# CSE 341 Lecture Notes - Reflection and Related Concepts

*Reflection* refers to facilities in a language that allow the programmer to examine aspects of the program from within the program itself. This is discussed in more detail in Chapter 26 of the *Programming Ruby* book.

## All Instances of a Given Class

Find all accessible objects in the namespace that are instances of the class Float or a subclass of Float:

```
ObjectSpace.each_object(Float) {|x| p x}
```

(`each_object` takes a class as an argument, and a block, and invokes the block for each instance in memory of the class or a subclass of it. `p` prints the object followed by a newline -- convenient for experimenting in irb.)

## Object Attributes

For any object x:

- `x.instance_variables`
- `x.methods` -- return a list of methods for x
- `x.respond_to?(:to_s)` -- test whether x understands the `to_s` message
- `x.class` -- return x's class
- `x.kind_of?(Numeric)` -- test whether is x an instance of Numeric or a subclass of Numeric
- `x.instance_of?(String)` -- test whether x is an instance of String

Note that using `respond_to?` is more duckly than using `kind_of?`.

## Class Attributes

Classes are also objects! Try these:

```
3.class
Fixnum.superclass
Object.superclass
Fixnum.ancestors  (note that this includes mixins)
```

## Classes are Objects

Examples to ponder:

```
3.class
3.class.class
3.class.class.class
3.class.class.class.class
```

What's going on here? Since classes are objects, they must be instances of some class. In Ruby, this is the class `Class`. And `Class` is an instance of itself!

## Some Useful Methods for Class

We can also ask a class more specific questions about its methods. First let's define an example class:

```
class Octopus
  @@octo_var = 2      # class variable
  TENTACLES = 8   # constant
  def initialize(n)
    @name = n
  end
  def speak
    puts "I'm an octopus named #{@name}"
  end
  def Octopus.classgreeting
    puts "hi from class Octopus"
  end
  private
    def private_method
    end
end
```

Now try these:

```
Octopus.private_instance_methods(false)
Octopus.public_instance_methods(false)
Octopus.public_instance_methods(true)  # include ancestors
Octopus.constants
Octopus.class_variables
Octopus.singleton_methods
```

Also see the Ruby docs for Class.

We can make new classes (i.e. instance of Class) by sending Class the new message (with an optional argument that is the superclass). This creates an anonymous class -- we haven't necessarily bound it to a name.

## Calling Methods Dynamically

```
o.send(:speak)

a = "s" + "peak"
b = a.to_sym
o.send(b)
```

How to grab a method:

```
s = o.method(:speak)
s.call

# m1 is an unbound method
m1 = Octopus.instance_method(:speak)
m2 = m1.bind(o)
m2.call
```

## Eval

Ruby has an eval method:

```
eval("3+4")
x =42
eval("x")
```

You can also pass an environment to eval:

```
def test
  v = 42
  return binding
end

b = test
# b is now a binding that includes the variable v
eval("v", b)
```

## Singleton Classes

An object can have its own methods or instance variables. This is implemented by creating a *singleton class* for it.

```
s = "I am a string ...."
class <<s
  attr_accessor :squid
  def greet
    return "hi there"
  end
end
```

Now s has a greet method and a squid attribute! (But just s, not all strings.) Try this:

```
s.singleton_methods
```

Singleton classes are also used to handle class variables and class methods. For example, the class `Octopus` above has a class method named `classgreeting`. This is in the singleton class for `Octopus`. See Chapter 24 of the Ruby book for more details.