Chapter 3

Classes, Objects, and Variables

From the examples we've shown so far, you may be wondering about our earlier assertion that Ruby is an object-oriented language. Well, this chapter is where we justify that claim. We're going to be looking at how you create classes and objects in Ruby and at some of the ways in which Ruby is more powerful than most object-oriented languages.

As we saw back on page 35, everything we manipulate in Ruby is an object. And every object in Ruby was generated either directly or indirectly from a class. In this chapter, we'll look in more depth at creating and manipulating those classes.

Let's give ourselves a simple problem to solve. Let's say that we're running a secondhand bookstore. Every week, we do stock control. A gang of clerks uses portable bar-code scanners to record every book on our shelves. Each scanner generates a simple comma-separated value (CSV) file containing one row for each book scanned. The row contains (among other things) the book's ISBN and price. An extract from one of these files looks something like this:

```
"Date","ISBN","Amount"
"2008-04-12","978-1-9343561-0-4",39.45
"2008-04-13","978-1-9343561-6-6",45.67
"2008-04-14","978-1-9343560-7-4",36.95
```

Our job is to take all the CSV files and work out how many of each title we have, as well as the total list price of the books in stock.

Whenever you're designing OO systems, a good first step is to identify the *things* you're dealing with. Typically each type of thing becomes a class in your final program, and the things themselves are instances of these classes.

It seems pretty clear that we'll need something to represent each data reading captured by the scanners. Each instance of this will represent a particular row of data, and the collection of all of these objects will represent all the data we've captured.

Let's call this class BookInStock. (Remember, class names start with an uppercase letter, and method names normally start with a lowercase letter.)

```
class BookInStock
end
```

As we saw in the previous chapter, we can create new instances of this class using new:

```
a_book = BookInStock.new
another_book = BookInStock.new
```

After this code runs, we'd have two distinct objects, both of class BookInStock. But, apart from the fact that they have different identities, these two objects are otherwise the same—there's nothing to distinguish one from the other. And, what's worse, these objects actually don't hold any of the information we need them to hold.

The best way to fix this is to provide the objects with an initialize method. This lets us set the state of each object as it is constructed. We store this state in *instance variables* inside the object. (Remember instance variables? They're the ones that start with an @ sign.) Because each object in Ruby has its own distinct set of instance variables, each object can have its own unique state.

So, here's our updated class definition:

Download **samples/tutclasses_4.rb**

```
class BookInStock
  def initialize(isbn, price)
    @isbn  = isbn
    @price = Float(price)
  end
end
```

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

For class BookInStock, the initialize method takes two parameters. These parameters act just like local variables within the method, so they follow the local variable naming convention of starting with a lowercase letter. But, as local variables, they would just evaporate once the initialize method returns, so we need to transfer them into instance variables. This is very common behavior in an initialize method—the intent is to have our object set up and usable by the time initialize returns.

This method also illustrates something that often trips up newcomers to Ruby. Notice how we say @isbn = isbn. It's easy to imagine that the two variables here, @isbn and isbn, are somehow related—it looks like they have the same name. But they don't. The former is an instance variable, and the "at" sign is actually part of its name.

Finally, this code illustrates a simple piece of validation. The Float method takes its argument and converts it to a floating-point number,[1] terminating the program with an error if that conversion fails. (Later in the book we'll see how to handle these exceptional situations.) What we're doing here is saying that we want to accept any object for the price parameter as long as that parameter can be converted to a float. We can pass in a float, an integer, and even a string containing the representation of a float, and it will work. Let's try this now. We'll create three objects, each with different initial state. The p method prints out an internal representation of an object. Using it, we can see that in each case our parameters got transferred into the object's state, ending up as instance variables:

Download **samples/tutclasses_5.rb**

```
class BookInStock
  def initialize(isbn, price)
    @isbn  = isbn
    @price = Float(price)
  end
end
b1 = BookInStock.new("isbn1", 3)
p b1
b2 = BookInStock.new("isbn2", 3.14)
p b2
b3 = BookInStock.new("isbn3", "5.67")
p b3
```

produces:

```
#<BookInStock:0x0a37f0 @isbn="isbn1", @price=3.0>
#<BookInStock:0x0a3584 @isbn="isbn2", @price=3.14>
#<BookInStock:0x0a3354 @isbn="isbn3", @price=5.67>
```

Why did we use p to write out our objects, rather than puts? Well, let's repeat the code using puts:

Download **samples/tutclasses_6.rb**

```
b1 = BookInStock.new("isbn1", 3)
puts b1
b2 = BookInStock.new("isbn2", 3.14)
puts b2
b3 = BookInStock.new("isbn3", 5.67)
puts b3
```

produces:

```
#<BookInStock:0x0a38cc>
#<BookInStock:0x0a3764>
#<BookInStock:0x0a36d8>
```

1. Yes, we know. We shouldn't be holding prices in inexact old floats. Ruby has classes that hold fixed-point values exactly, but we want to look at classes, not arithmetic, in this section.

Remember, puts simply writes strings to your program's standard output. When you pass it an object based on a class you wrote, it doesn't really know what to do with it, so it uses a very simple expedient: it writes the name of the object's class, followed by a colon and the object's unique identifier (a hexadecimal number). It puts the whole lot inside #<...>.

Our experience tells us that during development we'll be printing out the contents of a BookInStock object many times, and the default formatting leaves something to be desired. Fortunately, Ruby has a standard message, to_s, that it sends to any object it wants to render as a string. So, when we pass one of our BookInStock objects to puts, the puts method calls to_s in that object to get its string representation. So, let's override the default implementation of to_s to give us a better rendering of our objects:

Download **samples/tutclasses_7.rb**

```
class BookInStock
  def initialize(isbn, price)
    @isbn  = isbn
    @price = Float(price)
  end
  def to_s
    "ISBN: #{@isbn}, price: #{@price}"
  end
end
b1 = BookInStock.new("isbn1", 3)
puts b1
b2 = BookInStock.new("isbn2", 3.14)
puts b2
b3 = BookInStock.new("isbn3", "5.67")
puts b3
```

produces:

```
ISBN: isbn1, price: 3.0
ISBN: isbn2, price: 3.14
ISBN: isbn3, price: 5.67
```

There's something going on here that's both trivial and profound. See how the values we set into the instance variables @isbn and @price in the initialize method are subsequently available in the to_s method? That shows how instance variables work—they're stored with each object and available to all the instance methods of those objects.

Objects and Attributes

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

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

For our BookInStock objects, the first thing we may need is the ability to find out the ISBN and price (so we can count each distinct book and perform price calculations). One way of doing that is to write accessor methods:

Download samples/tutclasses_8.rb

```
class BookInStock
  def initialize(isbn, price)
    @isbn  = isbn
    @price = Float(price)
  end
  def isbn
    @isbn
  end
  def price
    @price
  end
  # ..
end
book = BookInStock.new("isbn1", 12.34)
puts "ISBN   = #{book.isbn}"
puts "Price  = #{book.price}"
```

produces:

```
ISBN   = isbn1
Price  = 12.34
```

Here we've defined two accessor methods to return the values of the two instance variables. The method isbn, for example, returns the value of the instance variable @isbn (because the last thing executed in the method is the expression that simply evaluates the @isbn variable).

Because writing accessor methods is such a common idiom, Ruby provides a convenient shortcut. attr_reader creates these attribute reader methods for you:

Download samples/tutclasses_9.rb

```
class BookInStock
  attr_reader :isbn, :price
  def initialize(isbn, price)
    @isbn  = isbn
    @price = Float(price)
  end
  # ..
end
book = BookInStock.new("isbn1", 12.34)
puts "ISBN   = #{book.isbn}"
puts "Price  = #{book.price}"
```

produces:

```
ISBN   = isbn1
Price  = 12.34
```

This is the first time we've used *symbols* in this chapter. As we discussed back on page 42, symbols are just a convenient way of referencing a name. In this code, you can think of

:isbn as meaning the *name* isbn and plain isbn as meaning the *value* of the variable. In this example, we named the accessor methods isbn and price. The corresponding instance variables are @isbn and @price. These accessor methods are identical to the ones we wrote by hand earlier.

There's a common misconception, particularly among people who come from languages such as Java and C#, that the attr_reader declaration somehow declares instance variables. It doesn't. It creates the accessor methods, but the variables themselves don't need to be declared—they just pop into existence when you use them. Ruby completely decouples instance variables and accessor methods, as we'll see in the section *Virtual Attributes* on the next page.

Writable Attributes

Sometimes you need to be able to set an attribute from outside the object. For example, let's assume that we sometimes have to discount the price of some titles after reading in the raw scan data.

In languages such as C# and Java, you'd do this with *setter functions*:

```
class JavaBookInStock {                        // Java code
  private double _price;
  public double getPrice() {
    return _price;
  }
  public void setPrice(double newPrice) {
    _price = newPrice;
  }
}
b = new JavaBookInStock(....);
b.setPrice(calculate_discount(b.getPrice()));
```

In Ruby, the attributes of an object can be accessed as if they were any other variable. We saw this earlier with phrases such as book.isbn. So, it seems natural to be able to assign to these variables when you want to set the value of an attribute. It turns out you do that by creating a Ruby method whose name ends with an equals sign. These methods can be used as the target of assignments:

Download **samples/tutclasses_11.rb**

```
class BookInStock
  attr_reader :isbn, :price
  def initialize(isbn, price)
    @isbn  = isbn
    @price = Float(price)
  end
  def price=(new_price)
    @price = new_price
  end
  # ...
end
```

```
book = BookInStock.new("isbn1", 33.80)
puts "ISBN      = #{book.isbn}"
puts "Price     = #{book.price}"
book.price = book.price * 0.75        # discount price
puts "New price = #{book.price}"
```

produces:

```
ISBN      = isbn1
Price     = 33.8
New price = 25.35
```

The assignment book.price = book.price * 0.75 invokes the method price= in the book object, passing it the discounted price as an argument. If you create a method whose name ends with an equals sign, that name can appear on the left side of an assignment.

Again, Ruby provides a shortcut for creating these simple attribute-setting methods. If you want a write-only accessor, you can use the form attr_writer, but that's fairly rare. You're far more likely to want both a reader and a writer for a given attribute, so you'll use the handy-dandy attr_accessor method:

Download **samples/tutclasses_12.rb**

```
class BookInStock
  attr_reader   :isbn
  attr_accessor :price
  def initialize(isbn, price)
    @isbn  = isbn
    @price = Float(price)
  end
  # ...
end
book = BookInStock.new("isbn1", 33.80)
puts "ISBN      = #{book.isbn}"
puts "Price     = #{book.price}"
book.price = book.price * 0.75        # discount price
puts "New price = #{book.price}"
```

produces:

```
ISBN      = isbn1
Price     = 33.8
New price = 25.35
```

Virtual Attributes

These attribute-accessing methods do not have to be just simple wrappers around an object's instance variables. For example, you may want to access the price as an exact number of cents, rather than as a floating-point number of dollars.[2]

2. We multiply the floating-point price times 100 to get the price in cents but then add 0.5 before converting to an integer. Why? Because floating-point numbers don't always have an exact internal representation. When we

Download samples/tutclasses_13.rb

```
class BookInStock
  attr_reader   :isbn
  attr_accessor :price
  def initialize(isbn, price)
    @isbn  = isbn
    @price = Float(price)
  end
  def price_in_cents
    Integer(price*100 + 0.5)
  end
  # ...
end
book = BookInStock.new("isbn1", 33.80)
puts "Price          = #{book.price}"
puts "Price in cents = #{book.price_in_cents}"
```

produces:

```
Price          = 33.8
Price in cents = 3380
```

We can take this even further and allow people to assign to our virtual attribute, mapping the value to the instance variable internally:

Download samples/tutclasses_14.rb

```
class BookInStock
  attr_reader   :isbn
  attr_accessor :price
  def initialize(isbn, price)
    @isbn  = isbn
    @price = Float(price)
  end
  def price_in_cents
    Integer(price*100 + 0.5)
  end
  def price_in_cents=(cents)
    @price = cents / 100.0
  end
  # ...
end
book = BookInStock.new("isbn1", 33.80)
puts "Price          = #{book.price}"
puts "Price in cents = #{book.price_in_cents}"
book.price_in_cents = 1234
```

multiply 33.8 times 100, we get 3379.99999999999954525265. The Integer method would truncate this to 3379. Adding 0.5 before calling Integer rounds up the floating-point value, ensuring we get the best integer representation. This is a good example of why you want to use BigDecimal, not Float, in financial calculations.

```
puts "Price          = #{book.price}"
puts "Price in cents = #{book.price_in_cents}"
```

produces:

```
Price          = 33.8
Price in cents = 3380
Price          = 12.34
Price in cents = 1234
```

Here we've used attribute methods to create a virtual instance variable. To the outside world, price_in_cents seems to be an attribute like any other. Internally, though, it has no corresponding instance variable.

This is more than a curiosity. In his landmark book *Object-Oriented Software Construction* [Mey97], Bertrand Meyer calls this the *Uniform Access Principle*. By hiding the difference between instance variables and calculated values, you are shielding the rest of the world from the implementation of your class. You're free to change how things work in the future without impacting the millions of lines of code that use your class. This is a big win.

Attributes, Instance Variables, and Methods

This description of attributes may leave you thinking that they're nothing more than methods —why'd we need to invent a fancy name for them? In a way, that's absolutely right. An attribute *is* just a method. Sometimes an attribute simply returns the value of an instance variable. Sometimes an attribute returns the result of a calculation. And sometimes those funky methods with equals signs at the end of their names are used to update the state of an object. So, the question is, where do attributes stop and regular methods begin? What makes something an attribute and not just a plain old method? Ultimately, that's one of those "angels on a pinhead" questions. Here's a personal take.

When you design a class, you decide what internal state it has and also decide how that state is to appear on the outside (to users of your class). The internal state is held in instance variables. The external state is exposed through methods we're calling *attributes*. And the other actions your class can perform are just regular methods. It really isn't a crucially important distinction, but by calling the external state of an object its *attributes*, you're helping clue people in to how they should view the class you've written.

Classes Working with Other Classes

Our original challenge was to read in data from multiple CSV files and produce various simple reports. So far, all we have is BookInStock, a class that represents the data for one book.

During OO design, you identify external things and make them classes in your code. But there's another source of classes in your designs. There are the classes that correspond to things inside your code itself. For example, we know that the program we're writing will need to consolidate and summarize CSV data feeds. But that's a very passive statement. Let's turn it into a design by asking ourselves *what* does the summarizing and consolidating.

And the answer (in our case) is a *CSV reader*. Let's make it into a class. Here it is in skeletal form:

```
class CsvReader
  def initialize
    # ...
  end
  def read_in_csv_data(csv_file_name)
    # ...
  end
  def total_value_in_stock
    # ...
  end
  def number_of_each_isbn
    # ...
  end
end
```

We'd call it using something like this:

```
reader = CsvReader.new
reader.read_in_csv_data("file1.csv")
reader.read_in_csv_data("file2.csv")
  :         :              :
puts "Total value in stock = #{reader.total_value_in_stock}"
```

We need to be able to handle multiple CSV files, so our reader object needs to accumulate the values from each CSV file it is fed. We'll do that by keeping an array of values in an instance variable. And how shall we represent each book's data? Well, we just finished writing the BookInStock class, so that problem is solved. The only other question is how we parse data in a CSV file. Fortunately, Ruby comes with a good CSV library (described on page 739). Given a CSV file with a header line, we can iterate over the remaining rows and extract values by name:

```
class CsvReader
  def initialize
    @books_in_stock = []
  end
  def read_in_csv_data(csv_file_name)
    CSV.foreach(csv_file_name, headers: true) do |row|
      @books_in_stock << BookInStock.new(row["ISBN"], row["Amount"])
    end
  end
end
```

Just because you're probably wondering what's going on, let's dissect that read_in_csv_data method. On the first line, we tell the CSV library to open the file with the given name. The headers: true option tells the library to parse the first line of the file as the names of the columns.

The library then reads the rest of the file, passing each row in turn to the block (the code between do and end).[3] Inside the block, we extract the data from the ISBN and Amount columns and use that data to create a new BookInStock object. We then append that object to an instance variable called @books_in_stock. And just where does that variable come from? It's an array that we created in the initialize method.

Again, this is the pattern you want to aim for. Your initialize method sets up an environment for your object, leaving it in a usable state. Other methods then use that state.

So, let's turn this from a code fragment into a working program. We're going to organize our source into three files. The first, book_in_stock.rb, will contain the definition of the class BookInStock. The second, csv_reader.rb, is the source for the CsvReader class. Finally, a third file, stock_stats.rb, is the main driver program.

Here's book_in_stock.rb:

Download **samples/book_in_stock.rb**

```
class BookInStock
  attr_reader :isbn, :price
  def initialize(isbn, price)
    @isbn  = isbn
    @price = Float(price)
  end
end
```

Here's the csv_reader.rb file. The CsvReader class has two external dependencies: it needs the standard CSV library, and it needs the BookInStock class that's defined in the file book_in_stock.rb. Ruby has a couple of helper methods that let us load external files. In this file we use require to load in the Ruby CSV library and require_relative to load in the book_in_stock class we wrote. (We use require_relative for this because the location of the file we're loading is relative to the file we're loading it from—they're both in the same directory.)

Download **samples/csv_reader.rb**

```
require 'csv'
require_relative 'book_in_stock'
class CsvReader
  def initialize
    @books_in_stock = []
  end
  def read_in_csv_data(csv_file_name)
    CSV.foreach(csv_file_name, headers: true) do |row|
      @books_in_stock << BookInStock.new(row["ISBN"], row["Amount"])
    end
  end
```

3. If you encounter an error along the lines of 'Float': can't convert nil into Float (TypeError) when you run this code, you've likely got extra spaces at the end of the header line in your CSV data file. The CSV library is pretty strict about the formats it accepts.

```
    # later we'll see how to use inject to sum a collection
    def total_value_in_stock
      sum = 0.0
      @books_in_stock.each {|book| sum += book.price}
      sum
    end
    def number_of_each_isbn
      # ...
    end
  end
```

And finally, here's our main program, in the file stock_stats.rb:

Download **samples/stock_stats.rb**

```
require_relative 'csv_reader'
reader = CsvReader.new
ARGV.each do |csv_file_name|
  STDERR.puts "Processing #{csv_file_name}"
  reader.read_in_csv_data(csv_file_name)
end
puts "Total value = #{reader.total_value_in_stock}"
```

Again, this file uses require_relative to bring in the library it needs (in this case, just the csv_reader.rb file). It uses the ARGV variable to access the program's command-line arguments, loading CSV data for each.

We can run this program using the simple CSV data file we showed on page 50:

```
$ ruby stock_stats.rb data.csv
```

produces:

```
Processing data.csv
Total value = 122.07
```

Do we need three source files for this? No. In fact, most Ruby developers would probably start off by sticking all this code into a single file—it would contain both class definitions as well as the driver code. But as your programs grow (and almost all programs grow over time), you'll find that this starts to get cumbersome. You'll also find it harder to write automated tests against the code if it is in a monolithic chunk. Finally, you won't be able to reuse classes if they're all bundled into the final program.

Anyway, let's get back to our discussion of classes.

Access Control

When designing a class interface, it's important to consider just how much of your class you'll be exposing to the outside world. Allow too much access into your class, and you risk increasing the coupling in your application—users of your class will be tempted to rely on details of your class's implementation, rather than on its logical interface. The good news

is that the only easy way to change an object's state in Ruby is by calling one of its methods. Control access to the methods, and you've controlled access to the object. A good rule of thumb is never to expose methods that could leave an object in an invalid state.

Ruby gives you three levels of protection:

- *Public methods* can be called by anyone—no access control is enforced. Methods are public by default (except for initialize, which is always private).
- *Protected methods* can be invoked only by objects of the defining class and its subclasses. Access is kept within the family.
- *Private methods* cannot be called with an explicit receiver—the receiver is always the current object, also known as *self*. This means that private methods can be called only in the context of the current object; you can't invoke another object's private methods.

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

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

Specifying Access Control

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

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

```
class MyClass
      def method1    # default is 'public'
        #...
      end
  protected          # subsequent methods will be 'protected'
      def method2    # will be 'protected'
        #...
      end
  private            # subsequent methods will be 'private'
      def method3    # will be 'private'
        #...
      end
  public             # subsequent methods will be 'public'
      def method4    # so this will be 'public'
        #...
      end
end
```

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

Download **samples/tutclasses_23.rb**

```
class MyClass
  def method1
  end
  # ... and so on
  public    :method1, :method4
  protected :method2
  private   :method3
end
```

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

Download **samples/tutclasses_24.rb**

```
class Account
    attr_accessor :balance
    def initialize(balance)
      @balance = balance
    end
end
class Transaction
    def initialize(account_a, account_b)
      @account_a = account_a
      @account_b = account_b
    end
  private
    def debit(account, amount)
      account.balance -= amount
    end
    def credit(account, amount)
      account.balance += amount
    end
  public
    #...
    def transfer(amount)
      debit(@account_a, amount)
      credit(@account_b, amount)
    end
    #...
end
savings = Account.new(100)
checking = Account.new(200)
trans = Transaction.new(checking, savings)
trans.transfer(50)
```

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

Download samples/tutclasses_25.rb

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

Because cleared_balance is protected, it's available only within Account objects.

Variables

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

Let's confirm this with some code:

Download samples/tutclasses_26.rb

```
person = "Tim"
puts "The object in 'person' is a #{person.class}"
puts "The object has an id of #{person.object_id}"
puts "and a value of '#{person}'"
```

produces:

```
The object in 'person' is a String
The object has an id of 338010
and a value of 'Tim'
```

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

So, is a variable an object? In Ruby, the answer is "no." A variable is simply a reference to an object. Objects float around in a big pool somewhere (the heap, most of the time) and are pointed to by variables. Let's make the example slightly more complicated:

Download samples/tutclasses_27.rb

```
person1 = "Tim"
person2 = person1
person1[0] = 'J'
puts "person1 is #{person1}"
puts "person2 is #{person2}"
```

produces:

```
person1 is Jim
person2 is Jim
```

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

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

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

Download **samples/tutclasses_28.rb**

```
person1 = "Tim"
person2 = person1.dup
person1[0] = "J"
puts "person1 is #{person1}"
puts "person2 is #{person2}"
```

produces:

```
person1 is Jim
person2 is Tim
```

You can also prevent anyone from changing a particular object by freezing it. Attempt to alter a frozen object, and Ruby will raise a RuntimeError exception:

Download **samples/tutclasses_29.rb**

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

produces:

```
prog.rb:4:in `[]=': can't modify frozen string (RuntimeError)
from /tmp/prog.rb:4:in `<main>'
```

There's more to say about classes and objects in Ruby. We still have to look at class methods and at concepts such as mixins and inheritance. We'll do that in Chapter 5 on page 91. But, for now, take away the fact that everything you manipulate in Ruby is an object and the fact that objects start life as instances of classes. And one of the most common things we do with objects is create collections of them. But that's the subject of our next chapter.

Figure 3.1. Variables Hold Object References