When we originally wrote this book, we had a grand plan (we were younger then). We wanted to document the language from the top down, starting with classes and objects, and ending with the nitty-gritty syntax details. It seemed like a good idea at the time. After all, most everything in Ruby is an object, so it made sense to talk about objects first.

Or so we thought.

Unfortunately, it turns out to be difficult to describe a language that way. If you haven't covered strings, if statements, assignments, and other details, it's difficult to write examples of classes. Throughout our top-down description, we kept coming across low-level details we needed to cover so that the example code would make sense.

So, we came up with another grand plan (they don't call us pragmatic for nothing). We'd still describe Ruby starting at the top. But before we did that, we'd add a short chapter that described all the common language features used in the examples along with the special vocabulary used in Ruby, a kind of minitutorial to bootstrap us into the rest of the book.

Ruby Is an Object-Oriented Language

Let's say it again. Ruby is a genuine object-oriented language. Everything you manipulate is an object, and the results of those manipulations are themselves objects. However, many languages make the same claim, and they often have a different interpretation of what object-oriented means and a different terminology for the concepts they employ.

So, before we get too far into the details, let's briefly look at the terms and notation that we'll be using.

When you write object-oriented code, you're normally looking to model concepts from the real world in your code. Typically during this modeling process you'll discover categories of things that need to be represented in code. In a jukebox, the concept of a "song" might be such a category. In Ruby, you'd define a class to represent each of these entities. A class is a combination of state (for example, the name of the song) and methods that use that state (perhaps a method to play the song).

Once you have these classes, you'll typically want to create a number of instances of each. For the jukebox system containing a class called Song, you'd have separate instances for popular hits such as "Ruby Tuesday," "Enveloped in Python," "String of Pearls," "Small talk," and so on. The word object is used interchangeably with class instance (and being lazy typists, we'll probably be using the word "object" more frequently).

In Ruby, these objects are created by calling a constructor, a special method associated with a class. The standard constructor is called new.

```ruby
song1 = Song.new("Ruby Tuesday")
song2 = Song.new("Enveloped in Python")
# and so on
```

These instances are both derived from the same class, but they have unique characteristics. First, every object has a unique object identifier (abbreviated as object id). Second, you can define instance variables, variables with values that are unique to each instance. These instance variables hold an object's state. Each of our songs, for example, will probably have an instance variable that holds the song title.

Within each class, you can define instance methods. Each method is a chunk of functionality which may be called from within the class and (depending on accessibility constraints) from outside. These instance methods in turn have access to the object's instance variables, and hence to the object's state.

Methods are invoked by sending a message to an object. The message contains the method's name, along with any parameters the method may need. When an object receives a message, it looks into its own class for a corresponding method. If found, that method is executed. If the method isn't found, well, we'll get to that later.

This business of methods and messages may sound complicated, but in practice it is very natural. Let's look at some method calls. (Remember that the arrows in the code examples show the values returned by the corresponding expressions.)
"gin joint".length » 9
"Rick".index("c") » 2
-1942.abs » 1942
sam.play(aSong) » "duh dum, da dum de dum ..."

Here, the thing before the period is called the receiver, and the name after the period is the method to be invoked. The first example asks a string for its length, and the second asks a different string to find the index of the letter ‘c.’ The third line has a number calculate its absolute value. Finally, we ask Sam to play us a song.

It’s worth noting here a major difference between Ruby and most other languages. In (say) Java, you’d find the absolute value of some number by calling a separate function and passing in that number. You might write

```java
number = Math.abs(number) // Java code
```

In Ruby, the ability to determine an absolute value is built into numbers—they take care of the details internally. You simply send the message abs to a number object and let it do the work.

```ruby
number = number.abs
```

The same applies to all Ruby objects: in C you’d write `strlen(name)`, while in Ruby it’s `name.length`, and so on. This is part of what we mean when we say that Ruby is a genuine OO language.

### Some Basic Ruby

Not many people like to read heaps of boring syntax rules when they’re picking up a new language. So we’re going to cheat. In this section we’ll hit some of the highlights, the stuff you’ll just have to know if you’re going to write Ruby programs. Later, in Chapter 18, which begins on page 199, we’ll go into all the gory details.

Let’s start off with a simple Ruby program. We’ll write a method that returns a string, adding to that string a person’s name. We’ll then invoke that method a couple of times.

```ruby
def sayGoodnight(name)
  result = "Goodnight, " + name
  return result
end
```

```ruby
# Time for bed...
puts sayGoodnight("John-Boy")
puts sayGoodnight("Mary-Ellen")
```

First, some general observations. Ruby syntax is clean. You don’t need semicolons at the ends of statements as long as you put each statement on a separate line. Ruby comments start with a `#` character and run to the end of the line. Code layout is pretty much up to you; indentation is not significant.

Methods are defined with the keyword `def`, followed by the method name (in this case, `"sayGoodnight"`) and the method’s parameters between parentheses. Ruby doesn’t use braces to delimit the bodies of compound statements and definitions. Instead, you simply finish the body with the keyword `end`. Our method’s body is pretty simple. The first line concatenates the literal string "Goodnight, " to the parameter name and assigns the result to the local variable result. The next line returns that result to the caller. Note that we didn’t have to declare the variable result; it sprang into existence when we assigned to it.

Having defined the method, we call it twice. In both cases we pass the result to the method puts, which simply outputs its argument followed by a newline.

```
Goodnight, John-Boy
Goodnight, Mary-Ellen
```

The line `"puts sayGoodnight("John-Boy")"` contains two method calls, one to `sayGoodnight` and the other to `puts`. Why does one call have its arguments in parentheses while the other doesn’t? In this case it’s purely a matter of taste. The following lines are all equivalent.

```ruby
puts sayGoodnight("John-Boy")
puts sayGoodnight("John-Boy")
puts(sayGoodnight("John-Boy"))
```

However, life isn’t always that simple, and precedence rules can make it difficult to know which argument goes with which method invocation, so we recommend using parentheses in all but the simplest cases.

This example also shows some Ruby string objects. There are many ways to create a string object, but probably the most
common is to use string literals: sequences of characters between single or double quotation marks. The difference between the
two forms is the amount of processing Ruby does on the string while constructing the literal. In the single-quoted case, Ruby
does very little. With a few exceptions, what you type into the string literal becomes the string's value.

In the double-quoted case, Ruby does more work. First, it looks for substitutions—sequences that start with a backslash character—and replaces them with some binary value. The most common of these is `\n`, which is replaced with a newline character. When a string containing a newline is output, the `\n` forces a line break.

puts "And Goodnight,\nGrandma"

produces:
And Goodnight,
Grandma

The second thing that Ruby does with double-quoted strings is expression interpolation. Within the string, the sequence `#{expression}` is replaced by the value of expression. We could use this to rewrite our previous method.

```ruby
def sayGoodnight(name)
  result = "Goodnight, #{name}"
  return result
end
```

When Ruby constructs this string object, it looks at the current value of name and substitutes it into the string. Arbitrarily complex expressions are allowed in the `#{...}` construct. As a shortcut, you don't need to supply the braces when the expression is simply a global, instance, or class variable. For more information on strings, as well as on the other Ruby standard types, see Chapter 5, which begins on page 47.

Finally, we could simplify this method some more. The value returned by a Ruby method is the value of the last expression evaluated, so we can get rid of the return statement altogether.

```ruby
def sayGoodnight(name)
  "Goodnight, #{name}"
end
```

We promised that this section would be brief. We've got just one more topic to cover: Ruby names. For brevity, we'll be using some terms (such as class variable) that we aren't going to define here. However, by talking about the rules now, you'll be ahead of the game when we actually come to discuss instance variables and the like later.

Ruby uses a convention to help it distinguish the usage of a name: the first characters of a name indicate how the name is used. Local variables, method parameters, and method names should all start with a lowercase letter or with an underscore. Global variables are prefixed with a dollar sign ($), while instance variables begin with an `at` sign (@). Class variables start with two `at` signs (@@). Finally, class names, module names, and constants should start with an uppercase letter. Samples of different names are given in Table 2.1 on page 10.

Following this initial character, a name can be any combination of letters, digits, and underscores (with the proviso that the character following an @ sign may not be a digit).

<table>
<thead>
<tr>
<th>Example variable and class names</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variables</strong></td>
</tr>
<tr>
<td><strong>Local</strong></td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>fishAndChips</td>
</tr>
<tr>
<td>x_axis</td>
</tr>
<tr>
<td>thx1138</td>
</tr>
<tr>
<td>_26</td>
</tr>
</tbody>
</table>

**Arrays and Hashes**

Ruby's arrays and hashes are indexed collections. Both store collections of objects, accessible using a key. With arrays, the key
is an integer, whereas hashes support any object as a key. Both arrays and hashes grow as needed to hold new elements. It's more efficient to access array elements, but hashes provide more flexibility. Any particular array or hash can hold objects of differing types; you can have an array containing an integer, a string, and a floating point number, as we'll see in a minute.

You can create and initialize a new array using an array literal---a set of elements between square brackets. Given an array
object, you can access individual elements by supplying an index between square brackets, as the next example shows.

```ruby
a = [ 1, 'cat', 3.14 ]  # array with three elements
# access the first element
```

# set the third element
a[2] = nil
# dump out the array
a » [1, "cat", nil]

You can create empty arrays either by using an array literal with no elements or by using the array object's constructor, `Array.new`.

```ruby
empty1 = []
empty2 = Array.new
```

Sometimes creating arrays of words can be a pain, what with all the quotes and commas. Fortunately, there's a shortcut: `%w` does just what we want.

```ruby
a = %w{ ant bee cat dog elk }
a[0] » "ant"
a[3] » "dog"
```

Ruby hashes are similar to arrays. A hash literal uses braces rather than square brackets. The literal must supply two objects for every entry: one for the key, the other for the value.

For example, you might want to map musical instruments to their orchestral sections. You could do this with a hash.

```ruby
instSection = {
    'cello'     => 'string',
    'clarinet'  => 'woodwind',
    'drum'      => 'percussion',
    'oboe'      => 'woodwind',
    'trumpet'   => 'brass',
    'violin'    => 'string'
}
```

Hashes are indexed using the same square bracket notation as arrays.

```ruby
instSection['oboe'] » "woodwind"
instSection['cello'] » "string"
instSection['bassoon'] » nil
```

As the last example shows, a hash by default returns `nil` when indexed by a key it doesn't contain. Normally this is convenient, as `nil` means `false` when used in conditional expressions. Sometimes you'll want to change this default. For example, if you're using a hash to count the number of times each key occurs, it's convenient to have the default value be zero. This is easily done by specifying a default value when you create a new, empty hash.

```ruby
histogram = Hash.new(0)
histogram['key1'] » 0
histogram['key1'] = histogram['key1'] + 1
histogram['key1'] » 1
```

Array and hash objects have lots of useful methods: see the discussion starting on page 33, and the reference sections starting on pages 278 and 317, for details.

**Control Structures**

Ruby has all the usual control structures, such as `if` statements and `while` loops. Java, C, and Perl programmers may well get caught by the lack of braces around the bodies of these statements. Instead, Ruby uses the keyword `end` to signify the end of a body.

```ruby
if count > 10
    puts "Try again"
elsif tries == 3
    puts "You lose"
else
    puts "Enter a number"
end
```

Similarly, while statements are terminated with `end`.
while weight < 100 and numPallets <= 30
  pallet = nextPallet()
  weight += pallet.weight
  numPallets += 1
end

Ruby statement modifiers are a useful shortcut if the body of an if or while statement is just a single expression. Simply write the expression, followed by if or while and the condition. For example, here's a simple if statement.

if radiation > 3000
  puts "Danger, Will Robinson"
end

Here it is again, rewritten using a statement modifier.

puts "Danger, Will Robinson" if radiation > 3000

Similarly, a while loop such as

while square < 1000
  square = square*square
end

becomes the more concise

square = square*square  while square < 1000

These statement modifiers should seem familiar to Perl programmers.

Regular Expressions

Most of Ruby's built-in types will be familiar to all programmers. A majority of languages have strings, integers, floats, arrays, and so on. However, until Ruby came along, regular expression support was generally built into only the so-called scripting languages, such as Perl, Python, and awk. This is a shame: regular expressions, although cryptic, are a powerful tool for working with text.

Entire books have been written about regular expressions (for example, Mastering Regular Expressions), so we won't try to cover everything in just a short section. Instead, we'll look at just a few examples of regular expressions in action. You'll find full coverage of regular expressions starting on page 56.

A regular expression is simply a way of specifying a pattern of characters to be matched in a string. In Ruby, you typically create a regular expression by writing a pattern between slash characters (/pattern/). And, Ruby being Ruby, regular expressions are of course objects and can be manipulated as such.

For example, you could write a pattern that matches a string containing the text `Perl` or the text `Python` using the following regular expression.

/Perl|Python/

The forward slashes delimit the pattern, which consists of the two things we're matching, separated by a pipe character (`|`). You can use parentheses within patterns, just as you can in arithmetic expressions, so you could also have written this pattern as

/P(eri|thon)/

You can also specify repetition within patterns. /ab+c/ matches a string containing an `a` followed by one or more `b`'s, followed by a `c`. Change the plus to an asterisk, and /ab*c/ creates a regular expression that matches an `a`, zero or more `b`'s, and a `c`.

You can also match one of a group of characters within a pattern. Some common examples are character classes such as `\s`, which matches a whitespace character (space, tab, newline, and so on), `\d`, which matches any digit, and `\w`, which matches any character that may appear in a typical word. The single character `.` (a period) matches any character.

We can put all this together to produce some useful regular expressions.

/\d\d:\d\d:\d\d/     # a time such as 12:34:56
/Perl|Python/      # Perl, zero or more other chars, then Python
/Perl+s+Python/    # Perl, one or more spaces, then Python
/Ruby (Perl|Python)/ # Ruby, a space, and either Perl or Python

Once you have created a pattern, it seems a shame not to use it. The match operator `=~` can be used to match a string against a regular expression. If the pattern is found in the string, =~ returns its starting position, otherwise it returns nil. This
means you can use regular expressions as the condition in if and while statements. For example, the following code fragment
writes a message if a string contains the text 'Perl' or 'Python'.

```ruby
if line =~ /Perl|Python/
  puts "Scripting language mentioned: #{line}"
end
```

The part of a string matched by a regular expression can also be replaced with different text using one of Ruby's substitution
methods.

```ruby
line.sub(/Perl/, 'Ruby')    # replace first 'Perl' with 'Ruby'
line.gsub(/Python/, 'Ruby') # replace every 'Python' with 'Ruby'
```

We'll have a lot more to say about regular expressions as we go through the book.

## Blocks and Iterators

This section briefly describes one of Ruby's particular strengths. We're about to look at code blocks: chunks of code that you
can associate with method invocations, almost as if they were parameters. This is an incredibly powerful feature. You can use
code blocks to implement callbacks (but they're simpler than Java's anonymous inner classes), to pass around chunks of code
(but they're more flexible than C's function pointers), and to implement iterators.

Code blocks are just chunks of code between braces or do...end.

```ruby
{ puts "Hello" }       # this is a block
do                     # and so is this
    club.enroll(person)  # and so is this
    person.socialize     #
end                    #
```

Once you've created a block, you can associate it with a call to a method. That method can then invoke the block one or more
times using the Ruby yield statement. The following example shows this in action. We define a method that calls yield twice.
We then call it, putting a block on the same line, after the call (and after any arguments to the method).

```ruby
[Some people like to think of the association of a block with a method as a kind of parameter passing. This works on one level, but it isn't really the whole story. You might be better off thinking of the block and the method as coroutines, which transfer control back and forth between themselves.]
```

```ruby
def callBlock
  yield
  yield
end
callBlock { puts "In the block" }
```

produces:

```
In the block
In the block
```

See how the code in the block (puts "In the block") is executed twice, once for each call to yield.

You can provide parameters to the call to yield: these will be passed to the block. Within the block, you list the names of the
arguments to receive these parameters between vertical bars (``|'`).

```ruby
def callBlock
  yield ,
end
callBlock { |, | ... }
```

Code blocks are used throughout the Ruby library to implement iterators: methods that return successive elements from some
kind of collection, such as an array.

```ruby
a = %w( ant bee cat dog elk )    # create an array
a.each { |animal| puts animal }  # iterate over the contents
```

produces:

```
ant
bee
```
Let's look at how we might implement the Array class's each iterator that we used in the previous example. The each iterator loops through every element in the array, calling yield for each one. In pseudo code, this might look like:

```ruby
# within class Array...
def each
  for each element
    yield(element)
  end
end
```

You could then iterate over an array's elements by calling its each method and supplying a block. This block would be called for each element in turn.

```
['cat', 'dog', 'horse'].each do |animal|
  print animal, " -- "
end
```

produces:

```
cat -- dog -- horse --
```

Similarly, many looping constructs that are built into languages such as C and Java are simply method calls in Ruby, with the methods invoking the associated block zero or more times.

```
5.times { print "*" }
3.upto(6) { |i| print i }
('a'..'e').each { |char| print char }
```

produces:

```
*****3456abcde
```

Here we ask the number 5 to call a block five times, then ask the number 3 to call a block, passing in successive values until it reaches 6. Finally, the range of characters from `a` to `e` invokes a block using the method each.

## Reading and 'Riting

Ruby comes with a comprehensive I/O library. However, in most of the examples in this book we'll stick to a few simple methods. We've already come across two methods that do output. puts writes each of its arguments, adding a newline after each. print also writes its arguments, but with no newline. Both can be used to write to any I/O object, but by default they write to the console.

Another output method we use a lot is printf, which prints its arguments under the control of a format string (just like printf in C or Perl).

```
printf "Number: %5.2f, String: %s", 1.23, "hello"
```

produces:

```
Number:  1.23, String: hello
```

In this example, the format string "Number: %5.2f, String: %s" tells printf to substitute in a floating point number (allowing five characters in total, with two after the decimal point) and a string.

There are many ways to read input into your program. Probably the most traditional is to use the routine gets, which returns the next line from your program's standard input stream.

```
line = gets
print line
```

The gets routine has a side effect: as well as returning the line just read, it also stores it into the global variable `$_`. This variable is special, in that it is used as the default argument in many circumstances. If you call print with no argument, it prints the contents of `$_`. If you write an if or while statement with just a regular expression as the condition, that expression is matched against `$_`. While viewed by some purists as a rebarbative barbarism, these abbreviations can help you write some concise programs. For example, the following program prints all lines in the input stream that contain the word "Ruby."

```
while gets         # assigns line to $_
  if /Ruby/       # matches against $_
    print $._     # prints $._
  end
end
```

The "Ruby way" to write this would be to use an iterator.
ARGF.each { |line| print line if line =~ /Ruby/ }

This uses the predefined object ARGF, which represents the input stream that can be read by a program.

**Onward and Upward**

That's it. We've finished our lightning-fast tour of some of the basic features of Ruby. We've had a brief look at objects, methods, strings, containers, and regular expressions, seen some simple control structures, and looked at some rather nifty iterators. Hopefully, this chapter has given you enough ammunition to be able to attack the rest of this book.

Time to move on, and up---up to a higher level. Next, we'll be looking at classes and objects, things that are at the same time both the highest-level constructs in Ruby and the essential underpinnings of the entire language.
Classes, Objects, and Variables

From the examples we've shown so far, you might be wondering about our earlier assertion that Ruby is an object-oriented language. Well, this chapter is where we justify that claim. We're going to be looking at how you create classes and objects in Ruby, and at some of the ways in which Ruby is more powerful than most object-oriented languages. Along the way, we'll be implementing part of our next billion-dollar product, the Internet Enabled Jazz and Blue Grass jukebox.

After months of work, our highly paid Research and Development folks have determined that our jukebox needs songs. So it seems like a good idea to start off by setting up a Ruby class that represents things that are songs. We know that a real song has a name, an artist, and a duration, so we'll want to make sure that the song objects in our program do, too.

We'll start off by creating a basic class Song,[As we mentioned on page 9, class names start with an uppercase letter, while method names start with a lowercase letter.] which contains just a single method, initialize.

class Song
  def initialize(name, artist, duration)
    @name = name
    @artist = artist
    @duration = duration
  end
end

initialize is a special method in Ruby programs. When you call Song.new to create a new Song object, Ruby creates an uninitialized object and then calls that object's initialize method, passing in any parameters that were passed to new. This gives you a chance to write code that sets up your object's state.

For class Song, the initialize method takes three parameters. These parameters act just like local variables within the method, so they follow the local variable naming convention of starting with a lowercase letter.

Each object represents its own song, so we need each of our Song objects to carry around its own song name, artist, and duration. This means we need to store these values as instance variables within the object. In Ruby, an instance variable is simply a name preceded by an `@` sign (``@``). In our example, the parameter name is assigned to the instance variable @name, artist is assigned to @artist, and duration (the length of the song in seconds) is assigned to @duration.

Let's test our spiffy new class.

```ruby
aSong = Song.new("Bicylops", "Fleck", 260)
aSonginspect » "#{Song::0x401b4924 @duration=260, @artist="Fleck\", @name="Bicylops\">"
```

Well, it seems to work. By default, the inspect message, which can be sent to any object, dumps out the object's id and instance variables. It looks as though we have them set up correctly.

Our experience tells us that during development we'll be printing out the contents of a Song object many times, and inspect's default formatting leaves something to be desired. Fortunately, Ruby has a standard message, to_s, which it sends to any object it wants to render as a string. Let's try it on our song.

```ruby
aSong = Song.new("Bicylops", "Fleck", 260)
aSong.to_s » "#{Song::0x401b499c}"
```

That wasn't too useful---it just reported the object id. So, let's override to_s in our class. As we do this, we should also take a
moment to talk about how we're showing the class definitions in this book.

In Ruby, classes are never closed: you can always add methods to an existing class. This applies to the classes you write as well as the standard, built-in classes. All you have to do is open up a class definition for an existing class, and the new contents you specify will be added to whatever's there.

This is great for our purposes. As we go through this chapter, adding features to our classes, we'll show just the class definitions for the new methods; the old ones will still be there. It saves us having to repeat redundant stuff in each example. Obviously, though, if you were creating this code from scratch, you'd probably just throw all the methods into a single class definition.

Enough detail! Let's get back to adding a to_s method to our Song class.

```ruby
class Song
  def to_s
    "Song: #{@name}--#{@artist} (#{@duration})"
  end
end
```

```ruby
aSong = Song.new("Bicylops", "Fleck", 260)
aSong.to_s
» "Song: Bicylops--Fleck (260)"
```

Excellent, we're making progress. However, we've slipped in something subtle. We said that Ruby supports to_s for all objects, but we didn't say how. The answer has to do with inheritance, subclassing, and how Ruby determines what method to run when you send a message to an object. This is a subject for a new section, so....

**Inheritance and Messages**

Inheritance allows you to create a class that is a refinement or specialization of another class. For example, our jukebox has the concept of songs, which we encapsulate in class Song. Then marketing comes along and tells us that we need to provide karaoke support. A karaoke song is just like any other (there's no vocal on it, but that doesn't concern us). However, it also has an associated set of lyrics, along with timing information. When our jukebox plays a karaoke song, the lyrics should flow across the screen on the front of the jukebox in time with the music.

An approach to this problem is to define a new class, KaraokeSong, which is just like Song, but with a lyric track.

```ruby
class KaraokeSong < Song
  def initialize(name, artist, duration, lyrics)
    super(name, artist, duration)
    @lyrics = lyrics
  end
end
```

The `"< Song` on the class definition line tells Ruby that a KaraokeSong is a subclass of Song. (Not surprisingly, this means that Song is a superclass of KaraokeSong. People also talk about parent-child relationships, so KaraokeSong's parent would be Song.) For now, don't worry too much about the initialize method; we'll talk about that super call later.

Let's create a KaraokeSong and check that our code worked. (In the final system, the lyrics will be held in an object that includes the text and timing information. To test out our class, though, we'll just use a string. This is another benefit of untyped languages---we don't have to define everything before we start running code.

```ruby
aSong = KaraokeSong.new("My Way", "Sinatra", 225, "And now, the...")
aSong.to_s
» "Song: My Way--Sinatra (225)"
```

Well, it ran, but why doesn't the to_s method show the lyric?

The answer has to do with the way Ruby determines which method should be called when you send a message to an object. When Ruby compiles the method invocation aSong.to_s, it doesn't actually know where to find the method to_s. Instead, it defers the decision until the program is run. At that time, it looks at the class of aSong. If that class implements a method with the same name as the message, that method is run. Otherwise, Ruby looks for a method in the parent class, and then in the grandparent, and so on up the ancestor chain. If it runs out of ancestors without finding the appropriate method, it takes a special action that normally results in an error being raised. *[In fact, you can intercept this error, which allows you to fake out methods at runtime. This is described under Object#method_missing on page 355.]*
So, back to our example. We sent the message `to_s` to `aSong`, an object of class `KaraokeSong`. Ruby looks in `KaraokeSong` for a method called `to_s`, but doesn't find it. The interpreter then looks in `KaraokeSong`'s parent, class `Song`, and there it finds the `to_s` method that we defined on page 18. That's why it prints out the song details but not the lyrics---class `Song` doesn't know anything about lyrics.

Let's fix this by implementing `KaraokeSong#to_s`. There are a number of ways to do this. Let's start with a bad way. We'll copy the `to_s` method from `Song` and add on the lyric.

```ruby
class KaraokeSong
  # ...
  def to_s
    "KS: #{@name}--#{@artist} (#{@duration}) [#{@lyrics}]"
  end
end

aSong = KaraokeSong.new("My Way", "Sinatra", 225, "And now, the...")
aSong.to_s  # "KS: My Way--Sinatra (225) [And now, the...]"
```

We're correctly displaying the value of the `@lyrics` instance variable. To do this, the subclass directly accesses the instance variables of its ancestors. So why is this a bad way to implement `to_s`?

The answer has to do with good programming style (and something called **decoupling**). By poking around in our parent's internal state, we're tying ourselves tightly to its implementation. Say we decided to change `Song` to store the duration in milliseconds. Suddenly, `KaraokeSong` would start reporting ridiculous values. The idea of a karaoke version of "My Way" that lasts for 3750 minutes is just too frightening to consider.

We get around this problem by having each class handle its own internal state. When `KaraokeSong#to_s` is called, we'll have it call its parent's `to_s` method to get the song details. It will then append to this the lyric information and return the result. The trick here is the Ruby keyword `super`. When you invoke `super` with no arguments, Ruby sends a message to the current object's parent, asking it to invoke a method of the same name as the current method, and passing it the parameters that were passed to the current method. Now we can implement our new and improved `to_s`.

```ruby
class KaraokeSong < Song
  # Format ourselves as a string by appending
  # our lyrics to our parent's #to_s value.
  def to_s
    super + " [#{@lyrics}]"
  end
end

aSong = KaraokeSong.new("My Way", "Sinatra", 225, "And now, the...")
aSong.to_s  # "Song: My Way--Sinatra (225) [And now, the...]"
```

We explicitly told Ruby that `KaraokeSong` was a subclass of `Song`, but we didn't specify a parent class for `Song` itself. If you don't specify a parent when defining a class, Ruby supplies class `Object` as a default. This means that all objects have `Object` as an ancestor, and that `Object`'s instance methods are available to every object in Ruby. Back on page 18 we said that `to_s` is available to all objects. Now we know why; `to_s` is one of more than 35 instance methods in class `Object`. The complete list begins on page 351.

### Inheritance and Mixins

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

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

Ruby offers an interesting and powerful compromise, giving you the simplicity of single inheritance and the power of multiple inheritance. A Ruby class can have only one direct parent, and so Ruby is a single-inheritance language. However, Ruby classes can include the functionality of any number of mixins (a mixin is like a partial class definition). This provides a controlled
multiple-inheritance-like capability with none of the drawbacks. We'll explore mixins more beginning on page 98.

So far in this chapter we've been looking at classes and their methods. Now it's time to move on to the objects, such as the instances of class Song.

**Objects and Attributes**

The Song objects we've created so far have an internal state (such as the song title and artist). That state is private to those objects—no other object can access an object's instance variables. In general, this is a Good Thing. It means that the object is solely responsible for maintaining its own consistency.

However, an object that is totally secretive is pretty useless—you can create it, but then you can't do anything with it. You'll normally define methods that let you access and manipulate the state of an object, allowing the outside world to interact with the object. These externally visible facets of an object are called its *attributes*.

For our Song objects, the first thing we may need is the ability to find out the title and artist (so we can display them while the song is playing) and the duration (so we can display some kind of progress bar).

```ruby
class Song
  def name
    @name
  end
  def artist
    @artist
  end
  def duration
    @duration
  end
end
aSong = Song.new("Bicylops", "Fleck", 260)
aSong.artist  »  "Fleck"
aSong.name    »  "Bicylops"
aSong.duration»  260
```

Here we've defined three accessor methods to return the values of the three instance attributes. Because this is such a common idiom, Ruby provides a convenient shortcut: `attr_reader` creates these accessor methods for you.

```ruby
class Song
  attr_reader :name, :artist, :duration
end
aSong = Song.new("Bicylops", "Fleck", 260)
aSong.artist  »  "Fleck"
aSong.name    »  "Bicylops"
aSong.duration»  260
```

This example has introduced something new. The construct `:artist` is an expression that returns a Symbol object corresponding to artist. You can think of `:artist` as meaning the *name* of the variable artist, while plain artist is the *value* of the variable. In this example, we named the accessor methods name, artist, and duration. The corresponding instance variables, `@name`, `@artist`, and `@duration`, will be created automatically. These accessor methods are identical to the ones we wrote by hand earlier.

**Writable Attributes**

Sometimes you need to be able to set an attribute from outside the object. For example, let's assume that the duration that is initially associated with a song is an estimate (perhaps gathered from information on a CD or in the MP3 data). The first time we play the song, we get to find out how long it actually is, and we store this new value back in the Song object.
In languages such as C++ and Java, you’d do this with setter functions.

```ruby
class JavaSong {
  private Duration myDuration;
  public void setDuration(Duration newDuration) {
    myDuration = newDuration;
  }
}
```

`s = new Song(....)`

```ruby
s.setDuration(length)
```

In Ruby, the attributes of an object can be accessed as if they were any other variable. We’ve seen this above with phrases such as `aSong.name`. So, it seems natural to be able to assign to these variables when you want to set the value of an attribute. In keeping with the Principle of Least Surprise, that’s just what you do in Ruby.

```ruby
class Song
  def duration=(newDuration)
    @duration = newDuration
  end
end
```

```ruby
aSong = Song.new("Bicylops", "Fleck", 260)
aSong.duration = 257   # set attribute with updated value
```

The assignment `aSong.duration = 257` invokes the method `duration=` in the `aSong` object, passing it 257 as an argument. In fact, defining a method name ending in an equals sign makes that name eligible to appear on the left-hand side of an assignment.

Again, Ruby provides a shortcut for creating these simple attribute setting methods.

```ruby
class Song
  attr_writer :duration
end
```

```ruby
aSong = Song.new("Bicylops", "Fleck", 260)
aSong.duration = 257
```

### Virtual Attributes

These attribute accessing methods do not have to be just simple wrappers around an object’s instance variables. For example, you might want to access the duration in minutes and fractions of a minute, rather than in seconds as we’ve been doing.

```ruby
class Song
  def durationInMinutes
    @duration/60.0   # force floating point
  end
  def durationInMinutes=(value)
    @duration = (value*60).to_i
  end
end
```

```ruby
aSong = Song.new("Bicylops", "Fleck", 260)
aSong.durationInMinutes = 4.2
```

Here we’ve used attribute methods to create a virtual instance variable. To the outside world, `durationInMinutes` seems to be an attribute like any other. Internally, though, there is no corresponding instance variable.

This is more than a curiosity. In his landmark book *Object-Oriented Software Construction*, Bertrand Meyer calls this the
Uniform Access Principle. By hiding the difference between instance variables and calculated values, you are shielding the rest of the world from the implementation of your class. You're free to change how things work in the future without impacting the millions of lines of code that use your class. This is a big win.

Class Variables and Class Methods

So far, all the classes we've created have contained instance variables and instance methods: variables that are associated with a particular instance of the class, and methods that work on those variables. Sometimes classes themselves need to have their own states. This is where class variables come in.

Class Variables

A class variable is shared among all objects of a class, and it is also accessible to the class methods that we'll describe later. There is only one copy of a particular class variable for a given class. Class variable names start with two `@` signs, such as `@@count`. Unlike global and instance variables, class variables must be initialized before they are used. Often this initialization is just a simple assignment in the body of the class definition.

For example, our jukebox may want to record how many times each particular song has been played. This count would probably be an instance variable of the Song object. When a song is played, the value in the instance is incremented. But say we also want to know how many songs have been played in total. We could do this by searching for all the Song objects and adding up their counts, or we could risk excommunication from the Church of Good Design and use a global variable. Instead, we'll use a class variable.

```ruby
class Song
  @@plays = 0
  def initialize(name, artist, duration)
    @name  = name
    @artist = artist
    @duration = duration
    @plays = 0
  end
  def play
    @plays += 1
    @@plays += 1
    "This song: #{@plays} plays. Total @@plays plays."
  end
end
```

For debugging purposes, we've arranged for Song#play to return a string containing the number of times this song has been played, along with the total number of plays for all songs. We can test this easily.

```ruby
s1 = Song.new("Song1", "Artist1", 234)  # test songs..
s2 = Song.new("Song2", "Artist2", 345)
s1.play  # "This song: 1 plays. Total 1 plays."
s2.play  # "This song: 1 plays. Total 2 plays."
s1.play  # "This song: 2 plays. Total 3 plays."
s1.play  # "This song: 3 plays. Total 4 plays."
```

Class variables are private to a class and its instances. If you want to make them accessible to the outside world, you'll need to write an accessor method. This method could be either an instance method or, leading us neatly to the next section, a class method.

Class Methods

Sometimes a class needs to provide methods that work without being tied to any particular object.

We've already come across one such method. The new method creates a new Song object but is not itself associated with a particular song.

```ruby
aSong = Song.new(....)
```

You'll find class methods sprinkled throughout the Ruby libraries. For example, objects of class File represent open files in the underlying file system. However, class File also provides several class methods for manipulating files that aren't open and
therefore don't have a File object. If you want to delete a file, you call the class method `File.delete`, passing in the name.

```
File.delete("doomedFile")
```

Class methods are distinguished from instance methods by their definition. Class methods are defined by placing the class name and a period in front of the method name.

```ruby
class Example
  def instMeth             # instance method
    end

  def Example.classMeth    # class method
    end
end
```

Jukeboxes charge money for each song played, not by the minute. That makes short songs more profitable than long ones. We may want to prevent songs that take too long from being available on the SongList. We could define a class method in `SongList` that checked to see if a particular song exceeded the limit. We'll set this limit using a class constant, which is simply a constant (remember constants? they start with an uppercase letter) that is initialized in the class body.

```ruby
class SongList
  MaxTime = 5*60           # 5 minutes
  def SongList.isTooLong(aSong)
    return aSong.duration > MaxTime
  end
end
```

```ruby
song1 = Song.new("Bicylops", "Fleck", 260)  
SongList.isTooLong(song1) » false
song2 = Song.new("The Calling", "Santana", 468)  
SongList.isTooLong(song2) » true
```

### Singletons and Other Constructors

Sometimes you want to override the default way in which Ruby creates objects. As an example, let's look at our jukebox. Because we'll have many jukeboxes, spread all over the country, we want to make maintenance as easy as possible. Part of the requirement is to log everything that happens to a jukebox: the songs that are played, the money received, the strange fluids poured into it, and so on. Because we want to reserve the network bandwidth for music, we'll store these logfiles locally. This means we'll need a class that handles logging. However, we want only one logging object per jukebox, and we want that object to be shared among all the other objects that use it.

Enter the Singleton pattern, documented in *Design Patterns*. We'll arrange things so that the only way to create a logging object is to call `Logger.create`, and we'll ensure that only one logging object is ever created.

```ruby
class Logger
  private_class_method :new
  @@logger = nil
  def Logger.create
    @@logger = new unless @@logger
    @@logger
  end
end
```

By making `Logger`'s method `new` private, we prevent anyone from creating a logging object using the conventional constructor. Instead, we provide a class method, `Logger.create`. This uses the class variable `@@logger` to keep a reference to a single instance of the logger, returning that instance every time it is called. *The implementation of singletons that we present here is not thread-safe; if multiple threads were running, it would be possible to create multiple logger objects. Rather than add thread safety ourselves, however, we'd probably use the Singleton mixin supplied with Ruby, which is documented on page...*
We can check this by looking at the object identifiers the method returns.

<table>
<thead>
<tr>
<th>Method</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logger.create.id</td>
<td>» 537766930</td>
</tr>
<tr>
<td>Logger.create.id</td>
<td>» 537766930</td>
</tr>
</tbody>
</table>

Using class methods as pseudo-constructors can also make life easier for users of your class. As a trivial example, let's look at a class Shape that represents a regular polygon. Instances of Shape are created by giving the constructor the required number of sides and the total perimeter.

```ruby
class Shape
  def initialize(numSides, perimeter)
    # ...
  end
end
```

However, a couple of years later, this class is used in a different application, where the programmers are used to creating shapes by name, and by specifying the length of the side, not the perimeter. Simply add some class methods to Shape.

```ruby
class Shape
  def Shape.triangle(sideLength)
    Shape.new(3, sideLength*3)
  end
  def Shape.square(sideLength)
    Shape.new(4, sideLength*4)
  end
end
```

There are many interesting and powerful uses of class methods, but exploring them won't get our jukebox finished any sooner, so let's move on.

### Access Control

When designing a class interface, it's important to consider just how much access to your class you'll be exposing to the outside world. Allow too much access into your class, and you risk increasing the coupling in your application. Users of your class will be tempted to rely on details of your class's implementation, rather than on its logical interface. The good news is that the only way to change an object's state in Ruby is by calling one of its methods. Control access to the methods and you've controlled access to the object. A good rule of thumb is never to expose methods that could leave an object in an invalid state.

Ruby gives us three levels of protection.

- **Public methods** can be called by anyone---there is no access control. Methods are public by default (except for `initialize`, which is always private).
- **Protected methods** can be invoked only by objects of the defining class and its subclasses. Access is kept within the family.
- **Private methods** cannot be called with an explicit receiver. Because you cannot specify an object when using them, private methods can be called only in the defining class and by direct descendents within that same object.

The difference between "protected" and "private" is fairly subtle, and is different in Ruby than in most common OO languages. If a method is protected, it may be called by any instance of the defining class or its subclasses. If a method is private, it may be called only within the context of the calling object---it is never possible to access another object's private methods directly, even if the object is of the same class as the caller.

Ruby differs from other OO languages in another important way. Access control is determined dynamically, as the program runs, not statically. You will get an access violation only when the code attempts to execute the restricted method.

### Specifying Access Control

You specify access levels to methods within class or module definitions using one or more of the three functions `public`, `protected`, and `private`. Each function can be used in two different ways.

If used with no arguments, the three functions set the default access control of subsequently defined methods. This is probably familiar behavior if you're a C++ or Java programmer, where you'd use keywords such as `public` to achieve the same effect.

```ruby
class MyClass
  def method1    # default is 'public'
end
```
protected       # subsequent methods will be 'protected'

    def method2   # will be 'protected'
        # ...        
    end

private        # subsequent methods will be 'private'

    def method3   # will be 'private'
        # ...        
    end

public         # subsequent methods will be 'public'

    def method4   # and this will be 'public'
        # ...        
    end

Alternatively, you can set access levels of named methods by listing them as arguments to the access control functions.

class MyClass

    def method1
        end

    # ... and so on

    public :method1, :method4
    protected :method2
    private  :method3
end

A class's initialize method is automatically declared to be private.

It's time for some examples. Perhaps we're modeling an accounting system where every debit has a corresponding credit. Because we want to ensure that no one can break this rule, we'll make the methods that do the debits and credits private, and we'll define our external interface in terms of transactions.

class Accounts

private

    def debit(account, amount)
        account.balance -= amount
    end
    def credit(account, amount)
        account.balance += amount
    end

public

    #...
    def transferToSavings(amount)
debit(@checking, amount)
credit(@savings, amount)
end
#
end

Protected access is used when objects need to access the internal state of other objects of the same class. For example, we may want to allow the individual Account objects to compare their raw balances, but may want to hide those balances from the rest of the world (perhaps because we present them in a different form).

class Account
  attr_reader :balance       # accessor method 'balance'
  protected :balance         # and make it protected
  def greaterBalanceThan(other)
    return @balance > other.balance
  end
end

Because the attribute balance is protected, it's available only within Account objects.

Variables

Now that we've gone to the trouble to create all these objects, let's make sure we don't lose them. Variables are used to keep track of objects; each variable holds a reference to an object.

Let's confirm this with some code.

```ruby
person = "Tim"
person.id  » 537771100
person.type » String
person    » "Tim"
```

On the first line, Ruby creates a new String object with the value `"Tim."` A reference to this object is placed in the local variable person. A quick check shows that the variable has indeed taken on the personality of a string, with an object id, a type, and a value.

So, is a variable an object?

In Ruby, the answer is `"no."` A variable is simply a reference to an object. Objects float around in a big pool somewhere (the heap, most of the time) and are pointed to by variables.

Let's make the example slightly more complicated.

```ruby
person1 = "Tim"
person2 = person1
person1[0] = 'J'
person1    » "Jim"
person2    » "Jim"
```

What happened here? We changed the first character of person1, but both person1 and person2 changed from `"Tim"` to `"Jim."`

It all comes back to the fact that variables hold references to objects, not the objects themselves. The assignment of person1 to person2 doesn't create any new objects; it simply copies person1's object reference to person2, so that both person1 and
person2 refer to the same object. We show this in Figure 3.1 on page 31.

Assignment aliases objects, potentially giving you multiple variables that reference the same object. But can’t this cause problems in your code? It can, but not as often as you’d think (objects in Java, for example, work exactly the same way). For instance, in the example in Figure 3.1, you could avoid aliasing by using the dup method of String, which creates a new String object with identical contents.

```ruby
person1 = "Tim"
person2 = person1.dup
person1[0] = "J"
person1  »  "Jim"
person2  »  "Tim"
```

You can also prevent anyone from changing a particular object by freezing it (we talk more about freezing objects on page 251). Attempt to alter a frozen object, and Ruby will raise a TypeError exception.

```ruby
person1 = "Tim"
person2 = person1
person1.freeze  # prevent modifications to the object
person2[0] = "J"
```

```ruby
produces:
prog.rb:4:in `=': can't modify frozen string (TypeError)
from prog.rb:4
```
Containers, Blocks, and Iterators

A jukebox with one song is unlikely to be popular (except perhaps in some very, very scary bars), so pretty soon we'll have to start thinking about producing a catalog of available songs and a playlist of songs waiting to be played. Both of these are containers: objects that hold references to one or more other objects.

Both the catalog and the playlist need a similar set of methods: add a song, remove a song, return a list of songs, and so on. The playlist may perform additional tasks, such as inserting advertising every so often or keeping track of cumulative play time, but we'll worry about these things later. In the meantime, it seems like a good idea to develop some kind of generic SongList class, which we can specialize into catalogs and playlists.

Containers

Before we start implementing, we'll need to work out how to store the list of songs inside a SongList object. We have three obvious choices. We could use the Ruby Array type, use the Ruby Hash type, or create our own list structure. Being lazy, for now we'll look at arrays and hashes, and choose one of these for our class.

Arrays

The class Array holds a collection of object references. Each object reference occupies a position in the array, identified by a non-negative integer index.

You can create arrays using literals or by explicitly creating an Array object. A literal array is simply a list of objects between square brackets.

```ruby
a = [ 3.14159, "pie", 99 ]
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.type</td>
<td>Array</td>
</tr>
<tr>
<td>a.length</td>
<td>3</td>
</tr>
<tr>
<td>a[0]</td>
<td>3.14159</td>
</tr>
<tr>
<td>a[1]</td>
<td>&quot;pie&quot;</td>
</tr>
<tr>
<td>a[2]</td>
<td>99</td>
</tr>
<tr>
<td>a[3]</td>
<td>nil</td>
</tr>
</tbody>
</table>

```ruby
b = Array.new
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.type</td>
<td>Array</td>
</tr>
<tr>
<td>b.length</td>
<td>0</td>
</tr>
</tbody>
</table>

```ruby
b[0] = "second"
b[1] = "array"
```

```ruby
b                  | ["second", "array"]
```

Arrays are indexed using the [] operator. As with most Ruby operators, this is actually a method (in class Array) and hence can be overridden in subclasses. As the example shows, array indices start at zero. Index an array with a single integer, and it returns the object at that position or returns nil if nothing's there. Index an array with a negative integer, and it counts from the end. This is shown in Figure 4.1 on page 35.
a = [ 1, 3, 5, 7, 9 ]
a[-1] » 9
a[-2] » 7
a[-99] » nil

You can also index arrays with a pair of numbers, [start, count]. This returns a new array consisting of references to count objects starting at position start.

a = [ 1, 3, 5, 7, 9 ]
a[1, 3] » [3, 5, 7]
a[3, 1] » [7]
a[-3, 2] » [5, 7]

Finally, you can index arrays using ranges, in which start and end positions are separated by two or three periods. The two-period form includes the end position, while the three-period form does not.

a = [ 1, 3, 5, 7, 9 ]
a[1..3] » [3, 5, 7]
a[1...3] » [3, 5]
a[-3..-1] » [5, 7, 9]

The [] operator has a corresponding []= operator, which lets you set elements in the array. If used with a single integer index, the element at that position is replaced by whatever is on the right-hand side of the assignment. Any gaps that result will be filled with nil.

a = [ 1, 3, 5, 7, 9 ]
a[1] = 'bat' » [1, "bat", 5, 7, 9]
a[-3] = 'cat' » [1, "bat", "cat", 7, 9]
a[3] = [ 9, 8 ] » [1, "bat", "cat", [9, 8], 9]
a[6] = 99 » [1, "bat", "cat", [9, 8], 9, nil, 99]

If the index to []= is two numbers (a start and a length) or a range, then those elements in the original array are replaced by whatever is on the right-hand side of the assignment. If the length is zero, the right-hand side is inserted into the array before the start position; no elements are removed. If the right-hand side is itself an array, its elements are used in the replacement. The array size is automatically adjusted if the index selects a different number of elements than are available on the right-hand side of the assignment.

a = [ 1, 3, 5, 7, 9 ]
a[2, 2] = 'cat' » [1, 3, "cat", 9]
a[2, 0] = 'dog' » [1, 3, "dog", "cat", 9]
a[1, 1] = [ 9, 8, 7 ] » [1, 9, 8, 7, "dog", "cat", 9]
a[0..3] = [] » ["dog", "cat", 9]

Arrays have a large number of other useful methods. Using these, you can treat arrays as stacks, sets, queues, dequeues, and fifos. A complete list of array methods starts on page 278.

Hashes

Hashes (sometimes known as associative arrays or dictionaries) are similar to arrays, in that they are indexed collectives of object references.

However, while you index arrays with integers, you can index a hash with objects of any type: strings, regular expressions, and so on. When you store a value in a hash, you actually supply two objects—the key and the value. You can subsequently
retrieve the value by indexing the hash with the same key. The values in a hash can be any objects of any type. The example that follows uses hash literals: a list of key => value pairs between braces.

```ruby
h = { 'dog' => 'canine', 'cat' => 'feline', 'donkey' => 'asinine' }

h.length  » 3
h['dog']  » "canine"
hs['cow'] = 'bovine'
h[12] = 'dodecine'
h['cat'] = 99
h  » {'cow' => "bovine", "cat" => 99, 12 => "dodecine", "donkey" => "asinine", "dog" => "canine"}
```

Compared with arrays, hashes have one significant advantage: they can use any object as an index. However, they also have a significant disadvantage: their elements are not ordered, so you cannot easily use a hash as a stack or a queue.

You'll find that hashes are one of the most commonly used data structures in Ruby. A full list of the methods implemented by class Hash starts on page 317.

### Implementing a SongList Container

After that little diversion into arrays and hashes, we're now ready to implement the jukebox's SongList. Let's invent a basic list of methods we need in our SongList. We'll want to add to it as we go along, but it will do for now.

- **append( aSong ) » list**
  Append the given song to the list.
- **deleteFirst() » aSong**
  Remove the first song from the list, returning that song.
- **deleteLast() » aSong**
  Remove the last song from the list, returning that song.
- **[ anIndex } » aSong**
  Return the song identified by anIndex, which may be an integer index or a song title.

This list gives us a clue to the implementation. The ability to append songs at the end, and remove them from both the front and end, suggests a deque---a double-ended queue---which we know we can implement using an Array. Similarly, the ability to return a song at an integer position in the list is supported by arrays.

However, there's also the need to be able to retrieve songs by title, which might suggest using a hash, with the title as a key and the song as a value. Could we use a hash? Well, possibly, but there are problems. First a hash is unordered, so we'd probably need to use an ancillary array to keep track of the list. A bigger problem is that a hash does not support multiple keys with the same value. That would be a problem for our playlist, where the same song might be queued up for playing multiple times. So, for now we'll stick with an array of songs, searching it for titles when needed. If this becomes a performance bottleneck, we can always add some kind of hash-based lookup later.

We'll start our class with a basic initialize method, which creates the Array we'll use to hold the songs and stores a reference to it in the instance variable @songs.

```ruby
class SongList
  def initialize
    @songs = Array.new
  end
end
```

The SongList#append method adds the given song to the end of the @songs array. It also returns self, a reference to the current SongList object. This is a useful convention, as it lets us chain together multiple calls to append. We'll see an example of this later.

```ruby
class SongList
  def append(aSong)
    @songs.push(aSong)
    self
  end
end
```

Then we'll add the deleteFirst and deleteLast methods, trivially implemented using Array#shift and Array#pop, respectively.
class SongList
  def deleteFirst
    @songs.shift
  end
  def deleteLast
    @songs.pop
  end
end

At this point, a quick test might be in order. First, we'll append four songs to the list. Just to show off, we'll use the fact that `append` returns the `SongList` object to chain together these method calls.

```ruby
list = SongList.new
list.append(Song.new('title1', 'artist1', 1)).
  .append(Song.new('title2', 'artist2', 2)).
  .append(Song.new('title3', 'artist3', 3)).
  .append(Song.new('title4', 'artist4', 4))
```

Then we'll check that songs are taken from the start and end of the list correctly, and that `nil` is returned when the list becomes empty.

```ruby
list.deleteFirst  » Song: title1--artist1 (1)
list.deleteFirst  » Song: title2--artist2 (2)
list.deleteLast   » Song: title4--artist4 (4)
list.deleteLast   » Song: title3--artist3 (3)
list.deleteLast   » nil
```

So far so good. Our next method is `[]`, which accesses elements by index. If the index is a number (which we check using `Object#kind_of?`), we just return the element at that position.

```ruby
class SongList
def [](key)
  if key.kind_of?(Integer)
    @songs[key]
  else
    # ...
  end
end
end
```

Again, testing this is pretty trivial.

```ruby
list[0]  » Song: title1--artist1 (1)
list[2]  » Song: title3--artist3 (3)
list[9]  » nil
```

Now we need to add the facility that lets us look up a song by title. This is going to involve scanning through the songs in the list, checking the title of each. To do this, we first need to spend a couple of pages looking at one of Ruby's neatest features: iterators.

## Blocks and Iterators

So, our next problem with `SongList` is to implement the code in method `[]` that takes a string and searches for a song with that title. This seems straightforward: we have an array of songs, so we just go through it one element at a time, looking for a match.

```ruby
class SongList
def [](key)
  if key.kind_of?(Integer)
    return @songs[key]
  else
    for i in 0...@songs.length
```
return @songs[i] if key == @songs[i].name
end
return nil
end

This works, and it looks comfortingly familiar: a for loop iterating over an array. What could be more natural?

It turns out there is something more natural. In a way, our for loop is somewhat too intimate with the array; it asks for a length, then retrieves values in turn until it finds a match. Why not just ask the array to apply a test to each of its members? That's just what the find method in Array does.

class SongList
def [](key)
  if key.kind_of?(Integer)
    result = @songs[key]
  else
    result = @songs.find { |aSong| key == aSong.name }
  end
  return result
end
end

We could use if as a statement modifier to shorten the code even more.

class SongList
def [](key)
  return @songs[key] if key.kind_of?(Integer)
  return @songs.find { |aSong| aSong.name == key }
end
end

The method find is an iterator—a method that invokes a block of code repeatedly. Iterators and code blocks are among the more interesting features of Ruby, so let's spend a while looking into them (and in the process we'll find out exactly what that line of code in our [] method actually does).

**Implementing Iterators**

A Ruby iterator is simply a method that can invoke a block of code. At first sight, a block in Ruby looks just like a block in C, Java, or Perl. Unfortunately, in this case looks are deceiving—a Ruby block is a way of grouping statements, but not in the conventional way.

First, a block may appear only in the source adjacent to a method call; the block is written starting on the same line as the method's last parameter. Second, the code in the block is not executed at the time it is encountered. Instead, Ruby remembers the context in which the block appears (the local variables, the current object, and so on), and then enters the method. This is where the magic starts.

Within the method, the block may be invoked, almost as if it were a method itself, using the yield statement. Whenever a yield is executed, it invokes the code in the block. When the block exits, control picks back up immediately after the yield. [Programming-language buffs will be pleased to know that the keyword yield was chosen to echo the yield function in Liskov's language CLU, a language that is over 20 years old and yet contains features that still haven't been widely exploited by the CLU-less.] Let's start with a trivial example.

```ruby
def threeTimes
  yield
  yield
  yield
end
threeTimes { puts "Hello" }
```

produces:
Hello
Hello
Hello

The block (the code between the braces) is associated with the call to the method threeTimes. Within this method, yield is called three times in a row. Each time, it invokes the code in the block, and a cheery greeting is printed. What makes blocks
interesting, however, is that you can pass parameters to them and receive values back from them. For example, we could write a simple function that returns members of the Fibonacci series up to a certain value.\[The basic Fibonacci series is a sequence of integers, starting with two 1's, in which each subsequent term is the sum of the two preceding terms. The series is sometimes used in sorting algorithms and in analyzing natural phenomena.\]

def fibUpTo(max)
  i1, i2 = 1, 1 # parallel assignment
  while i1 <= max
    yield i1
    i1, i2 = i2, i1+i2
  end
end

fibUpTo(1000) { |f| print f, " " }

produces:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

In this example, the yield statement has a parameter. This value is passed to the associated block. In the definition of the block, the argument list appears between vertical bars. In this instance, the variable f receives the value passed to the yield, so the block prints successive members of the series. (This example also shows parallel assignment in action. We'll come back to this on page 75.) Although it is common to pass just one value to a block, this is not a requirement; a block may have any number of arguments. What happens if a block has a different number of parameters than are given to the yield? By a staggering coincidence, the rules we discuss under parallel assignment come into play (with a slight twist: multiple parameters passed to a yield are converted to an array if the block has just one argument).

Parameters to a block may be existing local variables; if so, the new value of the variable will be retained after the block completes. This may lead to unexpected behavior, but there is also a performance gain to be had by using variables that already exist.\[For more information on this and other "gotchas," see the list beginning on page 127; more performance information begins on page 128.\]

A block may also return a value to the method. The value of the last expression evaluated in the block is passed back to the method as the value of the yield. This is how the find method used by class Array works.\[The find method is actually defined in module Enumerable, which is mixed into class Array.\] Its implementation would look something like the following.

class Array
  def find
    for i in 0...size
      value = self[i]
      return value if yield(value)
    end
    return nil
  end
end

[1, 3, 5, 7, 9].find { |v| v*v > 30 } # 7

This passes successive elements of the array to the associated block. If the block returns true, the method returns the corresponding element. If no element matches, the method returns nil. The example shows the benefit of this approach to iterators. The Array class does what it does best, accessing array elements, leaving the application code to concentrate on its particular requirement (in this case, finding an entry that meets some mathematical criteria).

Some iterators are common to many types of Ruby collections. We've looked at find already. Two others are each and collect. each is probably the simplest iterator---all it does is yield successive elements of its collection.

[ 1, 3, 5 ].each { |i| puts i }

produces:
1
3
5

The each iterator has a special place in Ruby; on page 85 we'll describe how it's used as the basis of the language's for loop,
and starting on page 102 we'll see how defining an each method can add a whole lot more functionality to your class for free.

Another common iterator is collect, which takes each element from the collection and passes it to the block. The results returned by the block are used to construct a new array. For instance:

```ruby
"H", "A", "L".collect { |x| x.succ }  »  ["I", "B", "M"]
```

### Ruby Compared with C++ and Java

It's worth spending a paragraph comparing Ruby's approach to iterators to that of C++ and Java. In the Ruby approach, the iterator is simply a method, identical to any other, that happens to call yield whenever it generates a new value. The thing that uses the iterator is simply a block of code associated with this method. There is no need to generate helper classes to carry the iterator state, as in Java and C++. In this, as in many other ways, Ruby is a transparent language. When you write a Ruby program, you concentrate on getting the job done, not on building scaffolding to support the language itself.

Iterators are not limited to accessing existing data in arrays and hashes. As we saw in the Fibonacci example, an iterator can return derived values. This capability is used by the Ruby input/output classes, which implement an iterator interface returning successive lines (or bytes) in an I/O stream.

```ruby
f = File.open("testfile")
f.each do |line|
  print line
end
f.close

produces:
This is line one
This is line two
This is line three
And so on...
```

Let's look at just one more iterator implementation. The Smalltalk language also supports iterators over collections. If you ask Smalltalk programmers to sum the elements in an array, it's likely that they'd use the inject function.

```ruby
sumOfValues
  ^self values
  inject: 0
  into: [:sum :element | sum + element value]
```

inject works like this. The first time the associated block is called, sum is set to inject’s parameter (zero in this case), and element is set to the first element in the array. The second and subsequent times the block is called, sum is set to the value returned by the block on the previous call. This way, sum can be used to keep a running total. The final value of inject is the value returned by the block the last time it was called.

Ruby does not have an inject method, but it’s easy to write one. In this case we'll add it to the Array class, while on page 100 we'll see how to make it more generally available.

```ruby
class Array
  def inject(n)
    each { |value| n = yield(n, value) }
    n
  end
  def sum
    inject(0) { |n, value| n + value }
  end
  def product
    inject(1) { |n, value| n * value }
  end
end

[ 1, 2, 3, 4, 5 ].sum  »  15
[ 1, 2, 3, 4, 5 ].product  »  120
```
Although blocks are often the target of an iterator, they also have other uses. Let’s look at a few.

**Blocks for Transactions**

Blocks can be used to define a chunk of code that must be run under some kind of transactional control. For example, you’ll often open a file, do something with its contents, and then want to ensure that the file is closed when you finish. Although you can do this using conventional code, there’s an argument for making the file responsible for closing itself. We can do this with blocks. A naive implementation (ignoring error handling) might look something like the following.

```ruby
class File
  def openAndProcess(*args)
    f = File.open(*args)
    yield f
    f.close()
  end
end

File.openAndProcess("testfile", "r") do |aFile|
  print while aFile.gets
end
```

produces:

This is line one
This is line two
This is line three
And so on...

This small example illustrates a number of techniques. The `openAndProcess` method is a class method—-it may be called independent of any particular File object. We want it to take the same arguments as the conventional `File.open` method, but we don’t really care what those arguments are. Instead, we specified the arguments as `*args`, meaning “collect the actual parameters passed to the method into an array.” We then call `File.open`, passing it `*args` as a parameter. This expands the array back into individual parameters. The net result is that `openAndProcess` transparently passes whatever parameters it received to `File.open`.

Once the file has been opened, `openAndProcess` calls `yield`, passing the open file object to the block. When the block returns, the file is closed. In this way, the responsibility for closing an open file has been passed from the user of file objects back to the files themselves.

Finally, this example uses `do...end` to define a block. The only difference between this notation and using braces to define blocks is precedence: `do...end` binds lower than `{...}`. We discuss the impact of this on page 234.

The technique of having files manage their own lifecycle is so useful that the class `File` supplied with Ruby supports it directly. If `File.open` has an associated block, then that block will be invoked with a file object, and the file will be closed when the block terminates. This is interesting, as it means that `File.open` has two different behaviors: when called with a block, it executes the block and closes the file. When called without a block, it returns the file object. This is made possible by the method `Kernel::block_given?`, which returns true if a block is associated with the current method. Using it, you could implement `File.open` (again, ignoring error handling) using something like the following.

```ruby
class File
  def myOpen(*args)
    aFile = File.new(*args)
    # If there’s a block, pass in the file and close
    # the file when it returns
    if block_given?
      yield aFile
    else
      aFile.close
      aFile = nil
    end
    return aFile
  end
end
```

**Blocks Can Be Closures**

Let’s get back to our jukebox for a moment (remember the jukebox?). At some point we’ll be working on the code that handles the user interface—-the buttons that people press to select songs and control the jukebox. We’ll need to associate actions with
those buttons: press STOP and the music stops. It turns out that Ruby's blocks are a convenient way to do this. Let's start out by assuming that the people who made the hardware implemented a Ruby extension that gives us a basic button class. (We talk about extending Ruby beginning on page 169.)

```ruby
bStart = Button.new("Start")
bPause = Button.new("Pause")  # ...
```

What happens when the user presses one of our buttons? In the Button class, the hardware folks rigged things so that a callback method, `buttonPressed`, will be invoked. The obvious way of adding functionality to these buttons is to create subclasses of Button and have each subclass implement its own `buttonPressed` method.

```ruby
class StartButton < Button
  def initialize
    super("Start") # invoke Button's initialize
  end
  def buttonPressed
    # do start actions...
  end
end

bStart = StartButton.new
```

There are two problems here. First, this will lead to a large number of subclasses. If the interface to Button changes, this could involve us in a lot of maintenance. Second, the actions performed when a button is pressed are expressed at the wrong level; they are not a feature of the button, but are a feature of the jukebox that uses the buttons. We can fix both of these problems using blocks.

```ruby
class JukeboxButton < Button
  def initialize(label, &action)
    super(label)
    @action = action
  end
  def buttonPressed
    @action.call(self)
  end
end

bStart = JukeboxButton.new("Start") { songList.start }
bPause = JukeboxButton.new("Pause") { songList.pause }
```

The key to all this is the second parameter to `JukeboxButton#initialize`. If the last parameter in a method definition is prefixed with an ampersand (such as `&action`), Ruby looks for a code block whenever that method is called. That code block is converted to an object of class `Proc` and assigned to the parameter. You can then treat the parameter as any other variable. In our example, we assigned it to the instance variable `@action`. When the callback method `buttonPressed` is invoked, we use the `Proc#call` method on that object to invoke the block.

```
def nTimes(aThing)
  return proc { |n| aThing * n }
end

p1 = nTimes(23)
p1.call(3)   » 69
p1.call(4)   » 92
```

Let's look at a contrived example. This example uses the method `proc`, which converts a block to a `Proc` object.
p2 = nTimes("Hello ")
p2.call(3)   »   "Hello Hello Hello 

The method nTimes returns a Proc object that references the method's parameter, aThing. Even though that parameter is out of scope by the time the block is called, the parameter remains accessible to the block.
Classes and Objects

Classes and objects are obviously central to Ruby, but at first sight they can seem a little confusing. There seem to be a lot of concepts: classes, objects, class objects, instance methods, class methods, and singleton classes. In reality, however, Ruby has just a single underlying class and object structure, which we'll discuss in this chapter. In fact, the basic model is so simple, we can describe it in a single paragraph.

A Ruby object has three components: a set of flags, some instance variables, and an associated class. A Ruby class is an object of class Class, which contains all the object things plus a list of methods and a reference to a superclass (which is itself another class). All method calls in Ruby nominate a receiver (which is by default self, the current object). Ruby finds the method to invoke by looking at the list of methods in the receiver's class. If it doesn't find the method there, it looks in the superclass, and then in the superclass's superclass, and so on. If the method cannot be found in the receiver's class or any of its ancestors, Ruby invokes the method method_missing on the original receiver.

And that's it—the entire explanation. On to the next chapter.

```
"But wait," you cry, "I spent good money on this chapter. What about all this other stuff—singleton classes, class methods, and so on. How do they work?"
```

Good question.

How Classes and Objects Interact

All class/object interactions are explained using the simple model given above: objects reference classes, and classes reference zero or more superclasses. However, the implementation details can get a tad tricky.

We've found that the simplest way of visualizing all this is to draw the actual objects that Ruby implements. So, in the following pages we'll look at all the possible combinations of classes and objects. Note that these are not class diagrams in the UML sense; we're showing structures in memory and pointers between them.

Your Basic, Everyday Object

Let's start by looking at an object created from a simple class. Figure 19.1 on page 239 shows an object referenced by a variable, lucille, the object's class, Guitar, and that class's superclass, Object. Notice how the object's class reference (called klass for historical reasons that really bug Andy) points to the class object, and how the super pointer from that class references the parent class.

```
Figure not available...
```

When Ruby executes Guitar.strings(), it follows the same process as before: it goes to the receiver, class Guitar, follows the klass reference to class Guitar$'$, and finds the method.

Finally, note that an `S` has crept into the flags in class Guitar$'$$. The classes that Ruby creates automatically are marked internally as singleton classes. Singleton classes are treated slightly differently within Ruby. The most obvious difference from the outside is that they are effectively invisible: they will never appear in a list of objects returned from methods such as Module#ancestors or ObjectSpace::each_object.

Object-Specific Classes
Ruby allows you to create a class tied to a particular object. In the following example, we create two String objects. We then associate an anonymous class with one of them, overriding one of the methods in the object's base class and adding a new method.

```ruby
a = "hello"
b = a.dup
class <<a
  def to_s
    "The value is '#{self}'"
  end
  def twoTimes
    self + self
  end
end
a.to_s  »  "The value is 'hello'"
a.twoTimes »  "hellohello"
b.to_s   »  "hello"
```

This example uses the `class << obj` notation, which basically says `build me a new class just for object obj." We could also have written it as:

```ruby
a = "hello"
b = a.dup
def a.to_s
  "The value is '#{self}'"
end
def a.twoTimes
  self + self
end
a.to_s  »  "The value is 'hello'"
a.twoTimes »  "hellohello"
b.to_s   »  "hello"
```

The effect is the same in both cases: a class is added to the object `a`. This gives us a strong hint about the Ruby implementation: a singleton class is created and inserted as a's direct class. a's original class, String, is made this singleton's superclass. The before and after pictures are shown in Figure 19.3 on page 242.

Ruby performs a slight optimization with these singleton classes. If an object's klass reference already points to a singleton class, a new one will not be created. This means that the first of the two method definitions in the previous example will create a singleton class, but the second will simply add a method to it.

```
module SillyModule

Mixin Modules
```

When a class includes a module, that module's instance methods become available as instance methods of the class. It's almost as if the module becomes a superclass of the class that uses it. Not surprisingly, that's about how it works. When you include a module, Ruby creates an anonymous proxy class that references that module, and inserts that proxy as the direct superclass of the class that did the including. The proxy class contains references to the instance variables and methods of the module. This is important: the same module may be included in many different classes, and will appear in many different inheritance chains. However, thanks to the proxy class, there is still only one underlying module: change a method definition in that module, and it will change in all classes that include that module, both past and future.

```ruby
module SillyModule
```
```ruby
def hello
  "Hello."
end

class SillyClass
  include SillyModule
end
s = SillyClass.new
s.hello  # "Hello."

module SillyModule
  def hello
    "Hi, there!"
  end
end
s.hello  # "Hi, there!"
```

The relationship between classes and the modules they include is shown in Figure 19.4 on page 243. If multiple modules are included, they are added to the chain in order.

If a module itself includes other modules, a chain of proxy classes will be added to any class that includes that module, one proxy for each module that is directly or indirectly included.

### Extending Objects

Just as you can define an anonymous class for an object using `\class <<obj \`, you can mix a module into an object using `Object#extend`. For example:

```ruby
module Humor
  def tickle
    "hee, hee!"
  end
end
a = "Grouchy"
a.extend Humor
a.tickle  # "hee, hee!"
```

There is an interesting trick with `extend`. If you use it within a class definition, the module's methods become class methods.

```ruby
module Humor
  def tickle
    "hee, hee!"
  end
end
class Grouchy
  include Humor
  extend Humor
end
Grouchy.tickle  # "hee, hee!"
a = Grouchy.new
```
a.tickle  »  "hee, hee!"

This is because calling extend is equivalent to self.extend, so the methods are added to self, which in a class definition is the class itself.

**Class and Module Definitions**

Having exhausted the combinations of classes and objects, we can (thankfully) get back to programming by looking at the nuts and bolts of class and module definitions.

In languages such as C++ and Java, class definitions are processed at compile time: the compiler loads up symbol tables, works out how much storage to allocate, constructs dispatch tables, and does all those other obscure things we’d rather not think too hard about.

Ruby is different. In Ruby, class and module definitions are executable code. Although parsed at compile time, the classes and modules are created at runtime, when the definition is encountered. (The same is also true of method definitions.) This allows you to structure your programs far more dynamically than in most conventional languages.

You can make decisions once, when the class is being defined, rather than each time that objects of the class are used. The class in the following example decides as it is being defined what version of a decryption routine to create.

```ruby
class MediaPlayer
  include Tracing if $DEBUGGING
  if ::EXPORT_VERSION
    def decrypt(stream)
      raise "Decryption not available"
    end
  else
    def decrypt(stream)
      # ...
    end
  end
end
```

If class definitions are executable code, this implies that they execute in the context of some object: self must reference something. Let’s find out what it is.

```ruby
class Test
  puts "Type of self = #{self.type}"  
  puts "Name of self = #{self.name}"  
end
produces:
Type of self = Class
Name of self = Test
```

This means that a class definition is executed with that class as the current object. Referring back to the section about metaclasses on page 238, we can see that this means that methods in the metaclass and its superclasses will be available during the execution of the method definition. We can check this out.

```ruby
class Test
  def Test.sayHello
    puts "Hello from #{name}"  
  end
end
produces:
Hello from Test
```

In this example we define a class method, Test.sayHello, and then call it in the body of the class definition. Within sayHello, we call name, an instance method of class Module. Because Module is an ancestor of Class, its instance methods can be called
without an explicit receiver within a class definition.

In fact, many of the directives that you use when defining a class or module, things such as alias_method, attr, and public, are simply methods in class Module. This opens up some interesting possibilities—you can extend the functionality of class and module definitions by writing Ruby code. Let's look at a couple of examples.

As a first example, let's look at adding a basic documentation facility to modules and classes. This would allow us to associate a string with modules and classes that we write, a string that is accessible as the program is running. We'll choose a simple syntax.

```ruby
class Example
  doc "This is a sample documentation string"
  # .. rest of class
end
```

We need to make doc available to any module or class, so we need to make it an instance method of class Module.

```ruby
class Module
  @@docs = Hash.new(nil)
  def doc(str)
    @@docs[self.name] = str
  end

  def Module::doc(aClass)
    # If we're passed a class or module, convert to string
    # ('<=_' for classes checks for same class or subtype)
    aClass = aClass.name if aClass.type <= Module
    @@docs[aClass] || "No documentation for #{aClass}"
  end
end
```

```ruby
class Example
  doc "This is a sample documentation string"
  # .. rest of class
end
```

```ruby
module Another
  doc <<-edoc
    And this is a documentation string
  edoc
  # rest of module
end
```

```ruby
puts Module::doc(Example)
p<br>uts Module::doc("Another")
```

produces:

This is a sample documentation string
And this is a documentation string
in a module

The second example is a performance enhancement based on Tadayoshi Funaba's date module (described beginning on page 439). Say we have a class that represents some underlying quantity (in this case, a date). The class may have many attributes that present the same underlying date in different ways: as a Julian day number, as a string, as a [year, month, day] triple, and so on. Each value represents the same date and may involve a fairly complex calculation to derive. We therefore would like to calculate each attribute only once, when it is first accessed.

The manual way would be to add a test to each accessor:

```ruby
class ExampleDate
  def initialize(dayNumber)
    @dayNumber = dayNumber
  end
end
```
This is a clunky technique---let's see if we can come up with something sexier.

What we're aiming for is a directive that indicates that the body of a particular method should be invoked only once. The value returned by that first call should be cached. Thereafter, calling that same method should return the cached value without reevaluating the method body again. This is similar to Eiffel's once modifier for routines. We'd like to be able to write something like:

```ruby
class ExampleDate
  def asDayNumber
    @dayNumber
  end

  def asString
    unless @string
      # complex calculation
      @string = result
    end
    @string
  end

  def asYMD
    unless @ymd
      # another calculation
      @ymd = [ y, m, d ]
    end
    @ymd
  end

  once :asString, :asYMD
end
```

We can use once as a directive by writing it as a class method of ExampleDate, but what should it look like internally? The trick is to have it rewrite the methods whose names it is passed. For each method, it creates an alias for the original code, then creates a new method with the same name. This new method does two things. First, it invokes the original method (using the alias) and stores the resulting value in an instance variable. Second, it redefines itself, so that on subsequent calls it simply returns the value of the instance variable directly. Here's Tadayoshi Funaba's code, slightly reformatted.

```ruby
def ExampleDate.once(*ids)
  for id in ids
    module_eval << "end_eval"
    alias_method :__#{id.to_i}__, #{id.inspect}
    def #{id.id2name}(*args, &block)
      def self.#{id.id2name}(*args, &block)
        @__#{id.to_i__} = __#{id.to_i}__(*args, &block)
      end
      @__#{id.to_i}__ = __#{id.to_i}__(*args, &block)
    end
  end
end
```
This code uses `module_eval` to execute a block of code in the context of the calling module (or, in this case, the calling class). The original method is renamed `__nnn__`, where the `nnn` part is the integer representation of the method name's symbol id. The code uses the same name for the caching instance variable. The bulk of the code is a method that dynamically redefines itself. Note that this redefinition uses the fact that methods may contain nested singleton method definitions, a clever trick.

Understand this code, and you'll be well on the way to true Ruby mastery.

However, we can take it further. Look in the date module, and you'll see method `once` written slightly differently.

```ruby
class Date
  class << self
    def once(*ids)
      # ...
      end
    end
    # ...
  end
end
```

The interesting thing here is the inner class definition, `class << self`. This defines a class based on the object `self`, and `self` happens to be the class object for `Date`. The result? Every method within the inner class definition is automatically a class method of `Date`.

The `once` feature is generally applicable---it should work for any class. If you took `once` and made it a private instance method of class `Module`, it would be available for use in any Ruby class.

### Class Names Are Constants

We've said that when you invoke a class method, all you're doing is sending a message to the Class object itself. When you say something such as `String.new("gumby")`, you're sending the message `new` to the object that is class `String`. But how does Ruby know to do this? After all, the receiver of a message should be an object reference, which implies that there must be a constant called `"String"` somewhere containing a reference to the String object. *It will be a constant, not a variable, because "String" starts with an uppercase letter.* And in fact, that's exactly what happens. All the built-in classes, along with the classes you define, have a corresponding global constant with the same name as the class. This is both straightforward and subtle. The subtlety comes from the fact that there are actually two things named (for example) `String` in the system. There's a constant that references an object of class `String`, and there's the object itself.

The fact that class names are just constants means that you can treat classes just like any other Ruby object: you can copy them, pass them to methods, and use them in expressions.

```ruby
def factory(klass, *args)
  klass.new(*args)
end

factory(String, "Hello")       » "Hello"
factory(Dir, ".")               » #<Dir:0x401b51bc>

flag = true
(flag ? Array : Hash)[1, 2, 3, 4]  » [1, 2, 3, 4]
flag = false
(flag ? Array : Hash)[1, 2, 3, 4]  » {1=>2, 3=>4}
```

### Top-Level Execution Environment

Many times in this book we've claimed that everything in Ruby is an object. However, there's one thing that we've used time and time again that appears to contradict this---the top-level Ruby execution environment.

```ruby
puts "Hello, World"
```

Not an object in sight. We may as well be writing some variant of Fortran or QW-Basic. But dig deeper, and you'll come across
objects and classes lurking in even the simplest code.

We know that the literal "Hello, World" generates a Ruby String, so there's one object. We also know that the bare method call to puts is effectively the same as self.puts. But what is `self`?

```ruby
self.type                => Object
```

At the top level, we're executing code in the context of some predefined object. When we define methods, we're actually creating (private) singleton methods for this object. Instance variables belong to this object. And because we're in the context of Object, we can use all of Object's methods (including those mixed-in from Kernel) in function form. This explains why we can call Kernel methods such as puts at the top level (and indeed throughout Ruby): these methods are part of every object.

### Inheritance and Visibility

There's one last wrinkle to class inheritance, and it's fairly obscure.

Within a class definition, you can change the visibility of a method in an ancestor class. For example, you can do something like:

```ruby
class Base
  def aMethod
    puts "Got here"
  end
  private :aMethod
end

class Derived1 < Base
  public :aMethod
end

class Derived2 < Base
end
```

In this example, you would be able to invoke `aMethod` in instances of class Derived1, but not via instances of Base or Derived2.

So how does Ruby pull off this feat of having one method with two different visibilities? Simply put, it cheats.

If a subclass changes the visibility of a method in a parent, Ruby effectively inserts a hidden proxy method in the subclass that invokes the original method using `super`. It then sets the visibility of that proxy to whatever you requested. This means that the code:

```ruby
class Derived1 < Base
  public :aMethod
end
```

is effectively the same as:

```ruby
class Derived1 < Base
  def aMethod(*args)
    super
  end
  public :aMethod
end
```

The call to `super` can access the parent's method regardless of its visibility, so the rewrite allows the subclass to override its parent's visibility rules. Pretty scary, eh?

### Freezing Objects

There are times when you've worked hard to make your object exactly right, and you'll be damned if you'll let anyone just change it. Perhaps you need to pass some kind of opaque object between two of your classes via some third-party object, and you want to make sure it arrives unmodified. Perhaps you want to use an object as a hash key, and need to make sure that no
one modifies it while it’s being used. Perhaps something is corrupting one of your objects, and you’d like Ruby to raise an exception as soon as the change occurs.

Ruby provides a very simple mechanism to help with this. Any object can be frozen by invoking `Object#freeze`. A frozen object may not be modified: you can’t change its instance variables (directly or indirectly), you can’t associate singleton methods with it, and, if it is a class or module, you can’t add, delete, or modify its methods. Once frozen, an object stays frozen: there is no `Object#thaw`. You can test to see if an object is frozen using `Object#frozen?`.

What happens when you copy a frozen object? That depends on the method you use. If you call an object’s clone method, the entire object state (including whether it is frozen) is copied to the new object. On the other hand, `dup` typically copies only the object’s contents---the new copy will not inherit the frozen status.

```ruby
str1 = "hello"
str1.freeze » "hello"
str1.frozen? » true
str2 = str1.clone
str2.frozen? » true
str3 = str1.dup
str3.frozen? » false
```

Although freezing objects may initially seem like a good idea, you might want to hold off doing it until you come across a real need. Freezing is one of those ideas that looks essential on paper but isn’t used much in practice.