CHAPTER 3

Mastering the Dynamic Toolkit

If you've done even a little bit of Ruby, you probably have a sense of the great flexibility and power that it offers as a language. This chapter is designed to underscore that point, specifically by showing you what can be accomplished when you unleash the power of Ruby onto itself. When you hear that everything is an object in Ruby, it's easy to forget that classes, modules, and methods all fall into that category as well. With enough imagination, we can think of all sorts of interesting applications that fall out of this elegant design.

Take the fact that all of our programmatic constructs can be represented as first-order data objects and combine it with the ability to modify any of them at runtime. Then mix in the idea that everything from defining a new function to calling a method that does not exist can be detected and handled by custom code. Top this off with first-rate reflection capabilities and you'll find that Ruby is a perfect foundation for writing highly dynamic applications.

On the surface, these ideas may seem a bit esoteric or academic in nature. But when you get down to it, there are a lot of practical uses for having such a high degree of flexibility baked into Ruby. Because Ruby's dynamic nature is a huge part of what makes the language what it is, you'll find no shortage of real examples in this chapter. These run the gamut from dynamic interface generation to safely modifying preexisting code at runtime. But to get your feet wet, we'll dive in with the same head-first approach found in the rest of the chapters of this book. We're about to look at the code behind Jim Weirich's `BlankSlate` class, which provides an excellent case study of what can be accomplished with the dynamic toolkit that Ruby provides for us.

If you feel a bit overwhelmed at first, don't be discouraged—each individual topic will be discussed later on in this chapter in greater detail. For now, let's just try to have fun and see just how powerful Ruby really is.

BlankSlate: A BasicObject on Steroids

Although Ruby 1.9 has `BasicObject` as a very lightweight object designed to be used for implementing objects with dynamic interfaces and other similar tasks, Ruby 1.8 users weren't so lucky. In light of this, `BlankSlate` became a fairly common tool for those who needed an object that didn't do much of anything. One of the practical applications of this somewhat abstract object was in implementing the XML generator in the *builder* gem. In case you've not seen XML Builder before, it is a tool that turns Ruby code like this:

```
builder = Builder::XmlMarkup.new(:target=>STDOUT, :indent=>2)
builder.person { |b| b.name("Jim"); b.phone("555-1234") }
```

into XML output like this:

```
<person>
  <name>Jim</name>
  <phone>555-1234</phone>
</person>
```

Without going into too much detail, it is obvious from this example that `Builder::XmlMarkup` implements a dynamic interface that can turn your method calls into matching XML output. But if it had simply inherited from `Object`, you'd run into certain naming clashes wherever a tag had the same name as one of `Object`'s instance methods.

Builder works by capturing calls to missing methods, which means it has trouble doing its magic whenever a method is actually defined. For example: if `XmlMarkup` were just a subclass of `Object`, with no methods removed, you wouldn't be able produce the following XML, due to a naming conflict:

```
<class>
  <student>Greg Gibson</student>
</class>
```

The underlying issue here is that `Kernel#class` is already defined for a different purpose. Of course, if we instead inherit from an object that has very few methods to begin with, this greatly lessens our chance for a clash.

`BasicObject` certainly fits the bill, as you can see with a quick glance at its instance methods via *irb*:

```
>> BasicObject.instance_methods
=> [:==, :equal?, :!, :!=, :instance_eval, :instance_exec, :__send__]
```

These methods form the lowest common denominator for Ruby, so `BasicObject` is pretty reasonable in its offerings. The key thing to remember is that a `BasicObject` is fully defined by this limited set of features, so you shouldn't expect anything more than that. Although this makes perfect sense in Ruby 1.9's object hierarchy, it's somewhat interesting to see that `BlankSlate` takes an entirely different approach.

On Ruby 1.8, there was no `BasicObject` class to speak of, so instead of starting off with a tiny set of methods, `BlankSlate` had to do something to get rid of the significant baggage that rides along with Ruby's `Object` class. This is done in an especially clever way, and a quick *irb* session complete with the expected noise that results from removing potentially important methods shows the primary interesting features of `BlankSlate`:

```
>> class A < BlankSlate; end
=> nil
>> A.new
NoMethodError: undefined method 'inspect' for #<A:0x42ac34>
...
>> A.reveal(:inspect)
=> #<Proc:0x426558@devel/rbp_code/dynamic_toolkit/blankslate.rb:43 (lambda)>
>> A.new
NoMethodError: undefined method 'to_s' for #<A:0x425004>
...
>> A.reveal(:to_s)
=> #<Proc:0x422e30@devel/rbp_code/dynamic_toolkit/blankslate.rb:43 (lambda)>
>> A.new
=> #<A:0x425004>
>> A.new.methods
NoMethodError: undefined method 'methods' for #<A:0x425004>
        from (irb):8
        from /Users/sandal/lib/ruby19_1/bin/irb:12:in '<main>'
>> A.reveal(:methods)
=> #<Proc:0x41ed6c@devel/rbp_code/dynamic_toolkit/blankslate.rb:43 (lambda)>
>> A.new.methods
=> [:inspect, :to_s, :methods, :__id__, :instance_eval, :__send__]
```

After reading through this code, you should be able to get a sense of how it works. `BlankSlate` isn't really a blank slate at all; instead, it's an object that acts like a blank slate by hiding all of its methods until you tell them explicitly to reveal themselves. This clever bit of functionality allows `BlankSlate`'s initial instance methods to be kept to the absolute minimum. Everything else can be explicitly revealed later, as needed.

`BlankSlate` does this per subclass, so you can have different customized minimal objects for different purposes in your system. Predictably, you can also rehide functions, including those that you add yourself:

```
>> A.new.foo
=> "Hi"
>> A.hide(:foo)
=> A
>> A.new.foo
NoMethodError: undefined method 'foo' for #<A:0x425004>
        from (irb):18
        from /Users/sandal/lib/ruby19_1/bin/irb:12:in '<main>'
>> A.hide(:inspect)
=> A
>> A.new
NoMethodError: undefined method 'inspect' for #<A:0x40a484>
...
```

All in all, although it is a bit more heavyweight than `BasicObject`, the `BlankSlate` class may have its uses even on Ruby 1.9, due to this ability to seamlessly hide and restore functionality on the fly. If you were thinking that this sounds like complicated stuff, you might be surprised. The core implementation of `BlankSlate` is relatively straightforward. Of course, the devil is in the details, but the most interesting bits can be understood with a little explanation:

```
class BlankSlate
  class << self

    # Hide the method named +name+ in the BlankSlate class.  Don't
    # hide +instance_eval+ or any method beginning with "__".
    def hide(name)
      if instance_methods.include?(name) and
        name !~ /^(__|instance_eval)/
        @hidden_methods ||= {}
        @hidden_methods[name] = instance_method(name)
        undef_method name
      end
    end

    def find_hidden_method(name)
      @hidden_methods ||= {}
      @hidden_methods[name] || superclass.find_hidden_method(name)
    end

    # Redefine a previously hidden method so that it may be called on a blank
    # slate object.
    def reveal(name)
      unbound_method = find_hidden_method(name)
      fail "Don't know how to reveal method '#{name}'" unless unbound_method
      define_method(name, unbound_method)
    end
  end

  instance_methods.each { |m| hide(m) }
end
```

As you can see, the class is simply three short class methods, followed by the call that causes all of `BlankSlate`'s instance methods to be hidden. Let's start by taking a closer look at the `hide()` method.

```
def hide(name)
  if instance_methods.include?(name) && name !~ /^(__|instance_eval)/
    @hidden_methods ||= {}
    @hidden_methods[name] = instance_method(name)
    undef_method name
  end
end
```

Here you can see that `BlankSlate` first checks to make sure that the method name passed to `hide()` exists within the currently visible instance methods. Once it checks to make

sure the method is not one of the special reserved methods, it begins the process of storing and hiding the specified method.

The technique used here is simply to initialize a `@hidden_methods` hash within the class, and then assign the method name as a key to the associated `UnboundMethod` object. An `UnboundMethod` can be thought of as roughly similar to a `Proc` object, but rather than being truly anonymous, it is later hooked up to an object that knows how to make use of the function, which is typically an object of the same class. As a trivial example, we can play around with `String#reverse` to illustrate this point:

```
>> a = String.instance_method(:reverse)
=> #<UnboundMethod: String#reverse>
>> a.bind("foo").call
=> "oof"
```

We'll take a closer look at this a little later, but suffice it to say that by grabbing the `UnboundMethod` before removing the method definition, we have a way of restoring the behavior in the future.

I assume you can get the gist of what's going on with `find_hidden_method()` just by inspection, so we can jump straight into the most interesting code in `BlankSlate`, the method that actually restores the old functionality:

```
# Redefine a previously hidden method so that it may be called on a blank
# slate object.
def reveal(name)
  unbound_method = find_hidden_method(name)
  fail "Don't know how to reveal method '#{name}'" unless unbound_method
  define_method(name, unbound_method)
end
```

Here, the `find_hidden_method()` helper is used to recall an `UnboundMethod` by name. If it doesn't manage to find a matching name in the `@hidden_methods` hash, an error is raised. However, assuming the lookup went according to plan, we can see that the method is redefined to call the newly rebound method. All the original arguments are passed on to the restored method call, so you end up with the original behavior restored.

Although we've shown the key components of `BlankSlate` here, we haven't gone into the full details yet. It's worth mentioning that because `BlankSlate` inherits from `Object` and not `BasicObject`, it has to do some additional magic to deal with module inclusion, and it also must handle methods added to `Object` and `Kernel`. We'll get to these a little later, in "Registering Hooks and Callbacks" on page 32. For now, let's just quickly review the concepts we've touched on.

In this initial exploration phase, we've caught a glimpse of `define_method`, `instance_methods`, `instance_method`, `undef_method`, and `UnboundMethod`. Or in English, we've seen an example of how to use reflection to determine the names of the instance methods on a class, copy their implementations into objects that could then be keyed by name in a hash, undefine them, and later restore them by building up a new definition programmatically. You have probably noticed that even though these concepts are very

high-level, they're essentially ordinary Ruby code, without any sort of magic. The rest of this chapter will serve to reinforce that point.

Now that we've seen a few of these concepts in action, we'll slow down and discuss what each one of them actually means, while diving even deeper into dynamic Ruby territory. In this next example, we'll look at a favorite topic for budding Rubyists. I'm going to share the secrets behind building flexible interfaces that can be used for domain-specific applications. The heavy-handed term for this sort of thing is an "internal domain-specific language," but we don't need to be so fancy, as it can create misconceptions. The ability to build pleasant domain-specific interfaces is a key feature of Ruby, and deserves some discussion—no matter what you want to call it.

Building Flexible Interfaces

Heads up: you might start to feel a bit of déjà vu in this section. What we'll cover here is basically a recap of what was discussed in Chapter 2, *Designing Beautiful APIs*, mixed in with a little dynamic help here and there. Though each step may seem fairly inconsequential, the end result is quite powerful.

When implementing a flexible domain-specific interface, the idea is that we want to strip away as much boilerplate code as possible so that every line expresses something meaningful in the context of our domain. We also want to build up a vocabulary to work with, and express our intents in that vocabulary as much as possible. A domain-specific interface puts Ruby in the background: available when you need it, but not as in-your-face as ordinary programmatic interfaces tend to be. An easy comparison would be to look at the difference between some elementary `Test::Unit` code and its RSpec equivalent.*

First, we'll look at the vanilla `Test::Unit` code:

```
class NewAccountTest < Test::Unit

  def setup
    @account = Account.new
  end

  def test_must_start_with_a_zero_balance
    assert_equal Money.new(0, :dollars), @account.balance
  end

end
```

To a Rubyist, this code might seem relatively clear, straightforward, and expressive. However, its defining characteristic is that it looks like any other Ruby code, with all the associated benefits and drawbacks. Others prefer a different approach, which you can clearly see in this RSpec code:

* This example is from the RSpec home page (*http://rspec.info*), with minor modifications.

```
describe Account, " when first created" do

  before do
    @account = Account.new
  end

  it "should have a balance of $0" do
    @account.balance.should eql(Money.new(0, :dollars))
  end

end
```

When we read RSpec code, it feels like we're reading specifications rather than Ruby code. Many people feel this is a major advantage, because it encourages us to express ourselves in a domain-specific context. When it comes to testing, this does create some controversy, because although the RSpec code is arguably more readable here, the `Test::Unit` code is certainly less magical. But in the interest of avoiding politics, I've shown this example to illustrate the difference between two styles, not to advocate one over the other.

Even though some particular uses of domain-specific interfaces can be a touchy subject, you'll find many cases where they come in handy. To help you get a feel for how they come together, we'll be looking at some problems and their solutions. We're about to walk through the lifecycle of wrapping a nice interface on top of Prawn's `Document` class. Don't worry about the particular domain; instead, focus on the techniques we use so that you can make use of them in your own projects.

Making instance_eval( ) Optional

In the previous chapter, we covered a common pattern for interface simplification, allowing you to turn code like this:

```
pdf = Prawn::Document.new
pdf.text "Hello World"
pdf.render_file "hello.pdf"
```

into something like this:

```
Prawn::Document.generate("hello.pdf") do
  text "Hello World"
end
```

As you'll recall, this trick is relatively straightforward to implement:

```
class Prawn::Document
  def self.generate(file, *args, &block)
    pdf = Prawn::Document.new(*args)
    pdf.instance_eval(&block)
    pdf.render_file(file)
  end
end
```

However, there is a limitation that comes with this sort of interface. Because we are evaluating the block in the context of a `Document` instance, we do not have access to anything but the local variables of our enclosing scope. This means the following code won't work:

```
class MyBestFriend

  def initialize
    @first_name = "Paul"
    @last_name  = "Mouzas"
  end

  def full_name
    "#{@first_name} #{@last_name}"
  end

  def generate_pdf
    Prawn::Document.generate("friend.pdf") do
      text "My best friend is #{full_name}"
    end
  end

end
```

It'd be a shame to have to revert to building this stuff up manually, and a bit messy to rely on storing things in local variables. Luckily, there is a middle-of-the-road option: we can optionally yield a `Document` object. Here's how we'd go about doing that:

```
class Prawn::Document
  def self.generate(file, *args, &block)
    pdf = Prawn::Document.new(*args)
    block.arity < 1 ? pdf.instance_eval(&block) : block.call(pdf)
    pdf.render_file(file)
  end
end
```

This new code preserves the old `instance_eval` behavior, but allows a new approach as well. We can now write the following code without worry:

```
class MyOtherBestFriend

  def initialize
    @first_name = "Pete"
    @last_name  = "Johansen"
  end

  def full_name
    "#{@first_name} #{@last_name}"
  end

  def generate_pdf
    Prawn::Document.generate("friend.pdf") do |doc|
      doc.text "My best friend is #{full_name}"
    end
  end
```

```
end
```

Here, the code is an ordinary closure, and as such, can access the instance methods and variables of the enclosing scope. Although we need to go back to having an explicit receiver for the PDF calls, our `Document.generate` method can still do its necessary setup and teardown for us, salvaging some of its core functionality.

The feature that makes this all possible is `Proc#arity`. This method tells you how many arguments, if any, the code block was given. Here's a few examples as an illustration:

```
>> lambda { |x| x + 1 }.arity
=> 1
>> lambda { |x,y,z| x + y + z }.arity
=> 3
>> lambda { 1 }.arity
=> 0
```

As you can see, because `Proc` objects are just objects themselves, we can do some reflective inquiry to find out how many arguments they're expecting to process. Although not strictly related to our task, it's worth mentioning that you can accomplish the same thing with methods as well:

```
>> Comparable.instance_method(:between?).arity
=> 2
>> Fixnum.instance_method(:to_f).arity
=> 0
```

Although our use of an `arity` check was confined to a relatively simple task here, the technique is general. Any time you want to conditionally handle something based on how many block arguments are present, you can use this general approach.

That having been said, even if you never use this trick for anything else, knowing how to selectively `instance_eval` a block is important. As you'll see through the rest of this section, a key part of developing a pleasant domain-specific interface is maintaining flexibility. If you limit yourself to an all-or-nothing choice between your sexy shortcuts and the bland, low-level API, frustration is inevitable. Of course, because Ruby is so dynamic, you should never be forced to make this decision.

We'll now move on to another key component of flexible interface design: the use of `method_missing` and `send` to dynamically route messages within your objects.

Handling Messages with method_missing( ) and send( )

Continuing on a theme, we can look at more Prawn code to see how to make things a bit more dynamic. We'll be looking at elementary drawing operations here, but you can substitute your own problem mentally. As in other examples, the actual domain does not matter.

In Prawn, there are two ordinary ways to generate some shapes and then draw them onto the page. The first is the most simple—just drawing the paths, and then calling one of `stroke`, `fill`, or `fill_and_stroke`:

```
Prawn::Document.generate("shapes.pdf") do
  fill_color "ff0000"

  # Fills a red circle
  circle_at [100,100], :radius => 25
  fill

  # Strokes a transparent circle with a black border and a line extending
  # from its center point
  circle_at [300,300] :radius => 50
  line [300,300], [350, 300]
  stroke

  # Fills and strokes a red hexagon with a black border
  polygon [100, 250], [200, 300], [300, 250],
          [300, 150], [200, 100], [100, 150]
  fill_and_stroke
end
```

This isn't too bad, but for some needs, a block form is better. This makes it clearer what paint operation is being used, and may be a bit easier to extend:

```
Prawn::Document.generate("shapes.pdf") do
  fill_color "ff0000"

  # Fills a red circle
  fill { circle_at [100,100], :radius => 25 }

  # Strokes a transparent circle with a black border and a line extending
  # from its center point
  stroke do
    circle_at [300,300] :radius => 50
    line [300,300], [350, 300]
  end

  fill_and_stroke do
    # Fills and strokes a red hexagon with a black border
    polygon [100, 250], [200, 300], [300, 250],
            [300, 150], [200, 100], [100, 150]
  end

end
```

This may be a bit more readable, especially the middle one, in which multiple paths need to be stroked. However, it still feels like more work than we'd really want. Wouldn't things be nicer this way?

```
Prawn::Document.generate("shapes.pdf") do
  fill_color "ff0000"

  fill_circle_at [100,100], :radius => 25
```

```
  stroke_circle_at [300,300] :radius => 50
  stroke_line [300,300], [350, 300]

  fill_and_stroke_polygon [100, 250], [200, 300], [300, 250],
                          [300, 150], [200, 100], [100, 150]
end
```

This has a nice, declarative feel to it. Obviously though, we don't want to define four methods for every graphics drawing operation. This is especially true when you think of the nature of what each of these would look like. Let's take `stroke` for example:

```
def stroke_some_method(*args)
  some_method(*args)
  stroke
end
```

Repeat that ad nauseam for every single drawing method, and keep up this pattern every time a new one is added? No way! Maybe this sort of repetition would be tolerated over in Java-land, but in Ruby, we can do better. The answer lies in dynamically interpreting method calls.

When you attempt to call a method that doesn't exist in Ruby, you see an exception raised by default. However, Ruby provides a way to hook into this process and intercept the call before an error can be raised. This is done through the method `method_missing`.

To give a very brief introduction to how it works, let's take a quick spin in *irb*:

```
>> def method_missing(name, *args, &block)
>>   puts "You tried to call #{name} with #{args.inspect}"
>>   puts "Epic Fail!"
>> end
=> nil
>> 1.fafsafs
You tried to call fafsafs with []
Epic Fail!
=> nil
>> "kitten".foo("bar", "baz")
You tried to call foo with ["bar", "baz"]
Epic Fail!
```

By including a `method_missing` hook at the top level, all unknown messages get routed through our new method and print out our message. As you can see, the name of the message as well as the arguments are captured. Of course, this sort of global change is typically a very bad idea, and serves only as an introduction. But if you're feeling ambitious, take a moment to think about how this technique could be used to solve the problem we're working on here, before reading on.

Did you have any luck? If you did attempt this exercise, what you would find is that `method_missing` isn't very useful on its own. Typically, it is used to do part of a job and then route the work to another function. The way we do this is by making use of

`Kernel#send`, which allows us to call a method by just passing a symbol or string, followed by any arguments:

```
>> "foo".send(:reverse)
=> "oof"
>> [1,2,3].send("join", "|")
=> "1|2|3"
```

Does this clue make things a bit clearer? For those who didn't try to build this on their own, or if you attempted it and came up short, here's how to make it all work:

```
# Provides the following shortcuts:
#
#    stroke_some_method(*args) #=> some_method(*args); stroke
#    fill_some_method(*args) #=> some_method(*args); fill
#    fill_and_stroke_some_method(*args) #=> some_method(*args); fill_and_stroke
#
def method_missing(id,*args,&block)
  case(id.to_s)
  when /^fill_and_stroke_(.*)/
    send($1,*args,&block); fill_and_stroke
  when /^stroke_(.*)/
    send($1,*args,&block); stroke
  when /^fill_(.*)/
    send($1,*args,&block); fill
  else
    super
  end
end
```

As the documentation describes, this hook simply extracts the paint command out from the method call, and then sends the remainder as the function to execute. All arguments (including an optional block) are forwarded on to the real method. Then, when it returns, the specified paint method is called.

It's important to note that when the patterns do not match, `super` is called. This allows objects up the chain to do their own `method_missing` handling, including the default, which raises a `NoMethodError`. This prevents something like `pdf.the_shiny_kitty` from failing silently, as well as the more subtle `pdf.fill_circle`.

Although this is just a single example, it should spark your imagination for all the possibilities. But it also hints at the sort of looseness that comes with this approach. Prawn will happily accept `pdf.fill_and_stroke_start_new_page` or even `pdf.stroke_stroke_stroke_line` without complaining. Any time you use the `method_missing` hook, these are the trade-offs you must be willing to accept. Of course, by making your hooks more robust, you can get a bit more control, but that starts to defeat the purpose if you take it too far.

The best approach is to use `method_missing` responsibly and with moderation. Be sure to avoid accidental silent failures by calling `super` for any case you do not handle, and don't bother using it if you want things to be ironclad. In cases where there is a relatively small set of methods you want to generate dynamically, a solution using

`define_method` might be preferred. That having been said, when used as a shortcut alternative to a less pleasant interface, `method_missing` can be quite helpful, especially in cases where the messages you'll need to accept are truly dynamic.

The techniques described so far combined with some of the methods shown in the previous chapter will get you far in building a domain-specific interface. We're about to move on to other dynamic Ruby topics, but before we do that, we'll cover one more cheap trick that leads to clean and flexible interfaces.

Dual-Purpose Accessors

One thing you will notice when working with code that has an `instance_eval`-based interface is that using ordinary setters can be ugly. Because you need to disambiguate between local variables and method calls, stuff like this can really cramp your style:

```
Prawn::Document.generate("accessors.txt") do

  self.font_size = 10
  text "The font size is now #{font_size}"

end
```

It's possible to make this look much nicer, as you can see:

```
Prawn::Document.generate("accessors.txt") do

  font_size 10
  text "The font size is now #{font_size}"

end
```

The concept here isn't a new one; we covered it in the previous chapter. We can use Ruby's default argument syntax to determine whether we're supposed to be getting or setting the attribute:

```
class Prawn::Document

  def font_size(size = nil)
    return @font_size unless size
    @font_size = size
  end

  alias_method :font_size=, :font_size

end
```

As I said before, this is a relatively cheap trick with not much that is special to it. But the first time you forget to do it and find yourself typing `self.foo = bar` in what is supposed to be a domain-specific interface, you'll be sure to remember this technique.

One thing to note is that you shouldn't break the normal behavior of setters from the outside. We use `alias_method` here instead of `attr_writer` to ensure down the line that there won't be any difference between the following two lines of code:

```
pdf.font_size = 16
pdf.font_size(16)
```

Though not essential, this is a nice way to avoid potential headaches at a very low cost, so it's a good habit to get into when using this technique.

When we combine all the tactics we've gone over so far, we've got all the essential components for building flexible domain-specific interfaces. Before we move on to the next topic, let's review the main points to remember about flexible interface design:

- As mentioned in the previous chapter, using `instance_eval` is a good base for writing a domain-specific interface, but has some limitations.
- You can use a `Proc#arity` check to provide the user with a choice between `instance_eval` and yielding an object.
- If you want to provide shortcuts for certain sequences of method calls, or dynamic generation of methods, you can use `method_missing` along with `send()`.
- When using `method_missing`, be sure to use `super()` to pass unhandled calls up the chain so they can be handled properly by other code, or eventually raise a `NoMethodError`.
- Normal attribute writers don't work well in `instance_eval`-based interfaces. Offer a dual-purpose reader/writer method, and then alias a writer to it, and both external and internal calls will be clear.

With these tips in mind, let's move on to another topic. It's time to shift gears from per-class dynamic behavior to individual objects.

Implementing Per-Object Behavior

An interesting aspect of Ruby is that not only can objects have per-class method definitions, but they can also have per-object behaviors. What this means is that each and every object carries around its own unique identity, and that the class definition is simply the blueprint for the beginning of an object's lifecycle.

Let's start with a simple *irb* session to clearly illustrate this concept:

```
>> a = [1,2,3]
=> [1, 2, 3]
>> def a.secret
>>   "Only this object knows the secrets of the world"
>> end
=> nil
>> a.secret
=> "Only this object knows the secrets of the world"
>> [1,2,3,4,5].secret
NoMethodError: undefined method 'secret' for [1, 2, 3, 4, 5]:Array
        from (irb):17
        from :0
>> [1,2,3].secret
NoMethodError: undefined method 'secret' for [1, 2, 3]:Array
```

```
        from (irb):18
        from :0
```

Here, using a familiar method definition syntax, we add a special method called `secret` to the array we've assigned to `a`. The remaining examples show that only `a` gets this new method definition. If the last one surprised you a bit, remember that most objects in Ruby are not immediate values, so two arrays set to `[1,2,3]` are not the same object, even if they contain the same data. More concisely:

```
>> [1,2,3].object_id
=> 122210
>> a.object_id
=> 159300
```

So when we talk about each object having its own behavior, we mean exactly that here. You may be wondering at this point what uses there might be for such a feature. An interesting abstract example might be to note that class methods are actually just per-object behavior on an instance of the class `Class`, but I think it'd be more productive to give you a concrete example to sink your teeth into.

We're going to take a look at how to build a simple stubbing system for use in testing. In the testing chapter, I recommended *flexmock* for this purpose, and I still do, but going through the process of building a tiny stubbing framework will show a good use case for our current topic.

Our goal is to create a system that will generate canned responses to certain method calls, without modifying their original classes. This is an important feature, because we don't want our stubbed method calls to have a global effect during testing. Our target interface will be something like this:

```
user = User.new
Stubber.stubs(:logged_in?, :for => user, :returns => true)
user.logged_in? #=> true
```

We'll start with a very crude approach in *irb* to get a feel for the problem:

```
>> class User; end
=> nil
>> user = User.new
=> #<User:0x636b4>
>> def user.logged_in?
>>   true
>> end
=> nil
>> user.logged_in?
=> true
>> another_user = User.new
=> #<User:0x598d0>
>> another_user.logged_in?
NoMethodError: undefined method 'logged_in?' for #<User:0x598d0>
        from (irb):40
        from :0
```

This is essentially the behavior we want to capture: per-object customization that doesn't affect the class definition generally. Of course, to do this dynamically takes a little more work than the manual version. Our first hurdle is that the technique used in the earlier `BlankSlate` example doesn't work out of the box here:

```
>> user.define_method(:logged_in?) { true }
NoMethodError: undefined method 'define_method' for #<User:0x40ed90>
        from (irb):17
        from /Users/sandal/lib/ruby19_1/bin/irb:12:in '<main>'
```

As it turns out, each object hides its individual space for method definitions (called a singleton class) from plain view. However, we can reveal it by using a special syntax:

```
>> singleton = class << user; self; end
=> #<Class:#<User:0x40ed90>>
```

My earlier clues about class methods being per-object behavior on an instance of `Class` should come to mind here. We often use this syntax when we need to define a few class methods:

```
class A
  class << self
    def foo
      "hi"
    end

    def bar
      "bar"
    end
  end
end
```

The possible new thing here is that `self` can be replaced by any old object. So when you see `class << user; self; end`, what's really going on is we're just asking our object to give us back its singleton class. Once we have that in hand, we can define methods on it. Well, almost:

```
>> singleton.define_method(:logged_in?) { true }
NoMethodError: private method 'define_method' called for #<Class:#<User:0x40ed90>>
        from (irb):19
        from /Users/sandal/lib/ruby19_1/bin/irb:12:in '<main>'
```

Because what we're doing is not exactly business as usual, Ruby is throwing some red flags up reminding us to make sure we know what we're doing. But because we do, we can use `send` to bypass the access controls:

```
>> singleton.send(:define_method, :logged_in?) { true }
=> #<Proc:0x3fc1f4@(irb):20 (lambda)>
>> user.logged_in?
=> true
>> User.new.logged_in?
NoMethodError: undefined method 'logged_in?' for #<User:0x3f62f4>
        from (irb):22
        from /Users/sandal/lib/ruby19_1/bin/irb:12:in '<main>'
```

Perfect! Now our dynamic approach matches our manual one, and we can proceed to building a `Stubber` module. We'll be a bit flexible and assume that any stubbed method can take any number of arguments, rather than assuming a certain amount or none at all. Beyond that, the definition is just a compilation of what we've done so far:

```
module Stubber
  extend self

  def stubs(method, options={})
    singleton(options[:for]).send(:define_method, method) do |*a|
      options[:returns]
    end
  end


  def singleton(obj)
    class << obj; self; end
  end
end
```

With this simple implementation, we're in business, doing everything we set out for in the beginning:

```
>> user = User.new
=> #<User:0x445bec>
>> Stubber.stubs(:logged_in?, :for => user, :returns => true)
=> #<Proc:0x43faa8@(irb):28 (lambda)>
>> user.logged_in?
=> true
>> User.new.logged_in?
NoMethodError: undefined method 'logged_in?' for #<User:0x439fe0>
        from (irb):40
        from /Users/sandal/lib/ruby19_1/bin/irb:12:in '<main>'
```

Beyond what we've already discussed, another important thing to remember is that the block passed to `define_method()` is a closure, which allows us to access the local variables of the enclosing scope. This is why we can pass the return value as a parameter to `Stubber.stubs()` and have it returned from our dynamically defined method.

This is a general feature of `define_method`, and is not restricted to singleton objects. Here's a quick demonstration to emphasize this point:

```
class Statistics
  def self.stat(attribute, value)
    define_method(attribute) { value }
  end

  stat :foo, :bar
  stat :baz, :quux
end

stats = Statistics.new
stats.foo #=> :bar
stats.baz #=> :quux
```

Be sure to remember this about `define_method`. It is pretty much the only clean way to dynamically define a method with a custom return value.

Returning to the core topic, we see that `Stubber`'s main trick is that it makes use of customized behaviors for individual objects. However, to do this, we need to temporarily jump into the scope of the special space reserved for this purpose in our object, so that we can pull back a reference to it. This is what the whole `class << obj; self; end` is about. Once we have this object, we can dynamically define methods on it using `define_method()` as we would in a class definition, but we need to access it via `send()` because this method is made private on singleton classes. Once we do this, we take advantage of the fact that `define_method()`'s block argument is a closure with access to the enclosing `scope`'s local variables. We set the return value this way, and complete our task of per-object stubbing.

Although this is only a single example, it demonstrates a number of key concepts about per-object behavior in Ruby:

- Using per-object behavior usually makes the most sense when you don't want to define something at the per-class level.
- Objects in Ruby may have individually customized behaviors that can replace, supplement, or amend the functionality provided by their class definitions.
- Per-object behavior (known as singleton methods), can be implemented by gaining access to the singleton class of an object using the `class << obj` notation.
- `define_method` is made private on singleton classes, so `send()` is needed to utilize it.
- When implementing nondynamic per-object behavior, the familiar `def obj.some_method` syntax may be used.

All that we've discussed so far about per-object behavior is sort of a special case of a more general topic. Ruby's open class system allows us to amend and modify the behavior of pretty much everything we can imagine, in a number of ways. This is one of the fairly unique aspects of Ruby, so there is a whole lot we can discuss here. We'll start with an anecdote and then move into some more focused details.

Extending and Modifying Preexisting Code

When I introduce Ruby to new students, my first example is often meant to shake them up a little. It is relatively unremarkable, but to those who have not worked in languages with an open class system before, it can be quite alarming:

```
class Fixnum

  def +(other)
    42
  end

end
```

The implications typically don't sink in until I fire up *irb*:

```
>> 2 + 2
=> 42
```

This demonstration is typically followed by a firm "never do this" reminder, but I continue to use it because it opens people's eyes to just how different Ruby is from most other languages. The standard response to this example is a flurry of questions about how Rubyists manage to make heads or tails of things when people can go all willy-nilly with extending classes as they see fit. That's what this section is all about.

We're going to talk about two related but somewhat distinct topics here. The first is extending Ruby classes with new behaviors by reopening classes; the second is actually modifying existing behaviors to fit new requirements. What separates the two is primarily the level of controversy, and hence the necessary level of care.

Because adding new functionality is the less dangerous of the two, we'll start with that and go over some of the specifics.

Adding New Functionality

In Chapter 1, *Driving Code Through Tests*, I mentioned and made use of a small extension to `Test::Unit` that dynamically defines test methods for us. As previously mentioned, we've borrowed this functionality from the granddaddy of Ruby extensions, ActiveSupport. We glossed over the implementation details before, but now that we're on the topic, this serves as a good example of the sorts of things you can accomplish by extending preexisting classes. We still don't need to worry about how it works; the focus is on how it extends `TestCase`:

```
module Test::Unit
  class TestCase

    def self.must(name, &block)
      test_name = "test_#{name.gsub(/\s+/,'_')}".to_sym
      defined = instance_method(test_name) rescue false
      raise "#{test_name} is already defined in #{self}" if defined
      if block_given?
        define_method(test_name, &block)
      else
        define_method(test_name) do
          flunk "No implementation provided for #{name}"
        end
      end
    end

  end
end
```

To recap, the purpose of the `must()` method is to allow you to write your test cases a bit more cleanly. Here's an example from a board game I've been working on:

```
class TestStone < Test::Unit::TestCase
  def setup
    @board = Pressman::Board.new
    @stone = Pressman::Stone.new(:black, :board    => @board,
                                         :position => [3,3] )
  end

  must "have a color" do
    assert_equal :black, @stone.color
  end

  must "have a default status of active" do
    assert_equal :active, @stone.status
  end

  must "be able to deactivate" do
    @stone.deactivate
    assert_equal :inactive, @stone.status
  end
end
```

Without this extension, you would need to write the full test method names out in the traditional way:

```
class TestStone < Test::Unit::TestCase
  def setup
    @board = Pressman::Board.new
    @stone = Pressman::Stone.new(:black, :board    => @board,
                                         :position => [3,3] )
  end

  def test_must_have_a_color
    assert_equal :black, @stone.color
  end

  def test_must_be_active_by_default
    assert_equal :active, @stone.status
  end

  def test_must_be_able_to_deactivate
    @stone.deactivate
    assert_equal :inactive, @stone.status
  end
end
```

Although this code might be a bit more conceptually simple, it is also a bit less readable and doesn't have the same shortcuts that `must()` provides. For example, if you just write a single line like this:

```
must "do something eventually"
```

The extension will create a failing test for you. For those familiar with RSpec, this is similar to the pending test functionality you'd find there. Of course, by tweaking `Test::Unit` a bit for our own needs, we can focus on adding only the functionality we're

missing, rather than jumping ship and moving to a whole other system. This is a key benefit of Ruby's open class system.

From what we've seen so far, it seems like adding functionality to a class definition is as easy as defining a new class. It uses the same syntax without any additional overhead. However, that is not to say it is without dangers.

If you can extend predefined objects for your own needs, so can everyone else, including any of the libraries you may depend on. Though we'll discuss safe techniques for combining partial definitions a little later, the technique shown here of simply opening up a class and defining a new method can result in naming clashes.

Consider two fictional units libraries, one of which converts things like `12.in` and `15.ft` into meters. We'll call this *metrics_conversions.rb*:

```
class Numeric
  def in
    self * 0.0254
  end

  def ft
    self.in * 12
  end
end
```

Our other library converts them into PDF points (1/72 of an inch). We'll call this *pdf_conversions.rb*:

```
class Numeric
  def in
    self * 72
  emd

  def ft
    self.in * 12
  end
end
```

If we load both libraries in, which one gets used? Let's ask *irb*:

```
>> require "metrics_conversions"
=> true
>> 1.in
=> 0.0254
>> require "pdf_conversions"
=> true
>> 1.in
=> 72
```

As you can see, whatever code is loaded last takes precedence. The way we have written it, the old definitions are completely clobbered and there is no easy way to recover them.

Because we'd almost never want two competing units systems loaded at the same time, it'd be better to see an error rather than a silent failure here. We can do that with the PDF conversion code and see what it looks like:

```
class Numeric

  [:in, :ft].each do |e|
    if instance_methods.include?(e)
      raise "Method '#{e}' exists, PDF Conversions will not override!"
    end
  end

  def in
    self * 72
  end

  def ft
    self.in * 12
  end

end
```

Loaded in on its own, this code runs without issue:

```
>> require "pdf_conversions"
=> true
>> 1.in
=> 72
>> 1.ft
=> 864
```

But when we revisit the original name clash problem, we have a much more explicit indicator of this issue:

```
>> require "metrics_conversions"
=> true
>> require "pdf_conversions"
RuntimeError: Method 'in' exists, PDF Conversions will not override!
        ...
```

Of course, if we do not modify *metrics_conversions.rb* as well, it will silently override *pdf_conversions.rb*. The ideal situation is for both libraries to use this technique, because then, regardless of the order in which they are required, the incompatibility between dependencies will be quickly spotted.

It is worth mentioning that it is possible for the library that is loaded first to detect the second library's attempt to override its definitions and act upon that, but this is generally considered aggressive and also results in fairly convoluted code, so it's better to address your own problems than the problems of others when it comes to extending an object's functionality.

So far, we've been talking about adding new functionality and dealing with accidental clashes. However, there are going to be times when you intentionally want to modify other code, while preserving some of its initial functionality. Ruby provides a number of ways to do that, so let's examine a few of them and weigh their risks and benefits.

Modification via Aliasing

We've used `alias_method` before for the purpose of making a new name point at an old method. This of course is where the feature gets its name: allowing you to create aliases for your methods.

But another interesting aspect of `alias_method` is that it doesn't simply create a new name for a method—it makes a copy of it. The best way to show what this means is through a trivial code example:

```
# define a method
class Foo
  def bar
    "baz"
  end
end

f = Foo.new
f.bar #=> "baz"


# Set up an alias
class Foo
  alias_method :kittens, :bar
end

f.kittens #=> "baz"

# redefine the original method
class Foo
  def bar
    "Dog"
  end
end

f.bar      #=> "Dog"
f.kittens  #=> "baz"
```

As you can see here, even when we override the original method `bar()`, the alias `kittens()` still points at the original definition. This turns out to be a tremendously useful feature.

Because I like to keep contrived examples to a minimum, we're going to take a look at a real use of this technique in code that we use every day, RubyGems.

When RubyGems is loaded, it provides access to the libraries installed through its package manager. However, we typically don't need to load these packages through some alternative interface, we just use `Kernel#require`, as we do when we're loading in our own application files. The reason this is possible is because RubyGems patches `Kernel#require` using the exact technique we've been talking about here. This is what the code looks like for *custom_require.rb*:

```
module Kernel

  ##
  # The Kernel#require from before RubyGems was loaded.

  alias_method :gem_original_require, :require

  def require(path) # :doc:
    gem_original_require path
  rescue LoadError => load_error
    if load_error.message =~ /#{Regexp.escape path}\z/ and
       spec = Gem.searcher.find(path) then
      Gem.activate(spec.name, "= #{spec.version}")
      gem_original_require path
    else
      raise load_error
    end
  end

end
```

This code first makes a copy of the original `require` method, then begins to define its enhanced one. It first tries to call the original method to see whether the requested file can be loaded that way. If it can, then it is exactly equivalent to before RubyGems was loaded, and no further processing is needed.

However, if the original require fails to find the requested library, the error it raises is rescued, and then the RubyGems code goes to work looking for a matching gem to activate and add to the loadpath. If it finds one, it then goes back to the original `require`, which will work this time around because the necessary files have been added to the path.

If the code fails to find a gem that can be loaded, the original `LoadError` is raised. So this means that in the end, it reverts to the same failure condition as the original `require` method, making it completely invisible to the user.

This is a great example of responsible modification to a preexisting method. This code does not change the signature of the original method, nor does it change the possible return values or failure states. All it does is add some new intermediate functionality that will be transparent to the user if it is not needed.

However, this concept of copying methods via `alias_method` might seem a bit foreign to some folks. It also has a bit of a limitation, in that you need to keep coming up with new aliases, as aliases are subject to collision just the same as ordinary methods are.

For example, although this code works fine:

```
class A

  def count
    "one"
  end
```

```
  alias_method :one, :count

  def count
    "#{one} two"
  end

  alias_method :one_and_two, :count

  def count
    "#{one_and_two} three"
  end

end

A.new.count #=> "one two three"
```

if we rewrote it this way, we'd blow the stack:

```
class A

  def count
    "one"
  end

  alias_method :old_count, :count

  def count
    "#{old_count} two"
  end

  alias_method :old_count, :count

  def count
    "#{old_count} three"
  end

end
```

Accidentally introducing infinite recursion by aliasing an old method twice to the same name is definitely not fun. Although this problem is rarer than you might think, it's important to know that there is a way around it.

Per-Object Modification

If we move our modifications from the per-class level to the per-object level, we end up with a pretty nice solution that gets rid of aliasing entirely, and simply leverages Ruby's ordinary method resolution path. Here's how we'd do it:

```
class A
  def count
    "one"
  end
end
```

```
module AppendTwo
  def count
    "#{super} two"
  end
end

module AppendThree
  def count
    "#{super} three"
  end
end

a = A.new
a.extend(AppendTwo)
a.extend(AppendThree)
a.count #=> "one two three"
```

Here, we have mixed two modules in an instance of `A`, each of them relying on a call to `super()`. Each method redefinition gets to use the same name, so we don't need to worry about naming clashes. Each call to `super` goes one level higher, until it reaches the top-level instance method as defined in the class.

Provided that all the code used by your application employs this approach instead of aliased method chaining, you end up with two main benefits: a pristine original class and no possibility for collisions. Because the amended functionality is included at the instance level, rather than in the class definition, you don't risk breaking other people's code as easily, either.

Note that not every single object can be meaningfully extended this way. Any objects that do not allow you to access their singleton space cannot take advantage of this technique. This mostly applies to things that are immediate values, such as numbers and symbols. But more generally, if you cannot use a call to `new()` to construct your object, chances are that you won't be able to use these tricks. In those cases, you'd need to revert to aliasing.

Even with this limitation, you can get pretty far with this approach. I don't want to end the section without one more practical example, so we'll look at a fun trick that earlier versions of Ruport did to manage a memory consumption issue in `PDF::Writer`.

Back before I maintained the Ruby PDF project, it went through a couple years of being relatively unsupported. However, when I ran into problems with it, Austin Ziegler was often willing to help me find workarounds for my own needs, even if he wasn't able to find the time to get them into a maintenance release for `PDF::Writer`.

One such fix resolved a memory consumption issue by setting up a hook for `transaction_simple` in `PDF::Writer`. I won't go into the details of how this works, but here is the module that implements it:

```
module PDFWriterMemoryPatch #:nodoc:
  unless self.class.instance_methods.include?("_post_transaction_rewind")
    def _post_transaction_rewind
      @objects.each { |e| e.instance_variable_set(:@parent,self) }
    end
  end
end
```

When people use Ruport, they use `PDF::Writer` indirectly through the object we instantiate for them. Because of this, it was easy to use the techniques described in this section. The following code should look similar to our earlier abstract examples:

```
class Ruport::Formatter::PDF

  # other implementation details omitted.

  def pdf_writer
    @pdf_writer ||= PDF::Writer.new( :paper       => paper_size || "LETTER",
                                     :orientation => paper_orientation || :portrait)
    @pdf_writer.extend(PDFWriterMemoryPatch)
  end
end
```

This code dynamically fixed an issue for Ruport users without making a global change that might conflict with other patches. Of course, it was no substitute for fixing the issue at its source, which eventually did happen, but it was a good stopgap procedure that made our users happy. When used appropriately, the power to resolve issues in other codebases whether or not you have direct access to the original code can really come in handy.

Here are the key points to remember from this section:

- All classes in Ruby are open, which means that object definitions are never finalized, and new behaviors can be added at runtime.
- To avoid clashes, conditional statements utilizing reflective features such as `instance_methods` and friends can be used to check whether a method is already defined before overwriting it.
- When intentionally modifying code, `alias_method` can be used to make a copy of the original method to fall back on.
- Whenever possible, per-object behavior is preferred. The `extend()` method comes in handy for this purpose.

So far, we've talked about extending objects others have created, as well as handling dynamic calls to objects we've created ourselves. But we can take this a step further by noticing that `Class` and `Module` are objects themselves, and as such, can be dynamically generated and molded to our needs.

Building Classes and Modules Programmatically

When I first started to get into higher-level Ruby, one of the most exciting finds was *why the lucky stiff*'s tiny web framework, Camping. This little package was packed with all sorts of wild techniques I had never seen before, including a way to write controllers to handle URL routing that just seemed out of this world:

```
module Camping::Controllers

  class Edit < R '/edit/(\d+)'
     def get(id)
       # ...
     end
  end

end
```

It didn't even occur to me that such things could be syntactically possible in Ruby, but upon seeing how it worked, it all seemed to make sense. We're not going to look at the real implementation here, but I can't resist pulling back the curtain just a little so that you can see the basic mechanics of how something like this might work.

The key secret here is that `R` is actually just a method, `Camping::Controllers::R()`. This method happens to return a class, so that means you can inherit from it. Obviously, there are a few more tricks involved, as the class you inherit from would need to track its children, but we'll get to those topics later.

For now, let's start with a simple example of how parameterized subclassing might work, and then move on to more examples of working with anonymous classes and modules in general.

First, we need a method that returns some classes. We'll call it `Mystery()`:

```
def Mystery(secret)
  if secret == "chunky bacon"
    Class.new do
      def message
        "You rule!"
      end
    end
  else
    Class.new do
      def message
        "Don't make me cry"
      end
    end
  end
end
```

Notice here that we call `Class.new()` with a block that serves as its class definition. New anonymous classes are generated on every call, which means they're basically throw-away here. That is, until we make use of them via subclassing:

```
class Win < Mystery "chunky bacon"

  def who_am_i
    "I am win!"
  end

end

class EpicFail < Mystery "smooth ham"

  def who_am_i
    "I am teh fail"
  end

end
```

Now, when we build up some instances, you can see what all of this bought us:

```
a = Win.new
a.message  #=> "You rule!"
a.who_am_i #=> "I am win!"

b = EpicFail.new
b.message  #=> "Don't make me cry"
b.who_am_i #=> "I am teh fail"
```

Even though this example doesn't really do anything useful on its own, the key concepts are still ripe for the picking. We can see that `Mystery()` conditionally chooses which class to inherit from. Furthermore, the classes generated by `Mystery()` are anonymous, meaning they don't have some constant identifier out there somewhere, and that the method is actually generating class objects, not just returning references to preexisting definitions. Finally, we can see that the subclasses behave ordinarily, in the sense that you can add custom functionality to them as needed.

When you put all of these ideas together, you might already have plans for how you could make use of this technique. Then again, it can't hurt to go over some more real-world code.

We're going to do a quick walk-through of the abstract formatting system at the heart of Ruport 2.0, called Fatty.† Despite the name, the implementation is quite slim and fairly easy to explain.

The main thing this library does is cleanly split out format-specific code, while handling parameter passing and validations. A simple example of using Fatty to just print out a greeting to someone in PDF and plain text might look like this:

```
options = { :first_name => "Chenoa", :last_name  => "Siegenthaler" }
MyReport.render_file("foo.pdf", options)
puts MyReport.render(:txt, options)
```

† Format abstraction toolkit-ty. See *http://github.com/sandal/fatty*.

We have support for a nonmagical interface, which—even without seeing the underlying implementation—shouldn't surprise anyone:

```
class MyReport < Fatty::Formatter
  module Helpers
    def full_name
      "#{params[:first_name]} #{params[:last_name]}"
    end
  end

  class Txt < Fatty::Format
    include MyReport::Helpers

     def render
      "Hello #{full_name} from plain text"
     end
  end

  # use a custom Fatty::Format subclass for extra features
  class PDF < Prawn::FattyFormat
    include MyReport::Helpers

    def render
      doc.text "Hello #{full_name} from PDF"
      doc.render
    end
  end

  formats.update(:txt => Txt, :pdf => PDF)
end
```

This looks almost entirely like ordinary Ruby subclassing and module inclusion. The only tricky thing might be the `formats()` class method, but it just points at a hash that links file extensions to the classes that handle them.

All in all, this doesn't look too bad. But it turns out that we can clean up the interface substantially if we use a bit of dynamic creativity. The following code is functionally equivalent to what you've just seen:

```
class MyReport < Fatty::Formatter
  required_params :first_name, :last_name

  helpers do
    def full_name
      "#{params[:first_name]} #{params[:last_name]}"
    end
  end

  format :txt do
    def render
      "Hello #{full_name} from plain text"
    end
  end
```

```
  format :pdf, :base => Prawn::FattyFormat do
    def render
      doc.text "Hello #{full_name} from PDF"
      doc.render
    end
  end

end
```

Our class definitions have been transformed into a domain-specific interface for format abstraction. Aside from having nicer names for things and a more pleasant syntax, we have gained some side benefits. We no longer need to manually map file extensions to class names: the `format()` method handles that for us. We also don't need to manually include our helper module; that is taken care of as well. As these are the two things that fit well with this topic, let's take a look at how both of them are handled.

First, we'll take a look at the `format()` helper, which is a simple one-liner:

```
def format(name, options={}, &block)
  formats[name] = Class.new(options[:base] || Fatty::Format, &block)
end
```

When this class method is called with just a block, it generates an anonymous subclass of `Fatty::Format`, and then stores this subclass keyed by extension name in the `formats` hash. In most cases, this is enough to do the trick. However, sometimes you will want to inherit from a base class that implements some additional helpers, as we did with `Prawn::FattyFormat`. This is why `options[:base]` is checked first. This one line of code with its two possible cases covers how the class definitions are created and stored.

We can now turn our eyes to the `helpers()` method, which is another one-liner. This one has two possible uses as well, but we showed only one in our example:

```
def helpers(helper_module=nil, &block)
  @helpers = helper_module || Module.new(&block)
end
```

As you can see here, modules can also be built up anonymously using a block. This code gives the user a choice between doing that or providing the name of a module like this:

```
class MyReport < Fatty::Formatter

  helpers MyHelperModule

  #...
end
```

Of course, the more interesting case is when you use the block form, but only marginally so. The real work happens in `render()`:

```
def render(format, params={})
  validate(format, params)

  format_obj = formats[format].new  ❶
  format_obj.extend(@helpers) if @helpers ❷
  format_obj.params = params
  format_obj.validate
  format_obj.render
end
```

I've marked the two lines we're interested in:

❶ This line uses the `formats` hash to look up our anonymous class by extension name.

❷ This line mixes in our helper module, whether it's a reference to an explicitly defined module or, as in our example, a block that defines the anonymous module's body.

The rest of the method is fairly self-explanatory but inconsequential. It was the dynamic class/module creation we were interested in—the rest is just mundane detail particular to Fatty's implementation.

With just a few lines of code, we've been able to show just how powerful Ruby is when you programmatically generate higher-level objects such as classes and modules. In this particular example, we've used this technique for interface cleanup and the associated organizational benefits. You may find a lot of other uses, or none at all, depending on your work.

Like with many other concepts in this chapter, we've truly been cooking with gas here. Let's go over a few tips to help you avoid getting burned, then hit one more short topic before finishing up:

- Classes and modules can be instantiated like any other object. Both constructors accept a block that can be used to define methods as needed.
- To construct an anonymous subclass, call `Class.new(MySuperClass)`.
- Parameterized subclassing can be used to add logic to the subclassing process, and essentially involves a method returning a class object, either anonymous or explicitly defined.

Registering Hooks and Callbacks

In the very beginning of the chapter, we looked at the `BlankSlate` class, and I had mentioned that there was some additional work that needed to be done to deal with things such as new method definitions on `Object` or module inclusion.

To recap the situation, `BlankSlate` is supposed to be a "naked" base class we can inherit from that doesn't reveal any of its methods until we tell it to. We have already covered the ins and outs of how `BlankSlate` hides its initial instance methods and how it can selectively reveal them. The problem that remains to be solved is how to accommodate for changes that happen at runtime.

Detecting Newly Added Functionality

As we've seen, due to the open class system, Ruby object definitions are never really finalized. As a consequence, if you add a method to `Object`, it becomes available immediately to every object in the system except instances of `BasicObject`. To put words into code, here's what that means:

```
>> a  = "foo"
=> "foo"
>> b = [1,2,3]
=> [1, 2, 3]
>> class C; end
=> nil
>> c = C.new
=> #<C:0x42a400>
>> class Object
>>   def party
>>     "wooohoo!"
>>   end
>> end
=> nil
>> a.party
=> "wooohoo!"
>> b.party
=> "wooohoo!"
>> c.party
=> "wooohoo!"
```

Now everyone is partying, except `BlankSlate`. Or more accurately, `BlankSlate` is being forced to party when it doesn't want to. The solution is to set up a hook that watches for newly defined methods and hides them:

```
class Object
  class << self
    alias_method :blank_slate_method_added, :method_added

    # Detect method additions to Object and remove them in the
    # BlankSlate class.
    def method_added(name)
      result = blank_slate_method_added(name)
      return result if self != Object
      BlankSlate.hide(name)
      result
    end

  end
end
```

This code uses techniques discussed earlier in this chapter to modify the behavior of a core Ruby function, `Object.method_added()`, while remaining transparent for the ordinary use cases. Classes inherited from `Object` will not affect `BlankSlate`, so this code is set to short-circuit in those cases. However, if `self` happens to be `Object`, the code tells

`BlankSlate` to hide it and then returns the results of the original `method_added()` function that has been aliased here.

You'd think that would do the trick, but as it turns out, `Object` includes the module `Kernel`. This means we need to track changes over there too, using nearly the same approach:

```
module Kernel
  class << self
    alias_method :blank_slate_method_added, :method_added

    # Detect method additions to Kernel and remove them in the
    # BlankSlate class.
    def method_added(name)
      result = blank_slate_method_added(name)
      return result if self != Kernel
      BlankSlate.hide(name)
      result
    end
  end
end
```

There isn't much new here, so it's safe to say that if you understood how this worked on `Object`, you can assume this is just more of the same stuff. However, it does give a hint about another problem: inclusion of modules into `Object` at runtime.

First, another quick illustration of the issue:

```
>> module A
>>   def foo
>>     "bar"
>>   end
>> end
=> nil
>> a = Object.new
=> #<Object:0x428ca4>
>> a.extend(A)
=> #<Object:0x428ca4>
>> a.foo
=> "bar"
>> module A
>>   def kittens
>>     "meow"
>>   end
>> end
=> nil
>> a.kittens
=> "meow"
```

Every module included in an object is like a back door for future expansion. When you first fire up Ruby, the only module you need to worry about is `Kernel`, but after that, all bets are off. So we end up jumping up one level higher to take care of module inclusion dynamically:

```
class Module
  alias blankslate_original_append_features append_features
  def append_features(mod)
    result = blankslate_original_append_features(mod)
    return result if mod != Object
    instance_methods.each do |name|
      BlankSlate.hide(name)
    end
    result
  end
end
```

In this example, `mod` is the class that was modified by a module inclusion. As in the other hooks, `BlankSlate` makes an alias of the original, calls it, and simply returns its result if the modified object isn't `Object` itself. In the case where a module is mixed into `Object`, `BlankSlate` needs to wipe out the instance methods added to its own class definition. After this, it returns the result of the original `append_features()` call.

This pretty much describes the key aspects of capturing newly added functionality at the top level. You can of course apply these hooks to individual classes lower in the chain and make use of them in other ways.

Tracking Inheritance

When you write unit tests via `Test::Unit`, you typically just subclass `Test::Unit::TestCase`, which figures out how to find your tests for you. Though we won't look at the details for how that is actually implemented, we can take a naive shot at it on our own using the `Class#inherited` hook.

We're going to implement the code to make this example functional:

```
class SimpleTest < SimpleTestHarness

  def setup
    puts "Setting up @string"
    @string = "Foo"
  end

  def test_string_must_be_foo
    answer = (@string == "Foo" ? "yes" : "no")
    puts "@string == 'Foo': " << answer
  end

  def test_string_must_be_bar
    answer = (@string == "bar" ? "yes" : "no")
    puts "@string == 'bar': " << answer
  end

end

class AnotherTest < SimpleTestHarness
```

```
  def test_another_lame_example
    puts "This got called, isn't that good enough?"
  end

  def helper_method
    puts "But you'll never see this"
  end

  def a_test_method
    puts "Or this"
  end

end

SimpleTestHarness.run
```

We must first identify each subclass as a test case, and store it in an array until `SimpleTestHarness.run` is called. Like `Test::Unit` and other common Ruby testing frameworks, we'll wipe the slate clean by reinstantiating our tests for each test method, running a `setup` method if it exists. We will follow the `Test::Unit` convention and run only the methods whose names begin with `test_`. We haven't implemented any assertions or anything like that, because it's not really the point of this exercise.

The task can easily be broken down into two parts: detecting the subclasses, and later manipulating them. The first part is where we use the `inherited` hook, as you can see:

```
class SimpleTestHarness

  class << self

    def inherited(base)
      tests << base
    end

    def tests
      @tests ||= []
    end

  end

end
```

Surprisingly enough, that was relatively painless. Each time a new subclass is derived from `SimpleTestHarness`, the `inherited()` hook is called, passing in the subclass as `base`. If we just store these in an array at class level, that's all we need for writing a test runner. Adding in `SimpleTestHarness.run`, our full class looks like this:

```
class SimpleTestHarness
  class << self

    def inherited(base)
      tests << base
    end
```

```
    def tests
      @tests ||= []
    end

    def run
      tests.each do |t|
        t.instance_methods.grep(/^test_/).each do |m|
          test_case = t.new
          test_case.setup if test_case.respond_to?(:setup)
          test_case.send(m)
        end
      end
    end
  end

end
```

This code walks over each class in the `tests` array, and then filters out the names of the instance methods that begin with `test_`. For each of these methods, it creates a new instance of the test case, calls `setup` if it exists, and then uses `send` to dynamically invoke the individual test. With this class definition in place, the original set of tests for which we were trying to implement this functionality can actually run, resulting in the following output:

```
Setting up @string
@string == 'Foo': yes
Setting up @string
@string == 'bar': no
This got called, isn't that good enough?
```

Pretty cool, huh? These hooks essentially provide an event system, giving you a way to handle changes to Ruby in a dynamic way. If you've ever had to do GUI programming or anything else that involved dynamic callbacks, you already grasp the core ideas behind this concept. The only difference is that rather than capturing a button press, you're capturing an inheritance event or an added method. When used appropriately, this can be a very powerful technique.

We're about to wrap things up here, but before we do, it's worth showing the equivalent of what we just did, but for modules. There happens to be a fairly standard Ruby idiom that takes advantage of that hook, so it's one you shouldn't skip over.

Tracking Mixins

You probably already know that if you use `include` to mix a module into a class, the methods become available at the instance level, and that if you use `extend`, they become available at the class level. However, an issue comes up when you want to provide both class and instance methods from a single module.

A naive workaround might look like this:

```
module MyFeatures

  module ClassMethods
    def say_hello
      "Hello"
    end

    def say_goodbye
      "Goodbye"
    end
  end

  def say_hello
    "Hello from #{self}!"
  end

  def say_goodbye
    "Goodbye from #{self}"
  end
end


class A
  include MyFeatures
  extend MyFeatures::ClassMethods
end
```

If we test this out in *irb*, we see that it does work:

```
?> A.say_hello
=> "Hello"
>> obj = A.new
=> #<A:0x1ee628>
>> obj.say_hello
=> "Hello from #<A:0x1ee628>!"
>> obj.say_goodbye
=> "Goodbye from #<A:0x1ee628>"
>> A.say_goodbye
=> "Goodbye"
```

Having to manually do the `extend` call seems a bit ugly, though. It's not terrible when we are writing it ourselves, but it would be a little weird to do this any time you used a third-party module. Of course, that's where a nice little hook comes in handy. The following code is functionally equivalent to our previous example:

```
module MyFeatures

  module ClassMethods
    def say_hello
      "Hello"
    end

    def say_goodbye
      "Goodbye"
    end
  end
```

```
  def self.included(base)
    base.extend(ClassMethods)
  end

  def say_hello
    "Hello from #{self}!"
  end

  def say_goodbye
    "Goodbye from #{self}"
  end

end # MyFeatures

class A
  include MyFeatures
end
```

Here, we were able to get rid of the manual `extend` call and automate it through the `included` hook. This hook gets called every time the module is included into a class, and passes the class object as the `base` object. From here, we simply call `extend` as before; it is just now wrapped up in the hook rather than manually specified in the class definition. Although this may seem like a small change, having a single entry point to the module's features is a major win, as it keeps implementation details off the mind as much as possible when simply including the module.

Although we could dig up more and more hooks provided by Ruby, we've already covered most of the ones that are used fairly often. There are certainly plenty that we didn't cover, and you might want to read over the core Ruby documentation a bit to discover some of the more obscure ones if you're either curious or have an uncommon need.

For the hooks we did cover, here are some things to remember:

- If you are making changes to any hooks at the top level, be sure to safely modify them via aliasing, so as not to globally break their behavior.
- Hooks can be implemented on a particular class or module, and will catch everything below them.
- Most hooks either capture a class, a module, or a name of a method and are executed after an event takes place. This means that it's not really possible to intercept an event before it happens, but it is usually possible to undo one once it is.

And with that, we can wrap up this intense chapter with some closing notes and a final challenge to the adventurous.

Conclusions

I found myself getting physically tired writing this chapter. If you feel that way after reading it, I don't really blame you. People will tell you again and again that this sort of coding is extremely hard or fueled by some sort of magic. Others will tell you it's the bee's knees and that you should use it all the time, everywhere, whenever you can. Neither statement is true.

The truth of the matter is that taken individually, each of Ruby's dynamic features is relatively straightforward, and can be a valuable tool if used properly. But looking at all of this stuff and trying to use it as much as possible in your code would be absolutely overwhelming.

My general rule of thumb is to ignore all of these advanced Ruby features until my code illustrates a need for them. If I write several method calls that appear to do almost the same thing with a different name, I might be able to leverage `method_missing`. If I want to endow certain objects with some handy shortcuts, but leave the option of instantiating a simple, unadorned core object, I might look into mixing in some singleton methods using `extend`. By the end of the day, in a large or complicated application, I may end up using a large subset of the techniques discussed here. But if I started out by thinking about what dynamic features my code needed rather than what requirements it must satisfy, development would come to a confusing, grinding halt.

So here's my advice about making use of the information in this chapter: just make a mental note of what you've learned here, and then wait until some code jumps out at you and seems to be begging to be cleaned up using one of the techniques shown here. If it works out well, you've probably made a good decision. If it seems like more trouble than it's worth, bail out and wait for the next bit of code to alert you again. Keep repeating this process and you'll find a good balance for how dynamic your code really needs to be.

Because this chapter is focused on a series of topics that are sort of a rite of passage as far as Ruby development goes, I'd like to end with a bit of code that might challenge your understanding a bit.

What follows is a simplistic approximation of Camping's routing magic. It is meant to help you learn, but is left without comments so that you can figure it out on your own. It does not introduce any new concepts beyond what was discussed in this chapter, so if you can figure out how it works, you can be sure that you have a fairly solid grasp of what we've been talking about here.

Enjoy!

```
module NaiveCampingRoutes

  extend self

  def R(url)
    route_lookup = routes

    klass = Class.new
    meta  = class << klass; self; end
    meta.send(:define_method, :inherited) do |base|
      raise "Already defined" if route_lookup[url]
      route_lookup[url] = base
    end
    klass
  end

  def routes
    @routes ||= {}
  end

  def process(url, params={})
    routes[url].new.get(params)
  end
end

module NaiveCampingRoutes
  class Hello < R '/hello'
    def get(params)
      puts "hello #{params[:name]}"
    end
  end

  class Goodbye < R '/goodbye'
    def get(params)
      puts "goodbye #{params[:name]}"
    end
  end
end

NaiveCampingRoutes.process('/hello', :name => "greg")
NaiveCampingRoutes.process('/goodbye', :name => "joe")
```