Chapter 5

Sharing Functionality: Inheritance, Modules, and Mixins

One of the accepted principles of good design is the elimination of unnecessary duplication. We work hard to make sure that each concept in our application is expressed just once in our code.[1]

We've already seen how classes help. All the methods in a class are automatically accessible to instances of that class. But there are other, more general types of sharing that we want to do. Maybe we're dealing with an application that ships goods. Many forms of shipping are available, but all forms share some basic functionality (weight calculation, perhaps). We don't want to duplicate the code that implements this functionality across the implementation of each shipping type. Or maybe we have a more generic capability that we want to inject into a number of different classes. For example, an online store may need the ability to calculate sales tax for carts, orders, quotes, and so on. Again, we don't want to duplicate the sales tax code in each of these places.

In this chapter, we'll look at two different (but related) mechanisms for this kind of sharing in Ruby. The first, *class-level inheritance*, is common in object-oriented languages. We'll then look at *mixins*, a technique that is often preferable to inheritance. We'll wind up with a discussion of when to use each.

Inheritance and Messages

In the previous chapter we saw that when puts needs to convert an object to a string, it calls that object's to_s method. But we've also written our own classes that don't explic-

1. Why? Because the world changes. And when you adapt your application to each change, you want to know that you've changed exactly the code you need to change. If each real-world concept is implemented at a single point in the code, this becomes vastly easier.

itly implement to_s. Despite this, objects of these classes respond successfully when we call to_s on them. How this works has to do with inheritance, subclassing, and how Ruby determines what method to run when you send a message to an object.

Inheritance allows you to create a class that is a refinement or specialization of another class. This class is called a *subclass* of the original, and the original is a *superclass* of the subclass. People also talk of *child* and *parent* classes.

The basic mechanism of subclassing is simple. The child inherits all of the capabilities of its parent class—all the parent's instance methods are available in instances of the child.

Let's look at a trivial example and then later build on it. Here's a definition of a parent class and a child class that inherits from it:

Download **samples/tutmodules_1.rb**

```
class Parent
  def say_hello
    puts "Hello from #{self}"
  end
end
p = Parent.new
p.say_hello
# Subclass the parent...
class Child < Parent
end
c = Child.new
c.say_hello
```

produces:

```
Hello from #<Parent:0x0a40c4>
Hello from #<Child:0x0a3d68>
```

The parent class defines a single instance method, say_hello. We call it by creating a new instance of the class and store a reference to that instance in the variable p.

We then create a subclass using class Child < Parent. The < notation means we're creating a subclass of the thing on the right; the fact that we use less-than presumably signals that the child class is supposed to be a specialization of the parent.

Note that the child class defines no methods, but when we create an instance of it, we can call say_hello. That's because the child inherits all the methods of its parent. Note also that when we output the value of self—the current object—it shows that we're in an instance of class Child, even though the method we're running is defined in the parent.

The superclass method returns the parent of a particular class:

Download **samples/tutmodules_2.rb**

```
class Parent
end
class Child < Parent
end
puts "The superclass of Child is #{Child.superclass}"
```

produces:

```
The superclass of Child is Parent
```

But what's the superclass of Parent?

```
class Parent
end
puts "The superclass of Parent is #{Parent.superclass}"
```

produces:

```
The superclass of Parent is Object
```

If you don't define an explicit superclass when defining a class, Ruby automatically makes the built-in class Object that class's parent. Let's go further:

```
puts "The superclass of Object is #{Object.superclass}"
```

produces:

```
The superclass of Object is BasicObject
```

1.9

Class BasicObject was introduced in Ruby 1.9. It is used in certain kinds of metaprogramming, acting as a blank canvas. What's its parent?

```
puts "The superclass of BasicObject is #{BasicObject.superclass.inspect}"
```

produces:

```
The superclass of BasicObject is nil
```

So, we've finally reached the end. BasicObject is the root class of our hierarchy of classes. Given any class in any Ruby application, you can ask for its superclass, then the superclass of that class, and so on, and you'll eventually get back to BasicObject.

We've seen that if you call a method in an instance of class Child and that method isn't in Child's class definition, Ruby will look in the parent class. It goes deeper than that, because if the method isn't defined in the parent class, Ruby continues looking in the parent's parent, the parent's parent's parent, and so on, through the ancestors until it runs out of classes.

And this explains our original question. We can work out why to_s is available in just about every Ruby object. to_s is actually defined in class Object. Because Object is an ancestor of every Ruby class (except BasicObject), instances of every Ruby class have a to_s method defined:

Download **samples/tutmodules_6.rb**

```
class Person
  def initialize(name)
    @name = name
  end
end
p = Person.new("Michael")
puts p
```

produces:

```
#<Person:0x0a4efc>
```

We saw in the previous chapter that we can override the to_s method:

Download samples/tutmodules_7.rb

```
class Person
  def initialize(name)
    @name = name
  end
  def to_s
    "Person named #{@name}"
  end
end
p = Person.new("Michael")
puts p
```

produces:

```
Person named Michael
```

Armed with our knowledge of subclassing, we now know there's nothing special about this. The puts method calls to_s on its arguments. In this case, the argument is a Person object. Because class Person defines a to_s method, that method is called. If it hadn't defined a to_s method, then Ruby looks for (and finds) to_s in Person's parent class, Object.

It is common to use subclassing to add application-specific behavior to a standard library or framework class. If you've used Ruby on Rails,[2] you'll have subclassed ActionController when writing your own controller classes. Your controllers get all the behavior of the base controller and add their own specific handlers to individual user actions. If you've used the FXRuby GUI framework,[3] you'll have used subclassing to add your own application-specific behavior to Fox's standard GUI widgets.

Here's a more self-contained example. Ruby comes with a library called GServer that implements basic TCP server functionality. You add your own behavior to it by subclassing the GServer class. Let's use that to write some code that waits for a client to connect on a socket and then returns the last few lines of the system log file. This is an example of something that's actually quite useful in long-running applications—by building in such a server, you can access the internal state of the application while it is running (possibly even remotely).

The GServer class handles all the mechanics of interfacing to TCP sockets. When you create a GServer object, you tell it the port to listen on.[4] Then, when a client connects, the GServer object calls its serve method to handle that connection. Here's the implementation of that serve method in the GServer class:

```
def serve(io)
end
```

2. http://www.rubyonrails.com

3. http://www.fxruby.org

4. You can tell it a lot more, as well. We chose to keep it simple here.

As you can see, it does nothing. That's where our own LogServer class comes in:

Download samples/tutmodules_9.rb

```
require 'gserver'
class LogServer < GServer
  def initialize
    super(12345)
  end
  def serve(client)
    client.puts get_end_of_log_file
  end
private
  def get_end_of_log_file
    File.open("/var/log/system.log") do |log|
      log.seek(-1000, IO::SEEK_END)   # back up 1000 characters from end
      log.gets                        # ignore partial line
      log.read                        # and return rest
    end
  end
end
server = LogServer.new
server.start.join
```

I don't want to focus too much on the details of running the server. Instead, let's look at how inheritance has helped us with this code. First, notice that our LogServer class inherits from GServer. This means that a log server is a kind of GServer, sharing all the GServer functionality. It also means we can add our own specialized behavior.

The first such specialization is the initialize method. We want our LogServer to run on TCP port 12345. That's a parameter that would normally be passed to the GServer constructor. So, within the initialize method of the LogServer, we want to invoke the initialize method of GServer, our parent, passing it the port number. We do that using the Ruby keyword super. When you invoke super, Ruby sends a message to the parent of the current object, asking it to invoke a method of the same name as the method invoking super. It passes this method the parameters that were passed to super.

This is a crucial step and one often forgotten by folks new to OO. When you subclass another class, you are responsible for making sure the initialization required by that class gets run. This means that, unless you know it isn't needed, you'll need to put a call to super somewhere in your subclass's initialize method. (If your subclass doesn't need an initialize method, then there's no need to do anything, because it will be the parent class's initialize method that gets run when your objects get created.)

So, by the time our initialize method finishes, our LogServer object will be a fully fledged TCP server, all without us having to write any protocol-level code. Down at the end of our program, we start the server. The call to join causes our program to wait for the server to exit before itself exiting.

While our server is running, it will receive connections from external clients. These invoke the serve method in the server object. Remember that empty method in class GServer? Well,

our LogServer class provides its own implementation. And because it gets found by Ruby first when it's looking for methods to execute, it's our code that gets run whenever GServer accepts a connection. And our code reads the last few lines of the log file and returns them to the client:[5]

```
$ telnet 127.0.0.1 12345
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Jul  7 13:39:44 dave com.apple.syncservices.SyncServer[54938]:...
Jul  7 13:39:44 dave com.apple.syncservices.SyncServer[54938]:...
Jul  7 13:39:44 dave com.apple.syncservices.SyncServer[54938]:...
Jul  7 13:42:40 dave login[54768]: DEAD_PROCESS: 54768 ttys001
Jul  7 13:45:34 dave mdworker[54977]: fcntl to turn on F_CHECK...
Jul  7 13:48:44 dave mdworker[54977]: fcntl to turn on F_CHECK...
Connection closed by foreign host.
```

The use of the serve method shows a common idiom when using subclassing. A parent class assumes that it will be subclassed and calls a method that it expects its children to implement. This allows the parent to take on the brunt of the processing but to invoke what are effectively hook methods in subclasses to add application-level functionality. As we'll see at the end of this chapter, just because this idiom is common doesn't make it good design.

So, instead, let's look at *mixins*, a different way of sharing functionality in Ruby code. But, before we look at mixins, we'll need to get familiar with Ruby *modules*.

Modules

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 support the mixin facility.

Namespaces

As you start to write bigger and bigger Ruby programs, you'll naturally find yourself producing chunks of reusable code—libraries of related routines that are generally applicable. You'll want to break this code into separate files so the contents can be shared among different Ruby programs.

Often this code will be organized into classes, so you'll probably stick a class (or a set of interrelated classes) into a file. However, there are times when you want to group things together that don't naturally form a class.

5. You can also access this server from a web browser by connecting to http://127.0.0.1:12345.

Inheritance and Mixins

Some object-oriented languages (such as C++) support multiple inheritance, where a class can have more than one immediate parent, inheriting functionality from each. Although powerful, this technique can be dangerous, because the inheritance hierarchy can become ambiguous.

Other languages, such as Java and C#, support single inheritance. Here, a class can have only one immediate parent. Although cleaner (and easier to implement), single inheritance also has drawbacks—in the real world objects often inherit attributes from multiple sources (a ball is both a *bouncing thing* and a *spherical thing*, for example).

Ruby offers an interesting and powerful compromise, giving you the simplicity of single inheritance and the power of multiple inheritance. A Ruby class has only one direct parent, so Ruby is a single-inheritance language. However, Ruby classes can include the functionality of any number of *mixins* (a mixin is like a partial class definition). This provides a controlled multiple-inheritance-like capability with none of the drawbacks. We'll explore mixins more beginning on the following page.

An initial approach may be to put all these things into a file and simply load that file into any program that needs it. This is the way the C language works. However, this approach has a problem. Say you write a set of the trigonometry functions sin, cos, and so on. You stuff them all into a file, trig.rb, for future generations to enjoy. Meanwhile, Sally is working on a simulation of good and evil, and she codes a set of her own useful routines, including be_good and sin, and sticks them into moral.rb. Joe, who wants to write a program to find out how many angels can dance on the head of a pin, needs to load both trig.rb and moral.rb into his program. But both define a method called sin. Bad news.

The answer is the module mechanism. 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. The trig functions can go into one module:

Download **samples/tutmodules_10.rb**

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

```
end
```

and the good and bad "moral" methods can go into another:

Download **samples/tutmodules_11.rb**

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

Module constants are named just like class constants, with an initial uppercase letter.[6] The method definitions look similar, too: module methods are defined just like class methods.

If a third program wants to use these modules, it can simply load the two files (using the Ruby require statement. In order to reference the name sin unambiguously, our code can then qualify the name using the name of the module containing the implementation we want, followed by ::, the scope resolution operator:

```
require 'trig'
require 'moral'
y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::VERY_BAD)
```

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.

Mixins

Modules have another, wonderful use. At a stroke, they pretty much eliminate the need for inheritance, providing a facility called a *mixin*.

In the previous section's examples, we defined module methods, methods whose names were prefixed by the module name. If this made you think of class methods, your next thought may well be "What happens if I define instance methods within a module?" Good question. A module can't have instances, because a module isn't a class. However, you can *include* a module within a class definition. When this happens, all the module's instance methods are suddenly available as methods in the class as well. They get *mixed in*. In fact, mixed-in modules effectively behave as superclasses.

6. But we will conventionally use all uppercase letters when writing them.

Download samples/tutmodules_13.rb

```
module Debug
  def who_am_i?
    "#{self.class.name} (\##{self.object_id}): #{self.to_s}"
  end
end
class Phonograph
  include Debug
  # ...
end
class EightTrack
  include Debug
  # ...
end
ph = Phonograph.new("West End Blues")
et = EightTrack.new("Surrealistic Pillow")

ph.who_am_i?   # =>   "Phonograph (#330450): West End Blues"
et.who_am_i?   # =>   "EightTrack (#330420): Surrealistic Pillow"
```

By including the Debug module, both the Phonograph and EightTrack classes gain access to the who_am_i? instance method.

We'll make a couple of points about the include statement before we go on. First, it has nothing to do with files. C programmers use a preprocessor directive called #include to insert the contents of one file into another during compilation. The Ruby include statement simply makes a reference to a module. If that module is in a separate file, you must use require (or its less commonly used cousin, load) to drag that file in before using include. Second, a Ruby include does not simply copy the module's instance methods into the class. Instead, it makes a reference from the class to the included module. If multiple classes include that module, they'll all point to the same thing. If you change the definition of a method within a module, even while your program is running, all classes that include that module will exhibit the new behavior.[7]

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's take the standard Ruby mixin Comparable as an example. The Comparable mixin adds the comparison operators (<, <=, ==, >=, and >), as well as the method between?, to a class. For this to work, Comparable assumes that any class that uses it defines the operator <=>. So, as a class writer, you define one method, <=>, include Comparable, and get six comparison functions for free.

Let's try this with a simple Person class.

7. Of course, we're speaking only of methods here. Instance variables are always per object, for example.

We'll make people comparable based on their names:

Download **samples/tutmodules_14.rb**

```
class Person
  include Comparable
  attr_reader :name
  def initialize(name)
    @name = name
  end
  def to_s
     "#{@name}"
  end
  def <=>(other)
    self.name <=> other.name
  end
end
p1 = Person.new("Matz")
p2 = Person.new("Guido")
p3 = Person.new("Larry")
# Compare a couple of names
if p1 > p2
  puts "#{p1.name}'s name > #{p2.name}'s name"
end
# Sort an array of Person objects
puts "Sorted list:"
puts [ p1, p2, p3].sort
```

produces:

```
Matz's name > Guido's name
Sorted list:
Guido
Larry
Matz
```

Note that we included Comparable in our Person class and then defined a <=>. We were then able to perform comparisons (such as p1 > p2) and even sort an array of Person objects.

Iterators and the Enumerable Module

The Ruby collection classes (Array, Hash, and so on) support a large number of operations that do various things with the collection: traverse it, sort it, and so on. You may be thinking, "Gee, it'd sure be nice if *my* class could support all these neat-o features, too!" (If you actually thought that, it's probably time to stop watching reruns of 1960s television shows.)

Well, your classes *can* support all these neat-o features, thanks to the magic of mixins and module Enumerable. All you have to do is write an iterator called each, which returns the elements of your collection in turn. Mix in Enumerable, and suddenly your class supports things such as map, include?, and find_all?. If the objects in your collection implement meaningful ordering semantics using the <=> method, you'll also get methods such as min, max, and sort.

Composing Modules

Enumerable is a standard mixin, implementing a bunch of methods in terms of the host class's each method. One of the methods defined by Enumerable is inject, which we saw back on page 80. This method applies a function or operation to the first two elements in the collection and then applies the operation to the result of this computation and to the third element, and so on, until all elements in the collection have been used.

Because inject is made available by Enumerable, we can use it in any class that includes the Enumerable module and defines the method each. Many built-in classes do this.

Download **samples/tutmodules_15.rb**

```
[ 1, 2, 3, 4, 5 ].inject(:+)   # =>   15
( 'a'..'m').inject(:+)         # =>   "abcdefghijklm"
```

We could also define our own class that mixes in Enumerable and hence gets inject support:

Download **samples/tutmodules_16.rb**

```
class VowelFinder
  include Enumerable
  def initialize(string)
    @string = string
  end
  def each
    @string.scan(/[aeiou]/) do |vowel|
      yield vowel
    end
  end
end
```

Download **samples/tutmodules_17.rb**

```
vf = VowelFinder.new("the quick brown fox jumped")

vf.inject(:+)   # =>   "euiooue"
```

Notice that we've used the same pattern in the call to inject in these examples—we're using it to perform a summation. When applied to numbers, it returns the arithmetic sum; when applied to strings, it concatenates them. We can use a module to encapsulate this functionality too:

Download **samples/tutmodules_18.rb**

```
module Summable
  def sum
    inject(:+)
  end
end
class Array
  include Summable
end
```

```
class Range
  include Summable
end
class VowelFinder
  include Summable
end
```

Download samples/tutmodules_19.rb

```
[ 1, 2, 3, 4, 5 ].sum   # =>   15
('a'..'m').sum          # =>   "abcdefghijklm"

vf = VowelFinder.new("the quick brown fox jumped")
vf.sum                  # =>   "euiooue"
```

Instance Variables in Mixins

People coming to Ruby from C++ often ask, “What happens to instance variables in a mixin? In C++, I have to jump through some hoops to control how variables are shared in a multiple-inheritance hierarchy. How does Ruby handle this?”

Well, for starters, it’s not really a fair question. Remember how instance variables work in Ruby: the first mention of an @-prefixed variable creates the instance variable *in the current object,* self.

For a mixin, this means that the module you mix into your client class (the *mixee*?) may create instance variables in the client object and may use attr_reader and friends to define accessors for these instance variables. For instance, the Observable module in the following example adds an instance variable @observer_list to any class that includes it:

```
module Observable
  def observers
    @observer_list ||= []
  end
  def add_observer(obj)
    observers << obj
  end
  def notify_observers
    observers.each {|o| o.update }
  end
end
```

However, this behavior exposes us to a risk. A mixin’s instance variables can clash with those of the host class or with those of other mixins. The example that follows shows a class that uses our Observer module but that unluckily also uses an instance variable called @observer_list. At runtime, this program will go wrong in some hard-to-diagnose ways:

```
class TelescopeScheduler
  # other classes can register to get notifications
  # when the schedule changes
  include Observable
```

```
  def initialize
    @observer_list = []  # folks with telescope time
  end
  def add_viewer(viewer)
    @observer_list << viewer
  end
  # ...
end
```

For the most part, mixin modules don't use instance variables directly—they use accessors to retrieve data from the client object. But if you need to create a mixin that has to have its own state, ensure that the instance variables have unique names to distinguish them from any other mixins in the system (perhaps by using the module's name as part of the variable name). Alternatively, the module could use a module-level hash, indexed by the current object ID, to store instance-specific data without using Ruby instance variables:

Download **samples/tutmodules_22.rb**

```
module Test
  State = {}
  def state=(value)
    State[object_id] = value
  end
  def state
    State[object_id]
  end
end
```

Download **samples/tutmodules_23.rb**

```
class Client
  include Test
end

c1 = Client.new
c2 = Client.new
c1.state = 'cat'
c2.state = 'dog'

c1.state   # =>   "cat"
c2.state   # =>   "dog"
```

A downside of this approach is that the data associated with a particular object will not get automatically deleted if the object is deleted.

Resolving Ambiguous Method Names

One of the other questions folks ask about mixins is, how is method lookup handled? In particular, what happens if methods with the same name are defined in a class, in that class's parent class, and in a mixin included into the class?

The answer is that Ruby looks first in the immediate class of an object, then in the mixins included into that class, and then in superclasses and their mixins. If a class has multiple modules mixed in, the last one included is searched first.

Inheritance, Mixins, and Design

Inheritance and mixins both allow you to write code in one place and effectively inject that code into multiple classes. So, when do you use each?

As is usual with most questions of design, the answer is, to some extent, it depends. However, over the years developers have come up with some pretty clear general guidelines to help us decide.

First, let's look at subclassing. Classes in Ruby are related to the idea of types. It would be natural to say that "cat" is a string and [1,2] is an array. And that's another way of saying that the class of "cat" is String and the class of [1,2] is Array. When we create our own classes, you can think of it as adding new types to the language. And when we subclass either a built-in class or our own class, we're creating a *subtype*.

Now, a lot of research has been done on type theories. One of the more famous results is the *Liskov Substitution Principle*. Formally, this states: "Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T." What this means is that you should be able to substitute an object of a child class wherever you use an object of the parent class—the child should honor the parent's contract. There's another way of looking at this: we should be able to say that the child object *is a* kind of the parent. We're used to saying this in English: a car *is a* vehicle, a cat *is an* animal, and so on. This means that a cat should, at the very least, be capable of doing everything we say that an animal can do.

So, when you're looking for subclassing relationships while designing your application, be on the lookout for these *is-a* relationships.

But...here's the bad news. In the real world, there really aren't that many true *is a* relationships. Instead, it's far more common to have *has a* or *uses a* relationships between things. The real world is built using composition, not strict hierarchies.

In the past, we've tended to gloss over that fact when programming. Because inheritance was the only scheme available for sharing code, we got lazy and said things like "My Person class is a subclass of my DatabaseWrapper class."[8] But a person object *is not* a kind of database wrapper object. A person object *uses* a database wrapper to provide persistence services.

Is this just a theoretical issue? No! Inheritance represents an incredibly tight coupling of two components. Change a parent class, and you risk breaking the child class. But, even worse, if code that uses objects of the child class relies on those objects also having methods defined

8. Indeed, the Rails framework makes just this mistake.

in the parent, then all that code will break, too. The parent class's implementation leaks through the child classes and out into the rest of the code. With a decent-sized program, this becomes a serious inhibitor to change.

And that's where we need to move away from inheritance in our designs. Instead, we need to be using *composition* wherever we see a case of A *uses a* B or A *has a* B. Our persisted Person object won't subclass DataWrapper. Instead, it'll construct a reference to a database wrapper object and use that object reference to save and restore itself.

But that can also make code messy. And that's where a combination of mixins and metaprogramming comes to the rescue, because we can say this:

```
class Person
  include Persistable
  # ...
end
```

instead of

```
class Person < DataWrapper
  # ...
end
```

If you're new to object-oriented programming, this discussion probably feels remote and abstract. But as you start to code larger and larger programs, I urge you to think about the issues discussed here. Try to reserve inheritance for the times where it is justified. And try to explore all the cool ways that mixins let you write decoupled, flexible code.