11

RUBINIUS: RUBY IMPLEMENTED WITH RUBY

Like JRuby, Rubinius is an alternative implementation of Ruby. Much of Rubinius's internal source code is written in Ruby itself instead of in only C or Java. Rubinius implements built-in classes, such as `Array`, `String`, and `Integer`, just as you would—with Ruby code!

This design offers a unique opportunity for you to learn about Ruby internals. If you aren't sure how a particular Ruby feature or method works, you can read the Ruby code inside Rubinius to find out, without special knowledge of C or Java programming.

Rubinius also includes a sophisticated virtual machine written in C++. This machine executes your Ruby program and, like JRuby, supports JIT and true concurrency and uses a sophisticated garbage collection algorithm.

This chapter starts with a high-level overview of Rubinius and an example of how to use backtrace output to dig through the Rubinius source code. Later in the chapter, we'll learn how Rubinius and MRI implement the `Array` class, including how Ruby saves data into an array and what happens when you remove an element from an array.

ROADMAP

The Rubinius Kernel and Virtual Machine

To run a Ruby program using Rubinius (see Figure 11-1), you typically use the `ruby` command (as with MRI) or `rbx` because the `ruby` command is actually a symbolic link to the executable *rbx* in Rubinius.

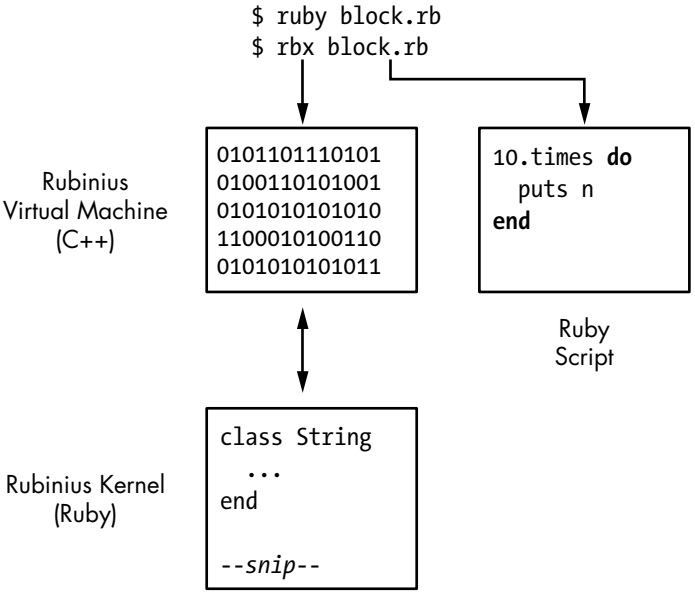

Figure 11-1: Rubinius consists of a C++ virtual machine and a Ruby kernel.

As with MRI, you launch Rubinius using an executable that reads and executes the Ruby program specified on the command line. But the Rubinius executable is completely different from the standard Ruby executable. As the preceding figure shows, Rubinius consists of two major pieces:

The Rubinius kernel This is the part of Rubinius written in Ruby. It implements a lot of the language, including the definitions of many built-in, core classes, such as `String` and `Array`. The Rubinius kernel is compiled into bytecode instructions that are installed onto your computer.

The Rubinius virtual machine The Rubinius virtual machine is written in C++. It executes the bytecode instructions from the Rubinius kernel and performs a range of other low-level tasks, such as garbage collection. The Rubinius executable contains a compiled, machine-language version of this virtual machine.

Figure 11-2 takes a closer look at Rubinius's virtual machine and kernel. The Rubinius kernel contains a set of Ruby classes, such as `String`, `Array`, and `Object`, as well as other Ruby classes that perform various tasks, such as compiling or loading code. The Rubinius virtual machine at the left of the figure is the *rbx* executable that you launch from the command line. The C++ virtual machine contains code to perform garbage collection, just-in-time compilation (and many other tasks), as well as additional code for built-in classes, such as `String` or `Array`. In fact, as indicated by the arrows, each Ruby class built into Rubinius consists of both C++ and Ruby code working together. Rubinius defines certain methods using Ruby and other methods using C++.

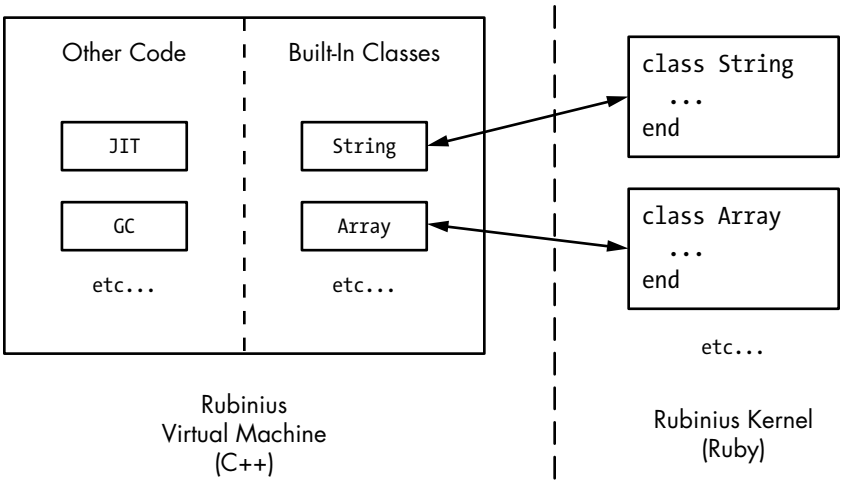

Figure 11-2: A closer view of Rubinius internals

Why implement Ruby using two languages? Because C++ speeds up Rubinius programs and allows them to interact with the operating system directly at a low level. The use of C++ instead of C also allows Rubinius to use an elegant object-oriented design internally. And the use of Ruby to implement built-in classes and other features makes it easy for Ruby developers to read and understand much of the Rubinius source code.

Tokenization and Parsing

Rubinius processes your Ruby program in much the same way that MRI does, as shown in Figure 11-3.

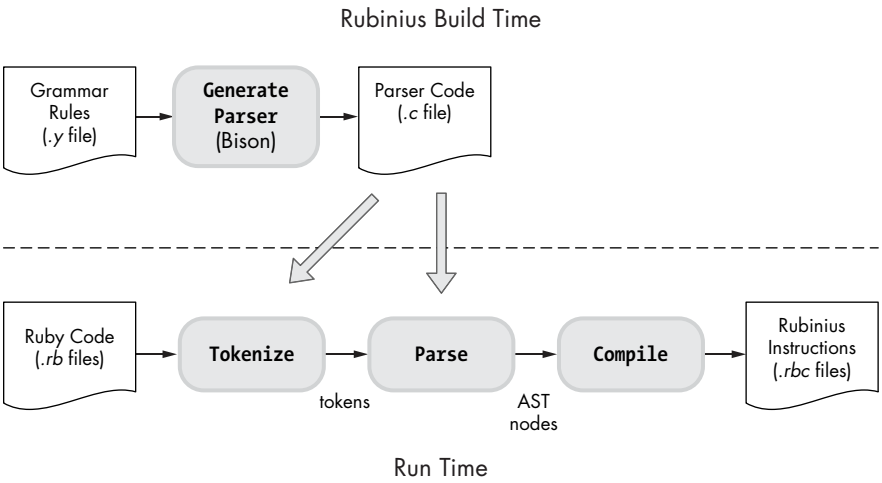

Figure 11-3: How Rubinius processes your code

Rubinius generates an LALR parser using Bison during its build process, just as MRI does. When you run your program, the parser converts your code into a token stream, an abstract syntax tree (AST) structure, and then a series of high-level virtual machine instructions called *Rubinius instructions*. Figure 11-4 compares the forms that your code takes inside MRI and Rubinius.

At first, Rubinius and MRI work similarly, but instead of interpreting your code as MRI does, Rubinius uses a compiler framework called the Low-Level Virtual Machine (LLVM) to compile your code again into lower-level instructions. LLVM, in turn, may compile these instructions all the way to machine language, using a JIT compiler.

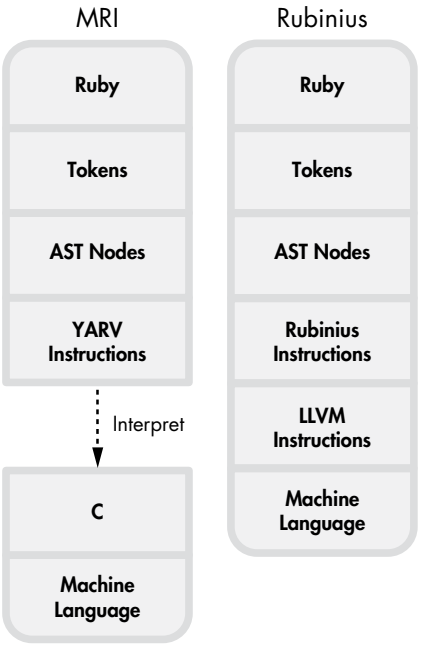

Figure 11-4: How MRI and Rubinius transform your code internally

Using Ruby to Compile Ruby

One of the most fascinating aspects of Rubinius is how it implements a Ruby compiler with a combination of Ruby and C++. When you run a program using Rubinius, your code is processed by both C++ and Ruby code, as shown in Figure 11-5.

At the top left of the diagram, Rubinius, like MRI, uses C code to parse Ruby code with a series of grammar rules. At right, Rubinius starts to process your Ruby program using Ruby code, representing each node in the AST with an instance of a Ruby class. Each Ruby AST node knows how to generate

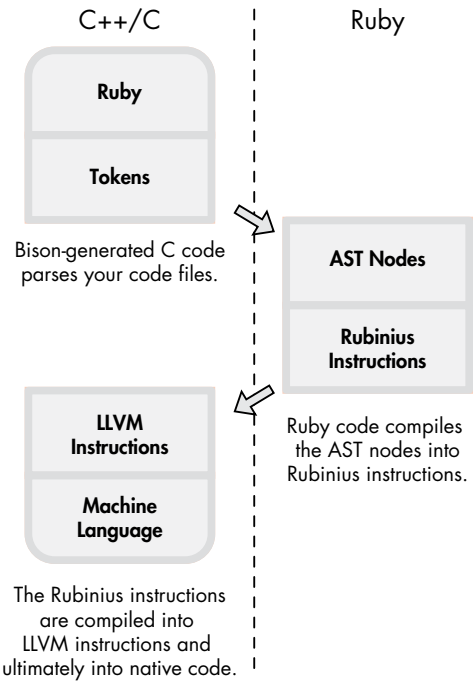

Figure 11-5: A high-level overview of how Rubinius compiles your code

Rubinius instructions for its piece of your program during compilation. Finally, at bottom left, the LLVM framework further compiles the Rubinius instructions into LLVM instructions and ultimately into machine language.

Rubinius Bytecode Instructions

To get a sense of Rubinius instructions, let's run a short program using Rubinius (see Listing 11-1).

```
$ cat simple.rb
puts 2+2
$ rbx simple.rb
4
```

Listing 11-1: Using Rubinius to calculate 2 + 2 = 4 (simple.rb)

When we rerun *simple.rb* using the `rbx compile` command with the `-B` option, Rubinius displays the bytecode instructions its compiler generates, as shown in Listing 11-2.

```
$ rbx compile simple.rb -B
============= :__script__ ==============
Arguments:   0 required, 0 post, 0 total
Arity:       0
Locals:      0
Stack size:  3
Literals:    2: :+, :puts
Lines to IP: 1: 0..12

  0000:  push_self
  0001:  meta_push_2
  0002:  meta_push_2
❶ 0003:  send_stack                :+, 1
  0006:  allow_private
❷ 0007:  send_stack                :puts, 1
  0010:  pop
  0011:  push_true
  0012:  ret
----------------------------------------
```

Listing 11-2: Displaying Rubinius bytecode instructions using the `rbx compile` command with the `-B` option

The instructions vaguely resemble MRI's YARV instructions. Each instruction typically pushes a value onto an internal stack, operates on stack values, or executes a method such as the `+` at ❶ or `puts` at ❷.

Figure 11-6 shows both the Ruby code and corresponding Rubinius instructions for *simple.rb* and part of the `Kernel` module.

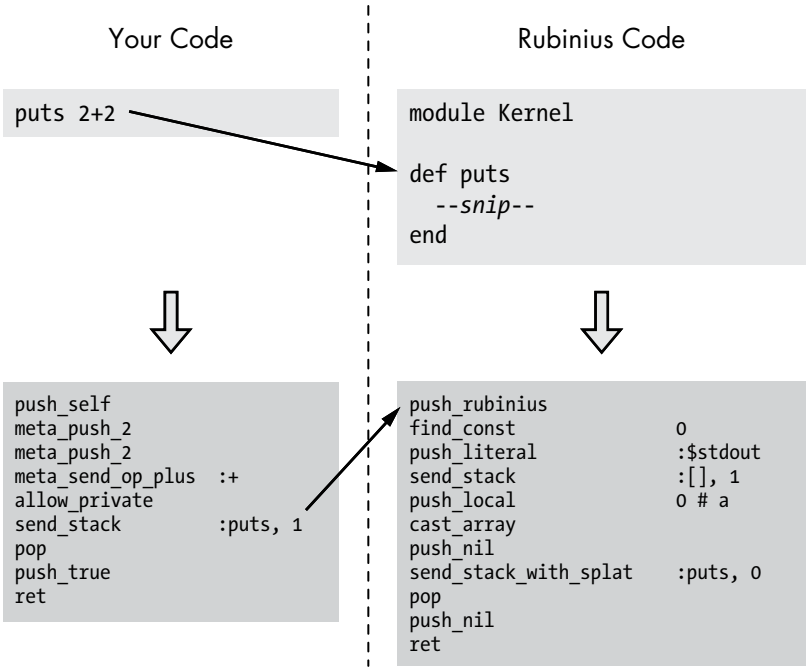

Figure 11-6: The puts method in Rubinius is implemented with Ruby code.

You can see Ruby code at the top of the figure: the `puts 2+2` code at left and Rubinius's definition of the `puts` method at right. Rubinius implements built-in Ruby classes, such as the `Kernel` module, in Ruby; therefore, when we call the `puts` method, Rubinius simply passes control to the Ruby code for the `Kernel#puts` method contained inside the Rubinius kernel.

The lower portion of the figure shows the Rubinius instructions into which the Ruby code is compiled. At left are the instructions for `puts 2+2`, and at right is the compiled version of the `Kernel#puts` method. Rubinius compiles its built-in Ruby code and your Ruby code in the same manner (except that Rubinius compiles the built-in Ruby code during the Rubinius build process).

Ruby and C++ Working Together

In order to handle certain low-level technical details and to speed things up, Rubinius uses C++ code in its virtual machine to help implement built-in classes and modules. That is, it uses both Ruby and C++ to implement the language's core classes.

To understand how this works, let's execute this short Ruby script in Rubinius (see Listing 11-3).

```
str = "The quick brown fox..."
puts str[4]
 => q
```

Listing 11-3: Calling the `String#[]` method

This simple program prints the fifth character (the letter *q* at index 4) in the sample string. Because the `String#[]` method is part of a built-in Ruby class, Rubinius implements it using Ruby code, as shown in Figure 11-7.

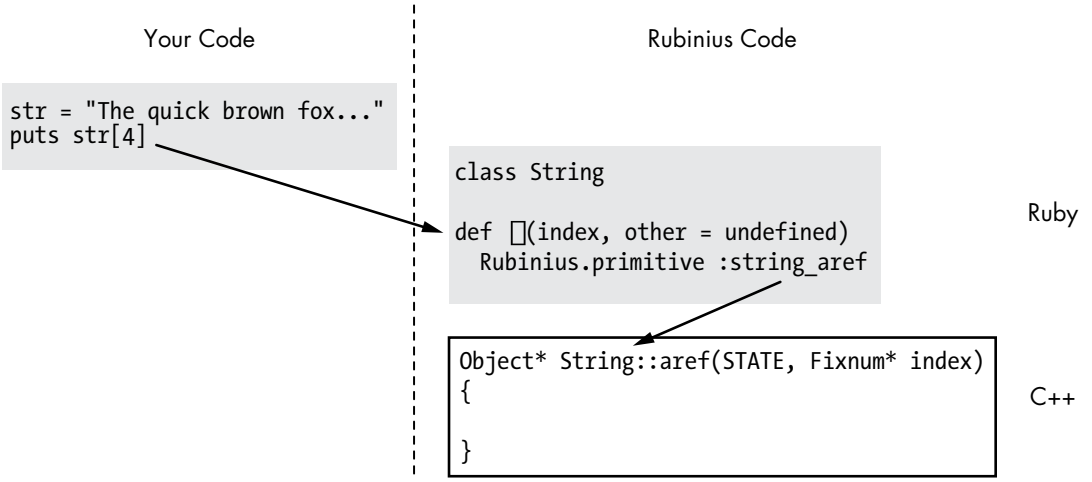

Figure 11-7: Rubinius implements built-in classes with a combination of Ruby and C++ code.

On the left of the figure is the Ruby script that prints the letter *q*. On the right is the Ruby code that Rubinius uses to implement the `String#[]` method, taken from a Rubinius source code file called *string.rb* (named after the `String` class). (We'll learn how to find Rubinius source code files in Experiment 11-1.)

Notice that the beginning of `String#[]` starts with the method call `Rubinius.primitive`. This indicates that Rubinius actually uses C++ code to implement this method; `Rubinius.primitive` is a directive that tells the Rubinius compiler to generate a call to the corresponding C++ code. The code that actually implements `String#[]` is a C++ method called `String::aref`, shown at the bottom right of Figure 11-7.

Implementing Ruby Objects with C++ Objects

Ruby's use of the object-oriented C++ allows its virtual machine to represent each Ruby object internally using a corresponding C++ object (see Figure 11-8).

Rubinius uses C++ objects the way that MRI uses the `RClass` and `RObject` C structures. When you define a class, Rubinius creates an instance of the `Class` C++ class. When you create a Ruby object, Rubinius creates an

instance of the `Object` C++ class. A `klass_` pointer in the `pythagoras` object indicates it is an instance of `Mathematician`, just as the `klass` pointer in the `RObject` C structure does in MRI.

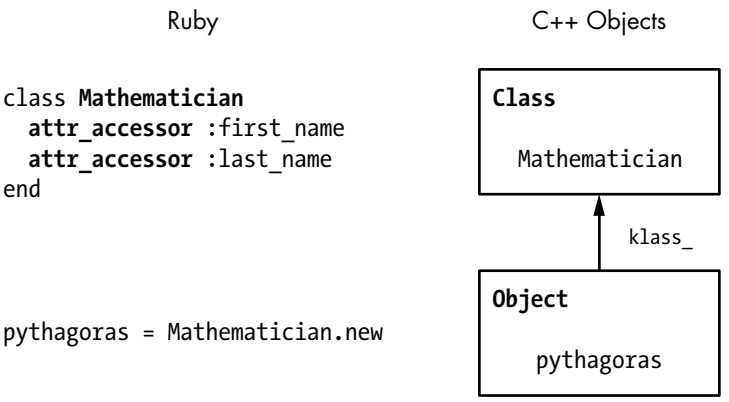

Figure 11-8: Rubinius represents classes and objects using C++ objects.

Experiment 11-1: Comparing Backtraces in MRI and Rubinius

Recall that Ruby displays a backtrace when an exception occurs in order to help you find the problem. Listing 11-4 shows a simple example.

```
10.times do |n|
  puts n
  raise "Stop Here"
end
```

Listing 11-4: A Ruby script that raises an exception

We call `raise` to tell Ruby to stop the first time it executes the block after displaying the value of the parameter `n`. Listing 11-5 shows the output from running Listing 11-4 with MRI.

```
$ ruby iterate.rb
0
iterate.rb:3:in 'block in <main>': Stop Here (RuntimeError)
    from iterate.rb:1:in 'times'
    from iterate.rb:1:in '<main>'
```

Listing 11-5: How MRI displays a backtrace for an exception

You probably see output like this many times while developing a Ruby program. However, one subtle detail is worth a closer look. Figure 11-9 shows a diagram of the MRI backtrace output.

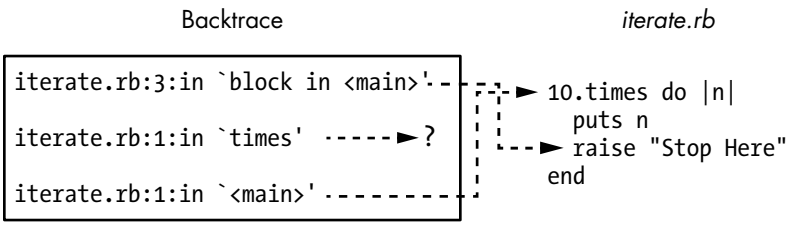

Figure 11-9: MRI displays where built-in CFUNC methods are called, not where they are defined.

Notice that line 3 of *iterate.rb*, containing the call to `raise`, is at the top of the call stack. At the bottom of the call stack, MRI displays `iterate.rb:1`, where the short script began.

Notice, too, that MRI's backtrace contains a broken link: *iterate.rb* doesn't contain a definition for the method `times`. Instead, MRI refers to the line of code that *calls* the `times` method: `10.times do`. The actual `times` method is implemented with C code inside MRI—a CFUNC method. MRI displays the location of calls to CFUNC methods in backtraces, not the location of the actual C implementation of these methods.

Backtraces in Rubinius

Unlike MRI, Rubinius implements built-in methods using Ruby, not C. This implementation allows Rubinius to include accurate source file and line number information about built-in methods in backtraces. To demonstrate, let's run Listing 11-4 again using Rubinius. Listing 11-6 shows the result.

```
$ rbx iterate.rb
0
An exception occurred running iterate.rb
    Stop Here (RuntimeError)

Backtrace:
          { } in Object#__script__ at iterate.rb:3
              Integer(Fixnum)#times at kernel/common/integer.rb:83
                  Object#__script__ at iterate.rb:1
  Rubinius::CodeLoader#load_script at kernel/delta/codeloader.rb:68
  Rubinius::CodeLoader.load_script at kernel/delta/codeloader.rb:119
           Rubinius::Loader#script at kernel/loader.rb:645
             Rubinius::Loader#main at kernel/loader.rb:844
```

Listing 11-6: How Rubinius displays a backtrace for an exception

Rubinius displays much more information! To understand this output a bit better, see Figures 11-10 and 11-11.

At left in Figure 11-10 is a simplified version of the backtrace information Rubinius displayed while running *iterate.rb*. Rubinius displays the two lines in the backtrace corresponding to *iterate.rb* just as MRI does. But Rubinius

also includes new entries in the Ruby call stack that correspond to Ruby source code files inside the Rubinius kernel. We can guess that the *loader.rb* and *codeloader.rb* files contain code that load and execute our script.

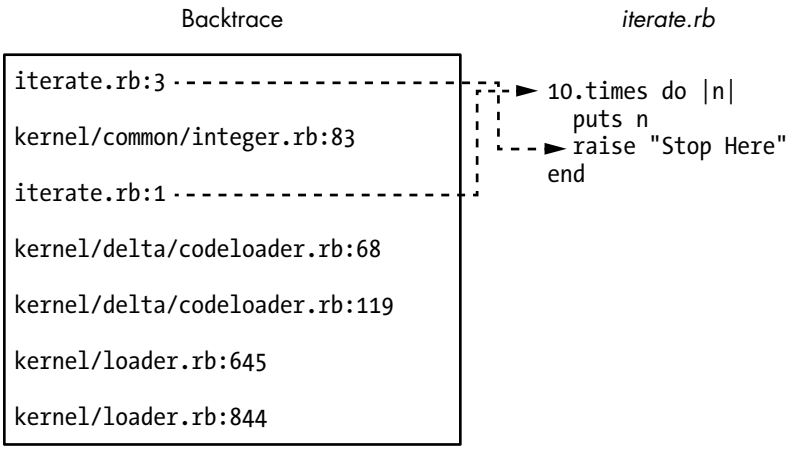

Figure 11-10: Like MRI, Rubinius includes information about your program in backtraces.

But the most interesting entry in the call stack is `kernel/common/integer.rb:83`. This entry tells us where the `Integer#times` method is implemented inside the Rubinius kernel, as shown in Figure 11-11.

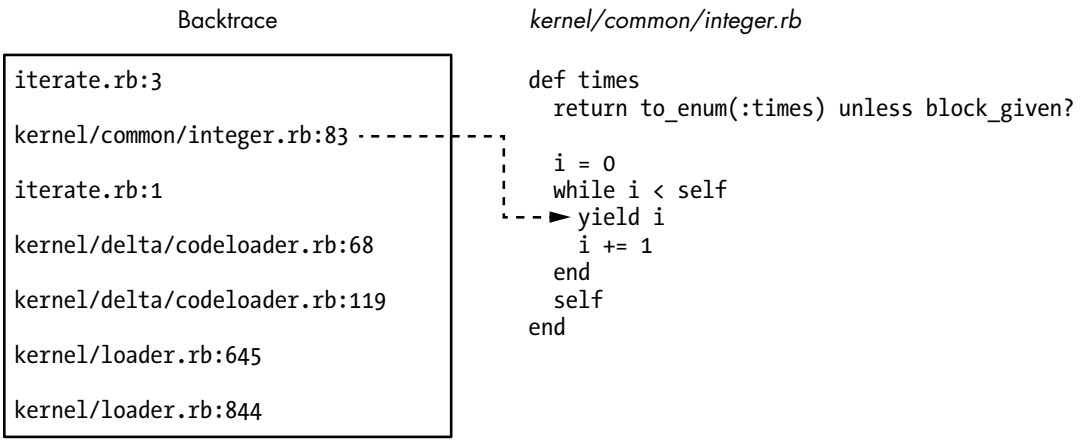

Figure 11-11: Rubinius includes information about its kernel in backtraces.

The backtrace information on the left of the figure is the same as that in Figure 11-10. The arrow points from the second level of the Ruby call stack to the code that calls the `puts n` block—the `yield` instruction in the `Integer#times` method.

Using Rubinius, *iterate.rb* becomes part of a larger Ruby program: the Rubinius kernel. When we call `10.times`, Rubinius calls the Ruby code shown at the right, which then executes our block using the `yield` keyword on line 83.

NOTE *The path* kernel/common/integer.rb *refers to a location in the Rubinius source code tree. If you installed Rubinius using a binary installer, you'll need to download the source code from* http://rubini.us/ *or GitHub in order to read it.*

Rubinius implements `Integer#times` by counting from 0 up to the specified integer (minus one), calling the block each time through the loop. Let's take a closer look at `Integer#times`, as shown in Listing 11-7.

```
❶ def times
❷   return to_enum(:times) unless block_given?

❸   i = 0
❹   while i < self
❺     yield i
      i += 1
    end
❻   self
end
```

Listing 11-7: The Rubinius implementation of `Integer#times`*, from* kernel/common/integer.rb

The definition of the `times` method starts at ❶. At ❷ Rubinius returns the result of `to_enum` if a block is not provided, as shown below. (The `to_enum` method returns a new enumerator object, which allows you to perform the enumeration later if you prefer.)

```
p 10.times
 => #<Enumerable::Enumerator:0x120 @generator=nil @args=[] @lookahead=[]
     @object=10 @iter=:times>
```

Rubinius continues to execute the rest of the method if you provide a block. At ❸ Rubinius creates a counter `i` and initializes it to 0. Next, it uses a while loop at ❹ to perform the iteration. Notice that the while loop condition `i < self` refers to the value of `self`. Inside `Integer#times`, `self` is set to the current integer object, or 10 in our script. At ❺ Rubinius yields to (calls) the given block, passing in the current value of `i`. This calls our `puts n` block. Finally, at ❻ Rubinius returns `self`, which means the return value of `10.times` will be 10.

Arrays in Rubinius and MRI

Arrays are so ubiquitous in Ruby that it's easy to take them for granted. But how do they work inside Ruby? Where does Ruby save objects that you place into an array, and how does it represent array objects internally? In the following sections, we'll look at the internal data structures that Rubinius and MRI use to hold values in an array.

Arrays Inside of MRI

Suppose you put the first six numbers from the Fibonacci sequence into an array.

```
fibonacci_sequence = [1, 1, 2, 3, 5, 8]
```

As Figure 11-12 illustrates, MRI creates a C structure for the array but saves its elements elsewhere.

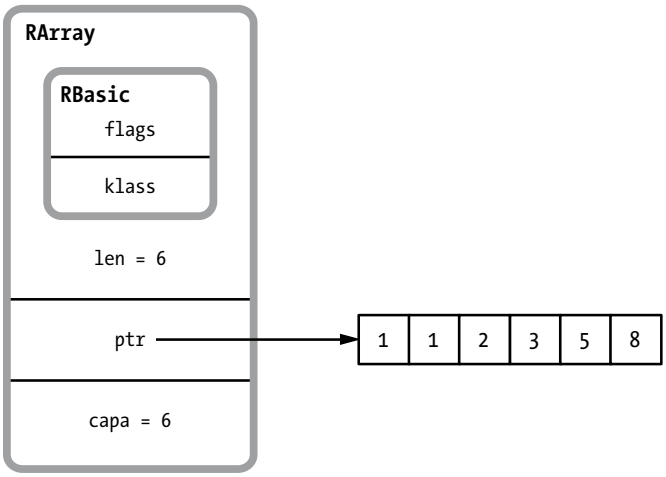

Figure 11-12: MRI uses the `RArray` C structure to represent arrays.

MRI uses one `RArray` structure to represent each array you create. Like `RString`, `RObject`, and other C structures, `RArray` uses the inner `RBasic` structure to hold the `klass` pointer and other technical information. (In this case, the `klass` pointer points to the `RClass` structure for the `Array` class.)

Below `RBasic` are a few additional values specific to arrays—`ptr`, `len`, and `capa`:

- `ptr` is a pointer to a memory segment Ruby allocates separately to store the array elements. The Fibonacci numbers appear in this memory segment at the right side of Figure 11-12.
- `len` is the length of the array—that is, the number of values saved in the separate memory segment.
- `capa` tracks the capacity of the memory segment. This number is often larger than `len`. MRI avoids continually resizing the memory segment each time you change the size of the array; instead, as you add array elements, it occasionally increases the size of the separate memory segment, each time allocating more memory than the new elements require.

Each value in the separate memory segment is actually a `VALUE` pointer to a Ruby object. In this case, the Fibonacci numbers would be saved directly inside the `VALUE` pointers because they are simple integers.

THE RARRAY C STRUCTURE DEFINITION

Listing 11-8 shows the definition of `RArray` from the MRI C source code.

```
#define RARRAY_EMBED_LEN_MAX 3
struct RArray {
  struct RBasic basic;
❶ union {
    struct {
❷     long len;
      union {
❸       long capa;
❹       VALUE shared;
      } aux;
❺     VALUE *ptr;
    } heap;
❻   VALUE ary[RARRAY_EMBED_LEN_MAX];
  } as;
};
```

Listing 11-8: The definition of `RArray` (from include/ruby/ruby.h*)*

This definition shows a few values that are missing from Figure 11-12. First, at ❶, notice that MRI uses a C `union` keyword to declare two alternative definitions for `RArray`. The first, an inner `struct`, defines `len` at ❷, `capa` at ❸, `shared` at ❹, and `ptr` at ❺. As with strings, MRI uses copy-on-write optimization with arrays, allowing two or more arrays to share the same underlying data. For arrays that share data, the `shared` value at ❹ refers to another `RArray` that contains the shared data.

The second half of the union at ❻ defines `ary`, a C array of `VALUE` pointers in `RArray`. This is an optimization that allows MRI to save the array data for arrays with three or fewer elements inside the `RArray` structure itself, avoiding the need to allocate the separate memory segment at all. MRI optimizes four other C structures in a similar way: `RString`, `RObject`, `RStruct` (used by the `Struct` class), and `RBignum` (used by the `Bignum` class).

Arrays Inside of Rubinius

Now let's see how Rubinius saves the same Fibonacci array internally. We learned earlier that Rubinius represents each Ruby object with a corresponding C++ object. This representation is true of arrays as well. For example, Figure 11-13 shows the C++ object that Rubinius would use to represent `fibonacci_sequence`.

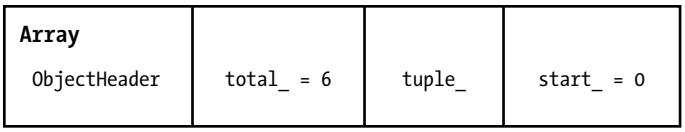

Figure 11-13: Rubinius uses C++ objects to represent arrays.

The combined four blocks represent an instance of the `Array` C++ class. Rubinius creates a C++ array object each time you create an array. From left to right, the fields are as follows:

- `ObjectHeader` contains technical information that Rubinius keeps track of inside each object, including a class pointer and an array of instance variables. `ObjectHeader` corresponds to the `RBasic` C structure in MRI and is one of the C++ superclasses of the `Array` C++ class inside the Rubinius virtual machine.
- `total_` is the length of the array, which is 6 for `fibonacci_sequence`.
- `tuple_` is a pointer to an instance of another C++ class, called `Tuple`, that contains the array data.
- `start_` indicates where the array data starts inside the tuple object. (The tuple may contain more data than your array needs.) Initially, Rubinius sets this to 0.

Rubinius doesn't save the array data in the C++ array object. It saves it in a tuple object, as shown in Figure 11-14.

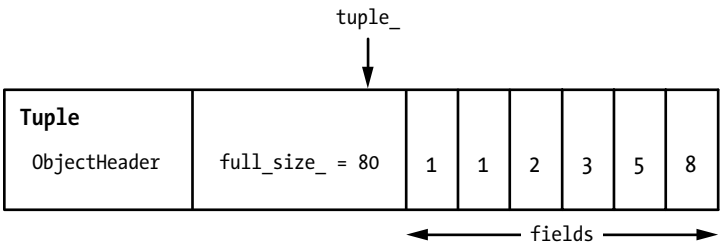

Figure 11-14: Rubinius saves array data in tuple objects.

Each tuple contains the same object header information as arrays. Rubinius saves this header information in every C++ object. Following the object header, tuple objects contain a value called `full_size_`, which keeps track of the size of this tuple object in bytes. Following this value, Rubinius saves the actual data values in a C++ array called `fields`. These data values are our six Fibonacci numbers, as shown at the right of Figure 11-14.

NOTE *Array data values are saved in the tuple C++ object. If we had created a larger array, Rubinius would have used a larger tuple object. If we change the size of an array, Rubinius allocates another tuple of the appropriate size or, as we'll see in Experiment 11-2, it can optimize certain array methods in order to avoid allocating new objects and speed up your program.*

Experiment 11-2: Exploring the Rubinius Implementation of Array#shift

We've seen that Rubinius uses C++ objects to represent arrays, but remember that Rubinius uses a combination of Ruby and C++ code to implement methods in the `Array` class. In this experiment, we'll learn more about how arrays work by looking at how Rubinius implements the `Array#shift` method.

But first a quick review of what `Array#shift` does. As you may know, calling `shift` removes one element from the beginning of an array and *shifts* the remaining elements to the left, as shown in Listing 11-9.

```
fibonacci_sequence = [1, 1, 2, 3, 5, 8]
p fibonacci_sequence.shift
❶ => 1
p fibonacci_sequence
❷ => [1, 2, 3, 5, 8]
```

Listing 11-9: `Array#shift` removes the first element from an array, shifting the remaining elements over.

At ❶ `Array#shift` returns the first element of `fibonacci_sequence`. We can see from the output at ❷ that `Array#shift` also removes the first element from the array, shifting the other five elements. But how does Ruby implement `Array#shift` internally? Does it actually copy the remaining array elements to the left, or does it copy them into a new array?

Reading Array#shift

First, let's find out where the `Array#shift` method is located inside Rubinius. Because we don't have a backtrace to refer to as in Experiment 11-1, we can ask Rubinius where to find the method using `source_location`.

```
p Array.instance_method(:shift).source_location
 => ["kernel/common/array.rb", 848]
```

This output tells us that Rubinius defines the `Array#shift` method at line 848 in the file *kernel/common/array.rb* in the Rubinius source tree. Listing 11-10 shows the Rubinius implementation of `Array#shift`.

```
❶ def shift(n=undefined)
    Rubinius.check_frozen

❷   if undefined.equal?(n)
      return nil if @total == 0
❸     obj = @tuple.at @start
      @tuple.put @start, nil
      @start += 1
      @total -= 1

      obj
```

```
❹   else
      n = Rubinius::Type.coerce_to(n, Fixnum, :to_int)
      raise ArgumentError, "negative array size" if n < 0

      Array.new slice!(0, n)
    end
  end
```

Listing 11-10: The implementation of `Array#shift` inside the Rubinius kernel

At ❶ `shift` takes an optional parameter `n`. If `shift` is called without a parameter `n`, as in Listing 11-9, it will remove the first element and shift the remaining elements by one position. If you provide a parameter `n` to `shift`, it will remove `n` elements and shift the remaining elements `n` positions to the left. At ❷ Rubinius checks whether the parameter `n` was supplied. If `n` was specified, it jumps to ❹ and uses `Array#slice!` to remove the first `n` elements and return them.

Modifying Array#shift

Now let's see what happens when you call `shift` with no parameters. How does Rubinius shift the array by one element? Unfortunately, the `Tuple#at` method called at ❸ is implemented by the C++ code inside the Rubinius virtual machine. (You won't find a definition for `at` in the Ruby *kernel/common/tuple.rb* file.) This means we won't be able to read the entire algorithm in Ruby.

We can, however, add Ruby code to Rubinius to display information about the array data when we call `shift`. Because the Rubinius kernel is written with Ruby, we can change it like any other Ruby program! First, we'll add a few lines of code to `Array#shift`, as shown in Listing 11-11.

```
if undefined.equal?(n)
  return nil if @total == 0

❶   fibonacci_array = (self == [1, 1, 2, 3, 5, 8])
❷   puts "Start: #{@start} Total: #{@total} Tuple: #{@tuple.inspect}" if
    fibonacci_array

    obj = @tuple.at @start
    @tuple.put @start, nil
    @start += 1
    @total -= 1

❸   puts "Start: #{@start} Total: #{@total} Tuple: #{@tuple.inspect}" if
    fibonacci_array

    obj
  end
```

Listing 11-11: Adding debug code to the Rubinius kernel

At ❶ we check whether this array is our Fibonacci array. Rubinius uses this method for every array in the system, but we want to display only information about our array. Then, at ❷ we display the values of `@start`, `@total`, and `@tuple`. Under the hood, `@tuple` is a C++ object, but in Rubinius it also functions as a Ruby object, allowing us to call its `inspect` method. At ❸ we display the same values once they've been changed by the `Array#shift` code.

Now we need to rebuild Rubinius to include our code changes. Listing 11-12 shows the output produced by the `rake install` command. (Run this at the root of the Rubinius source code tree.)

```
$ rake install

--snip--

RBC kernel/common/hash.rb
RBC kernel/common/hash19.rb
RBC kernel/common/hash_hamt.rb
❶ RBC kernel/common/array.rb
RBC kernel/common/array19.rb
RBC kernel/common/kernel.rb

--snip--
```

Listing 11-12: Rebuilding Rubinius

The Rubinius build process recompiled the *array.rb* source code file at ❶, along with many other kernel files. (`RBC` refers to the Rubinius compiler.)

NOTE *Don't try to use this sort of code change in a production environment.*

Now to rerun Listing 11-9 using our modified version of Rubinius. Listing 11-13 shows the output interspersed with our original code.

```
fibonacci_sequence = [1, 1, 2, 3, 5, 8]
p fibonacci_sequence.shift
❶ Start: 0 Total: 6 Tuple: #<Rubinius::Tuple: 1, 1, 2, 3, 5, 8>
❷ Start: 1 Total: 5 Tuple: #<Rubinius::Tuple: nil, 1, 2, 3, 5, 8>
 => 1
p fibonacci_sequence
 => [1, 2, 3, 5, 8]
```

Listing 11-13: Using our modified version of `Array#shift`

At ❶ and ❷ our new Ruby code inside `Array#shift` displays the internal contents of `fibonacci_sequence`: the `@start`, `@total`, and `@tuple` instance variables. Comparing ❶ with ❷, we can see how `Array#shift` works internally.

Rubinius hasn't allocated a new array object; it's reused the underlying tuple object. Rubinius has done the following:

- Changed `@total` from 6 to 5 because the length of the array has decreased by 1
- Changed `@start` from 0 to 1, which allowed it to continue to use the same value for `@tuple`; now the array contents start at the second value (index 1) in `@tuple`, not the first (index 0)
- Changed the first value in `@tuple` from 1 to `nil` because the first value is no longer used by the array

Creating new objects and allocating new memory can take a long time because Rubinius might have to ask for memory from the operating system. Its reuse of the underlying data in the tuple object, without copying or allocating memory for a new array, allows Rubinius to run faster.

Figures 11-15 and 11-16 summarize how `Array#shift` works. Figure 11-15 shows the array before calling `Array#shift`: `@start` pointed to the first value in the tuple, and `@length` was 6.

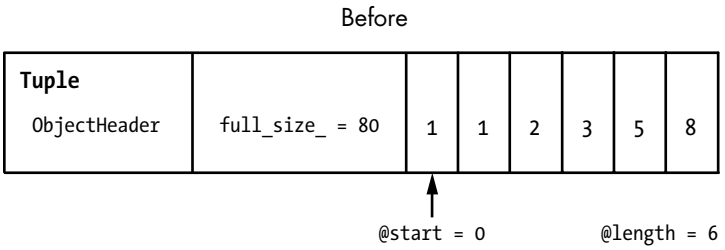

Figure 11-15: The tuple holding our Fibonacci numbers before calling `Array#shift`

Figure 11-16 shows the tuple after calling `Array#shift`; Rubinius has simply changed the values of `@start` and `@length` and set the first value in the tuple to `nil`.

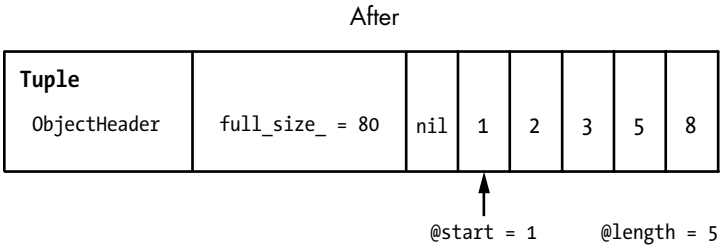

Figure 11-16: The same tuple after calling `Array#shift`

As you might guess, MRI uses a similar optimization for `Array#shift` by keeping track of where the array data starts in the original array. However, the C code it uses is more complex and difficult to understand. The Rubinius kernel gives us a much clearer view of this algorithm.

Summary

We've learned in this chapter that Rubinius uses a virtual machine implemented with C++ to run your Ruby code. Like YARV, the Rubinius virtual machine was custom designed to run Ruby programs, and it uses a compiler to convert your Ruby program into bytecode internally. We saw that these Rubinius instructions resemble YARV instructions; they operate on stack values in a similar way.

But what sets Rubinius apart from other Ruby implementations is its Ruby language kernel. The Rubinius kernel implements many built-in Ruby classes, such as `Array`, using Ruby code. This innovative design provides a window into Ruby internals—you can use Rubinius to learn how Ruby works internally without having to know C or Java. You can learn how Ruby implements strings, arrays, or other classes simply by reading the Ruby source code in the Rubinius kernel. Rubinius isn't just a Ruby implementation; it's a valuable learning resource for the Ruby community.