
    permission java.security.SecurityPermission "Security.removeProvider.*";
    permission java.security.SecurityPermission "Security.setProperty.*";
};
```

Далее идет конкретное разрешение, такое как `java.io.FilePermission` или `java.lang.RuntimePermission`.

Поле `signedBy` в записи разрешения является необязательной.

Если оно присутствует, оно указывает на подписанное разрешение.

То есть сам класс разрешения должен быть подписан.

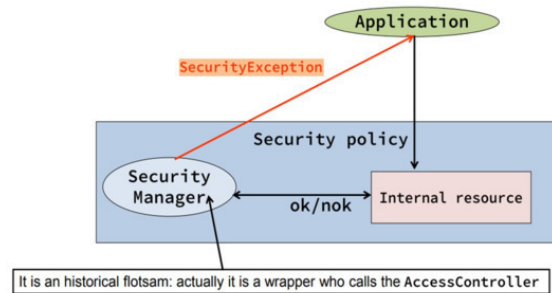
Здесь разрешение типа `Foo` предоставляется, если разрешение `Foo.class` было помещено в JAR-файл, а файл JAR был подписан приватным ключом, соответствующим публичному ключу в сертификате с алиасом `FooSoft`.

Когда указывается разрешение `java.io.FilePermission`, указывается путь к файлу, к которому разрешается доступ.

Ниже запись `grant` указывает, что только код, который удовлетворяет определенным условиям, может вызвать методы в классе безопасности для добавления или удаления провайдеров, или для установки свойств безопасности.

Этот код должен быть загружен из подписанного файла JAR, который находится в указанном каталоге, и подпись файла JAR может быть проверена с использованием публичного ключа, на который ссылается алиас «`sysadmin`» в хранилище ключей.

Перед выполнением операции менеджер безопасности `SecurityManager` определяет, может ли она выполняться в контексте безопасности.



Метод менеджера безопасности `checkPermission` определяет, должен ли быть предоставлен или запрещен запрос доступа, указанный разрешением в аргументе метода.

По умолчанию этот метод вызывает метод `checkPermission` контроллера доступа `AccessController`, который уже обращается к файлу политики.

Таким образом работает связка – менеджер безопасности, контроллер доступа и политика безопасности.

Основной объект, которым управляет контроллер доступа, является объектом разрешения – экземпляром класса `java.security.Permission`.

```

java.security
Class Permission
java.lang.Object
  java.security.Permission
All Implemented Interfaces:
  Serializable, Guard
Direct Known Subclasses:
  AllPermission, BasicPermission, FilePermission, MBeanPermission, PrivateCredentialPermission, ServicePermission, SocketPermission,
  UnresolvedPermission, URLPermission

java.security
Class Permissions
java.lang.Object
  java.security.PermissionCollection
  java.security.Permissions
All Implemented Interfaces:
  Serializable

public final class Permissions
  extends PermissionCollection
  implements Serializable
  
```

Сам класс `Permission` является абстрактным классом, который представляет собой конкретную операцию.

Объект разрешения может отражать две вещи.

Когда он связан с классом через источник кода и домен безопасности, объект разрешения представляет фактическое разрешение, предоставленное этому коду.

В противном случае объект разрешений позволяет спросить, есть ли конкретное разрешение.

Например, если мы создадим объект разрешения, представляющий доступ к файлу, владение этим объектом не означает, что у нас есть разрешение на доступ к файлу.

Скорее, владение объектом позволяет спросить, есть ли разрешение на доступ к файлу.

Экземпляр класса `Permission` представляет одно конкретное разрешение.

Набор разрешений – например, все разрешения, предоставляемые классам, подписанным конкретным подписантом, – представлен экземпляром класса `java.security.Permissions`.

Разрешения `Permission` имеют три свойства:

Все разрешения имеют базовый тип, который определяет, к чему разрешение относится.

Например, объект разрешения доступа к файлу будет иметь тип `FilePermission`.

И все разрешения имеют имя, которое идентифицирует конкретный объект, к которому относится разрешение.

Например, `FilePermission` имеет имя, которое является именем файла для доступа.

И эти имена должны войти в файл политики.

Некоторые разрешения несут с собой одно или несколько действий.

Наличие этих действий зависит от семантики конкретного типа разрешения.

Например, объект разрешения для доступа к файлу имеет список действий, которые включают в себя чтение, запись и удаление.

Разрешения могут выполнять две роли.

Первое, они позволяют Java API согласовывать доступ к ресурсам, файлам, сокетам и т. д.

Эти разрешения являются встроенными, и, например, вы можете создать объект, который представляет разрешение на чтение определенного файла, но вы не можете создать объект, который представляет разрешение на копирование определенного файла, так как действие копирования неизвестно в классе `FilePermission`.

С другой стороны, вы можете создавать произвольные свои разрешения для использования в своих собственных программах и полностью определять как имена в этих разрешениях, так и действия, которые должны применяться.

Например, если вы пишете программу расчета заработной платы, вы можете создать свой собственный класс разрешений, в котором используется соглашение о том, что имя разрешения является сотрудником, и вы можете определить, что действия в разрешении – это просмотр и обновление.

Затем вы можете использовать это разрешение вместе с контроллером доступа, чтобы позволить сотрудникам просматривать собственные данные о заработной плате и разрешать менеджерам изменять данные о заработной плате для сотрудников.

Java API предоставляет набор стандартных разрешений, каждое из которых реализуется как класс.

```
grant {
  permission java.io.FilePermission "c:\\temp\\foo", "read,write,delete";
  permission java.net.SocketPermission "***", "connect,accept,listen";
  permission java.util.PropertyPermission "java.version", "read";
  permission java.lang.RuntimePermission "stopThread";
  permission java.awt.AWTPermission "accessClipboard";
  permission java.net.NetPermission "***";
  permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
  permission java.security.AllPermission;
}
```

`FilePermission` представляет разрешения для доступа к файлам.

Допустимые действия для этого разрешения – это читать, записывать, удалять и выполнять.

`SocketPermission` представляет разрешения для взаимодействия с сетевыми сокетами.

Здесь имя сокета – это имя хоста и порт.

Допустимые действия для этого разрешения – это принимать входящее соединение с определенного хоста, соединиться с хостом, прослушивать и получить IP-адрес для конкретного хоста.

`PropertyPermission` представляет разрешения для доступа к свойствам Java.

Действия для этого класса – это читать и записывать.

`RuntimePermission` представляет разрешения для доступа к среде выполнения Java – по существу, разрешения для выполнения любой из операций, инкапсулированных классом `Runtime`, включая большинство операций с потоками.

Это разрешение не имеет связанных действий – у вас либо есть разрешение на выполнение этих операций, либо нет.

AWTPermission представляет разрешения для доступа к определенным ресурсам окон.

В этом классе есть три условных имени: showWindowWithoutWarningBanner, accessClipboard и accessEventQueue.

И нет действий, связанных с этим классом.

NetPermission представляет разрешения для взаимодействия с двумя разными классами.

Первый класс – это Authenticator, реализации этого класса предоставляют HTTP-аутентификацию для защищенных паролем веб-страниц.

Допустимыми именами, связанными с этим классом, являются setDefaultAuthenticator и requestPasswordAuthentication.

Кроме того, этот класс инкапсулирует различные разрешения, связанные с URL.

Разрешение указать обработчик потока в URL-классе называется namedStreamHandler.

И нет никаких связанных действий с этим разрешением.

AllPermission представляет собой разрешение на выполнение любой операции.

Предоставление такого разрешения явно опасно; это разрешение обычно предоставляется только классам Java API и классам в Java-расширениях.

У этого класса нет имени или действий.

Если вам нужно реализовать свой собственный класс разрешений, можно использовать класс BasicPermission.

```
import java.security.*;

public final class HighScorePermission extends BasicPermission
{
    public HighScorePermission(String name)
    {
        super(name);
    }

    // note that actions is ignored and not used,
    // but this constructor is still needed
    public HighScorePermission(String name, String actions)
    {
        super(name, actions);
    }
}

SecurityManager sm = System.getSecurityManager();
if (sm != null) {
    sm.checkPermission(
        new HighScorePermission(gameName));
}

keystore "keystore";

grant SignedBy "terry" {
    permission
    com.scores.HighScorePermission
    "ExampleGame", signedBy "chris";
};
```

Этот класс реализует базовое разрешение – то есть разрешение, которое не имеет действий.

Класс BasicPermission является абстрактным, хотя он не содержит абстрактных методов и полностью реализует все абстрактные методы класса Permission.

Следовательно, конкретная реализация BasicPermission должна содержать только конструктор для вызова правильного конструктора суперкласса, поскольку в классе BasicPermission не существует конструктора по умолчанию.

Таким образом, для реализации своего разрешения, вам нужно реализовать подкласс класса BasicPermission.

Далее указать разрешение в файле политики.

И в коде получить менеджер безопасности по умолчанию с помощью метода System.getSecurityManager и проверить наличие у приложения данного разрешения в файле политики.

И наконец, вы запускаете приложение командой java с опцией security.manager, чтобы включить менеджер безопасности, и опцией security.policy, где вы указываете файл политики.

Когда JVM запускается, она сначала проверяет, включен ли `SecurityManager`, проверяя системное свойство `java.security.manager`, и, если он включен, тогда будет создан экземпляр класса `SecurityManager` и он будет использоваться для проверки разных разрешений.

```
//Включение
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new SecurityManager());
}

//Выключение
SecurityManager sm=System.getSecurityManager();
if(sm!=null){
    System.setSecurityManager(null);
}

permission java.lang.RuntimePermission "setSecurityManager";
```

По умолчанию диспетчер безопасности выключен, но есть несколько способов включить `SecurityManager`.

Первый способ, как мы уже видели, когда мы запускаем программу, мы можем указать опцию `security.manager`.

Это самый распространенный способ включить `SecurityManager`.

`security.manager` – это системное свойство, и вы можете использовать вызов метода `System.getProperties`, чтобы проверить, установлено ли это системное свойство.

Вы можете подумать, что можете использовать `System.setProperty`, чтобы включить `SecurityManager`.

Однако это не работает.

Это системное свойство проверяется при запуске JVM.

Если мы установим это свойство программно, это не будет иметь никакого эффекта, поскольку JVM уже запустилась и проверила это свойство системы.

Второй способ, мы можем включить `SecurityManager` программно с помощью метода `setSecurityManager`.

Этот код будет работать только в том случае, если вы также указали соответствующее разрешение в файле политики.

Теперь, вопрос, можно ли программно использовать разрешения без использования файла политики?

```
public class MyPolicy extends Policy
{
    @Override
    public PermissionCollection getPermissions(CodeSource codesource)
    {
        Permissions p = new Permissions();
        p.add(new AllPermission());
        return p;
    }
}

Policy.setPolicy(new MyPolicy());
```

Для этого нужно расширить класс `java.security. Policy` и переопределить его метод `getPermissions` вернув в нем набор разрешений `Permissions`.

После этого установить эту политику методом `Policy.setPolicy`.

Чтобы написать свой собственный менеджер безопасности, вы должны создать подкласс класса `SecurityManager`.

```
import java.io.*;

class PasswordSecurityManager extends SecurityManager {
    private String password;
    PasswordSecurityManager(String password) {
        super();
        this.password = password;
    }
    private boolean accessOK() {
        int c;
        DataInputStream dis = new DataInputStream(System.in);
        String response;

        System.out.println("What's the secret password?");
        try {
            response = dis.readLine();
            if (response.equals(password))
                return true;
            else
                return false;
        } catch (IOException e) {
            return false;
        }
    }

    public void checkRead(FileDescriptor fileDescriptor) {
        if (!accessOK())
            throw new SecurityException("Not a Chance!");
    }
    public void checkRead(String filename) {
        if (!accessOK())
            throw new SecurityException("No Way!");
    }
    public void checkRead(String filename, Object exContext) {
        if (!accessOK())
            throw new SecurityException("Forget It!");
    }
    public void checkWrite(FileDescriptor fileDescriptor) {
        if (!accessOK())
            throw new SecurityException("Not!");
    }
    public void checkWrite(String filename) {
        if (!accessOK())
            throw new SecurityException("Not Even!");
    }
}
```

Подкласс `SecurityManager` будет переопределять различные методы `SecurityManager`, чтобы настроить проверки, необходимые в вашем приложении Java.

В этом примере создается менеджер безопасности, который ограничивает чтение и запись файловой системы.

Чтобы получить разрешение от менеджера безопасности, метод, который открывает файл для чтения, вызывает один из методов `checkRead SecurityManager`.

Аналогично, метод, который открывает файл для записи, вызывает один из методов `checkWrite SecurityManager`.

Если диспетчер безопасности одобряет операцию, тогда метод `check` возвращает, в противном случае метод `checkXXX` выбрасывает исключение `SecurityException`.

Политика, реализованная в этом примере, запрашивает у пользователя пароль, когда приложение пытается открыть файл для чтения или для записи.

Центральным здесь является метод `accessOK`.

Этот метод запрашивает у пользователя пароль и проверяет его.

Если пользователь вводит правильный пароль, метод возвращает `true`, в противном случае он возвращает `false`.

Все методы `check` менеджера безопасности вызывают метод `accessOK`, чтобы запросить у пользователя пароль.

Если пароль не верный, тогда метод `check` выбрасывает исключение `SecurityException`.

После того, как мы создали подкласс `SecurityManager`, его можно установить в качестве текущего менеджера безопасности для приложения Java.

```
try {
    System.setSecurityManager(new PasswordSecurityManager("passw"));
} catch (SecurityException se) {
    System.out.println("SecurityManager already set!");
}
```

Это можно сделать с помощью метода `setSecurityManager` класса `System`.

Вы можете установить диспетчер безопасности для своего приложения только один раз.

Другими словами, ваше Java-приложение может вызывать `System.setSecurityManager` только один раз за всю свою работу.

Любая последующая попытка установить диспетчера безопасности в Java-приложении приведет к выбросу исключения `SecurityException`.

Реализация менеджеров безопасности основана на контроллере доступа.

java.security
Class ProtectionDomain
java.lang.Object
java.security.ProtectionDomain

Constructors

Constructor and Description

`ProtectionDomain(CodeSource codesource, PermissionCollection permissions)`
Creates a new ProtectionDomain with the given CodeSource and Permissions.

`ProtectionDomain(CodeSource codesource, PermissionCollection permissions, ClassLoader classloader, Principal[] principals)`
Creates a new ProtectionDomain qualified by the given CodeSource, Permissions, ClassLoader and array of Principals.

Methods

Modifier and Type	Method and Description
ClassLoader	<code>getClassLoader()</code> Returns the ClassLoader of this domain.
CodeSource	<code>getCodeSource()</code> Returns the CodeSource of this domain.
PermissionCollection	<code>getPermissions()</code> Returns the static permissions granted to this domain.
Principal[]	<code>getPrincipals()</code> Returns an array of principals for this domain.
boolean	<code>implies(Permission permission)</code> Check and see if this ProtectionDomain implies the permissions expressed in the Permission object.
String	<code>toString()</code> Convert a ProtectionDomain to a String.

А контроллер доступа в свою очередь основан на четырех концепциях, каждая из которых связана, по крайней мере, с одним классом:

Это источник кода, связанный с классом `java.security.CodeSource`

Это разрешения, связанные с классом `java.security.Permission`

Это политики, связанные с классом `java.security.Policy`

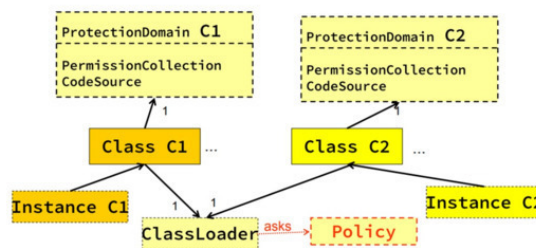
И это домены безопасности, связанные с классом `java.security.ProtectionDomain`.

Все эти концепции мы разобрали, за исключением домена безопасности.

Домен безопасности группирует объект `CodeSource` и разрешения.

В терминах файла политики, домен безопасности – это одна запись `grant`.

Каждый класс в виртуальной машине может принадлежать одному и только одному домену безопасности, который устанавливается загрузчиком класса, когда класс определяется.



Таким образом, класс `ProtectionDomain` содержит два класса – `PermissionCollection` для хранения набора разрешений и `CodeSource` для хранения информации о местоположении кода в виде URL-адреса и информации о подписи в классе `SignedBy`, который в цепочке содержит массив объектов `Certificate`.

Код, полученный от одного и того же источника и подписанный теми же подписантами, находится в одном и том же домене безопасности.

И метод `checkPermission` проверяет права доступа, связанные с объектами `ProtectionDomain`, которые установлены для текущего контекста.

Домен безопасности можно получить из объекта `Class`, после чего можно использовать его методы `getClassLoader`, `getCodeSource`, `getPermissions` и `getPrincipals`.

```
try {
    Class cls = Class.forName("ClassDemo");
    // returns the name of the class
    System.out.println("Class = " + cls.getName());

    // returns the ProtectionDomain of this class.
    ProtectionDomain p = cls.getProtectionDomain();
    System.out.println(p);

} catch (ClassNotFoundException ex) {
    System.out.println(ex.toString());
}
```

Вывод:

```
Class = ClassDemo
ProtectionDomain (file:/C:/Program%20Files/Java/jdk1.6.0_06/bin/ <no signer
certificates>)
sun.misc.Launcher$AppClassLoader@11b86e7
<no principals>
java.security.Permissions@3e25a5 {
  (java.io.FilePermission "C:\Program Files\Java\jdk1.6.0_06\bin\*.read")
  (java.lang.RuntimePermission "exitVM")
}
```


Привилегированные блоки



Tech Solutions

Объектно-ориентированное программирование на Java

Основы системы безопасности Java

Лекция 3

Привилегированные блоки

Мы уже рассмотрели, как контролировать и предоставлять доступ к чувствительным операциям с помощью менеджера безопасности и политики, содержащей разрешения.

Теперь, что, если нам нужно дать некоторому классу временное разрешение, которое он изначально не имеет?

Для этого можно использовать класс `AccessController` и его методы `doPrivileged`.

Напомним, что домен безопасности инкапсулирует экземпляр `CodeSource` и разрешения, предоставленные этому коду.

java security

Class AccessController

java.lang.Object
java.security.AccessController

public final class AccessController
extends Object

Modifier and Type	Method and Description
static void	<code>checkPermission(Permission perm)</code> Determines whether the access request indicated by the specified permission should be allowed or denied, based on the current <code>AccessControlContext</code> and security policy.
static <T> T	<code>doPrivileged(PrivilegedAction<T> action)</code> Performs the specified <code>PrivilegedAction</code> with privileges enabled.
static <T> T	<code>doPrivileged(PrivilegedAction<T> action, AccessControlContext context)</code> Performs the specified <code>PrivilegedAction</code> with privileges enabled and restricted by the specified <code>AccessControlContext</code> .
static <T> T	<code>doPrivileged(PrivilegedExceptionAction<T> action)</code> Performs the specified <code>PrivilegedExceptionAction</code> with privileges enabled.
static <T> T	<code>doPrivileged(PrivilegedExceptionAction<T> action, AccessControlContext context)</code> Performs the specified <code>PrivilegedExceptionAction</code> with privileges enabled and restricted by the specified <code>AccessControlContext</code> .
static <T> T	<code>doPrivilegedWithCombiner(PrivilegedAction<T> action)</code> Performs the specified <code>PrivilegedAction</code> with privileges enabled.
static <T> T	<code>doPrivilegedWithCombiner(PrivilegedExceptionAction<T> action)</code> Performs the specified <code>PrivilegedExceptionAction</code> with privileges enabled.
static AccessControlContext	<code>getContext()</code> This method takes a "snapshot" of the current calling context, which includes the current Thread's inherited <code>AccessControlContext</code> , and places it in an <code>AccessControlContext</code> object.

Таким образом, классы, подписанные одним и теми же ключами и с одного и того же URL-адреса, помещаются в один и тот же домен, и класс может принадлежать одному и только одному домену безопасности.

При этом классы, подписанные одними и теми же ключами и с одного и того же URL-адреса, но загруженные отдельными экземплярами загрузчика классов, помещаются в разные домены.

И классы, имеющие одинаковые разрешения, но имеющие разные `CodeSource`, принадлежат к разным доменам.

Все классы, поставляемые вместе с JDK, загружаются со всеми возможными разрешениями.

Большинство этих классов помещаются в уникальный системный домен.

Кроме того, загрузчик классов расширения загружает код из JAR-файлов, содержащихся в каталоге `ext JRE`, в отдельные домены, так как этот код имеет уникальные URL-адреса, и эти домены отделены от уникального системного домена, зарезервировано для классов, поставляемых с JDK.

Теперь, всякий раз, когда делается попытка доступа к чувствительному ресурсу, весь код, пройденный потоком выполнения до этой точки, должен иметь разрешение для доступа к этому ресурсу, если только какой-либо код в потоке не был отмечен как привилегированный.

То есть, предположим, что проверка доступа происходит в потоке выполнения, который имеет цепочку из нескольких вызывающих.

То есть происходит вызов нескольких методов, которые потенциально могут пересечь границы доменов защиты.

Когда метод `AccessController.checkPermission` вызывается последним вызывающим, алгоритм для решения, разрешать или запрещать доступ, выглядит следующим образом:

Если код для любого вызывающего в цепочке вызовов не имеет разрешения, тогда генерируется исключение `AccessControlException`, если только вызывающий код не отмечен как привилегированный, и все стороны, впоследствии вызываемые этим кодом, имеют указанное разрешение.

Напомним, что метод `AccessController.checkPermission` обычно вызывается косвенно с помощью методов `SecurityManager`.

Маркировка как привилегированный позволяет куску доверенного кода временно разрешать доступ к ресурсам.

Это необходимо в некоторых ситуациях.

Например, для приложения не разрешен прямой доступ к файлам, содержащим шрифты, но системная утилита для отображения документа должна получать эти шрифты от имени пользователя.

И эта системная утилита должна стать привилегированной для получения шрифтов.

Таким образом, если предположим, метод требует какого-либо разрешения, вызывается метод `checkPermission` `SecurityManager`, он в свою очередь вызывает метод `AccessController.checkPermission`, который выполняет проверку для каждой записи стека вызовов.

И если домен вызывающего не содержит нужного разрешения, выбрасывается исключение.

Если же вызывающий вызывал метод `doPrivileged` без контекста, он выполняется и возвращает.

Если же вызывающий вызывал метод `doPrivileged` с контекстом, он проверяет контекст и возвращает, если контекст содержит разрешение, иначе выбрасывается исключение.

Этот контекст определяется классом `AccessControlContext`.

`AccessControlContext` в свою очередь содержит метод `checkPermission`, который определяет, разрешен или запрещен запрос доступа, указанный разрешением, на основе действующей политики безопасности и контекста.

Вызывающий может быть отмечен как «привилегированный» с помощью вызова метода `doPrivileged`.

```
somemethod() {
    ...normal code here...
    AccessController.doPrivileged(new PrivilegedAction<Void>() {
        public Void run() {
            // privileged code goes here, for example:
            System.loadLibrary("awt");
            return null; // nothing to return
        }
    });
    ...normal code here...
}
```

Помимо контекста, метод `doPrivileged` может принимать в качестве параметра две категории действий – `PrivilegedAction` и `PrivilegedExceptionAction`.

Разница между ними заключается в том, что `PrivilegedAction` не будет генерировать исключение, даже если в методе `run` будет выбрасываться проверяемое исключение, в то время как `PrivilegedExceptionAction` будет генерировать исключение.

`PrivilegedAction` – это интерфейс с одним методом, который называется `run`.

В приведенном выше примере показано создание реализации этого интерфейса, при этом предоставляется конкретная реализация метода `run`.

Когда выполняется вызов `doPrivileged`, ему передается экземпляр реализации `PrivilegedAction`.

Метод `doPrivileged` вызывает метод `run` реализации `PrivilegedAction` после включения привилегий, и возвращает возвращаемое значение метода `run`, как возвращаемое значение метода `doPrivileged`, которое в этом примере отсутствует.

В этом примере показан возврат значения методом `run` и соответственно методом `doPrivileged`.

```
somemethod() {
    ...normal code here...
    String user = AccessController.doPrivileged(
        new PrivilegedAction<String>() {
            public String run() {
                return System.getProperty("user.name");
            }
        });
    ...normal code here...
}
```

Если действие, выполняемое в методе `run`, может выбросить проверяемое исключение, то есть исключение, которое должно быть указано ключевым словом `throws` метода, тогда нужно использовать интерфейс `PrivilegedExceptionAction` вместо интерфейса `PrivilegedAction`.

```
somemethod() throws FileNotFoundException {
    ...normal code here...
    try {
        FileInputStream fis = AccessController.doPrivileged(
            new PrivilegedExceptionAction<FileInputStream>() {
                public FileInputStream run() throws FileNotFoundException {
                    return new FileInputStream("someFile");
                }
            });
    } catch (PrivilegedActionException e) {
        // e.getException() should be an instance of FileNotFoundException,
        // as only "checked" exceptions will be "wrapped" in a
        // PrivilegedActionException.
        throw (FileNotFoundException) e.getException();
    }
    ...normal code here...
}
```

Если проверяемое исключение выбрасывается во время выполнения метода `run`, оно автоматически оборачивается в исключение `PrivilegedActionException`, как показано в этом примере.

Мы можем ограничить привилегии, создав объект `AccessControlContext` и передав его в метод `doPrivileged`.

```
private static final AccessControlContext NOPERMS_ACC;
private static final AccessControlContext PERMS_ACC;

static {
    Permissions permsNP = new Permissions();
    ProtectionDomain[] pdNP = { new ProtectionDomain(null, permsNP) };
    NOPERMS_ACC = new AccessControlContext(pdNP);

    Permissions permsP = new Permissions();
    permsP.add(new PropertyPermission("java.home", "read"));
    ProtectionDomain[] pdP = { new ProtectionDomain(null, permsP) };
    PERMS_ACC = new AccessControlContext(pdP);
}
```

Здесь мы создаем два контекста.

Один без всяких разрешений, и другой контекст с разрешением читать системное свойство.

Если мы вызовем метод `doPrivileged` без всякого контекста, метод будет выполнен и прочитает системное свойство без всяких проблем.

```
System.out.println(System.getSecurityManager());
AccessController.doPrivileged(
    new PrivilegedAction<Boolean>(){
        public Boolean run(){
            System.out.println(System.getProperty("java.home"));
            return Boolean.TRUE;
        }
    }
);

AccessController.doPrivileged(
    new PrivilegedAction<Boolean>(){
        public Boolean run(){
            System.out.println(System.getProperty("java.home"));
            return Boolean.TRUE;
        }
    }, PERMS_ACC
);

AccessController.doPrivileged(
    new PrivilegedAction<Boolean>(){
        public Boolean run(){
            System.out.println(System.getProperty("java.home"));
            return Boolean.TRUE;
        }
    }, NOPERMS_ACC
);
```

Если мы вызовем метод `doPrivileged` с контекстом без разрешений, будет выброшено исключение.

И если мы вызовем метод `doPrivileged` с контекстом с конкретным разрешением, метод будет выполнен и прочитает системное свойство.

До Java 8, методы `doPrivileged` имели только два параметра.

static <T> T	<code>doPrivilegedWithConbiner</code> (PrivilegedAction<T> action)	Performs the specified PrivilegedAction with privileges enabled.
static <T> T	<code>doPrivilegedWithConbiner</code> (PrivilegedAction<T> action, AccessControlContext context, Permission... perms)	Performs the specified PrivilegedAction with privileges enabled and restricted by the specified AccessControlContext and with a privilege scope limited by specified Permission arguments.
static <T> T	<code>doPrivilegedWithConbiner</code> (PrivilegedExceptionAction<T> action)	Performs the specified PrivilegedExceptionAction with privileges enabled.
static <T> T	<code>doPrivilegedWithConbiner</code> (PrivilegedExceptionAction<T> action, AccessControlContext context, Permission... perms)	Performs the specified PrivilegedExceptionAction with privileges enabled and restricted by the specified AccessControlContext and with a privilege scope limited by specified Permission arguments.

Как правило, класс, вызывающий `doPrivileged`, может иметь дополнительные разрешения, которые не требуются в этом коде, и которые также могут отсутствовать в некоторых вызываемых классах.

Чтобы не поднимать эти дополнительные разрешения с помощью `doPrivileged` и блокировать использование любого неправильного кода, который мог бы выполнять непреднамеренные действия, можно использовать вариант метода `doPrivileged` с тремя параметрами, где привелегии ограничиваются указанными разрешениями.

Однако такой метод выполняется менее эффективно, поэтому простой или критически важный код может не использовать такой метод.

Защищенные объекты

Разрешения и контроллер доступа могут быть инкапсулированы в один объект – защищенный объект, который реализуется классом `java.security.GuardedObject`.

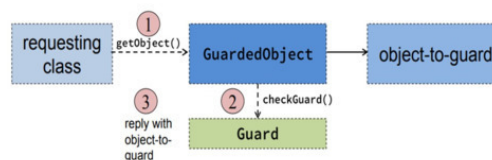


Объектно-ориентированное программирование на Java

Основы системы безопасности Java

Лекция 4

Защищенные объекты



Этот класс позволяет вставлять в него другой объект таким образом, чтобы весь доступ к объекту сначала должен был пройти через защиту, которая обеспечивается контроллером доступа.

Для использования защищенного объекта нужно сначала создать его экземпляр с помощью конструктора, который принимает два параметра – защищаемый объект и реализацию интерфейса `Guard`.

java.security Class GuardedObject java.lang.Object java.security.GuardedObject					
Constructors Constructor and Description GuardedObject(Object object, Guard guard) Constructs a GuardedObject using the specified object and guard.					
Methods <table> <tr> <th>Modifier and Type</th><th>Method and Description</th></tr> <tr> <td>Object</td><td>getObject() Retrieves the guarded object, or throws an exception if access to the guarded object is denied by the guard.</td></tr> </table>		Modifier and Type	Method and Description	Object	getObject() Retrieves the guarded object, or throws an exception if access to the guarded object is denied by the guard.
Modifier and Type	Method and Description				
Object	getObject() Retrieves the guarded object, or throws an exception if access to the guarded object is denied by the guard.				
java.security Interface Guard All Known Implementing Classes: AllPermission, AuthPermission, AuthZPermission, AllTPPermission, BasicPermission, DelegationPermission, FilePermission, JARFilePermission, LinkPermission, LoggingPermission, ManagementPermission, MBeanPermission, MBeanServerPermission, MBeanTrustPermission, NetPermission, Permission, PrivateCredentialPermission, PropertyPermission, ReflectPermission, RuntimePermission, SecurityPermission, SerializablePermission, ServicePermission, SocketPermission, SSLPermission, SubclassPermission, UncheckedPermission, UnsatisfiedPermission, ValidPermission					
Methods <table> <tr> <th>Modifier and Type</th><th>Method and Description</th></tr> <tr> <td>void</td><td>checkGuard(Object object) Determines whether or not to allow access to the guarded object object.</td></tr> </table>		Modifier and Type	Method and Description	void	checkGuard(Object object) Determines whether or not to allow access to the guarded object object.
Modifier and Type	Method and Description				
void	checkGuard(Object object) Determines whether or not to allow access to the guarded object object.				

Интерфейс `Guard` реализуется разрешениями, поэтому в качестве второго параметра передается экземпляр разрешения.

При вызове метода `getObject`, чтобы извлечь защищаемый объект, вызывается метод `checkGuard` объекта `Guard`, который защищает доступ.

Если доступ не разрешен, генерируется исключение.

В этом примере создается защищаемая строка и требуемое разрешение, которое будет защищать доступ к строке.

```
// Create the object that requires protection
String secretObj = "my secret";

// Create the required permission that will protect the object
Guard guard = new PropertyPermission("java.home", "read");

// Create the guard
GuardedObject gobj = new GuardedObject(secretObj, guard);

// Get the guarded object
try {
    Object o = gobj.getObject();
} catch (AccessControlException e) {
    // Cannot access the object
}
```

Далее создается защищенный объект и при вызове его метода `getObject` будет вызван метод `checkPermission` контроллера доступа.

Соответственно, чтобы разрешить доступ к строке, файл политики должен содержать нужное разрешение для данного кода.

Введение в Java криптографию



Объектно-ориентированное программирование на Java

Основы системы безопасности Java

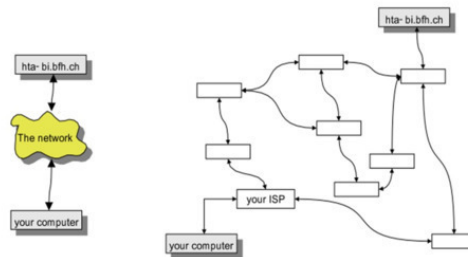
Лекция 5

Введение в Java криптографию

Программный интерфейс Java Cryptography API обеспечивает шифрование и аутентификацию данных.

Аутентификация данных – это процесс подтверждения происхождения и целостности данных.

Например, цифровая подпись может аутентифицировать класс Java, так что политика безопасности может позволить этому классу выполнять определенные операции.



Одна из задач системы безопасности – это возможность аутентификации классов, полученных из разных источников, например, загруженных из сети.

При этом возникает две задачи аутентификации.

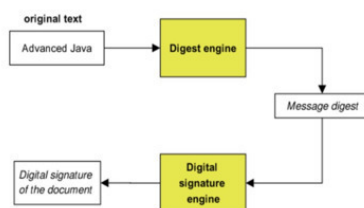
Это проверка идентичность источника, из которого был загружен класс, то есть аутентификация источника.

И проверка того, что класс не был изменен при передаче, то есть аутентификация данных.

Чтобы доверять классу, загруженному из стороннего источника, мы должны каким-то образом проверить, действительно ли класс получен из этого источника.

И эта аутентификация источника выполняется с помощью цифровой подписи, которая поставляется вместе с классом.

Отсутствие конфиденциальности при передаче данных является одной из причин, по которой вам может потребоваться шифрование передаваемых данных, чтобы их нельзя было прочитать.



Однако для целей аутентификации, шифрование данных не является необходимым.

Все, что необходимо, – это какая-то уверенность в том, что переданные данные, не были изменены в пути.

Это может быть достигнуто с помощью использования такого криптографического алгоритма, как цифровая подпись сообщения или цифровой отпечаток, вместо шифрования данных.

Этот алгоритм вычисляет на основе данных конечной длины, например, файла класса, строку бит длины n , которая является цифровым отпечатком данных.

Когда вы отправляете данные, вы можете использовать цифровой отпечаток этих данных, чтобы гарантировать, что данные не были изменены во время передачи.

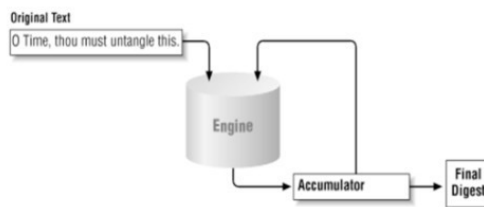
Однако эта аутентификация не препятствует чтению данных в процессе передачи, аутентифицированные данные не являются зашифрованными данными.

Аутентификация данных предотвращает запись данных, но не чтение данных.

Таким образом, Java обеспечивает два стандартных криптографических механизма: механизм цифровой подписи сообщений или цифровой отпечаток и механизм цифровой подписи.

И наконец, поскольку ключи шифрования являются центральными для этих механизмов, существует широкий набор классов, которые работают с ключами, включая механизмы, которые могут использоваться для генерации определенных типов ключей.

Концептуально, цифровая подпись сообщения или цифровой отпечаток представляет собой небольшую последовательность байтов или хэш, которая создается, когда данный набор данных передается через механизм цифрового отпечатка или хэширование.



В отличие от других криптографических двигателей, этот механизм не требует использования ключа.

Он принимает один поток данных в качестве своего ввода и производит один вывод или хэш.

Хэш, который соответствует определенному набору данных, не отражает никакой информации об этих данных – в частности, нет способа понять из хэша, сколько данных он представляет или какие данные фактически были.

Хэш полезен только тогда, когда доступны данные, которые он представляет.

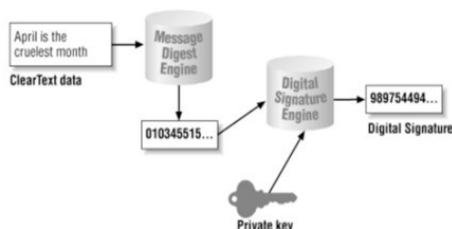
Если вы хотите определить, представляет ли конкретный хэш конкретный набор данных, вы должны пересчитать его и сравнить вновь вычисленный хэш с исходным хэшем.

Отметим также, что сам отпечаток сообщения не является защищенным объектом.

Хэш часто поставляется с данными, которые он представляет; получатель данных затем пересчитывает хэш, чтобы убедиться, что данные изначально не подделаны.

Но ничто в этом сценарии не мешает кому-то при передаче изменять как исходные данные, так и хэш, так как, и то и другое передается, и поскольку вычисление хэша является хорошо известной операцией, не требующей ключа.

Помимо хэширования, Java предлагает механизм цифровой подписи.



Как и настоящая подпись, цифровая подпись, как предполагается, однозначно идентифицирует объект, то есть физическое лицо или организацию.

Как и реальная подпись, цифровая подпись может быть подделана, хотя гораздо сложнее подделать цифровую подпись, чем реальную подпись.

Так как подделка цифровой подписи требует доступа к приватному ключу лица, подпись которого подделана.

Как и настоящая подпись, цифровая подпись может быть «размазана», так что она уже не будет распознаваться.

И так как цифровая подпись основана на сертификате, срок действия цифровой подписи может истечь.

Цифровые подписи получают в два этапа.

Сначала генерируется хэш сообщений, а затем этот хэш шифруется с помощью приватного ключа.

Обратите внимание, что шифрование выполняется хэша, а не в самих данных.

Чтобы представить эту подпись другому объекту, вы должны представить ему исходные данные – подпись – это просто зашифрованный отпечаток сообщения, и, вы не можете восстановить входные данные из отпечатка сообщения.

Для проверки цифровой подписи необходимо сначала вычислить хэш исходных данных.

Затем этот хэш передается через механизм шифрования, но на этот раз используется публичный ключ подписывающего лица.

Если цифровая подпись, созданная этой операцией, совпадает с цифровой подписью, которая была представлена, цифровая подпись считается действительной.

В качестве альтернативы, для некоторых алгоритмов цифровой подписи подписанный хэш может быть расшифрован с помощью публичного ключа, и хэши сравниваются.

Ничто не мешает перехвату подписанных данных.

Таким образом, данные, сопровождающие цифровую подпись, не могут быть конфиденциальными данными; цифровая подпись указывает только источник сообщения, но фактически не защищает это сообщение.

Однако это не значит, что данные можно подделать – если данные будут изменены, они не будут генерировать тот же хэш, который, в свою очередь, не будет генерировать ту же цифровую подпись.

И невозможно изменить данные, сгенерировать новый хэш этих данных, а затем восстановить цифровую подпись без доступа к приватному ключу.

Таким образом, помимо хэширования и цифровой подписи, Java предоставляет механизм генерации криптографических ключей.

В самом простом смысле ключ – это длинная строка чисел, которая имеет строгие математические свойства.

Математические свойства ключа варьируются в зависимости от криптографических алгоритмов, для которых ключ будет использоваться, но существует абстрактный набор свойств, который должны иметь все ключи.

В криптографии ключи могут использоваться либо по одному, и в этом случае они называются секретными ключами, либо попарно.

Пара ключей имеет два ключа: публичный ключ и приватный ключ.

Таким образом, есть три типа ключей – секретный, публичный и приватный.

Когда алгоритм требует секретного ключа, обе стороны, использующие этот алгоритм, будут использовать один и тот же ключ.

При использовании секретного ключа возникает ряд проблем.

Во-первых, для каждой пары сторон, которым необходимо обменяться зашифрованными данными, требуется отдельный ключ.

Если вы хотите отправить данные десяти разным сторонам, вам понадобится десять разных ключей.

Управление такими ключами – очень сложная проблема.

Другая проблема – это метод обмена ключами.

Крайне важно, чтобы ключ хранился в секрете, поскольку любой, у кого есть ключ, может расшифровать данные.

По этим причинам как правило используется пара ключей публичный ключ/приватный ключ.

Публичный и приватный ключи могут обеспечивать асимметричную работу криптографических алгоритмов.

Публичный ключ может использоваться одной стороной, участвующей в алгоритме, а приватный ключ может использоваться другой стороной.

Полезность этого типа ключей заключается в том, что один ключ может быть опубликован для всех.

Затем, когда кто-то хочет отправить вам конфиденциальную информацию, они могут использовать ваш публичный ключ для шифрования данных – и до тех пор, пока вы храните приватный ключ, вы будете единственным, кто сможет расшифровать эти данные.

Для цифровых подписей этот порядок ключей меняется: вы подписываете документ вашим приватным ключом, а получателю документа нужен ваш публичный ключ, чтобы проверить цифровую подпись.

При шифровании публичным ключом также существуют проблемы управления ключами.

Так как требуется надежный механизм получения публичного ключа.

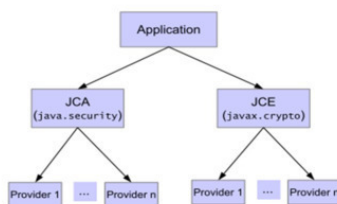
Помимо хэширования, цифровой подписи, и генерации криптографических ключей, Java предоставляет механизм шифрования.

Механизм шифрования обрабатывает шифрование и дешифрование произвольных данных.

Важно отметить, что механизмы шифрования Java, не используются для генерации и проверки цифровых подписей – цифровые подписи используют свои собственные алгоритмы для шифрования и расшифровки хэша сообщений.

Это различие позволяет механизму цифровой подписи использоваться там, где не требуется шифрование самих данных.

Java обеспечивает криптографическую функциональность с использованием двух API:



JCA – Java Cryptography Architecture: инфраструктура безопасности, интегрированная с основным Java API.

Этот API определяет инструменты и классы, связанные с безопасностью приложений Java, например, механизмы аутентификации.

И JCE – расширение криптографии Java Java Cryptography Extension: расширения для надежного шифрования.

Этот API предоставляет инструменты для шифрования контента.

Использование алгоритмов шифрования может быть ограничено в некоторых странах, поскольку они рассматриваются как военное оружие.

Поэтому необходима установка файлов политики JCE Unlimited Strength Jurisdiction Policy Files для обеспечения поддержки всех алгоритмов шифрования.

Для этого нужно загрузить файлы политики со страницы загрузки Oracle Java SE в разделе Дополнительные ресурсы.

Разархивировать загруженный файл.

И скопировать файлы в каталог `jre security`.

До JDK 1.4 JCE был отдельным продуктом, и поэтому JCA и JCE считались отдельными фреймворками.

Так как JCE теперь входит в JDK, различие становится менее очевидным.

Так как JCE использует ту же архитектуру, что и JCA, JCE следует рассматривать как часть JCA.

JCA включает в себя два программных компонента:

java.security	
java.security.cert	
java.security.spec	Независимость реализации и интегрируемость
java.security.interfaces	
javax.crypto	Независимость алгоритмов и расширяемость
javax.crypto.spec	
javax.crypto.interfaces	

Это фреймворк, который поддерживает криптографические сервисы, для которых провайдеры предоставляют реализации.

Этот фреймворк включает в себя такие пакеты, как `java.security`, `javax.crypto`, `javax.crypto.spec` и `javax.crypto.interfaces`.

Причем `javax.crypto` – это JCE.

И второй компонент – это провайдеры, которые содержат фактические криптографические реализации.

И JCA реализует два принципа:

Независимость реализации и интегрируемость

Независимость алгоритмов и расширяемость

Независимость алгоритма подразумевает, что каждая реализация предлагает один и тот же набор методов.

Расширяемость алгоритма подразумевает, что возможна простая модернизация классов с использованием новых алгоритмов.

Специфицировано поведение алгоритмов, а не их реализация.

Независимость реализации подразумевает возможность использования разных провайдеров криптографических сервисов.

Провайдеры предоставляют все алгоритмы для данного механизма.

И интегрируемость реализации подразумевает возможность провайдерам работать друг с другом.

Независимость реализации и независимость алгоритмов являются взаимодополняющими; вы можете использовать криптографические сервисы, такие как цифровые подписи и отпечатки сообщений, не беспокоясь о деталях реализации или даже алгоритмах.



Хотя полная независимость алгоритма невозможна, JCA предоставляет стандартизованные API-интерфейсы, специфичные для алгоритмов.

Независимость алгоритма достигается путем определения типов криптографических сервисов и определения классов, которые обеспечивают функциональность этих криптографических сервисов.

Независимость реализации достигается с использованием архитектуры, основанной на провайдере.

Программа может просто запросить объект определенного типа, например, объект подписи, представляющий конкретный алгоритм, и получить реализацию от одного из установленных провайдеров.

При желании программа может запросить реализацию у конкретного провайдера.

Интегрируемость реализации означает, что различные реализации могут работать друг с другом, использовать ключи друг друга или проверять подписи друг друга.

Это означает, например, что для одних и тех же алгоритмов, ключ, сгенерированный одним провайдером, может использоваться другим, а подпись, сгенерированная одним провайдером, поддается проверке другим.

Расширяемость алгоритма означает, что новые алгоритмы, которые вписываются в один из поддерживаемых классов, могут быть легко добавлены.

`java.security.Provider` – это базовый класс для всех провайдеров безопасности.

Каждый провайдер содержит экземпляр этого класса, который содержит имя провайдера и перечисляет все сервисы безопасности / алгоритмы, которые он реализует.

Когда необходим конкретный алгоритм, фреймворк JCA обращается к базе данных провайдеров, и, если найдено подходящее совпадение, создается экземпляр.

В JDK по умолчанию установлены и настроены один или несколько провайдеров.

Дополнительные поставщики могут добавляться статически или динамически.

И можно настроить среду выполнения для указания предпочтения провайдеров.

На слайде показано, как выполняется запрос отпечатка сообщений.

Здесь показаны три разных провайдера, которые реализуют различные алгоритмы отпечатка сообщений.

Провайдеры заказываются по предпочтению слева направо.

Слева приложение запрашивает реализацию алгоритма без указания имени провайдера.

И провайдеров ищут в порядке предпочтений, и возвращается реализация от первого провайдера.

Справа приложение запрашивает реализацию алгоритма у конкретного провайдера.

И провайдеры регистрируются статически с помощью конфигурационного файла `java.security` среды выполнения JRE, используя свойство `security.provider`.

Целостность и конфиденциальность данных

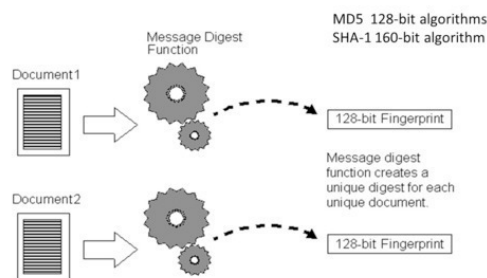


Объектно-ориентированное программирование на Java

Основы системы безопасности Java Лекция 6 Целостность и конфиденциальность данных

Отпечаток сообщения – это функция, которая обеспечивает целостность сообщения.

При создании отпечатка, сообщение принимается как вход и генерируется блок бит, обычно несколько сотен бит, который представляет отпечаток сообщения.



Небольшое изменение в сообщении создает заметное изменение в его отпечатке.

Функция отпечатка сообщения является односторонней функцией.

Просто создать отпечаток из сообщения, но довольно сложно создать сообщение, соответствующее данному отпечатку.

Отпечатки могут быть слабыми или сильными.

Вычисление контрольной суммы всех байтов сообщения, является примером слабой функции отпечатка.

Легко модифицировать один байт, чтобы сгенерировать любой желаемый отпечаток в виде контрольной суммы.

Большинство сильных функций отпечатка используют хеширование.

При этом 1-битное изменение сообщения приводит к большому изменению отпечатка.

Около 50% бит отпечатка изменяется.

Java поддерживает генерацию отпечатка длиной от 128 бит до 512 бит.

Наиболее используемые алгоритмы – это 128 битный MD5 и 160 битный SHA-1.

Для создания отпечатка используется класс MessageDigest.

Package java.security		
Class MessageDigest		
java.lang.Object java.security.MessageDigestSpi java.security.MessageDigest		
public abstract class MessageDigest extends MessageDigestSpi		
byte[]	digest()	Completes the hash computation by performing final operations such as padding.
byte[]	digest(byte[] input)	Performs a final update on the digest using the specified array of bytes, then completes the digest computation.
int	digest(byte[] buf, int offset, int len)	Completes the hash computation by performing final operations such as padding.
static MessageDigest	getInstance(String algorithm)	Returns a MessageDigest object that implements the specified digest algorithm.
static MessageDigest	getInstance(String algorithm, String provider)	Returns a MessageDigest object that implements the specified digest algorithm.
static MessageDigest	getInstance(String algorithm, Provider provider)	Returns a MessageDigest object that implements the specified digest algorithm.
void	update(byte input)	Updates the digest using the specified byte.
void	update(byte[] input)	Updates the digest using the specified array of bytes.
void	update(byte[] input, int offset, int len)	Updates the digest using the specified array of bytes, starting at the specified offset.
void	update(ByteBuffer input)	Update the digest using the specified ByteBuffer.

С помощью метода `getInstance` получается реализация указанного алгоритма от зарегистрированного провайдера.

После того, как вы создали экземпляр `MessageDigest`, вы можете использовать его для вычисления отпечатка из данных.

Если у вас есть один блок данных для вычисления отпечатка, используйте просто метод `digest`.

```
byte[] data1 = "0123456789".getBytes("UTF-8");

MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");
// print out the provider used
System.out.println("\n" + messageDigest.getProvider().getInfo());

byte[] digest = messageDigest.digest(data1);

byte[] data1 = "0123456789".getBytes("UTF-8");
byte[] data2 = "abcdefghijklmnopqrstuvwxyz".getBytes("UTF-8");

MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");
messageDigest.update(data1);
messageDigest.update(data2);

byte[] digest = messageDigest.digest();
```

Если у вас есть несколько блоков данных для включения в один и тот же отпечаток, вызовите метод `update` и завершите создание отпечатка вызовом метода `digest`.

Класс `javax.crypto.Mac` позволяет создавать код аутентификации сообщения `Message Authentication Code (MAC)` из бинарных данных.

```
byte[] data = "abcdefghijklmnopqrstuvwxyz".getBytes("UTF-8");

System.out.println("\nStart generating key");
KeyGenerator keyGen = KeyGenerator.getInstance("HmacMD5");
SecureRandom secureRandom = new SecureRandom();
int keyBitSize = 256;
keyGen.init(keyBitSize, secureRandom);
SecretKey MD5key = keyGen.generateKey();
System.out.println("Finish generating key");
//
// get a MAC object and update it with the plaintext
Mac mac = Mac.getInstance("HmacMD5");
byte[] macBytes = mac.doFinal(data);

byte[] data = "abcdefghijklmnopqrstuvwxyz".getBytes("UTF-8");
byte[] data2 = "0123456789".getBytes("UTF-8");

mac.update(data);
mac.update(data2);

byte[] macBytes = mac.doFinal();
```

MAC – это отпечаток сообщений, который был зашифрован секретным ключом.

Только если у вас есть секретный ключ, вы можете проверить MAC.

Java поддерживает алгоритмы аутентификации сообщений HMAC/SHA-1 и HMAC/MD5.

Экземпляр `Mac` создается с помощью метода `getInstance`.

Параметр `String`, переданный методу `getInstance`, содержит имя используемого алгоритма MAC.

После создания, экземпляр `Mac` должен быть инициализирован.

Вы инициализируете экземпляр `Mac`, вызывая его метод `init`, передавая в качестве параметра секретный ключ, который будет использоваться экземпляром `Mac`.

Класс `javax.crypto.KeyGenerator` используется для генерации симметричных ключей шифрования.

Симметричный ключ шифрования – это ключ, который используется для шифрования и дешифрования данных с помощью симметричного алгоритма шифрования.

Экземпляр `KeyGenerator` создается с помощью статического метода `getInstance`, передавая в качестве параметра имя алгоритма шифрования для создания ключа.

После создания экземпляра `KeyGenerator` его можно инициализировать методом `init`, указав размер бита для генерируемого ключа и случайное число `SecureRandom`, которое используется во время генерации ключа.

Генерирование ключа выполняется путем вызова метода `generateKey`.

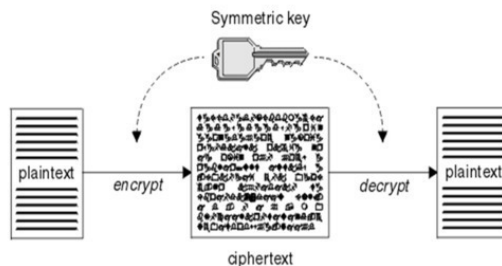
После инициализации экземпляра `Mac` вы можете начать вычислять значения MAC.

Чтобы вычислить значение MAC, вы вызываете метод `update` и `doFinal`.

Если у вас есть только один блок данных для вычисления MAC, вы можете напрямую вызвать `doFinal`.

Если у вас есть несколько блоков данных для вычисления MAC, например, если вы читаете блок за блоком, вы должны сначала вызывать метод `update` с каждым блоком и закончить вызовом метода `doFinal`.

Отпечаток сообщения может гарантировать целостность сообщения, но не может использоваться для обеспечения конфиденциальности сообщения.



Для этого нам нужно использовать криптографию секретным ключом для обмена приватными сообщениями.

Например, у Алисы и Боба есть общий ключ, который знают только они, и они соглашались использовать общий криптографический алгоритм или шифр.

Когда Алиса хочет отправить сообщение Бобу, она зашифровывает исходное сообщение, и затем отправляет зашифрованный текст Бобу.

Боб получает зашифрованный текст от Алисы и расшифровывает зашифрованный текст общим ключом.

Вы можете шифровать отдельные биты или фрагменты битов, называемые блоками.

Блоки шифрования, обычно имеют размер 64 бит.

Если сообщение не кратно 64 битам, тогда короткий блок должен быть дополнен.

Однобитовое шифрование чаще встречается в аппаратных реализациях.

И однобитовые шифры называются потоковыми шифрами.

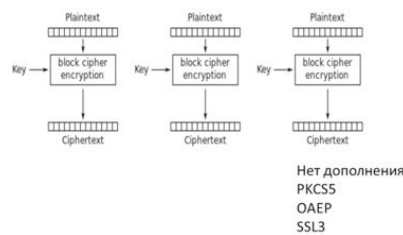
Сила шифрования секретным ключом определяется алгоритмом криптографии и длиной ключа.

Единственный способ атаковать алгоритм шифрования – это попробовать все возможные ключи, где количество попыток зависит от количества бит в ключе в степени.

Как правило, используют 128-битные ключи.

При этом, если бы миллион ключей можно было проверять каждую секунду, для поиска ключа понадобилось бы время, сравнимое с возрастом Вселенной.

Если используется блочное шифрование, а длина сообщения не кратна длине блока, последний блок должен быть дополнен байтами, чтобы получить полный размер блока.



Есть много способов дополнить блок, например, использовать нули или единицы.

С дополнением PKCS5 короткий блок дополняется повторяющимся байтом, значение которого представляет количество оставшихся байтов.

Блочное шифрование может использоваться в различных режимах.

Например, шифрование одного блока может зависеть от шифрования предыдущего блока или шифрование одного блока будет независимым от любых других блоков.

И Java поддерживает различные алгоритмы шифрования секретным ключом, например, AES (Advanced Encryption Standard) – 128-битный блочное шифрование с длиной ключа 128, 192 или 256 бит.

Блочное шифрование секретным ключом можно выполнить с помощью класса `javax.crypto.Cipher`.

```
byte[] plainText = "abcdefghijklmnopqrstuvwxyz".getBytes("UTF-8");
// get a DES private key
System.out.println("Start generating DES key");
KeyGenerator keyGen = KeyGenerator.getInstance("DES");
keyGen.init(56);
Key key = keyGen.generateKey();
System.out.println("Finish generating DES key");
// get a DES cipher object and print the provider
Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
System.out.println("n" + cipher.getProvider().getInfo());

// encrypt using the key and the plaintext
System.out.println("Start encryption");
cipher.init(Cipher.ENCRYPT_MODE, key);
byte[] cipherText = cipher.doFinal(plainText);
System.out.println("Finish encryption: ");
System.out.println(new String(cipherText, "UTF8"));

// decrypt the ciphertext using the same key
System.out.println("Start decryption");
cipher.init(Cipher.DECRYPT_MODE, key);
byte[] newPlainText = cipher.doFinal(cipherText);
System.out.println("Finish decryption: ");
System.out.println(new String(newPlainText, "UTF8"));
```

Cipher является стандартным термином для алгоритма шифрования в криптографии.

Вот почему класс Java называется Cipher, а не, например, Encrypter, Decrypter или что-то еще.

Экземпляр класса `Cipher` создается с помощью метода `getInstance` с параметром, указывающим, какой тип алгоритма шифрования вы хотите использовать.

После типа алгоритма вы можете указать через черту режим шифрования, например, режим электронной кодовой книги, когда каждый блок шифруется/расшифровывается независимо от других блоков.

О режимах блочного шифрования можно почитать в википедии.

Далее через черту вы можете добавить способ дополнения последнего блока, например, PKCS5, когда короткий блок дополняется повторяющимся байтом, значение которого представляет количество оставшихся байтов.

Имейте в виду, что не все алгоритмы шифрования и режимы поддерживаются провайдером шифрования Java по умолчанию.

Возможно, вам понадобится внешний провайдер, чтобы создать желаемый экземпляр `Cipher` с требуемым режимом и схемой дополнения.

Прежде чем вы используете экземпляр `Cipher`, вы его инициализируете.

Инициализация шифрования выполняется путем вызова метода `init`.

Метод `init` принимает два параметра:

Режим шифрования или дешифрования.

И ключ шифрования или дешифрования.

Далее вы вызываете методы `update` и `doFinal`.

Если вам нужно зашифровать или дешифровать один блок данных, просто вызовите метод `doFinal` с данными для шифрования или дешифрования.

Если вам необходимо зашифровать или дешифровать несколько блоков данных, например, несколько блоков из большого файла, вы вызываете метод `update` один раз для каждого блока данных и заканчиваете шифрование или дешифрование вызовом метода `doFinal` с последним блоком данных.

Инициализация экземпляра `Cipher` является дорогостоящей операцией.

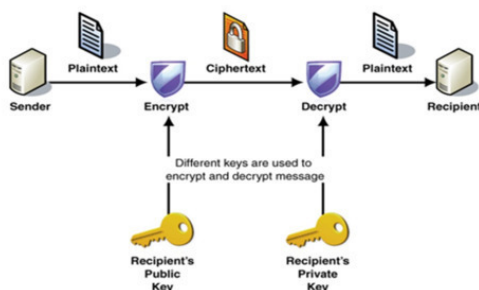
Поэтому рекомендуется повторно использовать экземпляры `Cipher`.

К счастью, класс `Cipher` был разработан с учетом повторного использования.

Когда вы вызываете метод `doFinal` для экземпляра `Cipher`, экземпляр `Cipher` возвращается в состояние, которое оно было сразу после инициализации.

Экземпляр `Cipher` затем может использоваться для повторного шифрования или дешифрования других данных.

Криптография с публичным ключом решает проблему шифрования сообщений между сторонами без предварительного соглашения о ключе.



Криптография с секретным ключом имеет один главный недостаток: каким образом секретный ключ попадет к Алисе и Бобу?

Криптография с публичным ключом, изобретенная в 1970-х годах, решает проблему шифрования сообщений между двумя сторонами без предварительного согласования ключа.

В криптографии с публичным ключом у Алисы и Боба есть не только разные ключи, но и каждый из них имеет два ключа.

Один ключ является приватным.

Другой ключ является публичным.

Когда Алиса хочет отправить защищенное сообщение Бобу, она шифрует сообщение с помощью публичного ключа Боба и отправляет результат Бобу.

Боб использует свой приватный ключ для дешифрования сообщения.

Когда Боб хочет отправить безопасное сообщение Алисе, он шифрует сообщение с помощью публичного ключа Алисы и отправляет результат Алисе.

Алиса использует свой приватный ключ для дешифрования сообщения.

Публичный и приватный ключи генерируются как пара и нуждаются в большей длине, чем ключи шифрования с секретным ключом той же силы.

Типичные длины ключей для алгоритма RSA составляют 1024 байта.

Шифрование с публичным ключом происходит медленно, от 100 до 1000 раз медленнее, чем шифрование с секретным ключом, поэтому обычно используется на практике смешанная техника.

Шифрование с публичным ключом используется для передачи секретного ключа, известного как ключ сеанса, а затем шифрование с секретным ключом используется для шифрования сообщений.

Класс `java.security.KeyPairGenerator` используется для генерации асимметричных пар ключей шифрования / дешифрования.

```
byte[] plainText = "abcdefghijklmnopqrstuvwxyz".getBytes("UTF-8");
// generate an RSA key
System.out.println("Start generating RSA key");
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(1024);
KeyPair key = keyGen.generateKeyPair();
System.out.println("Finish generating RSA key");
// get an RSA cipher object and print the provider
Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
System.out.println("Cipher provider: " + cipher.getProvider().getName());

// encrypt the plaintext using the public key
System.out.println("Start encryption");
cipher.init(Cipher.ENCRYPT_MODE, key.getPublic());
byte[] cipherText = cipher.doFinal(plainText);
System.out.println("Finish encryption");
System.out.println(new String(cipherText, "UTF8"));

// decrypt the ciphertext using the private key
System.out.println("Start decryption");
cipher.init(Cipher.DECRYPT_MODE, key.getPrivate());
byte[] newPlainText = cipher.doFinal(cipherText);
System.out.println("Finish decryption");
System.out.println(new String(newPlainText, "UTF8"));
```

Экземпляр `KeyPairGenerator` создается с помощью вызова метода `getInstance`.

Метод `getInstance` принимает имя алгоритма шифрования для генерации пары ключей.

В этом примере мы используем имя `RSA`.

Этот алгоритм является самым популярным алгоритмом шифрования с публичным ключом.

Инициализация экземпляра `KeyPairGenerator` выполняется путем вызова метода `initialize`, который принимает длину ключа.

Экземпляр `KeyPair` содержит пару ключей, которая извлекается методами `getPublic` и `getPrivate`.

Чтобы сгенерировать `KeyPair` с помощью `KeyPairGenerator`, вы вызываете метод `generateKeyPair`.

И для шифрования используется все тот же класс `Cipher`.

Метод `getInstance` создает объект `Cipher` (указывая алгоритм, режим и дополнение).

Метод `init` инициализирует объект `Cipher`.

И метод `doFinal` шифрует или дешифрует данные.

Цифровые подписи могут обеспечивать идентификацию сторон, которые обмениваются сообщениями.

```
byte[] plainText = "abcdefghijklmnopqrstuvwxyz".getBytes("UTF-8");
// generate an RSA keypair
System.out.println("Start generating RSA key");
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(1024);

KeyPair key = keyGen.generateKeyPair();
System.out.println("Finish generating RSA key");

// get a signature object using the MD5 and RSA combo
// and sign the plaintext with the private key,
// listing the provider along the way
Signature sig = Signature.getInstance("MD5withRSA");
sig.initSign(key.getPrivate());
sig.update(plainText);
byte[] signature = sig.sign();
System.out.println(sig.getProvider().getInfo());
System.out.println("Signature:");
System.out.println(new String(signature, "UTF8"));

// verify the signature with the public key
System.out.println("Start signature verification");
sig.initVerify(key.getPublic());
sig.update(plainText);
try {
    if (sig.verify(signature)) {
        System.out.println("Signature verified");
    } else System.out.println("Signature failed");
} catch (SignatureException se) {
    System.out.println("Signature failed");
}
```

Вы могли бы заметить недостаток в обмене сообщениями с публичным ключом.

Как Боб может проверить, что сообщение действительно пришло от Алисы?

Третья сторона могла бы заменить публичный ключ Алисы и прочитает сообщение.

Мы можем решить эту проблему, используя цифровую подпись, которая доказывает, что сообщение пришло от данной стороны.

Одним из способов реализации цифровой подписи является использование процесса обратного процессу шифрованию с публичным ключом.

Вместо публичного ключа, отправителем используется приватный ключ для подписания сообщения, а получатель использует публичный ключ отправителя для проверки подписи.

Так как только отправитель знает приватный ключ, получатель может быть уверен, что сообщение действительно пришло от отправителя.

Для создания подписи зашифровывается отпечаток сообщения, который затем расшифровывается и сравнивается с фактическим отпечатком сообщения.

Вы можете использовать алгоритм RSA для цифровых подписей и шифрования.

Алгоритм DSA может использоваться для цифровых подписей, но не для шифрования.

Для создания цифровой подписи можно сначала создать отпечаток данных, затем его зашифровать.

Но есть более простой способ.

Класс `java.security.Signature` сразу обеспечивает создание цифровой подписи для бинарных данных.

Экземпляр `Signature` создается с помощью статического метода `getInstance`.

Строка, передаваемая как параметр методу `getInstance` – это имя используемого алгоритма цифровой подписи.

Создав экземпляр `Signature`, вы должны инициализировать его, прежде чем сможете его использовать.

Вы инициализируете экземпляр `Signature`, вызывая его метод `init`.

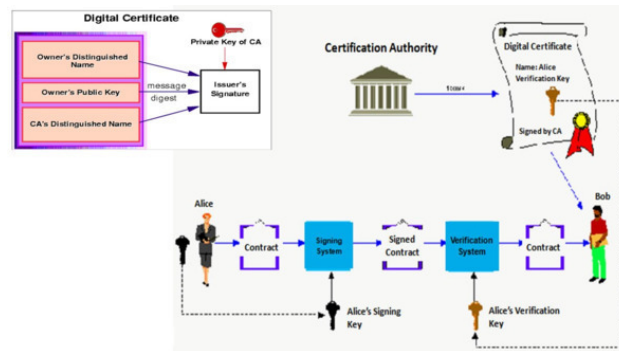
Как вы можете видеть, экземпляр `Signature` инициализируется закрытым ключом пары ключей, также может быть добавлено случайное число в виде экземпляра `SecureRandom`.

Когда экземпляр `Signature` инициализирован, вы можете использовать его для создания цифровых подписей.

Вы создаете цифровую подпись, вызывая метод `update` один или несколько раз, заканчивая вызовом метода `sign`.

Если вы хотите проверить цифровую подпись, созданную кем-то другим, вы должны инициализировать экземпляр `Signature` в режиме проверки, вместо режима подписи.

После инициализации в режиме проверки вы можете использовать экземпляр `Signature` для проверки цифровой подписи.



Цифровые сертификаты – это еще один способ для определения идентичности отправителя сообщения.

Вы могли, вероятно, заметить проблему со схемой цифровой подписи.

Цифровая подпись доказывает, что сообщение было отправлено данной стороной, но как мы точно узнаем, что отправитель действительно является тем, кем он себя заявляет.

Что, если кто-то утверждает, что он Алиса и подписывает сообщение, но на самом деле это Аманда?

Мы можем улучшить нашу безопасность, используя цифровые сертификаты, которые инкапсулируют идентичность вместе с публичным ключом и подписываются цифровой подписью третьей стороной, называемой центром сертификации.

Центр сертификации – это организация, которая проверяет идентичность в физическом смысле в реальном мире и подписывает публичный ключ и идентичность своим приватным ключом.

Получатель сообщения может получить цифровой сертификат отправителя и проверить или расшифровать его публичным ключом центра сертификации.

Это доказывает, что сертификат действителен и позволяет получателю извлечь публичный ключ отправителя для проверки его подписи или отправки ему зашифрованного сообщения.

Java поддерживает стандарт цифрового сертификата X.509.

Платформа Java использует хранилище ключей `keystore` в качестве хранилища ключей и сертификатов.

Физически хранилище ключей является файлом и есть возможность сделать его зашифрованным с именем по умолчанию. `keystore`.

Ключи и сертификаты могут иметь имена, называемые псевдонимами или алиасами, и каждый алиас может быть защищен уникальным паролем.

Хранилище `keystore` также защищено паролем, вы можете выбрать, чтобы каждый пароль алиаса соответствовал паролю хранилища ключей.

Платформа Java использует инструмент `keytool` для управления хранилищем ключей.

Этот инструмент предлагает множество опций, им можно генерировать публичный ключ и соответствующий сертификат.

И инструмент `keytool` может использоваться для экспорта ключа в файл в формате X.509, который может быть подписан центром сертификации, а затем повторно импортирован в хранилище ключей.

Существует также специальное хранилище ключей, которое используется для хранения доверенных сертификатов, которые, в свою очередь, содержат публичные ключи для проверки действительности других сертификатов.

Это хранилище ключей называется `truststore`.

Java поставляется с хранилищем по умолчанию `truststore` в файле `cacerts`.

```

KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
KeyStore keyStore = KeyStore.getInstance("PKCS12");

char[] keyStorePassword = "123abc".toCharArray();

try(InputStream keyStoreData = new FileInputStream("keystore.ks")){
    keyStore.load(keyStoreData, keyStorePassword);
}
keyStore.load(null, keyStorePassword);

KeyStore.ProtectionParameter entryPassword = new KeyStore.PasswordProtection(keyPassword);
KeyStore.Entry keyEntry = keyStore.getEntry("keyAlias", entryPassword);

KeyStore.PrivateKeyEntry privateKeyEntry = (KeyStore.PrivateKeyEntry)keyStore.getEntry("keyAlias", entryPassword);

SecretKey secretKey = getSecretKey();
KeyStore.SecretKeyEntry secretKeyEntry = new KeyStore.SecretKeyEntry(secretKey);

keyStore.setEntry("keyAlias2", secretKeyEntry, entryPassword);

try (FileOutputStream keyStoreOutputStream = new FileOutputStream("data/keystore.ks")) {
    keyStore.store(keyStoreOutputStream, keyStorePassword);
}

```

Хранилище ключей Java `keyStore` представлено классом `java.security.KeyStore`.

`KeyStore` можно записать на диск и прочитать.

И `KeyStore` в целом можно защитить паролем, и каждая запись ключа в `KeyStore` может быть защищена собственным паролем.

Это делает класс `KeyStore` полезным механизмом для безопасного шифрования ключами.

`KeyStore` может содержать следующие типы ключей:

Приватные ключи, публичные ключи + сертификаты, секретные ключи.

Приватные и публичные ключи используются в асимметричном шифровании.

И публичный ключ может иметь связанный сертификат.

Секретные ключи используются в симметричном шифровании.

Вы можете создать экземпляр Java `KeyStore`, вызвав его метод `getInstance`.

Вы можете создать экземпляр `KeyStore` по умолчанию.

Также возможно создать другие типы экземпляра `KeyStore`, передав тип хранилища методу `getInstance`.

Прежде чем использовать экземпляр `KeyStore`, он должен быть загружен.

Экземпляры `KeyStore` записываются на диск для последующего использования.

Вот почему класс `KeyStore` предполагает, что вы должны прочитать его данные, прежде чем сможете его использовать.

Тем не менее, можно инициализировать пустой экземпляр `KeyStore` без данных.

Загрузка данных `KeyStore` из выполняется путем вызова метода `load`, который принимает два параметра:

Входящий поток, из которого можно загрузить данные `KeyStore`.

И массив символов, содержащий пароль `KeyStore`.

В этом примере загружается файл `KeyStore`, расположенный в файле `keystore.ks`.

Если вы не хотите загружать какие-либо данные `KeyStore`, просто передайте значение `null` для параметра `InputStream`.

Вы всегда должны загружать экземпляр `KeyStore` с данными или с нулевым значением.

Иначе `KeyStore` не инициализируется, и все вызовы его методов будут выбрасывать исключение.

Вы можете получить ключи экземпляра `KeyStore` с помощью метода `getEntry`.

Запись `KeyStore` сопоставляется с алиасом, который идентифицирует ключ, и запись защищена паролем ключа.

Таким образом, для доступа к ключу вы должны передать алиас ключа и пароль ключа методу `getEntry`.

Если вы знаете, что ключ, к которому вы хотите получить доступ, является приватным ключом, вы можете использовать `PrivateKeyEntry`.

После получения `PrivateKeyEntry` вы можете получить доступ к приватному ключу, сертификату и цепочке сертификатов с помощью методов `getPrivateKey`, `getCertificate`, `getCertificateChain`.

Вы также можете установить ключи в экземпляр `KeyStore`.

И вы можете захотеть сохранить `KeyStore` на диск, чтобы загрузить его снова в другой раз.

Вы сохраняете `KeyStore`, вызывая метод `store`.

Класс `java.security.cert.CertificateFactory` позволяет создавать экземпляры Java-сертификатов из бинарной кодировки сертификатов, такой как X.509.

```

CertificateFactory certificateFactory = CertificateFactory.getInstance("X.509");
InputStream certificateInputStream = new FileInputStream("my-x509-certificate.crt");
Certificate certificate = certificateFactory.generateCertificate(certificateInputStream);
InputStream certificateChainInputStream = new FileInputStream("my-x509-certificate-chain.crt");
CertPath certPath = certificateFactory.generateCertPath(certificateChainInputStream);
List<Certificate> certificates = certPath.getCertificates();
String type = certPath.getType();

```

`CertificateFactory` также может создавать экземпляры `CertPath`.

`CertPath` – это цепочка сертификатов, где каждый сертификат в цепочке подписывается следующим сертификатом в цепочке.

Прежде чем создавать экземпляры сертификатов, вы должны создать экземпляр `CertificateFactory`.

В этом примере создается экземпляр `CertificateFactory`, способный создавать экземпляры сертификатов X.509.

После создания экземпляра `CertificateFactory` вы можете начать создавать экземпляры сертификатов.

Вы делаете это с помощью метода `generateCertificate`.

`CertificateFactory` также может создавать экземпляры `CertPath`.

Вы создаете экземпляр `CertPath`, вызывая метод `generateCertPath`.

Класс `java.security.cert.CertPath` представляет собой цепочку криптографических сертификатов идентичности, где каждый сертификат является цифровым подписчиком следующего сертификата в цепочке.

Класс `CertPath` обычно используется для проверки идентичности вместе с сертификатами сертификационных центров, которые подписали сертификат.

Обычно вы получите экземпляр `CertPath` из `CertificateFactory` или `CertPathBuilder`.

Когда у вас есть экземпляр `CertPath`, вы можете получить экземпляры сертификатов, из которых `CertPath` состоит с помощью вызова метода `getCertificates`.

И метод `getType` возвращает строку, указывающую, какой тип имеют сертификаты, например, X.509.

Класс `java.security.cert.Certificate` представляет собой сертификат криптографической идентичности.


```

byte[] encodedCertificate = certificate.getEncoded();

PublicKey certificatePublicKey = certificate.getPublicKey();

String certificateType = certificate.getType();

// get expected public key from somewhere else (not Certificate instance !!)
PublicKey expectedPublicKey = ...;

try{
    certificate.verify(expectedPublicKey);
} catch (InvalidKeyException e) {
    // certificate was not signed with given public key
} catch (NoSuchAlgorithmException |
        NoSuchProviderException |
        SignatureException |
        CertificateException e){
    // something else went wrong
}

```

Экземпляр класса `Certificate` содержит имя и другие данные идентифицируемого объекта, а также, возможно, цифровую подпись центра сертификации.

Класс `Certificate` является абстрактным классом, поэтому, хотя вы можете использовать сертификат как тип переменной, ваша переменная всегда будет указывать на подкласс сертификата.

Класс `Certificate` имеет один подкласс – класс `X509Certificate`.

Этот класс представляет сертификат X.509, который используется как сертификат в протоколах HTTPS и TLS.

Вы можете получить экземпляр сертификата из `CertificateFactory` и из `KeyStore`.

Метод `getEncoded` сертификата возвращает закодированную версию сертификата в виде байтового массива.

Например, если сертификат является сертификатом X509, возвращаемый массив байтов будет содержать закодированную версию экземпляра сертификата X.509.

Метод `getPublicKey` возвращает экземпляр `PublicKey` этого сертификата.

Метод `getType` возвращает тип экземпляра сертификата.

И класс `Certificate` содержит методы `verify`.

Эти методы могут использоваться для проверки того, что сертификат действительно подписан приватным ключом, который соответствует ожидаемому публичному ключу.

Метод `verify` возвращает `void`.

Если проверка завершится с ошибкой, будет выброшено исключение `InvalidKeyException`.

Если исключение не выброшено, экземпляр сертификата можно считать проверенным.

Файлы JAR являются эквивалентом ZIP-файлам, позволяя упаковывать несколько файлов классов Java в один файл с расширением `.jar`.

Затем этот JAR-файл может быть подписан цифровой подписью, подтверждая происхождение и целостность кода классов внутри.

Получатель JAR-файла может решить, следует ли доверять коду на основе подписи отправителя и быть уверенным, что содержимое не было подделано до получения.

JDK поставляется с инструментом `jarsigner`, который предоставляет эту функцию.

При развертывании приложения доступ к ресурсам может быть основан на идентичности подписывающего лица, помещая инструкции управления доступом в файл политики.

Инструмент `jarsigner` принимает JAR-файл и приватный ключ и соответствующий сертификат в качестве входных данных, а затем генерирует подписанную версию JAR-файла в качестве вывода.

Этот инструмент вычисляет цифровые отпечатки для каждого класса в JAR-файле, а затем подписывает эти отпечатки, чтобы обеспечить целостность файла и идентифицировать владельца файла.

Аутентификация и авторизация



Объектно-ориентированное программирование на Java

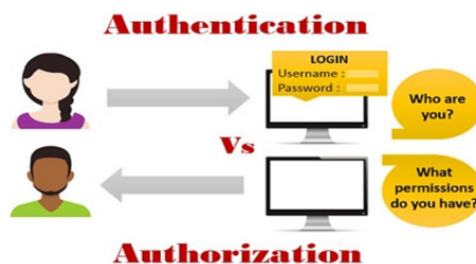
Основы системы безопасности Java

Лекция 7

Аутентификация и авторизация

Контроль доступа к ресурсам управляется на платформе Java службой аутентификации и авторизации Java (JAAS).

Аутентификация – это процесс, посредством которого пользователь или вычислительное устройство проверяет чью-то идентичность.



Авторизация – это процесс, посредством которого чувствительная часть программного обеспечения разрешает доступ и разрешаются операции, которые зависят от идентичности запрашивающего пользователя.

Эти две концепции идут рука об руку.

Без авторизации не зачем знать идентичность пользователя.

Без проверки идентичности невозможно отделить доверенных от ненадежных пользователей, что делает невозможным безопасное разрешение доступа к частям системы.

Не всегда необходимо идентифицировать или аутентифицировать отдельные объекты; в некоторых случаях вы можете выполнить аутентификацию группы, предоставляя определенное разрешение всем сущностям в данной группе.

В других случаях для безопасности системы имеет важное значение индивидуальная аутентификация.

Еще один интересный аспект аутентификации и авторизации заключается в том, что один объект может иметь несколько ролей в системе.

Например, пользователь может быть одновременно сотрудником компании, что означает, что ему нужен доступ к корпоративной электронной почте и бухгалтером в компании, что означает, что ему нужен доступ к системе бухгалтерского учета компании.

Аутентификация основана на одном или нескольких из следующих элементов:

Первый элемент – это «Что ты знаешь». Эта категория включает информацию, которую человек знает, и которая обычно не известна другим. Например, PIN-код, пароль и личную информацию, такую как девичья фамилия матери.

Следующий элемент – «Что у тебя есть». Эта категория включает физические элементы, которые обеспечивают индивидуальный доступ к ресурсам. Например, токены безопасности и кредитные карты.

И еще один элемент – это «Кто ты». Эта категория включает биометрические данные, такие как отпечатки пальцев, профили сетчатки и фотографии.

Зачастую недостаточно использовать только одну категорию для авторизации.

Например, банковская карта обычно используется в сочетании с ПИН-кодом.

Даже если физическая карта утеряна, безопасность сохраняется, так как вор должен знать PIN-код для доступа к ресурсам.

Существует два основных способа контроля доступа к чувствительному коду:

Это декларативная авторизация, которая может выполняться системным администратором, который настраивает доступ системы.

То есть статически объявляется, кто может получить доступ к каким приложениям в системе.

С помощью декларативного разрешения пользовательские права доступа могут быть добавлены, изменены или отменены, не затрагивая базовый код приложения.

И программируемая авторизация, которая использует код приложения для принятия решений об авторизации.

Программируемая авторизация необходима, когда решения об авторизации требуют более сложной логики, которая выходит за рамки возможностей декларативного разрешения.

Поскольку программируемая авторизация встроена в код приложения, для внесения изменений в авторизацию требуется, чтобы часть кода приложения была переписана.

Платформа Java позволяет осуществлять мелкомасштабный контроль доступа к вычислительным ресурсам, например, файлам на диске и сетевым соединениям на основе степени доверия, которую код имеет для пользователя.

Большинство базовых функций безопасности платформы Java предназначены для защиты пользователей от потенциально вредоносного кода.

Например, код с цифровой подписью, которая поддерживается сторонним сертификатом, обеспечивает идентификацию источника кода.

Основываясь на этом знании источника кода, пользователь может выбрать предоставление или отказ прав выполнения для этого кода.

Аналогично, пользователь может предоставить или запретить доступ на основе URL-адреса загрузки данного источника кода.

Контроль доступа в Java реализуется с помощью файла политики.

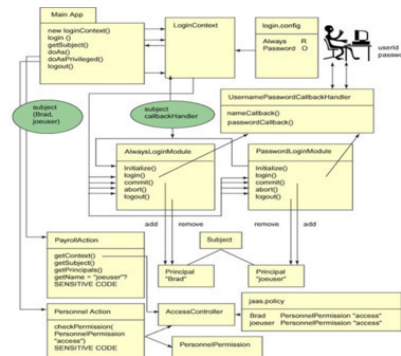
```
grant signedBy "Brad", codeBase "http://www.bradrubin.com" {  
    permission java.io.FilePermission "/tmp/abc", "read";  
};
```

В показанной записи файла политики, разрешается доступ чтения коду, который подписан «Brad» и загружен с <http://www.bradrubin.com>.

Другие встроенные возможности платформы Java, такие как отсутствие указателей, дополнительно защищают пользователей от потенциально вредоносного кода.

Службы аутентификации и авторизации JAAS работая вместе предоставляют дополнительную к этому функцию: они защищают конфиденциальный код приложения Java от потенциально злонамеренных пользователей.

Мы рассмотрим код примера использования JAAS, по частям.



Чтобы помочь отслеживать общую картину, на слайде показано, как все эти части относятся друг к другу.

При запуске основная программа сначала аутентифицирует пользователя, используя login модули, а затем разрешает или запрещает доступ к частям чувствительного кода на основе результатов этапа аутентификации.

Первый этап аутентификации заключается в создании контекста входа в систему и входе в систему.

Вверху LoginContext – это класс Java, который использует информацию в файле login.config, чтобы определить, какие login модули для входа в систему нужно использовать и какие критерии будут использоваться для определения успеха.

В этом примере есть два login модуля входа.

Первый, который называется AlwaysLoginModule, не требует пароля, поэтому он всегда разрешает вход.

Этот модуль отмечен ключевым словом `required`, что означает, что он должен всегда выполняться успешно.

Второй модуль, который называется PasswordLoginModule, требует пароля, но успех этого модуля является необязательным, поэтому он помечен ключевым словом `optional`.

И это означает, что общий вход может быть успешным даже в случае неудачи PasswordLoginModule.

После инициализации выбранные login модули входа в систему проходят двухфазный процесс фиксации, который контролируется LoginContext.

В рамках этого процесса вызывается обработчик UsernamePasswordCallbackHandler, чтобы получить имя пользователя и пароль от пользователя, который представлен объектом Субъекта Subject.

Если аутентификация прошла успешно, к Субъекту добавляется Принципал.

У Субъекта может быть много Принципов (в данном случае «Брэд» и «Джоузер»), каждый из которых разрешает пользователю или авторизует использовать разные уровни доступа к системе.

Здесь этап аутентификации завершается.

После завершения проверки идентичности мы используем объект Субъекта, чтобы попытаться выполнить чувствительный код, используя программируемую авторизацию и метод `doAs`.

JAAS проверяет, авторизован ли Субъект для доступа.

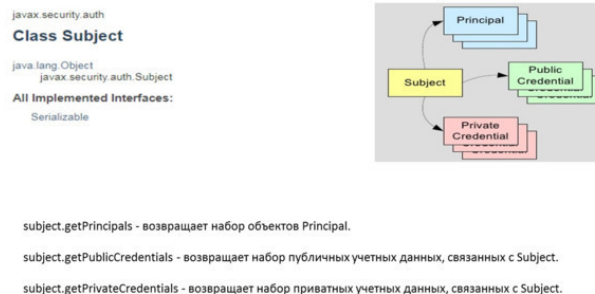
Если у Субъекта есть Принципы, который авторизован для доступа к коду, выполнение кода разрешается продолжить.

В противном случае выполнение будет отклонено.

Затем мы пытаемся выполнить чувствительный код с использованием декларативной авторизации и метода `doAsPrivileged`.

На этот раз JAAS использует пользовательское разрешение `PersonnelPermission`, файл политики и контроллер доступа `AccessController`, чтобы решить, можно ли авторизовать выполнение кода.

Субъект – это объект Java, который представляет собой единую сущность, такую как физическое лицо.



Один Субъект может иметь несколько связанных идентичностей, каждая из которых представлена объектом `Principal`.

Например, один Субъект представляет сотрудника, который требует доступа как к системе электронной почты, так и к системе учета.

И у этого Субъекта будут два Принципа, один из которых связан с идентификатором пользователя сотрудника для доступа к электронной почте, а другой – с его идентификатором пользователя для системы учета.

Принципы не являются постоянными, поэтому они должны быть добавлены к Субъекту каждый раз, когда пользователь входит в систему.

Принципал добавляется в Субъект как часть успешной процедуры аутентификации.

Аналогично, Принципал удаляется из Субъекта, если аутентификация терпит неудачу.

Независимо от успеха или не успеха аутентификации, все Принципы удаляются, когда приложение выполняет выход из системы.

В дополнение к содержанию набора Принципов, Субъект может содержать два набора учетных данных: один публичный и один приватный.

Учетные данные – это пароль, ключ, токен и т. д.

Доступ к публичным и приватным наборам учетных данных контролируется разрешениями Java.

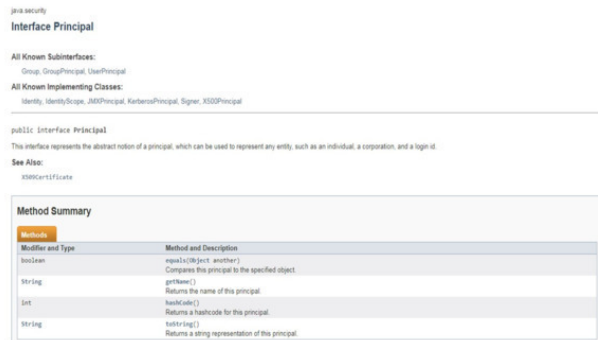
Объект `Subject` имеет несколько методов:

Это метод `getPrincipals`, который возвращает набор принципов. Поскольку результатом является набор, применяются операции `remove`, `add` и `contains`.

Метод `getPublicCredentials` возвращает набор публичных учетных данных, связанных с субъектом.

Метод `getPrivateCredentials` возвращает набор приватных учетных данных, связанных с субъектом.

Принципал – это интерфейс Java.



Разработчик должен реализовать интерфейс `Principal`, а также интерфейс `Serializable`, строку `name`, метод `getName`, который возвращает эту строку, и другие вспомогательные методы, такие как `hashCode`, `toString` и `equals`.

Принципалы добавляются к субъекту в процессе входа в систему.

```

1  import java.io.Serializable;
2  import java.security.Principal;
3  //
4  // This class defines the principle object, which is just an encapsulated
5  // String name
6  public class PrincipalImpl implements Principal, Serializable {
7
8      private String name;
9
10     public PrincipalImpl(String n) { name = n; }
11
12     public boolean equals(Object obj) {
13         if (!(obj instanceof PrincipalImpl)) {
14             return false;
15         }
16         PrincipalImpl pObj = (PrincipalImpl)obj;
17         if (name.equals(pObj.getName())) {
18             return true;
19         }
20         return false;
21     }
22
23     public String getName() { return name; }
24
25     public int hashCode() { return name.hashCode(); }
26
27     public String toString() { return getName(); }
28 }

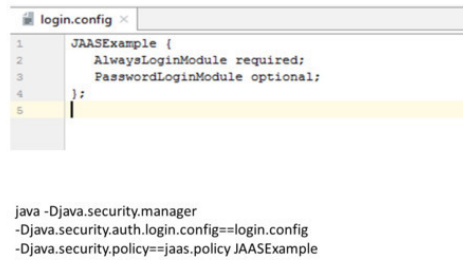
```

Декларативная авторизация основана на записях в файле политики.

Когда делается запрос авторизации, политика авторизации системы сравнивается с Принципами, содержащимися в Субъекте.

И авторизация предоставляется, если у Субъекта есть Принципал, который отвечает требованиям безопасности в файле политики; в противном случае авторизация отклоняется.

JAAS допускает гибкость в процедурах аутентификации, требуемых для субъекта, порядке их выполнения и комбинациях успешности аутентификации или сбоев, необходимых для того, чтобы субъект был признан аутентифицированным.



JAAS использует конфигурационный файл `login.config`, чтобы указать условия аутентификации для каждого логин модуля.

Файл `login.config` указывается в командной строке выполнения Java со свойством `security.auth.login.config`.

Java имеет файл конфигурации входа по умолчанию, поэтому знак (`==`) заменяет этот файл конфигурации входа в систему по умолчанию.

Если используется один знак равенства, конфигурационный файл добавляется, а не заменяет.

Конфигурационный логин файл содержит текстовую строку и список процедур входа в систему.

Несколько параметров используются для указания влияния успеха или отказа данной процедуры входа в общую процедуру проверки идентичности.

Эти параметры следующие:

`Required` означает, что модуль входа должен быть успешным.

При этом другие модули входа в систему также будут вызваны, даже если этот модуль не завершится успехом.

`Optional` означает, что модуль входа может быть не успешным, но общий вход может быть успешным, если другой модуль входа в систему успешно завершен.

Если все модули входа в систему являются необязательными, по крайней мере один из них должен завершиться успехом для успешной аутентификации.

`Requisite` означает, что модуль входа в систему должен быть успешным, и, если он не удастся, никакие другие модули входа в систему не будут вызваны.

`Sufficient` означает, что общий вход будет успешным, если модуль входа в систему завершится успешно, при условии, что нет других `required` или `requisite` модулей входа, которые не удались.

В этом примере, модуль `AlwaysLoginModule` должен быть успешным, и модуль `PasswordLoginModule` может либо быть успешным, либо дать сбой.

Важное значение для реализации этой технологии настройки входа в систему состоит в том, что она оставляет все важные решения, например, требуемые типы аутентификации и конкретные критерии для успеха или отказа аутентификации, на время развертывания.

Успешный вход в систему приведет к добавлению нового Субъекта к `LoginContext` с добавлением любого количества успешно прошедших аутентификацию Принципиалов к этому субъекту.

`LoginContext` – это класс Java, который используется для настройки процесса входа в систему, фактического входа в систему и получения субъекта, если вход успешно завершен.

<pre> java.security.auth.login Class LoginContext java.lang.Object java.security.auth.login.LoginContext </pre>	
Constructors	
Constructor and Description LoginContext(String name) Instantiate a new LoginContext object with a name. LoginContext(String name, CallbackHandler callbackHandler) Instantiate a new LoginContext object with a name and a CallbackHandler object. LoginContext(String name, Subject subject) Instantiate a new LoginContext object with a name and a Subject object. LoginContext(String name, Subject subject, CallbackHandler callbackHandler) Instantiate a new LoginContext object with a name, a Subject to be authenticated, and a CallbackHandler object. LoginContext(String name, Subject subject, CallbackHandler callbackHandler, Configuration config) Instantiate a new LoginContext object with a name, a Subject to be authenticated, a CallbackHandler object, and a Login Configuration.	
Method Summary	
Methods	
Subject	getSubject() Return the authenticated Subject.
void	login() Perform the authentication.
void	logout() Logout the Subject.

Этот класс имеет четыре основных метода:

Это конструктор, который в качестве первого параметра использует запись файла login.config, и обработчик обратного вызова, в качестве второго параметра.

Метод login фактически пытается войти в систему, подчиняясь правилам, указанным в конфигурационном логин файле.

Метод getSubject возвращает аутентифицированный субъект, если вход был успешным.

Метод logout отменяет регистрацию субъекта.

JAAS использует обработчик обратного вызова для получения информации входа от пользователей.

CallbackHandler указывается в конструкторе объекта LoginContext.

В этом примере обработчик обратного вызова использует несколько подсказок для получения имени и пароля от пользователя.

```

UsernamePasswordCallbackHandler.java
// This class implements a username/password callback handler that gets
// information from the user
public class UsernamePasswordCallbackHandler implements CallbackHandler {
    // The handle method does all the work and iterates through the array
    // of callbacks, examines the type, and takes the appropriate user
    // interaction action.
    public void handle(Callback[] callbacks) throws
        UnsupportedCallbackException, IOException {
        for(int i=0; i<callbacks.length; i++) {
            Callback cb = callbacks[i];
            // Handle username acquisition
            if (cb instanceof NameCallback) {
                NameCallback nameCallback = (NameCallback)cb;
                System.out.print( nameCallback.getPrompt() + "? ");
                System.out.flush();
                String username = new BufferedReader(
                    new InputStreamReader(System.in)).readLine();
                nameCallback.setName(username);
            }
            // Handle password acquisition
            else if (cb instanceof PasswordCallback) {
                PasswordCallback passwordCallback = (PasswordCallback)cb;
                System.out.print( passwordCallback.getPrompt() + "? ");
                System.out.flush();
                String password = new BufferedReader(
                    new InputStreamReader(System.in)).readLine();
                passwordCallback.setPassword(password.toCharArray());
                password = null;
            }
            // Other callback types are not handled here
        }
    }
}

```

Метод handle обработчика, который вызывается из логин-модуля, принимает в качестве параметра массив объектов Callback.

Во время входа в систему обработчик выполняет итерацию через массив Callback.

И метод handle проверяет тип объекта Callback и выполняет соответствующее действие пользователя.

Существуют следующие типы обратных вызовов – это NameCallback, PasswordCallback, TextInputCallback, TextOutputCallback, LanguageCallback, ChoiceCallback и ConfirmationCallback.

В некоторых приложениях взаимодействие с пользователем не требуется, так как JAAS используется для взаимодействия с механизмом аутентификации операционной системы.

В таких случаях параметр CallbackHandler в объекте LoginContext будет равен NULL.

Здесь приведен код обработчика UsernamePasswordCallbackHandler.

Он вызывается один раз с помощью модуля `AlwaysLoginModule` с одним обратным вызовом для получения имени пользователя и один раз с помощью модуля `PasswordLoginModule` с двумя обратными вызовами для получения имени пользователя и пароля.

`LoginModule` – это интерфейс для методов, необходимых для участия в процессе аутентификации JAAS.

javax.security.auth.spi
Interface LoginModule

Methods	
Modifier and Type	Method and Description
boolean	<code>abort()</code> Method to abort the authentication process (phase 2).
boolean	<code>commit()</code> Method to commit the authentication process (phase 2).
void	<code>initialize(Subject subject, CallbackHandler callbackHandler, Map<String,?> sharedState, Map<String,?> options)</code> Initialize this LoginModule.
boolean	<code>login()</code> Method to authenticate a Subject (phase 1).
boolean	<code>logout()</code> Method which logs out a Subject.

Поскольку успех или неудача конкретной процедуры входа в систему могут быть неизвестны до тех пор, пока не будут выполнены другие процедуры входа в систему, для определения успеха используется двухфазный процесс фиксации.

Объект `LoginModule` реализует следующие методы:

Метод `initialize` инициализирует `LoginModule`.

Метод `login` устанавливает любые необходимые обратные вызовы, вызывает `CallbackHandler` для их обработки и сравнивает возвращаемую информацию, то есть имя пользователя и пароль с допустимыми значениями.

Если есть совпадение, модуль входа в систему завершается успешно, хотя он все равно может быть прерван, если другой модуль входа не завершился успехом, в зависимости от настроек в файле `login.config`.

Метод `commit` вызывается для определения успеха как часть процесса двухфазной фиксации.

Если все модули входа успешно пройдены с ограничениями, указанными в файле `login.config`, создается новый Принципал вместе с именем пользователя и добавляется к основному набору субъекта.

Метод `abort` вызывается, если общий вход не увенчался успехом; если происходит прерывание, внутреннее состояние `LoginModule` должно быть очищено.

Метод `logout` вызывается для удаления Принципала из основного набора субъекта и выполнения внутренней очистки состояния.

В этом примере используются два модуля входа.

Первый, `AlwaysLoginModule`, всегда успешный.

Второй, `PasswordLoginModule`, является успешным только в том случае, если идентификатор пользователя и пароль соответствуют определенным жестко закодированным значениям.

Аутентификация `AlwaysLoginModule` всегда будет успешной, поэтому она используется только для получения имени пользователя через функцию `NameCallback`.

```
// This is a JaaS Login Module that always succeeds. While not realistic,
// it is designed to illustrate the bare bones structure of a Login Module
// and is used as example that show the login configuration file operation.
public class AlwaysLoginModule implements LoginModule {

    private Subject subject;
    private Principal principal;
    private CallbackHandler callbackHandler;
    private String username;
    private boolean loginSuccess;

    // Initialize sets up the login module. sharedState and options are
    // advanced features not used here
    public void initialize(Subject sub, CallbackHandler chb,
        Map sharedState, Map options) {
        subject = sub;
        callbackHandler = chb;
        loginSuccess = false;
    }

    // The login phase gets the userid from the user
    public boolean login() throws LoginException {
        // Since we need input from a user, we need a callback handler
        if (callbackHandler == null) {
            throw new LoginException("No CallbackHandler defined");
        }
        Callback[] callbacks = new Callback[1];
        callbacks[0] = new NameCallback(prompt("Username"));
        // Call the callback handler to get the username
    }

    try {
        System.out.println("AlwaysLoginModule login");
        callbackHandler.handle(callbacks);
        username = ((NameCallback)callbacks[0]).getName();
    } catch (IOException ioe) {
        throw new LoginException(ioe.toString());
    } catch (UnsupportedCallbackException uce) {
        throw new LoginException(uce.toString());
    }

    loginSuccess = true;
    System.out.println();
    System.out.println("Login: AlwaysLoginModule SUCCESS");
    return true;
}

// The commit phase adds the principal if both the overall authentication
// succeeds (which is why commit was called) as well as this particular
// login module
public boolean commit() throws LoginException {
    // Check to see if this login module succeeded (which it always will
    // is this example)
    if (loginSuccess == false) {
        System.out.println("Commit: AlwaysLoginModule FAIL");
        return false;
    }

    // If this login module succeeded too, then add the new principal
    // to the subject (if it does not already exist)
    principal = new PrincipalImpl(username);
    if (!subject.getPrincipals().contains(principal)) {
        subject.getPrincipals().add(principal);
    }
}
```

Предполагая, что другие модули входа успешно завершены, метод `commit` `AlwaysLoginModule` создаст новый объект `PrincipalImpl` с именем пользователя и добавляет его в основной набор субъекта.

Выход из системы удаляет принциपालа из основного набора субъекта.

Модуль `PasswordLoginModule` использует `NameCallback` для получения имени пользователя и `PasswordCallback` для получения пароля.

```
// This is a JaaS Login Module that requires both a username and a password.
// The username must equal the hardcoded "jones" and the password
// must match the hardcoded "jonesgrr".
public class PasswordLoginModule implements LoginModule {

    private Subject subject;
    private Principal principal;
    private CallbackHandler callbackHandler;
    private String username;
    private char[] password;
    private boolean loginSuccess;

    // Initialize sets up the login module. sharedState and options are
    // advanced features not used here
    public void initialize(Subject sub, CallbackHandler chb,
        Map sharedState, Map options) {
        subject = sub;
        callbackHandler = chb;
        loginSuccess = false;
        username = null;
        clearPassword();
    }

    // The login phase gets the userid and password from the user and
    // compares them to the hardcoded values "jones" and "jonesgrr".
    public boolean login() throws LoginException {
        // Since we need input from a user, we need a callback handler
        if (callbackHandler == null) {
            throw new LoginException("No CallbackHandler defined");
        }
        Callback[] callbacks = new Callback[2];
        callbacks[0] = new NameCallback(prompt("Username"));
        callbacks[1] = new PasswordCallback(prompt("Password"), false);

        // Call the callback handler to get the username and password
    }

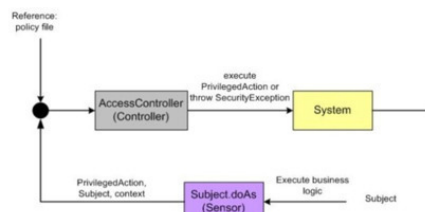
    try {
        System.out.println("PasswordLoginModule login");
        callbackHandler.handle(callbacks);
        username = ((NameCallback)callbacks[0]).getName();
        char[] temp = ((PasswordCallback)callbacks[1]).getPassword();
        password = new char[temp.length];
        System.arraycopy(temp, 0, password, 0, temp.length);
        ((PasswordCallback)callbacks[1]).clearPassword();
    } catch (IOException ioe) {
        throw new LoginException(ioe.toString());
    } catch (UnsupportedCallbackException uce) {
        throw new LoginException(uce.toString());
    }

    System.out.println();

    // If username matches, go on to check password
    if (username.equals("jones")) {
        System.out.println("Login: PasswordLoginModule Username Matches");
        if (password.length == 14
            password[0] == 'j'
            password[1] == 'o'
            password[2] == 'n'
            password[3] == 'e'
            password[4] == 's'
            password[5] == 'g'
            password[6] == 'r'
            password[7] == 'r'
            password[8] == ' '
            password[9] == 'j'
            password[10] == 'o'
            password[11] == 'n'
            password[12] == 'e'
            password[13] == 's') {
            // If userid and password match, then login is a success
            System.out.println("Login: PasswordLoginModule Password Matches");
            loginSuccess = true;
            System.out.println("Login: PasswordLoginModule SUCCESS");
        }
    }
}
```

Если имя пользователя и пароль – соответствуют указанным значениям, аутентификация будет успешной.

Важно понимать, как платформа Java реализует контроль доступа для авторизации.



Платформа Java использует понятие контекста управления доступом для определения полномочий текущего потока выполнения.

Концептуально это можно рассматривать как токен, прикрепленный к каждому потоку исполнения.

До JAAS управление доступом основывалось на знании источника кода текущего класс файла Java или идентичности подписывающего лица.

В рамках этой модели контроль доступа основывался на знании того, откуда пришел код.

С JAAS, добавляя субъект к контексту управления доступом, мы можем предоставлять или запрещать доступ на основе того, кто выполняет данный фрагмент кода.

Поскольку поток выполнения может использовать несколько модулей с различными характеристиками контекста, платформа Java реализует концепцию наименьших привилегий.

Во всем стеке вызывающих, относящихся к данному потоку выполнения, где члены стека вызовов имеют разные характеристики, результатом, используемым для определения полномочий, является тоска пересечения всех этих характеристик или наименьший общий знаменатель.

Например, если кусок вызывающего кода имеет ограниченные полномочия, то есть возможно, ему не доверяют, поскольку он не подписан, но он вызывает часть более надежного кода, у которого возможно есть подпись, тогда полномочия в вызываемом коде уменьшаются в соответствии с меньшим доверием вызывающего.

Характеристики полномочий, содержащиеся в контексте управления доступом, сравниваются с разрешениями в файле политики, чтобы определить, разрешены ли чувствительные операции.

Это выполняется контроллером `AccessController`, который имеет интерфейсы для программной проверки прав доступа и получения текущего субъекта, связанного с контекстом активного доступа.

Поскольку субъект может быть аутентифицирован после запуска приложения, должен быть способ динамически привязать субъект к контексту управления доступом для создания единого контекста, который содержит полномочия кода, откуда он был загружен, и кто его подписал, а также полномочия пользователя, то есть субъект.

Для этого используется метод `doAs` субъекта.

Этот метод `doAs` вызывает класс, специально предназначенный для авторизации, который реализует интерфейс `PrivilegedAction`.

Другим вызовом, который может использоваться для указания контекста управления доступом, является метод `doAsPrivileged` субъекта.

Особое использование этого метода – это установить для `AccessControlContext` значение `null`, которое замыкает стек вызовов в точке, где происходит вызов `doAsPrivileged`, что позволяет увеличить полномочия в объекте `PrivilegedAction`.

Этот метод создает новый `AccessControlContext` с пустой коллекцией доменов безопасности.

И платформа Java имеет ряд встроенных разрешений, которые используются для управления доступом к системным ресурсам, например, `java.io.FilePermission`.

Также платформа Java позволяет создавать собственные объекты разрешений.

Как и обычные разрешения, они могут быть помещены в файл политики и установлены во время развертывания.

В этом примере используется пользовательское разрешение, конструктор которого принимает определяемое имя разрешения.

```
import java.security.*;
//
// Implement a user defined permission for access to the personnel
// code for this example
public class PersonnelPermission extends BasicPermission {
    public PersonnelPermission(String name) {
        super(name);
    }
    public PersonnelPermission(String name, String action) {
        super(name);
    }
}

# jaas.policy
1 grant {
2     permission javax.security.auth.AuthPermission "createLoginContext";
3     permission javax.security.auth.AuthPermission "doAs";
4     permission javax.security.auth.AuthPermission "doAsPrivileged";
5     permission javax.security.auth.AuthPermission "modifyPrincipal";
6     permission javax.security.auth.AuthPermission "getSubject";
7 }
8
9 grant Principal PrincipalImpl "Brad" {
10     permission PersonnelPermission "access";
11 }
12
13
```

Второй конструктор принимает дополнительный параметр, называемый действием.

Файлы политики являются основным механизмом контроля доступа к системным ресурсам, включая чувствительный код.

Файл политики в этом примере называется `jaas.policy` и указывается в командной строке Java с помощью свойства `security.policy`.

Знак (`==`) заменяет файл системной политики вместо добавления.

Первые пять разрешений в этом файле политики для загрузки механизма JAAS.

В следующем разрешении принципу «Брэд», предоставляется доступ к пользовательскому разрешению `PersonnelPermission`.

Основной класс примера создает контекст входа в систему, осуществляет вход, и пытается выполнить два чувствительных объекта.

```
// This is the main program in the JAAS Example. It creates a Login Context,
// logs the user in based on the settings in the Login Configuration File,
// and calls two sensitive pieces of code, the first using programmatic
// authentication, and the second using declarative authorization.
public class JAASExample {
    static LoginContext lc = null;

    public static void main( String[] args) {
        //
        // Create a login context
        try {
            lc = new LoginContext( "JMSExample",
                new UsernamePasswordCallbackHandler());
        } catch (LoginException le) {
            System.out.println( "Login Context Creation Error" );
            System.exit( 0);
        }
        // login
        try {
            lc.login();
        } catch (LoginException le) {
            System.out.println( "JMSExample AUTHENTICATION FAILED" );
            System.exit( 0);
        }
        System.out.println( "JMSExample AUTHENTICATION SUCCEEDED" );
        System.out.println( lc.getSubject() );
        //
        // Call the sensitive PayrollAction code, which uses programmatic
        // authentication.
        try {
            Subject.doAs( lc.getSubject(), new PayrollAction() );
        } catch (AccessControlException e) {
            System.out.println( "Payroll Access DENIED" );
        }
        //
        // Call the sensitive PersonnelAction code, which uses declarative
        // authorization.
        try {
            Subject.doAsPrivileged( lc.getSubject(), new PersonnelAction(), null );
        } catch (AccessControlException e) {
            System.out.println( "Personnel Access DENIED" );
        }
        try {
            lc.logout();
        } catch (LoginException le) {
            System.out.println( "Logout FAILED" );
            System.exit( 0);
        }
        System.exit( 0);
    }
}
```

Один использует программную авторизацию, а другой – декларативную авторизацию, а затем выходит из системы.

В этом примере класс `PrivilegedAction` обеспечивает программируемую авторизацию.

```
// This class is a sensitive Payroll function that demonstrates the
// use of programmatic authorization which only allows a subject
// that contains the principal "joesuser" in
class PayrollAction implements PrivilegedAction {
    public Object run() {
        // Get the passed in subject from the code
        AccessControlContext context = AccessController.getContext();
        Subject subject = Subject.getSubject( context );
        if (subject == null) {
            throw new AccessControlException( "Denied" );
        }
        //
        // Iterate through the principal set looking for joesuser. If
        // he is not found,
        Set principals = subject.getPrincipals();
        Iterator iterator = principals.iterator();
        while (iterator.hasNext()) {
            PrincipalImpl principal = (PrincipalImpl) iterator.next();
            if (principal.getName().equals( "joesuser" )) {
                System.out.println( "joesuser has Payroll access\n" );
                return new Integer( 0 );
            }
        }
        throw new AccessControlException( "Denied" );
    }
}
```

Он вызывается методом `doAs` из основной программы, и аутентифицированный субъект привязывается к контексту приложения в потоке, при входе в метод `run`.

Здесь мы извлекаем текущий субъект из контроллера доступа и выполняем итерацию по всем аутентифицированным Принципам, ища «joeuser».

Если мы его найдем, мы сможем сделать чувствительную операцию.

Если нет, мы выбрасываем исключение `AccessControlException`.

В этом примере класс `PersonnelAction` обеспечивает декларативную авторизацию с помощью пользовательского разрешения `PersonnelPermission`.

```
// This class is a sensitive Personnel function that demonstrates
// the use of declarative authorization using the user defined
// permission PersonnelPermission, which throws an exception
// if it not granted
class PersonnelAction implements PrivilegedAction {
    public Object run() {
        AccessController.checkPermission(new PersonnelPermission( name: "access"));
        System.out.println( "Subject has Personnel access\n");
        return new Integer( value: 0);
    }
}

// Call the sensitive PersonnelAction code, which uses declarative
// authorization.
try {
    Subject.doAsPrivileged( In.getSubject(), new PersonnelAction(), null );
} catch (AccessControlException e) {
    System.out.println( "Personnel Access DENIED" );
}
```

Здесь мы просто запрашиваем `AccessController` для проверки этого разрешения.

Мы вызываем объект `PrivilegedAction` вызовом метода `doAsPrivileged` и нулевым контекстом управления доступом в главном классе для замыкания стека вызовов в точке вызова.

Это необходимо, так как до объединения Субъекта с контекстом в вызове метода `doAsPrivileged`, субъект не был частью контекста и не авторизовался для записи `grant`, и из-за принципа наименьших привилегий увеличение авторизации в противном случае не будет допущено.

Для запуска этого примера, необходимо скачать проект приложения и открыть его в среде IntelliJ IDEA.

Затем собрать проект и в каталоге `out` запустить файл `run.bat`.

После этого откроется окно терминала и у вас запросят имя пользователя и пароль.

```
AlwaysLoginModule Login
Username? Brad

Login: AlwaysLoginModule SUCCESS

PasswordLoginModule Login
Username? joeuser
Password? joeupw

Login: PasswordLoginModule Username Matches
Login: PasswordLoginModule Password Matches
Login: PasswordLoginModule SUCCESS
Commit: AlwaysLoginModule SUCCESS
Commit: PasswordLoginModule SUCCESS

OVERALL AUTHENTICATION SUCCEEDED

Subject:
Principal: Brad
Principal: joeuser

joeuser has Payroll access
Subject has Personnel access

Logout: AlwaysLoginModule SUCCESS
Logout: PasswordLoginModule SUCCESS
```

Файл `Login.config` определяет два модуля входа; первый модуль `required` `AlwaysLoginModule`. Он будет запущен первым.

`AlwaysLoginModule` начинается с фазы входа, которая вызывает обработчик обратного вызова, чтобы получить имя пользователя (Brad). И вход будет успешным.

Второй модуль входа, `PasswordLoginModule`, является `optional`.

Он запускается, вызывая обработчик обратного вызова, чтобы получить имя пользователя (joeuser), и пароль (joerw), при совпадении вход будет также успешным.

Так как модули завершаются успешно, метод commit вызывается в обоих модулях входа, и вся аутентификация завершается успешно.

В результате Субъект теперь содержит Принципалов Брэда и Джоюзера.

Объект действия, который использует программную авторизацию, проверяет, находится ли joeuser в субъекте, и предоставляет ему доступ.

Объект действия, использующий декларативное разрешение, видит, что в файле политики есть разрешение для Брэда, поэтому он также предоставляет ему доступ.