Chapter 1

Learning Ruby

Ruby on Rails is a Web application development framework built using the Ruby programming language. Ruby is a dynamic language that was created in Japan by Yukihiro Matsumoto. You'll often see Matsumoto referred to simply as Matz. While Ruby had been growing and flourishing in Japan and Europe, it took the Rails framework to finally thrust Ruby into the limelight in the United States. Ruby is steadily growing in popularity worldwide as a programming language of choice.

It is often described as an elegant language that allows developers to create concise and very readable code.

If you already consider yourself a Ruby expert, you can probably skip this chapter; otherwise, I highly recommend reading this chapter before getting into the details of using Rails. A solid knowledge of the Ruby programming language makes an excellent foundation for learning and using the Ruby on Rails framework. A solid understanding of Ruby will also help you if you want to explore the internals of Rails. Remember that Rails is an open source project, meaning that all of its source code is available to anyone who wants to look at it. You can learn a great deal about advanced Ruby techniques from reading the Rails source code. The power of Ruby plays a large part in the success of the Rails framework.

IN THIS CHAPTER

The Nature of Ruby

Programming languages tend to have various elements of commonality. I'm not referring to the syntax of a language, but rather higher-level designs that apply to a programming language. These elements are what make up the nature of the language. Here you begin by learning about the nature of Ruby,

or what kind of programming language Ruby is. Each of the characteristics you will read about in this section will help you better understand the type of programming language that Ruby is, and how it is different from or the same as other languages that you might have experience with. The elements of Ruby discussed here are dynamic or static typing systems, duck typing, and compiled or scripted language.

Dynamic or static typing

Programming languages can be classified by the type system they use. A *type system* defines how a programming language classifies its data and methods into types. Examples of types used in various languages include int, float, String, and Object. A *type* describes the kind of data used in a particular variable. There are two general classes of type systems that are used by programming languages: dynamic typing and static typing. In a *static typed* language, the compiler enforces type checking before run-time. In a *dynamic typed* language, type checking is deferred until run-time.

In a static typed system, the programmer uses variable declarations to provide type information. For example, in Java, which is a statically typed language, variables must be declared with their type prior to being used. Examples of statically typed languages include Java, C, C++, C#, and Pascal.

NOTE **Other languages, such as OCaml and Haskell, use *type inference*. This is a form of static typing where the type is determined at compile time but without the programmer having to declare it.**

In a dynamically typed language, the programmer does not have to declare data types with variable declarations. Data types are not known until run-time, and type checking of variables does not occur until run-time. Examples of dynamically typed languages include Python, JavaScript, Perl, Lisp, and Ruby.

A closely related concept is that of how strictly a type system enforces type rules. A *strongly typed system* enforces type rules strongly, allowing for automatic conversions between types only when information is not lost due to the conversion. A *weakly typed system* does not enforce type rules and allows you to easily convert from one type to another without complaint. Ruby is a weakly typed, dynamic programming language.

Languages that are statically typed are usually recommended for new developers, as they provide more protection from run-time errors than a dynamically typed language provides. In a dynamically typed language, it's easy to make programming errors that are not detected until run-time. However, if you have solid unit-testing practices, this can alleviate that concern. Because of this, unit testing is even more important when you're programming in a dynamically typed language such as Ruby.

Duck typing

You will likely hear someone refer to Ruby as having duck typing. The term d*uck typing* refers to the popular quote, "If it looks like a duck and quacks like a duck, it must be a duck." So how does that quote have anything to do with a programming language? Let's figure that out.

In Ruby, object instances are not forced to be of any certain type when they are used. As long as the object being used meets the requirements of the situation in which it is being used, Ruby will not complain. Another way of saying this is that in Ruby, an object's type is determined by what it can do, not by its class. Say you were calling a `calculate_average` method on an object. In Ruby, as long as the object you are calling that method on implements a `calculate_average` method, everything works fine.

Your code doesn't have to require the object you are calling the `calculate_average` method on to be of any certain class. You might have a method that is expecting an object of a certain class, but if you had some code that passed in a different class of object, but implemented all the methods used within that method, the code would execute perfectly fine. This is how programming in Ruby relates back to the quote, "If it looks like a duck and quacks like a duck, it must be a duck." If your objects behave like the type expected at any given place in your code, then as far as Ruby is concerned, they are of that type, regardless of their real class.

This is very different than the way that many other languages work, including Java. In Java, you must declare the class type of all of your method parameters. You cannot pass in an instance of a class that does not match the class type that the method is expecting, even if that instance implements the same methods as the expected class.

Compiled or scripting language

Another way you can classify languages is by whether they require a compile step or not. Depending on whether or not they require compilation, a language can be said to be a compiled language or a scripting language.

Compiled languages

A *compiled language* requires you to perform a compile step before running the application you are writing. The language's run-time executable that runs applications cannot directly understand the source code of a compiled language. The compiler converts your source code into a binary format that can be understood by the run-time executable. After you compile your source code, you end up with files in the compiled format, such as the .class files used by Java. Examples of compiled languages include C, C++, C#, and Java.

Scripted languages

A *scripted language* does not require you to compile your source code into another form. The source code that you write is also the code that the language's run-time executable uses to execute your application. The run-time executable of a scripting language is usually called an *interpreter*. The interpreter interprets the source code at run-time and converts it to a format that the computer can execute. The interpreter is specific to a particular language. Examples of scripted languages include Perl, Python, JavaScript, and Ruby.

Compiled languages are usually faster at run-time because the code is already closer to that of the computer, whereas code from a scripted language has to be interpreted at run-time. However, many people believe that scripted languages make up for the run-time performance deficit by

being faster to develop an application in. With a compiled language, every time you make a change, you have to go through a compile phase and an application restart to see the results of that change. In a scripting language, your application can immediately see the results of a source code change, as it is running directly from your source code.

Object Oriented Programming

Object oriented programming (OOP) is a style of programming that uses objects to represent data, and actions that you can perform on that data. OOP allows you to more closely model the real world with your objects than was possible prior to the advent of OOP. Instead of dealing with functions and procedures when designing an application, OOP allows you to model the application in terms of objects that make up the application's domain. For example, if you were creating an application that catalogued books, in an OOP design you would model the application using objects extracted from the domain, such as Books, Titles, Inventory, and Publisher.

In OOP, you'll often hear the terminology of sending messages to objects. Sending a message to an object is the equivalent of asking that object to perform some action for you. The action usually manipulates or provides you with data that the object contains. These actions are called *methods*. For example, with a Book object, you might have a method called `get_page_count` that would return the book's page count.

An object can have both methods and data. An object stores data in fields called *attributes*. Considering the Book object example again, a Book object may have attributes of `title`, `publisher`, and `publication_date`. Methods and attributes are the two components that make up the definition of an object.

The objects in your application will relate to the domain your application serves. For example, if you are writing an accounting application, you might have objects called Account, User, and Bank. Your Account object might contain methods for depositing and withdrawing money from an account. The attributes of the Account object might include an account number, an account name, and an account balance. When you are writing an application using an object-oriented language, your work consists of defining objects and using those objects to perform the logic of your application.

Ruby is a pure object-oriented programming language. In Ruby, everything is an object, including literal strings and numeric types. Objects are at the center of all the code you will write in Ruby. Unlike most other languages, Ruby does not have any native types that are not objects. Even numeric types such as integers and floats are represented as objects in Ruby.

People new to Ruby often don't initially grasp the fact that in Ruby, everything is an object. As an example, look at the following line of code:

```
3.methods
```

This is a valid line of Ruby code that might look a bit strange to you if you're coming to Ruby from an object-oriented language that considers numeric values as native types instead of objects. This

line asks for the methods that are available on the `3` object. If you wanted to find out what type of object the number 3 is, you could find that out using this line of code:

```
3.class
```

This will return the class `Fixnum`. In Ruby, integer numbers are instances of the `Fixnum` class. The methods `class` and `methods` are available on any object that you use in Ruby.

The Basics of Ruby

Before you get into the details of working with Ruby objects, this section provides you with some of the basics that you should be familiar with when writing and running Ruby programs. With the knowledge that this section provides you, you should have no problem walking through the examples that are used throughout the remainder of this chapter. You'll also know how to interactively follow along with the examples and run them on your own computer. If you are new to Ruby, an active learning style in which you try out the examples yourself will help you master the language more efficiently than if you choose to only read through all of the examples.

The basics of Ruby that will prepare you to successfully learn the remainder of the language are Ruby's interactive shell, Ruby syntax basics, and Running Ruby programs.

Ruby's interactive shell

Assuming that you already have Ruby installed on your computer, you have access to a powerful, interactive Ruby-programming environment called irb. The *irb* environment is a command-line Ruby interpreter that lets you enter any valid Ruby syntax and instantly see the results of your actions. This is being covered before you even learn Ruby because of its great use as a Ruby learning tool. Throughout this chapter, you will be able to try out the short snippets of Ruby code that are discussed so that you can interactively follow along as you read. That is a much better style of learning than just reading through the code samples.

Use the following steps to start irb:

1. **Start irb in any command-line environment simply by typing** `irb`**.** This assumes that the `bin` directory of the Ruby installation is in your executable path, which would be the case if you used an automatic installer like the one-click Ruby installer for Windows.

   ```
   C:\> irb
   ```

2. **After typing irb, you should see a command prompt that looks like this:**

   ```
   irb(main):001:0>
   ```

3. **At the command prompt, go ahead and type the following line of Ruby code:**

   ```
   irb(main):001:0> puts "Hello, World"
   ```

   The `puts` method writes the passed-in string to the console output.

4. **Press Enter.** You should see this:

```
Hello, World
=> nil
Irb(main):002:0>
```

You see the "Hello, World" string printed. You may not have expected the next line: `=> nil`. Anytime you execute a line of code in irb, the return value of the executed method is printed to the console after any values that the method itself may have printed. The `puts` method always returns a `nil` value. The value `nil` is Ruby's equivalent to a null or empty value.

You can even create methods and then execute them within irb. Try this within irb:

```
irb(main):001:0> def add_nums(a,b)
irb(main):001:0> return a+b
irb(main):001:0> end
=> nil
```

You have just created a method named `add_nums` that takes two parameters. The method returns the value of those two parameters added together. You can now try out your new method.

Make sure you are still in the same irb session and type this:

```
irb(main):001:0> add_nums(5,7)
=> 12
```

Here, you called the method that you created and passed the values 5 and 7. The method returned the sum of those two values, 12, and so that value is printed to the console.

The irb tool will become one of your best friends as a Ruby programmer.

As you work through the remainder of this chapter, I strongly suggest that you leave the irb console open and try out the small code snippets as you see them.

Ruby syntax basics

Ruby's syntax borrows some of the best features from languages such as Java and Perl. Before you begin to program in Ruby, there are a few basic syntax elements that you'll learn here. These include adding comments in Ruby, use of parentheses, use of white space, and use of semicolons.

Adding comments

A language's support for comments allows you to add lines to your source code that the interpreter or compiler ignores. Comments can be added to Ruby source code using the hash (or pound) symbol, #. All text that follows a # symbol is considered a comment and ignored by the Ruby interpreter.

```
# This is a comment in a Ruby source code file
puts 'Camden Fisher' # This line outputs a string to the console
```

As you see in the example above, a comment can be a complete line, or it can follow a line of Ruby code.

If you have a large block of text that you want to use as a comment, instead of beginning each line of the comment with a # symbol, you can use Ruby's multi-line comment syntax shown here:

```
=begin
This is a multi-line block of comments in a Ruby source file.
   Added: January 1, 2008
   By: Timothy Fisher
=end
Puts "This is Ruby code"
```

The `=begin` marks the beginning a multi-line comment, and the `=end` closes the multi-line comment.

It's a good idea to add comments explaining any code that is not understandable simply by looking at it. If you often find yourself writing complex code that requires comments to explain, you may consider refactoring that code to make it easier to understand and eliminate the need for the comments.

Using parentheses

The use of parentheses in Ruby is most often optional. For example, when you call a method that takes parameters, you could call it like this:

```
movie.set_title("Star Wars")
```

or you could call it like this without the parentheses:

```
movie.set_title "Star Wars"
```

If you are chaining methods together, you may get a warning (depending on the version of Ruby you are using) if you do not use parentheses around your parameters. For example, if you were writing this code:

```
puts movie.set_title "Star Wars"
```

You may see the a warning message similar to this:

```
warning: parenthesize argument(s) for future version
```

You can avoid the warning by using parentheses like this:

```
puts move.set_title("Star Wars")
```

It is a generally accepted convention amongst Ruby developers to use parentheses if they help a reader understand an expression. If the parentheses add no value to the readability of an expression, your code usually looks cleaner without them.

Using white space

White space is not relevant in Ruby source code. You can use indentation, blank lines, and other white space to make your code readable with no effect on its syntax. While white space has no effect on Ruby syntax, it does have a significant effect on the readability of Ruby code. You should therefore pick a consistent style that uses white space to enhance the readability of your code.

Common convention is to indent your class bodies, method bodies, and blocks. Here is an example showing recommended use of white space in a class:

```
class
    def a_method
        puts 'You called a method'
    end

    def b_method
        puts 'You called b method'
    end
end
```

Most text editors that understand Ruby syntax will help you apply appropriate indentation of your methods and code blocks. Many Ruby authors have adopted an informal standard in the Ruby community of indenting with two spaces and no tabs, so this may be the standard applied in much of the code that you find in the open source community. However, I believe that you should choose an indentation size that works best for you and your team.

Using semicolons

Semicolons are a common indicator of a line or statement ending. In Ruby, the use of semicolons to end your lines is not required. The only time using semicolons is required is if you want to use more than one statement on a single line.

Take a look at a method in Ruby:

```
def add_super_power(power)
    @powers.add(power)
end
```

Notice that there are no semicolons in any of this code, and yet this is perfectly valid Ruby code. Not requiring semicolons is part of what gives Ruby its reputation as allowing for very clean and readable code.

Here is an example of Ruby code that would require the use of a semicolon:

```
def add_super_power(power)
    @powers.add(power);puts "added new power"
end
```

In this method, two Ruby statements are being executed in one line of code. The semicolon separates the statements. In most cases, though, this style of coding is not recommended. Unless you

have a good reason to do otherwise, you should always give each statement its own line of code. This avoids the use of semicolons and makes the code more readable by other developers.

Running Ruby programs

The Ruby source files that you create become the input to the Ruby interpreter. Unlike with compiled languages, with Ruby there is no build step required prior to running your Ruby programs. Running a Ruby program is as simple as calling the Ruby executable and passing it the name of the file containing your Ruby code. The actual executable program that you use to run your Ruby source code files is named `ruby`. Throughout the book, when you see `ruby` written in lowercase letters in the mono-space code font, you can assume it is referring to the actual Ruby executable program.

CROSS-REF **Before you continue, you should have Ruby installed on your computer. Installation instructions were provided in the Quick Start chapter. If you skipped that, now is a good time to go back and get Ruby installed.**

You also need a text editor to create your Ruby source code in. You can use any text editor that you are comfortable with. The Quick Start chapter gave a few recommendations for good text editors to use that feature Ruby code recognition to give you syntax highlighting and some other nice features.

At this time, you should create a directory that you can use to store all of the samples you write in this chapter. Anytime you want to create a Ruby source file, go to that directory and create the file. From that same directory, you can run it with the `ruby` program.

Let's walk through an example of creating and running a simple Ruby program.

1. **In a text editor of your choice, create a file called** `test_app.rb`**.** Enter the following Ruby source code:

```
class SimpleRubyClass
    def simple_method
       puts 'You have successfully run a Ruby program.'
    end
end

my_class = SimpleRubyClass.new
my_class.simple_method
```

2. **From a command line, use the Ruby interpreter to run your program.**

```
> ruby test_app.rb
```

3. **You should see the output from the method you wrote.**

```
> You have successfully run a Ruby program.
```

In this example, you created a simple Ruby class and two additional statements outside of the Ruby class. When you run a Ruby source file, lines of code that are outside of a class definition are automatically executed. In the file you created, the last two lines are automatically executed when you call `ruby test_app.rb`. The first line executed creates an instance of the `SimpleRubyClass`, and the next line calls the `simple_method` on that instance.

When you create a Ruby source file, you do not have to use any classes. Many useful scripts can be written without using any Ruby classes at all.

Table 1.1 lists some commonly used options that you can use with the `ruby` interpreter. For example, especially if you have a large program file, you might find it useful to run a syntax check on the source file before you execute it. You can do that with the `-c` command-line option.

TABLE 1.1

Command-line Options Used with the Ruby Interpreter

Option	Description	Usage
`-c`	Checks the syntax of a source file without executing it.	`ruby -c test_script.rb`
`-e`	Executes code provided in quotation marks.	`ruby -e 'puts "Hello World"'`
`-l`	Prints a new line after every line; also called line mode.	`ruby -l -e 'print "Add a newline"'`
`-v`	Displays the Ruby version information and executes the program in verbose mode.	`ruby -v`
`-w`	Provides warnings during program execution.	`ruby -w test_script.rb`
`-r`	Loads the extension whose name follows the -r option.	`ruby -rprofile`
`--version`	Displays Ruby version information.	`ruby --version`

Classes, Objects, and Variables

Objects are not tacked on to the Ruby language as an afterthought as they are in some languages, such as Perl or early versions of PHP. Nor are objects optional as they are in C++. As you learned previously, Ruby is a pure object-oriented language. In Ruby, everything is an object. This makes learning about objects in Ruby very important. They are the foundation for all of the code you will write in Ruby, and so that is where you'll now begin to explore the details of the Ruby language.

Using objects in Ruby

Since objects and classes are core to Ruby programming, Ruby provides a rich syntax for using them. In this section, you'll learn how to create objects and classes in Ruby. You'll also learn how to create methods and variables that will be contained by the classes and objects that you create.

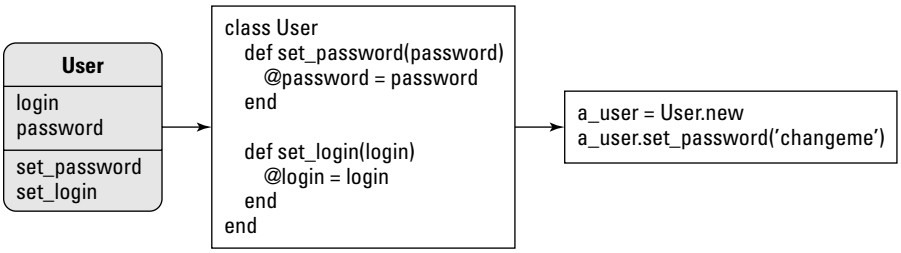
Defining objects

Objects provide a way of modeling your application's data and actions. In Ruby, you define the structure of your objects inside of a class. A class is similar to the concept of a type. A class defines a type of data structure. Looking at Figure 1.1, you see a `User` class that contains two attributes

(`login`, `password`) and two methods (`set_password`, `set_login`). When you use the `User` class, you create an instance of the class. An *instance* of a class is also called an *object*. In Figure 1.1, the object `a_user` is an instance of the `User` class. A class is a way of defining common behavior for all of the objects that are of that class type. In this example, all instances of the `User` class will have the `login` and `password` attributes, and the `set_password` and `set_login` methods.

FIGURE 1.1

The `User` class



You define a class in your source code using the class keyword. The minimum code you need to define a class is a class statement with your class name, and the end statement to close the class definition. The following code would define a `User` class:

```
class User
end
```

It is usually good practice to have each of your classes defined in a separate file. The `User` class would typically be stored in a file called `user.rb`. If you follow this recommendation, your code becomes better organized, and thus more readable and more maintainable.

Classes are made up of attributes and methods. The remainder of this section will show you the details of how to create each of these elements in the Ruby classes you write.

Writing methods

A class's methods define its behavior. Methods allow Ruby classes to perform useful actions and process data in useful ways. When you are writing a Rails application, your application's business logic will be contained in methods that you add to classes. In Ruby, you define methods within classes using syntax that looks like this:

```
class Notifier
    def print_message
        puts 'Wherever you go, there you are.'
    end
end
```

In this example, the class `Notifier` contains one method, named `print_message`. Take a look at that method definition line-by-line to understand all of its parts. The first line is

```
def print_message
```

Ruby uses the `def` keyword to signify the start of a method definition. The `def` keyword is followed by the name of the method you are defining. The method name is also used when the method is called someplace else in your code. In this example, the method name is `print_message`. Your method names should be concise, yet descriptive of the actions that are performed within the method. While some people don't like long method names, it is better to have longer names than short names that do not accurately convey the purpose of a method.

The next line of the method is the first line of the method body:

```
puts 'Wherever you go, there you are.'
```

This line prints a message to the console. The method `puts` is a built-in Ruby method for writing string output to the console. In this case, the method name, `print_message`, is good because it accurately describes what this method does. If you find yourself wondering what a method does after looking at its name, perhaps you should consider renaming the method.

The method body continues until an `end` statement is reached. The `end` statement marks the end of the method. For those coming from Java or Perl, note that you do not surround your method body in curly braces,{ and }, as you do in those languages.

Methods with parameters

You saw an example of a very simple method in the previous section. Methods can also have data passed to them. The data passed to a method can then be used within the body of the method. Data values passed to a method are called *parameters*, or *arguments*, of that method. Here is an example of a method that uses parameters:

```
def add_numbers(number1, number2)
    number1 + number2
end
```

This method, `add_numbers`, takes two parameters, `number1` and `number2`. The parameters are then used within the body of the method. The variables listed between the parentheses are called the *parameter list*. Anytime you use a parameter in the body of your method, you must use the same name for it that is given in the parameter list. Notice that in your parameter list, you do not declare any types for the parameters as you do in Java and other statically typed languages.

You might be wondering if a `return` statement was accidentally left out of the previous method. Perhaps you were expecting to see the method body written like this:

```
return number1 + number2
```

In Ruby, that line is actually equivalent to the line that does not contain the `return` statement. In Ruby, the value of the last statement executed is also returned from the method. Because the

statement `number1 + number2` is the last statement in this method body, its value is returned from the method.

Creating instances of a class

A class defines a type of object. To use an object of that type, you must create an instance of that class. Consider a class designed to implement simple math operations. You might start with a class defined like this:

```
class SimpleMath
    def add_numbers
        number1+number2
    end
end
```

This is the definition of a class called `SimpleMath` containing one method called `add_numbers`. To use the `SimpleMath` class as an object, you have to first create an instance of it. Every class has a method called `new` that is used to create instances of that class. The `new` method is called without any parameters, like this:

```
math = SimpleMath.new
```

The variable `math` now contains an instance of the `SimpleMath` class. Now you can call methods on that instance, like this:

```
result = math.add_numbers(3, 5)
```

This is the first example you've seen of how methods are called in Ruby. Ruby uses the dot operator (.) to indicate that what follows is the name of a method that is to be called on the object preceding the dot operator.

It is common naming practice in Ruby to begin class names with an uppercase letter and capitalize the first letter of each additional word in the class name. Instance names, and all other variables in Ruby, should begin with a lowercase letter and have multiple words joined with an underscore character.

Initializing instances with the initialize() method

Often, you'll want to initialize the state of an object when you create an instance. Many languages include a method that is called when instances are created. Often, this is called an *object constructor*. In Ruby, the concept of a constructor is implemented with the `initialize` method. You can include an `initialize` method in any of your classes, and it will be called when an instance is created using the `new` method. For example, you could have a class defined like this:

```
class PhotoAlbum
    def initialize
        @album_size = 10
    end
end
```

Here, you have a `PhotoAlbum` class containing an `initialize` method that sets the album size to 10 each time an instance of the class is created.

The `initialize` method can also take parameters. Instead of hard-coding an album size, you might prefer an `initialize` method like this:

```
class PhotoAlbum
    def initialize(album_size)
        @album_size = album_size
    end
end
```

In this example, the `initialize` method takes a single parameter, the album size. You pass this parameter to the `new` method when you create an instance of the `PhotoAlbum` class, like this:

```
my_photo_album = PhotoAlbum.new(20)
```

This creates your new instance, initialized with an album size of 20.

Instance and class methods

There are two types of methods that a class can define: instance methods and class methods. Instance methods allow you to interact with instance objects, and class methods allow you to interact with class objects.

Instance methods

In the previous example, the `add_numbers` method that is declared in the `SimpleMath` class is called an instance method. It can only be called on instances of the `SimpleMath` class. Instance methods manipulate only the instance on which they are called. An object instance must be created in order to use instance methods.

Any method defined in a class using the simple format of the `def` keyword followed by a method name is an instance method. Here is a class that contains three instance methods:

```
class SuperHero
    def add_power
        # method body here…
    end

    def use_power
        # method body here…
    end

    def find_enemy
        # method body here…
    end
end
```

The three methods defined in this class are instance methods. You must create an instance of the `SuperHero` class using the `new` method to be able to use any of these methods. Once an instance is created, an instance method is called using the instance variable followed by the dot operator, like this:

```
spiderman = SuperHero.new
spiderman.add_power('super_strength')
spiderman.use_power
```

The majority of the methods you write within your classes will probably be instance methods.

Class methods

There is another type of method that classes can define, called class methods. A class method can only be called on a class and cannot be called from an object instance. You've already seen one example of a class method — the `new` method that is used to create instances of a class.

Class methods can be defined in a few different ways. You do one of the following to define a class method:

- Prefix a method name with the class name and the dot operator.
- Prefix a method name with the `self` keyword and the dot operator.
- Use `class << self` syntax see the following example).

When you call methods or access attributes on a class, you are not using any specific instance of that class. Class methods are called like this:

```
methods = User.methods
```

This line calls the `methods` class method of the `User` class. This would return you an array of all the class methods for the `User` class.

The first way you can define a class method is to write it with a preceding class name, like this:

```
class PhotoAlbum
    def PhotoAlbum.delete(album_id)
        …
    end
end
```

In this example, the `delete` method is created as a class method of the `PhotoAlbum` class. The `delete` method cannot be called from an instance of this class. Instead, you call the `delete` method as shown in the following example, passing an integer that represents the ID of an album you want to delete:

```
PhotoAlbum.delete(12)
```

Another way to define a class method is to use the `self` keyword like this:

```
class PhotoAlbum
    def self.delete
        …
    end
end
```

This creates a `delete` class method for `PhotoAlbum` that behaves identically to the previous version.

The final style you see for defining class methods is useful when you have several class methods that you want to define in one class. You can define a group of class methods using the `class << self` syntax like this:

```
class PhotoAlbum
    class << self
        def delete(album_id)
            …
        end
            …
        def move(album_id)
            …
        end
            …
        def rename(album_id)
            …
        end
    end
end
```

In this example, all three of the methods contained within the block surrounded by `class << self` are defined as class methods.

Instance and class variables

Just as there are two types of methods that a class can contain, there are also two types of variables that a class can contain. The two types are the instance variables and the class variables.

Instance variables

It is very common that you'll want to associate data with specific instances of your classes. For example, you might have a `User` class, with each instance representing a different user. Each instance of user would need its own variables to maintain its object state. Variables that are associated with an instance of a class are called *instance variables*. The following is true of all instance variables:

- Instance variable names always begin with `@` (the at sign).
- You can access instance variables only through the specific class instance to which they belong. Each instance of a class has its own instance variables.

- You can define an instance variable anywhere within a class and it will still be visible to all instance methods within the class.

To illustrate these bullet points, consider this example:

```
class House
    def print_value
        puts @value
    end

    def set_value(a_value)
        @value = a_value
    end
end
```

In this example, because of the `@` symbol, you should be able to identify `@value` as an instance variable. Notice that you do not have to define instance variables outside of your methods as you do in some other languages, such as Java. Anytime you use a variable that begins with an `@` symbol, that variable becomes an instance variable. The `print_value` method accesses the same `@value` variable set by the `set_value` method. Each instance of the `House` class maintains its own copy of the `@value` variable.

Class variables

In addition to instance variables, a class can also define class variables. A *class variable* is a variable that is shared among all instances of a class. Class variables are not referred to in relation to an instance, as instance variables were. You reference a class variable by using the Class name and the dot operator, like this:

```
total_house_value = House.total_value
```

Using the example of the `House` class again, a house's value was stored as an instance variable. This makes sense because each instance of the `House` class represents a different house, and each house will have its own value. The total value of all houses is a good example of a field that could be represented as a class variable. Each instance, or house, does not need to maintain its own copy of the total house value. This value is not a data element of any individual house, but rather a data element that describes all of the houses. Therefore, it makes sense to represent this value as a class variable.

Class variable names start with two `@` signs, `@@`. The class definition for the `House` class, including the total value class variable, would look like this:

```
class House
    @@total_value = 0

    def print_value
        puts @value
    end
```

```
    def set_value(a_value)
        @value = a_value
    end
end
```

In this code example, the `@@total_value` class variable is initialized to a value of zero. You must initialize class variables before they are used. To keep track of the total value of all houses, this variable must be updated every time a house value is updated. This requires a slight modification of the `set_value` method, like this:

```
class House
    @@total_value = 0

    def print_value
        puts @value
    end

    def set_value(a_value)
        @value = a_value
        @@total_value = @@total_value + @value
    end
end
```

Now, every time the value of a house is set, that value is also added to the total value of all houses, which is tracked with the `@@total_value` class variable. There is actually a potential problem with this code. Did you spot it? If the `set_value` method is called more than once for a single instance — that is, a single house — rather than updating the total value with the new value being set for that particular house, both values that you've set for that house are added to the total value. This gives a false total value. Having noted that, the code accurately illustrates the use of class and instance variables.

Getters and setters in Ruby objects

If you've done any amount of object-oriented program in a different language, you are probably familiar with the terms *getters* and *setters*. Even if you are not, the concept is relatively simple. As you've learned, an object instance contains data stored in instance variables. Frequent tasks that you will want to perform are setting the value of those variables and getting the value of those variables. The methods that perform those actions of setting and getting the values of instance variables are known as *getters* and *setters*.

Getters and setters are also sometimes referred to as accessors and mutators in some such as C++.

In many other object-oriented languages, you must explicitly define these getter and setter methods using relatively verbose and repetitive syntax. For example, in Java you might see code that looks like this in many of the classes:

```
Class JavaObject {
```

```
    String stringVal
    int intVal;

    public String getStringVal() {
        return stringVal;
    }

    public void setStringVal(String stringVal) {
        this.stringVal = stringVal;
    }

    public int getIntVal() {
        return intVal;
    }

    public void setIntVal(int intVal) {
        this.intVal = intVal;
    }
}
```

While these methods are relatively simple, this can be very tedious and perhaps error-prone if you make any typographic mistakes as you write these methods for every instance variable that you want to access outside of a class instance. These methods clutter up your class definitions with many lines of code that do relatively little. You've probably also noticed that the pattern for each instance variable getter and setter is the same. It seems that by writing all of these methods, you are doing a task that is more ideally suited for the computer. Isn't getting the computer to do work for you precisely the reason you are writing a software application in the first place?

Fortunately, Ruby saves you from having to repeat these getter and setter methods in all of your classes by giving you a built-in method that automatically generates the methods for you at run-time. Before you see that, however, it is educational to see how you would implement getters and setters in Ruby.

Getters in Ruby

Getters are relatively simple if you recall that methods in Ruby return the value of the last statement executed, even if you do not include a `return` statement. Therefore, the above Java class could be rewritten in Ruby like this (for the moment, you include only the getter methods, not the setters):

```
class RubyObject
    def string_val
        @string_val
    end

    def int_val
        @int_val
    end
end
```

In this Ruby code, notice that the instance variable names have been changed to reflect the style commonly used in Ruby code: lowercase variable names with words separated by underscores. Also notice that in Ruby, you do not have to declare the instance variables prior to using them.

Setters in Ruby

Let's take a look at how you implement setters in Ruby code. With your growing knowledge of Ruby code, your first attempt at creating a setter might look like this:

```
def set_string_val(new_string_val)
    @string_val = new_string_val
end
```

You would use this method to set the `@string_val` instance variable like this:

```
my_ruby_obj.set_string_val('a good string')
```

This method is valid Ruby code and will work just fine, but you can do better. Keep reading to see a more elegant way to express this setter method.

Using the equal sign in method names

Ruby allows you to define setter methods for the purpose of a more elegant setter method by using an equal (=) sign at the end of the method name. The following example illustrates how you would do this using an equal sign method:

```
def string_val=(new_string_val)
    @string_val = new_string_val
end
```

It doesn't look like you've saved much in terms of the definition of the setter method. Its size is similar, and some might even think this definition is a bit more complex. But, look at how this method is used:

```
my_ruby_obj.string_val=('a good string')
```

Here you see the new method being called just as the `set_string_val` method was called; however, by ending the method name with an equal sign, you begin to see how this makes the method call look less like calling a method and more like just setting the attribute value directly. Go a step further and remember that, in Ruby, you do not have to surround your parameters with parentheses. You can now write the setter like this:

```
my_ruby_obj.string_val='a good string'
```

Ruby lets you go even a step further by providing you with a syntax that is special to methods that end with the equal sign. You can write these methods with a space between the method name and the equal sign. For example, you could also write the setter like this:

```
my_ruby_obj.string_val = 'a good string'
```

When the Ruby interpreter sees the `string_val` method followed by the `=`, it automatically ignores the space and assumes you are calling the `string_val=` method. This line makes for a very readable setter method. This type of special syntax is often referred to by Ruby programmers as syntactic sugar.

The attr_ methods

Earlier, I said that you didn't really have to write your own getter and setter methods in Ruby, and I mentioned that there was a way to have these automatically generated for you at run-time. This is where the `attr_` methods come in. You will find yourself often using these methods.

The attr_reader method

Using the `attr_reader` method, you can avoid having to create getter methods for instance variables that you want to be readable from outside of your class. To use the `attr_reader` method, simply call it with a symbol representing the instance variable that you want a getter method for, like this:

```
class Message
    attr_reader :body

    def initialize(body)
        @body = body
    end
end
```

The `@body` instance variable is now readable from outside of the class by referencing it through an instance, like this:

```
a_message = Message.new("Dear John")
message_body = a_message.body
```

The `message_body` variable would now contain the value `"Dear John"`, which was the value of the `@body` instance variable.

The attr_writer method

Ruby also provides a method that will automatically create a setter method for you. The `attr_writer` method creates an accessor method to allow assignment to an attribute that is equivalent to a setter method. Using `attr_writer` is just as easy as `attr_reader` was. Take a look at this example.

```
class Message
    attr_writer :body
end

a_message = Message.new
a_message.body = 'I like school.'
```

In this example, you are able to set the `body` attribute of the `a_message` instance because an `attr_writer` was created for the `body` attribute. In practice, you will not use the `attr_writer` method very often. For attributes that you want to have both read and write access to, the `attr_accessor` method, described next, is a better choice.

The attr_accessor method

If you want to use both getters and setters with instance variables, the `attr_accessor` method is what you want to use. The `attr_accessor` method generates both getters and setters for the instance variables you pass to it.

```
class Message
    attr_accessor :body, :recipients
end
```

Now the `@body` and `@recipients` instance variables can be read and set from outside of the class. You get or set these instance variables just as if you were directly accessing the variable, like this:

```
a_message = Message.new
a_message.body = ''
a_message.recipients = ['tim@timothyfisher.com','john@doe.com']
```

You can see how easy this makes it to set instance variables without having to write any setter code inside the class.

Inheritance

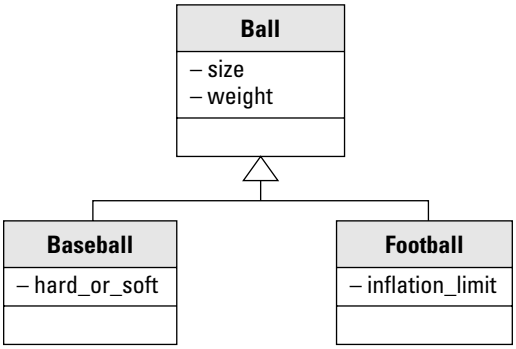
All object-oriented languages support inheritance. Inheritance is one of the ways in which classes can be related. You may often hear the term class hierarchy, or maybe object hierarchy. A *class hierarchy* is a hierarchical mapping of classes. *Inheritance* is the main building block of a class hierarchy, and specifically models the IS-A relationship. For example, a baseball IS-A ball. A football IS-A ball also.

Consider the example shown in Figure 1.2. You see the `Ball` class as a parent class of the `Baseball` and `Football` classes. The `Baseball` and `Football` classes will *inherit* all of the attributes and methods of the `Ball` class. In Figure 1.2, the Ball class has two attributes, a `size` and a `weight`. These attributes will be inherited by both the `Baseball` and `Football` classes. So instances of `Baseball` and `Football` will have size and weight attributes.

The `Baseball` and `Football` classes can also add their own attributes and methods to *specialize* the class to their particular type. Again referring to Figure 1.2, the Baseball class has a hard_or_soft attribute that is unique to the `Baseball` class. The `Football` class has an `inflation_limit` attribute that is unique to the `Football` class. These attributes specify things about the specific type of ball that the class represents that are not common to balls in general. Methods and attributes that are inherited from a parent class can be used in the child class just as if they were defined inside the child class. The parent class of an inheritance relationship is also commonly called the *base class*.

FIGURE 1.2

A class hierarchy



Inheritance is a technique that is very often used by object-oriented programmers. Rails applications rely very heavily on the use of inheritance. The classes that you write in a Rails application gain the power of the Rails framework primarily by inheriting from existing Rails classes. When you create a class that inherits from a parent class, you can also say that your new class *extends* the parent class. Referring back to Figure 1.2, the `Baseball` and `Football` classes extend the parent class, `Ball`.

To implement class inheritance in Ruby, you use the greater-than symbol (<) when you define your class, like this:

```
class Football < Ball
    …
end
```

This code means that the `Football` class is extending or inheriting from the `Ball` class.

Built-in Classes and Modules

Now that you have learned the basic syntax and structure of Ruby programming, it is time to learn about the built-in features of the language. Ruby contains a wealth of built-in capability that saves you from having to write a tremendous amount of low-level code in your applications.

The built-in classes and modules that you'll learn about in this section can be divided into two areas: scalar objects and collections. As you learn about these built-in features, it's a great idea to follow along with an open `irb` session. You can type all of the code snippets used in this section directly into `irb`. Lines in the code snippets that begin with => denote output that you will see in `irb` if you try out the code snippet.

Scalar objects

Scalar objects are objects that represent single values, as opposed to collections of values. In this section, you'll learn about some of the built-in scalar objects that you'll use often in Ruby programs. The scalar objects discussed here include the following:

- Strings
- Numerics
- Symbols
- Times and dates

Strings

Strings are used to represent text, or sequences of characters, in Ruby. You can create string literals in Ruby using single or double quotes, like this:

```
"This is a string in Ruby."
```

or like this:

```
'This is a string in Ruby.'
```

However, there are differences in how strings are interpreted that you should be aware of so that you can use the correct quote style in different situations.

Substitution in strings

Substitution occurs in strings when you type in one or more characters that the Ruby interpreter will change into different characters. A backslash character followed by another character is a common indicator for string substitution. For example, in a single quoted string, you can place a single quote inside of a string by escaping the quote with a backslash character, like this:

```
puts 'I went to Dad\'s house.'
```

This string outputs the string value:

```
I went to Dad's house.
```

The \' is turned into a single quote. This allows you to use single quotes within single-quoted strings. You can also use a backslash character within a single-quoted string by putting two backslashes in the string, like this:

```
puts 'A backslash looks like this: \\'
=> A backslash look like this: \
```

These are the only two substitutions that occur in single-quoted strings. Any other backslash characters remain just as you typed them.

Double-quoted strings, however, allow you to use a richer set of backslash sequences for substitution. For example, the \n sequence turns into a newline character in a double-quoted string.

String interpolation

String interpolation allows you to use Ruby expressions inside of double-quoted strings. Take a look at the following example:

```
subject = 'zombies'
   puts "Timmy likes #{subject}."
=> Timmy likes zombies.
```

The # and { sequences tell Ruby that what's enclosed is a Ruby expression that you would like evaluated, and its result is inserted into the string. In addition to using variables like this, you can also interpolate other expressions, such as this:

```
puts "If you add 2 and 5 you get the value #{2+5}."
=> If you add 2 and 5 you get the value 7.
```

You can skip the braces for instance, class, and global variables. For example, you could use interpolation with an instance variable like this:

```
@subject = 'cooking'
   puts "Camden likes #@subject."
=> Camden likes cooking.
```

String interpolation allows you to write concise code without having to perform a lot of string concatenation that you might do in other languages.

String operations

Ruby provides your strings with a great deal of built-in functionality. Here are some of the more common string methods that you'll use:

- `length`

  This method returns the length of the string that it is called on.

  ```
  short_str = "This is a short string."
  puts short_str.length
  => 22
  ```

- `include?`

  Returns `true` if the string it is called on contains the string passed as a parameter.

  ```
  "Superman can fly".include?('Superman')
  => true
  ```

- `slice`

  This method returns a substring of the string that it is called on. The substring is specified by passing an argument of one of the following types: `fixnum`, `range`, `regular expression`, or `string`. There is also a variant of this method that deletes the specified substring from the string that it is called on and returns the deleted substring. This variant is named `slice!`.

An example taken from the official Ruby documentation site at http://ruby-doc.org illustrates use of this function well:

```
string = "this is a string"
string.slice!(2)      #=> 105
string.slice!(3..6)   #=> " is "
string.slice!(/s.*t/) #=> "sa st"
string.slice!("r")    #=> "r"
string                #=> "thing"
```

In the above example, note that the end of the statements include a comment showing what the return value of that method call would be. For example, `string.slice!(2)` would return the ascii character value `105`. The string `#=>` is commonly used to indicate the return value of a statement in Ruby documentation.

The fourth line of the above example containing the expression `string.slice!(/s.*t/)` uses a regular expression as a parameter. In Ruby, regular expressions are created with the `/` delimiter. Ruby provides strong support for using regular expressions, and though this book does not get into the details of how to use regular expressions, I strongly advise you to become familiar with them. Regular expressions are very useful in any programming language.

- `gsub`

  This method allows you to specify a portion of a string to be replaced with a different string. Just as with the `slice` method, there is a variant available that will change the string that the method is called on. This variant is named `gsub!`. These methods take two parameters. The first parameter is a regular expression or a string to match on, and the second parameter is a string that you want to replace the matched text with. The following examples show how this method is used:

  ```
  "hello".gsub(/[aeiou]/, '*')        #=> "h*ll*"
  "Superman".gsub("Super", "Bat")     #=> "Batman"
  "hello".gsub(/([aeiou])/, '<\1>')   #=> "h<e>ll<o>"
  ```

  In the last line of the example, the replacement string is `'<\1>'`. In this string, the `\1` sequence will match the result of the `[aeiou]` regular expression. Surrounding the regular expression in parentheses, as in this example, creates a matching group. You could have additional matching groups in the regular expression by including additional regular expressions surrounded by more sets of parentheses. In the replacement string, you can match subsequent matching groups using `\2`, `\3`, and so on. See this line of code for an example of multiple matching groups:

  ```
  "hello".gsub(/(e)(ll)/, '<\1><\2>')      #=> "h<e><ll>o"
  ```

  In the previous line of code, the `\1` sequence matches the regular expression group `(e)` and the `\2` sequence matches the regular expression group `(ll)`.

There are many more methods that you can use on String objects. For a complete description of all of the methods available for String objects, you should refer to the official Ruby documentation for Strings at `www.ruby-doc.org/core/classes/String.html`.

Numerics

Ruby has special classes that represent numbers that you use in a Ruby application. These classes include `Float`, `Fixnum`, and `Bignum`. The `Bignum` and `Fixnum` classes represent integers. They both extend the `Integer` class. The classes `Float` and `Integer` extend the `Numeric` class, which provides basic functionality to all numeric objects.

You can find out the class that a particular number uses by calling its `class` method, like this:

```
1980.class
=> Fixnum

3.1459.class
=> Float

10000000000.class
=> Bignum
```

Now, take a look at a few methods that are commonly used with numbers:

- `integer?`

  Returns `true` if the number is an integer value.

  ```
  1980.integer?
  => true
  ```

- `round`

  Rounds the number to the nearest integer.

  ```
  18.3.round     #=> 18
  18.7.round     #=> 19
  ```

- `to_f`

  Converts a `Fixnum` or `Bignum` to a `Float`.

  ```
  15.to_f               #=> 15.0
  1000000000000.to_f    #=> 1000000000000.0
  ```

- `to_i`

  Converts a `Float` to an `Integer` type (either `Fixnum` or `Bignum` depending on its size). The decimal portion of the number is truncated. There is no rounding performed during the truncation.

  ```
  15.1.to_i      #=> 15
  15.8.to_i      #=> 15
  ```

- `zero?`

  Reruns true if the value has a zero value, otherwise it returns false.

There are many more methods available for the numeric classes. For complete method information, refer to the official Ruby doc Web site.

Symbols

If you are coming to Ruby from a Java or C language background, symbol objects are probably going to be something new to you. You can think of symbols as placeholders for strings. Symbols are easily recognized in Ruby code because they are always prefixed with a colon (:). You can convert any string into a symbol by using the `to_sym` method. The following is an example of creating a symbol using this method:

```
city = "Detroit"
city_sym = city.to_sym
puts city_sym
```

The `to_sym` method converts the string `"Detroit"` into an equivalent symbol object. The symbol, `city_sym`, contains the value `:Detroit`. When you print the symbol to the console using the `puts` method, the console output is:

```
Detroit
```

You might be wondering why the value printed was not `:Detroit`. The reason why you don't see that value printed is because the `puts` method automatically converts the symbol back into a string before printing it. The string equivalent of a symbol is the symbol value without the colon. However, if you look at the class type of the variable `city_sym`, you will see that it is indeed a Symbol object.

```
city_sym.class  #=> Symbol
```

You can convert a symbol back into a string using the `id2name` method. Here, you see how the `:Detroit` symbol is converted back to the original string value:

```
city_string = city_sym.id2name
```

When you get into Rails development, you will use symbols frequently. Although they may seem a bit foreign at first, they are simple to use and often make for cleaner and faster code.

Times and dates

In many applications you write, you will have to work with times and dates and usually perform some manipulation of those values. Ruby provides you with built-in classes to support times and dates in your application. The classes that provide Ruby's support for times and dates are `Date`, `Time`, and `DateTime`. The `Time` class is the only one of those three that is included with the Ruby core. The `Date` and `DateTime` classes are a part of the Ruby standard library which is included with Ruby but they must be explicitly included using a require statement when you want to use them. Below is an example of how you would include the `Date` and `DateTime` classes in your code:

```
require 'date'
```

The date library included using the above require statement will give you both the `Date` and the `DateTime` classes.

The `require` statements should always be placed at the very top of your source files. You can also use these classes within an `irb` session simply by using the same `require` syntax at the command prompt.

Below you'll see some of the methods and features of the date and time classes. There are many more methods available for these classes than what is covered in this book. For complete information, refer to the official Ruby documentation site `www.ruby-doc.org`.

Using the Time class

You can create instances of the Time class using the new method as shown here:

```
time = Time.new
```

When you use the new method of the `Time` class, an instance of the `Time` object representing the current time is created. To get a `Time` instance referring to the current time you can also use the `Time.now` method. If you want to create an instance that is preset to a given time, you the `Time.local` method as shown here:

```
time = Time.local(2008, "jun", 22, 10, 30, 25)
#=>  Sun Jun 10:30:25 -0400 2008
```

In this example, an instance of `Time` is created and set to the date June 22, 2008, and the time 10:30 and 25 seconds. The parameters passed to `Time.local` in this example are in this order year, month, date, hours, minutes, and seconds.

You can also call `Time.local` to create a time instance with these parameters: seconds, minutes, hour, day, month, year, day of the week, day of the year, is it daylight savings time?, and timezone. Here is an example of how you would create the same time using these parameters:

```
time = Time.local(25, 30, 10, 22, "jun", 2008, 0, 174, true,
   "EST")
#=>  Sun Jun 10:30:25 -0400 2008
```

Once you have a Time instance created, you can easily get specific field information from it using instance methods available. Here are some examples:

```
time.day        #=> 22
time.yday       #=> 174
time.wday       #=> 0
time.year       #=> 2008
time.month      #=> 6
time.zone       #=> "Eastern Daylight Time"
time.hour       #=> 10
time.min        #=> 30
time.sec        #=> 25
```

You can also perform addition and subtraction of time instances. To get the difference between two times, subtract them, as shown here:

```
time1 = Time.local(2008, "jun", 22, 10, 30, 25)
time2 = Time.local(2008, "jun", 20, 10, 30, 25)
time1 - time2   #=> 172800.0
```

The value returned from subtracting the two times is the time difference expressed in seconds. Knowing that there are 86,400 seconds in a day (60*60*24), you could convert the result to days by dividing the result by 86,400, to get two days.

You can add seconds to a time using the addition operator, as shown here:

```
time = Time.local(2008, "jun", 22, 10, 30, 25)
time + 60    #=>  Sun Jun 10:31:25 -0400 2008
```

In this example, 60 seconds are added to the time instance.

You can compare time instances using either the `eql?` method or the <=> operator. The `eql?` method will return true if the time that it is called on and the time passed to it are both `Time` objects with the same seconds and fractional seconds. The <=> operator also compares time objects down to the fractional seconds, however its return value is different. Instead of returning true or false, the <=> operator will return 0 if the time instances are equal, -1 if the time instance on the left occurs before the time instance on the right, and +1 if the time instance on the left occurs after the time instance on the right. Here are some comparison examples:

```
time1 = Time.local(2008, "jun", 22, 10, 30, 25)
time2 = Time.local(2008, "aug", 12, 10, 30, 25)
time3 = Time.local(2008, "jun", 22, 10, 30, 25)

time1.eql? time2       #=> false
time1.eql? time3       #=> true

time1 <=> time2        #=> -1
time2 <=> time1        #=> 1
time1 <=> time3        #=> 0
```

Using the Date class

To create a new `Date` instance, you use the `new` method, as you did with the `Time` class. However, unlike the `Time` class, when you create a date with the `new` method, you will not get the current date. To get a meaningful date instance you should pass parameters to the `new` method like this:

```
date = Date.new(2008, 3, 12)
```

This creates a date instance representing March 12, 2008. The `Date` class represents dates only and does not include time information. To see a string representation of the date, use the `to_s` method:

```
date.to_s       #=> "2008-03-12"
```

There are accessor methods provided for getting the year, month, and day components of the date:

```
date.year       #=> 2008
date.month      #=> 3
date.day        #=> 12
```

Another useful method is the `next` method. This method will return the next day, as shown here:

```
date.next.to_s  #=> "2008-03-13"
```

In the above example, the `next` method is chained with the `to_s` method to return the string representation of the next date. Method chaining can be a convenient way of writing concise expressions in Ruby. If you want to get the next month, or perform month addition, you can use the >> operator with the date instance. The >> operator will advance a date by the given number of months. Similarly, the << operator will subtract the given number of months from the date. Both of these operators will return the modified date but will not change the date instance on which they are called. Here are some examples:

```
(date >> 1).to_s        #=> "2008-04-12"
(date << 1).to_s        #=> "2008-02-12"
date.to_s               #=> "2008-03-12"
```

Just as with `Time` instances, you can test the equality of two dates using the `eql?` method, or the <=> operator. The operators behave just as they do for the Time instances, except dates are compared instead of times. Here are some examples:

```
date1 = Date.new(2008, 3, 12)
date2 = Date.new(2008, 7, 15)
date3 = Date.new(2008, 3, 12)

date1.eql? date2        #=> false
date1.eql? date3        #=> true

date1 <=> date2         #=> -1
date2 <=> date1         #=> 1
date1 <=> date3         #=> 0
```

Using the DateTime class

The `DateTime` class is a subclass of the `Date` class and thus inherits its methods and much of its behaviour. The `DateTime` class adds time information to the date information provided by the `Date` class.

You can create a DateTime instance with both date and time values set using the new method and parameters passed in this order (year, month, day, hour, minute, second) as shown here:

```
date_time = DateTime.new(2008, 3, 12, 10, 30, 25)
date_time.to_s   #=> "2008-03-12T10:30:25+00:00"
```

The DateTime class has access to these accessor methods for getting the time information:

```
date_time.hour   #=> 10
date_time.min    #=> 30
date_time.sec    #=> 25
date_time.zone   #=> "+00:00"
```

Formatting times and dates

All three of the time and date related classes, Time, Date, and DateTime, include a `to_s` method that allows you to get a string representation of the time or date. However, the format provided by the `to_s` method may not always be what you want. You can create a formatted date string using a format that you define using the `strftime` method that is available to all of these time and date classes. The `strftime` method takes a single parameter that is the format string. The format string can contain text and any of the format specifiers listed in Table 1.2 for printing date and time fields.

TABLE 1.2

Date and Time Formatting Codes for Use with strftime

Format Code	Description	Example
%a	The abbreviated weekday name	Sun
%A	The full weekday name	Sunday
%b	The abbreviated month name	Jan
%B	The full month name	January
%c	The preferred local date and time representation	03/12/08
%d	Day of the month	10
%H	Hour of the day, 24-hour clock	21
%I	Hour of the day, 12-hour clock	10
%j	Day of the year	215
%m	Month of the year	11
%M	Minute of the hour	25
%p	Meridian indicator	AM or PM
%S	Second of the minute	55
%U	Week number of the current year, starting with the first Sunday as the first day of the first week	5
%W	Week number of the current year, starting with the first Monday as the first day of the first week	4
%w	Day of the week (Sunday is 0)	2
%y	Year without a century	95
%Y	Year with century	1995
%Z	Time zone name	EST

Here are some examples of dates and times formatted using the `strftime` method:

```
date = Date.new(2008, 10, 18)
date.strftime("The day is %A, %B %d %Y")
#=> "The day is Saturday, October 18 2008

time = Time.local(2008, "jun", 22, 10, 30, 25)
time.strftime("Date: %a %b %d %Y, Time: %I:%M:%S")
#=> "Date: Sun Jun 22 2008, Time: 10:30:25"
```

Collections

All program languages support some method of representing groups of objects or other data elements. The objects that store collections of other objects are called the *collection objects*. These objects are defined by the collection classes, which are some of the most often used classes in any programming language. In almost any application you write, you will find times when you have to work with multiple items, and that is where collection classes help you.

Ruby provides you with built-in support for collections using the following collection classes, which you'll learn about in this section:

- Arrays
- Hashes
- Ranges

Arrays

The array is the most common collection class and is also one of the most often used classes in Ruby. An *array* stores an ordered list of indexed values, with the index starting at 0. Ruby implements arrays using the `Array` class. Here is an example of how arrays are used in Ruby:

```
great_lakes = ["Michigan","Erie","Superior","Ontario","Huron"]
puts great_lakes[0]
puts great_lakes[4]
```

This code creates an array containing the names of the Great Lakes, and stores it in the `great_lakes` variable. The second and third lines print the names of the first and fifth elements of the array. The output would be:

```
> Michigan
> Huron
```

Arrays do not have to be populated when they are created. You can also create an array object using the `Array.new` method, like this:

```
sports = Array.new
```

You can also create a new empty array using this style of declaration:

```
sports = []
```

The `Array` class also gives you plenty of built-in functionality. Here are some commonly used methods that you'll use when working with arrays:

- `empty?`

  Returns `true` if the array is empty.

  ```
  sports = Array.new
  puts sports.empty?
  => true
  ```

- `delete`

  Deletes the named element from the array and returns it.

  ```
  sports = ['Baseball','Football','Soccer']
  sports.delete('Soccer')
  sports
  => ['Baseball','Football']
  ```

- `first`

  Returns the first element of the array.

  ```
  names = ['Tim','John','Mike']
  puts names.first
  => Tim
  ```

- `last`

  Returns the last element of the array.

  ```
  names = ['Tim','John','Mike']
  puts names.last
  => Mike
  ```

- `push`

  Adds a new element to the array.

  ```
  sports = ['Baseball','Football','Soccer']
  sports.push('Tennis')
  => ['Baseball','Football','Soccer','Tennis']
  ```

- `size`

  Returns the number of elements contained in the array.

  ```
  sports = ['Baseball','Football','Soccer']
  puts sports.size
  => 3
  ```

Hashes

Like arrays, *hashes* store a list of values. However, if you use a hash instead of integer indexing, a hash lets you specify a unique index for each element that you store in the hash.

```
leagues = {"AL"=>"American League", "NL"=>"National League"}
puts leagues["AL"]
```

Once you have a hash, you can retrieve the value for an element in the hash by referencing its key value, as you see being done in the second line above. Notice that when you create a hash, you use the curly braces to enclose the hash, but when you refer to an element of the hash, you use the straight brackets. If you attempted to use the curly braces when referring to an element of the hash, you would get a syntax error.

You will also often hear the contents of a hash described as *key-value pairs*. The terms *index* and *key* are used interchangeably with respect to hashes.

Just as with arrays, the Hash class gives you plenty of built-in functionality. Here are some commonly used methods that you'll use when working with hashes:

- `empty?`

  Returns `true` if the hash is empty.

  ```
  leagues = {"AL"=>"American League", "NL"=>"National League"}
  puts leagues.empty?
  => false
  ```

- `keys`

  Returns an array of the hash's keys.

  ```
  leagues = {"AL"=>"American League", "NL"=>"National League"}
  leagues.keys
  => ['AL','NL']
  ```

- `values`

  Returns an array of the hash's values.

  ```
  leagues = {"AL"=>"American League", "NL"=>"National League"}
  leagues.values
  => ['American League','National League']
  ```

- `size`

  Returns the number of key or value pairs contained in the hash.

  ```
  leagues = {"AL"=>"American League", "NL"=>"National League"}
  leagues.size
  => 2
  ```

Ranges

Ruby provides another type of collection that you are probably not familiar with if you are new to Ruby: the `Range` class. You can use ranges to represent a sequence that has a defined start point, a defined end point, and a well-defined procession of elements. You create a range in Ruby using a start point, two dots, and an end point, like this:

```
(0..6)
```

This would create a range containing all the integer numbers from zero to six. You can verify that you have indeed created a `Range` object by looking at its class, using the following code:

```
(0..6).class
=> Range
```

A good way to verify what are all of the elements contained within a range is to convert the range into an array. You can convert the range into an array using the `to_a` method of the range, like this:

```
(0..6).to_a
=> [0,1,2,3,4,5,6]
```

You can use ranges not only for representing sequences of numbers, but also for representing any elements that have a well-defined sequence. Here is an example that expresses a sequence of letters as a range:

```
('a'..'e').to_a
=> ['a','b','c','d','e']
```

As with the other collection types in Ruby, you get plenty of built-in functionality with the `Range` class. Here are some common methods you can use with ranges:

- `first`

  Returns the first element of a range.

  ```
  (1..6).first  #=> 1
  ```

- `last`

  Returns the last element of a range.

  ```
  (1..6).last   #=> 6
  (1...6).last  #=> 6
  ```

  Notice that the last the last element specified in the Range declaration is returned as the last element of the range, even if that element is not included in the range, such as when you use the triple period range notation.

- `include?`

  Checks to see if the passed parameter value is included within the range.

  ```
  ('a'..'f').include? 'k'     #=> false
  ('a'..'f').include? 'd'     #=> true
  ```

  There is also a method available named `member?` that has the same behavior as `include?`.

- `each`

  This method allows you to iterate through each of the elements of a range and pass them to a block specified as a parameter. Blocks are covered later in this chapter, so if this doesn't make sense to you now, feel free to have another look after you've read about blocks.

```
(1..4).each do |number|
    puts number
end
```

Each element of the range 1, 2, 3, 4 will be printed to the screen on a separate line using the `puts` method.

- `step`

  Like the `each` method, the `step` method is also an iterator method. Using the `step` method, you can iterate through a range using a stepping size specified by the parameter passed.

  ```
  (1..6).step(2) do |number|
      puts number
  end
  ```

  This example will print out the numbers 1, 3, and 5 each on a separate line.

Control Flow

The control flow features of a programming language specify how the programming language allows you to control the path of execution through the code that you write. For example, there may be a group of statements that you only want to be executed under certain conditions, or there may be a group of statements that you want to repeat until a specified condition becomes true. These are the types of things that you will use control flow techniques to accomplish. Every programming language has control flow features built into it, and Ruby is no exception. Ruby's primary control flow mechanisms are:

- Conditionals
- Loops
- Blocks
- Iterators

Each of these mechanisms provides a different style of controlling the flow of your application. As you write more Ruby programs, you will find scenarios in which each of these mechanisms becomes valuable.

Conditionals

Conditionals allow you to specify a block of code that is executed conditionally, based on the result of some expression. Ruby supports three types of conditional statements:

- `if` statement
- `unless` statement
- `case` statement

The if statement

The `if` statement tests whether an expression is true or false. The expression being tested immediately follows the keyword `if` in a line of code. If the expression evaluates to true, the block of code following the `if` statement is executed. If the expression evaluates to false, the contained block of code is skipped.

In this example, the variable `value_a` is compared with the variable `value_b`. The statement `value_a is bigger` is only executed if the statement `value_a > value_b` is true.

```
if value_a > value_b
    puts 'value_a is bigger'
end
```

You can also specify a second block that is executed if the `if` expression evaluates to false. This is called the `else` block and is preceded by an `else` statement, like this:

```
if value_a > value_b
    puts 'value_a is bigger'
else
    puts 'value_b is bigger'
end
```

In this example, the correct statement is printed, depending on the values of the two variables.

There is one more statement that you can use with an `if` statement. That is the `elsif` statement. The `elsif` statement allows you to specify a block that is executed conditionally if the previous `if` or `elsif` statement did not evaluate to true. Here is an example:

```
if color == 'red'
    puts 'The color is red'
elsif color == 'blue'
    puts 'The color is blue'
else
    puts 'Could not determine color'
end
```

The unless statement

Another conditional supported by Ruby is the `unless` statement. The `unless` statement works opposite to how the `if` statement works. The block of code contained by the `unless` statement is executed only if the expression passed to the `unless` statement evaluates to false. Take a look at the following example:

```
unless value_a > value_b
    puts 'Value B is the larger number'
end
```

This code would print the message `'Value B is the larger number'` only if the value stored in `value_b` is larger than the value stored in `value_a`.

The case statement

The `case` statement allows you to compare a variable to a number of different possible values and execute a group of methods based on which of the values it matches. This construct can replace a series of `if..else` statements. Consider the following block of `if..else` statements:

```
if  color == 'red'
    puts 'The color is red'
elsif color == 'blue'
    puts 'The color is blue'
elsif color == 'green'
    puts 'The color is green'
else
    puts 'Unrecognized color name.'
end
```

In this series of `if..else` statements, the color variable is compared against a series of different values to find one that matches. If it does not find a match, there is an ending `else` to print a default message. This example illustrates how you could implement the very same logic using a `case` statement:

```
case color
    when 'red'
        puts 'The color is red'
    when 'blue'
        puts 'The color is blue'
    when 'green'
        puts 'The color is green'
    else
        puts 'Unrecognized color name.'
end
```

As you see here, after the `case` statement, you specify the variable that you want to match. Each `when` statement is the equivalent of an `elseif` in the previous implementation. When a matching condition is found, the statement or statements following that `when` statement (up until the next `when` statement) are executed. After executing those statements, the control flow passes to the line after the `case` statement's closing `end` statement.

You can also specify groups of valid values, as in the following example:

```
when 'red','purple'
```

Loops, blocks, and iterators

Loops, blocks, and iterators allow you to define sections of code that you want to execute repeatedly, often until a given condition is satisfied. The constructs you'll learn about here include:

- `for` loops
- `while` and `until` loops
- code blocks

If you have experience with other programming languages, you are probably familiar with the concept of `for`, `while`, and `until` loops. However, code blocks may be very new to you. They are a feature that gives Ruby a great deal of its unique power and capability for writing clean, elegant, and concise code.

for loops

The Ruby `for` loop allows you to execute a given block of code an amount of times specified by an expression preceding the block. If you are used to using the for loops in Java, JavaScript, C, C++, or a language similar to one of those, pay particular attention here, as the Ruby for loops are different than the for loops in those languages. Here is an example of a Ruby `for` loop:

```
cities = ['Southgate','Flat Rock','Wyandotte','Woodhaven']
for city in cities
    puts city
end
```

Executing this loop would result in each of the city names contained in the `cities` array being printed to the console. Here is another example of a `for` loop that iterates over a hash variable, using both the key and value elements as variables within the block.

```
hash = {:r=>'red', :b=>'blue, :y=>'yellow'}
for key,value in hash
    puts "#{key} => #{value}"
end
```

Executing this loop will result in the following output:

```
y => yellow
b => blue
r => red
```

while and until loops

In addition to the `for` loop, Ruby supports other looping constructs that are also common in many other programming languages: the `while` loop and the `until` loop. The `while` and `until` loops execute a block of code while a certain condition is true, or until the condition becomes true. Here are some examples:

```
num = 10
while num >= 0 do
    puts num
    num = num - 1
end

num = 0
until num > 10 do
    puts num
    num = num + 1
end
```

Blocks

In several of the previous examples that used iterators, such as the `each` or `step` method of a `Range` object, you have seen Ruby blocks in use. *Blocks* are groups of statements that can be passed into a method as a parameter. They are commonly used with iterators. The `each` method, which is available on any class that is enumerable in Ruby, is probably the place you will use blocks most often. Here is an example of a block used with the `each` statement:

```
colors = ['red','blue','yellow','green']
colors.each do |color|
    puts color
    color_count = color_count + 1
end
```

In this example, the block is enclosed by the `do` and `end` statements. The block is passed a single parameter which is enclosed in the pipes. The block is passed as a parameter to the `each` method. Blocks can also be enclosed by curly brackets. The example below is equivalent to the previous one:

```
colors = ['red','blue','yellow','green']
colors.each { |color|
    puts color
    color_count = color_count + 1
}
```

Although not a syntax rule, common usage is to use the curly brackets around blocks when you have a short block that will fit on the same line as the method invocation to which the block is passed, such as this example:

```
colors.each { |color| puts color}
```

If your block spans multiple lines, the `do/end` syntax is preferred.

Blocks are a construct that is new to many programmers, especially those coming from Java or C language backgrounds. They are frequently used in Ruby code so you should become very familiar with them. I have just touched on what you can do with blocks. There is a great deal more to learn about them. You can learn more with many good online references; just do a Google search on Ruby Blocks.

The yield statement

You can create your own methods that accept blocks as a parameter and be able to pass parameters into those blocks using the `yield` statement. Take a look at an example of a method that can accept a block as a parameter:

```
class TimsBooks
    def initialize
        @books = ['Ruby on Rails Bible', 'Java Phrasebook']
    end

    def each
        @books.each {|book| yield book }
```

```
    end
end

books = TimsBooks.new
books.each do |book|
    puts book
end
```

In this example, the `TimsBooks` class contains an instance variable that is an array of books. The `@books` variable is initialized at object creation time. The `each` method is implement to iterate through the `@books` array and yield each book value to the block that is passed to the `each` method. Toward the bottom of the example, you see how the `each` method can be used with an instance of `TimsBooks` to print the name of each book. Using this technique you could write your own `each` methods for any classes that you write that contain some data that can be iterated upon.

The `yield` statement calls the passed in block, passing any parameters that are passed to it along to the block. So in the above example, each time `yield` is called, the block containing the `puts book` statement is called passing the name of a book from the `@books` array. The resulting output will be a list of the books in the `@books` array.

Iterators

An *iterator* is a method that allows you to step through a group of values in a systematic way. Iterators are featured in many programming languages, and Ruby has rich support for them. You have seen some of the iterator methods already. Some of the iterator methods supported by Ruby described here.

- `each`

  The `each` method is the most common iterator. You can use the each method to step through any element that is enumerable such as an array or hash.

  ```
  students = ['Tim','Camden','Kerry','Timmy']
  students.each do |student|
      puts student.name
  end
  ```

- `times`

  The `times` method is an iterator used on integer values. It is used to repeatedly execute a block of code.

  ```
  3.times {puts 'Ruby rules'}
  ```

  This will print the line `'Ruby Rules'` three times.

- `map`

  The `map` method is commonly used with `Array` objects. It calls the passed block once for each element of the array on which it is called. Its return value is a new array containing each of the values returned by the subsequent calls to the block.

  ```
  [1,2,3].map {|x| x * x}
  #=> [1,4,9]
  ```

This example returns an array that contains the squares of each of the elements contained in the original array.

- `upto`

  The `upto` method is an iterator used with elements that have some form of ordering associated with them. Common examples of where you can use this method include integers and alphabetic characters as shown below:

```
4.upto(7) {|x| puts x}

'a'.upto('c') {|char| puts char}
```

  In the first example above, the values 4, 5, 6, and 7 are printed. In the second example, the characters a, b, and c are printed.

Exception handling

Every good developer should be familiar with error handling techniques and know how to handle errors that occur in a program. No matter how well you have written and tested your program, there will always be error conditions that occur in your program. These error conditions are not always the fault of the developer, but could be triggered by a number of things, including bad input from an external component , unavailable external resources, or incorrect usage by the end user

Before OOP became popular, error handling was mostly accomplished using return values and error codes. All of your functions would return a value that would indicate whether the function succeeded or failed. On failure, the return value would contain an error code or perhaps an error message. Unfortunately, this style of programming tends to require error-handling code around all of your functions and within the functions. Often, the purpose of a particular function is lost in so much error-handling code.

Object-oriented languages introduced a new style of error handling with a more object-oriented approach. This style uses *exception* objects that can be *thrown* and *caught* by your code and handled where appropriate. This style of error handling is usually referred to as *exception handling*. The exception handling features of Ruby allow you to handle unexpected conditions that occur while your code is running.

Exceptions in Ruby

In Ruby when an exceptional condition occurs, you can raise an exception using either the `raise` statement or the `throw` statement. When you raise an exception, control flow is diverted away from the current context to exception handling code. Exceptions that are raised can be caught with a rescue block. Rescue blocks are created with the `rescue` statement. Exceptions are represented as `Exception` objects. `Exception` objects are instances of the `Exception` class or a subclass of the `Exception` class. Ruby includes a hierarchy of built-in exception classes. There are seven classes that are direct subclasses of `Exception`. These are the following:

- `NoMemoryError`
- `ScriptError`

- SecurityError
- SignalException
- SystemExit
- SystemStackError
- StandardError

The StandardError exception class represents exceptions that are considered normal and that you should attempt to handle in your application code. The other exception classes represent lower-level and more serious errors that you most likely will not be able to recover from. Most programs do not attempt to handle these exception classes. There are many built-in subclasses of StandardError, and you are free to also create your own subclasses to define custom exceptions for your application.

The Exception class defines two methods that will help you get more information about the problem that occurred. These two methods should be implemented by all of its subclasses. The two methods are message and backtrace. The message method returns a string that gives human-readable information about the cause of the exception. The backtrace method returns an array of strings that represent the call stack at the point the exception was raised.

Using begin, raise, and rescue

The three statements that are used most often to perform exception handling in Ruby are the raise, begin, and rescue statements. The raise statement is used to create, or throw, and exception. You can call raise with zero, one, two, or three arguments. If you use raise with no arguments, a RuntimeError object is raised. If you use one argument with raise, one of the following conditions will apply:

- If the single argument is an Exception argument, that exception is raised.
- If the argument is a string, a RuntimeError is raised and the string is set as its message.
- If the argument is an object that has an exception method, that method should return an Exception class. The Exception class returned will be raised.

If you use raise with two arguments, the second argument should be a string that will get set as the message of the exception defined by the first argument. Finally, you can call raise with three arguments also. In that case, the first argument will define an exception class, the second argument will define a string to be set as the exception's message, and the third argument will contain an array of strings which will be set as the backtrace for the exception object.

Here is an example of how you might raise a RuntimeError exception with a specified message:

```
raise RuntimeError, "Bad value used."
```

The begin statement designates the start of a block of code for which you want to apply exception handling. The rescue statement specifies the start of a block of code that is executed if an exceptional condition occurs within the block of code that began with the begin statement. To

illustrate the uses of exception handling in Ruby, you'll see how exception handling is commonly used along with Ruby's built-in file support to catch errors that might occur when you are trying to open a file.

```
def read_file(file_name)
    begin
        afile = File.open(file_name, "r")
        buffer = afile.read(512)
    end

    rescue SystemCallError
        # handle error
    end

    rescue StandardError
        # handle error
    end

    rescue
        # default exception handler
    end
end
```

This method attempts to open a file with the name you pass into the method, and to read the first 512 bytes from it. An exception can be raised from within either the `File.open` or the `afile.read` methods. If an exception is raised within either of those methods, the control flow of the code will jump out of the `begin` block. The block that begins with the code `rescue SystemCallError` will be executed if a `SystemCallError` exception is raised. If the exception raised is a `StandardError` exception, the block that rescues `StandardError` will be executed. If the exception thrown is neither of those two types, the default exception handling block will be executed (this is the `rescue` block that does not specify a parameter).

As you saw in the previous example, a `rescue` block can specify a specific type of exception to handle, or not specify an exception type at all. If no exception type is specified, the block will handle any exception type that has not been handled by a previous `rescue` block. You can specify more than one exception type for a `rescue` block to handle also. For example, if you wanted to handle `SystemCallError` and `StandardError` the same way, you might write an exception handler like this:

```
rescue SystemCallError, StandardError
    # handle error
end
```

In many cases, you will want to get information about the exception that occurred in the rescue block that handles it. You can access the exception object by defining a rescue block like this:

```
rescue => ex
    puts "#{ex.class}: #{ex.message}"
end
```

In the above example, the exception object is stored in the `ex` variable. You can access any of the exception's methods through the `ex` variable. If your rescue clause is for a specific type of exception, the syntax to get the exception object would look like this:

```
rescue ArgumentError => ex
    puts "#{ex.class}: #{ex.message}"
end
```

More exception handling using ensure, retry, and else

Now that you have the basics of Ruby exception handling down, let's look at three additional statements that are part of Ruby's exception handling support. These are the `ensure`, `retry`, and `else` statements.

The retry statement

If you put a `retry` statement inside of a rescue block, the block of code that the rescue block is attached to will be run again. This is a good option for errors that are likely to resolve themselves. For example, if the load on a server was too high when you called it the first time, if you wait a bit and attempt the call again, it may succeed. The following code illustrates that scenario:

```
network_access_attempts = 0
begin
    network_access_attempts += 1
    open('http://www.timothyfisher.com/resource') do |f|
        puts f.readlines
    end
rescue OpenURI::HTTPError => ex
    if (network_access_attempts < 4)
        sleep(100)
        retry
    else
        # handle error condition
    end
end
```

In the begin block of this code, it attempts to open a network resource. If an exception is thrown while attempting to open that resource, the `rescue` block will be executed. Within the `rescue` block, we check to see if we have attempted to access the resource less than four times. If so, the code sleeps for 100 mS and then uses the `retry` statement to retry the `begin` block. If the same exception occurs four times, we give up and attempt to handle the error.

The else statement

A `begin-rescue` code block may also include an `else` block. The `else` block will be executed if the code in the begin block completes without raising any exceptions. Below is an example of how you might use an `else` block:

```
begin
    network_access_attempts += 1
```

```
    open('http://www.timothyfisher.com/resource') do |f|
        puts f.readlines
    end
rescue => ex
    puts 'Error reading file'
    puts "#{ex.class}: #{ex.message}"
else
    puts 'Successfully read the entire remote file'
end
```

If any exceptions are raised in the `else` block, they are not caught by any of the `rescue` statements attached to the `begin` block.

The ensure statement

The `ensure` statement is used to start a block that will always be executed, no matter what happens in the preceding `begin` block. The `ensure` block will be run after the `begin` block completes, or after a `rescue` statement completes if the `begin` block resulted in an exception. If the code also contains an `else` block, the `else` block will be run before the `ensure` block. The `ensure` block will always be the last block run. If control is transferred away from the `begin` block before it completes, perhaps by using a `return` statement, the `ensure` block will still be run, however the `else` block would not be run in that case. An `else` block is only run if the `begin` block runs to completion. An `ensure` block is always run no matter what happens in the `begin` block. Here is an example of exception handling code that uses an `ensure` block:

```
begin
    file = open("/some_file", "w")
    # write to the file
rescue => ex
    puts 'Error writing file'
    puts "#{ex.class}: #{ex.message}"
else
    puts 'Successfully updated file'
ensure
    file.close
end
```

In this example, the code opens a file and would then attempt to write to that file. If an exception occurs, the exception is printed to the screen. If the write completes successfully, a success message is printed to the screen using the `else` block. In either case, the `ensure` block runs to make sure that the file gets closed.

The normal use of an `ensure` block is to ensure that your code performs necessary housekeeping tasks, such as closing files, close database connections, or completing database transactions. Unless an `ensure` block contains an explicit `return` statement, it will not affect the return value of your method. For example, in the following code, the value returned will be `hello` and not `goodbye`. If you're wondering why `hello` is used as a return value, recall that the last value of a method is also the value that gets returned. The `ensure` block will not overwrite that return value.

```
begin
    'hello'
ensure
    'goodbye'
end
```

Organizing Code with Modules

One of the most commonly touted benefits of object oriented programming is that it can result in more reusable code. You can use reusable code in multiple applications, and it saves developers time and money. Organizing your code into classes and separating your classes into different files is one way of creating reusable chunks of code. Often, though, you may have a situation where you have a bunch of methods that don't naturally fall into a specific class, and yet they are methods that you find yourself using again and again, perhaps in many of your classes. This is where Ruby's concept of a module can help you out.

A *module* in Ruby provides a namespace that allows you to group methods and constants together, similar to the way a class groups methods and attributes. A Ruby module definition looks like this:

```
module Messaging
    def send_email
        ..
    end

    def send_im
        …
    end

    def send_text_message
        …
    end
end
```

This creates a `Messaging` module that bundles together methods related to sending a message over various protocols, e-mail, instant messaging, or text messaging. Any place where you wanted to use these methods, you could include this module as a mixin.

In addition to providing a convenient namespace and place to put methods and constants that do not fall naturally into a class definition, modules also give you the ability to use mixins. The Ruby concept of a *mixin* is a way of including methods and constants defined in a module into another module or class. Previously you saw how to define a Messaging module. Now if you have a `Notifier` class that you want to use these methods in, you would simply include this module like this:

```
require 'messaging'

class Notifier
```

```
    include Messaging
    …
end
```

The `Notifier` class uses a `require` statement to import the file containing the `Messaging` module. This example assumes that the module is stored in a file contained in the same directory as the `Notifier` class, with a filename `messaging.rb`. The `include` statement imports all of the methods contained in the `Messaging` module into the `Notifier` class.

Perhaps the most common examples of mixins are the `Enumerable` and `Comparable` modules that are included with Ruby. These modules are mixed into quite a few classes by default, and you can easily mix them into your own classes as well. The `Enumerable` module defines useful iterators for any class that defines an `each` method. It is important to remember that the `Enumerable` module does not define the `each` method. You must define the `each` method in any class that you include the `Enumerable` module into. `Enumerable` defines methods such as `all?`, `any?`, `collect`, `find`, `find_all`, `include?`, `inject`, `map`, and `sort`. See the Ruby documentation Web site for a complete description of the methods of the `Enumerable` module `www.ruby-doc.org/core/classes/Enumerable.html`.

The `Comparable` module defines general comparison methods for any class that defines the `<=>` method. You can include the `Comparable` module into any class for which you have defined the `<=>` method. The `Comparable` module defines methods that look like operators such as: `<`, `<=`, `==`, `>`, and `>=`.

Advanced Ruby Techniques

In this section, you'll learn some additional techniques that will be useful to you when you are writing and studying Rails programs. The techniques described in this section are also used internally by Rails.

Variable length argument lists

All of the method examples that you've seen so far in this chapter have used fixed argument lists. Ruby also supports variable length argument lists. A method that allows a variable length argument list lets you call it with different numbers of methods in different situations. Take a look at the following example:

```
def print_strings(*strings)
    strings.each { |str| puts str }
end
```

This is a method that will accept a variable number of arguments. The `strings` variable contains an array holding all of the arguments that are passed to this method. In the body of a method, the `each` iterator is used to step through each of the strings passed in and to print its value.

Dynamic programming with method_missing

The `method_missing` method is a feature of Ruby that you will find very useful in certain situations. Before you get into the details of that, though, let's talk about what is meant by the term *dynamic programming*. Dynamic programming is a style of programming in which you create code or change the nature of your program's code at run-time.

If you attempt to call a method that does not exist for the object you are using it on, you normally get an undefined method error. For example, try typing this code in `irb`:

```
class EmptyClass
end

obj = EmptyClass.new
obj.say_hello
```

In this code, you are attempting to call the method `say_hello` on an instance of the `EmptyClass`. Because this method does not exist, you will see an error message like the following printed to the console:

```
NoMethodError: undefined method 'say_hello' for
   #<EmptyClass:0x28f7d64>
    from (irb):31
    from :0
```

Here, `irb` is telling you that it cannot find this method in your class. Go ahead and exit that `irb` session to clear its memory and restart `irb`. Recreate the `EmptyClass`, slightly modified, as shown here:

```
class EmptyClass
    def method_missing(method, *args)
        puts 'Sorry, I could not find the method you are
   calling.'
        Puts "The method you called is #{method}."
    end
end

obj = EmptyClass.new
obj.say_hello
```

Now when you call the `say_hello` method in `irb`, you see this output:

```
Sorry, I could not find the method you are calling.
The method you called is say_hello.
```

As you can see, because the method you called could not be found in the `EmptyClass`, the `method_missing` method was called. The `method_missing` method is called by Ruby anytime you try to call a method that does not exist. The name of the method, and any arguments that you passed to the method you were trying to call, are also passed to the `method_missing` method.

Reopening classes

In Ruby, no class definition is ever final. You can reopen the definition of any Ruby class, including classes that you previously defined, even classes that are built into Ruby, and modify those class definitions to change the behavior of those classes.

Let's look at an example where you will reopen a commonly used built-in Ruby class, the `String` class. Try this out by typing the following code into an `irb` session:

```
class String
   def reverse_and_capitalize
      self.reverse.capitalize
   end
end
```

You've added a new instance method named `reverse_and_capitalize` to the `String` class. This method combines the features of the built-in `reverse` and `capitalize` methods. The `reverse_and_capitalize` method is now available on any string that you create. Try it out:

```
str = "say hello"
str.reverse_and_capitalize
=> "Olleh yas"
```

You created a string object the normal way and called the new method that you added. Your method is now a part of the `String` class, just like any other method that you use with the `String` class. In addition to adding methods, you could also redefine a method by reopening the class.

You can use this technique to extend external libraries that you use, as well as the built-in Ruby classes.

CAUTION **Developers have expectations from commonly used methods, and if you change the behavior of those methods, you must make sure that it is well documented and everyone who uses your modification is aware of those changes.**

Summary

This chapter has provided you with a basic overview of the Ruby programming language. While what it provided is far from a complete overview of Ruby, it should be more than enough to get you started writing Rails applications, which is the ultimate goal of this book.

As you begin writing Rails applications and as you gain more experience with both Ruby and Rails, your Ruby skills will increase, and I am certain you will seek out additional resources to further enhance your Ruby programming skills. *Programming Ruby: The Pragmatic Programmers' Guide* is often referred to as the Ruby Bible (also commonly called the pickaxe book because of the image of a pickaxe depicted on its cover) and is probably a book that you will want to own at some point. This book is written by a Ruby pioneer, Dave Thomas, and was one of the first Ruby language books published in the United States. It remains the most referenced and most used Ruby language book.