Функции в языке программирования Ruby

Декомпозиция алгоритма

Если из алгоритма, реализующего сложную задачу, выделить несколько более простых «вспомогательных» алгоритмов, то исходный алгоритм будет являться комбинацией этих вспомогательных алгоритмов.

Метод как способ реализации вспомогательного алгоритма

Методы, наряду с переменными и операторами, являются основными строительными блоками в Ruby. Вспомогательные методы называют также функциями или подпрограммами.

Математическая аналогия

Функция — это четверка:
имя, откуда, куда, как.

$$f : X \to Y$$

Тело функции
(правило вычисления)

$$f(x) = x + 2$$

Вызов функции

$$f(5)$$

```ruby
def f(x)
  x+2
end

puts f(5)
```

```
7
```

Вызов метода

Метод должен быть определен до своего использования!

Каждый метод возвращает значение. Это результат последней выполненной операции или выражение, следующее за словом return.

Имя метода

Имя метода должно начинаться со строчной латинской буквы (от a до z). Называйте функции «говорящими» (осмысленными) именами.

Если имя состоит из нескольких слов, то их разделяют символом подчеркивания, или каждое слово, начиная со второго, выделяют заглавной буквой.

Метод без аргументов

```
def имя_метода
    ...
  тело_метода
    ...
end
```

Пример

```
1 def helloWorld # имя метода
2   puts "Hello, World!" # тело метода
3 end
4
5 helloWorld # вызов метода
```

```
Hello, World!
```

Метод с параметрами

```
def имя_метода(арг1, арг2, ...)
    ...
  тело_метода
    ...
end
```

Пример

```
1 def saySomething(text)
2     puts text
3 end
4
5 saySomething("Hello, World!")
6 saySomething("Привет, мир!")
```

```
Hello, World!
Привет, мир!
```

Вызов private метода

Методам, определяемым вне тела того или иного класса, присваивается специальная пометка private (частный), называемая *спецификатором доступа*. При обращении к такому методу используется функциональная форма, а не оператор вызова метода . (точка).

Вызов public метода

Методы, определенные внутри тела класса, по умолчанию помечаются как public. При вызове методов, определенных таким образом, используется «точечная» форма вызова метода.

Вызов метода

Так как наши программы пока не содержат определений новых классов, то для всех методов, разработанных нами, используется функциональный стиль вызова (аналогично `sin`, `sqrt` и т. п.).

Параметры метода

Параметры метода являются локальными переменными. Это означает, что метод оперирует копиями переменных, переданных ему (часто говорят, что параметры передаются по значению).

Пример

```
def printOneMoreThan(x)
    x += 1
    puts "Во время выполнения: #{x}"
end
x = 1; puts "До: #{x}"
printOneMoreThan(x)
puts "После: #{x}"
```

```
До: 1
Во время выполнения: 2
После: 1
```

Параметры по умолчанию

Параметры по умолчанию, дают возможность вызова метода без обязательного указания значения *всех* параметров.

Метод с параметрами, заданными по умолчанию

```
def имя_метода(арг1 = знач1, арг2 = знач2, ...)
      ...
   тело_метода
       ...
end
```

Пример

```
def saySomething(text = 'Hello, World!')
   puts text
end
saySomething
saySomething()
saySomething "Пробел между именем и аргументами"
saySomething(" или использование скобок")
```

```
Hello, World!
Hello, World!
Пробел между именем и аргументами
 или использование скобок
```

Вызов метода с аргументами

Рекомендуется всегда заключать аргументы метода в круглые скобки. Это позволит избежать некоторых ошибок и недоразумений. Общепринятыми исключениями являются методы печати — puts, print и т. д., хотя и при их вызове аргументы полезно заключать в скобки.

О методах, получающих массив в качестве параметра

Массив — это набор ССЫЛОК на некоторые элементы (их адресов в памяти компьютера). При передаче массива-параметра он копируется аналогично другим объектам (но копируются при этом ссылки!)

Операция [ ]= осуществляет переход по ссылке к соответствующей ячейки памяти и изменяет её содержимое. Следовательно, меняется и сам исходный массив.

```ruby
def modify(ar)
  for i in 0 ... ar.size
    ar[i] = ar[i]*2
  end
  return ar
end
a = [1, 2, 3, 4]
puts "Массив до вызова функции modify"
p a
puts "Вызвали функцию modify"
p modify(a)
puts "Массив после вызова функции modify"
p a
```

```
Массив до вызова функции modify
[1, 2, 3, 4]
Вызвали функцию modify
[2, 4, 6, 8]
Массив после вызова функции modify
[2, 4, 6, 8]
```

```ruby
def dont_modify(ar)
  arNew = ar.clone
  for i in 0 ... arNew.size
    arNew[i] = arNew[i]*2
```

```
  end
  return arNew
end
a = [1, 2, 3, 4]
puts "До вызова функции"
p a
puts "Вызвали функцию"
p dont_modify(a)
puts "Исходный массив не изменился!"
p a
```

```
До вызова функции
[1, 2, 3, 4]
Вызвали функцию
[2, 4, 6, 8]
Исходный массив не изменился!
[1, 2, 3, 4]
```

Глава 1

Комплексные числа в языке программирования Ruby

§ 1. Реализация на базе массива из двух элементов

Создание комплексного числа

Как известно, любому комплексному числу $z = x + yi$ соответствует точка плоскости с координатами (x, y). Поэтому для задания комплексного числа следует указать два действительных числа x и y. Самым простым способом хранения этих двух чисел в одном объекте является хранение их в массиве из двух элементов `z = [x, y]`, например,

```
z1 = [1, 1]
z2 = [0, -3]
```

Недостатком такого представления является отсутствие проверки на количество элементов массива и соответствие типа каждого из элементов массива одному из числовых типов (Integer, Float). Предполагается, что пользователь программы сам контролирует корректность ввода данных и их использования.

Печать комплексного числа

Одной из самых распространенных операций для любого типа данных является их представление в виде строки, которое используется при их печати. Поэтому первой функцией, которую мы реализуем будет функция `comp_str` для заданного комплексного числа `z = [a, b]` возвращающая строку `a+bi` (без пробелов и знака умножения), напри-

мер, для числа, заданного массивом `[1, 2]` будет возвращена строка `"1+2i"` , а для числа `[0, -3]` — `"0-3i"`.

```
def comp_str(z)
  "#{z[0]}+#{z[1]}i"
end
puts comp_str(z1)
```

```
1+1i
```

На первый взгляд функция работает правильно, однако, если мы попробуем применить её к числу z2, то получим не тот результат, который ожидали:

```
puts comp_str(z2)
```

```
0+-3i
```

Изменим функцию так, чтобы знак мнимой части отображался корректно:

```
def comp_str(z)
  "#{z[0]}"+(z[1] < 0 ? '-' : '+')+"#{z[1].abs}i"
end
puts comp_str(z1)
puts comp_str(z2)
```

```
1+1i
0-3i
```

Сложение комплексных чисел

Определим функцию `plus`, возвращающую сумму двух комплексных чисел:

```
def plus(z, w)
  [z[0]+w[0], z[1]+w[1]]
end

z3 = plus(z1, z2)
puts comp_str(z3)
```

```
1-2i
```

Умножение комплексных чисел

Функция `mult` предназначена для вычисления произведения двух комплексных чисел:

```
def mult(z, w)
  [z[0]*w[0] - z[1]*w[1], z[0]*w[1] + z[1]*w[0]]
end
z4 = mult(z1, z2)
puts comp_str(z4)
```

```
3-3i
```

Вычисление модуля комплексного числа

Так как в нашей программе используется функция с именем `abs`, то нам придется дать функции для определения модуля комплексного числа какое-нибудь другое имя. Пусть эта функция называется `dist`, т. к. модуль комплексного числа $z = x + yi$ есть расстояние от точки с координатами (x, y) до начала координат.

```
def dist(z)
  Math.sqrt(z[0]**2+z[1]**2)
end
puts dist(z1)
```

```
1.4142135623731
```

Упражнения

1. Измените функцию `comp_str` так, чтобы нулевые действительная или мнимая части числа не отображались, например для `z = [0, -2]` результат работы функции должен равняться `"-2i"`.

2. Измените функцию `comp_str` так, чтобы равный 1 или -1 коэффициент мнимой части не печатался, например, для `z = [2, -1]` функция должна возвратить значение `"2-i"`.

3. Напишите функцию вычитания двух чисел `minus`.

4. Напишите функцию `conjugate`, находящую сопряженное число.

5. Напишите функцию `divide`, определяющую частное двух комплексных чисел.

Глава 1

Объектно-ориентированное программирование

§ 1. ООП — результат эволюции методологий программирования

Потребность в ООП связана со стремительным усложнением разрабатываемых программ и, как следствие, их недостаточной надежностью. Объектно-ориентированный подход позволяет программисту писать более надежный код, избавленный от многих ошибок, которые могут проявиться в коде, разработанном в процедурном стиле.

Другим немаловажным качеством ООП является возможность многократного использования кода. Реализованные однажды библиотеки классов программист может использовать в различных программах, что обеспечивает повышение производительности труда программиста и увеличивает надежность готового программного продукта.

Практически все современные языки программирования, независимо от принадлежности к тому или иному стилю (директивному или декларативному), поддерживают концепцию ООП. Среди них C++, Java, Ruby.

Основные свойства ОО стиля

Программа, реализованная в объектно-ориентированном стиле программирования, представляет собой некий сценарий взаимодействия *объектов*, которые являются *представителями того или иного класса*.

Малозначащие детали объекта (экземпляра класса) скрыты от пользователя. Взаимодействие между объектами происходит путем передачи (посылки) им сообщений, на которые они реагируют заранее предопределённым способом. Если дается команда какому-то объекту, то он «знает», как ее выполнить. *Фундаментальной концепцией в ООП является понятие обязанности или ответственности за выполнение действия*.

Можно сказать, что в основе ООП лежат три понятия:

- инкапсуляция (сокрытие данных в классе или методе);
- наследование;
- полиморфизм.

Инкапсуляцию данных можно представить, как защитную оболочку вокруг кода данных, с которыми этот код работает. Оболочка задает поведение и защищает код от произвольного доступа извне. Рассмотрим пример из повседневной жизни. Пусть у нас есть стиральная машина, которая по нажатию на кнопку пуска начинает процесс стирки. Пользователь данной машины не обязан знать, каким образом реализована функция стирки в машине: в какой плоскости вращается барабан, с какой скоростью, и т. д. Механизм функционирования машины скрыт от пользователя (инкапсулирован внутри корпуса машины). Все изменения, производимые разработчиками данного изделия (например, изменение направления движения барабана для улучшения качества стирки) не должны влиять на процесс управления данным изделием. Так и в программировании: разработчик того или иного класса может вносить изменения в код класса, но они не должны отражаться на программах, которые используют данные классы.

Все объекты являются представителями, или *экземплярами*, классов. Метод, активизируемый объектом в ответ на сообщение, определяется классом, к которому принадлежит получатель сообщения. Все объекты одного класса используют одни и те же методы в ответ на одинаковые сообщения.

Классы представляются в виде иерархической древовидной структуры (*иерархии классов*), в которой классы с более общими чертами (родительские классы) располагаются в корне дерева, а специализированные классы (дочерние) и в конечном итоге индивидуумы располагаются в ветвях.

Наследование — это процесс, в результате которого один класс (тип) наследует свойства другого. Наследование позволяет избежать дублирования кода: общие свойства нескольких дочерних классов следует реализовывать в родительском классе.

Использование классов и наследование — одна из причин повышения надежности кода, так как если для класса не задан тот или метод, то он и не будет выполнен. Так, например, для экземпляров класса String не определена операция ** (возведение в степень), поэтому при попытке выполнить эту операцию мы получим сообщение об ошибке:

```
a = "hello"
b = "world"
puts a**b
```

```
undefined method `**' for "hello":String (NoMethodError)
```

Полиморфизм — это концепция, позволяющая иметь различные реализации для одного и того же метода, которые будут выбираться в зависимости от типа объекта, переданного методу при вызове. Например, команда «взлет», примененная к самолету, приведет к всё более убыстряемому горизонтальному движению самолета по земле и последующему полету, в то время как эта команда, примененная к вертолету, вызовет вращение лопастей вертолета и вертикальному взлету.

Примером полиморфизма в программировании может случить метод +, определенный для многих классов:

```
puts 10 + 4
p [2, 3, 4] + [1, 3, 4]
```

```
14
[2, 3, 4, 1, 3, 4]
```

§ 2. Разработка класса Complex

Создание класса в Ruby

Для создания класса в Ruby следует весь код, относящийся к данному классу, заключить между ключевыми словами `class` и `end`.

```
class Имя_класса
   def initialize(параметры)
     ...
   end
   def один_из_методов
      ...
   end
```

```
  ...
end
```

По традиции, имя класса должно начинаться с заглавной буквы, а имена методов — со строчной. Для вызова метода указывается имя экземпляра, к которому собираются применять метод, символ . (точка) и имя метода, например, `"hello".size, [2, 3, 4].shift`.

Метод с зарезервированным именем `initialize` является конструктором класса. Он вызывается, когда мы создаем экземпляр класса (указывая ключевое слово `new`). С его помощью можно создавать экземпляры класса с различными характеристиками (задавая различные аргументы при вызове).

Информация об особенностях того или иного экземпляра класса хранится, в так называемых, переменных экземпляра; имя этих переменных должно начинаться с символа `@`. Для каждого экземпляра в программе создаются отдельные наборы таких переменных. Эти переменные доступны в любом методе класса.

Дальнейший рассказ об особенностях объектно-ориентированного программирования в языке Ruby мы будем иллюстрировать примерами разработки класса комплексных чисел `Complex`. Любой представитель данного класса однозначно характеризуется двумя числовыми значениями, определяющими действительную и мнимую части числа, поэтому при создании нового экземпляра комплексного числа следует задать значения двух переменных экземпляра.

```
class Complex
  def initialize(x, y)
    @re, @im = x, y
  end
end
z1 = Complex.new(1, 2)
z2 = Complex.new(0, -2)
```

В языке Ruby имеется специальный метод `inspect`, который можно применить к любому объекту. По умолчанию он создаёт строку, содержащую уникальный идентификатор данного объекта (ID) и значения его переменных экземпляра.

```
puts z1.inspect
puts z2.inspect
```

```
#<Complex:0xbf51ce6c @image=2, @real=1>
#<Complex:0xbf51ce1c @image=-2, @real=0>
```

Строковое представление объекта

Мы видим, что метод `inspect` печатает всю информацию об объекте, но вид выводимой информации оставляет желать лучшего. К счастью, Ruby позволяет послать объекту сообщение в виде метода `to_s`, который преобразует объект к строковому виду. По умолчанию этот метод печатает ID объекта:

```
puts z1.to_s
puts z2.to_s
```

```
#<Complex:0xbf51ce6c>
#<Complex:0xbf51ce1c>
```

Но, нам хотелось бы получить стандартное представление комплексного числа, например, из объекта, созданного командой `Complex.new(2,3)`, получить строку вида `2+3i`.

Ruby позволяет программисту в любой момент изменить тот или иной метод класса. Давайте определим метод `to_s` для класса Complex. В теле метода для преобразования к строковому виду чисел `@re` и `@im` (действительной и мнимой частей числа) воспользуемся методом `to_s`, уже определенным в Ruby для чисел, и не забудем учесть знак мнимой части:

```
def to_s
  @re.to_s + (@im < 0 ? '' :'+') + @im.to_s + "i"
end
```

Отметим, что метод `to_s` автоматически вызывается всегда, когда требуется строковое представление объекта, например, при использовании методов `puts` или `print`.

```
z1 = Complex.new(1,2)
z2 = Complex.new(3,-2)
puts z1.to_s
puts z1
puts z2
```

```
1+2i
1+2i
3-2i
```

Объекты и атрибуты

Принцип инкапсуляции не позволяет «видеть» служебную информацию класса и его экземпляра (в том числе и переменные) извне. Но если бы объект был бы полностью закрыт от посторонних взглядов, то он был бы абсолютно бесполезен. Назовем атрибутами те части объекта, которые мы вынуждены приоткрыть для внешнего мира. Определим методы, позволяющие просматривать или при необходимости изменять атрибуты объекта.

В случае с комплексными числами следует задать возможность узнать действительную и мнимые части числа:

```
  def re
    @re
  end
  def im
    @im
  end
```

```
puts z1.re
puts z2.im
```

```
1
-2
```

Методы `re` и `im` возвращают значения соответствующих переменных экземпляра. Если бы в нашем классе было бы значительно больше атрибутов, предназначенных для просмотра, то написание подобных методов стало бы утомительным. Поэтому используется специальная конструкция языка Ruby, которая позволяет без написания методов узнать значение того или иного атрибута, т.е. указать *атрибуты чтения*:

```
  attr_reader :re, :im
```

Если возникает необходимость в изменении атрибутов, то используют конструкцию `attr_writer`, если требует читать и изменять атрибуты, то можно воспользоваться конструкцией `attr_accessor`.

Сложение комплексных чисел

Определим метод `plus`, возвращающий сумму двух комплексных чисел:

```
  def plus(z)
```

```
    Complex.new(@re+z.re, @im+z.im)
  end
```

Данная функция создает новый экземпляр класса Complex, действительная и мнимая части, которого получены путем сложения соответствующих атрибутов исходного объекта и аргумента метода:

```
z = z1.plus(z2)
puts z
```

```
4+0i
```

Однако, мы привыкли пользоваться *оператором* сложения +, который размещается между своими аргументами: 2+3. В Ruby список операторов фиксирован (в него входят +, -, *, /, ** и некоторые другие), нельзя создать новый оператор, но, реализуя возможность полиморфизма, которую мы обсуждали выше, можно определить эти операторы для вновь создаваемых классов (и даже переопределить ранее определённые):

```
  def +(z)
    Complex.new(@re+z.re, @im+z.im)
  end
```

```
w = z1 + z2
puts w
```

```
4+0i
```

Равенство комплексных чисел

По опрделению, два комплексных числа равны, если совпадают их действительные и мнимые части.

```
  def == (other)
    @re == other.re and @im == other.im
  end
```

```
puts Complex.new(1+1,1*(-1)) == Complex.new(2.0,-1)
```

```
true
```

Обратите внимание, что атрибуты класса Complex в нашей реализации могут принадлежать как классу `Integer`, так и классу `Float`.

Определение полярных координат комплексного числа

```
  def abs
    Math.sqrt(@re**2+@im**2)
  end
  def arg
    Math.atan2(@im.to_f,@re)
  end
  def polar
    return abs, arg
  end
```

```
puts z1.abs
puts z1.arg
puts z1.polar
```

```
1.4142135623731
0.785398163397448
1.4142135623731
0.785398163397448
```

Методы класса

Иногда возникает необходимость в создании методов, которые нельзя применить к какому-либо конкретному экземпляру класса. Например, пусть нам требуется создать комплексное число, указав его полярные координаты. В подобной ситуации нельзя использовать конструктор класса, так как он создает число по его действительной и мнимой частям. В этом случае используют специальную синтаксическую конструкцию — метод класса. Если перед именем метода указать имя класса, то это означает, что применяться он будет не к конкретному экземпляру, а к самому классу.

```
  def Complex.polar(r,theta)
    Complex.new(r*Math.cos(theta), r*Math.sin(theta))
  end
```

```
z2 = Complex.polar(Math.sqrt(2), Math::PI/4)
puts z2
```

```
1+1.0i
```

Отметим, что методы `polar` и `Complex.polar` — это различные методы: первый применяется к экземпляру и возвращает его полярные координаты (два числа), а второй — к классу и создает новый экземпляр класса.

О точности вычислений

В домашнем задании по математике была задача перевода комплексного числа из алгебраической формы в тригонометрическую. Представленные выше методы экземпляра `abs` и `arg` могли бы нам помочь в решении данной задачи. Но, к сожалению, результаты работы этих методов являются числами, а нам хотелось бы видеть символьные значения: вместо числа 0.785398163397448 — величину $\pi/4$. Тем не менее, мы можем попытаться использовать нашу программу для проверки полученных на бумаге ответов. Например, для числа $z = 1 - i$, укажем действительную и мнимую части, и сравним полученное число с числом, заданным с помощью полярных координат (с модулем $r = \sqrt{2}$ и аргументом $\varphi = -\pi/4$):

```
z3 = Complex.new(1, -1)
puts z3
z4 = Complex.polar(Math.sqrt(2), -Math::PI/4)
puts z4
```

```
1-1i
1-1.0i
```

Мы видим, что они равны. Давайте напечатаем результат проверки на равенство этих чисел:

```
puts z3 == z4
```

```
false
```

Несколько неожиданный результат... Возможно, он вызван тем, что атрибуты `z3` — целые числа. Изменим `z3`:

```
z3 = Complex.new(1.0, -1.0)
puts z3
puts z3 == z4
```

```
1.0-1.0i
false
```

Результат тот же. Значит, причина в другом.

Визуально, действительные и мнимые части чисел равны. Но попробуем напечатать разность атрибутов данных чисел:

```
puts (z3.re-z4.re).abs
puts (z3.im-z4.im).abs
```

```
2.22044604925031e-16
0.0
```

Оказывается, атрибуты `@re` этих объектов действительно различны![1] Попробуем разобраться в чем же дело. Из курса математики мы знаем, что $\sin(\pi/4) = \cos(\pi/4) = \sqrt{2}/2$, а равны ли эти значения при вычислении этих значений на компьютере?

```
puts Math.sin(Math::PI/4)
puts Math.cos(Math::PI/4)
puts Math.sin(Math::PI/4) - Math.cos(Math::PI/4)
```

```
0.707106781186547
0.707106781186548
-1.11022302462516e-16
```

Множествам целых $\mathbb{Z}$ и действительных чисел $\mathbb{R}$ в большинстве языков программирования соответствуют их машинные аналоги. В случае языка Ruby используемые в программах переменные и константы классов Integer и Float принимают значения из множеств $\mathbb{Z}_M$ и $\mathbb{R}_M$ соответственно.

Мы уже отмечали, что Ruby в отличии от многих других языков позволяет оперировать целыми числами из практически неограниченного диапазона значений (ограничение накладывается только возможностями вашего компьютера). Пока целые числа находятся в диапазоне от -2^{30} до 2^{30-1}, они являются экземплярами класса Fixnum. Если в результате расчетов получившийся результат не входит в указанный диапазон, то он автоматически преобразуется в экземпляр класса Bignum и наоборот, если в результате работы с объектами класса Bignum, получается число из диапазона от -2^{30} до 2^{30-1}, то оно автоматически преобразуется к объекту типа Fixnum.

Отметим, что метод преобразования к строковому виду `to_s`, применимый к объектам класса Integer, допускает указание аргумента —

[1] Первое из чисел при печати было преобразовано к экспоненциальной форме, так как очень мало (напомним, что величина AeB обозначает выражение $A \cdot 10^B$).

целого числа (от 2 до 36), являющегося основанием системы счисления, в которой надо отобразить число.

```
puts (2**50).to_s
puts (2**50).to_s(2)
puts (2**50).to_s(16)
puts (2**50).to_s(31)
```

```
1125899906842624
10000000000000000000000000000000000000000000000000
4000000000000
1bi31arsea1
```

Ситуация с вещественными числами во всех языках программирования (и Ruby здесь не исключение) значительно сложнее. В отличии от целых чисел $\mathbb{Z}$ вещественные числа $\mathbb{R}$ обладают свойством полноты: между любыми двумя различными числами всегда найдется отличное от них третье. Понятно, что невозможно сохранить это свойство бесконечного множества действительных чисел, как бы мы ни представляли их в виде конечного множества $\mathbb{R}_M$. Наиболее распространенным способом реализации вещественных чисел является использование чисел с плавающей точкой. При этом множество $\mathbb{R}_M$ оказывается состоящим из элементов вида

$$f = \pm\left(\frac{\alpha_1}{p} + \frac{\alpha_2}{p^2} + \ldots + \frac{\alpha_n}{p^n}\right) p^e,$$

где целые числа α_i для любого i из диапазона от 1 до n удовлетворяют соотношению $0 \leqslant \alpha_i \leqslant p-1$, а величина e, называемая экспонентой, лежит в диапазоне от L до U ($L \leqslant e \leqslant U$). Число p является основанием системы счисления (чаще всего это 2), константа n задает точность представления чисел, а диапазон $[L, U]$ определяет область значений экспоненты. Число $\pm\left(\frac{\alpha_1}{p} + \frac{\alpha_2}{p^2} + \ldots + \frac{\alpha_n}{p^n}\right)$ принято называть мантиссой числа f с плавающей точкой.

Множество $\mathbb{R}_M$ симметрично относительно начала координат; нуль принадлежит нашему множеству, Ближайшая к нулю положительная точка получается при $\alpha_1 = 1, \alpha_2 = \ldots = \alpha_n = 0$, $e = L$; т.е. в двоичной системе счисления задается формулой $f_{min} = \frac{1}{2} \cdot 2^L$.

В языке Ruby наименьшее из положительных чисел класса Float представляется константой `MIN`, максимальное — константой `MAX`, значения L и U задаются константами `MIN_EXP` и `MAX_EXP` соответственно, а число n — константой `DIG`:

```
puts Float::MIN
puts Float::MAX
puts Float::MAX_EXP
puts Float::MIN_EXP
puts Float::DIG
```

```
2.2250738585072e-308
1.7976931348623157e+308
1024
-1021
15
```

Убедимся, что величина константы `MIN` совпадает со значением f_{min}:

```
puts Float::MIN
puts 0.5*2**Float::MIN_EXP
```

```
2.2250738585072e-308
2.2250738585072e-308
```

Кроме вышеперечисленных констант в классе Float определена ещё одна — `EPSILON`, она определяется из условия, что величина `EPSILON` есть наименьшее из всех x, таких что $1.0 + x \neq 1.0$.

```
puts Float::EPSILON
```

```
2.22044604925031e-16
```

Возвращаясь к задаче сравнения двух чисел, переопределим метод проверки на равенство двух чисел с учетом того, что два числа следует считать равными, если их разность не превышает `EPSILON`:

```
def == (other)
  (@re - other.re).abs <= Float::EPSILON and
    (@im - other.im).abs <= Float::EPSILON
end
```

```
1-1i
1-1.0i
true
```

Упражнения

1. В разработанном нами классе Complex

1) Измените метод `to_s` так, чтобы нулевые действительная или мнимая части числа не отображались.
2) Измените метод `to_str` так, чтобы равный 1 или −1 коэффициент мнимой части не отображался, например, строковое представление числа `z = Complex.new(2, -1)` должно совпасть со строкой `"2-i"`.

2. Определите оператор `-` для вычисления разности двух комплексных чисел.

3. Задайте метод `conjugate`, вычисляющий сопряженное комплексное число.

4. Определите оператор `/`, определяющий частное двух комплексных чисел.

Глава 2

Стандартные библиотеки классов

В начале предыдущей главы уже упоминалось о том, что использование объектно-ориентированного программирования позволило облегчить труд программистов за счет многократного использования стандартных библиотек классов. Не следует каждый раз заново «изобретать велосипед»: множество классов уже разработано и включено в стандартную поставку языка. Для Ruby создан специальный сайт RAA (Ruby Application Arhiv), предназначенный для размещения новых проектов. Если Вы не нашли нужного Вам класса среди поставляемых вместе с Ruby, то попробуйте поискать на указанных сайтах.

§ 1. Класс Complex

Комплексные числа не являются исключением и библиотека для работы с этим классом входит в дистрибутив языка Ruby. Если в Вашей системе установлена программа `ri`, позволяющая просматривать описание классов Ruby, то по команде `ri Complex`, Вы получите следующее описание:

```
------------------------------ Class: Complex < Numeric
 The complex number class. See complex.rb for an overview.
------------------------------------------------------------
Constants: I: Complex(0,1)
----------
Class methods: new, new!, polar
--------------
Instance methods: %, *, **, +, -, /, <=>, ==, abs, abs2,
```

```
    angle, arg, coerce, conj, conjugate, denominator, hash,
    inspect, numerator, polar, to_s
Attributes: image, real
```

Как можно заметить, атрибуты и методы очень похожи на разрабатываемые нами.

Для того чтобы воспользоваться библиотекой, содержащей этот класс, следует подключить его командой `require`, и затем можно уже создавать новые объекты — комплексные числа.

```
require 'complex'
z = Complex.new(0, -1)
puts z
puts z**50
puts (z**50).polar
```

```
-1i
-1+0i
1.0
3.14159265358979
```

А как работает сравнение комплексных чисел?

```
w = Complex.polar(Math::sqrt(2), -Math::PI/4)
puts w
puts Complex.new(1, -1) == w
class Complex
  def == (other)
```

```
1-1.0i
false
```

На этот раз мы уже ожидали подобный результат ожидаем и знаем как бороться с этой проблемой. Придётся подправить метод `==`.

Заметим, что классы в языке Ruby — открыты, т.е. программист в любой момент может добавить к классу новый метод или атрибут, или изменить уже определенные. Для этого следует открыть определение класса и вписать в него требуемый код:

```
class Complex
  def == (other)
    (@re - other.re).abs <= Float::EPSILON and
      (@im - other.im).abs <= Float::EPSILON
  end
```

```
end
puts Complex.new(1, -1) == w
```

```
true
```

Упражнения

1. Дополните класс Complex методом `sqrt`, извлекающий квадратный корень из комплексного числа.

2. Измените метод `to_str` так, чтобы равный 1 или -1 коэффициент мнимой части не отображался, например, строковое представление числа `z = Complex.new(2, -1)` должно совпасть со строкой `"2-i"`.

3. Определите метод `to_str` так, чтобы нулевые части числа не печатались.

Глава 1

Графический интерфейс

Как и для большинства скриптовых языков, не имеющих встроенных средств для создания оконных приложений, для Ruby разработано несколько графических интерфейсов (GUI — graphical user interface), то есть средств создания оконных приложений и размещений в них различных компонент: кнопок, меток, областей для отображения графических примитивов (линий, окружностей и т.п.). Среди них: Tk, GTK+, FOX, wxWindows. Так как современное программирование предполагает наличие навыков работы с графическими интерфейсами, то рассмотрим одно из средств создания GUI — Tk, входящее в состав всех современных дистрибутивов Ruby, являющееся межплатформенным и широко распространенным средством создания оконных приложений.

§ 1. Tk — стандартный GUI для Ruby

Интерфейс Tk был разработан как GUI для скриптового языка Tcl еще в середине 80-х годов прошлого века, однако вскоре был адаптирован в качестве межплатформенного (то есть выполняемого в различных операционных системах) графического интерфейса для всех популярных скриптовых языков (включая Perl и Python).

В Ruby библиотека для работы с Tk содержит большое число классов, обеспечивающих возможность использования всех его возможностей. Типичная структура программы на Ruby, используйщей Tk, образуется путем создания главного или «корневого» (root) окна (экземпляра класс TkRoot), добавления к нему различных компонент (widgets) и, наконец, запуска основного цикла вызовом метода `Tk.mainloop`. Тра-

дициoнный пример в Ruby/Tk для печати приветствия «Hello, World!» выглядит так:

```
require 'tk'
root = TkRoot.new
button = TkButton.new(root) {
  text "Hello, World!"
  command proc { puts "I said, Hello!" }
}
button.pack
Tk.mainloop
```

При каждом нажатии на клавишу с меткой «Hello, World!» на консоль выводится текст приветствия.

§ 2. Создание вспомогательного класса на базе Tk

Так как мы планируем использовать Tk для «рисования» в окнах геометрических объектов, графиков или множеств на плоскости, ограничим рассмотрение только некоторыми возможностями Ruby/Tk. Создадим класс, реализующий нужные нам возможности.

```
require "tk"

class TkDraw
  def TkDraw.create(h = 200, w = 200, name = "Tk")
    @@height, @@width = h, w
    @@root = TkRoot.new{
      title name
      geometry "#{w}x#{h}"
    }
    @@canvas = TkCanvas.new(@@root,
                            "height" => h,
                            "width" => w)
    @@canvas.pack{ padx 5; pady 5 }
    Thread.new{Tk.mainloop}
    TkcRectangle.new(@@canvas, 0, 0, h, w,
                     "width"=>0) { fill("white") }
  end
  def TkDraw.point(x, y, color)
    TkcLine.new(@@canvas, x, y, x+1, y) { fill(color) }
  end
```

```
    def TkDraw.line(x1, y1, x2, y2, color)
      TkcLine.new(@@canvas, x1, y1, x2, y2) { fill(color) }
    end
    def TkDraw.label(x, y, str, size)
      TkcText.new(@@canvas,
                  @@canvas.canvasx(x),
                  @@canvas.canvasy(y) ) do
        font TkFont.new(['helvetica', size, ['bold']])
        text(str)
        tags(text)
      end
    end
    def TkDraw.oval(x1, y1, x2, y2, color, w = 0)
      TkcOval.new(@@canvas, x1, y1, x2, y2) {
        fill(color)
        width(w)
      }
    end
    def TkDraw.clean
      TkcRectangle.new(@@canvas, 0, 0, @@width, @@height,
                       "width"=>0) {fill("white")}
    end
  end
```

Не будем подробно объяснять назначение каждой строчки этого кода, отметим только, что в данном классе определены методы создания окна нужного размера, очистки его от всех объектов, рисования точки, линии и овала указанным цветом, помещение текста заданного размера в область окна.

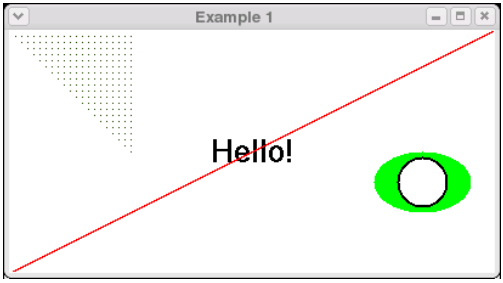

Рис. 1.1.

Следующая программа с помощью данного класса создает окно, содержащее группу точек, линию и надпись.

```
require "TkDraw"
TkDraw.create(200, 400, "Example 1")
TkDraw.clean
0.step(100, 5) { |x|
  0.step(x, 5) { |y|
    TkDraw.point(x, y,"#244a0d")
  }
}
TkDraw.label(200, 100, "Hello!", 12)
TkDraw.line(0, 200, 400, 0, "red")
TkDraw.oval(300, 100, 380, 150, "green")
TkDraw.oval(320, 105, 360, 145, "white", 2)
Tk.mainloop
```

Этот пример показывает, что точка с координатами $(0, 0)$ находится в верхнем левом углу окна, текст по умолчанию центрируется относительно указанных координат, а цвет объекта задается либо указанием наименования цвета, либо его шестнадцатиричного кода.

§ 3. Преобразование координат

Итак, мы видели, что координаты (i, j) каждого пиксела окна меняются от 0 до некоторой величины, указанной при создании окна. Это, так называемые, *экранные координаты*. Однако изначально декартовы координаты (x, y) точек могут быть заданы вещественными числами, вроде $x \in [0, 1], y \in [-1, 1]$. Последние мы будем называть *мировыми координатами*.

Из мировых в экранные

Пусть $x_э \in [0, w]$, $y_э \in [0, h]$, а $x_м \in [a, b]$, $y_м \in [c, d]$. Преобразование координат от мировых к экранным задается следующими условиями: $a \mapsto 0$; $b \mapsto w$; $c \mapsto h$; $d \mapsto 0$. Очевидно, что прямоугольное окно на экране получается из исходного прямоугольника в результате растяжения и сдвига, то есть функция преобразования является линейной функцией:

$$x_э = f(x_м) = p_1 \cdot x_м + q_1,$$
$$y_э = f(y_м) = p_2 \cdot y_м + q_2.$$

Подставив в уравнения вместо $x_\text{М}$ величины a и b, а вместо $y_\text{М}$ числа c и d, найдем формулы преобразования мировых координат в экранные:

$$x_\text{Э} = \frac{w}{b-a}(x_\text{М} - a); \quad y_\text{Э} = \frac{h}{c-d}(y_\text{М} - d).$$

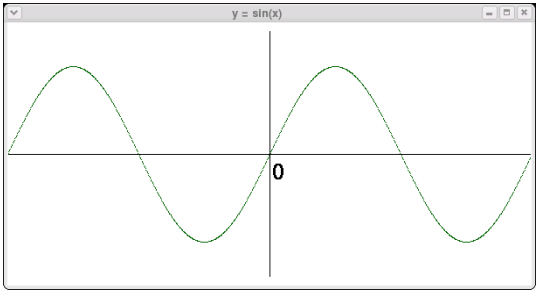

Рис. 1.2.

Теперь мы можем приступать к изображению на экране графиков функций. Пусть требуется изобразить график функции $f(x)$, где $x \in [a, b]$. Как правило, отрезок $[a, b]$, которому принадлежат значения аргумента, делится на n равных частей и значения функции вычисляются последовательно в точках $x_0 = a, x_1 = x_0 + \dfrac{b-a}{n}, x_2 = x_1 + \dfrac{b-a}{n}, \ldots$ Для каждой точки $\big(x_i, f((x_i)\big)$ осуществляется преобразование координат в экранные и с помощью метода `TkDraw.point` рисуется точка на экране. Заметим, что если n мало, то мы получим отдельно расположенные точки, а не сплошную линию.

Пример 3.1. Следующая программа для $x \in [-2\pi, 2\pi]$ рисует график функции $y = \sin(x)$ (рис. 1.2).

```
require "TkDraw"
include Math

h, w = 300, 600
a, b = -2*PI, 2*PI
c, d = -1.5, 1.5
n = 1000 # число точек для построения графика

TkDraw.create(h, w, "y = sin(x)")
TkDraw.clean
```

```
a.step(b, (b-a)/n) { |xw|
  yw = sin(xw)
  xs = w/(b-a)*(xw - a)
  ys = h/(c-d)*(yw - d)
  TkDraw.point(xs, ys,"darkgreen")
}
TkDraw.label(310, 170, "0", 24)
TkDraw.line(0, 150, 600, 150, "black")
TkDraw.line(300, 10, 300, 290, "black")
Tk.mainloop
```

Из экранных в мировые

В ряде задач требуется изобразить все точки, удовлетворяющие некоторому условию, например, $|x| + |y| = 1$. В таком случае лучше исходить не из мировых, а из экранных координат.

Найдем формулы преобразования экранных координат в мировые, ограничившись для простоты случаем работы с квадратным окном. Пусть точка с мировыми координатами (a, b) задает середину квадрата со стороной s. Тогда для окна, размером $p \times p$ пикселей, имеем следующие зависимости: точка с мировыми координатами $(a - s/2, b + s/2)$ переходит в точку с экранными координатами $(0, 0)$, а точка (a, b) в точку $(p/2, p/2)$. Нетрудно показать, что преобразование в данном случае осуществляется по формулам

$$x_{\text{м}} = f(x_{\text{э}}) = \frac{s}{p} \cdot x_{\text{э}} + a - \frac{s}{2}; \quad y_{\text{м}} = f(y_{\text{э}}) = \frac{-s}{p} \cdot y_{\text{э}} + b + \frac{s}{2}.$$

Пример 3.2. Вот программа, изображающая множество точек плоскости, удовлетворяющих условию $|x| + |y| = 1$.

```
require "TkDraw"
include Math

p = 600
a, b, s = 0, 0, 3.0

TkDraw.create(p, p, "|x| + |y| = 1")
TkDraw.clean
for x in 1 .. p
  for y in 1 .. p
    xw = a - s/2 + x*s/p
```

```
      yw = b + s/2 - y*s/p
      if xw.abs + yw.abs == 1
        TkDraw.point(x, y, "midnightblue")
      end
    end
  end

  Tk.mainloop
```

Как и следовало ожидать получился квадрат (рис. 1.3). Но почему линия, его ограничивающая, не сплошная? Это снова проявилась проблема машинного представления действительных чисел. В результате преобразования мы получаем машинные аналоги мировых координат, для которых данное *точное* соотношение не выполнено. Если мы заменим условие точного равенства на неравенство, то изображение получится более четким (рис. 1.4).

```
      if (xw.abs + yw.abs - 1).abs < s/p
```

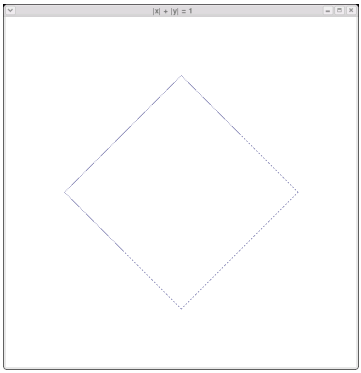

Рис. 1.3.

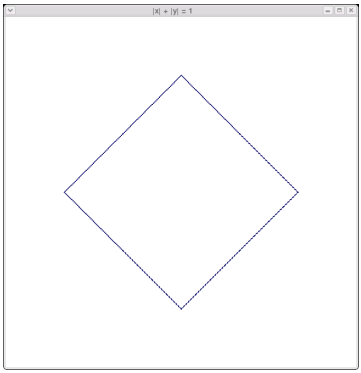

Рис. 1.4.

Пример 3.3. Следующая программа рисует множество, заданное условием

$$(x^2+y^2-25)(16x^2+y^2-4)(x^2+16y^2+96y+140)\times \\ \times(4x^2-16x\operatorname{sign}x+4y^2-16y+31)=0$$

Напомним, что функция sign задает знак числа, она равна -1 для отрицательных чисел, 1 — для положительных и 0 для нуля.

```
require "TkDraw"
```

```
p = 500
a, b, s = 0, 0, 15.0
eps = 0.5
def sign(x)
  return (if x < 0 then -1
          elsif x > 0 then 1
          else 0
          end)
end
def f1(x, y)
  x**2+y**2-25
end
def f2(x, y)
  16*x**2+y**2-4
end
def f3(x, y)
  x**2+16*y**2+96*y+140
end
def f4(x, y)
  4*x**2-16*x*sign(x)+4*y**2-16*y+31
end

TkDraw.create(p, p, "Face 1")
TkDraw.clean
for x in 1 .. p
  for y in 1 .. p
    xw = a - s/2 + x*s/p
    yw = b + s/2 - y*s/p
    if f1(xw, yw).abs < eps ||
                f2(xw, yw).abs < eps ||
                f3(xw, yw).abs < eps ||
                f4(xw, yw).abs < eps
      TkDraw.point(x, y, "firebrick")
    end
  end
end

Tk.mainloop
```

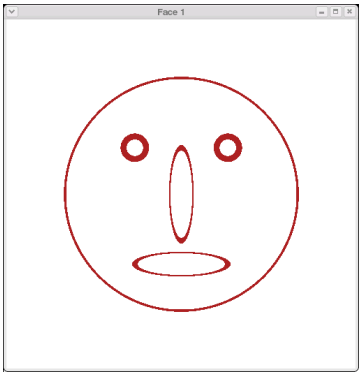

Рис. 1.5.

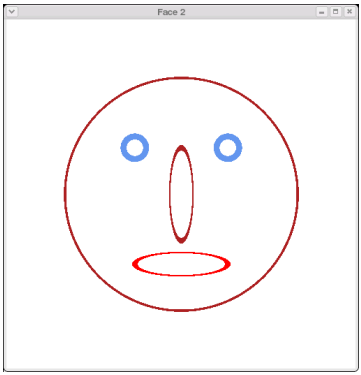

Рис. 1.6.

Упражнения

1. Выведите формулы преобразования мировых координат в экранные, приведенные на стр. 6.

2. Напишите программу рисования графиков, соединяющую две последовательные точки отрезком прямой (совет: добавьте еще две переменные, в которых будут храниться координаты предыдущей точки).

3. Измените программу, приведенную на стр. 8 так, чтобы отдельные элементы лица выделялись цветом (рис. 1.6).

4. Напишите программу для изображения множества точек плоскости, заданных условием $|y + |y|| = |x - |x||$. Что представляет собой данное множество?

5. Напишите программу для изображения множества точек плоскости, заданных условием

$$\big((4y + x^2)^2 + \operatorname{sign}(x^2 + 2x) + 1\big) \cdot \big((x^2 + y^2)^{5/2} - 4x(x^2 - y^2)\big) = 0.$$

§ 4. Итерированные функции

Пусть у нас имеется некоторая функция $f(x)$. Как ведет себя последовательность $f(x)$, $f(f(x))$, $f(f(f(x)))$, ... для различных значений аргумента? Поиск ответа на этот вопрос для функций комплексного аргумента привел к разработке такого интересного направления современной математики как *алгебраические фракталы*.

Рассмотрим функцию $f(z) = z^2 + c$, где c есть некоторая комплексная постоянная. Оказывается, иногда данная последовательность сходится к некоторой точке, а иногда «устремляется» в бесконечность. Давайте проследим за значениями этой функции, стартующей в точке $(0, 0)$ для разных значений комплексной константы.

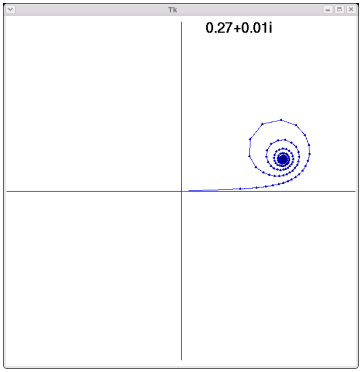

Рис. 1.7.

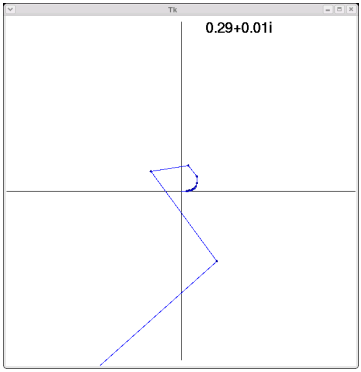

Рис. 1.8.

Рисунок 1.7 получен с помощью следующей программы:

```
require "TkDraw"
require 'complex'
include Math

h, w = 600, 600
a, b = -0.8, 0.8
c, d = -0.8, 0.8
cons = Complex.new(0.27, 0.01)

TkDraw.create(h,w)
TkDraw.clean
TkDraw.label(400, 20, cons, 24)
TkDraw.line(0, 300, 600, 300, "black")
TkDraw.line(300, 10, 300, 590, "black")

z = Complex.new(0, 0)
x_old, y_old = 300, 300

for n in 1 .. 500
```

```
    z = z**2 + cons
    break if z.abs > 2*([b-a, d-c].min)
    x = z.real
    y = z.image
    xs = w/(b-a)*(x - a)
    ys = h/(c-d)*(y - d)
    TkDraw.oval(xs-2, ys-2, xs+2, ys+2,"navy")
    TkDraw.line(x_old, y_old, xs, ys, "blue")
    x_old, y_old = xs, ys
  end
  Tk.mainloop
```

Программа для рисунка 1.8 отличается от предыдущей лишь изменением мировых координат и комплексной константы:

```
a, b = -10, 10
c, d = -10, 10
cons = Complex.new(0.29, 0.01)
```

Несмотря на то, что две константы лишь незначительно отличаются своей действительной компонентой, поведение последовательностей кардинально различается. Оказывается, существует область значений параметра c, при которых точки, выходящие из начала координат имеют некоторую конечную орбиту, при других же значениях они устремляются в бесконечность. Эта область называется множеством Мандельбрата. Граница этой области устроена весьма причудливо. Если строить ее во более крупном масштабе, то мы будем получать уменьшенные изображения исходного множества, то есть оно является самоподобным (фракталом).

```
require 'complex'
require "TkDraw"
#require 'profile'

p = 400
a, b, s = -0.3, 0.0, 3.5
TkDraw.create(p, p, "Mandelbrot")
TkDraw.clean
for x in 1 .. p
  for y in 1 .. p
    c1 = a - s/2 + x*s/p
    c2 = b + s/2 - y*s/p
    c = Complex.new(c1, c2)
```

```
      z1 = Complex.new(0,0)
      for i in 1 ... 20
        z1 = z1*z1 + c
        r = z1.abs2
        break if r > 4
      end
      TkDraw.point(x, y,"#244a0d") if r < 4
    end
  end
  Tk.mainloop
```

Если раскрашивать точки в соответствии с их «скоростью убегания», то получаются очень красивые изображения.

```
require 'complex'
require "TkDraw"

p = 300
a, b, s = -0.3, 0.0, 3.5
TkDraw.create(p, p, "Mandelbrot")
TkDraw.clean
colors = %w( azure blueviolet coral gold
             maroon orchid springgreen chocolate
             lavender salmon saddlebrown)
for x in 1 .. p
  for y in 1 .. p
    c1 = a - s/2 + x*s/p
    c2 = b + s/2 - y*s/p
    c = Complex.new(c1, c2)
    z1 = Complex.new(0,0)
    r, i = 0, 1
    while i < 10 && r < 4
      i += 1
      z1 = z1*z1 + c
      r = z1.abs2
    end
    TkDraw.point(x, y, colors[i])
  end
end
Tk.mainloop
```

§ 5. Полярные координаты

Пример 5.1. Кривая, называемая спиралью Архимеда, задается в полярных координатах следующим уравнением: $r = \varphi$. Изобразим ее для значений $\varphi \in [0; 12\pi]$.

```
require "TkDraw"
include Math

h, w = 600, 600
a, b = -12*PI, 12*PI
c, d = -12*PI, 12*PI
n = 10000 # число точек для построения графика
TkDraw.create(h, w, "Example 2")
TkDraw.clean
0.step(b, b/n) { |fi|
  xm = fi*cos(fi)
  ym = fi*sin(fi)
  xs = w/(b-a)*(xm - a)
  ys = h/(c-d)*(ym - d)
  TkDraw.point(xs, ys,"darkgreen")
}
#TkDraw.label(500, 20, "Спираль Архимеда", 24)
TkDraw.line(300, 300, 600, 300, "red")
Tk.mainloop
```

Мы добавили к спирали отрезок прямой для иллюстрации одного из её характеристических свойств: расстояние между витками постоянно и равно $k \cdot (\varphi + 2\pi) - k \cdot \varphi = 2\pi k$.

Пример 5.2. Семество роз итальянского геометра Гвидо Гранди описывается уравнениями $r = a \sin k\varphi$, где a и k — некоторые постоянные.

Мы объединили все эти кривые в одной программе, которая нарисовав фигуру ждет 4 секунды, а затем рисует следующую.

```
require "TkDraw"
include Math
# Розы Гвидо Гранди: k = 2, 3, 5/3, 4/3, 1/2, 1/3
# r = a*sin(k*fi)

h, w = 600, 600
a, b = -2.2, 2.2
c, d = -2.2, 2.2
n = 10000 # число точек для рисования
```

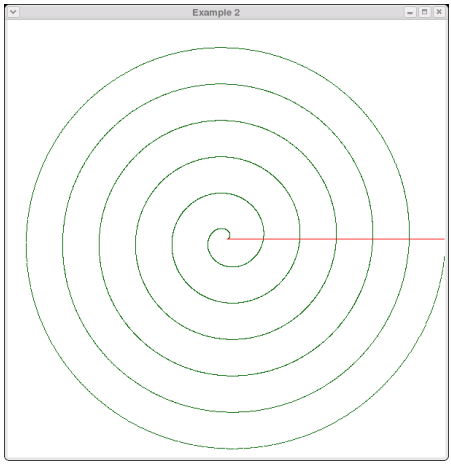

Рис. 1.9.

```
TkDraw.create(h, w, "Roses")

for ks in ['2', '3', '5/3', '4/3', '1/2', '1/3']
  k = eval(ks+".0")
  TkDraw.clean
  TkDraw.label(300, 20, "r = 2*sin(#{ks}*fi)", 24)
  0.step(8*PI, 8*PI/n) { |fi|
    r = 2*sin(k*fi)
    xm = r*cos(fi)
    ym = r*sin(fi)
    xs = w/(b-a)*(xm - a)
    ys = h/(c-d)*(ym - d)
    TkDraw.point(xs, ys,"darkgreen")
  }
  sleep 4
end
Tk.mainloop
```

Пример 5.3. В 19 веке немецкий геометр Хабенихт, полагая что очертания листа или цветка задаются в полярных координатах соотношением $r = f(\varphi)$, подобрал опытным путем интересные комбинации тригонометрических функций. Среди полученных им замечательных «растений»

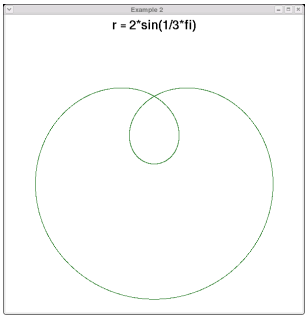

Рис. 1.10.

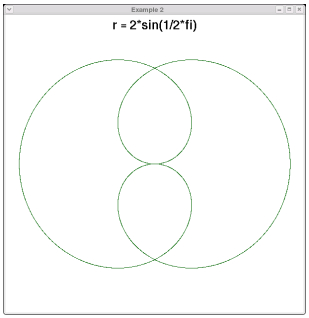

Рис. 1.11.

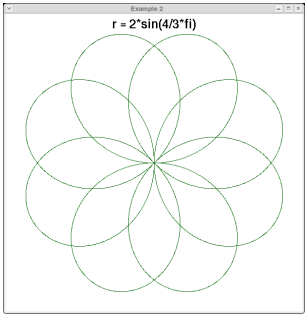

Рис. 1.12.

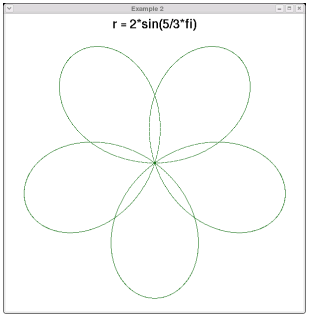

Рис. 1.13.

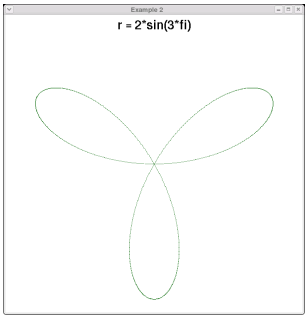

Рис. 1.14.

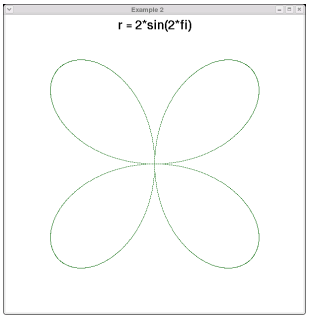

Рис. 1.15.

встретилось и, изображенное на рис. 1.16, задаваемое соотношением $r = 4(1 + \cos 3\varphi) + 4\sin^2 3\varphi$.

```
require "TkDraw"
```

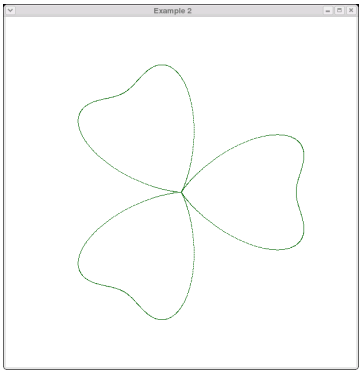

Рис. 1.16.

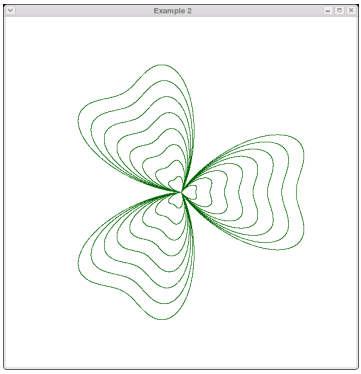

Рис. 1.17.

```
include Math

h, w = 600, 600
a, b = -12.2, 12.2
c, d = -12.2, 12.2
n = 10000 # число точек для построения графика
TkDraw.create(h, w, "r = 4(1+cos(3*fi))+4sin^2(3fi)")
TkDraw.clean
0.step(8*PI, 8*PI/n) { |fi|
  r = 4*(1+cos(3*fi))+4*sin(3*fi)**2
  xm = r*cos(fi)
  ym = r*sin(fi)
  xs = w/(b-a)*(xm - a)
  ys = h/(c-d)*(ym - d)
  TkDraw.point(xs, ys,"darkgreen")
}
Tk.mainloop
```

Упражнения

1. Какое множество точек задается соотношением $r = 1/\sin\varphi$?

2. Напишите программу для рисования других «растений Хабенихта», задаваемых соотношениями

1) $r = 4(1 + \cos 3\varphi) - 4\sin^2 3\varphi$

2) $r = 3(1 + \cos^2\varphi) - 4\sin^2 3\varphi$