Classes in Ruby

Motivation
- defining new types
- reusing existing code

Class definitions in Ruby

Examples

Every program we’ve seen this term uses predefined classes:
- String, Fixnum, Array, Hash, ...
- a big part of learning to program in Ruby is learning what classes are available
- you need to learn how to create objects that are instances of these classes and what sorts of methods can be applied to the objects

Today: an introduction to defining new classes
- motivation: why make a new class?
- organization: what goes in a class definition?
- implementation: Ruby syntax for creating a class and defining its methods
- many examples
- suggested projects

Why Define a New Class?

- One important reason for defining a new class is **better program organization**
  - we’ve already seen how writing a new method often makes it easier to write a program
  - it also makes the program easier to understand

- A small investment of time spent on defining a class is repaid several times when you write code that uses the class
  - example: define a new class called Car for the mpg.rb program
  - put code that creates the representation of a car (name, miles driven, fuel consumed) inside the class
  - create methods that define operations on cars (computing miles per gallon, comparing cars based on mileage, displaying a car) inside the class
  - The total amount of code is a bit larger since the class definition requires some overhead, but the main loop is much easier to write and understand

Why Define a New Class?

- Compare the old version of the main loop in mpg.rb with the new version:

```ruby
best_name = ""
best_mpg = 0.0
while line = gets
  name, miles, gals = line.split
  mpg = miles.to_f / gals.to_f
  if mpg > best_mpg
    best_mpg = mpg
    best_name = name
  end
end
```

```ruby
best = make_car(gets)
while line = gets
  x = make_car(line)
  if x.mpg > best.mpg
    best = x
  end
end
```

Today we’ll see how to define a new Car class:
- create an instance of Car object from the specs on an input line
- x.mpg returns the miles-per-gallon of x
Why Define a New Class?

- This program shows how defining a new class is an example of abstraction at work:
  - the new class (Car) defines a new type of object
  - encapsulate the representation of the type and the operations defined for the new type inside the class definition

```ruby
best = make_car(gets)
while line = gets
  x = make_car(line)
  if x.mpg > best.mpg
    best = x
  end
end
```

People who read this code can get the general idea of what the program is supposed to do: read descriptions of cars and compare cars.

Someone who needs to know the details (e.g., how mpg is computed) can look at the definition of make_car and the Car class.

Why Define a New Class?

- A second important reason: reuse
  - when you define a new class, you can build on existing code
  - Reusing existing code, rather than writing each new application from scratch, has several advantages
    - less code to write for the new program
    - existing code and design have been verified and tested
    - less code in your program means (in general) fewer chances for errors
  - Example: sequences in evolve.rb
    - a sequence is a type of string
    - define a new Sequence class so it extends the built-in String class
    - just write methods for operations unique to Sequence objects, e.g., mutate or transpose
    - you won’t have to write methods for insert, reverse, etc

Examples

- Here are some programs you wrote this term and examples of data items that could be used to define new classes
- Most of these are examples of abstraction: a grouping of code into a convenient package that implements a new type
- Others are examples of reuse, where new code builds on existing classes

- mpg.rb  class named Car to represent individual cars
- gpa.rb  class named Student (abstraction)
- invoice.rb  class named Book (abstraction)
- f2c.rb  classes named Celsius and Fahrenheit (reuse, build on Fixnum)
- diffs.rb  class named Sequence (reuse, build on String)
- evolve.rb
- overlap.rb

Properties of Objects

- When defining a new type of object (either through simple abstraction or by reusing an existing class) there are two main considerations

  1. You need to define the object’s attributes: what is it about the object that needs to be represented in the program?
    - for Car objects, we need to know the name, miles driven, and fuel consumed
    - we are not concerned with seating capacity, color, body style, etc
      A helpful guide: attributes define an object’s state
  2. You need to define the types of operations that will be performed on the objects
    - for Cars, we need to compute miles per gallon, print a string representing the car
    - we do not need to compute cost, depreciation, performance, etc
      In many cases operations update the state (e.g., a top-up operation might add to miles driven and gallons consumed) or compare two objects
Classes in Ruby

- It is very easy to create a class in Ruby
  - just introduce the class with the keyword `class` and its name
  - methods and attributes will be defined between the class name and the matching `end`

```ruby
class Car
  ...
end
```

- The class name becomes a new constant

```ruby
>> x = Car.new
... 
>> x.class
=> Car
>> Car.class
=> Class
```

Constructors

- Ruby automatically generates a method named `new` for each class

- This is the complete text of a trivial class named `Foo`:

```ruby
class Foo
end
```

- To see that Ruby made the method named `new`:

```ruby
>> x = Foo.new
=> #<Foo:0x3232dc>
>> x.class
=> Foo
>> Foo.class
=> Class
```

- Not very useful, but it verifies the claim that there is a method named `Foo.new`.

Methods

- Methods defined inside a class become members of the class
  - you can only call these methods via an instance of the class

```ruby
class Foo
  def bar
    puts "this is bar"
  end
end
```

- A test using `irb`:

```ruby
>> load "foo.rb"
>> bar
NameError: undefined local variable or method `bar'
>> x = Foo.new
=> #<Foo:0x62778>
>> x.bar
this is bar
```

Attributes

- Attributes are special variables that have names starting with `@`
  - in programming language terminology these are often known as `instance variables` or `state variables`

- Attributes are created by methods of the class

```ruby
class Foo
  def set_vars(x,y)
    @alpha = x
    @beta = y
  end
  def print_vars
    puts "a Foo with alpha = #{@alpha} and beta = #{@beta}"
  end
end
```

```ruby
>> x = Foo.new
=> #<Foo:0x62778>
>> x.set_vars 42, 3.14
>> x.print_vars
a Foo with alpha = 42 and beta = 3.14
```
Example irb session with this latest version of class Foo:

```
> load "foo.rb"
=> true
> x = Foo.new
=> #<Foo:0x85fe8>
```

```
def set_vars(x, y)
  @alpha = x
  @beta = y
end
```

```
> x.print_vars
a Foo with alpha = and beta =
=> nil
```

```
> x.set_vars(0, 99)
=> 99
> x.print_vars
a Foo with alpha = 0 and beta = 99
=> nil
```

```
> @alpha
=> nil
```

```
 ranch: the @alpha defined inside the class is not the same as @alpha outside the class!
```

The easiest way to define attributes for your objects is to use a special method named `attr_accessor`

+ the arguments to `attr_accessor` are `symbols`
+ place a call to this method at the start of the class
+ use one symbol for each attribute you want the objects to have

```
class Foo
  attr_accessor :alpha, :beta
end
```

`attr_accessor` creates methods that return or set the value of an attribute

```
> x = Foo.new
ArgumentError: wrong number of arguments (0 for 2)
```

**Attributes**

The Car Class

Putting all these ideas to work, here is the first versions of a class for car objects

```
class Car
  attr_accessor :name, :miles, :gallons
  def initialize(name, miles, gallons)
    @name = name
    @miles = miles
    @gallons = gallons
  end
  def mpg
    return @miles.to_f / @gallons.to_f
  end
end
```

**initialize method**

To set initial values or perform other operations when an object is created add a method named `initialize` to your class

+ `new` will call `initialize` after it creates the object

```
class Foo
  def initialize(alpha, beta)
    @alpha = alpha
    @beta = beta
  end
end
```

- Note: if you define an `initialize` method with `n` arguments, you must then pass `n` arguments to `new`

```
> x = Foo.new
ArgumentError: wrong number of arguments (0 for 2)
```
An irb session with the Car class

```ruby
>> load "mpg2.rb"
=> true
>> x = Car.new("prius", 500, 12)
=> #<Car:0x809d0 @gallons=12, @name="prius", @miles=500>
>> x.class
=> Car
>> x.mpg
=> 41.6666666666667
>> x.name = "toyota"
=> "toyota"
>> x.miles
=> 500
>> x
=> #<Car:0x809d0 @gallons=12, @name="toyota", @miles=500>
```

Notice what Ruby prints as the value of an object

Two other methods that are defined for every object (and thus automatically part of your new classes) are `to_s` and `inspect`.

To see what the predefined versions do:

```ruby
>> x.to_s
=> "#<Car:0x80cc8>
>> x.inspect
=> "#<Car:0x80cc8 @gallons=12, @name="prius", @miles=500>"
```

By convention `to_s` returns a String that summarizes the state of the object.

```
puts x.to_s
puts x.inspect
puts x
```

`to_s` returns a String that summarizes the state of the object.

```
puts x.to_s
puts x.inspect
puts x
```

You can write your own versions of these two methods.

```
class Car
  def to_s
    return "#{name}: #{mpg} mpg"
  end
end
```

Back in irb:

```ruby
>> x = Car.new("prius", 500, 12)
=> #<Car:0x31c3ec @miles=500, @name="prius", @gallons=12>
>> x.to_s
=> "prius: 41.6666666666667 mpg"
>> puts "Best car: #{x}"
Best car: prius: 41.6666666666667 mpg
```

Note variable interpolation also calls `to_s`
Summary

- `attr_accessor` does three things for each symbol `:x` in its argument list:
  - create an instance variable named `@x`
  - define a method named `x`, which by default returns the value of `@x`
  - define a method named `x=`, which assigns a value to `@x`

```ruby
class Foo
  def alpha
    @alpha.capitalize
  end

  attr_accessor :alpha, :beta

  def alpha=(val)
    @alpha = val.downcase
  end

  def beta=(val)
    raise "beta must not be nil" if val.nil?
  end
end
```

Class Methods

- In many cases we want to write a method that should be called on its own
  - it does not need to be (or can’t be) associated with any object
- Ruby (and other OOP languages) call these sorts of methods class methods
  - they are defined inside a class
  - they can be called from inside or outside the class
  - they typically perform operations related to the class
- Constructors are good examples of class methods
  - to make a new object, call the `new` method of the class
  - when we call `X.new`, we expect `X` to be the name of a class, not the name of an object
- `new` is class method -- it belongs to the class, and performs an operation related to the class

```ruby
class Foo
  def initialize(x,y)
    @alpha = x
    @beta = y
    $nfoo += 1
  end

  def count
    puts "There are #$nfoo Foos out there"
  end
end
```

Class Methods

- Let’s modify the `initialize` method in the `Foo` class so it keeps track of the number of `Foo` objects created
  - for now we’ll keep the count in a global variable (generally a bad idea)
  - to find out how many objects have been created, we’ll call a class method
    ```ruby
    $nfoo = 0
    class Foo
      def initialize(x,y)
        @alpha = x
        @beta = y
        $nfoo += 1
      end
    end
    
    def Foo.count
      puts "There are #$nfoo Foos out there"
    end
    ```

- Testing the new version of `Foo` in irb:
  ```ruby
  >> load "foo.rb"
  => true
  >> x = Foo.new("one","five")
  => #<Foo:0x3406fc @beta="five", @alpha="one">
  >> y = Foo.new("six","ten")
  => #<Foo:0x328f5c @beta="ten", @alpha="six">
  >> Foo.count
  There are 2 Foos out there
  ```
Class Variables

- A **class variable** is similar to a **class method**
  - it is defined inside the class
  - there is just one copy of the variable; it belongs to the class, not to any instance
  - it is only available to code inside the class; it is not accessible from outside the class

- Using a class variable to keep track of the number of objects created is a **much** better idea than using a global variable

- A class variable has two `@`s at the front of its name:
  ```ruby
class Foo
    @@nfoo = 0
    def Foo.count
      puts "There are #{@nfoo} Foos out there"
    end
  end
  ```

Self

- There may be times when you want to use the value of an object when you are writing code for a method
  - the word *self* means "this object", i.e. "the object to which this method is being applied"

- Example: extend the Foo class so we not only keep track of the number of Foos created but also a list of the objects
  ```ruby
class Foo
    @@a = Array.new
    def initialize(x,y)
      ...
      @@a << self
    end
    def Foo.list
      puts "Foo objects created so far: #{@a.inspect}"
    end
  end
  ```

Adding Code to an Existing Class

- Ruby makes it easy to add code (method definitions, class variables, etc) to existing classes

- When your program has
  ```ruby
  class Foo
    ...
  end
  ```
  the Ruby system checks to see if there is a class named *Foo*

- if not, it makes a new class

- If Foo exists already, the code between `class...end` is **added to the existing class**
  - this is true for classes you define (Foo, Car, ...)
  - it is also true for built-in classes (Fixnum, String, ...)

  *Obviously you want to use this "feature" sparingly*
  *Probably not a good idea to redefine the length method in String,...*

Adding Code to an Existing Class

- This code adds a new `permute` method to the String class
  ```ruby
  class String
    def permute!
      for i in 0..length-2
        r = rand(length-i) + i # i <= r < length
        self[i],self[r] = self[r],self[i]
      end
    end
  end
  ```

- Now we can permute any String object
  ```bash
  $ load "permute.rb"
  $ s = "supercalifragilisticexpialidocious"
  $ s.permute!
  ```
Design: Attribute or Method?

- The hardest part about making a new class is the original design
- The syntax (in Ruby, at least) is very easy and straightforward
- The hard part is figuring out
  - when will a program benefit from having a new class? (pretty clear for mpg.rb, not so clear for invoice.rb)
  - what are the attributes of objects of the new class?
  - what are the operations (methods) of the new class?
- Example: is miles per gallon an attribute of a car?
  - argument in favor: yes, it's what this program is all about; we should include :mpg in the items produced by attr_accessor
  - argument against: no, it's redundant; we already have miles and gallons, so mpg should be a method that computes mileage based on current values of miles and gallons

Refactoring

- Advice: go with the simplest design first
  - plan on modifying code and reorganizing as need change
  - the current version might be too slow
  - a new requirement for the application might force a new internal organization
- These sorts of changes, where the internal organization of a program is modified but the input and output remain the same, are examples of refactoring
- Refactoring is a big part of incremental design...
One way to reuse code is to define a new class that extends an existing class. In computer programming terminology, we say the new class inherits from the old one.

**Existing class:** base class, parent class, superclass

**New class:** derived class, child class, subclass

Example: Suppose we want to define a new Sequence class.

- Sequences are types of strings.
- In Ruby, give the name of the parent class after the name of the new class:

```
class Sequence < String
  attr_reader :sequence, :defline
  def initialize(defstring, seqstring)
    @defline = defstring
    super(seqstring)
  end
end
```

To define a Sequence object in Ruby:

```
s = Sequence.new
s = Sequence.new("test","ATGTAA")
```

Getting the length of a sequence, appending new characters to the end, reversing the letters, and more are all things we will want to do with Sequence objects.

By defining Sequence as a subclass of String we get these operations for free.

```
5
```

To solve this problem, Ruby has a method named `super`.

It means "pass this item to the method of the same name in my superclass".

```
class Sequence < String
  attr_reader :sequence, :defline
  def initialize(defstring, seqstring)
    @defline = defstring
    super(seqstring)
  end
end
```

```
5
```

```
6
```

It works!
Now we can add items that are specific to sequences

```ruby
class Sequence < String
  Alphabet = ['A', 'T', 'C', 'G']  # Alphabet is a constant defined in the class

  def mutate
    self[rand(self.length)] = Alphabet[rand(Alphabet.length)]
  end

  def to_fa
    val = ''
    val << '>' unless @defline[0] == ?>
    val << @defline << '
'  # sequence
    return val
  end
end
```

```ruby
class Sequence < String
  Alphabet = ['A', 'T', 'C', 'G']

  def mutate
    self[rand(self.length)] = Alphabet[rand(Alphabet.length)]
  end

  def to_fa
    val = ''
    val << '>' unless @defline[0] == ?>
    val << @defline << '
'  # sequence
    return val
  end
end
```

Sequence.new("test", "ATGTAA")
=> "ATGTAA"
```
```