

The Nature of Objects

Imagine that you have just returned home from some fantastic and strenuous journey. You unlock the door to your house with one of the many keys on your key ring, walk inside and grab a glass from the cupboard. You place your keys back in your pocket and then walk into the kitchen and turn the light on. You take the glass and walk over to the fridge, placing your cup under the ice dispenser, filling the glass with ice. You then press the button for the water dispenser, filling the remaining space in the glass with water. Finally, you walk with your glass of ice water into the living room, sit down on your sofa, kick your feet up on the coffee table, and then relax and enjoy a nice drink.

To perform this simple task, you have had to interact with a number of different objects, including your pants pocket, the keyring containing the key for your front door, the doorknob, the kitchen light switch, the cupboard's handle, and the buttons on your fridge. There isn't anything remarkable about your interactions with these objects, in fact, the whole sequence sounds like something a trained monkey could do.

However, if you look just below the surface, you'll see that each of these objects has been carefully designed to make interacting with it as simple as possible. You do not need to think about the tumblers of a lock to successfully make use of a key, nor do you need to think about the hinges of a cupboard to open it. The same goes for the electrical wiring and water lines for your house. Since you don't even need to think about how these primitive systems work, you certainly don't need to contemplate the inner workings of a mechanical ice dispenser to be able to push a button and watch ice pop out.

Just as mechanical design can be used to simplify interactions with physical objects, software can be designed in the same manner. Programming languages that provide the building blocks that carry this analogy into the digital sphere are known as object-oriented languages. Among object-oriented languages, Ruby stands out as being particularly rich. We can catch a glimpse of why this is the case by exploring a simple example.

If we adapt our previous thought experiment, we can imagine a home that rather uses a digital keypad instead of a key for unlocking its doors. In such a system, you would key in password that would be verified by a program before the door could be unlocked via an electrical signal. While I've simplified greatly, the program for such a system might look something like what you see below if it were written in Ruby.

```
#####  
# Combination-lock Implementation #  
#####  
  
combo_lock = Object.new  
  
def combo_lock.set_password(new_password)  
  @stored_password = new_password unless locked?  
end  
  
def combo_lock.lock  
  @locked = true if @stored_password  
end  
  
def combo_lock.unlock(entered_password)  
  @locked = false if entered_password == @stored_password  
end  
  
def combo_lock.locked?  
  @locked  
end  
  
#####  
# Combination-lock Interface #  
#####  
  
combo_lock.set_password("1337")  
combo_lock.lock  
  
p combo_lock.locked? #=> true  
  
combo_lock.unlock("1336")  
  
p combo_lock.locked? #=> true  
  
combo_lock.unlock("1337")  
  
p combo_lock.locked? #=> false
```

In this example, we've used a freshly created Ruby object as a building block and imbued it with the behaviors of a combination lock by defining certain methods on it. Each method performs a procedure, possibly doing some manipulation of the object's internal data, such as its locked status or stored password. However, looking at the implementation is a bit like looking at the tumblers of a mechanical lock: it tells you a lot about how it works, but isn't

particularly good at illustrating how the object ought to be used. Conversely, if we look at the code that interfaces with the `combo_lock` object, we are able to see a whole lot about how it's meant to be used without catching even a glimpse of how it is implemented.

While we're only scratching the surface with this trivial example, the benefits of being able to separate the concept of an interface from its implementation are vast. For example, we could swap out our implementation that stored passwords in plain text with one that uses cryptographic hashes without ever having to change the interface code, as illustrated in the example below.

```
#####  
# Crypto-combination lock #  
#####  
  
require "digest/sha1"  
  
combo_lock = Object.new  
  
def combo_lock.set_password(new_password)  
  hashed_password = Digest::SHA1.hexdigest(new_password)  
  @stored_password = hashed_password unless locked?  
end  
  
def combo_lock.lock  
  @locked = true if @stored_password  
end  
  
def combo_lock.unlock(entered_password)  
  hashed_password = Digest::SHA1.hexdigest(entered_password)  
  @locked = false if hashed_password == @stored_password  
end  
  
def combo_lock.locked?  
  @locked  
end
```

While building a single logical machine is easy, the interesting work is in composing behaviors of many objects in a system to model complex scenarios. Because the examples we've looked at so far use only the most primitive tools Ruby has to offer, they may seem quite boring. However, as we explore more elaborate problems, we'll see Ruby's object system get a lot more interesting.

Ruby is intensely human-centric language that is designed to allow us to model our software in ways that closely mirror how we model things in the physical world. This book challenges you to gain an intuitive understanding of what that means by exploring its consequences in realistic scenarios.

The Mixin Concept

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 independent, 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.

Classes as Blueprints

This would hardly be a book on object oriented programming if we didn't promptly find ourselves discussing object classes. In many languages, the class construct is a golden hammer, and writing object oriented programs is simply a metaphor for working with classes. While Ruby is a bit more flexible than purely class-based languages about how you can organize and interact with your objects, coding without the class construct would feel like you had both hands tied behind your back.

To explore some of the benefits of using classes, we can revisit the lockbox example from the previous chapters one last time. I've made a couple modifications to highlight the benefits of using classes, but otherwise, the code below should look familiar.

```
class ComboLock
  def initialize(new_password)
    @stored_password = new_password
    @locked          = true
  end

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

  def lock
    @locked = true
  end

  def locked?
    @locked
  end
end
```

The following example demonstrates how to make use of the `ComboLock` class that we just created.

```
combo_lock_a = ComboLock.new("1337")
combo_lock_b = ComboLock.new("1234")
```

```
combo_lock_a.unlock("1337")
combo_lock_b.unlock("1337")

puts combo_lock_a.locked?
puts combo_lock_b.locked?

combo_lock_b.unlock("1234")
puts combo_lock_b.locked?
```

The main difference between this code and the examples shown in the previous two chapters is that when we use classes, we have control over what happens at the time the object is being created. In a sense, our class definition acts as a blueprint for a special kind of object that has its own behaviors associated with it.

An interesting detail about Ruby classes is that they provide a way to utilize mixins as well. For example, the hybrid approach shown below is functionally equivalent to our previous `ComboLock` definition, but uses a module to provide most of the functionality.

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

  def lock
    @locked = true
  end

  def locked?
    @locked
  end
end

class ComboLock
  include Locklike

  def initialize(new_password)
    @stored_password = new_password
    @locked = true
  end
end
```

The `include` method is very similar to `extend`, except that the methods get mixed into the methods defined by a class or a module, rather than the receiving object itself. In the context of class definitions, this means that methods mixed in via `include` become accessible on instances of the class they were included into.

This technique becomes more interesting when you implementation details that you want to share across several classes. For example, in the very first

chapter we demonstrated a combolock object that had an identical interface to the one we've been working with here, but used a cryptographic hash function to store digital fingerprints of the passwords rather than the passwords themselves. Here's an example of how to build both types of combolocks while sharing some common code.

```
require "digest/sha1"

module Locklike
  def unlock(entered_password)
    @locked = false if valid_password?(entered_password)
  end

  def lock
    @locked = true
  end

  def locked?
    @locked
  end
end

class ComboLock
  include Locklike

  def initialize(new_password)
    @stored_password = new_password
    @locked = true
  end

  def valid_password?(entered_password)
    @stored_password == entered_password
  end
end

class CryptoComboLock
  include Locklike

  def initialize(new_password)
    @stored_password = sha1(new_password)
    @locked = true
  end

  def valid_password?(entered_password)
    @stored_password == sha1(entered_password)
  end

  def sha1(password)
    Digest::SHA1.hexdigest(password)
  end
end
```

Changing a single line in our example code from before illustrates that this modular approach does work as advertised.

```

combo_lock_a = ComboLock.new("1337")
combo_lock_b = CryptoComboLock.new("1234") #<--- this is the change

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? #=> false

```

Those experienced with other object oriented languages but relatively inexperienced with Ruby's modules may be thinking that this looks suspiciously similar to class inheritance and may be wondering what the tradeoffs are. The good news is that converting to a traditional inheritance based design is trivial, as demonstrated by the code shown below.

```

require "digest/sha1"

class Locklike
  # methods implemented same as in modular example
  # provides lock, unlock, locked?
end

class ComboLock < Locklike
  def initialize(new_password)
    @stored_password = new_password
    @locked          = true
  end

  def valid_password?(entered_password)
    @stored_password == entered_password
  end
end

class CryptoComboLock < Locklike
  def initialize(new_password)
    @stored_password = sha1(new_password)
    @locked          = true
  end

  def valid_password?(entered_password)
    @stored_password == sha1(entered_password)
  end

  def sha1(password)
    Digest::SHA1.hexdigest(password)
  end
end

```

Writing the code this way makes `ComboLock` and `CryptoComboLock` direct descendants of the abstract base class `Locklike`. However, in this particular scenario, this provides absolutely no practical benefits, only a few warts.

- While the code is functionally equivalent to our modular example, the conversion of `Locklike` into a class makes it possible to initialize instances of that class directly, unless we add some logic to prevent that from happening. In other words, `Locklike.new` will generate a useless object unless you take additional steps to prevent that from happening.
- Ruby does not allow inheriting from multiple classes, so once you pick a parent class, you are stuck with that parent for life. On the other hand, you can mix as many modules as you'd like into a single class.
- While it doesn't apply to this particular example, those who are familiar with using `super` to forward method calls upstream to be handled by a parent may be concerned that the same would not apply to modules. However, Ruby uses a very nice way of doing method resolution, and `super` calls work as gracefully with modules as they do with classes. If this doesn't make sense to you right now, don't worry, it'll be covered in great detail later in the book.

All of the above points are things that you need to keep in mind when building object oriented systems in Ruby. Classes are very useful, but inheritance is less widespread and less valuable as a tool in Ruby than it might be in other languages due to Ruby's mixin concept. That does not mean that inheritance is never useful, but is only meant to say that if you've always thought in terms of it, you'll need to add a few more tools to your toolbox to effectively design and implement Ruby programs in a natural way.

At this point, we've caught a glimpse of what Ruby Objects, Classes, and Modules have to offer, and hopefully at least generated the kinds of questions that the rest of this book is meant to answer. While these opening examples have been a bit contrived for the sake of simplicity, I can promise you that the rest of the book is chock full of practical examples that will help you think about these issues in the context of realistic scenarios. Please feel free to skip around to what interests you, and enjoy the rest of the materials to come!