Reflection and Metaprogramming

We’ve already seen a few examples of where Ruby programs can discover things about themselves at run time.

For example, we have seen calls like

3.14159.methods

Why do we call this “discovering things about [3.14159] at run time”?

Reflection allows program entities to discover things about themselves through introspection.

For example, an object can ask what its methods are, and a class can tell what its ancestors are.

While Java also provides reflection, it does so much more verbosely than Ruby. One way that reflection can be used is in aspect-oriented programming (AOP).

The related technique of metaprogramming allows one to create new program entities, such as methods or classes, at run time. Why is this called metaprogramming?

puts 1.class # print the class of 1
>>
puts 123456789012345.class
>>
puts 123456789012345.kind_of? Integer
>>
puts 123456789012345.instance_of? Integer
>>
puts 123456789012345.instance_of? BigNum
>>
puts [1, 2, 3, 4, 5].length
>>
puts "Hey".class
>>
puts "John".class.superclass # print the superclass of a String
>>
puts String.ancestors # print the hierarchy
>>
puts Object.methods # print all the methods
>> [inspect, send, display, class_eval, clone..] # Output has been truncated

Note: While it may be useful, in debugging, to print out the class of an object, it is almost always a mistake to test the class of an object:
if s.kind_of? Integer then this_method else that_method end

Why?
(As an aside, why should there be a difference between a Fixnum and a Bignum?)

In a language like C, C++, or Ada, an assignment like a = b

- is interpreted as “copy b into a,” and
- is implemented by copying the contents of b into the space occupied by a.

This implies that a and b must be the same type, and, more importantly, the same size.

But Ruby employs dynamic binding, not static binding. This means that the type of object “stored” in a variable is determined at run time, not at compile time.
Therefore, \( a = b \)

- is interpreted as “bind \( a \) to the same object that \( b \) is bound to,” and
- is implemented by copying the reference stored in \( b \) into the (pointer-sized) memory cell \( a \).

Before \( a = b \)

\[
\begin{align*}
    a & \quad \text{Object 1} \\
    b & \quad \text{Object 2}
\end{align*}
\]

Assignments, then, physically copy references, but \textit{do not copy the objects}.

It is also possible to get a \textit{reference} to a method.

This can be done using the method \texttt{methods} in the class \texttt{Object}.

It is available to all classes, since \texttt{Object} is the superclass of all classes.

The reference so saved can be used to invoke that particular method.

\texttt{str = "Hello, World"}

The name of the method has to be passed as a symbol to \texttt{method}:

\[
\begin{align*}
    m &= \texttt{str.method(:upcase)} \quad \text{# returns a Method object} \\
    \texttt{puts m.call}
\end{align*}
\]

Why is the name of the method passed as a symbol? (What is a symbol anyway?)

- Symbols are immutable.
- Symbols are unique. Every instance of a particular symbol is the \textit{same} symbol.

\[
\begin{align*}
    \texttt{puts :mysymbol.object_id} \\
    \texttt{puts :mysymbol.object_id} \\
    \texttt{puts "mystring".object_id} \\
    \texttt{puts "mystring".object_id}
\end{align*}
\]
One very useful application of passing methods as parameters comes in quadrature (numerical integration). Suppose we want to compute the area under a curve described by some function, e.g., \( x^3 + 2x^2 + 3x + 4 \).

Then we can define

```ruby
class Float
  def poly
    self*self*self + 2*self*self + 3*self + 4
  end
end
```

And we can pass `poly` to an integration routine:

```ruby
area = integrate(:poly, 0, 10)
```

**Intercepting calls to undefined methods**
Whenever a call to an undefined method is made on an object, Ruby provides an option to intercept the call.

This is done by implementing the method `method_missing` within the class definition. Ruby passes as parameters the name of the method called and the arguments passed to it.

```ruby
class Cat
  def mew
    puts "Meow"
  end
  def method_missing(meth, *args)
    puts "Sorry, I do not #{meth}" end
end
```

c = Cat.new
c.mew
>> Meeow
c.bark
>> Sorry, I do not bark
An interesting use of `method_missing` can be found on p. 579 of
http://redhanded.hobix.com/inspect/theBestOfMethod_missing.html

Define a module (or class) `Roman`. This class contains a `method_missing`
method that intercepts calls to “class methods” that are undefined. It then
tries to interpret the method name as a Roman numeral.

For example,

- evaluating `Roman.xix` calls the ____ method of module `Roman`.
- `Roman` has no `xix` method, so ____________ is invoked with `:xix` as
  the argument.
- The `id2name` method of class `Symbol` is invoked on ____, returning
  "xix".
- The "xix" is then parsed according to the rules for evaluating Roman
  numerals, and evaluates to 19.

This functionality can be used to implement proxies.

**Metaprogramming**

Metaprogramming means writing code that writes code. This allows us to
modify the behavior of a program at run time.

Ruby has several metaprogramming facilities. One can add create a new
class and add methods to it using `define_method`.

c = Class.new
c.class_eval do
  define_method :hi do
    puts "Hey"
  end
end

```
c.new.hi
>>Hey
```
One can evaluate any valid string as code at run time using `eval`.

```ruby
class MyClass
  eval %{def hi
    puts "Hello"
  end}
end
m = MyClass.new
m.hi
```

This simple and powerful technique allows one to add any type of new code and modify behavior of a program at run time. (Explain how.)

**Method aliasing**
Method aliasing allows one to give new names to existing methods.

Suppose you have a `MyArray` class which has an attribute `size`. But people are also used to `Array` classes having an attribute `length`.

So, we provide an alias for it using the `alias` method.

Method aliasing is used to implement aspect-oriented programming in Ruby. This technique, which is not available in languages such as Java, will be useful in creating an AOP library.

```ruby
class MyArray
  def initialize(size)
    @size = size
  end
  attr_accessor :size
  alias :length :size  # alias for getter
  alias :length= :size=  # alias for setter
end
a = MyArray.new(5)
p = a.length
>>5
```
Aspect-Oriented Programming
A “cross-cutting concern” is an action that needs to be performed from many different classes.

One such concern is logging: We want many actions in a program to be logged.

Aspect-oriented-programming (AOP) helps in separating cross-cutting concerns by providing ways to intercept method calls.

In Ruby, AOP can be performed using aliasing, reflection and metaprogramming. These language facilities make it much easier to perform AOP than in Java.

There is an AspectR library which further facilitates performing AOP in Ruby. Here is how one would create a Logger using AspectR:

(Internally, this is implemented by

- aliasing the method code to a method of a different name, e.g., old_methodname, and then
- creating a new method with the original name that
  - just invokes the “before” code of the aspect,
  - then invokes the original method,
  - then invokes the “after” code of the aspect).

This is a often called a wrapper. In Java, AspectJ has to go in & modify the byte code.)

```ruby
require 'aspectr'
include AspectR
class Logger < Aspect  # Aspect is a class defined in AspectR
def before(method, object, exitstatus, *args)  # logs method calls
  puts "#{self.class}##{method}: args = #{args.inspect}"
end

def after(method, object, exitstatus, *args)  # logs returns from methods
  print "#{self.class}##{method}: exited "
end
end
```

```ruby
```
class SomeClass
  def some_method(a)
    puts "Hello!"
  end
end
Logger.new.wrap(SomeClass, :before, :after, /some/)
SomeClass.new.some_method 1

>>> Logger#some_method: args = [1]
>>> Hello!
>>> Logger#some_method: exited

In the above, code we wrap the method some_method of SomeClass with Logger's before and after methods.

The wrapper method aliases the old implementation of some_method to a new name and redirects calls to the new implementation, which invokes before and after in the new code.

It demonstrates how concisely AOP can be realized in Ruby.