

Covers Ruby 1.9.1



The **WELL-GROUNDED**  
**Rubyist**

SAMPLE CHAPTER

David A. Black

 MANNING



*The Well-Grounded Rubyist*

by David A. Black

**Chapter 9**

# *brief contents*

---

## **PART 1 RUBY FOUNDATIONS ..... 1**

- 1 ■ Bootstrapping your Ruby literacy 3
- 2 ■ Objects, methods, and local variables 32
- 3 ■ Organizing objects with classes 60
- 4 ■ Modules and program organization 90
- 5 ■ The default object (self), scope, and visibility 115
- 6 ■ Control-flow techniques 148

## **PART 2 BUILT-IN CLASSES AND MODULES ..... 183**

- 7 ■ Built-in essentials 185
- 8 ■ Strings, symbols, and other scalar objects 213
- 9 ■ Collection and container objects 247
- 10 ■ Collections central: Enumerable and Enumerator 278
- 11 ■ Regular expressions and regexp-based string operations 319
- 12 ■ File, I/O, and system operations 347

<b>PART 3</b>	<b>RUBY DYNAMICS .....</b>	<b>373</b>
13	■ Object individuation	375
14	■ Callable and runnable objects	405
15	■ Callbacks, hooks, and runtime introspection	441

# Collection and container objects

---

## ***In this chapter***

- Sequentially ordered collections with arrays
- Keyed collections with hashes
- Inclusion and membership tests with ranges
- Unique, unordered collections with sets

In programming generally, you deal not only with individual objects but with *collections* of objects. You search through collections to find an object that matches certain criteria (like a magazine object containing a particular article); you sort collections for further processing or visual presentation; you filter collections to include or exclude particular items; and so forth. All of these operations, and similar ones, depend on objects being accessible in collections.

Ruby represents collections of objects by putting them inside *container* objects. In Ruby, two built-in classes dominate the container-object landscape: *arrays* and *hashes*. We'll start this chapter by looking at the `Array` and `Hash` classes: first in comparison with each other, to establish an overall understanding, and then separately.

We're also going to look at two other classes: `Range` and `Set`. Ranges are a bit of a hybrid: they work partly as boolean filters (in the sense that you can perform a true/false test as to whether a given value lies inside a given range) but also, in some contexts, as collections. Sets are collections through and through. The only reason the `Set` class requires special introduction is that it isn't a core Ruby class; it's a standard library class, and although we're not looking at many of those in depth in this book, sets fall nicely into place beside arrays and hashes and merit our attention.

In reading this chapter, keep in mind that it represents a first pass through a kind of mega-topic that we'll be visiting in the next chapter too. Ruby implements collections principally through the technique of defining classes that mix in the `Enumerable` module. That module gives you a package deal on methods that sort, sift, filter, count, and transform collections. In this chapter, we're going to look primarily at what you can do with the major collection classes *other* than take advantage of their `Enumerable` nature. The next chapter will deal directly with `Enumerable` and how it's used. We'll look at enough of enumerability here to bootstrap this chapter, and then we'll come back to it in the next.

Also, keep in mind that collections are, themselves, objects. You send them messages, assign them to variables, and so forth, in normal object fashion. They just have an extra dimension, beyond the scalar.

## 9.1 **Arrays and hashes in comparison**

An *array* is an ordered collection of objects—*ordered* in the sense that you can select objects from the collection based on a consistent numerical index. You'll have noticed that we've already used arrays in some of the examples earlier in the book. It's hard *not* to use arrays in Ruby.

*Hashes* in Ruby 1.9 are also ordered collections—and that's a big change from previous versions of Ruby, where hashes are unordered (in the sense that they have no idea of what their first, last, or *n*th element is). Hashes store objects in pairs, each pair consisting of a *key* and a *value*. You retrieve a value by means of the key. Hashes remember the order in which their keys were inserted; that's the order in which the hash replays itself for you if you iterate through the pairs in it or print a string representation of it to the screen.

Any Ruby object can serve as a hash key and/or value, but keys are unique: you can have only one key/value pair for any given key. Hashes (or similar data storage types) are sometimes called *dictionaries* or *associative arrays* in other languages. They offer a tremendously—sometimes surprisingly—powerful way of storing and retrieving data.

Arrays and hashes are closely connected. An array is, in a sense, a hash, where the keys happen to be consecutive integers. Hashes are, in a sense, arrays, where the indexes are allowed to be anything, not just integers. If you do use consecutive integers as hash keys, arrays and hashes start behaving similarly when you do lookups:

```
array = ["ruby", "diamond", "emerald"]
hash = { 0 => "ruby", 1 => "diamond", 2 => "emerald" }
```

```
puts array[0] # ruby
puts hash[0]  # ruby
```

Even if you don't use integers, hashes exhibit a kind of "meta-index" property, based on the fact that they have a certain number of key/value pairs and that those pairs can be counted off consecutively. You can see this property in action by stepping through a hash with the `each_with_index` method, which yields a counter value to the block along with the key and value:

```
hash = { "red" => "ruby", "white" => "diamond", "green" => "emerald" }
hash.each_with_index {|(key,value),i| puts "Pair #{i} is: #{key}/#{value}"
}
```

The output from this code snippet is

```
Pair 0 is: red/ruby
Pair 1 is: white/diamond
Pair 2 is: green/emerald
```

The "index" is an integer counter, maintained as the pairs go by. The pairs are the actual content of the hash.

**TIP** The parentheses in the block parameters `(key,value)` serve to split apart an array. Each key/value pair comes at the block as an array of two elements. If the parameters were `key,value,i`, then the parameter `key` would end up bound to the entire `[key,value]` array; `value` would be bound to the index; and `i` would be `nil`. That's obviously not what we want. The parenthetical grouping of `(key,value)` is a signal that we want the array to be distributed across those two parameters, element by element.

Conversions of various kinds between arrays and hashes are common. Some such conversions are automatic: if you perform certain operations of selection or extraction of pairs from a hash, you'll get back an array. Other conversions require explicit instructions, such as turning a flat array `([1,2,3,4])` into a hash `({1 => 2, 3 => 4})`. You'll see a good amount of back and forth between these two collection classes, both here in this chapter and in lots of Ruby code.

In the next two sections, we'll look at arrays and hashes in depth. Let's start with arrays.

## 9.2 Collection handling with arrays

An *array* is an object whose job is to store other objects. Arrays are ordered collections; you can get at their contents by the use of numerical indexes. The contents of an array always remain in the same order, unless you change it. Any object can be stored in an array, including other arrays, hashes, filehandles, classes, `true` and `false` ... any object at all.

Arrays are the bread-and-butter way to handle collections of objects. We'll put arrays through their paces in this section; we'll look at the varied techniques available

for creating arrays; how to insert, retrieve, and remove array elements; combining arrays with each other; transforming arrays (for example, flattening a nested array into a one-dimensional array); and querying arrays as to their properties and state.

### 9.2.1 **Creating a new array**

You can create an array four ways:

- With the `new` method
- With the literal array constructor (square brackets)
- With a top-level method called `Array`
- With the special `%w{...}` notation

You'll see all of these techniques in heavy rotation in Ruby code, so they're all worth knowing. We'll look at each in turn.

#### **ARRAY#NEW**

The `new` method on the array class works in the usual way:

```
a = Array.new
```

You can then add objects to the array using techniques we'll look at later.

`Array.new` lets you specify the size of and, if you wish, initialize the contents of the array. Here's an irb exchange that illustrates both possibilities:

```
>> Array.new(3)           ❶
=> [nil, nil, nil]
>> Array.new(3, "abc")   ❷
=> ["abc", "abc", "abc"]
```

If you give one argument to `Array.new` ❶, you get an array of the size you asked for, with all elements set to `nil`. If you give two arguments ❷, you get an array of the size you asked for, with each element initialized to contain the second argument.

You can even supply a code block to `Array.new`. In that case, the elements of the array are initialized by repeated calls to the block:

```
>> n = 0
=> 0
>> Array.new(3) { n += 1; n * 10 } ❶
=> [10, 20, 30]                    ❷
```

In this example, the `new` array has a size of three. Each of the three elements is set to the return value of the code block. The code inside the block ❶, executed three times, produces the values 10, 20, and 30—and those are the initial values in the array ❷.

**WARNING** When you initialize multiple elements of an array using a second argument to `Array.new`—as in `Array.new(3, "abc")`—*all the elements* of the array are initialized *to the same object*. If you do `a = Array.new(3, "abc"); a[0] << "def"; puts a[1]`, you'll find that the second element of the array is now "abcdef", even though you appended "def" to the first element. That's because the first and second positions in the array contain

the same string object, not two different strings that happen to both consist of “abc”. To create an array that inserts a different “abc” string into each slot, you should use `Array.new(3) { "abc" }`. The code block runs three times, each time generating a new string (same characters, different string object).

Pre-initializing arrays isn’t always necessary, because your arrays grow as you add elements to them. But if and when you need this functionality—and/or if you see it in use and want to understand it—it’s there.

#### THE LITERAL ARRAY CONSTRUCTOR: []

The second way to create an array is by using the *literal array constructor* [] (square brackets):

```
a = []
```

When you create an array with the literal constructor, you can put objects into the array at the same time:

```
a = [1,2,"three",4, []]
```

Notice that the last element in this array is another array. That’s perfectly legitimate; you can nest arrays to as many levels as you wish.

Square brackets can mean a lot of different things in Ruby: array construction, array indexing (as well as string and hash indexing), character classes in regular expressions, delimiters in %q[]-style string notation, even the calling of an anonymous function. You can make an initial division of the various uses of square brackets by distinguishing cases where they’re a semantic construct from cases where they’re the name of a method. It’s worth practicing on a few examples like this to get a feel for the way the square brackets play out in different contexts:

```
[1,2,3][1]    ← Index 1 on the array [1,2,3]
```

Now, back to array creation.

#### THE ARRAY METHOD

The third way to create an array is with a *method* (even though it looks like a class name!) called `Array`. As you know from having seen the `Integer` and `Float` methods, it’s legal to define methods whose names begin with capital letters. Those names look exactly like constants, and in core Ruby itself, capitalized methods tend to have the same names as classes to which they’re related.

#### Some more built-in methods that start with uppercase letters

In addition to the `Array` method and the two uppercase-style conversion methods you’ve already seen (`Integer` and `Float`, the “fussy” versions of `to_i` and `to_f`), Ruby provides a few other top-level methods whose names look like class names: `Complex`, `Rational`, and `String`. In each case, the method returns an object of the class that its name looks like.

**Some more built-in methods that start with uppercase letters (continued)**

The `String` method is a wrapper around `to_s`, meaning `String(obj)` is equivalent to `obj.to_s`. `Complex` and `Rational` correspond to the `to_c` and `to_r` methods available for numerics and strings—except `Complex` and `Rational`, like `Integer` and `Float`, are fussy: they don't take kindly to non-numeric strings. `"abc".to_c` gives you `(0+0i)`; but `Complex("abc")` raises `ArgumentError`, and `Rational` and `to_r` behave similarly.

We're not covering rational and complex numbers here, but now you know how to generate them, in case they're of interest to you!

The `Array` method creates an array from its single argument. If the argument object has a `to_ary` method defined, then `Array` calls that method on the object to generate an array. (Remember that `to_ary` is the quasi-typecasting array conversion method.) If there's no `to_ary` method, it tries to call `to_a`. If `to_a` isn't defined either, `Array` wraps the object in an array and returns that:

```
>> string = "A string"
=> "A string"
>> string.respond_to?(:to_ary)
=> false
>> string.respond_to?(:to_a)
=> false
>> Array(string)           ❶
=> ["A string"]
>> def string.to_a         ❷
>>   split(//)
>> end
=> nil
>> Array(string)
=> ["A", " ", "s", "t", "r", "i", "n", "g"]
```

In this example, the first attempt to run `Array` on the string ❶ results in a one-element array, where the one element is the string. That's because strings have neither a `to_ary` nor a `to_a` method. But after `to_a` is defined for the string ❷, the result of calling `Array` is different: it now runs the `to_a` method and uses that as its return value.

Among the various array constructors, the literal `[]` is probably the most common, followed by `Array.new` and the `Array` method, in that order. But each has its place. The literal constructor is the most succinct; and when you learn what it means, it clearly announces “array” when you see it. The `Array` method is constrained by the discipline of there having to be a `to_ary` or `to_a` method available.

**THE %w AND %W ARRAY CONSTRUCTORS**

As a special dispensation to help you create arrays of strings, Ruby provides a `%w` operator, much in the same family as the `%q`-style operators you've seen already, that automatically generates an array of strings from the space-separated strings you put inside it. You can see how it works by using it in `irb` and looking at the result:

```
>> %w{ David A. Black }
=> ["David", "A.", "Black"]
```

If any string in the list contains a whitespace character, you need to escape that character with a backslash:

```
>> %w{ David\ A.\ Black is a Rubyist. }
=> ["David A. Black", "is", "a", "Rubyist."]
```

The strings in the list are parsed as single-quoted strings. But if you need double-quoted strings, you can use %W instead of %w:

```
>> %W{ David is #{2008 - 1959} years old. }
=> ["David", "is", "49", "years", "old."]
```

Let's proceed now to the matter of handling array elements.

## 9.2.2 Inserting, retrieving, and removing array elements

An array is a numerically ordered collection. Any object you add to the array goes at the beginning, at the end, or somewhere in the middle. The most general technique for inserting one or more items into an array is the setter method []= (square brackets and equal sign). This looks odd as a method name in the middle of a paragraph like this, but thanks to its syntactic sugar equivalent, []= works smoothly in practice.

In order to use []=, you need to know that each item (or element) in an array occupies a numbered position. The first element is at position *zero* (not position *one*). The second element is at position one, and so forth.

To insert an element with the []= method—using the syntactic sugar that allows you to avoid the usual method-calling dot—you do this:

```
a = []
a[0] = "first"
```

The second line is syntactic sugar for a.[0] = ("first"). In this example, you end up with a one-element array whose first (and only) element is the string "first".

When you have objects in an array, you can *retrieve* those objects by using the [] method, which is the getter equivalent of the []= setter method:

```
a = [1, 2, 3, 4, 5]
p a[2]
```

In this case, the second line is syntactic sugar for a.[2]. You're asking for the third element (based on the zero-origin indexing), which is the integer 3.

You can also perform these get-and-set methods on more than one element at a time.

### SETTING OR GETTING MORE THAN ONE ARRAY ELEMENT AT A TIME

If you give either Array#[ ] or Array#[ ]= (the get or set method) a second argument, it's treated as a length—a number of elements to set or retrieve. In the case of retrieval, the results are returned inside a new array.

Here's some irb dialogue, illustrating the multi-element operations of the [] and []= methods:

```

>> a = ["red", "orange", "yellow", "purple", "gray", "indigo", "violet"]
=> ["red", "orange", "yellow", "purple", "gray", "indigo", "violet"]
>> a[3,2] 1
=> ["purple", "gray"]
>> a[3,2] = "green", "blue" 2
=> ["green", "blue"]
>> a
=> ["red", "orange", "yellow", "green", "blue", "indigo", "violet"] 3

```

After initializing the array `a`, we grab 1 two elements, starting at index 3 (the fourth element) of `a`. The two elements are returned in an array. Next, we set the fourth and fifth elements, using the `[3,2]` notation 2, to new values; these new values are then present in the whole array 3 when we ask `irb` to display it at the end.

There's a synonym for the `[]` method: `slice`. Like `[]`, `slice` takes two arguments: a starting index and an optional length. In addition, a method called `slice!` removes the sliced items permanently from the array.

Another technique for extracting multiple array elements is the `values_at` method. `values_at` takes one or more arguments representing indexes and returns an array consisting of the values stored at those indexes in the receiver array:

```

array = ["the", "dog", "ate", "the", "cat"]
articles = array.values_at(0,3)
p articles ← Output: ["the", "the"]

```

You can perform set and get operations on elements anywhere in an array. But operations affecting, specifically, the beginnings and ends of arrays crop up most often. Accordingly, a number of methods exist for the special purpose of adding items to or removing them from the beginning or end of an array, as we'll now see.

#### **SPECIAL METHODS FOR MANIPULATING THE BEGINNINGS AND ENDS OF ARRAYS**

To add an object to the beginning of an array, you can use `unshift`. After this operation

```

a = [1,2,3,4]
a.unshift(0)

```

the array `a` now looks like this: `[0,1,2,3,4]`.

To add an object to the end of an array, you use `push`. Doing this

```

a = [1,2,3,4]
a.push(5)

```

results in the array `a` having a fifth element: `[1,2,3,4,5]`.

You can also use a method called `<<` (two less-than signs), which places an object on the end of the array. Like many methods whose names resemble operators, `<<` offers the syntactic sugar of usage as an infix operator. The following code adds 5 as the fifth element of `a`, just like the `push` operation in the last example:

```

a = [1,2,3,4]
a << 5

```

The methods `<<` and `push` differ in that `push` can take more than one argument. This code

```
a = [1,2,3,4,5]
a.push(6,7,8)
```

adds three elements to `a`, resulting in `[1,2,3,4,5,6,7,8]`.

Corresponding to `unshift` and `push` are their opposite numbers, `shift` and `pop`. `shift` removes one object from the beginning of the array (thereby “shifting” the remaining objects to the left by one position). `pop` removes an object from the end of the array. `shift` and `pop` both return the array element they have removed, as this example shows:

```
a = [1,2,3,4,5]
print "The original array: "
p a
popped = a.pop
print "The popped item: "
puts popped
print "The new state of the array: "
p a
shifted = a.shift
print "The shifted item: "
puts shifted
print "The new state of the array: "
p a
```

The output is as follows:

```
The original array: [1, 2, 3, 4, 5]
The popped item: 5
The new state of the array: [1, 2, 3, 4]
The shifted item: 1
The new state of the array: [2, 3, 4]
```

As you can see from the running commentary in the output, the return value of `pop` and `shift` is the item that was removed from the array. The array is permanently changed by these operations; the elements are removed, not just referred to or captured.

We’ll turn next from manipulating one array to looking at ways to combine two or more arrays.

### 9.2.3 Combining arrays with other arrays

Several methods allow you to combine multiple arrays in various ways—something that, it turns out, is common and useful when you begin manipulating lots of data in lists. Remember that in every case, even though you’re dealing with two (or more) arrays, *one* array is always the receiver of the message. The other arrays involved in the operation are arguments to the method.

To add the contents of one array to another array, you can use `concat`:

```
>> [1,2,3].concat([4,5,6])
=> [1, 2, 3, 4, 5, 6]
```

Note that `concat` differs in an important way from `push`. Try replacing `concat` with `push` in the example, and see what happens.

concat permanently changes the contents of its receiver. If you want to combine two arrays into a third, new array, you can do so with the + method:

```
>> a = [1,2,3]
=> [1, 2, 3]
>> b = a + [4,5,6]
=> [1, 2, 3, 4, 5, 6]
>> a
=> [1, 2, 3] ❶
```

The receiver of the + message—in this case, the array a—remains unchanged by the operation (as irb tells you ❶).

Another useful array-combining method, at least given a fairly liberal interpretation of the concept of “combining,” is replace. As the name implies, replace replaces the contents of one array with the contents of another:

```
>> a = [1,2,3]
=> [1, 2, 3]
>> a.replace([4,5,6]) ❶
=> [4, 5, 6]
>> a
=> [4, 5, 6]
```

The original contents of a are gone, replaced ❶ by the contents of the argument array [4,5,6]. Remember that a replace operation is different from reassignment. If you do this

```
a = [1,2,3]
a = [4,5,6]
```

the second assignment causes the variable a to refer to a completely different array object than the first. That’s not the same as replacing the elements of the *same* array object. This starts to matter, in particular, when you have another variable that refers to the original array, as in this code:

```
>> a = [1,2,3]
=> [1, 2, 3]
>> b = a ❶
=> [1, 2, 3]
>> a.replace([4,5,6])
=> [4, 5, 6]
>> b ❷
=> [4, 5, 6]
>> a = [7,8,9] ❸
=> [7, 8, 9]
>> b
=> [4, 5, 6] ❹
```

Once you’ve performed the assignment of a to b ❶, *replacing* the contents of a means you’ve replaced the contents of b ❷, because the two variables refer to the same array. But when you reassign to a ❸, you break the binding between a and the array; a and b now refer to different array objects: b to the same old array ❹, a to a new one.

In addition to combining multiple arrays, you can also transform individual arrays to different forms. We’ll look next at techniques along these lines.

### 9.2.4 Array transformations

A useful array transformation is `flatten`, which does an un-nesting of inner arrays. Furthermore, you can specify how many levels of flattening you want, with the default being the full un-nesting.

Here's a triple-nested array being flattened by various levels:

```
>> array = [1,2,[3,4,[5]],[6,[7,8]]]
=> [1, 2, [3, 4, [5]], [6, [7, 8]]]
>> array.flatten      ← Flattens completely
=> [1, 2, 3, 4, 5, 6, 7, 8]
>> array.flatten(1)   ← Flattens by one level
=> [1, 2, 3, 4, [5], [6, [7, 8]]]
>> array.flatten(2)
=> [1, 2, 3, 4, 5, 6, [7, 8]] ← Flattens by two levels
```

There's also an in-place `flatten!` method, which makes the change permanently in the array.

Another array-transformation method is `reverse`, which does exactly what it says:

```
>> [1,2,3,4].reverse
=> [4, 3, 2, 1]
```

Like its string counterpart, `Array#reverse` also has a bang (!) version, which permanently reverses the array that calls it.

Another important array-transformation method is `join`. The return value of `join` isn't an array but a string, consisting of the string representation of all the elements of the array strung together:

```
>> ["abc", "def", 123].join
=> "abcdef123"
```

`join` takes an optional argument; if given, the argument is placed between each pair of elements:

```
>> ["abc", "def", 123].join(", ")
=> "abc, def, 123"
```

Joining with commas (or comma-space, as in the last example) is a fairly common operation.

You can also transform an array with `uniq`. `uniq` gives you a new array, consisting of the elements of the original array with all duplicate elements removed:

```
>> [1,2,3,1,4,3,5,1].uniq
=> [1, 2, 3, 4, 5]
```

Duplicate status is determined by testing pairs of elements with the `==` method. Any two elements for which the `==` test returns true are considered duplicates of each other. `uniq` also has a bang version, `uniq!`, which removes duplicates permanently from the original array.

Sometimes you have an array that includes one or more occurrences of `nil`, and you want to get rid of them. You might, for example, have an array of the zip codes of

all the members of an organization. But maybe some of them don't have zip codes. If you want to do a histogram on the zip codes, you'd want to get rid of the `nil` ones first.

You can do this with the `compact` method. This method returns a new array identical to the original array, except that all occurrences of `nil` have been removed:

```
>> zip_codes = ["06511", "08902", "08902", nil, "10027",
"08902", nil, "06511"]
=> ["06511", "08902", "08902", nil, "10027", "08902", nil, "06511"]
>> zip_codes.compact
=> ["06511", "08902", "08902", "10027", "08902", "06511"]
```

Once again, there's a bang version (`compact!`) available.

In addition to transforming arrays in various ways, you can query arrays on various criteria.

### 9.2.5 Array querying

Several methods allow you to gather information about an array from the array. Table 9.1 summarizes some of them. Other query methods arise from `Array`'s inclusion of the `Enumerable` module and will therefore come into view in the next chapter.

**Table 9.1** Summary of common array query methods

Method name/sample call	Meaning
<code>a.size</code> (synonym: <code>length</code> )	Number of elements in the array
<code>a.empty?</code>	True if <code>a</code> is an empty array; false if it has any elements
<code>a.include?(item)</code>	True if the array includes <code>item</code> ; false otherwise
<code>a.count(item)</code>	Number of occurrences of <code>item</code> in array
<code>a.first(n=1)</code>	First <code>n</code> elements of array
<code>a.last(n=1)</code>	Last <code>n</code> elements of array

Next up: hashes. They've crossed our path here and there along the way, and now we'll look at them in detail.

## 9.3 Hashes

Like an array, a hash is a collection of objects. A hash consists of *key/value* pairs, where any key and any value can be any Ruby object. Hashes let you perform lookup operations based on keys. In addition to simple key-based value retrieval, you can also perform more complex filtering and selection operations.

A typical use of a hash is to store complete strings along with their abbreviations. Here's a hash containing a selection of names and two-letter state abbreviations, along with some code that exercises it. The `=>` operator connects a key on the left with the value corresponding to it on the right:

```
state_hash = { "Connecticut" => "CT",
              "Delaware"    => "DE",
```

```

    "New Jersey" => "NJ",
    "Virginia"   => "VA" }

print "Enter the name of a state: "
state = gets.chomp
abbr = state_hash[state]
puts "The abbreviation is #{abbr}."

```

When you run this snippet (assuming you enter one of the states defined in the hash), you see the abbreviation.

Hashes remember the insertion order of their keys. Insertion order isn't always terribly important; one of the merits of a hash is that it provides quick lookup in better than linear time. And in many cases, items get added to hashes in no particular order; ordering, if any, comes later, when you want to turn (say) a hash of names and birthdays that you've created over time into a chronologically or alphabetically sorted array. Still, however useful it may or may not be for them to do so, hashes remember their key-insertion order and observe that order when you iterate over them or examine them.

Like arrays, hashes can be created in several different ways.

### 9.3.1 **Creating a new hash**

There are three ways to create a hash: with the literal constructor (curly braces), with the `Hash.new` method, and with the `Hash.[]` method (a square-bracket class method of `Hash`).

When you type out a literal hash inside curly braces, you separate keys from values with the `=>` operator (unless you're using the special `{ key: value }` syntax for symbol keys). After each complete key/value pair (except the last pair, where it's optional), you put a comma.

The literal hash constructor is convenient when you have values you wish to hash that aren't going to change; you'll type them into the program file once and refer to them from the program. State abbreviations are a good example.

You can use the literal hash constructor to create an empty hash:

```
h = {}
```

You'd presumably want to add items to the empty hash at some point; techniques for doing so will be forthcoming in section 9.3.2.

The second way to create a hash is with the traditional `new` constructor:

```
Hash.new
```

This always creates an empty hash. But if you provide an argument to `Hash.new`, it's treated as the default value for nonexistent hash keys. (We'll return to the matter of default values after looking at key/value insertion and retrieval.)

The third way to create a hash involves another class method of the `Hash` class: the method `[]` (square brackets). This method takes a comma-separated list of items and, assuming there's an even number of arguments, treats them as alternating keys and values, which it uses to construct a hash. Thanks to Ruby's syntactic sugar, you can

put the arguments to `[]` directly inside the brackets and dispense with the method-calling dot:

```
>> Hash["Connecticut", "CT", "Delaware", "DE" ]
=> {"Connecticut"=>"CT", "Delaware"=>"DE" }
```

If you provide an odd number of arguments, a fatal error is raised, because an odd number of arguments can't be mapped to a series of key/value pairs.

Now, let's turn to the matter of manipulating a hash's contents. We'll follow much the same path as we did with arrays, looking at insertion and retrieval operations, combining hashes with other hashes, hash transformations, and querying hashes. We'll also take a separate look, along the way, at setting default values for nonexistent hash keys.

### 9.3.2 **Inserting, retrieving, and removing hash pairs**

As you'll see as we proceed, hashes have a lot in common with arrays when it comes to the get- and set-style operations—although important differences and techniques are specific to each.

#### **ADDING A KEY/VALUE PAIR TO A HASH**

To add a key/value pair to a hash, you use essentially the same technique as for adding an item to an array: the `[]=` method plus syntactic sugar.

To add a state to `state_hash`, you do this

```
state_hash["New York"] = "NY"
```

which is the sugared version of this:

```
state_hash.[]=("New York", "NY")
```

You can also use the synonymous method `store` for this operation. `store` takes two arguments (a key and a value):

```
state_hash.store("New York", "NY")
```

When you're adding to a hash, keep in mind the important principle that *keys are unique*. You can have only one entry with a given key. Hash *values* don't have to be unique; you can assign the same value to two or more keys. But you can't have duplicate keys.

If you add a key/value pair to a hash that already has an entry for the key you're adding, the old entry is overwritten. Here's an example:

```
h = Hash.new
h["a"] = 1
h["a"] = 2
puts h["a"]      ← Output: 2
```

This code assigns two values to the "a" key of the hash `h`. The second assignment clobbers the first, as the `puts` statement shows by outputting 2.

If you reassign to a given hash key, that key still maintains its place in the insertion order of the hash. The change in the value paired with the key isn't considered a new insertion into the hash.

**RETRIEVING VALUES FROM A HASH**

The workhorse technique for retrieving hash values is the `[]` method. For example, to retrieve “CT” from `state_hash` and assign it to a variable, you’d do this:

```
conn_abbrev = state_hash["Connecticut"]
```

Using a hash key is much like indexing an array—except that the index (the key) can be anything, whereas in an array it’s always an integer.

Hashes also have a `fetch` method, which gives you an alternative way of retrieving values by key:

```
conn_abbrev = state_hash.fetch("Connecticut")
```

`fetch` differs from `[]` in the way it behaves when you ask it to look up a nonexistent key: `fetch` raises an exception, whereas `[]` gives you either `nil` or a default you’ve specified (as discussed in a moment).

You can also retrieve values for multiple keys in one operation, with `values_at`:

```
two_states = state_hash.values_at("New Jersey", "Delaware")
```

This code returns an array consisting of `["NJ", "DE"]` and assigns it to the variable `two_states`.

Now that you have a sense of the mechanics of getting information into and out of a hash, let’s circle back and look at the matter of supplying a default value (or default code block) when you create a hash.

**9.3.3 Specifying default hash values and behavior**

By default, when you ask a hash for the value corresponding to a nonexistent key, you get `nil`:

```
>> h = Hash.new
=> {}
>> h["no such key!"]
=> nil
```

But you can specify a *different* default value by supplying an argument to `Hash.new`:

```
>> h = Hash.new(0)
=> {}
>> h["no such key!"]
=> 0
```

Here, we get back the hash’s default value, `0`, when we use a nonexistent key. (You can also set the default on an already existing hash, with the `default` method.)

It’s important to remember that whatever you specify as the default value is what you get when you specify a *nonexistent* key—and that the key remains nonexistent until you assign a value to it. In other words, saying `h["blah"]` doesn’t mean that `h` now has a “blah” key. If you want that key in the hash, you have to put it there. You can verify the fact that the hash `h` has no keys by examining it after performing the nonexistent key lookup in the last example:

```
>> h
=> {}
```

If you want references to nonexistent keys to cause the keys to come into existence, you can so arrange by supplying a code block to `Hash.new`. The code block will be executed every time a nonexistent key is referenced. Furthermore, two objects will be yielded to the block: the hash and the (nonexistent) key.

This technique gives you a foot in the door when it comes to setting keys automatically when they're first used. It's not the most elegant or streamlined technique in all of Ruby, but it does work. You write a block that grabs the hash and the key, and you do a set operation.

For example, if you want every nonexistent key to be added to the hash with a value of 0, you create your hash like this:

```
h = Hash.new {|hash, key| hash[key] = 0 }
```

When the hash `h` is asked to retrieve the value for a key it doesn't have, the block is executed with `hash` set to the hash itself and `key` set to the nonexistent key. And thanks to the code in the block, the key is added to the hash after all, with the value 0.

Given this assignment of a new hash to `h`, you can trigger the block like this:

```
>> h["new key! "]      ❶
=> 0
>> h                  ❷
=> {"new key!" => 0}
```

When you try to look up the key `"new key!"` ❶, it's not there; but thanks to the block, it gets added, with the value 0. Next, when you ask irb to show you the whole hash ❷, it contains the automatically added pair.

This technique has lots of uses. It lets you make assumptions about what's in a hash, even if nothing is there to start with. It also shows you another facet of Ruby's extensive repertoire of dynamic programming techniques and the flexibility of hashes.

We'll turn now to ways you can combine hashes with each other, as we did with strings and arrays.

### 9.3.4 **Combining hashes with other hashes**

The process of combining two hashes into one comes in two flavors: the destructive flavor, where the first hash has the key/value pairs from the second hash added to it directly; and the nondestructive flavor, where a new, third hash is created that combines the elements of the original two.

The destructive operation is performed with the `update` method. Entries in the first hash are overwritten permanently if the second hash has a corresponding key:

```
h1 = {"Smith" => "John",
      "Jones" => "Jane" }
h2 = {"Smith" => "Jim" }
h1.update(h2)
puts h1["Smith"]      ← Output: Jim
```

In this example, h1’s “Smith” entry has been changed (updated) to the value it has in h2. You’re asking for a refresh of your hash to reflect the contents of the second hash. That’s the destructive version of combining hashes.

To perform nondestructive combining of two hashes, you use the `merge` method, which gives you a third hash and leaves the original unchanged:

```
h1 = { "Smith" => "John",
      "Jones" => "Jane" }
h2 = { "Smith" => "Jim" }
h3 = h1.merge(h2)
p h1["Smith"]      ←— Output: John
```

Here, h1’s Smith/John pair isn’t overwritten by h2’s Smith/Jim pair. Instead, a new hash is created, with pairs from both of the other two.

Note that h3 has a decision to make: which of the two `Smith` entries should it contain? The answer is that when the two hashes being merged share a key, the second hash (h2, in this example) wins. h3’s value for the key “Smith” will be “Jim”.

Incidentally, `merge!`—the bang version of `merge`—is a synonym for `update`. You can use either name when you want to perform that operation.

In addition to being combined with other hashes, hashes can also be transformed in a number of ways, as you’ll see next.

### 9.3.5 Hash transformations

You can perform several transformations on hashes. *Transformation*, in this context, means that the method is called on a hash, and the result of the operation (the method’s return value) is a hash. In chapter 10, you’ll see other filtering and selecting methods on hashes that return their result sets in arrays. Here, we’re looking at hash-to-hash operations.

#### INVERTING A HASH

`Hash#invert` flips the keys and the values. Values become keys, and keys become values:

```
>> h = { 1 => "one", 2 => "two" }
=> {1=>"one", 2=>"two"}
>> h.invert
=> {"two"=>2, "one"=>1}
```

Be careful when you invert hashes. Because hash keys are unique, but values aren’t, when you turn duplicate values into keys, one of the pairs is discarded:

```
>> h = { 1 => "one", 2 => "more than 1", 3 => "more than 1" }
=> {1=>"one", 2=>"more than 1", 3=>"more than 1"}
>> h.invert
=> {"one"=>1, "more than 1"=>3}
```

Only one of the two “more than 1” values can survive as a key when the inversion is performed; the other is discarded. You should invert a hash only when you’re certain the values as well as the keys are unique.

**CLEARING A HASH**

Hash#clear empties the hash:

```
>> {1 => "one", 2 => "two" }.clear
=> {}
```

This is an in-place operation: the empty hash is the same hash (the same object) as the one to which you send the `clear` message.

**REPLACING THE CONTENTS OF A HASH**

Like strings and arrays, hashes have a `replace` method:

```
>> { 1 => "one", 2 => "two" }.replace({ 10 => "ten", 20 => "twenty"})
=> {10 => "ten", 20 => "twenty"}
```

This is also an in-place operation, as the name `replace` implies.

We'll turn next to hash query methods.

**9.3.6 Hash querying**

Like arrays (and many other Ruby objects), hashes provide a number of methods with which you can query the state of the object. Table 9.2 shows some common hash query methods.

**Table 9.2 Common hash query methods and their meanings**

Method name/sample call	Meaning
<code>h.has_key?(1)</code>	True if <code>h</code> has the key 1
<code>h.include?(1)</code>	Synonym for <code>has_key?</code>
<code>h.key?(1)</code>	Synonym for <code>has_key?</code>
<code>h.member?(1)</code>	Another synonym for <code>has_key?</code>
<code>h.has_value?("three")</code>	True if any value in <code>h</code> is "three"
<code>h.value?("three")</code>	Synonym for <code>has_value?</code>
<code>h.empty?</code>	True if <code>h</code> has no key/value pairs
<code>h.size</code>	Number of key/value pairs in <code>h</code>

None of the methods in table 9.2 should offer any surprises at this point; they're similar in spirit, and in some cases in letter, to those you've seen for arrays. With the exception of `size`, they all return either `true` or `false`. The only surprise may be how many of them are synonyms. Four methods test for the presence of a particular key: `has_key?`, `include?`, `key?`, and `member?`. A case could be made that this is two or even three synonyms too many. `has_key?` seems to be the most popular of the four and is the most to-the-point with respect to what the method tests for.

The `has_value?` method has one synonym: `value?`. As with its key counterpart, `has_value?` seems to be more popular.

The other methods—`empty?` and `size`—tell you whether the hash is empty and what its size is. (`size` can also be called as `length`.) The size of a hash is the number of key/value pairs it contains.

As simple as their underlying premise may be, hashes are a powerful data structure. Among other uses, you'll see them a lot as method arguments. There's nothing surprising about that, but there's something interesting about the special allowances Ruby makes for hash notation in argument lists.

### 9.3.7 Hashes as method arguments

If you call a method in such a way that the *last* argument in the argument list is a hash, Ruby allows you to write the hash without curly braces. This perhaps trivial-sounding special rule can, in practice, make argument lists look much nicer than they otherwise would.

Here's an example, involving a method called `link_to` which is part of the Ruby on Rails web-application development framework:

```
link_to "Click here",
      :controller => "work",
      :action     => "show",
      :id         => work.id
```

What you're seeing is a method call with two arguments: the string "Click here" and *a three-key hash*. Written with the curly braces and parentheses, the same method call would look like this:

```
link_to("Click here", { :controller => "work",
                       :action     => "show",
                       :id         => work.id })
```

But because the hash is the last thing in the argument list, the curly braces are optional. So are the parentheses (as they generally are for simple method calls where there's no danger of ambiguity). The result, as the first version of the `link_to` call shows, is a method call that almost looks more like a stanza from a configuration file than a method call.

The use of hashes as method arguments, in any position in the argument list, also confers the benefit of a kind of pseudo-keyword argument syntax. In the `link_to` example, you can see at a glance what each key/value pair is. If `link_to` didn't use a hash argument to convey the information, it would look more like this

```
link_to "Click here", "work", "show", work.id
```

#### Hashes as first arguments

In addition to learning about the special syntax available to you for using hashes as final method arguments without curly braces, it's worth noting a pitfall of using a hash as the *first* argument to a method. The rule in this case is that you must not only put curly braces around the hash but also put the entire argument list in parentheses. If you don't, Ruby will think your hash is a code block. In other words, when you do this

```
my_method { "NY" => "New York" }, 100, "another argument"
```

Ruby interprets the expression in braces as a block. If you want to send a hash along as an argument in this position, you have to use parentheses around the entire argument list to make it clear that the curly braces are hash-related and not block-related.

and you'd have to know the exact order of the arguments. (When you use a hash, you can list the key/value pairs in any order.)

The method, of course, has to “unwrap” the hash and use the keys appropriately. Let's take another example:

```
add_to_city_database("New York City", :state => "New York",
                    :population => 7000000,
                    :nickname => "Big Apple")
```

In a case like this, you can make use of the special syntactic sugar for hashes whose keys are symbols:

```
add_to_city_database("New York City", state: "New York",
                    population: 7000000,
                    nickname: "Big Apple")
```

You'd probably want the name of the city to be part of the hash too. It's separated out in this example just to make the point that the hash can be, and often is, the last of several arguments.

The method `add_to_city_database` has to do a little more work to gain access to the data being passed to it than it would if it were binding parameters to arguments in left-to-right order through a list:

```
def add_to_city_database(name, *info)
  c = City.new
  c.name = name
  c.state = info[:state]
  c.population = info[:population]
  # etc.
```

Of course, the exact process involved in unwrapping the hash will vary from one such case to another. (Perhaps `City` objects store their information as a hash; that would make the method's job a little easier.) But one way or another, the method has to handle the hash.

And that tends to be the tradeoff when you're deciding whether to use a hash as a method argument: convenience on the calling end versus more work on the method end.

Finally, keep in mind that although you get to leave the curly braces off the hash literal when it's the last thing in an argument list, you can have as many hashes as you wish as method arguments, in any position. Just remember that it's only when a hash is in final argument position that you're allowed to dispense with the braces.

We'll look next at ranges—which aren't exactly collection objects, arguably, but which turn out to have a lot in common with collection objects.

## 9.4 **Ranges**

A *range* is an object with a start point and an end point. The semantics of range operations involve two major concepts:

- *Inclusion*—Does a given value fall inside the range?
- *Enumeration*—The range is treated as a traversable collection of individual items.

The logic of inclusion applies to all ranges; you can always test for inclusion. The logic of enumeration kicks in only with certain ranges—namely, those that include a finite number of discrete, identifiable values. You can't iterate over a range that lies between two floating-point numbers, because the range encompasses an infinite number of values. But you can iterate over a range between two integers.

We'll save further analysis of range iteration and enumeration logic for the next chapter, where we'll be looking at enumeration and the `Enumerable` module in depth. In this section, we'll look primarily at the other semantic concept: inclusion logic. We'll start with some range-creation techniques.

### 9.4.1 Creating a range

You can create range objects with `Range.new`. If you do so in `irb`, you're rewarded with a view of the syntax for literal range construction:

```
>> r = Range.new(1,100)
=> 1..100
```

The literal syntax can, of course, also be used directly to create a range:

```
>> r = 1..100
=> 1..100
```

When you see a range with two dots between the start-point and end-point values, as in the previous example, you're seeing an *inclusive* range. A range with three dots in the middle is an *exclusive* range.

```
>> r = 1...100
=> 1...100
```

The difference lies in whether the end point is considered to lie inside the range. Coming full circle, you can also specify inclusive or exclusive behavior when you create a range with `Range.new`: the default is an inclusive range, but you can force an exclusive range by passing a third argument of `true` to the constructor:

```
>> Range.new(1,100)
=> 1..100
>> Range.new(1,100,true)
=> 1...100
```

Unfortunately, there's no way to remember which behavior is the default and which is triggered by the `true` argument except to memorize it.

Also notoriously hard to remember is the matter of which number of dots goes with which type of range.

**REMEMBERING .. VS. ...**

If you follow Ruby discussion forums, you’ll periodically see messages and posts from people who find it difficult to remember which is which: two versus three dots, inclusive versus exclusive range.

One way to remember is to think of a range as always reaching to the point represented by whatever follows the second dot. In an inclusive range, the point after the second dot is the end value of the range. In this example, the value 100 is included in the range:

```
1..100
```

But in this exclusive range, the value 100 lies *beyond* the effective end of the range:

```
1...100
```

In other words, you can think of 100 as having been “pushed” to the right in such a way that it now sits outside the range.

We’ll turn now to range-inclusion logic—a section that closely corresponds to the “query” sections from the discussions of strings, arrays, and hashes, because most of what you do with ranges involves querying them on criteria of inclusion.

**9.4.2 Range-inclusion logic**

Ranges have `begin` and `end` methods, which report back their starting and ending points:

```
>> r = 1..10
=> 1..10
>> r.begin
=> 1
>> r.end
=> 10
```

A range also knows whether it’s an exclusive (three-dot) range:

```
>> r.exclude_end?
=> false
```

With the goal posts in place, you can start to test for inclusion.

Two methods are available for testing inclusion of a value in a range: `cover?` and `include?` (which is also aliased as `member?`).

**TESTING RANGE INCLUSION WITH COVER?**

The `cover?` method performs a simple test: if the argument to the method is greater than the range’s start point and less than its end point (or equal to it, for an inclusive range), then the range is said to *cover* the object. The tests are performed using boolean comparison tests, with a result of `false` in cases where the comparison makes no sense.

All of the following comparisons do make sense; one of them fails because the item isn’t in the range:

```

>> r = "a".."z"
=> "a".."z"
>> r.cover?("a")      ← true: "a" >= "a" and "a" <= "z"
=> true
>> r.cover?("abc")   ← true: "abc" >= "a" and "abc" <= "z"
=> true
>> r.cover?("A")     ← false: "A" < "a"
=> false

```

But this test fails because the item being tested for inclusion isn't comparable with the range's start and end points:

```

>> r.cover?([])
=> false

```

It's meaningless to ask whether an array is greater than the string "a". If you try such a comparison on its own, you'll get a fatal error. Fortunately, ranges take a more conservative approach and tell you that the item isn't covered by the range.

Whereas `cover?` performs start- and end-point comparisons, the other inclusion test, `include?` (or `member?`), takes a more collection-based approach.

#### TESTING RANGE INCLUSION WITH INCLUDE?

The `include?` test treats the range as a kind of crypto-array—that is, a collection of values. The "a".."z" range, for example, is considered to include (as measured by `include?`) only the 26 values that lie inclusively between "a" and "z"

Therefore, `include?` produces results that differ from those of `cover?`:

```

>> r.include?("a")
=> true
>> r.include?("abc")
=> false

```

In cases where the range can't be interpreted as a finite collection, such as a range of floats, the `include?` method falls back on numerical order and comparison:

```

>> r = 1.0..2.0
=> 1.0..2.0
>> r.include?(1.5)
=> true

```

#### Are there backward ranges?

The anticlimactic answer to the question of backward ranges is this: yes and no. You can create a backward range, but it won't do what you probably want it to:

```

>> r = 100...1
=> 100...1
>> r.include?(50)
=> false

```

The range happily performs its usual inclusion test for you. The test calculates whether the candidate for inclusion is greater than the start point of the range and less than the end point. Because 50 is neither greater than 100 nor less than 1, the test fails. And it fails silently; this is a logic error, not a fatal syntax or runtime error. It's worth making a mental note of the fact that ranges only go one way.

**Are there backward ranges?(continued)**

As you've seen, backward ranges do show up in one particular set of use cases: as index arguments to strings and arrays. They typically take the form of a positive start point and a negative end point, with the negative end point counting in from the right:

```
>> "This is a sample string"[10..-5]
=> "sample st"
>> ['a', 'b', 'c', 'd'][0..-2]
=> ["a", "b", "c"]
```

You can even use an exclusive backward range:

```
>> ['a', 'b', 'c', 'd'][0...-2]
=> ["a", "b"]
```

In these cases, what doesn't work (at least, the way you might have expected) in a range on its own does work when applied to a string or an array.

You'll see more about ranges as quasi-collections in the next chapter, as promised. In this chapter, we've got one more basic collection class to examine: the `Set` class.

**9.5 Sets**

`Set` is the one class under discussion in this chapter that isn't, strictly speaking, a Ruby core class. It's a standard library class, which means that in order to use it, you have to do this:

```
require 'set'
```

The general rule in this book is that we're looking at the core language rather than the standard library; but the `Set` class makes a worthy exception because it fits in so nicely with the other container and collection classes we've looked at.

A *set* is a unique collection of objects. The objects can be anything—strings, integers, arrays, other sets—but no object can occur more than once in the set. Uniqueness is also enforced at the common-sense content level: if the set contains the string "New York", you can't add the string "New York" to it, even though the two strings may technically be different objects. The same is true of arrays with equivalent content.

**NOTE** Internally, sets use a hash to enforce the uniqueness of their contents. When an element is added to a set, the internal hash for that set gets a new key. Therefore, any two objects that would count as duplicates if used as hash keys can't occur together in a set.

Let's look now at how to create sets.

**9.5.1 Set creation**

To create a set, you use the `Set.new` constructor. You can create an empty set, or you can pass in a collection object. In the latter case, all the elements of the collection are placed individually in the set:

```
>> new_england = ["Connecticut", "Maine", "Massachusetts",
                  "New Hampshire", "Rhode Island", "Vermont"]
=> ["Connecticut", "Maine", "Massachusetts",
    "New Hampshire", "Rhode Island", "Vermont"]
>> state_set = Set.new(new_england)
=> #<Set: {"Connecticut", "Maine", "Massachusetts",
          "New Hampshire", "Rhode Island", "Vermont"}>
```

Here, we've created an array, `new_england`, and used it as the constructor argument for the creation of the `state_set` set. Note that there's no literal set constructor (no equivalent to `[]` for arrays or `{}` for hashes). There can't be: sets are part of the standard library, not the core, and the core syntax of the language is already in place before the set library gets loaded.

You can also provide a code block to the constructor, in which case every item in the collection object you supply is passed through the block (yielded to it) with the resulting value being inserted into the set. For example, here's a way to initialize a set to a list of uppercased strings:

```
>> names = ["David", "Yukihiro", "Chad", "Amy"]
=> ["David", "Yukihiro", "Chad", "Amy"]
>> name_set = Set.new(names) { |name| name.upcase }
=> #<Set: {"AMY", "YUKIHIRO", "CHAD", "DAVID"}>
```

Rather than using the array of names as its initial values, the set constructor yields each name to the block and inserts what it gets back (an uppercase version of the string) into the set.

Now that we've got a set, we can manipulate it.

### 9.5.2 Manipulating set elements

Like arrays, sets have two modes of adding elements: either inserting a new element into the set or drawing on another collection object as a source for multiple new elements. In the array world, this is the difference between `push` and `concat`. For sets, the distinction is reflected in a variety of methods, which we'll look at here.

#### ADDING/REMOVING ONE OBJECT TO/FROM A SET

To add a single object to a set, you can use the `<<` operator/method:

```
>> tri_state = Set.new(["New Jersey", "New York"])
=> #<Set: {"New Jersey", "New York"}>           ← Whoops, only two!
>> tri_state << "Connecticut"                  ← Adds third
=> #<Set: {"New Jersey", "New York", "Connecticut"}>
```

Here, as with arrays, strings, and other objects, `<<` connotes appending to a collection or mutable object. If you try to add an object that's already in the set (or an object that's content-equal to one that's in the set), nothing happens:

```
>> tri_state << "Connecticut"                  ← Second time
=> #<Set: {"New Jersey", "New York", "Connecticut"}>
```

To remove an object, use `delete`:

```
>> tri_state << "Pennsylvania"
=> #<Set: {"New Jersey", "New York", "Connecticut", "Pennsylvania"}>
```

```
>> tri_state.delete("Connecticut")
=> #<Set: {"New Jersey", "New York", "Pennsylvania"}>
```

Deleting an object that isn't in the set doesn't raise an error. As with adding a duplicate object, nothing happens.

The `<<` method is also available as `add`. There's also a method called `add?`, which differs from `add` in that it returns `nil` (rather than returning the set itself) if the set is unchanged after the operation:

```
>> tri_state.add?("Pennsylvania")
=> nil
```

You can test the return value of `add?` to determine whether to take a different conditional branch if the element you've attempted to add was already there.

### **SET INTERSECTION, UNION, AND DIFFERENCE**

Sets have a concept of their own intersection, union, and difference with other sets—and, indeed, with other enumerable objects. The `Set` class comes with the necessary methods to perform these operations.

These methods have English names, and also symbolic aliases. The names are

- intersection, aliased as `&`
- union, aliased as `+` and `|`
- difference, aliased as `-`

Each of these methods returns a new set consisting of the original set plus or minus the appropriate elements from the object provided as the method argument. The original set is unaffected.

Let's shift our tri-state grouping back to the East and look at some set operations:

```
>> tri_state = Set.new(["Connecticut", "New Jersey", "New York"])
=> #<Set: {"Connecticut", "New Jersey", "New York"}>

# Subtraction (difference/-)
>> state_set - tri_state
=> #<Set: {"Maine", "Massachusetts", "New Hampshire", "Rhode Island",
"Vermont"}>

# Addition (union/+/)
>> state_set + tri_state
=> #<Set: {"Connecticut", "Maine", "Massachusetts", "New Hampshire",
"Rhode Island", "Vermont", "New Jersey", "New York"}>

# Intersection (&)
>> state_set & tri_state
=> #<Set: {"Connecticut"}>

>> state_set | tri_state
=> #<Set: {"Connecticut", "Maine", "Massachusetts", "New Hampshire",
"Rhode Island", "Vermont", "New Jersey", "New York"}>
```

There's also an exclusive-or operator, `^`, which you can use to take the exclusive union between a set and an enumerable—that is, a set consisting of all elements that occur in either the set or the enumerable but not both:

```
>> state_set ^ tri_state
=> #<Set: {"New Jersey", "New York", "Maine", "Massachusetts",
  "New Hampshire", "Rhode Island", "Vermont"}>
```

You can extend an existing set using a technique very similar in effect to the `Set.new` technique: the `merge` method, which can take as its argument any object that responds to `each`. That includes arrays, hashes, and ranges—and, of course, other sets.

#### MERGING A COLLECTION INTO ANOTHER SET

What happens when you merge another object into a set depends on what that object's idea of “each” is. Here's an array example, including a check on `object_id` to confirm that the original set has been altered in-place:

```
>> tri_state = Set.new(["Connecticut", "New Jersey"])
=> #<Set: {"Connecticut", "New Jersey"}>
>> tri_state.object_id
=> 2703420
>> tri_state.merge(["New York"])
=> #<Set: {"Connecticut", "New Jersey", "New York"}>
>> tri_state.object_id
=> 2703420
```

Merging a hash into a set results in the addition of two-element, key/value arrays to the set—because that's how hashes break themselves down when you apply the `each` method to them. Here's a slightly non-real-world example that nonetheless demonstrates the technology:

```
>> s = Set.new([1,2,3])
=> #<Set: {1, 2, 3}>
>> s.merge({ "New Jersey" => "NJ", "Maine" => "ME" })
=> #<Set: {1, 2, 3, ["New Jersey", "NJ"], ["Maine", "ME"]}>
```

(If you provide a hash argument to `Set.new`, the behavior is the same: you get a new set with two-element arrays based on the hash.)

You might want to merge not a hash, but the keys of a hash into a set. After all, set membership is based on hash key uniqueness, under the hood. You can do that with the `keys` method:

```
>> state_set = Set.new(["New York", "New Jersey"])
=> #<Set: {"New York", "New Jersey"}>
>> state_hash = { "Maine" => "ME", "Vermont" => "VT" }
=> { "Maine"=>"ME", "Vermont"=>"VT" }
>> state_set.merge(state_hash.keys)
=> #<Set: {"New York", "New Jersey", "Maine", "Vermont"}>
```

Try out some permutations of set merging, and you'll see that it's quite open-ended (just like set creation), as long as the argument is an object with an `each` method.

Sets wouldn't be sets without subsets and supersets, and Ruby's set objects are sub- and super-aware.

### 9.5.3 Subsets and supersets

You can test for subset/superset relationships between sets (and the arguments really have to be sets, not arrays, hashes, or any other kind of enumerable or collection) using the unsurprisingly named `subset` and `superset` methods:

```
>> small_states = Set.new(["Connecticut", "New Jersey", "Rhode Island"])
=> #<Set: {"Connecticut", "New Jersey", "Rhode Island"}>
>> small_states.subset?(state_set)
=> true
>> state_set.superset?(small_states)
=> true
```

Also available to you are the `proper_subset` and `proper_superset` methods. A *proper subset* is a subset that is smaller than the parent set by at least one element. If the two sets are equal, then they're subsets of each other but not proper subsets. Similarly, a *proper superset* of a set is a second set that contains all the elements of the first set plus at least one element not present in the first set. The “proper” concept is a way of filtering out the case where a set is a superset or subset of itself—because all sets are both.

We'll pick up the set thread in the next chapter, where we'll take another pass through collection objects in the interest of getting more deeply into the `Enumerable` module and the collection-based services it provides. Meanwhile, we'll turn a corner here. Unlike `Array`, `Hash`, and `Range`, the `Set` class is defined *in Ruby*. This gives us a great opportunity to look at some Ruby code.

## 9.6 Exploring the `set.rb` source code

You'll find the file `set.rb` in your Ruby distribution. Look in the `lib` subdirectory of the source tree or in the main directory of the installation.

It's in `set.rb` that the `Set` class is defined. Here, we'll look at a handful of the methods defined in that class, starting with `initialize`. The goal here is to give you the flavor of Ruby by exploring some production code—and also to show you that you've already learned enough that you can make sense of at least a good bit of the Ruby code that's used in the parts of Ruby that are written in Ruby.

One or two of the methods we'll look at here also sneak in a couple of techniques you haven't seen yet but will soon. Don't worry; they're reasonably easy to follow, and if you're at all puzzled, come back after the next chapter and you won't be!

### 9.6.1 `Set#initialize`

As you saw in section 9.5.1, you have three choices when you initialize a set: create an empty set; populate the set at creation with the items in an array or other collection object; or provide both a collection and a code block, populating the set with the cumulative result of yielding each of the collection's items to the block. (Have another look at section 9.5.1 to see examples.)

Here's the `Set#initialize` method, which handles these three scenarios:

```
def initialize(enum = nil, &block)
  @hash ||= Hash.new
```

❶

```

enum.nil? and return      ❷
if block                  ❸
  enum.each { |o| add(block[o]) }  ❹
else
  merge(enum)            ❺
end
end
end

```

Note the presence among the method parameters of the special parameter `&block`, which has the effect of capturing the code block as a function object (an object of class `Proc`) in the variable `block`. (The special part is the prepended `&`. You can name the variable itself anything.) This variable can then be used explicitly to execute the block, using any of several function-calling techniques including the `[]` (square brackets) method ❹. In effect, the statement `add(block[o])` is the same as `add(yield(o))`. Executing the block by binding it to a variable and executing it by yielding to it amount to the same thing. There are reasons to prefer one technique over the other in various circumstances, as you'll see when we look more closely at callable objects, including `Proc` objects, in chapter 14. In this example, either would work.

With that piece in place, let's look at the logic of the method. The first order of business is creating a hash ❶, which is stored in the instance variable `@hash`. This hash turns out to be the behind-the-scenes storage device for the set: when you examine or traverse the elements of a set, you're actually looking at this hash's keys. Using hash keys as the basis for set members is a handy way to guarantee their uniqueness. That's about three-quarters of the semantics of sets taken care of right there.

The `enum.nil?` test ❷ tells the new set whether it has any values to initialize itself to. If it doesn't, we're done: the caller wants an empty set, so we can return. If `enum` isn't `nil`, we keep going. Note the use of `and` as a kind of logic gate. Control only gets to the right-hand side of the `and` if the expression on the left is true—that is, if `enum` is `nil`.

What happens next depends on whether a block has been passed in. And that's determined by the conditional test `if block` ❸. If the variable `block` tests as false, that means no block was passed in because if one had been, it would have been stored as a `Proc` object in that variable. If that's the case—if no block was passed—the second conditional branch is executed and the set being created is merged with `enum` ❺. Merging has the effect of inserting all the items in `enum` into the set. (The special `&block` parameter comes with an implicit default-to-`nil` behavior if there's no block; you don't need to write `&block=nil`.)

But if a block was passed, things get more interesting. An iteration takes place through the object `enum` ❹ using its `each` method. On each iteration through `enum`, the current item (bound to the block parameter `o`) is added to the set, using the `add` method. But it's not just added raw; instead, it's passed to the block. What gets added to the set is the next element in `enum`, operated on in whatever manner the function in `block` is coded to do.

If you look at the `initialize` code alongside the examples from section 9.5.1, you can trace what happens in each of the various scenarios.

One interesting lesson to take away from examining `Set#initialize` is how it treats `enum`. In most cases, `enum` will be an array. But notice that there's no `if enum.is_a?(Array)` test. The only thing required of `enum` is that it have an `each` method. Thus you're free to initialize a set from any "each"-responsive object. Furthermore, the `merge` method takes the same approach: it has some special handling of the case where `enum` (`merge` uses the same variable name, as it happens) is itself a set object; but if it's anything other than a set, `merge` walks through it with `each` without examining what class it belongs to.

This approach to objects—ignoring what class they belong to and concentrating only on what they can *do*—is strongly characteristic of idiomatic, scalable Ruby programming. You don't refuse to do business with an object because its class is `MyCoolClass` rather than `Array`. Instead, you tell the object what to do—and if it can't do it, you'll know soon enough. Yes, there are cases where nothing but a classname check will do. (As already noted, `Set#merge` singles out the case where `enum` is a `Set` instance.) But for the most part, Ruby programming is about directly asking objects to do things and then reacting to the results.

Let's look at a couple more method examples—shorter ones!—from the `Set` class.

### 9.6.2 **Set#include?**

Not surprisingly, a set is considered to include an object if that object occurs as a key in `@hash`. Here's how `Set` defines its `include?` method:

```
def include?(o)
  @hash.include?(o)
end
```

The set simply bumps its logic to the hash it created when it itself was created. To a large extent, hash keys are already a set—so they can serve as the basis for one.

### 9.6.3 **Set#add and Set#add?**

How about the `add` method, with which you add an element to a set? Here's what it looks like:

```
def add(o)
  @hash[o] = true
  self
end
```

All it does is set the hash key for the object, `o`, to `true`, in the hash `@hash`. The hash takes care of the uniqueness. If `@hash[o]` is already `true`, it gets reset to `true`—no harm done. If it isn't already `true`, it now is.

The `add?` method is a slight elaboration of `add`. As you'll recall, the difference between the two is that `add?` returns `nil` in the event that the object it's trying to add already occurs in the set, whereas `add` always returns the set itself. Here's what `add?` looks like:

```
def add?(o)
  if include?(o)
    nil
  else
    add(o)
  end
end
```

A simple test determines the branch taken: `nil` if the object is in the set, the outcome of calling `add` with the same argument if it isn't.

The Ruby source code is a great source of Ruby examples. Much of the time you'll come across techniques that you're not yet familiar with—but you'll also see much (and increasingly more) that you do recognize, and working through some of the logic and syntax can be a beneficial exercise.

## 9.7 Summary

In this chapter, we've looked at Ruby's major core container classes, `Array` and `Hash`. We've also looked at ranges, which principally operate as inclusion test criteria but know how to behave as collections when their makeup permits them to. After ranges, we looked at sets, which are defined in the standard library and which add another important tool to Ruby's collection toolset. The source code for the `Set` class is written in Ruby; that gave us an opportunity to look at some real production Ruby code.

The concept of the *collection* in Ruby is closely associated with the `Enumerable` module and its principle of dependence on an `each` method. In this chapter, we've used the rudiments of the *each* concept so we can see how enumerability relates to collections.

In the next chapter, we're going to go more deeply into `Enumerable`—which means looking at the many searching, filtering, sorting, and transforming operations available on objects whose classes mix in that module.