

15

Callbacks, hooks, and runtime introspection

In this chapter:

- Runtime callbacks: inherited, included, etc.
- `respond_to?` and `method_missing`
- Introspection of object and class method lists
- Trapping unresolved constant references
- Examining in-scope variables and constants
- Parsing caller and stack-trace information

In keeping with its dynamic nature, and its encouragement of flexible, supple object and program design, Ruby provides you with a large number of ways to examine what's going on while your program is running, and to set up event-based callbacks and hooks—essentially, “tripwires” that get pulled at specified times and for specific reasons—in the form of methods with special, reserved names that you can, if you wish, provide definitions for. Thus you can rig a module so that a particular method gets called every time a class includes that module. Or write a callback method for a class that gets called every time the class is inherited. And several more.

In addition to runtime callbacks, Ruby lets you perform more passive but often critical acts of examination: you can ask objects what methods they can execute, or what instance variables they have. You can query classes and modules for their constants and their instance methods. You can examine a stack trace, to determine what method calls got you to

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

a particular point in your program—and you even get access to the filenames and line numbers of all the method calls along the way.

In short, Ruby invites you to the party: you get to see what's going on, in considerable detail, via techniques for runtime introspection; and you can order Ruby to push certain buttons in reaction to runtime events. This chapter, the last in the book, will explore a variety of these introspective and callback techniques, and will equip you to take ever greater advantage of the facilities offered by this remarkable, and remarkably dynamic, language.

15.1 Callbacks and hooks

The use of *callbacks* and *hooks* is a fairly common meta-programming technique. These methods are called when a particular event takes place during the run of a Ruby program. An event is something like

- a nonexistent method being called on an object
- a module being mixed in to a class or another module
- an object being extended with a module
- a class being subclassed (inherited from)
- an instance method being added to a class
- a singleton method being added to an object
- a reference being made to a non-existent constant

For every event in that list, you can (if you choose) write a callback method that will be executed when the event happens. These callback methods are per-object or per-class, not global; if you want a method called when the class `Ticket` gets subclassed, you have to write the appropriate method specifically for class `Ticket`.

What follows are descriptions of each of these runtime event hooks. We'll look at them in the order they're listed above.

15.1.1 Intercepting unrecognized messages with `method_missing`

When you send a message to an object, the object executes the first method it finds on its method lookup path with the same name as the message. If it fails to find any such method, it raises a `NoMethodError` exception—*unless* you have provided the object with a method called `method_missing`.

`method_missing` is in part a safety net: it gives you a way to intercept unanswerable messages and handle them gracefully:

```
class C
  def method_missing(m)
    puts "There's no method called #{m} here -- please try again."
  end
end
```

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

```
C.new.anything
```

You can also use `method_missing` to bring about an automatic extension of the way your object behaves. For example, let's say you're modeling an object that in some respects is a container but also has other characteristics—perhaps, just for the sake of variety, a cookbook. You want to be able to program your cookbook as a collection of recipes, but it also has certain characteristics (title, author, perhaps a list of people with whom you've shared it or who have contributed to it) that need to be stored and handled separately from the recipes. Thus the cookbook is both a collection and the repository of metadata about the collection.

One way to do this would be to maintain an array of recipes and then forward any unrecognized messages to that array. A simple implementation might look like this:

```
class Cookbook
  attr_accessor :title, :author

  def initialize
    @recipes = []
  end

  def method_missing(m, *args, &block)
    @recipes.send(m, *args, &block)
  end
end
```

Now we can perform manipulations on the collection of recipes, taking advantage of any array methods we wish. (Let's assume there's a `Recipe` class, separate from the `Cookbook` class, and we've already created some recipe objects.)

```
cb = Cookbook.new
cb << recipe_for_cake
cb << recipe_for_chicken
beef_dishes = cb.find_all {|recipes| recipe.main_ingredient ==
  "beef" }
```

The cookbook instance, `cb`, doesn't have methods called `<<` and `find_all`, so those messages are passed along to the `@recipes` array, courtesy of `method_missing`. We can still define any methods we want directly in the `Cookbook` class—we can even override array methods, if we want a more cookbook-specific behavior for any of those methods—but `method_missing` has saved us from having to define a whole parallel set of methods for handling pages as an ordered collection.

TIP: RUBY HAS LOTS OF METHOD-DELEGATING TECHNIQUES

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

In this `method_missing` example, we've *delegated* the processing of messages (the unknown ones) to the array `@pages`. Ruby has several mechanisms for delegating actions from one object to another. We won't go into them here, but you may come across both the `Delegator` class and the `SimpleDelegator` class in your further encounters with Ruby.

While `method_missing` is a useful event-trapping tool—perhaps the most widely used among all the standard Ruby hooks and callbacks—it's far from the only one, though.

15.1.2 Trapping include operations with `Module#include`

When a module is included (mixed in) to a class, *if* a method called `included` is defined for that module, then that method is called. The method receives the name of the class as its single argument.

You can do a quick test of `included` by having it trigger a message printout and then perform an include operation:

```
module M
  def self.included(c)
    puts "I have just been mixed into #{c}."
  end
end

class C
  include M
end
```

You'll see the message "I have just been mixed into C." printed out as a result of the execution of `M.included` when `M` gets included by (mixed into) `C`. (Because you can also mix modules into modules, the example would also work if `C` were another module.)

When would it be useful for a module to intercept its own inclusion like this? One commonly discussed case revolves around the difference between instance and class methods. When you mix a module into a class, you're ensuring that all the *instance methods* defined in the module become available to instances of the class. But the class object isn't affected. The following question often arises: What if you want to add *class methods* to the class by mixing in the module along with adding the instance methods?

Courtesy of `included`, you can trap the include operation and use the occasion to add class methods to the class that's doing the including. Listing 15.1 shows an example.

Listing 15.1 Using the `included` callback to add a class method as part of a mix-in operation

```
module M
  def self.included(c1)
```

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

```

    def cl.a_class_method
      puts "Now the class has a new class method."
    end
  end

  def an_inst_method
    puts "This module supplies this instance method."
  end
end

class C
  include M
end

c = C.new
c.an_inst_method
C.a_class_method

```

The output from the Listing 15.1 is as follows:

```

This module supplies this instance method.
Now the class has a new class method.

```

When class C included module M, two things happened. First, an *instance* method called `an_inst_method` appeared in the lookup path of its instances (such as `c`). Second, thanks to M's included callback, a *class* method called `a_class_method` was defined for the class object C.

`Module#include` is a useful way to hook into the class/module engineering of your program. Meanwhile, let's look at another callback in the same general area of interest: `Module#extended`.

15.1.3 Intercepting extend

As you know from Chapter 13, extending individual objects with modules is one of the most powerful techniques available to you in Ruby for taking advantage of the flexibility of objects and their ability to be customized. It's also the beneficiary of a runtime hook: using the `Module#extended` method, you can set up a callback that will be triggered whenever an object performs an `extend` operation that involves the module in question.

Listing 15.2 shows a modified version of Listing 15.1. The modified version illustrates the workings of `Module#extended`.

Listing 15.2 Triggering a callback from an extend event

```

module M
  def self.extended(obj)
    puts "Module #{self} is being used by #{obj}."
  end

  def an_inst_method
    puts "This module supplies this instance method."
  end
end

```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

```

end

my_object = Object.new
my_object.extend(M)
my_object.an_inst_method

```

The output from Listing 15.2 is:

```

Module M is being used by #<Object:0x28ff0>.
This module supplies this instance method.

```

(Of course, the hexadecimal number in the object's string representation will vary between machines.)

It's useful to look at how the `included` and `extended` callbacks work in conjunction with singleton classes. There's nothing too surprising here; in fact, what you learn is how consistent Ruby's object and class model is.

SINGLETON CLASS BEHAVIOR WITH EXTENDED AND INCLUDED

In effect, *extending* an object with a module is the same as *including* that module in the object's singleton class. Whichever way you describe it, the upshot is that module gets added to the object's method lookup path, entering the chain right after the object's singleton class.

However, the two operations trigger different callbacks--namely, `extended` and `included`. Listing 15.3 demonstrates the relevant behaviors.

Listing 15.3 Extending an object and including into its singleton class trigger different callbacks

```

module M
  def self.included(c) #1
    puts "#{self} included by #{c}."
  end

  def self.extended(obj) #2
    puts "#{self} extended by #{obj}."
  end
end

obj = Object.new
puts "Including M in object's singleton class:"
class << obj #3
  include M
end

puts

obj = Object.new
puts "Extending object with M:" #4
obj.extend(M)

```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

Both callbacks are defined in the little module M: `included #1` and `extended #2`. Each callback prints out a report of what it's doing. Starting with a freshly-minted, generic object, we include M in the object's singleton class, `#3` and then repeat the process, using another new object and extending the object with M directly. `#4`

The output from List 15.3 is as follows:

```
Including M in object's singleton class:
M included by #<Class:#<Object:0x1c898c>>.
```

```
Extending object with M:
M extended by #<Object:0x1c889c>.
```

Sure enough, the `include` triggers the `included` callback, and the `extend` triggers `extended`, even though in this particular scenario the results of the two operations are the same: the object in question has M added to its method lookup path. It's a nice illustration of some of the subtlety and precision of Ruby's architecture, and also a useful reminder that manipulating an object's singleton class directly isn't *quite* identical to doing singleton-level operations directly on the object.

As modules can intercept `include` and `extend` operations, so too can classes tell when they're being subclassed.

15.1.4 Intercepting inheritance with `Class#inherited`

You can hook into the subclassing of a class by defining a special class method called `inherited` for that class. If `inherited` has been defined for a given class, then when you subclass the class, `inherited` is called with the name of the new class as its single argument:

Here's a simple example, where the class C simply reports on the fact that it has been subclassed.

```
class C
  def self.inherited(subclass)
    puts "#{self} just got subclassed by #{subclass}."
  end
end

class D < C
end
```

The subclassing of C by D automatically triggers a call to `inherited` and therefore produces the following output:

```
C just got subclassed by D.
```

`inherited` is a class method, so descendants of the class that defines it are also able to call it. The actions you define in `inherited` cascade: If you inherit from a subclass, that

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

subclass triggers the `inherited` method, and similarly down the chain of inheritance. If you do this

```
class E < D
end
```

you're informed that `D` just got subclassed by `E`. You get similar results if you subclass `E`, and so forth.

The limits of the inherited callback

Everything has its limits, including the inherited callback. When `D` inherits from `C`, `C` is `D`'s superclass but, in addition, `C`'s singleton class is the superclass of `D`'s singleton class. That's how `D` manages to be able to call `C`'s class methods. But there's no callback triggered. Even if you define `inherited` in `C`'s singleton class, it never gets called.

Here's a little testbed. Note how `inherited` is defined inside the singleton class of `C` itself. But even when `D` inherits from `C`--and even after the explicit creation of `D`'s singleton class--the callback isn't triggered.

This and the next code line are part of this sidebar.

```
class C
  class << self
    def self.inherited
      puts "Singleton class of C just got inherited!"
      puts "But you'll never see this message."
    end
  end
end

class D < C
  class << self
    puts "D's singleton class now exists, but no callback!"
  end
end
```

The output from this little program is just:

```
D's singleton class now exists, but no callback!
```

You are extremely unlikely ever to come across a situation where this behavior matters, but it gives you a nice X-ray of how Ruby's class model interoperates with its callback layer.

Let's look now at intercepting events having to do with constants: their absence, and their addition to a class or module.

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

15.1.5 Module#const_missing and #const_added

Module#const_missing is another commonly used callback. As the name implies, this method is called whenever an unidentifiable constant is referred to inside a given module or class:

```
class C
  def self.const_missing(const)
    puts "#{const} is undefined—setting it to 1."
    const_set(const,1)
  end
end

puts C::A
puts C::A
```

The output of this code is as follows:

```
A is undefined—setting it to 1.
1
1
```

Thanks to the callback, C::A is defined automatically when you use it without defining it. This is taken care of in such a way that puts can print the value of the constant; puts never has to know that the constant wasn't defined in the first place. Then, on the second call to puts, the constant is already defined, and const_missing isn't called.

One of the most powerful event callback facilities in Ruby is method_added, which lets you trigger an event when a new instance method is defined.

15.1.6 method_added and singleton_method_added

If you define method_added as a class method in any class or module, it will be called when any instance method is defined. Here's a very basic example.

```
class C
  def self.method_added(m) #A
    puts "Method #{m} was just defined."
  end

  def a_new_method #B
  end
end
#A Define the callback
#B Trigger it by defining an instance method
```

The output from this little program is:

```
Method a_new_method was just defined.
```

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

The `singleton_method_added` callback does much the same thing, but for singleton methods. In fact, perhaps surprisingly, it even triggers itself. If you run this very small snippet:

```
class C
  def self.singleton_method_added(m)
    puts "Method #{m} was just defined."
  end
end
```

you'll see that the callback--which is, itself, a singleton method on the class object `C`--triggers its own execution:

```
Method singleton_method_added was just defined.
```

The callback will also be triggered by the definition of another singleton (class) method. If you expand the previous example to include such a definition:

```
class C
  def self.singleton_method_added(m)
    puts "Method #{m} was just defined."
  end

  def self.new_class_method
    end
end
```

the new output will be:

```
Method singleton_method_added was just defined.
Method new_class_method was just defined.
```

In most cases, you'll want to use `singleton_method_added` with objects other than class objects. Here's how its use might play out with a generic object.

```
obj = Object.new

def obj.singleton_method_added(m)
  puts "Singleton method #{m} was just defined."
end

def obj.a_new_singleton_method
end
```

The output in this case is:

```
Singleton method singleton_method_added was just defined.
Singleton method a_new_singleton_method was just defined.
```

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

Again, you get the somewhat surprising effect that the defining of `singleton_method_added` triggers the callback's own execution.

Putting the class-based and object-based approaches together, you can of course achieve the object-specific effect by defining the relevant methods in the object's singleton class.

```
obj = Object.new

class << obj
  def singleton_method_added(m)
    puts "Singleton method #{m} was just defined."
  end

  def a_new_singleton_method
  end
end
```

The output for this snippet will be exactly the same as for the previous example. Finally, and coming full circle, you can define `singleton_method_added` as a regular instance method of a class, in which case every instance of that class will follow the rule that the callback will be triggered by the creation of a singleton method.

```
class C
  def singleton_method_added(m)           #1
    puts "Singleton method #{m} was just defined."
  end
end

c = C.new

def c.a_singleton_method                 #2
end
```

Here, the definition of the callback #1 governs every instance of C. The definition of a singleton method on such an instance #2 therefore triggers the callback, resulting in the output:

```
Singleton method a_singleton_method was just defined.
```

It's very possible that you won't use either `method_added` or `singleton_method_added` very often in your Ruby applications. But they provide a functionality for which there's no substitute; and experimenting with them is a great way to get a deeper feel for how the various parts of the class, instance, and singleton class picture fit together.

We'll turn now to the subject of examining object capabilities, a topic that you've glimpsed (in some examples of the `methods` method) but that we haven't yet gone into in full depth.

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

15.2 Examining object capabilities

Ruby objects are defined and recognized by what they can do: what messages they respond to or, to put essentially the same thing a different way, what methods they can execute. Ruby provides an extensive toolset for examining objects' capabilities. The toolset consists of a family of methods whose names include the word `methods`. There are several such methods, in keeping with the fact that methods in Ruby can fall into any of several categories: public, private; instance, class.

We'll look at the "methods" methods starting with the simplest: `methods`. From there we'll proceed to the various elaborations and permutations involving method visibility and the instance/class distinction.

15.2.1 Listing an object's non-private methods

To list the non-private methods that an object knows about, you use the method `methods`. "Non-private" means public or protected. In its simplest form, `methods` provides an array of symbols--possibly a very large array, depending on which object you're querying. Typically you'll filter the array in some way so as to get a useful subset of methods.

Here, for example, is how you might ask a string object what methods it knows about that involve modification of case:

```
>> string = "Test string"
=> "Test string"
>> string.methods.grep(/case/).sort
=> [:casecmp, :downcase, :downcase!, :swapcase, :swapcase!, :upcase,
:upcase!]
```

The `grep` filters out any symbol that doesn't have "case" in it. (Remember that though they're not strings, symbols exhibit a number of string-like behaviors, such as being greppable.) The `sort` command at the end is useful for most method-listing operations. It doesn't make much of a difference in this example, since there are only seven methods; but when you get back arrays of a hundred or more symbols, sorting them can help a lot.

Some of the "case" methods are also "bang" (!) methods. You can easily find out what bang methods an object has, through a similar `grep` operation.

```
>> string.methods.grep(/!/).sort
=> [!:, !=, !~, :capitalize!, :chomp!, :chop!, :delete!, :downcase!,
:encode!, :gsub!, :lstrip!, :next!, :reverse!, :rstrip!, :slice!,
:squeeze!,
:strip!, :sub!, :succ!, :swapcase!, :tr!, :tr_s!, :upcase!]
```

You can even use `methods` to answer more complex questions than the question of what methods an object has. Do strings have any bang methods that do not have corresponding non-bang methods?

```
string = "Test string"
methods = string.methods
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

```

bangs = string.methods.grep(/!/) #1
unmatched = bangs.reject {|b| methods.include?(b.chomp("!")) } #2
if unmatched.empty? #3
  puts "All bang methods are matched by non-bang methods."
else
  puts "Some bang methods have no non-bang partner: "
  puts unmatched
end

```

The code works by collecting all of a string's public methods and, separately, all of its bang methods. #1 Then a `reject` operation filters out all bang method names for which a corresponding non-bang name can be found in the larger method-name list. #2 If the filtered list is empty, that means that no unmatched bang method names were found. If it isn't empty, then at least one such name was found, and can be printed out. #3

If you run the script as it is, it will always take the first (true) branch of the if statement. If you want to see a list of unmatched bang methods, you can add a line to the program, just after the first line:

```

def string.surprise!; end #A
#A Add as line two of program

```

When you run the modified version of the script, you'll see this:

```

Some bang methods have no non-bang partner:
surprise!

```

As we've already seen, writing bang methods without non-bang partners is bad practice--but it's a good way to see the `methods` method at work.

You can of course ask class and module objects what their methods are. After all, they're just objects. Remember, though, that the `methods` method always lists the public methods of the object itself. In the case of classes and modules, that means you're *not* getting a list of the methods that instances of the class, or instances of classes that mix in the module, can call. You're getting the methods that the class or module itself knows about. Here's a (partial) result from calling `methods` on a newly-created class object.

```

>> class C; end
=> nil
>> C.methods.sort
=> [!~, !=, !~, :<, :<=, :<=>, ==, ===, =~, :>, :>=,
: __id__, : __send__, :allocate, :ancestors, :autoload, :autoload?, :class,
:class_eval, :class_exec, :class_variable_defined?, :class_variable_get,
:class_variable_set, :class_variables, etc.

```

Class and module objects will share some methods with their own instances, since they're all objects and objects in general share certain methods. But the methods you see are those

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

that the class or module itself can call. You can also ask classes and modules about the instance methods they define. We'll return to that technique shortly. First, let's look at how to list an object's private and protected methods.

15.2.2 Listing private and protected methods

Every object (except instances of `BasicObject`) has a `private_methods` method and a `protected_methods` method. They work as you would expect; they provide arrays of symbols, but containing private and protected method names, respectively.

Freshly-minted Ruby objects have a lot of private methods, and no protected methods.

```
>> object = Object.new
=> #<Object:0x3a1a24>
>> object.private_methods.size
=> 66
>> object.protected_methods.size
=> 0
```

What are those private methods? They're the top-level methods, defined as private instance methods of the `Kernel` module. Naturally, if you define a private method yourself, it will also appear in the list. Here's an example: a simple `Person` class in which assigning a name to the person, via the `name=` method, triggers a name-normalization method which removes everything other than letters and selected punctuation characters from the name. The `normalize_name` method is private.

```
class Person
  attr_reader :name
  def name=(name)           #A
    @name = name
    normalize_name         #B
  end

  private
  def normalize_name
    name.gsub!(/[^\-a-z'\.\s]/i, "") #C
  end
end

david = Person.new
david.name = "123David!! Bl%a9ck"
raise "Problem" unless david.name == "David Black" #D
puts "Name has been normalized."

p david.private_methods.sort.grep(/normal/)        #E

#A Define non-default write accessor
#B Normalize the name when it's assigned
#C Remove all undesired characters from the name
#D Quick spot-check to make sure normalization works
#E Result of private method inspection: [ :normalize_name ]
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

Protected methods can be examined in much the same way, using the `protected_methods` method.

Method queries, `method_missing`, and `respond_to?`

As you know, you can ask an object whether or not it knows about a particular method with the `respond_to?` method. You can also intercept unknown messages, and take action based on them, using `method_missing`. And you can ask an object to list all of its public methods.

It's good to know how these three techniques interact. `respond_to?` only pertains to public methods. Given a private method called `my_private_method` on an object `x`, `x.respond_to?(:my_private_method)` will return `false`. That's the case even if you run `respond_to?` inside another instance method of `x`. In other words, `x` will never tell you that it responds to `:my_private_method`, even when it does.

With `method_missing` you can arrange for an object to provide a response when sent a message for which it has no corresponding method. However, `respond_to?` won't know about such messages, and will tell you that the object does not respond to the message even though you get a useful response when you send the message to the object. Some Rubyists like to override `respond_to?` so that it incorporates the same logic as `method_missing`, for a given class. That way the results of `respond_to?` correspond more closely to the specifics of what messages an object can and cannot make sense of.

Others prefer to leave `respond_to?` as it stands, on the grounds that it's a way to check whether or not an object *already* has the ability to respond to a particular message, without the intervention of `method_missing`. On that interpretation, `respond_to?` corresponds very closely to the results of `methods`. In both cases, the scope of operations is the entirety of all public methods of a given object.

In addition to asking objects what methods they know about, it's frequently useful to ask classes and modules what methods they provide.

15.2.3 Getting class and module instance methods

Classes and modules come with a somewhat souped-up set of method-querying methods. Examining those available in `String` illustrates the complete list. The methods that are specific to classes and modules are in **bold**.

```
>> String.methods.grep(/methods/).sort
=> [:instance_methods, :methods, :private_instance_methods,
    :private_methods,
    :protected_instance_methods, :protected_methods, :public_instance_methods,
    :public_methods, :singleton_methods]
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

The methods shown in bold give you lists of instance methods of various kinds defined in the class or module. The four methods work as follows:

- `instance_methods` returns all public and protected instance methods
- `public_instance_methods` returns all public instance methods
- `protected_instance_methods` and `private_instance_methods` return all protected and private instance methods, respectively.

When calling any of these methods, you have the option of passing in an argument. If you pass in the argument `false`, then the list of methods you get back will include only those defined in the class or module you're querying. If you pass in any argument with Boolean truth (i.e., anything other than `false` or `nil`), or if you pass in no argument, the list of methods will include those defined in the class or module you're querying and all of its ancestor classes and modules.

For example, you can find out which instance methods the `Range` class defines like this:

```
>> Range.instance_methods(false).sort
=> [:=, :==, :begin, :cover?, :each, :end, :eql?, :exclude_end?, :first,
:hash, :include?, :inspect, :last, :max, :member?, :min, :step, :to_s]
```

Going one step further, what if you want to know which of the methods defined in the `Enumerable` module are overridden in `Range`? You can find this out by performing an “and” (&) operation on the two lists of instance methods: those defined in `Enumerable` and those defined in `Range`.

```
>> Range.instance_methods(false) & Enumerable.instance_methods(false)
=> [:first, :min, :max, :member?, :include?]
```

As you can see, `Range` redefines five methods that `Enumerable` already defines.

We'll look shortly at the last of the “methods”-style methods, namely `singleton_methods`. But first, let's create a little program that produces a list of all the overrides of all classes that mix in `Enumerable`.

GETTING ALL THE ENUMERABLE OVERRIDES

The strategy here will be to find out which classes mix in `Enumerable`, and then to perform on each such class an “and” operation like the one in the last example, storing the results and, finally, printing them out. Listing 15.4 shows the code.

Listing 15.4 Printing out a report on all `Enumerable` descendants' overrides of `Enumerable` instance methods

```
overrides = {} #1
enum_classes = ObjectSpace.each_object(Class).select do |c| #2
  c.ancestors.include?(Enumerable)
end
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>


```

enum_classes.sort_by {|c| c.name}.each do |c|                               #3
  overrides[c] = c.instance_methods(false) &
    Enumerable.instance_methods(false)
end

overrides.delete_if {|c, methods| methods.empty? }                       #4
overrides.each do |c, methods|                                           #5
  puts "Class #{c} overrides: #{methods.join(", ")}"
end

```

First, we create an empty hash, in the variable `overrides`. #1 We then get a list of all classes that mix in `Enumerable`. The technique for getting this list involves the `ObjectSpace` module, and its `each_object` method. #2 This method takes a single argument representing the class of the objects you want it to find. In this case, we're interested in objects of class `Class`, and we're only interested in ones that have `Enumerable` among their ancestors. The `each_object` method itself returns an enumerator, and the call to `select` on that enumerator has the desired effect of filtering the list of all classes down to a list of only those that have mixed in `Enumerable`.

Now it's time to populate the `overrides` hash. For each class in `enum_classes` (nicely sorted by class name), we'll put an entry in `overrides`. The key will be the class itself, and the value will be an array of methods names--the names of the `Enumerable` methods that this class overrides. #3 After removing any entries representing classes that have not overridden any `Enumerable` methods, #4 we proceed to print out the results, using `sort` and `join` operations to make the output look consistent and clear. #5

The output looks like this:

```

Class ARGF.class overrides: to_a
Class Array overrides: collect, count, cycle, drop, drop_while, find_index,
first, include?, map, reject, reverse_each, select, sort, take, take_while,
to_a, zip
Class Enumerable::Enumerator overrides: each_with_index
Class Hash overrides: include?, member?, reject, select, to_a
Class Range overrides: first, include?, max, member?, min
Class Struct overrides: select, to_a

```

The first line pertains to the somewhat anomalous object designated as `ARGF.class`, which is a unique, specially engineered object involved in the processing of program input. The other lines pertain to several familiar classes that mix in `Enumerable`. In each case, you see how many, and which, `Enumerable` methods the class in question has overridden.

The one "methods"-style method we haven't looked at is `singleton_methods`, which you can call on any object.

15.2.4 Listing objects' singleton methods

A singleton method, as you know, is a method defined for the sole use of a particular object (or, if the object is a class, for the use of the object and its subclasses), and stored in that

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

object's singleton class. You can use the `singleton_methods` method to list all such methods. Note that `singleton_methods` lists public and protected singleton methods, but not private ones. Here's an example.

```
class C
end

c = C.new #1

class << c #2
  def x
  end

  def y
  end

  def z
  end

  protected :y #3
  private :z
end

p c.singleton_methods.sort #4
```

An instance of class `C` is created, #1 and its singleton class opened up. #2 Three methods are defined in the singleton class, one each at the public (`x`), protected (`y`), and private (`z`) levels. #3 The printout of the singleton methods of `c` #4 looks like this:

```
[:x, :y]
```

Singleton methods are also considered just "methods". The methods `:x` and `:y` will show up if you call `c.methods` too. Furthermore, you can use the class-based method query methods on the singleton class itself. Add this code to the end of the last example:

```
class << c
  p private_instance_methods(false)
end
```

and when you run it, you'll see this:

```
[:z]
```

The method `:z` is a singleton method of `c`, which means that it's an instance method (a private instance method, as it happens) of `c`'s singleton class.

You can ask a class for its singleton methods, and you'll get the singleton methods defined for that class and for all of its ancestors. Here's an irb-based illustration:

```
>> class C
>>   def self.my_class_method
>>   end
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

```
>> end
=> nil
>> def C.another_method_on_C
>> end
=> nil
>> C.singleton_methods
=> [:my_class_method, :another_method_on_C]
```

Once you get some practice using the various “methods” methods, you’ll find them useful for studying and exploring how and where methods are defined. For example, you can use method queries to examine how the class methods of `File` are composed. To start with, find out which class methods `File` inherits from its ancestors, as opposed to those it defines itself.

```
>> File.singleton_methods - File.singleton_methods(false)
=> [:new, :open, :sysopen, :for_fd, :popen, :foreach, :readlines,
:read, :select, :pipe, :try_convert, :copy_stream]
```

The call to `singleton_methods(false)` provides only the singleton methods defined on `File` itself. The call without the `false` argument provides all the singleton methods defined on `File` and its ancestors. The difference is therefore the ones defined by the ancestors.

The superclass of `File` is `IO`. Interestingly, though not surprisingly, all twelve of the ancestral singleton methods available to `File` are defined in `IO`. You can confirm this by another query.

```
>> IO.singleton_methods(false)
=> [:new, :open, :sysopen, :for_fd, :popen, :foreach, :readlines,
:read, :select, :pipe, :try_convert, :copy_stream]
```

The relationship among classes--in this case, the fact that `File` is a subclass of `IO`, and therefore shares its singleton methods (i.e., its class methods)--is directly visible in the method-name arrays. The various “methods” methods allow for almost unlimited inspection and exploration of this kind.

At this point, we’re going to take a little bit of a side-trip, for the purpose of seeing some of the techniques from the last couple of sections in use.

The next stop on the tour is the introspection of variables and constants.

15.3 Introspection of variables and constants

Ruby can tell you several things about which variables and constants you have access to at a given point in runtime. You can get a listing of local or global variables; an object’s instance variables; the class variables of a class or module; and the constants of a class or module.

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

15.3.1 Listing local, global, and instance variables

The local and global variable inspections are straightforward: you use the top-level methods `local_variables` and `global_variables`, and in each case you get back an array of symbols corresponding to the local or global variables currently defined.

```
x = 1
p local_variables
[:x]

p global_variables
[:$, :$-F, :$@, :$!, :$SAFE, :$~, :$&, :$\`, :$\', :$+, :$=,
:$KCODE, :$-K, :$,, :$/, :$-0, :$\, :$_, :$stdin, :$stdout,
:$stderr, :$>, :$<, :$. , :$FILENAME, :$-i, :$*, :$?, :$$, :$: ,
:$-I, :$LOAD_PATH, :$, , :$LOADED_FEATURES, :$VERBOSE, :$-v, :$-w,
:$-W, :$DEBUG, :$-d, :$0, :$PROGRAM_NAME, :$-p, :$-l, :$-a, :$1,
:$2, :$3, :$4, :$5, :$6, :$7, :$8, :$9]
```

The global variable list includes globals like `$:` (the library load path, also available as `$LOAD_PATH`), `$~` (a globally-available `MatchData` object based on the most recent pattern-matching operation), `$0` (the name of the file in which execution of the current program was initiated), `$FILENAME` (the name of the file currently being executed), and others. The local variable list includes all currently defined local variables.

Note that `local_variables` and `global_variables` do not give you the values of the variables they report on. They just give you the names. The same is true of the `instance_variables` method, which you can call on any object. Here's another rendition of a simple `Person` class, which will illustrate what's involved in an instance variable query.

```
class Person
  attr_accessor :name, :age
  def initialize(name)
    @name = name
  end
end

david = Person.new("David")
david.age = 49

p david.instance_variables      #A

#A Output: [:@name, :@age]
```

The object `david` has two instance variables initialized at the time of the query. One of them, `@name`, was assigned a value at the time of the object's creation. The other, `@age`, is present because of the accessor attribute "age". Attributes are implemented as read and/or write methods around instance variables; so even though `@age` does not appear explicitly anywhere in the program, it does get initialized when the object is assigned an age.

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

All instance variables begin with the @ character, and all globals begin with \$. You might therefore expect Ruby not to bother with those characters when it gives you lists of variable names. But the names you get in the lists do, in fact, include the beginning characters.

Now we'll look at in this chapter is execution-tracing techniques that help you determine the method-calling history at a given point in run-time.

15.4 Tracing execution

No matter where you are in the execution of your program, you got there somehow. Either you're at the top level, or you're one or more method calls deep. Ruby provides you with information about how you got where you are.

The chief tool for examining method-calling history is the `caller` method.

15.4.1 Examining the stack trace with `caller`

The `caller` method provides you with an array of strings. Each string represents one step in the stack trace; that is, a description of a single method call along the way to where you are now. The strings contain information about the file or program where the method call was made; the line on which the method call occurred; and the method from which the current method was called, if any.

Here's an example. Put these lines in a file called `tracedemo.rb`.

```
def x
  y
end

def y
  z
end

def z
  puts "Stacktrace: "
  p caller
end

x
```

All this little program does is bury itself in a stack of method calls: `x` calls `y`, `y` calls `z`. Inside `z`, we get a `stacktrace`, courtesy of `caller`. Here's the output:

```
Stacktrace:
["tracedemo.rb:6:in `y'", "tracedemo.rb:2:in `x'", "tracedemo.rb:14:in
`<main>'"]
```

Each string in the `stacktrace` array contains one link in the chain of method calls that got us to the point where `caller` was called. The first string represents the most recent call in the history: we were at line 6 of `tracedemo.rb`, inside the method `y`. The second string shows

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

that we got to `y` via `x`. The third, final string tells us that we were in `<main>`, which means the call to `x` was made from the top level, rather than from inside a method.

You may recognize the `stacktrace` syntax from the messages you've seen from fatal errors. If you rewrite the `z` method to look like this:

```
def z
  raise
end
```

the output will now look like this:

```
tracedemo.rb:10:in `z': unhandled exception
  from tracedemo.rb:6:in `y'
  from tracedemo.rb:2:in `x'
  from tracedemo.rb:13:in `<main>'
```

which is, of course, a slightly prettified version of the `stacktrace` array we got the first time around from `caller`.

Ruby stacktraces are useful, but they're also looked askance at a bit because they consist solely of strings. If you want to do anything with the information a stacktrace provides you, you have to scan or parse the string and extract the useful information. Another approach, though, is to write a Ruby tool for parsing stacktraces and turning them into objects.

15.4.2 Writing a tool for parsing stacktraces

Given a `stacktrace`--an array of strings--we want to generate an array of objects, each of which has knowledge of a program or file name, a line number, and a method name (or `<main>`). We'll write a `Call` class, which will represent one stacktrace step per object, and a `Stack` class that will represent an entire stacktrace, consisting of one or more `Call` objects. And just to minimize the risk of name clashes, let's put both of these classes inside a module, `CallerTools`.

We can start by describing in more detail what each of the two classes will do.

`CallerTools::Call` will have three reader attributes: `program`, `line`, and `meth`. (It's better to use `meth` than `method` as the name of the third attribute, because classes already have a method called `method` and we don't want to override it.) Upon initialization, an object of this class will parse a stacktrace string and save the relevant substrings to the appropriate instance variables, for later retrieval via the attribute reader methods.

`CallerTools::Stack` will store one or more `Call` objects in an array, which in turn will be stored in the instance variable `@backtrace`. We'll also write a `report` method, which will produce a (reasonably) pretty-printable representation of all the information in this particular stack of calls.

Now, let's write them.

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

CALLERTOOLS::CALL

Listing 15.5 shows the Call class, along with the first line of the entire program, which wraps everything else in the CallerTools module.

Listing 15.5 The beginning of the CallerTools module, including the Call class

```

module CallerTools

  class Call
    CALL_RE = /(.*):(\d+):in `(.*)' / #1
    attr_reader :program, :line, :meth
    def initialize(string) #2
      @program, @line, @meth = CALL_RE.match(string).captures
    end

    def to_s #3
      "%30s%5s%15s" % [program, line, meth]
    end
  end
end

```

We're going to need a regular expression with which to parse the stacktrace strings; that regular expression is stored in the CALL_RE constant. #1 CALL_RE has three parenthetical capture groupings, separated by uncaptured literal substrings. Here's how the regular expression matches up against a typical stacktrace string. Bold type shows the substrings that are captured by the corresponding regular expression sub-patterns. The non-bold characters are not included in the captures, but are matched literally.

```

myrubyfile.rb:234:in `a_method'
  .*      :\d+:in `.*'

```

The class has, as specified, three reader attributes for the three components of the call. #2 Initialization requires a string argument; the string is matched against CALL_RE and the results, available via the captures method of the MatchData object, are placed in the three instance variables corresponding to the attributes. #3 (We'll get a fatal error for trying to call captures on nil if there's no match. You can alter the code to handle this condition directly if you wish.)

We'll also define a to_s method for Call objects. #3 This method will come into play in situations where it's useful to print out a report of a particular backtrace element. It involves Ruby's handy % technique. On the left of the % is a sprintf-style formatting string. On the right is an array of replacement values. You might want to tinker with the lengths of the fields in the replacement string--or, for that matter, write your own to_s method, if you prefer a different style of output.

Now it's time for the Stack class.

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

CALLERTOOLS::STACK

The `Stack` class, along with the closing `end` instruction for the entire `CallerTools` module, is shown in Listing 15.6.

Listing 15.6 The `CallerTools::Stack` class

```
class Stack
  def initialize
    stack = caller #1
    stack.shift
    @backtrace = stack.map do |call| #2
      Call.new(call) #3
    end
  end

  def report
    @backtrace.map do |call|
      call.to_s #5
    end
  end

  def find(&block) #6
    @backtrace.find(&block)
  end
end
```

Upon initialization, a new `Stack` object calls `caller`, and saves the resulting array. #1 It then shifts that array, removing the first string; that string reports on the call to `Stack.new` itself, and is therefore just noise.

The stored `@backtrace` should consist of one `Call` object for each string in the `my_caller` array. That's a job for `map`. #2 Note that there's no `backtrace` reader attribute. In this particular case, all we need is the instance variable, for internal use by the object.

Next comes the `report` method, which uses `map` on the `@backtrace` array to generate an array of strings for all the `Call` objects in the stack. This report array will be suitable for printing or, if need be, for searching and filtering.

There's one final method in the `Stack` class: `find`. It works by forwarding its code block to the `find` method of the `@backtrace` array. It works a lot like some of the deck-of-cards methods you've seen, which forward a method to an array containing the actual cards that make up the deck. Techniques like this allow you to fine-tune the interface of your objects, using underlying objects to provide them with exactly the functionality they need. (You'll see the specific usefulness of `find` shortly.)

Now let's try out `CallerTools`.

USING THE CALLERTOOLS MODULE

You can use a modified version of the little "x, y, z" demo from section 15.5.1 to demo `CallerTools`. Put this code in a file called `callerdemo.rb`:

```
require 'callertools'
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>


```

def x
  y
end

def y
  z
end

def z
  stack = CallerTools::Stack.new
  puts stack.report
end

x

```

When you run this program, you'll see this output:

```

callertest.rb 12      z
callertest.rb  8      y
callertest.rb  4      x
callertest.rb 16      <main>

```

Nothing too fancy, but a nice programmatic way to address a stacktrace, rather than having to munge the strings directly every time.

Next on the agenda, and the last stop for this chapter, is another project, one which ties together a number of the techniques we've been looking at: stack tracing, method querying, and callbacks, as well as some techniques you know from elsewhere in the book. We're going to write a little test framework.

15.5 Callbacks and method inspection in practice

In this section, we're going to implement `MicroTest`, a tiny test framework. It won't have many features, but the ones it has will demonstrate some of the power and expressiveness of the callbacks and inspection techniques you've just learned.

First, a bit of back-story.

15.5.1 The `MicroTest` background: `TestUnit`

Ruby ships with a testing framework called `TestUnit`. The way you use `TestUnit` is that you write a class that inherits from the class `Test::Unit::TestCase`, and every method inside that class whose name begins with the string `test` is automatically executed when you run the file. Inside those methods, you write *assertions*. The truth or falsehood of your assertions determines whether or not your tests pass.

NOTE: MINITEST

As of Ruby 1.9.1, the `TestUnit` framework is provided alongside a new test framework called `Minitest`. Both are available and usable. `MiniTest`'s syntax differs somewhat from that of `TestUnit`, as does its output format, but the examples in this section should run

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

with either. You just have to change `require 'test/unit'` to `require 'minitest/unit'`, and change the class name `Test::Unit` to `MiniTest::Unit`.

So as not to keep you in suspense, the exercise we'll do here is to write a very simple testing utility based on some of the same principles as `TestUnit`. In order to help you get your bearings further, we'll look first at a full example of `TestUnit` in action, and then do the implementation exercise.

We'll test dealing cards. Listing 15.4 shows a version of a class for a deck of cards. The deck consists of an array of fifty-two strings, held in the `@cards` instance variable. Dealing one or more cards means popping that many cards off the top of the deck.

Listing 15.7 A simple deck of cards implementation with card-dealing capabilities

```
module PlayingCards
  RANKS = %w{ 2 3 4 5 6 7 8 9 10 J Q K A }
  SUITS = %w{ clubs diamonds hearts spades }
  class Deck
    def initialize #1
      @cards = []
      RANKS.each do |r|
        SUITS.each do |s|
          @cards << "#{r} of #{s}"
        end
      end
      @cards.shuffle!
    end

    def deal(n=1) #2
      n.times.map { @cards.pop }
    end

    def size #3
      @cards.size
    end
  end
end
```

Creating a new deck #1 involves initializing `@cards`, inserting fifty-two strings into it, and shuffling the whole array. Each string takes the form "*rank of suit*", where *rank* is one of the ranks in the constant array `RANKS`, and *suits* is one of `SUITS`. Dealing from the deck #2 depends on the fact that `times` returns an enumerator. Chaining a `map` operation onto that enumerator ensures that the whole thing will be run `n` times, while returning an array representing one execution of the code block for each time through. Thus we end up with an array of `n` card objects, popped successively off the end of `@cards`.

So far so good. Now, let's test it. Enter `TestUnit`. Listing 15.8 shows the test code for the `cards` class. The test code assumes that you've saved the `cards` code itself to a separate file

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

called `cards.rb`, in the same directory as the test-code file (which you can call `cardtest.rb`).

Listing 15.8 The `cardtest.rb` file, testing the dealing accuracy of the `PlayingCards::Deck` class

```
require 'test/unit'           #1
require 'cards'

class CardTest < Test::Unit::TestCase #2
  def setup                    #3
    @deck = PlayingCards::Deck.new
  end

  def test_deal_one           #4
    @deck.deal                #5
    assert_equal(51, @deck.size) #5
  end

  def test_deal_many         #6
    @deck.deal(5)
    assert_equal(47, @deck.size)
  end
end
```

The first order of business is to require both the TestUnit library and the `cards.rb` file. #1 TestUnit is installed on your system in such a way that you need to use the double-barreled `test/unit` syntax to specify it. Next, we create a class, `CardTest`, that inherits from `Test::Unit::TestCase`. #2 In this class we define three methods. The first is `setup`. The method name `setup` is “magic” to TestUnit. If you define this method (which you don’t have to), it will be executed before every single test method in your test class. Running the `setup` method before each test method contributes to keeping the test methods independent of each other; and that independence is an important part of the architecture of test suites.

Now come the two test methods, `test_deal_one` #4 and `test_deal_many`. #6 These methods define the actual tests. In each case, we’re dealing from the deck, and then making an assertion about the size of the deck subsequent to the dealing. Remember that `setup` is executed before each test method, which means that `@deck` contains a full 52-card deck for each method.

The assertions are performed using the `assert_equal` method. #5 This method takes two arguments. If the two are equal (using `==` to do the comparison behind the scenes), the assertion succeeds. If not, it fails.

Now execute `cardtest.rb` from the command line. Here’s what you’ll see (probably with a different time measurement):

```
$ ruby cardtest.rb
Loaded suite cardtest
Started
..
```

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

```
Finished in 0.000768 seconds.
```

```
2 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

The last line tells you that there were two methods whose names began with `test` (2 tests), and a total of two assertions (the two calls to `assert_equal`). It tells you further that both assertions passed (no failures), and that nothing went drastically wrong (no errors; an error is something unrecoverable like a reference to an unknown variable, whereas a failure is simply an incorrect assertion).

The most striking thing about running this test file is that at no point did you have to *instantiate* the `CardTest` class, nor did you have to call the test methods, or the setup method, explicitly. TestUnit figures out what you want to do, simply by your defining the class and the methods.

That--or at least a subset of it--is what we're going to implement in our exercise.

15.5.2 Specifying and implementing *MicroTest*

Here's what we'll want from our *MicroTest* utility:

- Automatic execution of setup method and test methods, based on class inheritance.
- A simple assertion method which either succeeds or fails.

The first specification will entail most of the work.

We need a class which, upon being inherited, observes the new subclass and executes the methods in that subclass as they're defined. For the sake of (relative!) simplicity, we'll execute them in definition order, which means that `setup` should be defined first.

Here's a more detailed description of the steps needed to implement *MicroTest*.

- Define the class `MicroTest`
- Define `MicroTest.inherited`
- Inside `inherited`, the inheriting class should:
- define its own `method_added` callback, which should:
- instantiate the class and execute the new method if it starts with "test", but first:
- execute the setup method, if there is one

And here's a non-working, commented mock-up of *MicroTest*, in Ruby.

```
class MicroTest
  def self.inherited(c)           #1
    c.class_eval do             #2
      def self.method_added(m)   #3
        # If m starts with "test"
        #   Create an instance of c
        #   If there's a setup method
        #     Execute setup
        #   Execute the method m
      end
    end
  end
end
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

```
end
```

There's a kind of logic cascade here. Inside `MicroTest` we define `self.inherited`, which receives the inheriting class (the new subclass) as its argument. We then enter into that class's definition scope using `class_eval`. Inside that scope, we implement `method_added`, which will be called every time a new method is defined in the class.

Writing the full code follows directly from the comments inside the code mockup. Listing 15.9 shows the full version of `micro_test.rb`.

Listing 15.9 `MicroTest`, a tiny testing class that emulates some of `TestUnit`'s functionality

```
require 'callertools'

class MicroTest
  def self.inherited(c)
    c.class_eval do
      def self.method_added(m)
        if m.to_s =~ /^test/ #1
          obj = self.new #2
          if self.instance_methods.include?(:setup) #3
            obj.setup
          end
          obj.send(m)
        end
      end
    end
  end

  def assert(assertion) #4
    if assertion
      puts "Assertion passed"
      true
    else
      puts "Assertion failed:"
      stack = CallerTools::Stack.new #5
      failure = stack.find {|call| call.meth !~ /assert/ } #6
      puts failure
      false
    end
  end

  def assert_equal(expected, actual) #7
    result = assert(expected == actual)
    puts "(#{actual} is not #{expected})" unless result #8
    result
  end
end
```

Once inside the class definition (`class_eval`) scope of the new subclass, we define `method_added`, and that's where most of the action is. If the method being defined starts with "test", #1 we create a new instance of the class. #2 If there's a `setup` method defined,

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=417>

#3 we call it on that instance. Then (and whether or not there was a `setup` method; that's optional) we call the newly-added method itself, using `send` since we don't know the method's name.

The `assert` method tests the truth of its single argument. If the argument is true (in the Boolean sense; it doesn't have to be the actual object `true`), a message is printed out, indicating success. If the assertion fails, the message printing gets a little more intricate. We create a `CallerTools::Stack` object, and pinpoint the first `Call` object in that stack whose method name does not contain the string "assert". The purpose here is to make sure we don't report the failure as having occurred in the `assert` method itself, nor in the `assert_equal` method (described shortly). It's not very robust; you might have a method with "assert" in it that you *did* want an error reported from. But it illustrates the kind of manipulation that the `find` method of `CallerTools::Stack` allows for.

The second assertion method, `assert_equal`, tests for equality between its two arguments. #7 It does this by calling `assert` on a comparison. If the result is not true, an error message showing the two compared objects is displayed. #8 Either way—success or failure—the result of the `assert` call is returned from `assert_equal`.

To try out `MicroTest`, put the following code in a file called `microcardtest.rb`, and run it from the command line.

```
require 'microtest'
require 'cards'

class CardTest < MicroTest
  def setup
    @deck = PlayingCards::Deck.new
  end

  def test_deal_one
    @deck.deal
    assert_equal(51, @deck.size)
  end

  def test_deal_many
    @deck.deal(5)
    assert_equal(47, @deck.size)
  end
end
```

As you can see, this code is almost identical to the `TestUnit` test file we wrote before. The only differences, in fact, are the names of the test library and parent test class. And when you run it, you get these somewhat obscure but encouraging results:

```
Assertion passed
Assertion passed
```

If you want to see a failure, change 51 to 50 in `test_deal_one`.

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=417>

```
Assertion failed:
      microcardtest.rb  11  test_deal_one
(51 is not 50)
Assertion passed
```

MicroTest isn't going to supplant TestUnit any time soon. But it does do a couple of the most "magic" things that TestUnit does. It's all made possible by Ruby's introspection and callback facilities, techniques that put extraordinary power and flexibility in your hands.

15.6 Summary

We've covered a lot of ground in this chapter, and practicing the techniques covered here will contribute greatly to your grounding as a Rubyist. We looked at intercepting unknown messages with `method_missing`, along with other runtime hooks and callbacks like `Module.included`, `Module.extended`, and `Class.inherited`. The chapter also took us into method querying in its various nuances: `public`, `protected`, `private`; `class`, `instance`, `singleton`. You've seen some examples of how this kind of querying can help you derive information about how Ruby does its own class, module, and method organization.

The last overall topic was the handling of stacktraces, which we put to use in the `CallerTools` module. Finally, the chapter ended with the extended exercise consisting of implementing `MicroTest`, which pulled together a number of topics and threads from this chapter and elsewhere.

We've been going through the material methodically and deliberately, as befits a grounding or preparation. But if you look at the results, particular `MicroTest`, you can see how much power Ruby gives you in exchange for relatively little effort. That's why it pays to know about even what might seem to be the "magic" or "meta" parts of Ruby. They really aren't. It's all just Ruby, and once you internalize the principles of class and object structure and relationships, everything else follows.

And that's that! Enjoy your groundedness as a Rubyist, and the many structures you will build on top of foundation you've acquired through this book.