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 similiar 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!