11

SYMBOLS

Many newcomers to Ruby are confused by symbols. A symbol is an identifier whose first character is a colon (`:`), so `:this` is a symbol and so is `:that`. Symbols are, in fact, not at all complicated—and, in certain circumstances, they may be extremely useful, as you will see shortly.

Let's first be clear about what a symbol is *not*: It is not a string, it is not a constant, and it is not a variable. A symbol is, quite simply, an identifier with no intrinsic meaning other than its own name. Whereas you might assign a value to a variable like this . . .

```
name = "Fred"
```

you would *not* assign a value to a symbol:

```
:name = "Fred"    # Error!
```

The value of a symbol is itself. So, the value of a symbol called `:name` is `:name`.

NOTE *For a more technical account of what a symbol is, refer to"Digging Deeper" on page 190.*

You have, of course, used symbols before. In Chapter 2, for instance, you created attribute readers and writers by passing symbols to the `attr_reader` and `attr_writer` methods, like this:

```
attr_reader( :description )
attr_writer( :description )
```

You may recall that the previous code causes Ruby to create a `@description` instance variable plus a pair of getter (reader) and setter (writer) methods called `description`. Ruby takes the value of a symbol literally. The `attr_reader` and `attr_writer` methods create, from that name, variables and methods with matching names.

Symbols and Strings

It is a common misconception that a symbol is a type of string. After all, isn't the symbol `:hello` pretty similar to the string `"hello"`? In fact, symbols are quite unlike strings. For one thing, each string is different—so, as far as Ruby is concerned, `"hello"`, `"hello"`, and `"hello"` are three separate objects with three separate `object_ids`.

symbol_ids.rb

```
# These 3 strings have 3 different object_ids
puts( "hello".object_id ) #=> 16589436
puts( "hello".object_id ) #=> 16589388
puts( "hello".object_id ) #=> 16589340
```

But a symbol is unique, so `:hello`, `:hello`, and `:hello` all refer to the same object with the same `object_id`.

```
# These 3 symbols have the same object_id
puts( :hello.object_id ) #=> 208712
puts( :hello.object_id ) #=> 208712
puts( :hello.object_id ) #=> 208712
```

In this respect, a symbol has more in common with an integer than with a string. Each occurrence of a given integer value, you may recall, refers to the same object, so `10`, `10`, and `10` may be considered to be the same object, and they have the same `object_id`. Remember that the actual IDs assigned to objects will change each time you run a program. The number itself is not

significant. The important thing to note is that each separate object always has a unique ID, so when an ID is repeated, it indicates repeated references to the same object.

ints_and_symbols.rb

```
# These three symbols have the same object_id
puts( :ten.object_id )  #=> 20712
puts( :ten.object_id )  #=> 20712
puts( :ten.object_id )  #=> 20712

# These three integers have the same object_id
puts( 10.object_id )    #=> 21
puts( 10.object_id )    #=> 21
puts( 10.object_id )    #=> 21
```

You can also test for equality using the `equal?` method:

symbols_strings.rb

```
puts( :helloworld.equal?( :helloworld ) )     #=> true
puts( "helloworld".equal?( "helloworld" ) )   #=> false
puts( 1.equal?( 1 ) )                         #=> true
```

Being unique, a symbol provides an unambiguous identifier. You can pass symbols as arguments to methods, like this:

```
amethod( :deletefiles )
```

A method might contain code to test the value of the incoming argument:

symbols_1.rb

```
def amethod( doThis )
    if (doThis == :deletefiles) then
       puts( 'Now deleting files...')
    elsif (doThis == :formatdisk) then
       puts( 'Now formatting disk...')
    else
        puts( "Sorry, command not understood." )
    end
end
```

Symbols can also be used in `case` statements where they provide both the readability of strings and the uniqueness of integers:

```
case doThis
    when :deletefiles then puts( 'Now deleting files...')
    when :formatdisk then puts( 'Now formatting disk...')
    else  puts( "Sorry, command not understood." )
end
```

The scope in which a symbol is declared does not affect its uniqueness. Consider the following:

symbol_ref.rb

```
module One
     class Fred
     end
     $f1 = :Fred
end

module Two
     Fred = 1
     $f2 = :Fred
end

def Fred()
end

$f3 = :Fred
```

Here, the variables `$f1`, `$f2`, and `$f3` are assigned the symbol `:Fred` in three different scopes: module `One`, module `Two`, and the "main" scope. Variables starting with `$` are global, so once created, they can be referenced anywhere. I'll have more to say on modules in Chapter 12. For now, just think of them as "namespaces" that define different scopes. And yet each variable refers to the same symbol, `:Fred`, and has the same `object_id`.

```
# All three display the same id!
puts( $f1.object_id )  #=> 208868
puts( $f2.object_id )  #=> 208868
puts( $f3.object_id )  #=> 208868
```

Even so, the "meaning" of the symbol changes according to its scope. In module `One`, `:Fred` refers to the class `Fred`; in module `Two`, it refers to the constant `Fred = 1`; and in the main scope, it refers to the method `Fred`.

A rewritten version of the previous program demonstrates this:

symbol_ref2.rb

```
module One
    class Fred
    end
    $f1 = :Fred
    def self.evalFred( aSymbol )
        puts( eval( aSymbol.id2name ) )
    end
end

module Two
    Fred = 1
    $f2 = :Fred
    def self.evalFred( aSymbol )
        puts( eval( aSymbol.id2name ) )
    end
end
```

```
def Fred()
    puts( "hello from the Fred method" )
end

$f3 = :Fred
```

First I access the `evalFred` method inside the module named `One` using two colons (`::`), which is the Ruby "scope resolution operator." I then pass `$f1` to that method:

```
One::evalFred( $f1 )
```

In this context, `Fred` is the name of a class defined inside module `One`, so when the `:Fred` symbol is evaluated, the module and class names are displayed:

```
One::Fred
```

Next I pass `$f2` to the `evalFred` method of module `Two`:

```
Two::evalFred( $f2 )
```

In this context, `Fred` is the name of a constant that is assigned the integer 1, so that is what is displayed: `1`. And finally, I call a special method called simply `method`. This is a method of Object. It tries to find a method with the same name as the symbol passed to it as an argument and, if found, returns that method as an object that can then be called:

```
method($f3).call
```

The `Fred` method exists in the main scope, and when called, its output is this string:

```
"hello from the Fred method"
```

Naturally, since the variables `$f1`, `$f2`, and `$f3` reference the same symbol, it doesn't matter which variable you use at any given point. Any variable to which a symbol is assigned, or, indeed, the symbol itself, will produce the same results. The following are equivalent:

```
One::evalFred( $f1 )    #=> One::Fred
Two::evalFred( $f2 )    #=> 1
method($f3).call        #=> hello from the Fred method

One::evalFred( $f3 )    #=> One::Fred
Two::evalFred( $f1 )    #=> 1
method($f2).call        #=> hello from the Fred method
```

```
One::evalFred( :Fred )  #=> One::Fred
Two::evalFred( :Fred )  #=> 1
method(:Fred).call      #=> hello from the Fred method
```

Symbols and Variables

To understand the relationship between a symbol and an identifier such as a variable name, take a look at the *symbols_2.rb* program. It begins by assigning the value 1 to a local variable, `x`. It then assigns the symbol `:x` to a local variable, `xsymbol`:

symbols_2.rb

```
x = 1
xsymbol = :x
```

At this point, there is no obvious connection between the variable `x` and the symbol `:x`. I have declared a method that simply takes some incoming argument and inspects and displays it using the `p` method. I can call this method with the variable and the symbol:

```
def amethod( somearg )
    p( somearg )
end

# Test 1
amethod( x )
amethod( :x )
```

This is the data that the method prints as a result:

```
1
:x
```

In other words, the value of the `x` variable is 1, since that's the value assigned to it and the value of `:x` is `:x`. But the interesting question that arises is this: If the value of `:x` is `:x` and this is also the symbolic name of the variable `x`, would it be possible to use the symbol `:x` to find the value of the variable `x`? Confused? I hope the next line of code will make this clearer:

```
# Test 2
amethod( eval(:x.id2name))
```

Here, `id2name` is a method of the Symbol class. It returns the name or string corresponding to the symbol (the `to_s` method would perform the same function); the end result is that, when given the symbol `:x` as an argument, `id2name` returns the string "x." Ruby's `eval` method (which is defined in the Kernel

class) is able to evaluate expressions within strings. In the present case, that means it finds the string "x" and tries to evaluate it as though it were executable code. It finds that x is the name of a variable and that the value of x is 1. So, the value 1 is passed to amethod. You can verify this by running *symbols2.rb*.

NOTE *Evaluating data as code is explained in more detail in Chapter 20.*

Things can get even trickier. Remember that the variable xsymbol has been assigned the symbol :x.

```
x = 1
xsymbol = :x
```

That means that if you eval :xsymbol, you can obtain the name assigned to it—that is, the symbol :x. Having obtained :x, you can go on to evaluate this also, giving the value of x, namely, 1:

```
# Test 3
amethod( xsymbol )                                          #=> :x
amethod( :xsymbol )                                         #=> :xsymbol
amethod( eval(:xsymbol.id2name))                            #=> :x
amethod( eval( ( eval(:xsymbol.id2name)).id2name ) )        #=> 1
```

As you've seen, when used to create attribute accessors, symbols can refer to method names. You can make use of this by passing a method name as a symbol to the method method and then calling the specified method using the call method:

```
#Test 4
method(:amethod).call("")
```

The call method lets you pass arguments, so, just for the heck of it, you could pass an argument by evaluating a symbol:

```
method(:amethod).call(eval(:x.id2name))
```

If this seems complicated, take a look at a simpler example in *symbols_3.rb*. This begins with this assignment:

symbols_3.rb

```
def mymethod( somearg )
   print( "I say: " << somearg )
end

this_is_a_method_name = method(:mymethod)
```

Here `method(:mymethod)` looks for a method with the name specified by the symbol passed as an argument (`:mymethod`), and if one is found, it returns the Method object with the corresponding name. In my code I have a method called `mymethod`, and this is now assigned to the variable `this_is_a_method_name`.

When you run this program, you will see that the first line of output prints the value of the variable:

```
puts( this_is_a_method_name )  #=> #<Method: Object#mymethod>
```

This shows that the variable `this_is_a_method_name` has been assigned the method, `mymethod`, which is bound to the Object class (as are all methods that are entered as "freestanding" functions). To double-check that the variable really is an instance of the Method class, the next line of code prints out its class:

```
puts( "#{this_is_a_method_name.class}" )  #=> Method
```

Okay, so if it's really and truly a method, then you should be able to call it, shouldn't you? To do that, you need to use the `call` method. That is what the last line of code does:

```
this_is_a_method_name.call( "hello world" )  #=>I say: hello world
```

Why Use Symbols?

Some methods in the Ruby class library specify symbols as arguments. Naturally, if you need to call those methods, you are obliged to pass symbols to them. Other than in those cases, however, there is no absolute requirement to use symbols in your own programming. For many Ruby programmers, the "conventional" data types such as strings and integers are perfectly sufficient. However, many Ruby programmers do like to use symbols as the keys into hashes. When you look at the Rails framework in Chapter 19, for example, you will see examples similar to the following:

```
{ :text => "Hello world" }
```

Symbols do have a special place in "dynamic" programming, however. For example, a Ruby program is able to create a new method at runtime by calling, within the scope of a certain class, `define_method` with a symbol representing the method to be defined and a block representing the code of the method:

add_method.rb

```
class Array
    define_method( :aNewMethod, lambda{
        |*args| puts( args.inspect)
    } )
end
```

After the previous code executes, the Array class will have gained a method named `aNewMethod`. You can verify this by calling `method_defined?` with a symbol representing the method name:

```
Array.method_defined?( :aNewMethod )   #=> true
```

And, of course, you can call the method itself:

```
[].aNewMethod( 1,2,3     #=> [1,2,3]
```

You can remove an existing method at runtime in a similar way by calling `remove_method` inside a class with a symbol providing the name of the method to be removed:

```
class Array
    remove_method( :aNewMethod )
end
```

Dynamic programming is invaluable in applications that need to modify the behavior of the Ruby program while that program is still executing. Dynamic programming is widely used in the Rails framework, for example, and it is discussed in depth in the final chapter of this book.

DIGGING DEEPER

Symbols are fundamental to Ruby. Here you will learn why that is so and how you can display all the symbols available.

What Is a Symbol?

Previously, I said that a symbol is an identifier whose value is itself. That describes, in a broad sense, the way that symbols behave from the point of view of the Ruby programmer. But it doesn't tell you what symbols are *literally* from the point of view of the Ruby interpreter. A symbol is, in fact, a pointer into the symbol table. The symbol table is Ruby's internal list of known identifiers—such as variable and method names.

If you want to take a peek deep inside Ruby, you can display all the symbols that Ruby knows about like this:

allsymbols.rb

```
p( Symbol.all_symbols )
```

This will shows thousands of symbols including method names such as `:to_s` and `:reverse`, global variables such as `:$/` and `:$DEBUG`, and class names such as `:Array` and `:Symbol`. You may restrict the number of symbols displayed using array indexes like this:

```
p( Symbol.all_symbols[0,10] )
```

In Ruby 1.8, you can't sort symbols since symbols are not considered to be inherently sequential. In Ruby 1.9, sorting is possible, and the symbol characters are sorted as though they were strings:

```
# In Ruby 1.9
p [:a,:c,:b].sort       #=> [:a,:b,:c]

# In Ruby 1.8
p [:a,:c,:b].sort       #=> 'sort': undefined method '<=>' for :a:Symbol
```

The easiest way to display a sorted list of symbols in a way that avoids incompatibility problems related to Ruby versions is to convert the symbols to strings and sort those. In the following code, I pass all the symbols known to Ruby into a block, convert each symbol to a string, and collect the strings into a new array that is assigned to the `str_array` variable. Now I can sort this array and display the results:

```
str_arr = Symbol.all_symbols.collect{ |s| s.to_s }
puts( str_arr.sort )
```