Access Control

O-o languages have different levels of access control, e.g., `private` and `public`.

Ruby provides three levels of access control –

- Public methods can be called by anyone—no access control is enforced. Methods are public by default (except for _____, which is always private).
- Protected methods can be invoked only by objects of the defining class and its subclasses. Access is kept within the family.
- Private methods cannot be called with an explicit receiver—the receiver is always `self`. This means that private methods can be called only in the context of the current object; you can’t invoke another object’s private methods.

If a method is private, it may be called only within the context of the calling object—it is never possible to access another object’s private methods directly, even if the object is of the same class as the caller.

By contrast, if a method is protected, it may be called by any instance of the defining class or its subclasses.

Can you think of a case where a private method would need to be called by another object of the same class?

class MyClass
    def method1 # default is “public”
        #...
    end
    protected # subsequent methods will be “protected”
    def method2 # will be “protected”
        #...
    end
    private # subsequent methods will be “private”
def method3 # will be “private”
  #...
end
public # subsequent methods will be “public”
def method4 # and this will be “public”
  #...
end

Alternatively, you can set access levels of named methods by listing them as arguments to the access-control functions.

```ruby
class MyClass
  def method1
  end
  # ... and so on
  public :method1, :method4
  protected :method2
  private :method3
end
```

### Unbounded vs. subtype polymorphism

In a statically typed o-o language like Java or C++, you can declare a variable in a superclass, then assign a subclass object to that type:

```java
public class Bicycle {
  protected int gear;
  public void setGear(int nextGear) {
    gear = nextGear;
  }
}
public class MountainBike extends Bicycle {
  protected int seatHeight;
  public void setSeatHeight(int newSeatHeight) {
    seatHeight = newSeatHeight;
  }
}
```
Which statement is illegal in the code above? Why?

In most dynamically typed oo languages, including Ruby, that statement would be legal. In Ruby, if a method is defined on an object, the object can respond to the message.

It doesn’t matter what class the object is declared as … in fact, the object isn’t declared!

This is called unbounded polymorphism—the polymorphism is not limited by the declared class of the object.

In contrast, statically typed oo languages usually have subtype polymorphism—the compiler checks that the invoked method is defined in the type that the object is declared as.

Unbounded polymorphism is related to duck typing, which is discussed in next week’s online lectures.

Abstract Methods
Ruby does not have abstract methods like Java or pure virtual functions like C++.

However the same functionality can be simulated as follows (similarly to subclassResponsibility in Smalltalk):

class Shape
```ruby
def draw
  raise NotImplementedError.new("Method not implemented")
end
end
```

What will happen if we execute …

```ruby
s = Shape.new.draw
```

Subclasses, then, have to provide an implementation for the `draw` method.

Why do you think that Ruby and other dynamic o-o languages don’t have an official “abstract method” construct?

---

**Modules and Mixins**

Modules in Ruby are a way to group together methods, classes and constants. They are similar to namespaces in languages such as C++.

Of course, whenever you mix namespaces, you have the possibility of name clashes.

Say you have a graphics library that contains classes for Window and (`window.rb`) and Border (`border.rb`).

Both `window.rb` and `border.rb` have a `top` method, which gives the position of the top of the Window and the top of the Border, respectively.

An application programmer wants to lay out windows with borders, and thus loads both `window.rb` and `border.rb` into the program.

---

1 A considerably cornier example can be found in the Module chapter of *Programming Ruby*. 
What's the problem here?

Ruby's answer is the module mechanism.

Modules define a namespace. What's a namespace?

The window functions can go into one module

```ruby
module Window
  def Window.top
    # ..
  end
  def Window.bottom
    # ..
  end
end
```

and the border functions can go into another

```ruby
module Border
  def Border.top
    # ...
  end
  def Border.width
    # ..
  end
end
```

Look at the names of module methods. Given their syntax, what do they remind you of?

When a program needs to use these modules, it can simply load the two files using the Ruby `require` statement, and reference the qualified names.

```ruby
require 'window'
require 'border'
```
trueTop = Window.top + Border.top

Ruby’s require, include, and load statements have similar functionality.

- include makes features available, but does not execute the code.
- require loads and executes the code one time (somewhat like a C #include).
- load loads and executes the code every time it is encountered.

To understand their behavior, it’s important to realize that Ruby is a dynamic language—it supports not only dynamic typing, but metaprogramming: code can be written at run time (as we will see in Lecture 10).

Any idea how a C #include differs from a Ruby require?

Any idea how a C #include differs from a Ruby require?

Now, why do you think you might want to load code instead of require it?

Mixins

The most interesting use of modules is to define mixins.

When you include a module within a class, all its functionality becomes available to the class.

The methods of the module become instance methods in the class that includes it.
Let’s contrast this with `#include` in (what languages?) How are mixins different?

Let’s contrast this with multiple inheritance in (what languages?). How are mixins different?

So, instead of implementing interfaces as in Java, one uses mixins in Ruby when one wants to __________.

Consider the following code:

```ruby
module Introspect
  def kind
    puts "This object is a #{self.class.name}"
  end
end

class Animal
  include Introspect
  def initialize(name)
    @name = name
  end
end

class Car
  include Introspect
  def initialize(model)
    @model = model
  end
end

d = Animal.new("Cat")
c = Car.new("Ferrari")
d.kind # kind method is available through …
c.kind # .. the mixin Introspect

>>This object is a Animal
>>This object is a Car

*Exercise:* Take a look at the TeachingAssistant example defined [here](#) and answer [here](#).

- What happens if you remove the different to_s methods?
- What does this tell you about the way that Ruby resolves conflicts among names inherited from mixins?
- Can you affect this by using qualified names, i.e., by prefixing the method name by the class name, e.g., `ed.Employee::to_s`?

**Comparable**

A good example of the power of modules is Comparable, from the Ruby library.

To use comparable in a class, the class needs to define a method called `<=>` (sometimes called “rocket”).

Once we define this method, we get a lot of useful comparison functions, such as `<`, `>`, `<=`, `>=`, `==` and the method `between?` for free.

What is this like in Java? In Java, similar functionality is achieved through the use of the Comparable interface.

Here is an example. Suppose that we have a Line class:

```ruby
class Line
  def initialize(x1, y1, x2, y2)
    @x1, @y1, @x2, @y2 = x1, y1, x2, y2
  end
end
```
We compare two lines on the basis of their lengths.

We add the `Comparable` mixin as follows:

class Line
  include Comparable
  def length_squared
    (@x2-@x1) * (@x2-@x1) + (@y2-@y1) * (@y2-@y1)
  end
  def <=>(other)
    self.length_squared <=> other.length_squared
  end
end

` <=> ` returns 1, 0, or -1, depending on whether the receiver is greater than, equal to, or less than the argument.

We delegate the call to ` <=> ` of the `Fixnum` class, which compares the squares of the lengths.

Now we can use the `Comparable` methods on `Line` objects:

```ruby
l1 = Line.new(1, 0, 4, 3)
l2 = Line.new(0, 0, 10, 10)
puts l1.length_squared
if l1 < l2
  puts "Line 1 is shorter than Line 2"
else if l1 > l2
  puts "Line 1 is longer than Line 2"
else
  puts "Line 1 is just as long as Line 2"
end
```

```
>> Line 1 is shorter than Line 2
```
Composing Modules

Enumerable [SAAS §3.7] is a standard mixin, which can be included in any class.

It has a very useful method **inject**, which can be used to repeatedly apply an operation to adjacent elements in a set:

```
[1, 2, 3, 4, 5].inject {|v, n| v+n }
```

>> 15

Many built-in classes include **Enumerable**, including **Array** and **Range**.

```
('a' .. 'z').inject {|v, n| v+n }
```

*Exercise*: Use **inject** to define a **factorial** method.

Let's define a **VowelFinder** class that includes **Enumerable**. It will have an **each** method for returning successive vowels from a string. This method **yields** each time it encounters a vowel.

```
class VowelFinder
  include Enumerable

  def initialize(string)
    @string = string
  end

  def each
    @string.scan(/[aeiou]/) do |vowel|
      yield vowel
    end
  end
end
```
Here's an example of its use:

```ruby
VowelFinder.new("abacadabra").inject {|v, n| v+n}
```

```ruby
>>
```

However, would be nice not to have to type out the “inject” each time, if we just want to work on a different collection. “+” does different things for different classes.

- When used with numbers, it ____________.
- When used with strings, it ____________.

Let's define a module that encapsulates the call to `inject` and `+`.

```ruby
module Reducible
  def sum_reduce
    inject {|v, n| v+n}
  end
end

class Array
  include Reducible
end

class Range
  include Reducible
end

class VowelFinder
  include Reducible
end

[1, 2, 3, 4, 5].sum_reduce
('a' .. 'z').sum_reduce

vf = VowelFinder.new
("The quick brown fox jumped over the lazy dog.").sum_reduce
```

```ruby
puts vf.sum_reduce
```
Exercise: Define a method that returns a string containing the vowels, so that the above can be done with only one method call.

Simulating Multiple Inheritance

Modules can be used to simulate multiple inheritance in Ruby. Suppose one wants to add tags to Strings. Then one can define a Taggable module and include it into the class.

```ruby
require 'set' # A collection of unordered values with no duplicates

# Include this module to make your class taggable. The names of the instance variable and the setup method are prefixed with "taggable_" to reduce the risk of namespace collision. You must call taggable_setup before you can use any of this module's methods.
module Taggable
  attr_accessor :tags

  def taggable_setup
    @tags = Set.new
  end

  def add_tag(tag)
    @tags << tag
  end

  def remove_tag(tag)
    @tags.delete(tag)
  end
end

class TaggableString < String
```
include Taggable
  def initialize(arg)
    super
    taggable_setup
  end
end

s = TaggableString.new('It was the best of times, it was the worst of times."

Exercise: Add a to_s method to the taggable-string example. Then submit it here.

Reflection

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.

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
   # Fixnum
puts 123456789012345.class
   # Bignum
puts 123456789012345.kind_of? Integer
   # true
puts 123456789012345.instance_of? Integer
   # false
puts 123456789012345.instance_of? Bignum
   # true
```

*Exercise:* What’s the largest `Fixnum`? What’s the smallest `Fixnum`? How do you explain this?

Why is there a difference between a `Fixnum` and a `Bignum`?

```
puts [1, 2, 3, 4, 5].length
   # 5
puts "Hey".class
   # String
puts "John".class.superclass  # print the superclass of a String
   # Object
puts String.ancestors  # print the hierarchy
   # String
```
Enumerable
Comparable
Object
Kernel

For String and Fixnum, which of these ancestors are classes, and which are mixins? Explain.

*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:

```ruby
if s.kind_of? Integer then this_method else that_method end
```

Why?