Modules and Mixins

A **Module** is a construct in Ruby that we haven't seen before.

A module looks a bit like a class, but it's not a class. Modules serve two purposes in Ruby:

1. Namespaces
2. Mixins

We'll look at these separately. Modules as namespaces are fairly straightforward; mixins are more fun.

Bertie

If someone says "Bertie" what do you think of?

In the **namespace** of general conversation in Ireland, the name "Bertie" binds to the man above.

Another Bertie

Bertie is also a brand of shoe.

In the **namespace** of shoe fashion, the name "Bertie" binds to a brand of shoe.

Namespaces

A namespace then is a scope within which names have certain meanings.

Classes and methods form namespaces. An instance variable `@count` in a class is different from a variable of the same name in an unrelated class.

As we've seen, in Ruby a class and all its superclasses form a single namespace.

So if a class and its (indirect) superclass both introduce an instance variable `@count`, it's the same instance variable in both cases.

Modules in Ruby

A **module** in Ruby is a namespace.

It packages together related classes, constants and methods.

As it’s a namespace, you needn’t worry about clashes with other names in the rest of the program

We’ll look at an example...

A Simple Module

```
module Trig
   PI = 3.141592654

   def Trig.sin(x)
      # implementation missing
   end

   def Trig.cos(x)
      # implementation missing
   end
end
```

Note that these are **not** instance methods.

A module may look like a class, but it **cannot be instantiated**. Use a module as follows:

```
puts “value of pi is #{Trig::PI}”

tan_x = Trig.sin(x) / Trig.cos(x)
```

Modules and files

Usually a module will be defined in a file (or several files).

If the Trig module were defined in a file called trig.rb, it would be used as follows:

```
require ‘trig’

puts “value of pi is #{Trig::PI}”

tan_x = Trig.sin(x) / Trig.cos(x)
```

A Software Engineering Problem

David and Aoife work together using Ruby on an enterprise application for Opticians.

David is building the user interface.

Aoife is building the core application.

David creates a class called **Frame** to represent a frame in the user interface.

Aoife creates a class called **Frame** to represent the frame of a pair of glasses.

They test their own code -- all is well.

They integrate their code -- name clashes appear.

Modules to the rescue

The solution is for John to work in a module called e.g. `GUI`

Mary works in a module called e.g. `Core`.

(A module can contain multiple classes and span multiple files of course.)

A class in the `GUI` module that needs to use the `Frame` class in the `GUI` module simply refers to it as:

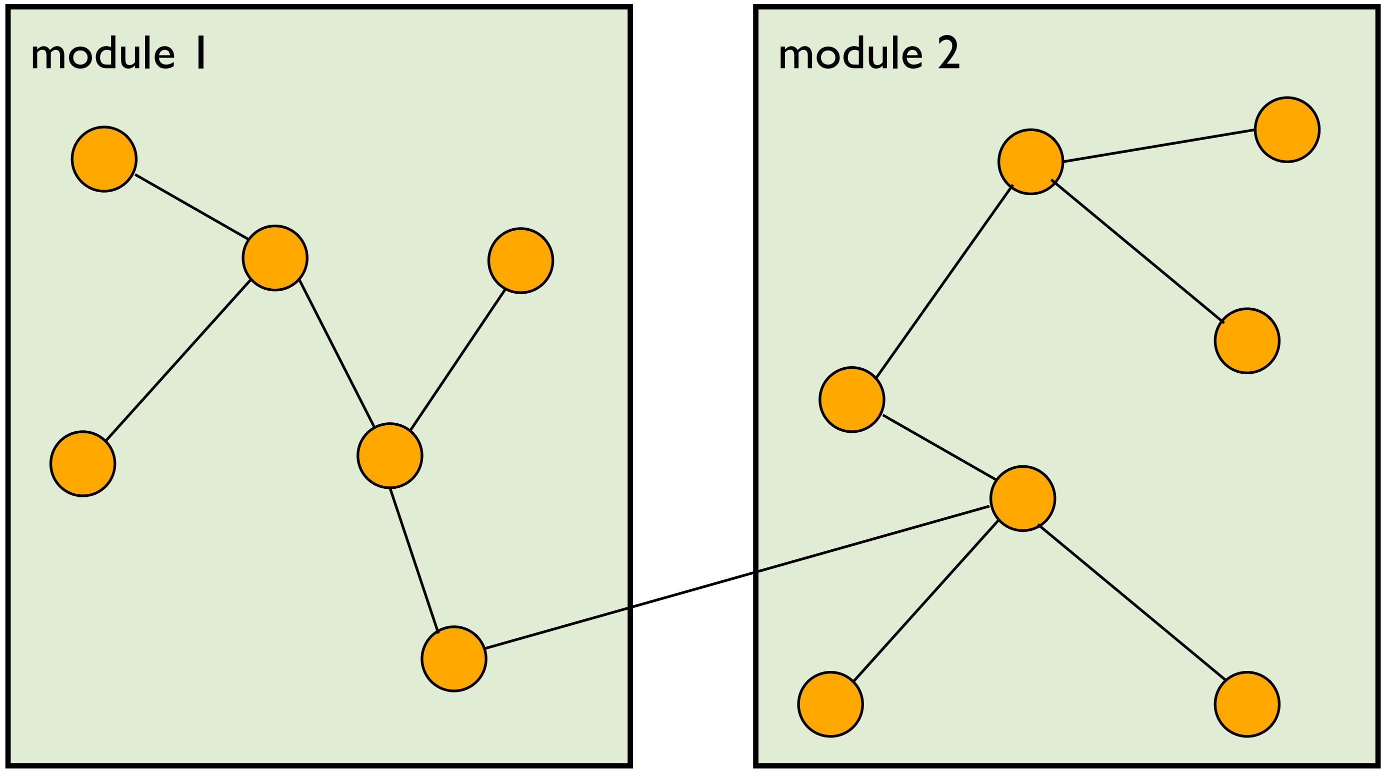
```
Frame
```

A class in the `GUI` module that needs to use the `Frame` class in the `Core` module refers to it as:

```
Core::Frame
```

Modules provide structure

A Ruby program of 13 classes might be divided into 2 modules as follows:



Ruby modules are similar Java packages in this sense.

The really exciting part of Ruby modules is that they can be used as **mixins**.

A simple Mixin

Modules can be used to represent traits that many classes might wish to have. If you were modelling different types of people, you might want to make the ‘chatty’ trait a module:

```
module Chatty
  def hi
    puts ‘How are you? You look great!’
  end
end
```

This is a normal instance method.

This looks odd -- the **`hi`** method is a normal instance method, not a module method (prefixed with `Chatty.`). How is this method used?

This method can only be invoked when we **mix** the module **in** with a class...

Using a mixin

Say we have a Person class and we want Person objects to be chatty.

So we **mix in** the Chatty module:

```
class Person
  include Chatty
  attr_accessor :name
  def initialize name
    @name = name
  end
end
```

This means that the hi method of the Chatty module is now a public method of the Person class:

```
p = Person.new ‘Mike’
p.hi  # prints ‘How are you? You look great!’
```

A bit more mixing

The really neat thing with mixins is that the module can use variables that bind to the variables of the class it's mixed with:

```
module Chatty
  def hi
    puts “Hey, how are you? My name is #{@name}”
  end
end
```

This looks funny... the module uses the instance variable **`@name`** but this is never given a value.

The hi method of the Chatty module now accesses the private instance variable **`@name`** of the Person class:

```
p = Person.new ‘Mike’
p.hi  # prints ‘How are you? My name is Mike’
```

The Comparable mixin

Many useful modules for mixing are provided to you. One is the **Comparable** mixin.

Comparable provides the following 6 comparison operators as public methods:

==

!=

>

<

<=

>=

Of course, in your class you have to define how to do the comparisons. This is easier than you think...

The <=> operator

The <=> operator is defined as follows.

x <=> y returns

0	if x and y are equal
1	if x is greater than y
-1	if x is less than y

To mix Comparable with one of your own classes, you simply have to define the <=> operator on your own class. The 6 comparison operators on the previous page you then get “for free.”

Comparable example

To mix Comparable with one of your own classes, you simply have to define the <=> operator on your own class. The 6 comparison operators on the previous page you then get "for free."

```
class Employee
  attr_accessor :salary

  include Comparable

  def initialize (name, salary)
    @name = name
    @salary = salary
  end

  def <=> (emp)
    if @salary > emp.salary
      return 1
    elsif @salary == emp.salary
      return 0
    else
      return -1
    end
  end
end
```

Comparing Mixins and Inheritance

A class is a big thing. A domain abstraction. Inheriting from a class is a big deal. The new class gets everything from the superclass (and adds more itself).

A module for mixing is a small thing. Think of it as a **trait**, like printable, comparable, sortable. Or even sporty, thoughtful, lazy...

Use inheritance for fixed, unchanging relationships between classes.

A use mixins when there are traits that several classes in your application share. Think of a cheerful, 'mix and match' approach to class creation.

Mixins and Multiple Inheritance

C++ provides **multiple inheritance**. This means that a class can have several superclasses.

This led to a number of theoretical and practical problems.

The designers of Java therefore only permitted single inheritance, but permitted a class to implement any number of **interfaces**.

A Java interface carries no implementation of its methods, so it doesn't help in implementing the class that uses it.

A **mixin** is a bit like an interface that has its own implementation.

Aren't we back to multiple inheritance?

Summary

Modules provide two new facilities.

They can be used to structure larger programs, as a way to package related program elements (classes, constants) together.

They can be mixed in with a class to create a mixin. In this case the public instance methods of the module become part of the interface of the class, and instance variables used in the module bind to the instance variables defined in the class.

Creating a module for mixing is a great way to share functionality between a number of classes.