Table of Contents

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Chapter 1. Ruby in Review

IN THIS CHAPTER

Language shapes the way we think and determines what we can think about.
—Benjamin Lee Whorf

It is worth remembering that a new programming language is sometimes viewed as a panacea, especially by its adherents; however, there is no one language that will supplant all the others, no one tool that is unarguably the best for every possible task. There are many different problem domains in the world, and there are many possible constraints on problems within those domains.

Above all, there are different ways of *thinking* about these problems, stemming from the diverse backgrounds and personalities of the programmers themselves. For these reasons, no foreseeable end to the proliferation of languages is in site. As long as there is a multiplicity of languages, there will be a multiplicity of personalities defending and attacking them. In short, there will always be "language wars." In this book, however, we do not intend to participate in them.

Yet, in the constant quest for what is newer and better program notations, we have stumbled across ideas that endure, that transcend the context in which they were created. Just as Pascal borrowed from Algol, just as Java borrowed from C, so will every language borrow from its predecessors. A language is both a toolbox and a playground; it has its extremely practical side, but it also serves as a test bed for new ideas that may or may not be widely accepted by the computing community.

One of the most far reaching of these ideas is the concept of object-oriented programming (OOP). Although many would argue that the overall significance of OOP is evolutionary rather than revolutionary, no one can say that it has not had an impact on the industry. Twenty years ago, object orientation was for the most part an academic curiosity; today it is a universally accepted paradigm.

In fact, the ubiquitous nature of OOP has led to a significant amount of "hype" in the industry. In a classic paper of the late 80s, Roger King observed, "If you want to sell a cat to a computer scientist, you have to tell him it's object-oriented." Additionally, there are differences of opinion about what OOP really is, and even among those who are essentially in agreement there are differences in terminology.

It is not our purpose here to contribute to the hype. We do find OOP to be a useful tool and a meaningful way of thinking about problems; however, we do not claim that it cures cancer. As for the exact nature of OOP, we have our pet definitions and favorite terminology, but we make these known only in order to communicate effectively, not to quibble over semantics.

We mention all this because it is necessary to have a basic understanding of OOP in order to proceed to the bulk of this book and understand the examples and techniques. Whatever else might be said about Ruby, it is definitely an object-oriented language.

Some Words on Object Orientation

Before talking about Ruby specifically, it is a good idea to talk about object-oriented programming in the abstract. These first few pages will provide a review of those concepts with only cursory references to Ruby, before we proceed in a few pages to the review of the Ruby language itself.

In object-oriented programming, the fundamental unit is the *object,* which is an entity that serves as a container for data and also controls access to the data. Associated with an object is a set of *attributes,* which are essentially no more than variables belonging to the object. (In this book, we will loosely use the ordinary term *variable* for an attribute.) Also associated with an object is a set of functions that provide an interface to the functionality of the object. These functions are called *methods.*

It is essential that any OOP language provide *encapsulation.* As the term is commonly used, it means first that the attributes and methods of an object are associated specifically with that object or bundled with it. Secondly, it means that the scope of those attributes and methods is by default the object itself (an application of the well-known principle of *data hiding,* which is not specific to OOP).

An object is considered to be an instance or manifestation of an *object class* (usually simply called a *class*). The class may be thought of as the blueprint or pattern; the object itself is the thing created from that blueprint or pattern. A class is often thought of as an *abstract type*—a more complex type than, for example, an integer or character string.

When an object (an instance of a class) is created, it is said to be *instantiated.* Some languages have the notion of an explicit *constructor* and *destructor* for an object—functions that perform whatever tasks are needed to initialize an object and, respectively, to "destroy" it. We may as well mention prematurely that Ruby has what might be considered a constructor but certainly does not have any concept of a destructor (because of its well-behaved garbage-collection mechanism).

Occasionally a situation arises in which a piece of data is more "global" in scope than a single object, and it is inappropriate to put a copy of the attribute into each instance of the class.

For example, consider a class called `MyDogs`, from which three objects are created: `fido`, `rover`, and `spot`. For each dog, there might be such attributes as age and date of vaccination. But suppose we want to store the owner's name. We could certainly put it in each object, but that is wasteful of memory and at the very least a misleading design. Clearly the `owner_name` attribute belongs not to any individual object but rather to the class itself. When it is defined that way (and the syntax will vary from one language to another), it is called a *class attribute* (or *class variable*).

Of course, there are many situations in which a class variable might be needed. For example, suppose we want to keep a count of how many objects of a certain class have been created. We could use a class variable that was initialized to zero and incremented with every instantiation; the class variable would be associated with the class and not with any particular object. In scope, this variable would be just like any other attribute, but there would only be one copy of it for the entire class and the entire set of objects created from that class.

To distinguish between class attributes and ordinary attributes, the latter are sometimes explicitly called *object attributes* (or *instance attributes*). We will use the convention that any attribute is assumed to be an instance attribute unless we explicitly call it a class attribute.

Just as an object's methods are used to control access to its attributes and provide a clean interface to them, so is it sometimes appropriate or necessary to define a method that is associated with a class. A *class method,* not surprisingly, controls access to the class variables and also performs any tasks that might have class-wide effects rather than merely object-wide effects. As with data attributes, methods are assumed to belong to the object rather than the class, unless stated otherwise.

It is worth mentioning that there is a sense in which all methods are class methods. We should not suppose that when a hundred objects are created, we actually copy the code for the methods a hundred times! However, the rules of scope assure us that each object method operates only on the object whose method is being called, providing us with the extremely necessary illusion that object methods are associated strictly with their objects.

We come now to one of the real strengths of object-oriented programming: inheritance. *Inheritance* is a mechanism that allows us to extend a previously existing entity by adding features to create a new entity. In short, inheritance is a way of reusing code. (Easy, effective code reuse has long been the Holy Grail of computer science, resulting in the invention decades ago of parameterized subroutines and code libraries. OOP is only one of the later efforts in realizing this goal.)

Typically we think of inheritance at the class level. If we have a specific class in mind and there is a more general case already in existence, we can define our new class to inherit the features of the old one. For example, suppose we have the class `Polygon`, which describes convex polygons. If we then find ourselves wanting to deal with the `Rectangle` class, we can inherit from `Polygon` so that `Rectangle` now has all the attributes and methods that `Polygon` has. For example, there might be a method that would calculate perimeter by iterating over all the sides and adding their lengths. Assuming everything is implemented properly, this method would automatically work for the new class; the code would not have to be rewritten.

When class `B` inherits from class `A`, we say that `B` is a *subclass* of `A`—or conversely, `A` is the *superclass* of `B`. In slightly different terminology, we may say that `A` is a*base class* or *parent class,* and `B` is a *derived class* or *child class.*

A derived class, as you have seen, may treat a method inherited from its base class as if it were its own. On the other hand, it may redefine that method entirely, if it is necessary to provide a different implementation; this is referred to as *overriding* a method. In addition, most languages provide a way for an overridden method to call its namesake in the parent class; that is, the method `foo` in `B` knows how to call method `foo` in `A` if it wants to. (Any language not providing this feature is under suspicion of not being truly object oriented.) Essentially the same is true for data attributes.

The relationship between a class and its superclass is an interesting and important one; it is usually described as the *is-a* relationship, because a `Square` "is a" `Rectangle`, and a `Rectangle` "is a" `Polygon`, and so on. Therefore, if we create an inheritance hierarchy (which tends to exist in one form or another in any OOP language), we see that the more specific entity "is a" subclass of the more general entity at any given point in the hierarchy. Note that this relationship is transitive—in the preceding example, you can easily see that a `Square` "is a" `Polygon`. Note also that the relationship is not commutative—we know that every `Rectangle` is a `Polygon`, but not every `Polygon` is a `Rectangle`.

This brings us to the topic of *multiple inheritance.* It is conceivable that there might be more than one class from which a new class could inherit. For example, the classes `Dog` and `Cat` can both inherit from the class `Mammal`, and `Sparrow` and `Raven` can inherit from `WingedCreature`. But what if we want to define the class `Bat`? It can reasonably inherit from both `Mammal` and `WingedCreature`. This corresponds well with our experience in real life, in which things are not members of just one category but of many non-nested categories.

Multiple inheritance (MI) is probably the most controversial area in OOP. One camp will point out the potential for ambiguity that must be resolved. For example, if `Mammal` and `WingedCreature` both have an attribute called `size` (or a method called `eat`), which one will be referenced when we refer to it from a `Bat` object? Another related difficulty is the "diamond inheritance problem" (so called because of the shape of its inheritance diagram), with both superclasses inheriting from a single common superclass. For example, imagine that `Mammal` and `WingedCreature` both inherit from `Organism`; the hierarchy from `Organism` to `Bat` forms a diamond. But what about the attributes that the two intermediate classes both inherit from their parent? Does `Bat` get two copies of each of them, or are they merged back into single attributes because they come from a common ancestor in the first place?

These are both issues for the language designer rather than the programmer. Different OOP languages deal with the issues in different ways. Some will provide rules allowing one definition of an attribute to "win out," or a way to distinguish between attributes of the same name, or even a way of aliasing or renaming the identifiers. This in itself is considered by many to be an argument against MI—the mechanisms for dealing with name clashes and the like are not universally agreed upon but are very much language dependent. C++ offers a fairly minimal set of features for dealing with ambiguities; those of Eiffel are probably better, and those of Perl are different from both.

The alternative, of course, is to disallow MI altogether. This is the approach taken by such languages as Java and Ruby. This sounds like a drastic compromise; however, as you'll see later, it is not as bad as it sounds. We will look at a viable alternative to traditional multiple inheritance, but we must first discuss yet another OOP buzzword: polymorphism.

Polymorphism is the term that perhaps inspires the most semantic disagreement in the field. Everyone seems to know what it is, but everyone has a different definition. (In recent years, "What is polymorphism?" has become a popular interview question. If it is asked of you, I recommend quoting an expert like Bertrand Meyer or Bjarne Stroustrup; that way, if the interviewer disagrees, his beef is with the expert and not with you.)

The literal meaning of polymorphism is "the ability to take on multiple forms or shapes." In its broadest sense, this refers to the ability of different objects to respond in different ways to the same message (or method invocation).

Damian Conway, in his book *Object-Oriented Perl*, distinguishes meaningfully between two kinds of polymorphism. The first,*inheritance polymorphism,* is what most programmers are referring to when they talk about polymorphism.

When a class inherits from its superclass, we know (by definition) that any method present in the superclass is also present in the subclass. Therefore, a chain of inheritance represents a linear hierarchy of classes that can respond to the same set of methods. Of course, we must remember that any subclass can redefine a method; that is what gives inheritance its power. If I call a method on an object, typically it will be either the one it inherited from its superclass or a more appropriate (more specialized) method tailored for the subclass.

In strongly typed languages such as C++, inheritance polymorphism establishes type compatibility down the chain of inheritance (but not in the reverse direction). For example, if `B` inherits from `A`, then a pointer to an `A` object can also point to a `B` object. However, the reverse is not true. This type compatibility is an essential OOP feature in such languages—indeed it almost sums up polymorphism—but polymorphism certainly exists in the absence of static typing (as in Ruby).

The second kind of polymorphism Conway identifies is *interface polymorphism*. This does not require any inheritance relationship between classes; it only requires that the interfaces of the objects have methods of a certain name. The treatment of such objects as being the same "kind" of thing is therefore a type of polymorphism (although in most writings it is not explicitly referred to as such).

Readers familiar with Java will recognize that it implements both kinds of polymorphism. A Java class can extend another class, inheriting from it via the `extends` keyword, or it may implement an interface, acquiring a known set of methods (which must then be overridden) via the `implements` keyword. Because of the syntax requirements, the Java interpreter is able to determine at compile time whether a method can be invoked on a particular object.

Ruby supports interface polymorphism but in a different way, providing*modules* whose methods may be *mixed in* to existing classes (interfacing to user-defined methods that are expected to exist). This, however, is not the way modules are usually used. A module consists

of methods and constants that may be used as though they were actual parts of that class or object; when a module is mixed in via the `include` statement, this is considered to be a restricted form of multiple inheritance. (According to the language designer Yukihiro Matsumoto, this can be viewed as "single inheritance with implementation sharing.") This is a way of preserving the benefits of MI without suffering all the consequences.
It's worth noting that Ruby supports implicit interface polymorphism by virtue of the simple fact that any class can "masquerade" as another class. In many cases, the only type information we care about is whether a certain set of methods is implemented—that is, whether an object responds to certain messages. Sometimes we write code for a `Duck` object when really all we care about is for it to implement a `quack` method. Yet, if something "quacks" like a `Duck`, for our purposes it *is* a `Duck` (with no need to inherit from that class at all). The set of available methods is arguably the most important type information.
Languages such as C++ contain the concept of *abstract classes*—classes that must be inherited from and cannot be instantiated on their own. This concept does not exist in the more dynamic Ruby language, although if the programmer really wants, it is possible to fake this kind of behavior by forcing the methods to be overridden. Whether this is useful or not is left as an exercise for you, the reader.
The creator of C++, Bjarne Stroustrup, also identifies the concept of a concrete type. This is a class that exists only for convenience; it is not designed to be inherited from, nor is it expected that there will ever be another class derived from it. In other words, the benefits of OOP are basically limited to encapsulation. Ruby does not specifically support this concept through any special syntax (nor does C++), but it is naturally well suited for the creation of such classes.
Some languages are considered to be more "purely" object-oriented than others. (We also use the term*radically object oriented.*) This refers to the concept that *every* entity in the language is an object; every primitive type is represented as a full-fledged class, and variables and constants alike are recognized as object instances. This is in contrast to such languages as Java, C++, and Eiffel. In these, the more primitive data types (especially constants) are not first-class objects, although they may sometimes be treated that way with "wrapper" classes.
Most object-oriented languages are fairly static; the methods and attributes belonging to a class, the global variables, and the inheritance hierarchy are all defined at compile time. Perhaps the largest conceptual leap for a Ruby programmer is that these are all handled *dynamically* in Ruby. Definitions and even inheritance can happen at runtime—in fact, we can truly say that every declaration or definition is actually *executed* during the running of the program. Among many other benefits, this obviates the need for conditional compilation and can produce more efficient code in many circumstances.
This sums up the whirlwind tour of OOP. Throughout the rest of the book, we have tried to make consistent use of the terms introduced here. Let's proceed now to a brief review of the Ruby language itself.

Basic Ruby Syntax and Semantics

Bring forth that ruby gem of Badakhshan, That heart's delight, that balm of Turkestan….
—*The Rubaiyat*, Omar Khayyam (trans. E. H. Whinfield)

In the previous pages, you have already seen that Ruby is a pure, dynamic, OOP language. Let's now look briefly at some other attributes before summarizing the syntax and semantics.
Ruby is a scripting language. This should not be construed as meaning that it is not powerful. It can serve as a "glue language" in the tradition of KornShell and others, or it can serve as an environment for creating larger self-contained applications. Readers who are interested in the industry trend toward scripting languages should refer to John Ousterhout's article "Scripting: Higher-Level Programming for the 21st Century" in the March 1998 issue of *IEEE Computer*. (Ousterhout is the creator of the Tcl language.)
Ruby is an interpreted language. Of course, there may be later implementations of a Ruby compiler for performance reasons, but we maintain that an interpreter yields great benefits not only in rapid prototyping but in the shortening of the development cycle overall.
Ruby is an expression-oriented language. Why use a statement when an expression will do? This means, for instance, that code becomes more compact as the common parts are factored out and repetition is removed.
Ruby is a very high-level language (VHLL). One principle behind the language design is that the computer should work for the programmer rather than vice versa. The "density" of Ruby means that sophisticated and complex operations can be carried out with relative ease as compared to lower-level languages.
Having said all that, let's look more closely at Ruby. This section and the rest of the chapter concentrate on the Ruby language itself. As mentioned before, this is only a quick summary, so if you haven't learned it somewhere else, you won't learn it here.
Our first look at Ruby will not concentrate on the language's more complex features. These are covered in the next two sections. Here, we are concerned with the overall look and feel of the language and some of its terminology. We'll briefly examine the nature of a Ruby program before looking at examples.
To begin with, Ruby is essentially a line-oriented language—more so than languages such as C but not so much as antique languages such as FORTRAN. Tokens can be crowded onto a single line as long as they are separated by whitespace, as needed. Statements may occur more than one to a line if they are separated by semicolons; this is the only time the terminating semicolon is really needed. A line may be continued to the next line by ending it with a backslash or by letting the parser know that the statement is not complete—for example, by ending a line with a comma.
There is no main program as such; execution proceeds in general from top to bottom. In more complex programs, there may be numerous definitions at the top, followed by the (conceptual) main program at the bottom. However, even in that case, execution proceeds from the top down because definitions in Ruby are executed.

Keywords and Identifiers

The keywords (or reserved words) in Ruby typically cannot be used for other purposes. These are as follows:

`BEGIN`	`END`	`alias`	`and`	`begin`
`break`	`case`	`class`	`def`	`defined`
`do`	`else`	`elsif`	`end`	`ensure`
`false`	`for`	`if`	`in`	`module`
`next`	`nil`	`not`	`or`	`redo`
`rescue`	`retry`	`return`	`self`	`super`
`then`	`true`	`undef`	`unless`	`until`
`when`	`while`	`yield`		

Variables and other identifiers normally start with an alphabetic letter or a special modifier. The basic rules are as follows:

- Local variables (and pseudo-variables such as `self` and `nil`) begin with a lowercase letter.
- Global variables begin with a dollar sign (`$`).
- Instance variables (within an object) begin with an "at" sign (`@`).
- Class variables (within a class) begin with two "at" signs (`@@`).
- Constants begin with capital letters.

For purposes of forming identifiers, the underscore (`_`) may be used as a lowercase letter. Special variables starting with a dollar sign (such as `$1` and `$/`) are not dealt with here. The following list provides some examples:

- Local variables: `alpha`, `_ident`, `some_var`
- Pseudo-variables: `self`, `nil`, `__FILE__`
- Constants: `K6chip`, `Length`, `LENGTH`
- Instance variables: `@foobar`, `@thx1138`, `@NOT_CONST`
- Class variable: `@@phydeaux`, `@@my_var`, `@@NOT_CONST`
- Global variables: `$beta`, `$B12vitamin`, `$NOT_CONST`

Comments and Embedded Documentation

Comments in Ruby begin with a pound sign (`#`) outside of a string or character constant and proceed to the end of the line:

```
x = y + 5 # This is a comment.
# This is another comment.
print "# But this isn't."
```

Embedded documentation is intended to be retrieved from the program text by an external tool such as RDTOOL. From the point of view of the interpreter, it is like a comment and can be used as such. Given two lines starting with `=begin` and `=end`, everything between those lines (inclusive) is ignored by the interpreter:

```
=begin
The purpose of this program
is to cure cancer
and instigate world peace.
=end
```

Constants, Variables, and Types

In Ruby, variables do not have types, but the data they contain still has types. The simplest data types are numeric, character, and string.
Some numeric constants are shown in the following list:

- Integer: 237
- Integer (with sign): -123
- Integer (with underscore spacing): 1_048_576
- Octal integer: 0377
- Hexadecimal integer: 0xBEEF
- Floating point: 3.14159
- Floating point (scientific notation): 6.02e23

Character constants in Ruby actually evaluate to integers according to the ASCII code and are therefore interchangeable with the corresponding integer constants. Here are some character constants:

- Lowercase x (120): `?x`
- Newline: `?\n`
- Backslash: `?\\`
- Ctrl+D: `?\cd`
- Ctrl+X: `?\C-x`
- Meta+X (x `OR`ed with 0x80): `?\M-x`
- Meta+Ctrl+X: `?\M-\C-x`

Note that all these forms (and some others) can also be embedded in strings, as you'll see shortly.
Ruby has a wealth of notations available for representing strings; different ones may be convenient in different situations. Most commonly, a string constant in Ruby is enclosed between double quotes, as in C.
It is possible to embed "escaped" character constants in a Ruby string in order to express control characters and the like:

```
"This is a single line.\n"
"Here are three tabs\ t\t\tand then more text."
"A backslash (\\) must be doubled."
```

It is also possible to embed variables or even expressions inside these strings. The pound sign (`#`) is used to signal that this is being done; typically the variable or expression is enclosed in braces, but they may be omitted if the expression consists of a single variable beginning with `$` or `@`. Here are examples:

```
"The tax rate is #{ taxrate} ."
"Hello, #@yourname; my name is #{ myname} ."
"The sum is #{ a+b+c} ."
"#$num1 times #$num2 equals #{ $num1*$num2} ."
```

A single-quoted string in Ruby is the same except that no expression substitution is performed and no backslash processing is done, except `\\` and `\'`. Single-quoted strings are useful when the strings are to be used more "as is," without the special interpretations. Here are examples:

```
'The notation #{ alpha}  will not be expanded.'
'We can embed the \\  (backslash) character'
'or the \' (single quote) character.'
```

For cases where the strings contain punctuation that would normally have to be escaped, there is a more general form of quote. The percent sign (`%`) introduces a string delimited according to rules determined by the character following the percent sign. Basically this character may be a lowercase *q,* an uppercase *Q,* a brace or parenthesis, or some other character. In the first two cases, there is still a delimiter character following the letter. We'll discuss each case briefly.

A `%q` string is a generalized single-quoted string; as such, there is no expression substitution and minimal backslash processing. The delimiter may be any character, including newline. If an opening brace or parenthesis is used, the corresponding closing brace or parenthesis closes the string; otherwise, the same character opens and closes the string. Here are examples:

```
%q(The notation #{ alpha}  will not be expanded.)
%q{ We can embed \\  and \}  in this string.}
%q/These characters are not special: " ' # () { } /
```

A `%Q` string is a generalized double-quoted string, meaning that substitution and backslash processing both occur as they normally would. The delimiters behave as with the `%q` string. Here are examples:

```
%Q(We can embed tabs \t\t and newlines and so on.)
%Q/Here, these characters are not special: () " '/
%Q("Hello, #{ name} ," I said to her.)
%Q(He said, "She said, 'Hello.'")
```

The `q` or `Q` may be left out entirely so that the delimiter immediately follows the percent sign. This delimiter obviously may not be `q` or `Q` but also may not be `r`, `w`, or `x`, for reasons you'll see shortly. In this case, the string once again acts like a double-quoted string. Here are examples:

```
%(The variable alpha = #{ alpha} .)
%/Tab \t  Carriage return \r  Newline \n/
%{ Using a brace makes substitution hard: #\{ beta\} }
%<Less-than greater-than will also work.>
%[As will square brackets.]
```

Note once again that the closing delimiter is the same as the opening delimiter for most characters, but a "paired character" used as a delimiter requires the opposite paired character to close the string. The paired characters are parentheses, brackets, braces, and the so-called "angle brackets": `()`, `[]`, `{}`, and `<>`, respectively. Note, however, that the grave accent (`` ` ``) and single quote (`'`) are *not* paired characters as some might think.
A special kind of string is worth mentioning here that's primarily useful in small scripts used to glue together larger programs. The command output string will be sent to the operating system as a command to be executed, whereupon the output of the command is substituted back into the string. The simple form of this string uses the grave accent (sometimes called a *back tick* or *back quote*) as a beginning and ending delimiter; the more complex form uses the `%x` notation:

```
`whoami`
`ls -l`
%x[grep -i meta *.html | wc -l]
```

Regular expressions in Ruby look similar to character strings, but they are used differently. Many operations in Ruby make sense with regular expressions but not with strings.
For those familiar with Perl, regular expression handling is similar in Ruby. Incidentally, we'll use the abbreviation *regex* throughout the remainder of the book; many abbreviate it as *regexp,* but that is not as pronounceable.
The typical regex is delimited by a pair of slashes; the `%r` form can also be used. Here are some simple regular expressions:

- `/Ruby/` Matches the single word *Ruby*
- `/[Rr]uby/` Matches *Ruby* or *ruby*
- `/^abc/` Matches an instances of *abc* at the beginning of a line
- `%r(xyz$)` Matches an instance of *xyz* at the end of a line
- `%r|[0-9]*|` Matches any sequence of (zero or more) digits

It is also possible to place a modifier, consisting of a single letter, immediately after a regex. The modifiers are as follows:

- `i`Ignores case in regex

- oPerforms expression substitution only once
- mMultiline mode (dot matches newline)
- xExtended regex (allows whitespace and comments)

To complete our introduction to regular expressions, here's a list of the most common symbols and notations available:

- ^ Beginning of a line or string
- $End of a line or string
- . Any character except newline (unless POSIX)
- \wWord character (digit, letter, or underscore)
- \W Non-word character
- \s Whitespace character (space, tab, newline, and so on)
- \S Non-whitespace character
- \d Digit (same as [0-9])
- \D Non-digit
- \A Beginning of a string
- \Z End of a string or before newline at the end
- \z End of a string
- \b Word boundary (outside [ ] only)
- \B Non-word boundary
- \b Backspace (inside [ ] only)
- [ ] Any single character of set
- * Zero or more of the previous subexpression
- *? Zero or more of the previous subexpression (non-greedy)
- + One or more of the previous subexpression
- +? One or more of the previous subexpression (non-greedy)
- {m,n}*M* to *n* instances of the previous subexpression
- {m,n}?*M* to *n* instances of the previous subexpression (non-greedy)
- ? Zero or one instance of the previous regular expression
- | Alternatives
- ( ) Grouping of subexpressions
- (?#) Comment

An understanding of regex handling is a powerful tool for the modern programmer. A complete discussion is far beyond the scope of this book. Instead, we refer you to the definitive work *Mastering Regular Expressions* by Jeffrey Friedl.
An *array* in Ruby is a very powerful construct; it may contain data of any type or even mixed types. As you'll see in a later section, all arrays are instances of the class Array and therefore have a rich set of methods that can operate on them. An array constant is delimited by brackets. The following are all valid array expressions:

```
[1, 2, 3]
[1, 2, "buckle my shoe"]
[1, 2, [3,4], 5]
["alpha", "beta", "gamma", "delta"]
```

The second example shows an array containing both integers and strings, the third example shows a nested array, and the fourth shows an array of strings. As in most languages, arrays are "zero indexed;" for instance, in the last array, "gamma" is element number 2. Arrays are dynamic and do not need to have a size specified when they are created.
Because the array of strings is so common (and so inconvenient to type), a special syntax has been set aside for it, similar to what you have seen before:

```
%w[alpha beta gamma delta]
%w(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)
%w/am is are was were be being been/
```

In these examples, the quotes and commas are not needed; only whitespace separates the individual elements. In the case of an element that contains whitespace, of course, this would not work.
An array variable can use brackets to index into the array. The resulting expression can be both examined and assigned to:

```
val = myarray[0]
print stats[j]
x[i] = x[i+1]
```

Another extremely powerful construct in Ruby is the *hash,* which is also commonly called an *associative array* or *dictionary.* A hash is a set of associations between paired pieces of data; it is typically used as a lookup table or a kind of generalized array in which the index need not be an integer. Each hash is an instance of the class `Hash`.
A hash constant is typically represented between delimiting braces, with the symbol=> separating the individual keys and values. The key can be thought of as an index where the corresponding value is stored. There is no restriction on the types of the keys or the corresponding values. Here are some hashes:

```
{1=>1, 2=>4, 3=>9, 4=>16, 5=>25, 6=>36}
{ "cat"=>"cats", "ox"=>"oxen", "bacterium"=>"bacteria"}
{ "hydrogen"=>1, "helium"=>2, "carbon"=>12}
{ "odds"=>[1,3,5,7], "evens"=>[2,4,6,8]}
{"foo"=>123, [4,5,6]=>"my array", "867-5309"=>"Jenny"}
```

A hash variable can have its contents accessed by essentially the same bracket notation that arrays use:

```
print phone_numbers["Jenny"]
plurals["octopus"] = "octopi"
```

It should be stressed, however, that both arrays and hashes have many methods associated with them; these methods give them their real usefulness. The next section, covering Ruby OOP, will expand on this a little more.

Operators and Precedence

Now that we have established our most common data types, let's look at Ruby's operators. They are arranged here in order from highest to lowest precedence:

- Scope `::`
- Indexing `[]`
- Exponentiation `**`
- Unary positive/negative, etc. `+ - ! ~`
- Multiplication, etc. `* / %`
- Addition/subtraction `+ -`
- Logical shifts, etc. `<< >>`
- Bitwise and `&`
- Bitwise or, xor `| ^`
- Comparison `> >= < <=`
- Equality, etc. `== === <=> != =~ !~`
- Boolean and `&&`
- Boolean or `||`
- Range operators `.. ...`
- Assignment `=` (also `+=`, `-=`, `*=`, etc.)
- Ternary decision `?:`
- Boolean negation `not`
- Boolean and, or `and or`

Some of these operators serve more than one purpose; for example, the operator << is a bitwise left shift but is also an append operator (for arrays, strings, and so on) and a marker for a here-document. Likewise, the plus sign (+) is for addition and for string concatenation. As you'll see later, many of these operators are just shortcuts for method names.
Now we have defined most of the data types and many of the possible operations on them. Before going any further, let's look at an actual sample program.

A Sample Program

In a tutorial, the first program is always "Hello, World!" But in a whirlwind tour like this one, let's start with something slightly more advanced. Here's a small program to convert between Fahrenheit and Celsius temperatures:

```
print "Please enter a temperature and scale (C or F): "
str = gets
exit if not str or not str[0]
```

```
str.chomp!
temp, scale = str.split(" ")
if temp !~ /-?[0-9]+/
  print temp, " is not a valid number.\n"
  exit 1
end
temp = temp.to_f

case scale
  when "C", "c"
    f = 1.8*temp + 32
  when "F", "f"
    c = (5.0/9.0)*(temp-32)
  else
    print "Must specify C or F.\n"
    exit 2
end

if c != nil then
  print "#{ c}  degrees C\n"
else
  print "#{ f}  degrees F\n"
end
```

Here are some examples of running this program. These show that the program can convert from Fahrenheit to Celsius and from Celsius to Fahrenheit and that it can handle an invalid scale or an invalid number:

```
Please enter a temperature and scale (C or F): 98.6 F
37.0 degrees C

Please enter a temperature and scale (C or F): 100 C
212.0 degrees F

Please enter a temperature and scale (C or F): 92 G
Must specify C or F.

Please enter a temperature and scale (C or F): junk F
junk is not a valid number.
```

Now, as for the mechanics of the program, we begin with a `print` statement, which is actually a call to the predefined function `print`, to write to standard output. Following this, we call `gets` (get string from standard input), assigning the value to `str`.

Note that any apparently "free-standing" function calls such as `print` and `gets` are actually methods of various predefined classes or objects. In the same way, `chomp` is a method that is called with `str` as a receiver. Method calls in Ruby generally can omit the parentheses; for example, `print "foo"` is the same as `print("foo")`.

The variable `str` holds a character string, but there is no reason it could not hold some other type instead. In Ruby, data has types but variables do not.

The special method call `exit` will terminate the program. On this same line is a control structure called an `if`*modifier.* This is like the `if` statement that exists in most languages, but backwards; it comes after the action, does not permit an `else`, and does not require

closing. As for the condition, we are checking two things: Does `str` have a value, and is it non-null? In the case of an immediate end-of-file, our first condition will hold; in the case of a newline with no preceding data, the second condition will hold.
The reason these tests work is that a variable that is undefined has a `nil` value, and `nil` evaluates to false in Ruby. In fact, `nil` and `false` evaluate as false, and everything else evaluates as true. Specifically, the null string `""` does *not* evaluate as false, as it does in some other languages.
The next statement performs a `chomp!` operation on the string (to remove any trailing newline characters). The exclamation point as a prefix serves as a warning that the operation actually changes the value of its receiver rather than just returning a value. The exclamation point is used in many such instances to remind the programmer that a method has a side effect or is more "dangerous" than its unmarked counterpart. The method `chomp`, for example, will return the same result but will not modify its receiver.
The next statement is an example of multiple assignment. The `split` method splits the string into an array of values, using the space as a delimiter. The two assignable entities on the left side will be assigned the respective values resulting on the right side.
The `if` statement that follows uses a simple regex to determine whether the number is valid; if the string fails to match a pattern consisting of an optional minus sign followed by one or more digits, it is an invalid number and the program exits. Note that the `if` statement is terminated by the keyword `end`; although it's not needed here, we could have had an `else` clause before `end`. The keyword `then` is optional; this statement does not use `then`, but the one below it does. As for the output, recall that the variable `temp` could also have been embedded in the string (as below).
The `to_f` method is used to convert the string to a floating-point number. We are actually assigning this floating-point value back to `temp`, which originally held a string.
The `case` statement chooses between three alternatives: the case in which the user specifies a C, specifies an F, or uses an invalid scale. In the first two instances, a calculation is done; in the third, we print an error and exit.
Ruby's `case` statement, by the way, is far more general than the example shown here. There is no limitation on the data types, and the expressions used are all arbitrary and may even be ranges or regular expressions.
There is nothing mysterious about the computation. However, consider the fact that the variables `c` and `f` are referenced first inside the branches of the case. There are no declarations as such in Ruby; a variable comes into existence when it is assigned. This means that when we fall through the `case` statement, only one of these variables will have a value.
We use this to determine after the fact which branch was followed so that we can create a slightly different output in each instance. The comparison of `c` with `nil` is effectively a test of whether `c` has a value. We do this here only to show that it can be done; obviously two different `print` statements could be used inside the `case` statement if we wished.
You may have noticed that we've used only "local" variables here. This might be somewhat confusing, because their scope certainly appears to cover the entire program. What's happening here is that the variables are all local to the *top level* of the program (written as

toplevel by some). The variables appear "global" because there are no lower-level contexts in a program this simple; however, if we declared classes and methods, these top-level variables would not be accessible within them.

Looping and Branching

Let's spend some time looking at control structures. We have already seen the simple `if` statement and the `if` modifier; there are also corresponding structures based on the keyword `unless` (which also has an optional `else`) as well as expression-oriented forms of `if` and `unless`. We summarize all these as follows:

```
if x < 5 then
  statement1
end

unless x >= 5 then
  statement1
end

if x < 5 then
  statement1
else
  statement2
end

unless x < 5 then
  statement2
else
  statement1
end

statement1 if y == 3

statement1 unless y != 3

x = if a>0 then b else c end

x = unless a<=0 then c else b end
```

In this summary, the `if` and `unless` forms behave exactly the same. Note that the keyword `then` may be omitted except in the final (expression-oriented) cases. Note also that the modifier forms cannot have an `else` clause.

The `case` statement in Ruby is more powerful than in most languages. This multiway branch can even test for conditions other than equality—for example, a matched pattern. The test done by the `case` statement corresponds to the relationship operator (===), which has a behavior that varies from one object to another. Let's look at an example:

```
case "This is a character string."
  when "some value"
    print "Branch 1\n"
  when "some other value"
    print "Branch 2\n"
  when  /char/
    print "Branch 3\n"
  else
    print "Branch 4\n"
end
```

This code will print `Branch 3`. Why? It first tries to check for equality between the tested expression and one of the strings `"some value"` or `"some other value"`. This fails, so it proceeds. The third test is for the presence of a pattern within the tested expression; that pattern is there, so the test succeeds and the third `print` statement is performed. The `else` clause will always handle the default case in which none of the preceding tests succeeds.

If the tested expression is an integer, the compared value can be an integer range (for example, `3..8`). In this case, the expression will be tested for membership in that range. In all instances, the first successful branch will be taken.

As for looping mechanisms, Ruby has a rich set. The `while` and `until` control structures are both pretest loops, and both work as expected: One specifies a continuation condition for the loop, and the other specifies a termination condition. They also occur in "modifier" form like `if` and `unless`. There is also the `loop` method of the `Kernel` module (by default an infinite loop), and iterators (described later) are associated with various classes.

The following examples assume an array called `list`, defined something like this:

```
  list = %w[alpha bravo charlie delta echo]
```

They all step through the array and write out each element:

```
  # Loop 1 (while)
  i=0
  while i < list.size do
    print "#{ list[i]}  "
    i += 1
  end

  # Loop 2 (until)
  i=0
  until i == list.size do
    print "#{ list[i]}  "
    i += 1
  end

  # Loop 3 (post-test while)
  i=0
  begin
    print "#{ list[i]}  "
    i += 1
  end while i < list.size

  # Loop 4 (post-test until)
  i=0
  begin
    print "#{ list[i]}  "
    i += 1
```

```
end until i == list.size

# Loop 5 (for)
for x in list do
  print "#{ x}  "
end

# Loop 6 ('each' iterator)
list.each do |x|
  print "#{ x}  "
end

# Loop 7 ('loop' method)
i=0
n=list.size-1
loop do
  print "#{ list[i]}  "
  i += 1
  break if i > n
end

# Loop 8 ('loop' method)
i=0
n=list.size-1
loop do
  print "#{ list[i]}  "
  i += 1
  break unless i <= n
end

# Loop 9 ('times' iterator)
n=list.size
n.times do |i|
  print "#{ list[i]}  "
end

# Loop 10 ('upto' iterator)
n=list.size-1
0.upto(n) do |i|
  print "#{ list[i]}  "
end

# Loop 11 (for)
n=list.size-1
for i in 0..n do
  print "#{ list[i]}  "
end

# Loop 12 ('each_index')
list.each_index do |x|
  print "#{ list[x]}  "
end
```

Let's examine these in a little detail. Loops 1 and 2 are the "standard" forms of the `while` and `until` loops; they behave essentially the same, but their conditions are negations of each other. Loops 3 and 4 are the same thing in "post-test" versions; the test is performed at the end of the loop rather than at the beginning. Note that the use of `begin` and `end` in this context is strictly a kludge or hack; what is really happening is that a `begin/end` block (used

for exception handling) is followed by a `while` or `until` modifier. For someone really wanting a post-test loop, however, this is effectively the same.
Loops 5 and 6 are arguably the "proper" ways to write this loop. Note the simplicity of these two compared with the others; there is no explicit initialization and no explicit test or increment. This is because an array "knows" its own size, and the standard iterator `each` (loop 6) handles such details automatically. Indeed, loop 5 is merely an indirect reference to this same iterator because the `for` loop will work for any object having the iterator `each` defined. The `for` loop is only shorthand for a call to `each`; such shorthand is frequently called *syntax sugar* because it offers a more convenient alternative to another syntactic form.
Loops 7 and 8 both make use of the `loop` construct; as mentioned earlier, `loop` looks like a keyword introducing a control structure, but it is really a method of the module `Kernel`, not a control structure at all.
Loops 9 and 10 take advantage of the fact that the array has a numeric index; the `times` iterator will execute a specified number of times, and the `upto` iterator will carry its parameter up to the specified value. Neither of these is truly suitable for this instance.
Loop 11 is a `for` loop that operates specifically on the index values, using a range, and loop 12 likewise uses the `each_index` iterator to run through the list of array indexes.
In the preceding examples, we have not laid enough emphasis on the "modifier" form of the `while` and `until` loops. These are frequently useful, and they have the virtue of being concise. We offer these additional examples, both of which mean the same thing:

```
perform_task() until finished
perform_task() while not finished
```

One fact is largely ignored here: Loops do not always run smoothly from beginning to end, in a predictable number of iterations, or ending in a single predictable way. We need ways to control these loops further.
The first of these is the `break` keyword, which you see in loops 7 and 8. This is used to "break out" of a loop; in the case of nested loops, only the innermost one is halted. This will be intuitive for C programmers.
The keyword`retry` is used in two contexts: in the context of an iterator and in the context of a `begin/end` block (exception handling). Within the body of any iterator (or `for` loop) it will force the iterator to restart, reevaluating any arguments passed to the iterator. Note that it will not work for loops in general (`while` and `until`).
The`redo` keyword is the generalized form of `retry` for loops. It works for `while` and `until` loops just as `retry` works for iterators.
The`next` keyword will effectively jump to the end of the innermost loop and resume execution from that point. It works for any loop or iterator.
The iterator is an important concept in Ruby, as you have already seen. What you have not seen is that the language allows user-defined iterators in addition to the predefined ones.

The default iterator for any object is called`each`. This is significant because it allows the `for` loop to be used. However, iterators may be given different names and used for varying purposes.
As a crude example, consider this multipurpose iterator, which mimics a post-test loop (like C's `do-while` or Pascal's `repeat-until`):

```
def repeat(condition)
  yield
  retry if not condition
end
```

In this example, the keyword `yield` is used to call the block that is specified when the iterator is called in this way:

```
j=0
repeat (j<10) do {  j+=1; print j,"\n"}
```

It is also possible to pass parameters via yield, which will be substituted into the block's parameter list (between vertical bars). As a somewhat contrived example, the following iterator does nothing but generate integers from 1 to 10, and the call of the iterator generates the first 10 cubes:

```
def my_sequence
  for i in 1..10 do
    yield i
  end
end

my_sequence { |x| print x**3, "\n"}
```

Note that `do` and `end` may be substituted for the braces that delimit a block. There are differences, but they are fairly subtle.

Exceptions

Like many other modern programming languages, Ruby supports *exceptions.* An exception is a means of handling errors that has significant advantages over older methods. Return codes are avoidable, as is the "spaghetti logic" that results from checking them. Also, the code that detects the error can be distinguished from the code that knows how to handle the error (because these are often separate anyway).
The `raise` statement will raise an exception. Note that `raise` is not a reserved word but rather a method of the module `Kernel`. (Its alias is named `fail`.) Here are examples:

```
raise                                           # Example 1
raise "Some error message."                     # Example 2
raise ArgumentError                             # Example 3
raise ArgumentError, "Invalid data."            # Example 4
raise ArgumentError.new("Invalid data.")        # Example 5
raise ArgumentError, "Invalid data.", caller[0] # Example 6
```

In example 1, the last exception encountered is re-raised. In example 2, a `RuntimeError` (the default error) is created using the message `"Some error message."` In example 3, an `ArgumentError` is raised; in example 4, this same error is raised with the message `"Invalid data."` Example 5 behaves exactly the same as example 4. Finally, example 6 adds traceback information of the form `"filename:line"` or ``"filename:line:in `method'"`` (as stored in the array returned by the `caller` method or stored in the `$a` special variable).
Now, how do we handle exceptions in Ruby? The `begin-end` block is used for this purpose. The simplest form is a `begin-end` block with nothing but our code inside:

```
begin  # No real purpose.
  # ...
end
```

This, however, is of no value in catching errors. The block, however, may have one or more `rescue` clauses in it. If an error occurs at any point in the code, between `begin` and `rescue`, control will be passed immediately to the appropriate `rescue` clause. Here's an example:

```
begin
  x = Math.sqrt(y/z)
  # ...
rescue ArgumentError
  print "Error taking square root.\n"
rescue ZeroDivisionError
  print "Attempted division by zero.\n"
end
```

Essentially the same thing can be accomplished by this fragment:

```
begin
  x = Math.sqrt(y/z)
  # ...
rescue => err
  print err, "\n"
end
```

Here, the variable `err` is used to store the value of the exception; printing it causes it to be translated to some meaningful character string. Note that because the error type is not specified, the `rescue` clause will catch every kind of error. The notation `rescue => variable` can be used with or without an error type before the `=>` symbol.
In the event that error types are specified, it may be that an exception does not match any of these types. For that situation, we are allowed to use an `else` clause after all the `rescue` clauses:

```
begin
  # Error-prone code...
rescue Type1
  # ...
rescue Type2
  # ...
else
  # Other exceptions...
end
```

In many cases, we will want to do some kind of recovery. In that event, the keyword `retry` (within the body of a `rescue` clause) will restart the `begin` block and try those operations again:

```
begin
  # Error-prone code...
rescue
  # Attempt recovery...
  retry
end
```

Finally, it is sometimes necessary to write code that "cleans up" after a `begin-end` block. In the event this is necessary, an `ensure` clause can be specified:

```
begin
  # Error-prone code...
rescue
  # Handle exceptions
ensure
  # This code is always executed
end
```

The code in an `ensure` clause is always executed before the `begin-end` block exits. This happens regardless of whether an exception occurred.
There are two other ways in which exceptions may be caught. First of all, there is a modifier form of the `rescue` clause:

```
x = a/b  rescue print "Division by zero!\n"
```

In addition, the body of a method definition is an implicit `begin-end` block; the `begin` is omitted, and the entire body of the method is subject to exception handling, ending with the `end` of the method:

```
def some_method
    # Code...
rescue
    # Recovery...
end
```

This sums up the discussion of exception handling as well as the discussion of fundamental syntax and semantics.

Numerous aspects of Ruby have not been discussed here. The next two sections are devoted to the more advanced features of the language, and the final section is mostly a collection of Ruby lore that will help the intermediate programmer learn to "think in Ruby."

OOP in Ruby

Ruby has all the elements more generally associated with OOP languages, such as objects with encapsulation and data hiding, methods with polymorphism and overriding, and classes with hierarchy and inheritance. It goes farther and adds limited metaclass features, singleton methods, modules, and mixins.
Similar concepts are known by other names in other OOP languages, but concepts of the same name may have subtle differences from one language to another. This section elaborates on the Ruby understanding and usage of these elements of OOP.

Objects

In Ruby, all numbers, strings, arrays, regular expressions, and many other entities are actually objects. Work is done by executing the methods belonging to the object:

```
3.succ                  # 4
 "abc".upcase           # "ABC"
 [2,1,5,3,4].sort       # [1,2,3,4,5]
someObject.someMethod   # some result
```

In Ruby, *every* object is an instance of some class; the class contains the implementation of the methods. The object's class is essentially its type:

```
"abc".type     # String
"abc".class    # String
```

In addition to encapsulating its own attributes and operations, an object in Ruby has an identity.

```
"abc".id      #  53744407
```

Built-in Classes

More than 30 built-in classes are predefined in the Ruby class hierarchy. Like many other OOP languages, Ruby does not allow multiple inheritance, but that does not necessarily make it any less powerful. Modern object-oriented languages frequently follow the single-inheritance model. Ruby does support modules and mixins, which are discussed in the next section. It also implements object IDs, which support the implementation of persistent, distributed, and relocatable objects.
To create an object from an existing class, the `new`method is typically used:

```
myFile = File.new("textfile.txt","w")
myString = String.new("this is a string object")
```

This is not always explicitly required, however, as shown here:

```
yourString = "this is also a string object"
aNumber = 5
```

Variables are used to hold references to objects. As previously mentioned, variables themselves have no type, nor are they objects themselves; they are simply references to objects. Here's an example:

```
x = "abc"
```

An exception to this is that small immutable objects of some built-in classes, such as `Fixnum`, are copied directly into the variables that refer to them. (These objects are no bigger than pointers, and it is more efficient to deal with them in this way.) In this case, the assignment makes a copy of the object, and the heap (memory allocation area) is not used. Variable assignment causes object references to be shared:

```
y = "abc"
x = y
x            # "abc"
```

After `x = y` is executed, variables `x` and `y` both refer to the same object:

```
x.id         # 53732208
y.id         # 53732208
```

If the object is mutable, a modification done to one variable will be reflected in the other one:

```
x.gsub!(/a/,"x")
y                   # "xbc"
```

Reassigning one of these variables has no effect on the other, however:

```
x = "abc"
y                   # still has value "xbc"
```

A mutable object can be made immutable using the `freeze` method:

```
x.freeze
x.gsub!(/b/,"y")    #  error
```

A symbol in Ruby refers to a variable by ID rather than by reference. When we say `:x`, we are saying basically the same as `x.id` (which you saw previously). A colon applied to an identifier results in a symbol; if the identifier does not already exist, it is created. Among other uses, a symbol may be used when we want to *mention* an identifier as opposed to *using* it (the classical use/mention distinction); for example, the special method `method_missing`, called when a method is not found, gets passed a symbol corresponding to the unknown method. Any `Symbol` object has a method called `id2name` that returns a string corresponding to the identifier name. Here are examples:

```
Hearts    = :Hearts   # This is one way of assigning
Clubs     = :Clubs    #   unique values to constants,
Diamonds  = :Diamonds #   somewhat like an enumeration
Spades    = :Spades   #   in Pascal or C.

print Hearts.id2name  # Prints "Hearts"
```

Modules and Mixins

Many built-in methods are available from class ancestors. Of special note are the `Kernel` methods mixed in to the `Object` superclass; because `Object` is universally available, the methods that are added to it from `Kernel` are also universally available. These methods form a very important part of Ruby.

The terms *module* and *mixin* are nearly synonymous. A module is a collection of methods and constants that is external to the Ruby program. It can be used simply for namespace management, but the most common use of a module is to have its features "mixed" in to a class (by using `include`). In this case, it is used as a mixin. (This term, apparently borrowed from Python, is sometimes written as *mix-in,* but we write it as a single word.)

An example of using a module for namespace management is the frequent use of the `Math` module. To make use of the definition of pi, for example, it is not necessary to include the `Math` module; you can simply use `Math::PI` as the constant.

A mixin provides a way of getting some of the benefits of multiple inheritance without dealing with all the difficulties. It can be considered a restricted form of multiple inheritance, but the language creator Matz has called it "single inheritance with implementation sharing."

Note that `include` appends features of a namespace (a module) to the current space. The `extend` method appends functions of a module to an object. With `include`, the module's methods become available as instance methods; with `extend`, they become available as class methods.

We should mention that `load` and `require` do not really relate to modules but rather to non-module Ruby sources and binaries (statically or dynamically loadable). A `load` operation essentially reads a file and inserts it at the current point in the source file so that its definitions become available at that point. A `require` operation is similar to a `load`, but it will not load a file if it has already been loaded.

Creating Classes

Ruby has numerous built-in classes, and additional classes may be defined in a Ruby program. To define a new class, the following construct is used:

```
class ClassName
  # ...
end
```

The name of the class is itself a global constant and therefore must begin with an uppercase letter. The class definition can contain class constants, class variables, class methods, instance variables, and instance methods. Class data is available to all objects of the class, whereas instance data is only available to the one object. Here's an example:

```
class Friend
  @@myname = "Fred"             # a class variable

  def initialize(name, sex, phone)
    @name, @sex, @phone = name, sex, phone
    # These are instance variables
  end
  def hello                     # an instance method
    print "Hi, I'm #{ @name} .\n"
  end
  def Friend.our_common_friend  # a class method
    print "We are all friends of #{ @@myname} .\n"
  end
end
f1 = Friend.new("Susan","F","555-0123")
f2 = Friend.new("Tom","M","555-4567")
f1.hello                     # Hi, I'm Susan.
f2.hello                     # Hi, I'm Tom.
Friend.our_common_friend     # We are all friends of Fred.
```

Because class-level data is accessible throughout the class, it can be initialized at the time the class is defined. If a method named `initialize` is defined, it is guaranteed to be executed right after an instance is allocated. The `initialize` method is similar to the traditional concept of a constructor, but it does not have to handle memory allocation. Allocation is handled internally by `new`, and deallocation is handled transparently by the garbage collector.

Now consider this fragment, paying attention to the `getmyvar`, `setmyvar`, and `myvar=` methods:

```
class MyClass

  NAME = "Class Name"  # class constant

  def initialize       # called when object is allocated
    @@count += 1
    @myvar = 10
  end
```

```
    def MyClass.getcount # class method
      @@count            # class variable
    end

    def getcount         # instance returns class variable!
      @@count            # class variable
    end

    def getmyvar         # instance method
      @myvar             # instance variable
    end

    def setmyvar(val)    # instance method sets @myvar
      @myvar = val
    end
    def myvar=(val)      # Another way to set @myvar
      @myvar = val
    end
  end

  foo = MyClass.new      # @myvar is 10
  foo.setmyvar 20        # @myvar is 20
  foo.myvar = 30         # @myvar is 30
```

Here, you see that `getmyvar` returns the value of `@myvar`, and `setmyvar` sets it. (In the terminology of many programmers, these would be referred to as a *getter* and a *setter,* respectively.) These work fine, but they do not exemplify the Ruby way of doing things. The method `myvar=` looks like assignment overloading (although strictly speaking, it isn't); it is a better replacement for `setmyvar`, but there is a better way yet.

The class `Module` contains methods called `attr`, `attr_accessor`, `attr_reader`, and `attr_writer`. These can be used (with symbols as parameters) to automatically handle controlled access to the instance data. For example, the three methods named previously can be replaced by a single line in the class definition:

```
  attr_accessor :myvar
```

This will create the method `myvar`, which returns the value of `@myvar`, and the method `myvar=`, which enables the setting of the same variable. Methods `attr_reader` and `attr_writer` create read-only and write-only versions of an attribute, respectively. For more details, consult a Ruby reference.

Within the instance methods of a class, the pseudo-variable `self` can be used as needed. This is only a reference to the current receiver, the object on which the instance method is invoked.

The modifying methods `private`, `protected`, and `public` can be used to control the visibility of methods in a class. (Instance variables are always private and inaccessible from outside the class except by means of accessors.) Each of these modifiers takes a symbol such as `:foo` as a parameter; if this is omitted, the modifier applies to all subsequent definitions in the class. Here's an example:

```
  class MyClass
```

```
    def method1
      # ...
    end
    def method2
      # ...
    end
    def method3
      # ...
    end
    private :method1
    public  :method2
    protected :method3
    private

    def my_method
      # ...
    end
    def another_method
      # ...
    end
  end
```

In this example, `method1` will be private, `method2` will be public, and `method3` will be protected. Because of the `private` method with no parameters, both `my_method` and `another_method` will be private.

The `public` access level is self-explanatory; there are no restrictions on access or visibility. The `private` level means that the method is accessible only within the class or its subclasses, and it is callable only in "function form," with `self` (implicit or explicit) as a receiver. The `protected` level means that a method is callable only from within its class, but unlike a private method, it can be called with a receiver other than `self`, such as another instance of the same class.

The default visibility for the methods defined in a class is `public`. The exception is the instance-initializing method `initialize`, which is private because it is intended to be called only from the `new` method. Methods defined at the top level are also public by default; if they are private, they can be called only in function form (as, for example, the methods defined in `Object`).

Ruby classes are themselves objects, being instances of the metaclass `Class`. Ruby classes are always concrete; there are no abstract classes. However, it is theoretically possible to implement abstract classes in Ruby if you really wish to do so.

The class`Object` is at the root of the hierarchy. It provides all the methods defined in the built-in `Kernel` module.

To create a class that inherits from another class, define it in this way:

```
  class MyClass < OtherClass
    # ...
  end
```

In addition to using built-in methods, it is only natural to define your own and also to redefine and override existing ones. When you define a method with the same name as an existing

one, the previous method is overridden. If a method needs to call the "parent" method that it overrides (a frequent occurrence), the keyword `super` can be used for this purpose.
Operator overloading is not strictly an OOP feature, but it is very familiar to C++ programmers and certain others. Because most operators in Ruby are simply methods anyway, it should come as no surprise that these operators can be overridden or defined for user-defined classes. Overriding the meaning of an operator for an existing class may be rare, but it is common to want to define operators for new classes.
It is possible to create aliases or synonyms for methods. The syntax (used inside a class definition) is as follows:

```
alias newname oldname
```

The number of parameters will be the same as for the old name, and it will be called in the same way.

Methods and Attributes

In a previous section, methods were used with simple class instances and variables by separating the receivers from the methods with a period (*`receiver.method`*). In the case of method names that are punctuation, the period is omitted. Methods can take arguments:

```
Time.mktime( 2000, "Aug", 24, 16, 0 )
```

Because method calls return objects, method calls may typically be chained or stacked:

```
3.succ.to_s
      /(x.z).*?(x.z).*?/.match("x1z_1a3_x2z_1b3_").to_a[1..3]
      3+2.succ
```

Note that problems can arise if the cumulative expression is of a type that does not support that particular method. Specifically, some methods return `nil` under certain conditions, and this will usually cause any methods tacked onto that result to fail.
Certain methods may have blocks passed to them. This is true of all iterators, whether built in or user defined. A block is usually passed as a `do-end` block or a brace-delimited block; it is not treated like the other parameters preceding it, if any. See especially the `File.open` example:

```
my_array.each do |x|
  some_action
end

File.open(filename) {  |f| some_action }
```

Named parameters will be supported in the future but are not supported at the time of this writing. These are called *keyword arguments* in the Python realm.
Methods may take a variable number of arguments:

```
receiver.method(arg1, *more_args)
```

In this case, the method called will treat `more_args` as an array that it deals with as it would any other array. In fact, an asterisk in the list of formal parameters (on the last or only parameter) can likewise "collapse" a sequence of actual parameters into an array:

```
def mymethod(a, b, *c)
  print a, b
  c.each do |x| print x end
end

mymethod(1,2,3,4,5,6,7)     # a=1, b=2, c=[3,4,5,6,7]
```

Ruby has the ability to define methods on a per-object basis (rather than per class). Such methods are called *singletons;* they belong solely to that object and have no effect on its class or superclasses. As an example, this might be useful in programming a GUI; you can define a button action for a widget by defining a singleton method for the button object.
It is theoretically possible to create a prototype-based object system using singleton methods. This is a less traditional form of OOP without classes. The basic structuring mechanism is to construct a new object using an existing object as a delegate; the new object is exactly like old except for the items that are overridden. This enables you to build prototype/delegation-based systems rather than inheritance based. Although we do not have experience in this area, we do feel that this demonstrates the power of Ruby.

Dynamic Aspects of Ruby

Ruby is a very dynamic language in the sense that objects and classes may be altered at runtime. It has the ability to construct and evaluate pieces of code in the course of executing the existing statically coded program. It has a sophisticated reflection API that makes it very "self-aware"; this enables the easy creation of debuggers, IDEs, and similar tools, and it also makes certain advanced coding techniques possible.
This is perhaps the most difficult area a programmer will encounter in learning Ruby. Here, we briefly examine some of the implications of Ruby's dynamic nature.

Coding at Runtime

We have already mentioned `load` and `require` earlier. However, it is important to realize that these are not built-in statements or control structures or anything of that nature; they are actual methods. Therefore, it is possible to call them with variables or expressions as parameters or to call them conditionally. Contrast with this the `#include` directive in C and C++, which is evaluated and acted on at compile-time.
Ruby also enables the program to get access to the names of its own variables. Here's a variable called `foobar` assigned the value `3`; following that assignment, the `print` method prints out not only the value but the name of the variable:

```
foobar = 3
print "The value is ", foobar, "\n"
print "The variable name is ", :foobar.id2name, "\n"
```

Of course, this contrived example is not truly useful; the point is that the user's code can retrieve and manipulate internal names at will. Similar but much more sophisticated operations can be done with the reflection API, as you'll see in the next section.
Code can be constructed piecemeal and evaluated. As another contrived example, consider this `calculate` method and the code calling it:

```
def calculate(op1, operator, op2)
  string = op1.to_s + operator + op2.to_s
  # operator is assumed to be a string; make one big
  # string of it and the two operands
  eval(string)   # Evaluate and return a value
end

$alpha = 25
$beta = 12
puts calculate(2, "+", 2)          # Prints 4
puts calculate(5, "*", "$alpha")   # Prints 125
puts calculate("$beta", "**", 3)   # Prints 1728
```

As an even more extreme example, the following code will prompt the user for a method name and a single line of code; then it will actually define the method and call it:

```
puts "Method name: "
meth_name = gets.chomp
puts "Line of code: "
line = gets.chomp

# Build a string
string = %[def #{ meth_name} \n #{ line} \n end]
eval(string)       # Define the method
eval(meth_name)    # Call the method
```

Frequently, programmers wish to code for different platforms or circumstance and still maintain only a single code base. In such a case, a C programmer would use `#ifdef` directives; in Ruby, however, definitions are executed. There is no "compile time," and everything is dynamic rather than static. Therefore, if we want to make some kind of decision like this, we can simply evaluate a flag at runtime:

```
if platform == Windows
  action1
elsif platform == Linux
  action2
else
  default_action
end
```

Of course, there is a small runtime penalty for coding in this way, because the flag may be tested many times in the course of execution. However, the next example does essentially the same thing, enclosing the platform-dependent code in a method whose name is the same across all platforms:

```
if platform == Windows
  def my_action
    action1
  end
elsif platform == Linux
  def my_action
    action2
  end
else
  def my_action
    default_action
  end
end
```

In this way, the same result is achieved, but the flag is only evaluated *once*. When the user's code calls `my_action`, it will already have been defined appropriately.

Reflection

Languages such as Smalltalk, LISP, and Java implement the notion of a *reflective* programming language—one in which the active environment can query the objects that define it as well as extend or modify them at runtime.
Ruby allows reflection quite extensively but does not go as far as Smalltalk, which even represents control structures as objects. Ruby control structures and blocks are *not* objects (a `Proc` object can be used to "objectify" a block, but control structures are never objects).
The keyword `defined` (with an appended question mark) may be used to determine whether an identifier name is in use, as shown here:

```
if defined? some_var
  print "some_var = #{ some_var} \n"
else
  print "The variable some_var is not known.\n"
end
```

In most if not all cases, this is equivalent to comparing the variable to `nil`.
In a similar way, the method `respond_to?` determines whether an object can respond to the specified method call (that is, whether that method is defined for that object). The `respond_to?` method is defined in class `Object`.
Ruby supports runtime type information in a radical way. The type (or class) of an object can be determined at runtime using the method `type` (defined in `Object`). Similarly, `is_a?` will tell whether an object is of a certain class (including the superclasses), and `kind_of?` is the alias. Here's an example:

```
print "abc".type              # Prints String
print 345.type                # Prints Fixnum
rover = Dog.new
print rover.type              # Prints Dog
if rover.is_a? Dog
  print "Of course he is.\n"
end
if rover.kind_of? Dog
  print "Yes, still a dog.\n"
end
if rover.is_a? Animal
  print "Yes, he's an animal, too.\n"
end
```

It is possible to retrieve an exhaustive list of all the methods that can be invoked for a given object; this is done by using the `methods` method, defined in `Object`. There are also variations such as `private_instance_methods`, `public_instance_methods`, and so on.
In a similar way, you can determine the class variables and instance variables associated with an object. By the very nature of OOP, the lists of methods and variables include the entities defined not only in the object's class but in its superclasses. The `Module` class has a method called `constants` that's used to list all the constants defined.
The class `Module` has a method `ancestors` that will return a list of modules that are included in the given module. This list is self-inclusive; `Mod.ancestors` will always have at least `Mod` in the list. The class `Object` has a method called `superclass` that returns the superclass of the object or returns `nil`. Because `Object` itself is the only object without a superclass, it is the only case in which `nil` will be returned.
The `ObjectSpace` module is used to access any and all "living" objects. The method `_idtoref` can be used to convert an object ID to an object reference; it can be considered the inverse of the colon notation. `ObjectSpace` also has an iterator called `each_object` that will iterate over all the objects currently in existence, including many that you will not otherwise explicitly know about. (Remember that certain small immutable objects, such as objects of class `Fixnum`, `NilClass`, `TrueClass`, and `FalseClass` are not kept on the stack for optimization reasons.)

Missing Methods

When a method is invoked (`myobject.mymethod`), Ruby first searches for the named method according to this search order:

1. Singleton methods in the receiver `myobject`.
2. Methods defined in `myobject`'s class.
3. Methods defined among `myobject`'s ancestors.

If the method `mymethod` is not found, Ruby searches for a default method called `method_missing`. If this method is defined, it is passed the name of the missing method (as a symbol) and all the parameters that were passed to the nonexistent `mymethod`.

Garbage Collection

Managing memory on a low level is hard and error prone, especially in a dynamic environment such as Ruby. Having a garbage-collection facility is a very significant advantage. In languages such as C++, memory allocation and deallocation are handled by the programmer; in more recent languages such as Java, memory is reclaimed (when objects go out of scope) by a garbage collector.

Memory management done by the programmer is the source of two of the most common kinds of bugs. If an object is freed while still being referenced, a later access may find the memory in an inconsistent state. These so-called "dangling pointers" are difficult to track down because they often cause errors in code that is far removed from the offending statement. A related bug is a "memory leak," caused when an object is not freed even though there are no references to it. Programs with this bug typically use up more and more memory until they crash; this kind of error is also difficult to find. Ruby uses a GC facility that tracks down unused objects and reclaims the storage that was allocated to them. For those who care about such things, Ruby's GC is done using a "mark and sweep" algorithm rather than reference counting (which frequently has difficulties with recursive structures).

Certain performance penalties may be associated with garbage collection. There are some limited controls in the GC module so that the programmer can tailor garbage collection to the needs of the individual program.

Training Your Intuition: Things to Remember

It may truly be said that "everything is intuitive once you understand it." This verity is the heart of this section, because Ruby has many features and personality quirks that may be different from what the traditional programmer is used to.

Some readers may feel their time is wasted by a reiteration of some of these points; if that is the case for you, you are free to skip the paragraphs that seem obvious to you. Programmers' backgrounds vary widely; an old-time C hacker and a Smalltalk guru will each approach Ruby from different viewpoints. We hope, however, that a perusal of these following paragraphs will assist many readers in following what some call the Ruby Way.

Syntax Issues

The Ruby parser is very complex and relatively forgiving. It tries to make sense out of what it finds, rather than forcing the programmer into slavishly following a set of rules. However, this behavior may take some getting used to. Here's a list of things you should know about Ruby syntax:

- Parentheses are usually optional with a method call. These calls are all valid:

```
foobar

foobar()

foobar(a,b,c)

foobar a, b, c
```

- Given that parentheses are optional, what does `x y z` mean, if anything? As it turns out, this means, "Invoke method `y`, passing `z` as a parameter, and then pass the result as a parameter to method `x`." In short, the statement `x(y(z))` means the same thing.
- Let's try to pass a hash to a method: `my_method {a=>1, b=>2}`
  This results in a syntax error, because the left brace is seen as the start of a block. In this instance, parentheses are necessary: `my_method({a=>1, b=>2})`
- Now let's suppose that the hash is the *only* parameter to a method. Ruby very forgivingly lets us omit the braces: `my_method(a=>1, b=>2)`
  Some people might think that this looks like a method invocation with named parameters, which it emphatically is not.
- Now consider this method call:

  ```
  foobar.345
  ```

  Looking at it, one might think that `foobar` is an object and `345` is a method being invoked, but obviously a method name can't start with a digit! The parser interprets this as a call to method `foobar`, passing the number `0.345` as a parameter. Here, you see that the parentheses and the intervening space have all been omitted. Needless to say, the fact that you can code this way does not imply that you should.
- There are other cases in which blank spaces are somewhat significant. For example, these expressions may all seem to mean the same:

  ```
  x = y + z

  x = y+z
  x = y+ z

  x = y +z
  ```

  In fact, the first three do mean the same. However, in the fourth case, the parser thinks that `y` is a method call and `+z` is a parameter passed to it! It will then give an error message for that line if there is no method named `y`. The moral is to use blank spaces in a reasonable way.
- Similarly, `x = y*z` is a multiplication of `y` and `z`, whereas `x = y *z` is an invocation of method `y`, passing an expansion of array `z` as a parameter.
- In constructing identifiers, the underscore is considered to be lowercase. Therefore, an identifier may start with an underscore, but it will *not* be a constant even if the next letter is uppercase.

- In linear, nested `if` statements, the keyword `elsif` is used rather than `else if` or `elif`, as in some languages.
- Keywords in Ruby are not really "reserved words." In many circumstances, a keyword can actually be used as an identifier as long as the parser is not confused. We won't attempt to state the conditions under which this may and may not be done; we mention this only to say that it can often be done if you really need to do it—and as a warning to those who might be confused by this. In general, using a keyword as an identifier should be done with caution, keeping readability in mind.
- The keyword`then` is optional (in `if` and `case` statements). Those who wish to use it for readability may do so. The same is true for `do` in `while` and `until` loops.
- The question mark and exclamation point are not really part of the identifier that they modify but should be considered as suffixes. Therefore, although `chop` and `chop!`, for example, are considered different identifiers, it is not permissible to use these characters in any other position in the word. Likewise, we use `defined?` in Ruby, but `defined` is the keyword.
- Inside a string, the pound sign (`#`) is used to signal expressions to be evaluated. That means that in some circumstances, when a pound sign occurs in a string, it has to be escaped with a backslash, but this is *only* when the next character is a left brace (`{`), a dollar sign (`$`), or an "at" sign (`@`).
- The ternary decision operator (`?:`), which originated in the C language, has sometimes been said to be "undocumented" in Ruby. For this reason, programmers may wish not to use it (though we personally do not shy away from it).
- Because of the fact that the question mark may be appended to an identifier, care should be taken with spacing around the ternary operator. For example, suppose we have the variable `my_flag`, which stores either `true` or `false`. Then the first line of code shown here will be correct, but the second will give a syntax error:

```
x = my_flag ? 23 : 45   # OK

x = my_flag? 23 : 45    # Syntax error
```

- The ending marker `=end` for embedded documentation should not be considered a token. It marks the entire line; therefore, any characters on the rest of that line are not considered part of the program text but belong to the embedded document.
- There are no arbitrary blocks in Ruby; that is, you can't start a block whenever you feel like it, as in C. Blocks are allowed only where they are needed (for example, attached to an iterator). That is why any post-test loops in Ruby are kludged by using a `begin-end` pair even though no exception handling is being done.
- Remember that the keywords `BEGIN` and `END` are completely different from the `begin` and `end` keywords.
- When strings bump together (static concatenation), the concatenation is of a higher precedence than a method call. Here's an example:

```
# These three all give the same result.
str1 = "First " 'second'.center(20)
str2 = ("First " + 'second').center(20)
str3 = "First second".center(20)
```

Precedence is different.

- Ruby has several pseudo-variables that look like local variables but really serve specialized purposes. These are `self`, `nil`, `true`, `false`, `__FILE__`, and `__LINE__`.

Perspectives in Programming

Presumably everyone who knows Ruby (at this point in time) has been a student or user of other languages in the past. This of course makes learning Ruby easy, in the sense that numerous features in Ruby are just like the corresponding features in other languages. On the other hand, the programmer may be lulled into a false sense of security by some of the familiar constructs in Ruby and may draw unwarranted conclusions based on past experience —which we might term *geek baggage*.

Many people are coming to Ruby from Smalltalk, Perl, C/C++, and various other languages. Their presuppositions and expectations may all vary somewhat, but they will always be present. For this reason, here are a few of the things that some programmers may "trip over" in using Ruby:

- A character in Ruby truly is an integer. It is not a type of its own, as in Pascal, and is not the same as a string of length 1. Consider the following code fragment:

```
x = "Hello"
y = ?A
print "x[0] = #{ x[0]} \n"  # Prints: x[0] = 72
print "y = #y\n"           # Prints: y = 65
if y == "A"                # Prints: no
  print "yes\n"
else
  print "no\n"
end
```

- There is no Boolean type such as many languages have. `TrueClass` and `FalseClass` are distinct classes, and their only instantiations are `true` and `false`. Many of Ruby's operators are similar or identical to those in C. Two notable exceptions are the increment and decrement operators (`++` and `--`). These are not available in Ruby, either in "pre" or "post" forms.
- The modulus operator is known to work somewhat differently in different languages with respect to negative numbers. The two sides of this argument are beyond the scope of this book; suffice to say that Ruby's behavior is as follows:

```
print  5 %  3  # Prints 2
print -5 %  3  # Prints 1
print  5 % -3  # Prints -1
print -5 % -3  # Prints -2
```

- Some may be used to thinking that a false value may be represented as a zero, a null string, a null character, or various other things. However, in Ruby, all of these are true; in fact, *everything* is true except `false` and `nil`.
- Always recall that in Ruby, variables don't have types; only values have types.
- To say that a value is undefined (for example, a variable not declared) is essentially the same as saying that it is `nil`. Such a value will pass a test for equality with `nil` and will evaluate to `false` if used by itself in a condition. The principle exception relates to hashes; because `nil` is a valid value to be stored in a hash, it is not appropriate to compare against `nil` to find whether a value exists in a hash. (There are several correct ways to perform this test by means of method calls.)
- Recall that a post-test loop can be faked in Ruby by using a `begin-end` construct followed by the "modifier" form of `while` or `until`.
- Recall that there are no declarations of variables in Ruby. It is good practice, however, to assign `nil` to a variable initially. This certainly does not assign a type to the variable and does not truly initialize it, but it does inform the parser that this is a variable name rather than a method name. Ruby interprets an identifier as a method name unless it has seen a previous assignment indicating that the name refers to a variable.
- Recall that `ARGV[0]` is truly the first of the command-line parameters, numbering naturally from zero; it is not the file or script name preceding the parameters, such as `argv[0]` in C.
- Most of Ruby's operators are really methods; the "punctuation" form of these methods is provided for familiarity and convenience. The first exception is the set of reflexive assignment operators (`+=`, `-=`, `*=`, and so on); the second exception is the following set:

```
= .. ... ! not && and || or != !~
```

- Like most (though not all) modern languages, Boolean operations are always short-circuited; that is, the evaluation of a Boolean expression stops as soon as its truth value is known. In a sequence of `or` operations, the first `true` will stop evaluation; in a string of `and` operations, the first `false` will stop evaluation.
- Recall that the prefix `@@` is used for class variables (which are associated with the class rather than the instance).
- Recall that `loop` is not a keyword; it is a `Kernel` method, not a control structure.
- Some may find the syntax of `unless-else` to be slightly unintuitive. Because `unless` is the opposite of `if`, the `else` clause will be executed if the condition is *false*.
- Ordinarily a parameter passed to a method is really a reference to an object; as such, the parameter can potentially be changed from within the method.
- The simpler `Fixnum` type is passed as an immediate value and therefore may not be changed from within methods. The same is true for `true`, `false`, and `nil`.

- Do not confuse the `&&` and `||` operators with the `&` and `|` operators. These are used as in C; the former are for Boolean operations, and the latter are for arithmetic or bitwise operations.
- There are interesting differences between the `&&-||` operators and the `and-or` operators. The former are more general purpose and may result in an expression other than `true` or `false`. The latter always result in `true` or `false`; they are specifically for joining Boolean expressions in conditions (and therefore are susceptible to syntax errors if an operand does not evaluate to `true` or `false`). See the following code fragment:

```
print (false || "string1\n")     # Prints string1
# print (false or "string2\n")   #   Syntax error!
print (true && "string3\n")      # Prints string3
# print (true and "string4\n")   #   Syntax error!
print (true || "string5\n")      # Prints true
# print (true or "string6\n")    #   Syntax error!
print (false && "string5\n")     # Prints false
# print (false or "string6\n")   #   Syntax error!
```

- The`and-or` operators also have lower precedence than the `&&-||` operators. See the following code fragment:

```
a = true
b = false
c = true
d = true
a1 = a && b or c && d    # &&'s are done first
a2 = a && (b or c) && d  # or is done first
print a1                 # Prints false
print a2                 # Prints true
```

- Additionally, be aware that the assignment operator has a *higher* precedence than the `and` and `or` operators! (This is also true for the reflexive assignment operators +=, -=, and the others.) For example, line 3 of the following code looks like a normal assignment statement, but it is really a free-standing expression (equivalent to line 5, in fact). Line 7 is a real assignment statement, which may be what the programmer really intends:

```
y = false
z = true
x = y or z     # Line 3: = is done BEFORE or!
print x, "\n"  # Prints false
(x = y) or z   # Line 5: Same as line 3
print x, "\n"  # Prints false
x = (y or z)   # Line 7: or is done first
print x, "\n"  # Prints true
```

- Don't confuse object attributes and local variables. If you are accustomed to C++ or Java, you might forget this. The variable `@my_var` is an instance variable (or attribute) in the context of whatever class you are coding; but `my_var`, used in the same circumstance, is only a local variable within that context.
- Many languages have some kind of `for` loop, as does Ruby. The question sooner or later arises as to whether the index variable can be modified. Some languages do not allow

the control variable to be modified at all (printing a warning or error either at compile time or runtime); and some will cheerfully allow the loop behavior to be altered in midstream by such a change. Ruby takes yet a third approach. When a variable is used as a `for` loop control variable, it is an ordinary variable and can be modified at will; however, such a modification does not affect the loop behavior! The `for` loop sequentially assigns the values to the variable on each iteration without regard for what may have happened to that variable inside the loop. For example, this loop will execute exactly 10 times and print the values `1` through `10`:

```
for var in 1..10
  print "var = #{ var} \n"
  if var > 5
    var = var + 2
  end
end
```

- Recall that variable names and method names are not always distinguishable "by eye" in the immediate context. How does the parser decide whether an identifier is a variable or a method? The rule is that if the parser sees the identifier being assigned a value prior to its being used, it will be considered a variable; otherwise, it is considered to be a method name.
- The `while` and `until` modifiers are *not* post-test loops. These two loops will not execute:

```
puts "looping" while false
puts "still looping" until true
```

Ruby's `case` Statement

Every modern language has some kind of multiway branch, such as the `switch` statement in C/C++ and Java or the `case` statement in Pascal. These serve basically the same purpose, and they function much the same in most languages.

Ruby's `case` statement is superficially similar to these others, but on closer examination it is so unique that it makes C and Pascal look like close friends. The `case` statement in Ruby has no precise analogue in any other language that we (the authors) are familiar with, and this makes it worth additional attention here.

You have already seen the syntax of this statement. We will concentrate here on its actual semantics:

- To begin with, consider the trivial `case` statement shown here. The expression shown is compared with the value, not surprisingly, and if they correspond, `some_action` is performed:

```
case expression
  when value
```

```
    some_action
end
```

But what do we mean by "compare" and "correspond"? As it turns out, Ruby uses the special operator === (sometimes called the *relationship operator*) for this. This operator is also referred to (somewhat inappropriately) as the *case equality operator*. Therefore, the preceding simple statement is equivalent to this statement:

```
if value === expression
  some_action
end
```

However, do not confuse the relationship operator with the equality operator (==). They are utterly different, although their behavior may be the same in many circumstances. The relationship operator is defined differently for different classes, and for a given class, it may behave differently for different operand types passed to it.

- Also, do not fall into the trap of thinking that the tested expression is the receiver and the value is passed as a parameter to it. The opposite is true.
- This brings up the fact that `x === y` is *not* typically the same as `y === x`! There will be situations in which this is true, but overall the relationship operator is not commutative. (That is why we do not favor the term *case equality operator,* because equality is always commutative.) In other words, reversing our original example, this code does not behave the same way:

```
case value
  when expression
    some_action
end
```

- As an example, consider the string `str` and the pattern (regular expression) `pat`, which matches that string. The expression `str =~ pat` is true, just as in Perl. Because Ruby defines the opposite meaning for =~ in Regexp, one can also say that `pat =~ str` is true. Following this logic further, we find that (because of how `Regexp::===` is defined) `pat === str` is also true. However, note that `str === pat` is *not* true. This means that the code fragment

```
case "Hello"
  when /Hell/
    print "We matched.\n"
  else
    print "We didn't match.\n"
end
```

does not do the same thing as this fragment:

```
case /Hell/
  when "Hello"
    print "We matched.\n"
  else
```

```
    print "We didn't match.\n"
end
```

If this confuses you, just memorize the behavior. If it does not confuse you, so much the better.

- Programmers accustomed to C may be puzzled by the absence of `break` statements in the `case` statement; such a usage of `break` in Ruby is unnecessary (and illegal). This is due to the fact that "falling through" is very rarely the desired behavior in a multiway branch. There is an implicit jump from each `when` clause (or *case limb,* as it is sometimes called) to the end of the `case` statement. In this respect, Ruby's `case` statement resembles the one in Pascal.
- The values in each case limb are essentially arbitrary. They are not limited to any certain type. They need not be constants but can be variables or complex expressions. Ranges or multiple values can be associated with each case limb.
- Case limbs may have empty actions (null statements) associated with them. The values in the limbs need not be unique but may overlap. Look at this example:

```
case x
  when 0
  when 1..5
    print "Second branch\n"
  when 5..10
    print "Third branch\n"
  else
    print "Fourth branch\n"
end
```

  Here, a value of `0` for `x` will do nothing, and a value of `5` will print `Second branch`, even though `5` is also included in the next limb.
- The fact that case limbs may overlap is a consequence of the fact that they are evaluated in sequence *and* that short-circuiting is done. In other words, if evaluation of the expressions in one limb results in success, then the limbs that follow are never evaluated. Therefore, it is a bad idea for case limb expressions to have method calls that have side effects. (Of course, such calls are questionable in most circumstances anyhow.) Also, be aware that this behavior may mask runtime errors that would occur if expressions were evaluated. Here's an example:

```
case x
  when 1..10
    print "First branch\n"
  when foobar()              # Possible side effects?
    print "Second branch\n"
  when 5/0                   # Dividing by zero!
    print "Third branch\n"
  else
    print "Fourth branch\n"
end
```

As long as `x` is between `1` and `10`, `foobar()` will not be called, and the expression `5/0` (which would naturally result in a runtime error) will not be evaluated.

Rubyisms and Idioms

Much of this material will overlap conceptually with the preceding pages. Don't worry too much about why we divided it as we did; many of these tidbits are hard to classify or organize. Our most important motivation is simply to break the information into digestible chunks.

Ruby was designed to be consistent and orthogonal. However, it is also a very complex entity. Therefore, like every language, it has its own set of idioms and quirks. We discuss some of these here:

- Remember that `alias` can be used to give alternate names for global variables and methods. Remember that the numbered global variables `$1`, `$2`, `$3`, and so on cannot be aliased.
- We do not recommend the use of the "special variables," such as `$=`, `$_`, `$/`, and the rest. Although they can sometimes make code more compact, they rarely make it any clearer; we use them very sparingly in this book and recommend the same practice. In many cases, the names can be clarified by using the `English.rb` library; in other cases, a more explicit coding style makes them unnecessary.
- Do not confuse the .. and ... range operators. The former is *inclusive* of the upper bound, and the latter is *exclusive*. For example, `5..10` includes the number `10`, but `5...10` does not.
- There is a small detail relating to ranges that may cause slight confusion. Given the range `m..n`, the method `end` will return the endpoint of the range; its alias, `last`, will do the same thing. However, these methods will return the same value, `n`, for the range `m...n`, even though `n` is not included in the latter range. The method `end_excluded?` is provided to distinguish between these two situations.
- Do not confuse ranges with arrays. These two assignments are entirely different:

  ```
  x = 1..5

  x = [1, 2, 3, 4, 5]
  ```

  However, there is a convenient method, `to_a`, for converting ranges to arrays. (Many other types also have such a method.)
- Keep a clear distinction in your mind between *class* and *instance*. For example, a class variable such as `@@foobar` has a class-wide scope, but an instance variable such as `@foobar` has a separate existence in each object of the class.
- Similarly, a class method is associated with the class in which it is defined; it does not belong to any specific object and cannot be invoked as though it did. A class method is invoked with the name of a class, and an instance method is invoked with the name of an object.

- In writing about Ruby, the "pound notation" is sometimes used to indicate an instance method—for example, we use `File.chmod` to denote the class method `chmod` of class `File`, and we use `File#chmod` to denote the instance method that has the same name. This notation is not part of Ruby syntax, but only Ruby folklore. We have tried to avoid it in this book.
- In Ruby, constants are not truly *constant*. They cannot be changed from within instance methods, but otherwise their values *can* be changed.
- In writing about Ruby, the word *toplevel* is common as both an adjective and a noun. We prefer to use *top level* as a noun and *top-level* as an adjective, but our meaning is the same as everyone else's.
- The keyword `yield` comes from CLU and may be misleading to some programmers. It is used within an iterator to invoke the block with which the iterator is called. It does not mean "yield," as in producing a result or returning a value, but is more like the concept of "yielding a timeslice."
- Remember that the reflexive assignment operators `+=`, `-=`, and the rest are not methods (nor are they really operators); they are only "syntax sugar" or "shorthand" for their longer forms. Therefore, to say `x += y` is really identical to saying `x = x + y`, and if the `+` operator is overloaded, the `+=` operator is defined "automagically" as a result of this predefined shorthand.
- Because of the way the reflexive assignment operators are defined, they cannot be used to initialize variables. If the first reference to `x` is `x += 1`, an error will result. This will be intuitive to most programmers, unless they are accustomed to a language where variables are initialized to some sort of zero or null value.
- It is actually possible in some sense to get around this behavior. One can define operators for `nil` such that the initial `nil` value of the variable produces the desired result. Here is a method (`nil.+`) that will allow `+=` to initialize a `String` or a `Fixnum` value, basically just returning `other` and thus ensuring that `nil + other` is equal to `other`:

  ```
  def nil.+(other)
    other
  end
  ```

  This illustrates the power of Ruby, but whether it is useful or appropriate to code this way is left as an exercise for the reader.
- It is wise to recall that `Class` is an *object* and that `Object` is a *class*. We will try to make this clear in a later chapter; for now, simply recite it every day as a mantra.
- Some operators can't be overloaded because they are built in to the language rather than implemented as methods. These operators are as follows:

  ```
  =   ..   ...   and   or   not   &&   ||   !   !=   !~
  ```

Additionally, the reflexive assignment operators (`+=`, `-=`, and so on) cannot be overloaded. These are not methods, and it can be argued they are not true operators either.

- Be aware that although assignment is not overloadable, it is still possible to write an instance method with a name such as `foo=` (thus allowing statements such as `x.foo = 5`). Consider the equal sign to be like a suffix.
- Recall that a "bare" scope operator has an implied `Object` before it, so that `::Foo` means `Object::Foo`.
- Recall that `fail` is an alias for `raise`.
- Recall that definitions in Ruby are executed. Because of the dynamic nature of the language, it's possible, for example, to define two methods completely differently based on a flag that is tested at runtime.
- Remember that the `for` construct (`for x in a`) is really calling the default iterator `each`. Any class having this iterator can be walked through with a `for` loop.
- Recall that the term *iterator* is sometimes a misnomer. Any method that invokes a block passed as a parameter is an iterator. Some of the predefined ones do not really look like looping mechanisms at all (see `File.open`).
- Be aware that a method defined at the top level is a member of `Object`.
- A setter method (such as `foo=`) must be called with a receiver; otherwise, it will look like a simple assignment to a local variable of that name.
- Recall that `retry` can be used in iterators but not in general loops. In iterators, it causes the reassignment of all the parameters and the restarting of the current iteration.
- The keyword `retry` is also used in exception handling. Don't confuse the two usages.
- An object's `initialize` method is always private.
- Where an iterator ends in a left brace (or in `end`) and results in a value, that value can be used as the receiver for further method calls. Here's an example:

```
squares = [1,2,3,4,5].collect do |x| x**2 end.reverse
# squares is now [25,16,9,4,1]
```

- The idiom `if $0 == __FILE__` is sometimes seen near the bottom of a Ruby program. This is a check to see whether the file is being run as a standalone piece of code (`true`) or is being used as some kind of auxiliary piece of code such as a library (`false`). A common use of this is to put a sort of "main program" (usually with test code in it) at the end of a library.
- Recall that normal subclassing or inheritance is done with the < symbol:

```
class Dog < Animal
   # ...
end
```

However, creation of a singleton class (an anonymous class that extends a single instance) is done with the << symbol:

```
class << platypus
  # ...
end
```

- When a block is passed to an iterator, the difference between braces (`{ }`) and a `do-end` pair is a matter of precedence, as shown here:

```
mymethod param1, foobar do ... end
# Here, do-end binds with mymethod
mymethod param1, foobar {  ... }
# Here, { }  binds with foobar, assumed to be a method
```

- It is somewhat traditional in Ruby to put single-line blocks in braces and multiline blocks in `do-end` pairs. Here are examples:

```
my_array.each { |x| print x, "\n"}

my_array.each do |x|
  print x
  if x % 2 == 0
    print " is even\n"
  else
    print " is odd\n"
  end
end
```

  This habit is not required, and there may conceivably be occasions where it is inappropriate to follow this rule.
- Bear in mind that strings are in a sense two-dimensional; they can be viewed as sequences of characters or sequences of lines. Some may find it surprising that the default iterator `each` operates on lines (where a *line* is a group of characters terminated by a record separator that defaults to newline); an alias for `each` is `each_line`. If you want to iterate by characters, you can use `each_byte`. The iterator `sort` also works on a line-by-line basis. There is no iterator called `each_index` because of the ambiguity involved—do we want to handle the string by character or by line? This all becomes habitual with repeated use.
- A closure remembers the context in which it was created. One way to create a closure is by using a `Proc` object. As a crude example, consider the following:

```
def power(exponent)
  proc { |base| base**exponent}
end

square = power(2)
cube   = power(3)

a = square(11)    # Result is 121
b = square(5)     # Result is  25
c = cube(6)       # Result is 216
d = cube(8)       # Result is 512
```

Observe that the closure "knows" the value of exponent that it was given at the time it was created.

- However, let's assume that a closure uses a variable defined in an outer scope (which is perfectly legal). This property can be useful, but here we show a misuse of it:

```
$exponent = 0

def power
  proc { |base| base**$exponent}
end

$exponent = 2
square = power
$exponent = 3
cube   = power

a = square.call(11)    # Wrong! Result is 1331
b = square.call(5)     # Wrong! Result is  125
# The above two results are wrong because the CURRENT value
# of $exponent is being used, since it is still in scope.
c = cube.call(6)       # Result is 216
d = cube.call(8)       # Result is 512
```

- Finally, consider this somewhat contrived example. Inside the block of the `times` iterator, a new context is started, so that `x` is a local variable. The variable `closure` is already defined at the top level, so it will not be defined as local to the block:

```
closure = nil   # Define closure so the name will be known
1.times {        # Start a new context
  x = 5         # x is local to this block
  closure = Proc.new {
    print "In closure, x = #{ x} \n"
  }
}

x = 1           # Define x at top level

closure.call    # Prints: In closure, x = 5
```

Now note that the variable `x` that is set to `1` is a new variable, defined at the top level. It is not the same as the other variable of the same name. The closure therefore prints `5` because it remembers its creation context, with the previous variable `x` and its previous value.

- Variables starting with a single `@`, defined inside a class, are generally instance variables. However, if they are defined outside of any method, they are really *class instance* variables. (This usage is somewhat contrary to most OOP terminology, in which a class instance is regarded to be the same as an instance or an object.) Here's an example:

```
class Myclass
  @x = 1       # A class instance variable
  @y = 2       # Another one
```

```
     def mymethod
       @x = 3     # An instance variable
       # Note that @y is not accessible here.
     end

   end
```

The preceding class instance variable `@y` is really an attribute of the class object `Myclass`, which is an instance of the class `Class`. (Remember, `Class` is an object and `Object` is a class.) Class instance variables cannot be referenced from within instance methods and, in general, are not very useful.

- Remember that `attr`, `attr_reader`, `attr_writer`, and `attr_accessor` are shorthand for the actions of defining setters and getters; they take symbols as arguments.
- Remember that there is never any assignment with the scope operator; for example, the assignment `Math::PI = 3.2` is illegal.
- Note that closures have to return values implicitly (by returning the value of the last expression evaluated). The `return` statement can be used only in actual method returns.
- A closure is associated with a block at the time it is created. Therefore, it is never useful to associate a block with the `call` method; this will result in a warning.
- To recognize an identifier as a variable, Ruby only has to *see* an assignment to it; the assignment does not have to be executed. This can lead to a seeming paradox, as shown here:

  ```
  name = "Fred" if ! defined? name
  ```

  Here, the assignment to `name` is seen, so that the variable is defined. Because it is defined (and the test is false), it is never assigned, and it will be `nil` after this statement is executed.
- Some of the "bang" methods (with names ending in an exclamation point) behave in a slightly confusing way. Normally they return `self` as a return value, but some of them return `nil` in certain circumstances (to indicate that no work was actually done). In particular, this means that these cannot always be chained safely. Here's an example:

  ```
  str = "defghi"
  str.gsub!(/def/,"xyz").upcase!
  #  str is now "XYZGHI"

  str.gsub!(/abc/,"klm").downcase!
  # Error (since nil has no downcase! method)
  ```

  Other such methods are `sort!` and `sub!` (although `sort!` may change soon).

Expression Orientation and Other Miscellaneous Issues

In Ruby, expressions are nearly as significant as statements. If you are a C programmer, this will be of some familiarity to you; if your background is in Pascal, it may seem utterly foreign. However, Ruby carries expression orientation even further than C.
In addition, we use this section to remind you of few little issues regarding regular expressions. Consider them to be tiny bonuses:

- In Ruby, any kind of assignment returns the same value that was assigned. Therefore, we can sometimes take little shortcuts, as shown here:

```
x = y = z = 0     # All are now zero.
a = b = c = []    # Danger! a, b, and c now all refer
                  #   to the SAME empty array.
x = 5
y = x += 2        # Now x and y are both 7
```

  Be very careful when you are dealing with objects! Remember that these are nearly always *references* to objects.
- Many control structures, such as `if`, `unless`, and `case`, return values. The code shown here is all valid; it demonstrates that the branches of a decision need not be statements but can simply be expressions:

```
a = 5
x = if a < 8 then 6 else 7 end  # x is now 6
y = if a < 8                    # y is 6 also; the
      6                         # if-statement can be
    else                        # on a single line
      7                         # or on multiple lines.
    end
# unless also works; z will be assigned 4
z = unless x == y then 3 else 4 end
t = case a                      # t gets assigned
      when 0..3                 # the value
        "low"                   # "medium"
      when 4..6
        "medium"
      else
        "high"
    end
```

- Note, however, that the `while` and `until` loops do *not* return usable values. For example, this fragment is not valid:

```
i  = 0
x = while (i < 5)      # Error!
      print "#{ i} \n"
    end
```

- Note that the ternary decision operator can be used with statements or expressions. For syntactic reasons, the parentheses here are necessary:

```
x = 6
y = x == 5 ? 0 : 1                      # y is now 1
x == 5 ? print("Hi\n") : print("Bye\n") # Prints Bye
```

- The `return` at the end of a method can be omitted. A method will always return the last expression evaluated in its body, regardless of where that happens.
- When an iterator is called with a block, the last expression evaluated in the block will be returned as the value of the block. Therefore, if the body of an iterator has a statement such as `x = yield`, that value can be captured.
- When necessary, we can use parentheses to convert a statement into an expression, as shown here:

```
a = [1, 2, 3 if x==0]      # Illegal syntax
a = [1, 2, (3 if x==0)]    # Valid syntax

mymeth(a, b, if x>5 then c else d end)   # Illegal
mymeth(a, b, (if x>5 then c else d end)) # Valid
```

- For regular expressions, recall that the multiline modifier `/m` can be appended to a regex, in which case a dot `(.)` will match a newline character.
- For regular expressions, beware of zero-length matches. If all elements of a regex are optional, then "nothingness" will match that pattern, and a match will always be found at the very beginning of a string. This is a common error for regex users, particularly novices.