5

OBJECTS AND CLASSES

We learn early on that Ruby is an object-oriented language, descended from languages like Smalltalk and Simula. Every value is an object, and all Ruby programs consist of a set of objects and the messages sent between them. Typically, we learn about object-oriented programming by looking at how to use objects and what they can do: how they can group together data values and behavior related to those values; how each class should have a single responsibility or purpose; and how different classes can be related to each other through encapsulation or inheritance.

But what are Ruby objects? What information does an object contain? If we were to look at a Ruby object through a microscope, what would we see? Are there any moving parts inside? And what about Ruby classes? What exactly is a class?

I'll answer these questions in this chapter by exploring how Ruby works internally. By looking at how Ruby implements objects and classes, you'll learn how to use them and how to write object-oriented programs using Ruby.

ROADMAP

Inside a Ruby Object

Ruby saves each of your custom objects in a C structure called RObject, which looks like Figure 5-1 in Ruby 1.9 and 2.0.

At the top of the figure is a pointer to the RObject structure. (Internally, Ruby always refers to any value with a VALUE pointer.) Below this pointer, the RObject structure contains an inner RBasic structure and information specific to custom objects. The RBasic section contains information that all objects use: a set of Boolean values called flags that store a variety of internal technical values, and a class pointer called klass. The class pointer indicates

If I could slice open a Ruby object, what would I see?

which class an object is an instance of. In the RObject section, Ruby saves an array of instance variables that each object contains, using numiv, the instance variable count, and ivptr, a pointer to an array of values.

If we were to define the Ruby object structure in technical terms, we could say

> Every Ruby object is the combination of a class pointer and an array of instance variables.

At first glance, this definition doesn't seem very useful because it doesn't help us understand the meaning or purpose behind objects or how to use them in a Ruby program.

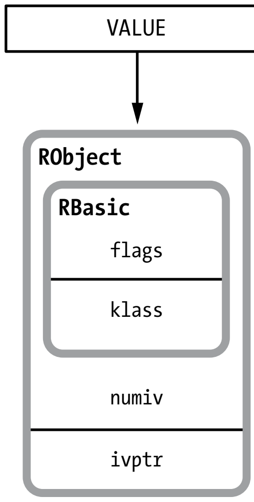


Figure 5-1: The RObject structure

Inspecting klass and ivptr

To understand how Ruby uses RObject in programs, we'll create a simple Ruby class and then inspect an instance of this class using IRB. For example, suppose I have the simple Ruby class shown in Listing 5-1.

```
class Mathematician
  attr_accessor :first_name
  attr_accessor :last_name
end
```

Listing 5-1: A simple Ruby class

Ruby needs to save the class pointer in RObject because every object must track the class you used to create it. When you create an instance of a class, Ruby internally saves a pointer to that class inside RObject, as shown in Listing 5-2.

```
$ irb
> euler = Mathematician.new
❶ => #<Mathematician:0x007fbd738608c0>
```

Listing 5-2: Creating an object instance in IRB

By displaying the class name #<Mathematician at ❶, Ruby displays the value of the class pointer for the euler object. The hex string that follows is actually the VALUE pointer for the object. (This will differ for every instance of Mathematician.)

Ruby also uses the instance variable array to track the values you save in an object, as shown in Listing 5-3.

```
> euler.first_name = 'Leonhard'
 => "Leonhard"
> euler.last_name  = 'Euler'
 => "Euler"
> euler
❶ => #<Mathematician:0x007fbd738608c0 @first_name="Leonhard", @last_name="Euler">
```

Listing 5-3: Inspecting instance variables in IRB

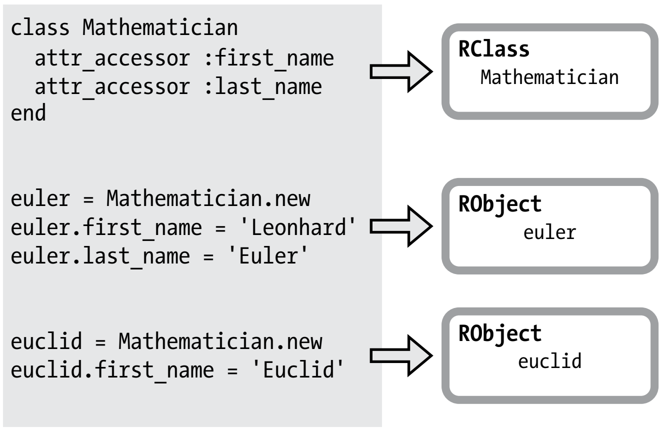
As you can see, in IRB Ruby also displays the instance variable array for `euler` at ❶. Ruby needs to save this array of values in each object because every object instance can have different values for the same instance variables, as shown at ❶ in Listing 5-4.

```
> euclid = Mathematician.new
> euclid.first_name = 'Euclid'
> euclid
❶ => #<Mathematician:0x007fabdb850690 @first_name="Euclid">
```

Listing 5-4: A different instance of the `Mathematician` class

Visualizing Two Instances of One Class

Let's look at Ruby's C structures in a bit more detail. When you run the Ruby code shown in Figure 5-2, Ruby creates one `RClass` structure and two `RObject` structures.



```
class Mathematician
  attr_accessor :first_name
  attr_accessor :last_name
end

euler = Mathematician.new
euler.first_name = 'Leonhard'
euler.last_name = 'Euler'

euclid = Mathematician.new
euclid.first_name = 'Euclid'
```

Figure 5-2: Creating two instances of one class

I'll discuss how Ruby implements classes with the `RClass` structure in the next section. For now, let's look at Figure 5-3, which shows how Ruby saves the `Mathematician` information in the two `RObject` structures.

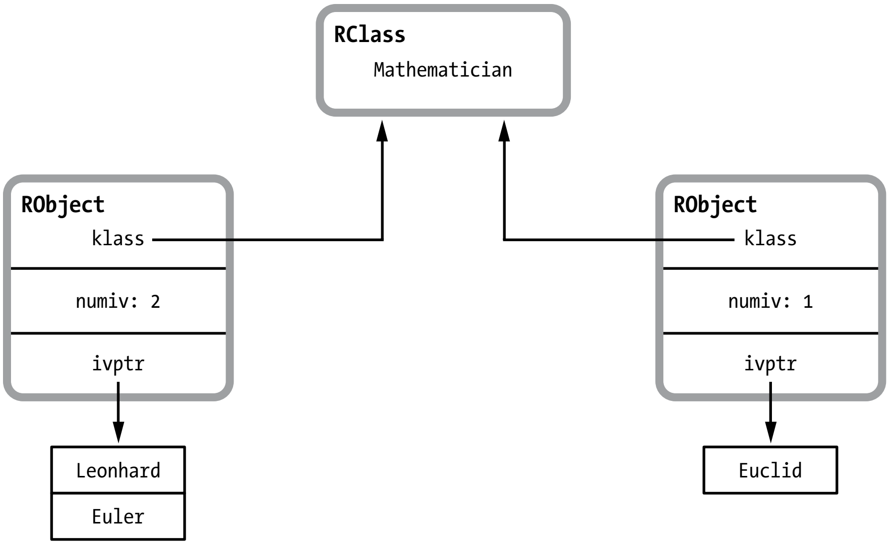


Figure 5-3: Visualizing two instances of one class

As you can see, each `klass` value points to the `Mathematician` `RClass` structure, and each `RObject` structure has a separate array of instance variables. Both arrays contain `VALUE` pointers—the same pointer that Ruby uses to refer to the `RObject` structure. (Notice that one of the objects contains two instance variables, while the other contains only one.)

Generic Objects

Now you know how Ruby saves custom classes, like the `Mathematician` class, in `RObject` structures. But remember that every Ruby value—including basic data types such as integers, strings, and symbols—is an object. The Ruby source code internally refers to these built-in types as "generic" types. How does Ruby store these generic objects? Do they also use the `RObject` structure?

The answer is no. Internally, Ruby uses a different C structure, not `RObject`, to save values for each of its generic data types. For example, Ruby saves string values in `RString` structures, arrays in `RArray` structures, regular expressions in `RRegexp` structures, and so on. Ruby uses `RObject` only to save instances of custom object classes that you create and a few custom object classes that Ruby creates internally. However, all of these different structures share the same `RBasic` information that we saw in `RObject`, as shown in Figure 5-4.

Since the `RBasic` structure contains the class pointer, each of these generic data types is also an object. Each is an instance of some Ruby class, as indicated by the class pointer saved inside `RBasic`.

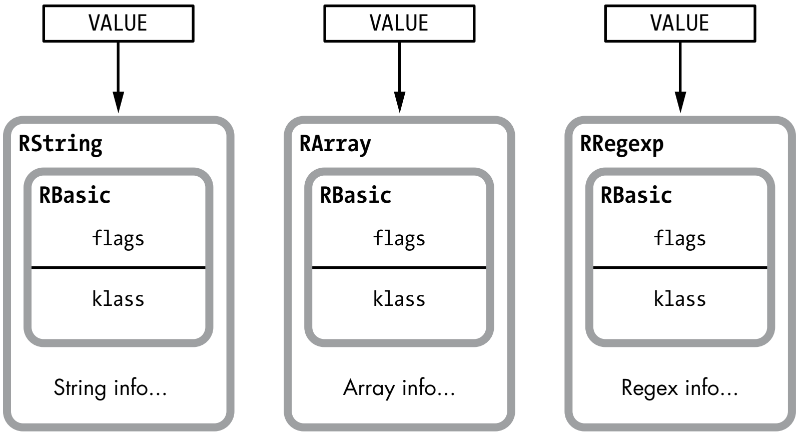


Figure 5-4: Different Ruby object structures all use the `RBasic` *structure.*

Simple Ruby Values Don't Require a Structure at All

As a performance optimization, Ruby saves small integers, symbols, and a few other simple values without any structure at all, placing them right inside the `VALUE` pointer, as shown in Figure 5-5.

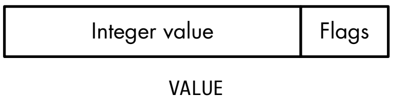


Figure 5-5: Ruby saves integers in the `VALUE` *pointer.*

These `VALUE`s are not pointers at all; they're values themselves. For these simple data types, there is no class pointer. Instead, Ruby remembers the class using a series of bit flags saved in the first few bits of the `VALUE`. For example, all small integers have the `FIXNUM_FLAG` bit set, as shown in Figure 5-6.

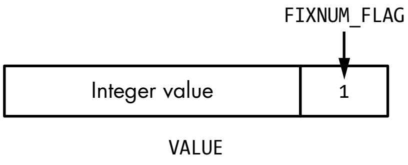


Figure 5-6: `FIXNUM_FLAG` *indicates this is an instance of the* `Fixnum` *class.*

Whenever the `FIXNUM_FLAG` is set, Ruby knows that this `VALUE` is really a small integer, an instance of the `Fixnum` class, and not a pointer to a value structure. (A similar bit flag indicates whether the `VALUE` is a symbol, and values such as `nil`, `true`, and `false` also have their own flags.)

It's easy to see that integers, strings, and other generic values are all objects by using IRB, as you can see in Listing 5-5.

```
$ irb
> "string".class
 => String
> 1.class
 => Fixnum
> :symbol.class
 => Symbol
```

Listing 5-5: Inspecting classes for some generic values

Here, we see that Ruby saves a class pointer or the equivalent bit flag for all values by calling the `class` method on each. In turn, the `class` method returns the class pointer, or at least the name of the class that each `klass` pointer refers to.

Do Generic Objects Have Instance Variables?

Let's go back to our definition of a Ruby object:

> Every Ruby object is the combination of a class pointer and an array of instance variables.

What about instance variables for generic objects? Do integers, strings, and other generic data values have instance variables? That would seem odd, but if integers and strings are objects, this must be true! And if it is true, where does Ruby save these values if it doesn't use the `RObject` structure?

Using the `instance_variables` method, shown in Listing 5-6, you can see that each of these basic values can also contain an array of instance variables, strange as that may seem.

```
$ irb
> str = "some string value"
 => "some string value"
> str.instance_variables
 => []
> str.instance_variable_set("@val1", "value one")
 => "value one"
> str.instance_variables
 => [:@val1]
> str.instance_variable_set("@val2", "value two")
 => "value two"
> str.instance_variables
 => [:@val1, :@val2]
```

Listing 5-6: Saving instance variables in a Ruby string object

Repeat this exercise using symbols, arrays, or any Ruby value, and you'll find that every Ruby value is an object and every object contains a class pointer and an array of instance variables.

READING THE RBASIC AND ROBJECT C STRUCTURE DEFINITIONS

Listing 5-7 shows the definitions of the `RBasic` and `RObject` C structures. (You can find this code in the *include/ruby/ruby.h* header file.)

```
   struct RBasic {
❶    VALUE flags;
❷    const VALUE klass;
   };

   #define ROBJECT_EMBED_LEN_MAX 3
   struct RObject {
❸    struct RBasic basic;
     union {
       struct {
❹        long numiv;
❺        VALUE *ivptr;
❻        struct st_table *iv_index_tbl;
❼      } heap;
❽      VALUE ary[ROBJECT_EMBED_LEN_MAX];
     } as;
   };
```

Listing 5-7: The definitions of the `RBasic` and `RObject` C structures

At the top, you see the definition of `RBasic`. This definition contains the two values: `flags` ❶ and `klass` ❷. Below, you see the `RObject` definition. Notice that it contains a copy of the `RBasic` structure at ❸. Following this, the `union` keyword contains a structure called `heap` at ❼, followed by an array called `ary` at ❽.

The `heap` structure at ❼ contains the following values:

- First, the value `numiv` at ❹ tracks the number of instance variables contained in this object.
- Next, `ivptr` at ❺ is a pointer to an array containing the values of this object's instance variables. Notice that the names, or IDs, of the instance variables are not stored here; only the values are stored.
- `iv_index_tbl` at ❻ points to a hash table that maps between the name, or ID, of each instance variable and its location in the `ivptr` array. This value is actually stored in the `RClass` structure for this object's class; this pointer is simply a cache, or shortcut, that Ruby uses to obtain that hash table quickly. (The `st_table` type refers to Ruby's implementation of hash tables, which I'll discuss in Chapter 7.)

The last member of the `RObject` structure, `ary` at ❽, occupies the same memory space as all previous values because of the `union` keyword at the top. Using this `ary` value, Ruby can save all of the instance variables right inside the `RObject` structure—if they'll fit. This eliminates the need to call `malloc` to allocate extra memory to hold the instance variable value array. (Ruby also uses this sort of optimization for the `RString`, `RArray`, `RStruct`, and `RBignum` structures.)

Where Does Ruby Save Instance Variables for Generic Objects?

Internally, Ruby uses a bit of a hack to save instance variables for generic objects—that is, for objects that don't use an `RObject` structure. When you save an instance variable in a generic object, Ruby saves it in a special hash called `generic_iv_tbl`. This hash maintains a map between generic objects and pointers to other hashes that contain each object's instance variables. Figure 5-7 shows how this would look for the `str` string example in Listing 5-6.

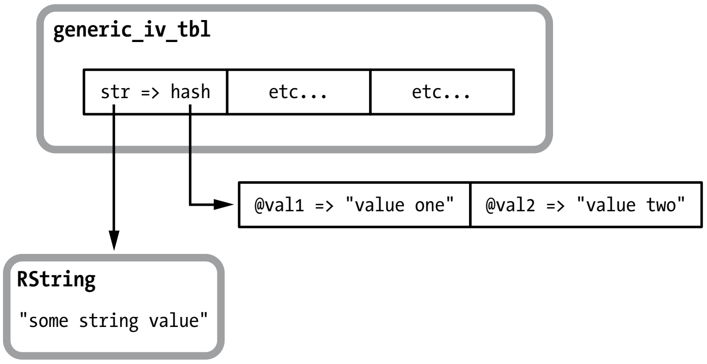


Figure 5-7: `generic_iv_tbl` stores instance variables for generic objects.

Experiment 5-1: How Long Does It Take to Save a New Instance Variable?

To learn more about how Ruby saves instance variables internally, let's measure how long it takes Ruby to save one in an object. To do this, I'll create a large number of test objects, as shown in Listing 5-8.

```
  ITERATIONS = 100000
❶ GC.disable
❷ obj = ITERATIONS.times.map { Class.new.new }
```

Listing 5-8: Creating test objects using `Class.new`

Here, I'm using `Class.new` at ❷ to create a unique class for each new object in order to make sure they're all independent. I've also disabled garbage collection at ❶ to avoid skewing the results with GC operations. Then, in Listing 5-9, I add instance variables to each.

```
Benchmark.bm do |bench|
  20.times do |count|
    bench.report("adding instance variable number #{count+1}") do
      ITERATIONS.times do |n|
        obj[n].instance_variable_set("@var#{count}", "value")
      end
    end
  end
end
```

Listing 5-9: Adding instance variables to each test object

Listing 5-9 iterates 20 times, repeatedly saving one more new instance variable to each of the objects. Figure 5-8 shows the time that it takes Ruby 2.0 to add each variable: The first bar on the left is the time it takes to save the first instance variable in all the objects, and each subsequent bar is the additional time taken to save one more instance variable in each object.

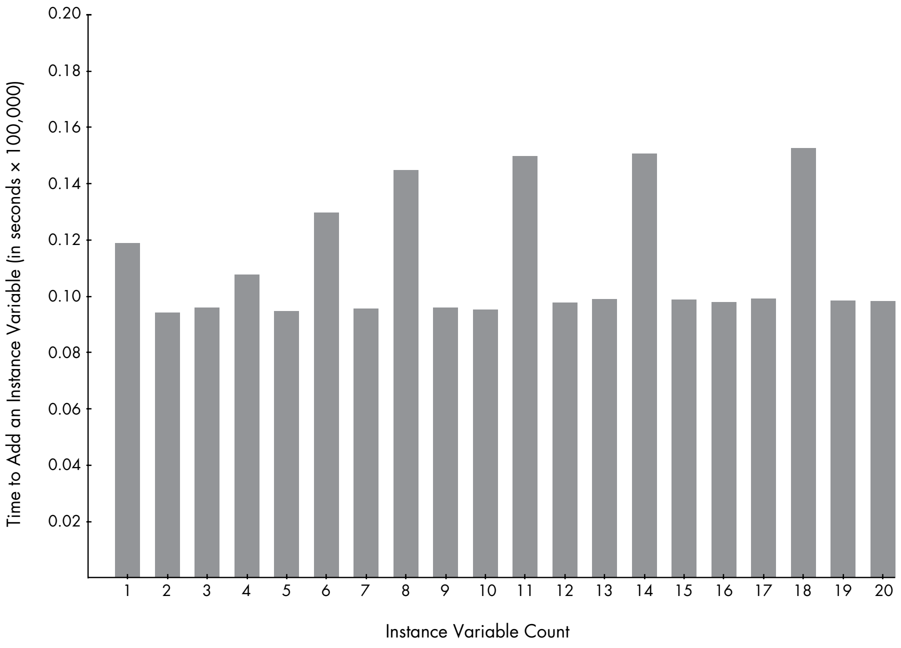


Figure 5-8: Time to add one more instance variable (in seconds x 100,000) vs. instance variable count

Figure 5-8 shows a strange pattern. Sometimes it takes Ruby longer to add a new instance variable. What's going on here?

The reason for this behavior has to do with the `ivptr` array where Ruby stores the instance variables, as shown in Figure 5-9.

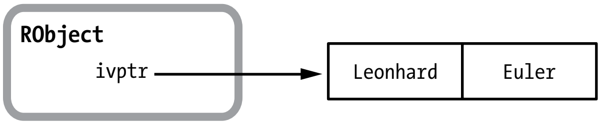


Figure 5-9: Two instance variables saved in an object

In Ruby 1.8 this array is a hash table containing both the variable names (the hash keys) and the values, which will automatically expand to accommodate any number of elements.

Ruby 1.9 and later save memory by storing the values in a simple array. The instance variable names are saved in the object's class instead, because they're the same for all instances of a class. As a result, Ruby 1.9 and 2.0 need to either preallocate a large array to handle any number of instance variables or repeatedly increase the size of this array as you save more variables.

In fact, as you can see in Figure 5-8, Ruby 1.9 and 2.0 repeatedly increase the array size. For example, suppose you have seven instance variables in a given object, as shown in Figure 5-10.

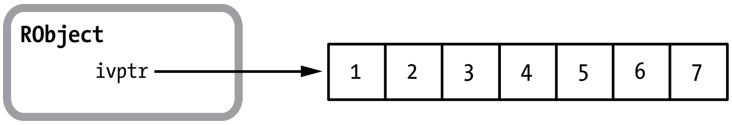


Figure 5-10: Seven instance variables in an object

When you add the eighth variable—bar 8 in Figure 5-8—Ruby 1.9 and 2.0 increase the array size by three, anticipating that you will soon add more variables, as shown in Figure 5-11.

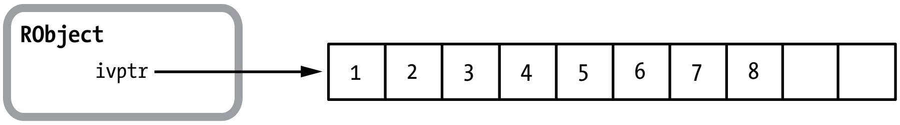


Figure 5-11: Adding an eighth value allocates extra space.

Allocating more memory takes extra time, which is why bar 8 is higher. Now if you add two more instance variables, Ruby 1.9 and 2.0 won't need to reallocate memory for this array because the space will already be available. This explains the shorter times for bars 9 and 10.

What's Inside the RClass Structure?

Every object remembers its class by saving a pointer to an `RClass` structure. What information does each `RClass` structure contain? What would we see if we could look inside a Ruby class? Let's build a model of the information that must be present in `RClass`. This model will give us a technical definition of what a Ruby class is, based on what we know classes can do.

Two objects, one class

Every Ruby developer knows how to write a class: You type the `class` keyword, specify a name for the new class, and then type in the class's methods. Listing 5-10 shows a familiar example.

```
class Mathematician
  attr_accessor :first_name
  attr_accessor :last_name
end
```

Listing 5-10: The same simple Ruby class we saw in Listing 5-1

`attr_accessor` is shorthand for defining get and set methods for an attribute. (The methods defined by `attr_accessor` also check for `nil` values). Listing 5-11 shows a more verbose way of defining the same `Mathematician` class.

```
class Mathematician
  def first_name
    @first_name
  end
  def first_name=(value)
    @first_name = value
  end
  def last_name
    @last_name
  end
  def last_name=(value)
    @last_name = value
  end
end
```

Listing 5-11: The same class written without `attr_accessor`

It appears that this class—and every Ruby class—is just a group of method definitions. You can assign behavior to an object by adding methods to its class, and when you call a method on an object, Ruby looks for the method in the object's class. This leads to our first definition of a Ruby class:

> A Ruby class is a group of method definitions.

Therefore, the `RClass` structure for `Mathematician` must save a list of all the methods defined in the class, as shown in Figure 5-12.

Notice in Listing 5-11 that I've also created two instance variables: `@first_name` and `@last_name`. We saw earlier how Ruby stores these values in each `RObject` structure, but you may have noticed that only the *values* of these variables are stored in `RObject`, not their names. (Ruby 1.8 does store the names in `RObject`.) Ruby must store the attribute names in `RClass`, which makes sense because the names will be the same for every `Mathematician` instance.

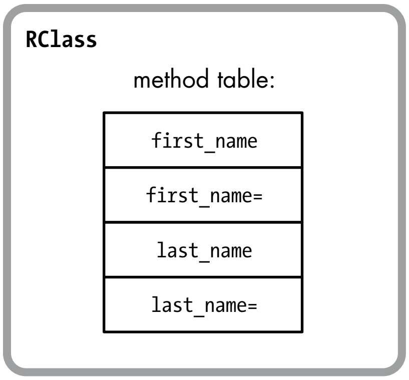


Figure 5-12: Ruby classes contain a method table.

Let's redraw `RClass` again and include a table of attribute names this time, as shown in Figure 5-13.

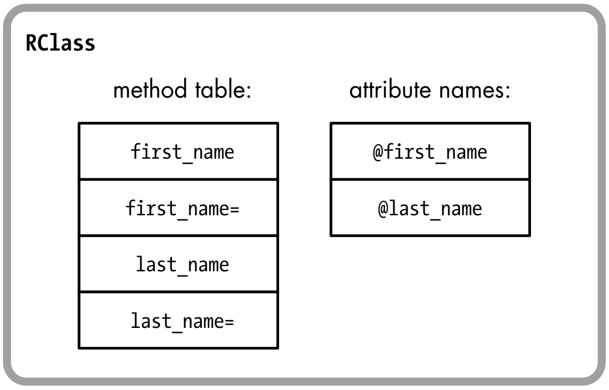


Figure 5-13: Ruby classes also contain a table of attribute names.

Now our definition of a Ruby class is as follows:

> A Ruby class is a group of method definitions and a table of attribute names.

At the beginning of this chapter, I mentioned that every value in Ruby is an object. This might be true for classes, too. Let's prove this using IRB.

```
> p Mathematician.class
 => Class
```

As you can see, Ruby classes are all instances of the `Class` class; therefore, classes are also objects. Now to update our definition of a Ruby class again:

> A Ruby class is a Ruby object that also contains method definitions and attribute names.

Because Ruby classes are objects, we know that the RClass structure must also contain a class pointer and an instance variable array, the values that we know every Ruby object contains, as shown in Figure 5-14.

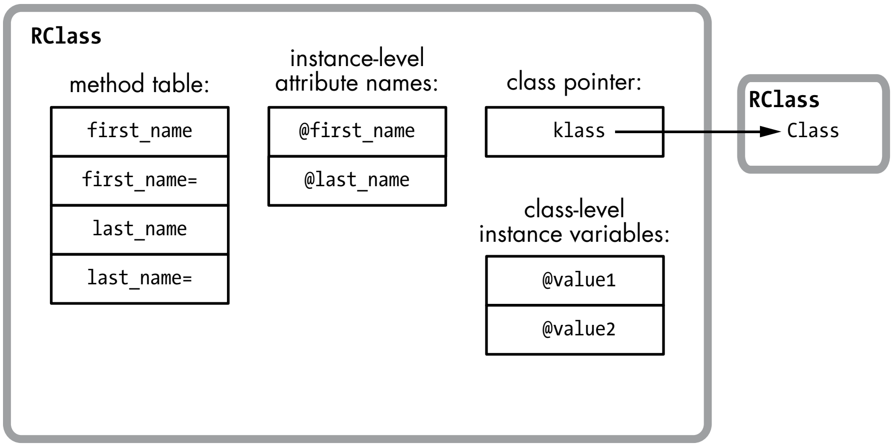


Figure 5-14: Ruby classes also contain a class pointer and instance variables.

As you can see, I've added a pointer to the Class class, which is in theory the class of every Ruby class object. However, in Experiment 5-2 on page 127, I'll show that this diagram is in fact not accurate—that klass actually points to something else! I've also added a table of instance variables.

NOTE *These are the class-level instance variables. Don't confuse these with the table of attribute names for the object-level instance variables.*

This is rapidly getting out of control! The RClass structure seems to be much more complex than the RObject structure. But don't worry—we're getting close to an accurate picture of the RClass structure. Next we need to consider two more important types of information contained in each Ruby class.

Inheritance

Inheritance is an essential feature of object-oriented programming. Ruby implements single inheritance by allowing us to optionally specify one superclass when we create a class. If we don't specify a superclass, Ruby assigns the Object class as the superclass. For example, we could rewrite the Mathematician class using a superclass like this:

```
class Mathematician < Person
--snip--
```

Now every instance of Mathematician will include the same methods that instances of Person have. In this example, we might want to move the first_name and last_name accessor methods into Person. We could also move the @first_name and @last_name attributes into the Person class. Every instance of Mathematician will contain these methods and attributes, even though we moved them to the Person class.

The Mathematician class must contain a reference to the Person class (its superclass) so that Ruby can find any methods or attributes defined in the superclass.

Let's update our definition again, assuming that Ruby tracks the superclass using another pointer similar to klass:

> A Ruby class is a Ruby object that also contains method definitions, attribute names, and a superclass pointer.

And let's redraw the RClass structure to include the new superclass pointer, as shown in Figure 5-15.

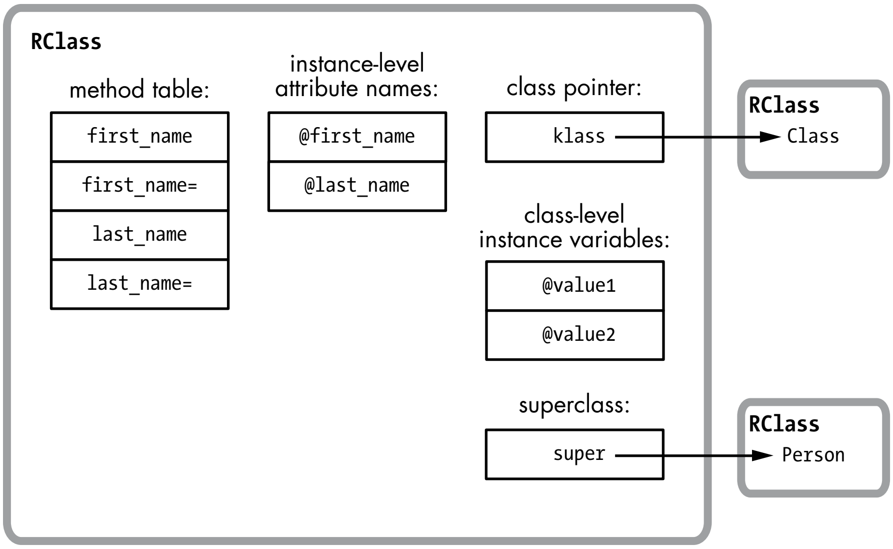


Figure 5-15: Ruby classes also contain a superclass pointer.

At this point, it is critical to understand the difference between the klass pointer and the super pointer. The klass pointer indicates which class the Ruby class object is an instance of. This will always be the Class class:

```
> p Mathematician.class
 => Class
```

Ruby uses the klass pointer to find the methods of the Mathematician class object, such as the new method that every Ruby class implements. However, the super pointer records the class's superclass:

```
> p Mathematician.superclass
 => Person
```

Ruby uses the super pointer to help find methods contained in each Mathematician instance, such as first_name= or last_name. As we'll see next, Ruby also uses the super pointer when getting or setting class variables.

Class Instance Variables vs. Class Variables

One confusing bit of Ruby syntax is the concept of *class variables*. You might think that these are simply the instance variables of a class (the class-level instance variables from Figure 5-14), but class instance variables and class variables are distinctly different.

To create a class instance variable, you simply create an instance variable using the @ symbol, but in the context of a class rather than an object. For example, Listing 5-12 shows how we could use an instance variable of Mathematician to indicate a branch of mathematics this class corresponds to. We create the @type instance variable at ❶.

```
  class Mathematician
❶   @type = "General"
    def self.type
      @type
    end
  end

  puts Mathematician.type
   => General
```

Listing 5-12: Creating a class-level instance variable

In contrast, to create a class variable, you would use the @@ notation. Listing 5-13 shows the same example, with the class variable @@type ❶ created.

```
  class Mathematician
❶   @@type = "General"
    def self.type
      @@type
    end
  end

  puts Mathematician.type
   => General
```

Listing 5-13: Creating a class variable

What's the difference? When you create a class variable, Ruby creates a single value for you to use in that class and in any subclasses you might define. On the other hand, using a class *instance* variable causes Ruby to create a separate value for each class or subclass.

Let's review Listing 5-14 to see how Ruby handles these two types of variables differently. First, I define a class instance variable called `@type` in the `Mathematician` class and set its value to the string `General`. Next, I create a second class called `Statistician`, which is a subclass of `Mathematician`, and change the value of `@type` to the string `Statistics`.

```
class Mathematician
  @type = "General"
  def self.type
    @type
  end
end

class Statistician < Mathematician
  @type = "Statistics"
end

puts Statistician.type
❶ => Statistics
puts Mathematician.type
❷ => General
```

Listing 5-14: Each class and subclass has its own instance variables.

Notice that the values of `@type` in `Statistician` at ❶ and `Mathematician` at ❷ are different. Each class has its own separate copy of `@type`.

However, if I use a class variable instead, Ruby shares that value between `Mathematician` and `Statistician`, as demonstrated in Listing 5-15.

```
class Mathematician
  @@type = "General"
  def self.type
    @@type
  end
end

class Statistician < Mathematician
  @@type = "Statistics"
end

puts Statistician.type
❶ => Statistics
puts Mathematician.type
❷ => Statistics
```

Listing 5-15: Ruby shares class variables among a class and all of its subclasses.

Here, Ruby shows the same value for `@@type` in `Statistician` at ❶ and in `Mathematician` at ❷.

Internally, however, Ruby actually saves both class variables and class instance variables in the same table inside the `RClass` structure. Figure 5-16 shows how the `Mathematician` class would save the `@type` and `@@type` values if you created both of them. The extra `@` symbol in the name allows Ruby to distinguish between the two types of variables.

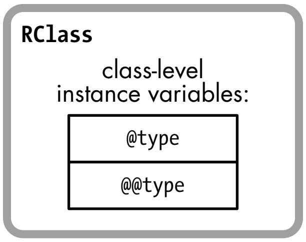
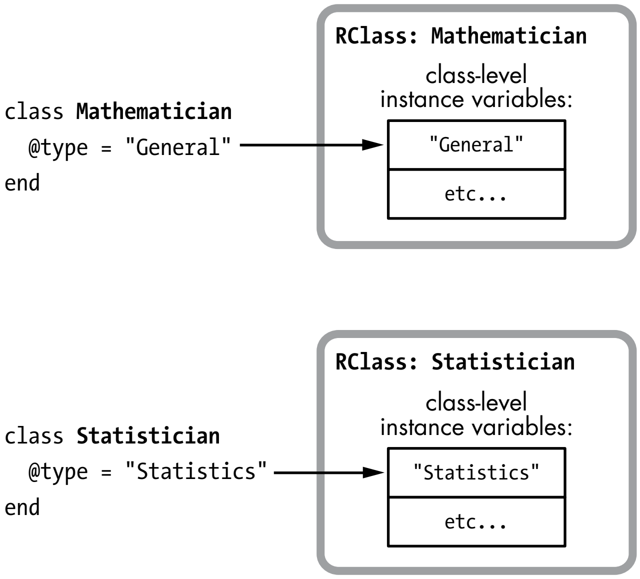


Figure 5-16: Ruby saves class variables and class instance variables in the same table.

Getting and Setting Class Variables

It's true: Ruby saves both class variables and class instance variables in the same table. However, the ways Ruby gets or sets these two types of variables are quite different.

When you get or set a class instance variable, Ruby looks up the variable in the `RClass` structure corresponding to the target class and either saves or retrieves the value. Figure 5-17 shows how Ruby saves the class instance variables from Listing 5-14.



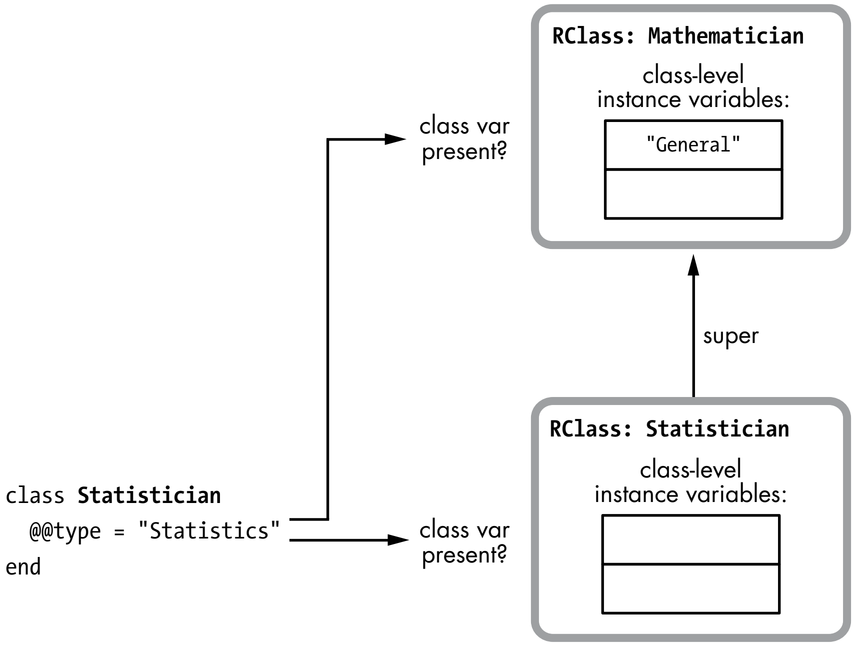
```
class Mathematician
  @type = "General"
end
```

```
class Statistician
  @type = "Statistics"
end
```

Figure 5-17: Ruby saves class instance variables in the `RClass` structure of the target class.

At the top of the figure, you can see a line of code that saves a class instance variable in `Mathematician`. Below that is a similar line of code that saves a value in `Statistician`. In both cases, Ruby saves the class instance variable in the `RClass` structure for the current class.

Ruby uses a more complex algorithm for class variables. To produce the behavior we saw in Listing 5-15, Ruby needs to search through all the superclasses to see whether any of them define the same class variable. Figure 5-18 shows an example.



```
class Statistician
  @@type = "Statistics"
end
```

Figure 5-18: Before saving it, Ruby checks whether the class variable exists in the target class or any of its superclasses.

When you save a class variable, Ruby looks in the target class and all of its superclasses for an existing variable. It will find `@@type` in the highest superclass. In Figure 5-18 you can see Ruby checks both the `Statistician` and `Mathematician` classes when saving the `@@type` class variable in `Statistician`. Because I already saved the same class variable in `Mathematician` (Listing 5-15), Ruby will use that and overwrite it with the new value, as shown in Figure 5-19.

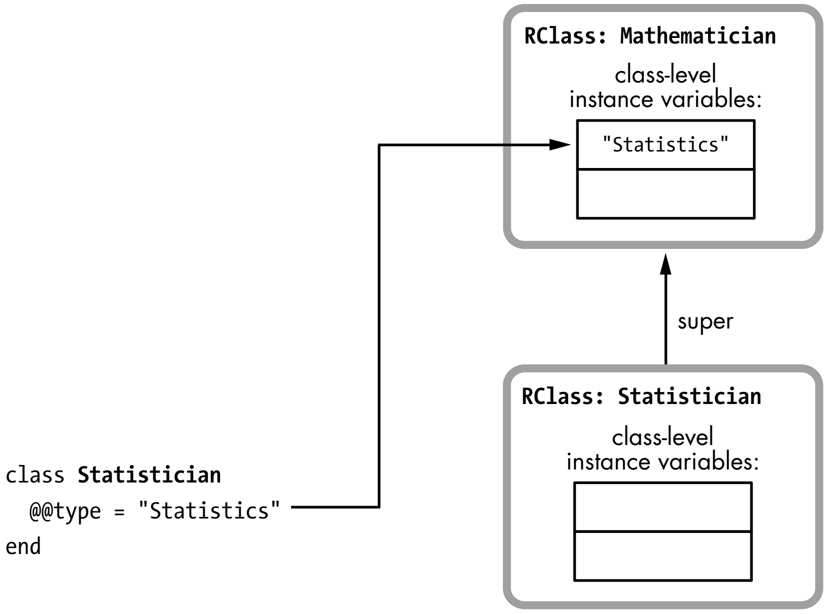


Figure 5-19: Ruby uses the class variable copy found in the highest superclass.

Constants

We have one more feature of Ruby classes to cover: *constants*. As you may know, Ruby allows you to define constant values inside a class, like this:

```
class Mathematician < Person
  AREA_OF_EXPERTISE = "Mathematics"
  --snip--
```

Constant values must start with a capital letter, and they are valid within the scope of the current class. (Curiously, Ruby allows you to change a constant value, but it will display a warning when you do so.) Let's add a constant table to our `RClass` structure, because Ruby must save these values inside each class, as shown in Figure 5-20.

Now we can write a complete, technical definition of a Ruby class:

> A Ruby class is a Ruby object that also contains method definitions, attribute names, a superclass pointer, and a constants table.

Granted, this isn't as concise as the simple definition we had for a Ruby object, but each Ruby class contains much more information than each Ruby object. Ruby classes are obviously fundamental to the language.

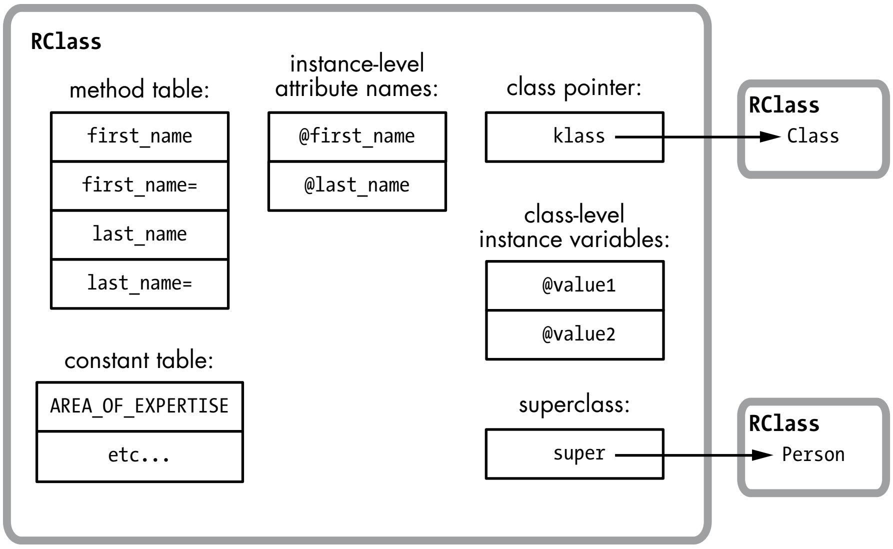


Figure 5-20: Ruby classes also contain a constants table.

The Actual RClass Structure

Having built up a conceptual model for what information must be stored in RClass, let's look at the actual structure that Ruby uses to represent classes, as shown in Figure 5-21.

As you can see, Ruby uses two separate structures to represent each class: RClass and rb_classext_struct. But these two structures act as one large structure because each RClass always contains a pointer (ptr) to a corresponding rb_classext_struct. You might guess that the Ruby core team decided to use two different structures because there are so many different values to save, but in fact they probably created rb_classext_struct to save internal values that they didn't want to expose in the public Ruby C extension API.

Like RObject, RClass has a VALUE pointer (shown on the left of Figure 5-21). Ruby always accesses classes using these VALUE pointers. The right side of the figure shows the technical names for the fields:

- flags and klass are the same RBasic values that every Ruby value contains.
- m_tbl is the method table, a hash whose keys are the names, or IDs, of each method and whose values are pointers to the definition of each method, including the compiled YARV instructions.
- iv_index_tbl is the attribute names table, a hash that maps each instance variable name to the index of the attribute's value in each RObject instance variable array.
- super is a pointer to the RClass structure for this class's superclass.

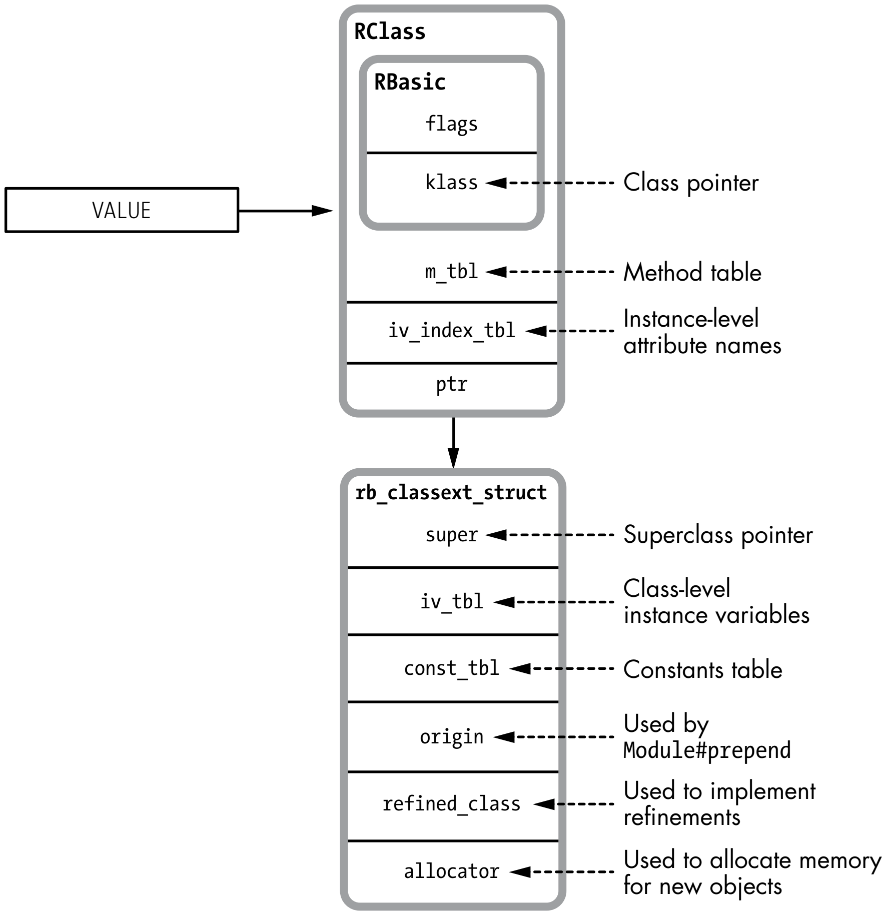


Figure 5-21: How Ruby actually represents a class

- `iv_tbl` contains the class-level instance variables and class variables, including both their names and values.
- `const_tbl` is a hash containing all of the constants (names and values) defined in this class's scope. You can see that Ruby implements `iv_tbl` and `const_tbl` in the same way: Class-level instance variables and constants are almost the same thing.
- Ruby uses `origin` to implement the `Module#prepend` feature. I'll discuss what `prepend` does and how Ruby implements it in Chapter 6.
- Ruby uses the `refined_class` pointer to implement the new experimental refinements feature, which I'll discuss further in Chapter 9.
- Finally, Ruby uses `allocator` internally to allocate memory for new instances of this class.

READING THE RCLASS C STRUCTURE DEFINITION

Now for a quick look at the actual `RClass` structure definition, as shown in Listing 5-16.

```
typedef struct rb_classext_struct rb_classext_t;
struct RClass {
    struct RBasic basic;
    rb_classext_t *ptr;
    struct st_table *m_tbl;
    struct st_table *iv_index_tbl;
};
```

Listing 5-16: The definition of the `RClass` C structure

Like the `RObject` definition we saw in Listing 5-7, this structure definition—including all of the values shown in Figure 5-21—can be found in the *include/ruby/ruby.h* file.

The `rb_classext_struct` structure definition, on the other hand, can be found in the *internal.h* C header file, as shown in Listing 5-17.

```
struct rb_classext_struct {
    VALUE super;
    struct st_table *iv_tbl;
    struct st_table *const_tbl;
    VALUE origin;
    VALUE refined_class;
    rb_alloc_func_t allocator;
};
```

Listing 5-17: The definition of the `rb_classext_struct` C structure

Again, you can see the values from Figure 5-21. Notice that the `st_table` C type appears four times in Listings 5-16 and 5-17; this is Ruby's hash table data structure. Internally, Ruby saves much of the information for each class using hash tables: the attribute names table, the method table, the class-level instance variable table, and the constants table.

Experiment 5-2: Where Does Ruby Save Class Methods?

We've seen how each `RClass` structure saves all methods defined in a certain class. In this example, Ruby stores information about the `first_name` method inside the `RClass` structure for `Mathematician` using the method table:

```
class Mathematician
  def first_name
    @first_name
  end
end
```

But what about class methods? It's common in Ruby to save methods in a class directly, using the syntax shown in Listing 5-18.

```
class Mathematician
  def self.class_method
    puts "This is a class method."
  end
end
```

Listing 5-18: Defining a class method using `def self`

Alternatively, you can use the syntax shown in Listing 5-19.

```
class Mathematician
  class << self
    def class_method
      puts "This is a class method."
    end
  end
end
```

Listing 5-19: Defining a class method using `class << self`

Are they saved in the `RClass` structure along with the normal methods for each class, perhaps with a flag to indicate they are class methods and not normal methods? Or are they saved somewhere else? Let's find out!

It's easy to see where class methods are *not* saved. They are obviously not saved in the `RClass` method table along with normal methods, because instances of `Mathematician` cannot call them, as demonstrated here:

```
> obj = Mathematician.new
> obj.class_method
 => undefined method `class_method' for
#< Mathematician:0x007fdd8384d1c8 (NoMethodError)
```

Now, keeping in mind that `Mathematician` is also a Ruby object, recall the following definition:

> A Ruby class is a Ruby object that also contains method definitions, attribute names, a superclass pointer, and a constants table.

We assume that Ruby should save methods for `Mathematician` just as it saves them for any object: in the method table for the object's class. In other words, Ruby should get `Mathematician`'s class using the `klass` pointer and save the method in the method table in that `RClass` structure, as shown in Figure 5-22.

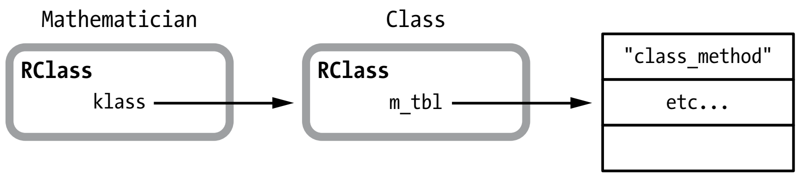


Figure 5-22: Shouldn't Ruby save class methods in the method table for the class's class?

But Ruby doesn't actually do this, as you can discover by creating another class and trying to call the new method:

```
> class AnotherClass; end
> AnotherClass.class_method
 => undefined method `class_method' for AnotherClass:Class (NoMethodError)
```

If Ruby had added the class method to the method table in the `Class` class, all classes in your application would have the method. Obviously this isn't what we intended by writing a class method, and thankfully Ruby doesn't implement class methods this way.

Then where do the class methods go? For a clue, use the method `ObjectSpace.count_objects`, shown in Listing 5-20:

```
  $ irb
❶ > ObjectSpace.count_objects[:T_CLASS]
❷  => 859
  > class Mathematician; end
   => nil
  > ObjectSpace.count_objects[:T_CLASS]
❸  => 861
```

Listing 5-20: Using `ObjectSpace.count_objects` with `:T_CLASS`

`ObjectSpace.count_objects` at ❶ returns the number of objects of a given type that exist. In this test, I'm passing the `:T_CLASS` symbol to get the count of class objects that exist in my IRB session. Before I create `Mathematician`, there are 859 classes at ❷. After I declare `Mathematician`, there are 861 at ❸—two more. That's odd. I declared one new class, but Ruby actually created two! What is the second one for and where is it?

It turns out that whenever you create a new class, internally Ruby creates two classes! The first class is your new class: Ruby creates a new `RClass` structure to represent your class, as described above. But internally Ruby also creates a second, hidden class called the *metaclass*. Why? To save any class methods that you might later create for your new class. In fact, Ruby sets the metaclass to be the class of your new class: It sets the `klass` pointer of your new `RClass` structure to point to the metaclass.

Without writing C code, there's no easy way to see the metaclass or the klass pointer value, but you can obtain the metaclass as a Ruby object like this:

```
class Mathematician
end

p Mathematician
 => Mathematician

p Mathematician.singleton_class
 => #<Class:Mathematician>
```

The first print statement displays the object's class, while the second displays the object's metaclass. The odd #<Class:Mathematician> syntax indicates that the second class is the metaclass for Mathematician. This is the second RClass structure that Ruby automatically created for me when I declared the Mathematician class. And this second RClass structure is where Ruby saves my class method, as shown in Figure 5-23.

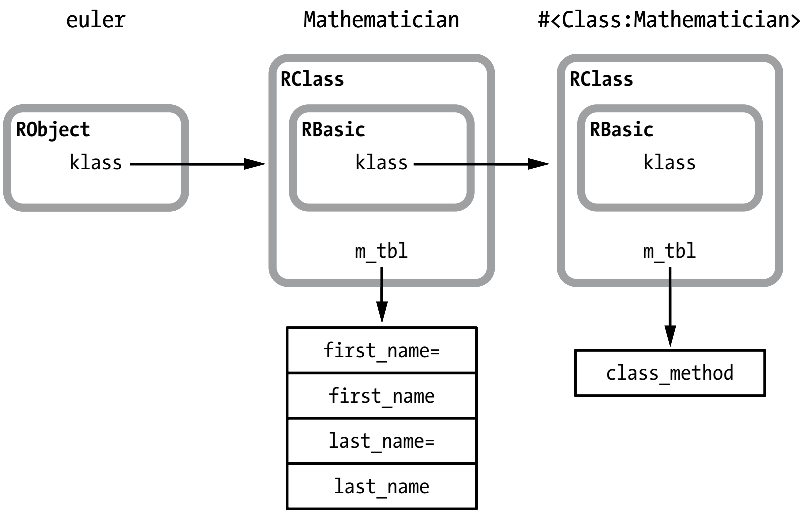


Figure 5-23: An object, its class, and its metaclass

If we now display the methods for the metaclass, we'll see all the methods of the Class class, along with the new class method for Mathematician:

```
p obj.singleton_class.methods
 => [ ... :class_method, ...  ]
```

Summary

In this chapter we've seen how Ruby represents objects and classes internally: Ruby uses the `RObject` structure to represent instances of any custom classes you define in your code and of some classes predefined by Ruby itself. The `RObject` structure is remarkably simple, containing just a pointer to the object's class and a table of instance variable values, along with a count of the variables. The simplicity of its structure leads us to a very simple definition of a Ruby object:

> Every Ruby object is the combination of a class pointer and an array of instance variables.

This definition is powerful and useful because everything in Ruby is an object: Whenever you use a value in your Ruby program, regardless of what it is, remember that it will be an object and will therefore have a class pointer and instance variables.

We also saw that Ruby uses special C structures to represent instances of many commonly used, built-in Ruby classes called "generic" objects. For example, Ruby uses the `RString` structure to represent an instance of the `String` class, `RArray` for an instance of the `Array` class, or `RRegexp` for an instance of the `Regexp` class. While these structures are different, Ruby also saves a class pointer and an array of instance variables for each of these generic objects. Finally, we saw that Ruby saves some simple values, such as small integers and symbols, without using a C structure at all. Ruby saves these values right inside the `VALUE` pointers that otherwise would point to the structure holding the value.

While Ruby objects are simple, we learned in this chapter that Ruby classes aren't quite so simple. The `RClass` structure working with the `rb_classext_struct` structure saves a large set of information. Learning this forced us to write a more complex definition for Ruby classes:

> A Ruby class is a Ruby object that also contains method definitions, attribute names, a superclass pointer, and a constants table.

Looking inside `RClass` and `rb_classext_struct`, we saw that Ruby classes are also Ruby objects, which therefore also contain instance variables and a class pointer. We looked at the difference between a class's instance variables and class variables and learned that Ruby saves both of these variable types in the same hash table. We discovered how classes also contain a series of hash tables that store their methods, the names of the object-level instance variables, and constants defined within the class. Finally, we saw how each Ruby class records its superclass using the `super` pointer.