Ruby provides many different constructs that can be used to develop object oriented programs. Some closely resemble the tools you'd expect to find in most mainstream languages, but others are a bit more exotic. In particular, Ruby's modules provide a treasure trove of features not typically found elsewhere.

One thing that modules do is give your objects a way to share bits of common functionality, allowing you to cut down on the amount of duplication in your programs. We can explore one way to use modules for this purpose by rewriting our combo lock example from Chapter 1.

To start, we need to extract the various methods we wrote into a module, rather than defining them directly on an individual object. Something like the code below will do the trick.

```
module Locklike
  def set_password(new_password)
    @stored_password = new_password unless locked?
  end

  def lock
    @locked = true if @stored_password
  end

  def unlock(entered_password)
    @locked = false if entered_password == @stored_password
  end

  def locked?
    @locked
  end
end
```

At this point, the methods that implement the combo lock behavior all live inside the Locklike module, but are not directly callable yet. In order to actually use them, they need to be mixed into at least one object. Every Ruby object provides an extend method for this purpose, and you can see the basic idea of how it works through the following example.

```
combo_lock_a = Object.new
combo_lock_a.extend(Locklike)
combo_lock_a.set_password("1337")
combo_lock_a.lock

combo_lock_b = Object.new
combo_lock_b.extend(Locklike)
combo_lock_b.set_password("1234")
combo_lock_b.lock

combo_lock_a.unlock("1337")
```

```
combo_lock_b.unlock("1337")

puts combo_lock_a.locked? #=> false
puts combo_lock_b.locked? #=> true

combo_lock_b.unlock("1234")
puts combo_lock_b.locked? #=> true
```

While the example itself may seem a bit dull, it illustrates the mixin concept well. We can see from it that each time an object is extended by a module, the code from that module effectively becomes a part of that object's definition. References to instance variables in modules point to whatever object they were mixed into, and so there is no danger of corruption due to shared state between combo_lock_a and combo_lock_b. This allows the two to share a bit of common behavior while still operating as idependent, standalone objects.

While we'll see in the next chapter that there is probably a better way to solve this particular problem, the ability to mix in modules on a per-object basis is a very useful technique. Before we move on, I'd like to underscore this point by taking a look at a real use case from Ruby's standard library.

The open-uri standard library is one example of a particularly clever Ruby API. It allows you to treat online resources as if they were ordinary file handles, as shown in the example below.

```
require "open-uri"
puts open("http://google.com").read #=> outputs the page source
```

By patching the globally available open method, open-uri extends a familiar API to give it some new functionality. To do this in a natural way, open-uri uses another standard library, stringio, to emulate the behavior of a file handle. That's where the read method comes from in the example above.

While what we've seen so far is useful as-is, open-uri goes a step farther by extending the StringIO objects it creates with context-specific metadata, some of which is shown in the following irb session.

```
>> page = open("http://google.com")
=> #<StringIO:0x00000100b27838>
>> page.base_uri
=> #<URI::HTTP:0x00000100b27d60 URL:http://www.google.com/>
>> page.content_type
=> "text/html"
>> page.status
=> ["200", "OK"]
```

Since the purpose of StringIO is simply to allow you to represent a string as an IO like object, it's pretty clear that these http specific methods are not implemented on StringIO objects directly. Instead, they come from a module that adds some attribute accessors for this purpose, called OpenURI::Meta. The following code demonstrates how to implement this sort of design.

```
module Meta
```

```
    attr_accessor :base_uri, :content_type, :status
  end

  page = StringIO.new("Hello World")
  page.extend(Meta)
  page.base_uri = "http://example.com"

  puts page.base_uri  #=> "http://example.com"
  puts page.read      #=> "Hello World"
```

The only thing new here is the use of attr_accessor, a method provided by modules that can be used for defining simple accessor methods without typing them out explicitly. We won't get bogged down in the details of how that works now, so the important thing to take away from this example is that the rest of what's going on here exactly mirrors our Locklike example, in which a bit of new functionality is being mixed into an existing object to extend its capabilities.

You're certainly not expected to fully understand the hows and whys of the module construct and the concept of mixins from seeing these two examples alone. We've got a whole book to explore these ideas in more detail. This chapter was just meant to open your eyes to what is possible, and hopefully cause some interesting questions to arise in your mind as well.