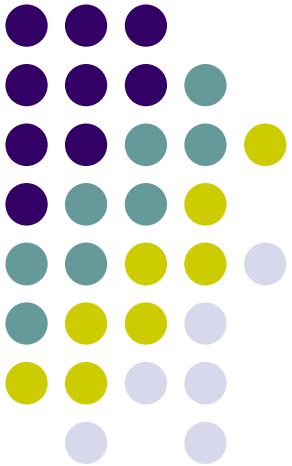
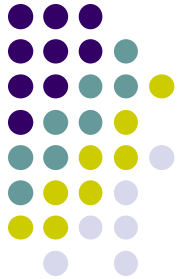
Объектная модель Ruby

Денис С. Мигинский



Основные характеристики объектной модели



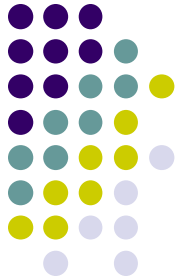
Обязательные:

- Поддержка классов
- Поддержка описания поведения класса
- Поддержка наследования
- Sub-typing полиморфизм

Опциональные:

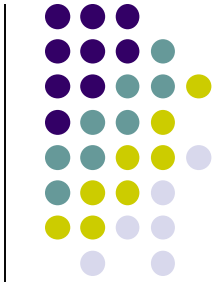
- Инкапсуляция
- Параметрический полиморфизм

Вопросы



- Что такое класс в общем случае?
- Как описывается поведение класса?
- Что такое sub-typing полиморфизм, как он связан с наследованием?

Класс

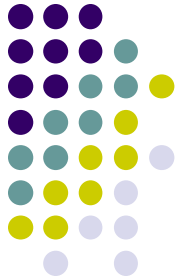


Концептуально класс – множество объектов, обладающих подобными свойствами (поведением в первую очередь)

В языке программирование класс:

1. Описание свойств экземпляров
2. Тип данных

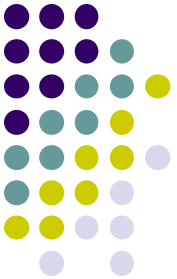
Описание поведения классов и полиморфизм



- Посылка сообщений и обработка их методами, входящими в описание класса. Полиморфизм реализуется на основе класса обработчика сообщения.

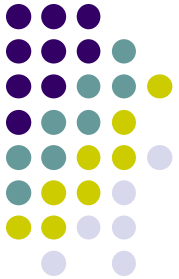
- Обобщенные функции. Полиморфизм реализуется на основе классов параметров.

Множественное наследование: проблемы



- Диспетчеризация вызовов методов суперклассов по нескольким путям
- Ромбовидные иерархии

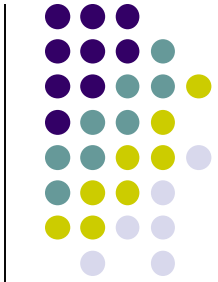
Объектная модель Ruby



В Ruby нет элементарных типов в обычном понимании, все является объектом.

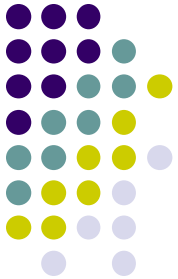
Для некоторых типов имеются специальные синтаксические конструкции, встроенные в язык: числа, строки, регулярные выражения, массивы, ассоциативные массивы, интервалы

Объектная форма операторов



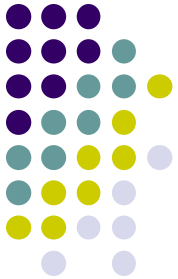
```
a = 1; b = 2; c = 3
d=a+b+c
d=5.+(b).+(c)

f=Proc.new{|name| puts “Hello, #{name}!”}
f.call(“Richard I”)
```

Особенности объектной модели Ruby



- Отсутствие интерфейсов
- Одиночное наследование
- Поддержка «примесей» (mixins)
- Перегрузка операторов
- Специфический контроль доступа

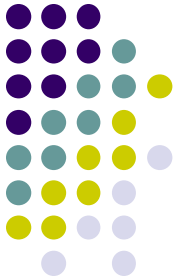
Определение класса



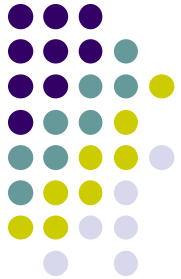
```
class Person
#конструктор
  def initialize (name, age)
    @name=name
    @age=age
  end
#определение метода
  def info
    return "#{@name}, age #{@age} years"
  end
end
```

Аксессоры

```
class Person
#getters
   def name
      @name
   end
   def age
      @age
   end
#setter (приватный в терминах Java/C++)
protected
   def age=(newAge)
      @age=newAge
   end
end
```

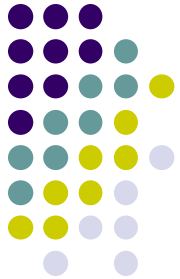


Методы и атрибуты класса



```
class Person
  @@count = 0                    #атрибут класса
  def initialize (name, age)
    @name=name
    @age=age
    @@count+=1
  end
  def Person.count               #метод класса
    @@count
  end
end
```

Наследование

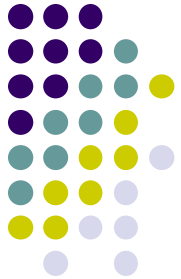


```
class Monarch < Person
  def initialize (name, dynasty, age)
    super(name, age) #вызов конструктора суперкласса
    @dynasty = dynasty
  end
#переопределение метода info
  def info
     return “House of #{@dynasty}, ”+super
  end
end
```

Модификаторы доступа

- доступ к методам контролируется динамически, относительно объектов
- public – метод доступен всем
- protected – метод доступен всем объектам данного класса
- private – метод доступен только данному объекту (в т.ч. если он является экземпляром подкласса)
- все методы по умолчанию публичные
- все атрибуты приватные

Динамическое изменение модификаторов доступа



#здесь метод age= защищенный

```
class Person
  public :age=;
end
```

#метод стал публичным

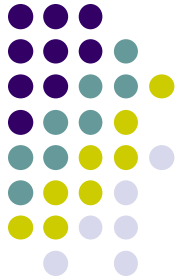
#без кода выше получим ошибку времени исполнения

```
king=Person.new (“Richard I”, 820)
king.age=830
```

Модули в качестве пространств имен

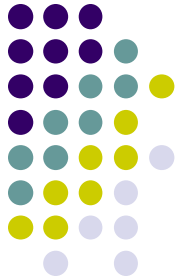
```
module History
   class Person
      …
   end
   class Monarch <Person
      …
   end
end

History.Person.new (“Adam”, “unknown”)
```

Модули в качестве примесей (mixin)



```
module SignificantEvents
  def addEvent (…)
    …
  end
end

class Person
  include SignificantEvents
  …
end
```

Концепция Meta-Object Protocol

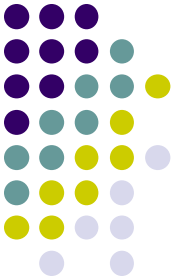


Любой объект является экземпляром некоторого класса, в том числе методы и сам класс.

Экземпляры пользовательских и системных классов неразличимы.

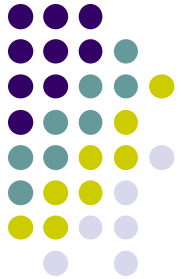
Объекта, позволяющие создавать собственные экземпляры называются метаобъектами. В том числе любой класс является метаобъектом.

Интроспекция



```
class A
  def initialize param
    @param=param
  end
  def method1
    puts @param
  end
end

puts A.superclass                          #Object
puts A.class                               #Class
puts A.included_modules                    #Kernel
puts A.methods                             #...
puts A.instance_methods                    #...
puts A.instance_method(:method1).arity     #0
```

Динамическое конструирование классов

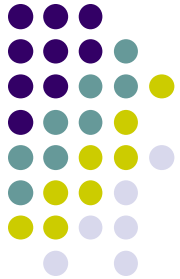


```
myClass = Class.new do
  def self.create_method (name, &block)
    define_method name, &block
  end
end

myClass.create_method(:initialize){|param|@param=param}

myClass.create_method(:method1){puts "param=#{@param}"}

a = myClass.new 5

puts a.method1
```

Обработка произвольных сообщений

```
module MissingMethodInterceptor
  def method_missing meth
    begin
      super meth
    rescue =>e
      puts e.message, e.backtrace
    end
    raise Exception.new("Critical error")
  end
end

class A
  include MissingMethodInterceptor
  def method1
    puts "method1"
  end
end

A.new.method1
A.new.method2
```

Проксирование вызовов: задача



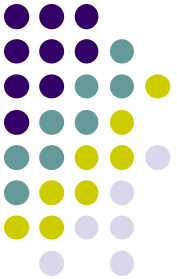
```
class Base
  def print_name
    puts "Base"
  end
end

class Derived < Base
  def print_name
    puts "Derived"
  end
end

Base.new.print_name
Derived.new.print_name
```

Постановка задачи:

Требуется перехватывать все вызовы методов и выводить информацию о них (например для отладки или профилирования).

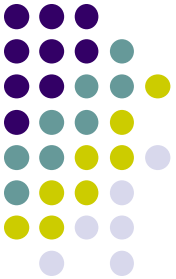
Решение не должно изменять существующий код.

Проксирование вызовов: решение



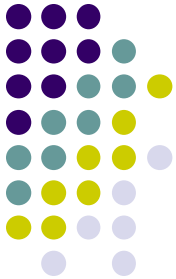
```
class Base
  def self.method_added(method_sym)
    return if @adding_method
    @adding_method = true
    orig_method = "orig_#{method_sym}"
    proxy_method = "proxy_#{method_sym}"
    define_method(proxy_method){|*args|
        puts "Called " + method_sym.to_s +
          " of class " + self.class.to_s
        send orig_method, *args
    }
    alias_method orig_method, method_sym
    alias_method method_sym, proxy_method
    @adding_method = false
  end
end
```

Взаимодействие с Java



```
include Java
#доступен Java API с полными именами
import org.xml.sax.helpers.DefaultHandler
#DefaultHandler доступен без квалификации пакета
#наследование от Java-класса
class MySAXHandler < DefaultHandler
#перегрузка метода Java-класса
  def startElement(uri, localName, qName, attrs)
#тело метода
  end
#определение других методов
end

#создание экземпляра класса (Ruby или Java)
MySAXHandler.new
DefaultHandler.new
java.lang.Thread.new {puts “Thread started”}
```

Задача 2



Реализуйте класс, аналогичный Array, с многопоточной реализацией итераторов: map, any?, all?, select. Объясните, можно ли таким образом реализовать итератор inject?

Ограничения:

- решение должны быть выражено через базовые итераторы и операции над коллекциями и потоками (threads)
- вся логика работы с потоками должны быть вынесена в отдельный метод, общий для всех итераторов.