rubylearning.com

Helping Ruby Programmers become Awesome!

Home | Tutorial | Downloads | Testimonials | Certification | Mentor | Blog | Online Course | Challenge | About

Writing our own Class in Ruby

<Regular Expressions | TOC | Method Missing>

So far, the procedural style of programming (this continues to be used in languages such as C) was used to write our programs. Programming procedurally means you focus on the steps required to complete a task without paying particular attention to how the data is managed.

In the Object-Orientation style, objects are your agents, your proxies, in the universe of your program. You ask them for information. You assign them tasks to accomplish. You tell them to perform calculations and report back to you. You hand them to each other and get them to work together.

When you design a class, think about the objects that will be created from that class type. Think about the things the object **knows** and the things the object **does**.

Things an object **knows** about itself are called *instance variables*. They represent an object's state (the data - for example, the quantity and the product id), and can have unique values for each object of that type.

Things an object can **do** are called *methods*.

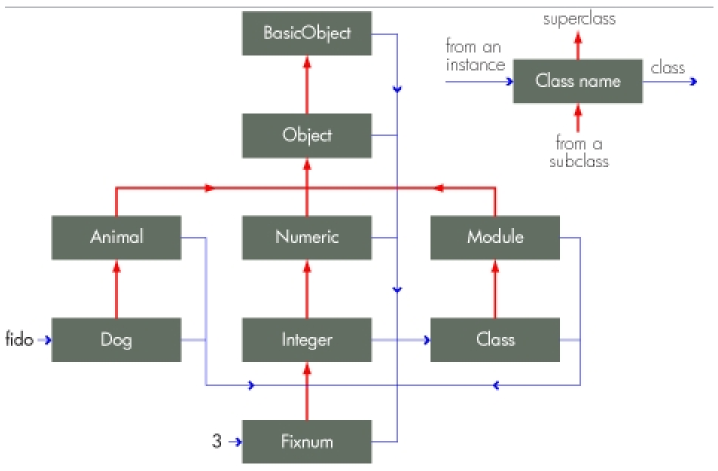
> An object is an entity that serves as a container for data and also controls access to the data. Associated with an object is a set of attributes, which are essentially no more than variables belonging to that object. Also associated with an object is a set of functions that provide an interface to the functionality of the object, called methods. - Hal Fulton

An object is a combination of state and methods that use the state.

Hence a class is used to construct an object. A class is a blueprint for an object. For example, you might use a Button class to make dozens of different buttons, and each button might have its own color, size, shape, label, and so on. An object is an instance of a class.

Read this very carefully, it's a brain bender!
Classes in Ruby are first-class objects - each is an instance of class **Class**. When a new class is defined (typically using **class** Name ... **end**), an object of type **Class** is created and assigned to a constant (Name. in this case). When Name.**new** is called to create a new object, the **new** class method in **Class** is run by default, which in turn invokes **allocate** to allocate memory for the object, before finally calling the new object's **initialize** method. The constructing and initializing phases of an object are separate and both can be over-ridden. The construction is done via the **new** class method, the initialization is done via the **initialize** instance method. **initialize** is not a constructor!

The following class hierarchy (courtesy: Nick Morgan) is informative:



Let's write our first, simple class - **p029dog.rb**

```ruby
# p029dog.rb
# define class Dog
class Dog
  def initialize(breed, name)
    # Instance variables
    @breed = breed
    @name = name
  end

  def bark
    puts 'Ruff! Ruff!'
  end

  def display
    puts "I am of #{@breed} breed and my name is #{@name}"
  end
end

# make an object
# Objects are created on the heap
d = Dog.new('Labrador', 'Benzy')

=begin
  Every object is "born" with certain innate abilities.
  To see a list of innate methods, you can call the methods
  method (and throw in a sort operation, to make it
  easier to browse visually). Remove the comment and execute.
=end
# puts d.methods.sort

# Amongst these many methods, the methods object_id and respond_to? are important.
# Every object in Ruby has a unique id number associated with it
puts "The id of d is #{d.object_id}."

# To know whether the object knows how to handle the message you want
# to send it, by using the respond_to? method.
if d.respond_to?("talk")
  d.talk
else
  puts "Sorry, d doesn't understand the 'talk' message."
end

d.bark
d.display

# making d and d1 point to the same object
d1 = d
d1.display

# making d a nil reference, meaning it does not refer to anything
d = nil
d.display

# If I now say
d1 = nil
# then the Dog object is abandoned and eligible for Garbage Collection (GC)
```

The output is:

```
>ruby p029dog.rb
The id of d is 22982920.
Sorry, d doesn't understand the 'talk' message.
Ruff! Ruff!
I am of Labrador breed and my name is Benzy
I am of Labrador breed and my name is Benzy
>Exit code: 0
```

The method **new** is used to create an object of class Dog. Objects are created on the heap. The variable d is known as a reference variable. It does not hold the object itself, but it holds something like a pointer or an address of the object. You use the dot operator (.) on a reference variable to say, "use the thing before the dot to get me the thing after the dot." For example:
d.bark

IN RAILS: If you're writing a Rails application in which one of your entity models is, say, Customer, then when you write the code that causes things to happen – a customer logging into a site, updating a customer's phone number, adding an item to a customer's shopping cart – in all likelihood you'll be sending messages to customer objects.

Even a newly created object isn't a blank slate. As soon as an object comes into existence, it already responds to a number of messages. Every object is "born" with certain innate abilities. To see a list of innate methods, you can call the **methods** method (and throw in a **sort** operation, to make it easier to browse visually):
puts d.methods.sort
The result is a list of all the messages (methods) this newly minted object comes bundled with. Amongst these many methods, the methods **object_id** and **respond_to?** are important.

Every object in Ruby has a unique id number associated with it. You can see an object's id by asking the object to show you its **object_id**:

puts "The id of d is #{d.object_id}."

You can determine in advance (before you ask the object to do something) whether the object knows how to handle the message you want to send it, by using the **respond_to?** method. This method exists for all objects; you can ask any object whether it responds to any message. **respond_to?** usually appears in connection with conditional (if) logic.

```
if d.respond_to?("talk")
  d.talk
else
  puts "Sorry, the object d doesn't understand the 'talk' message."
end
```

Now, the statements:
d1 = d
d1.display
makes d and d1 point to the same object.

You can ask any object of which class it's a member by using its **Object.class** method. In the above program, if we write the statement:

```
d = Dog.new('Alsatian', 'Lassie')
puts d.class.to_s
```

The output is:

```
>ruby p029dog.rb
Dog
>Exit code: 0
```

instance_of? returns true if object is an instance of the given class, as in this example:

```
num = 10
puts(num.instance_of? Fixnum) # output true
```

Often, it is better to ask **respond_to?** rather than **instance_of?** as we usually care about 'ability' rather than 'type'.

Literal Constructors

That means you can use special notation, instead of a call to **new**, to create a new object of that class. The classes with literal constructors are shown in the table below. When you use one of these literal constructors, you bring a new object into existence

Examples:

String - 'hello' or "hello"
Symbol - :symbol or :"hello world"
Array - [x, y, z]
Hash - {"India" => "IN"}
Range - 3..7 or 3...7

Observe that there is no visible call to **new**.

Garbage Collection

The statement:
d =nil
makes d a **nil** reference, meaning it does not refer to anything. If I now say:
d1 = nil
then the Dog object is abandoned and eligible for **Garbage Collection** (GC). The Ruby object heap allocates a minimum of *8 megabytes*. Ruby's GC is called *mark-and-sweep*. The "mark" stage checks objects to see if they are still in use. If an object is in a variable that can still be used in the current scope, the object (and any object inside that object) is marked for keeping. If the variable is long gone, off in another method, the object isn't marked. The "sweep" stage then frees objects which haven't been marked.

Ruby uses a conservative mark-and-sweep GC mechanism. There is no guarantee that an object will undergo garbage collection before the program terminates.

Class Methods

The idea of a class method is that you send a message to the object that is the class rather than to one of the class's instances. Class methods serve a purpose. Some operations pertaining to a class can't be performed by individual instances of that class. **new** is an excellent example. We call Dog.new because, until we've created an individual dog instance, we can't send it any messages! Besides, the job of spawning a new object logically belongs to the class. It doesn't make sense for instances of Dog to spawn each other. It does make sense, however, for the instance-creation process to be centralized as an activity of the class Dog. It's vital to understand that by Dog.new, we have a method that we can access through the class object Dog but not through its instances. Individual dog objects (instances of the class Dog) do not have this method. A class object (like Dog) has its own methods, its own state, its own identity. It doesn't share these things with instances of itself.

Here's an example:

Dog#bark - the instance method **bark** in the class **Dog**
Dog.color - the class method **color**, in the class **Dog**
Dog::**color** - another way to refer to the class method **color**

In writing about Ruby, the *pound notation* (#) is sometimes used to indicate an instance method - for example, we say **File.chmod** to denote the class method **chmod** of class **File**, and **File#chmod** to denote the instance method that has the same name. **This notation is not part of Ruby syntax.**

You will learn how to write class methods, later on here.

<Regular Expressions | TOC | Method Missing>