RUBY MODULES AND MIXINS

Modules are a way of grouping together methods, classes, and constants. Modules give you two major benefits.

- Modules provide a *namespace* and prevent name clashes.
- Modules implement the *mixin* facility.

Modules define a namespace, a sandbox in which your methods and constants can play without having to worry about being stepped on by other methods and constants.

Syntax:

```
module Identifier
   statement1
   statement2
   ...........
end
```

Module constants are named just like class constants, with an initial uppercase letter. The method definitions look similar, too: Module methods are defined just like class methods.

As with class methods, you call a module method by preceding its name with the module's name and a period, and you reference a constant using the module name and two colons.

Example:

```
#!/usr/bin/ruby

# Module defined in trig.rb file

module Trig
   PI = 3.141592654
   def Trig.sin(x)
   # ..
   end
   def Trig.cos(x)
   # ..
   end
end
```

We can define one more module with same function name but different functionality:

```
#!/usr/bin/ruby

# Module defined in moral.rb file

module Moral
   VERY_BAD = 0
   BAD = 1
   def Moral.sin(badness)
   # ...
   end
end
```

Like class methods, whenever you define a method in a module, you specify the module name followed by a dot and then the method name.

Ruby *require* Statement:

The require statement is similar to the include statement of C and C++ and the import statement of Java. If a third program wants to use any defined module, it can simply load the module files using the Ruby *require* statement:

Syntax:

```
require filename
```

Here, it is not required to give **.rb** extension along with a file name.

Example:

```
$LOAD_PATH << '.'

require 'trig.rb'
require 'moral'

y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::VERY_BAD)
```

Here we are using **$LOAD_PATH << '.'** to make Ruby aware that included files must be searched in the current directory. If you do not want to use $LOAD_PATH then you can use **require_relative** to include files from a relative directory.

IMPORTANT: Here, both the files contain same function name. So, this will result in code ambiguity while including in calling program but modules avoid this code ambiguity and we are able to call appropriate function using module name.

Ruby *include* Statement:

You can embed a module in a class. To embed a module in a class, you use the *include* statement in the class:

Syntax:

```
include modulename
```

If a module is defined in a separate file, then it is required to include that file using *require* statement before embedding module in a class.

Example:

Consider following module written in *support.rb* file.

```
module Week
   FIRST_DAY = "Sunday"
   def Week.weeks_in_month
      puts "You have four weeks in a month"
   end
   def Week.weeks_in_year
      puts "You have 52 weeks in a year"
   end
end
```

Now, you can include this module in a class as follows:

```
#!/usr/bin/ruby
$LOAD_PATH << '.'
require "support"

class Decade
include Week
   no_of_yrs=10
   def no_of_months
      puts Week::FIRST_DAY
      number=10*12
      puts number
   end
end
d1=Decade.new
puts Week::FIRST_DAY
Week.weeks_in_month
Week.weeks_in_year
d1.no_of_months
```

This will produce the following result:

```
Sunday
You have four weeks in a month
You have 52 weeks in a year
Sunday
120
```

Mixins in Ruby:

Before going through this section, I assume you have knowledge of Object Oriented Concepts.

When a class can inherit features from more than one parent class, the class is supposed to show multiple inheritance.

Ruby does not support multiple inheritance directly but Ruby Modules have another wonderful use. At a stroke, they pretty much eliminate the need for multiple inheritance, providing a facility called a *mixin*.

Mixins give you a wonderfully controlled way of adding functionality to classes. However, their true power comes out when the code in the mixin starts to interact with code in the class that uses it.

Let us examine the following sample code to gain an understand of mixin:

```
module A
   def a1
   end
   def a2
   end
end
module B
   def b1
   end
   def b2
   end
end

class Sample
include A
include B
   def s1
   end
end

samp=Sample.new
samp.a1
samp.a2
samp.b1
samp.b2
samp.s1
```

Module A consists of the methods a1 and a2. Module B consists of the methods b1 and b2. The class Sample includes both modules A and B. The class Sample can access all four methods, namely, a1, a2, b1, and b2. Therefore, you can see that the class Sample inherits from both the modules. Thus, you can say the class Sample shows multiple inheritance or a *mixin*.