CHAPTER 12
Create Classes That Understand Equality

Have you ever noticed that it's smallest words that have the longest entries in the dictionary? My copy of *Webster*'s dismisses a ten-dollar word like *cryptozoology* with a single three-line definition.[1] In contrast, although we all would like to think that we know right from wrong, the word *right* consumes almost two pages and sports no less than 58 separate definitions. Fortunately, software engineers don't usually have to wrestle with the finer points of moral philosophy while coding. We do, however, have something almost as difficult: We have object equality. Like right and wrong, object equality is a much tougher question than it appears on first blush.

Since understanding object equality is one of the keys to creating well-behaved Ruby objects, we are going to spend this chapter looking at the various Ruby definitions of equality. We will see that there are a lot of different ways that Ruby objects can be equal, and that all these definitions of equality are rooted firmly in helping you make your objects behave the way you want them to behave.

An Identifier for Your Documents

To see just how slippery object equality can be, let's return again to our documents and imagine that everyone likes your `Document` class so much that they've been creating

1. It's the study of animals like the Loch Ness Monster and Yeti, animals that may or may not exist. Probably not.

more and more of them at a furious pace. There are now so many documents that managing them has become a problem, so your company has started a second project, a project aimed at building a system to store and manage all of those documents. To this end, you decide that you will need some kind of identifier object for your documents, a key that will allow you to pick a given document out of a crowd. And thus is born the `DocumentIdentifier` class:

```
class DocumentIdentifier
  attr_reader :folder, :name

  def initialize( folder, name )
    @folder = folder
    @name = name
  end
end
```

The idea behind `DocumentIdentifier` is very simple: The management system assigns each document a (presumably unique) name and groups the documents into folders. To locate any given document you need both the document name and its folder, which is exactly the information that `DocumentIdentifier` instances carry around. You release the `DocumentIdentifier` class to the group that is building the document management system and then lean back and put your feet up on the desk.

Sadly, your "not much to that job" feeling does not survive the afternoon. Your colleagues are back, and they have a complaint: As it stands, there is no easy way to compare two instances of `DocumentIdentifier` to see whether they actually point at the same document. They want you to enhance your class so that they can tell whether one document identifier is equal to another.

An Embarrassment of Equality

The question is, equal in what sense? It turns out that Ruby's `Object` class defines no less than four equality methods. There is `eql?` and `equal?` as well as == (that's two equal signs), not to mention === (that's three equal signs). For a language that prides itself on simplicity and conciseness, that's an awful lot of ways of saying nearly the same thing. Nearly, but not exactly the same.

The good news is that we can dismiss one of these equality methods, `equal?`, right up front. Ruby uses the `equal?` method to test for object identity. In other words, the only way that this:

```
x.equal?(y)
```

Should ever return true is if `x` and `y` are both references to identically same objects. If `x` and `y` are different objects, then `equal?` should always return false, no matter how similar `x` and `y` might be. This brings us to the reason why `equal?` is unlikely to cause you any problems: Ruby is really good at figuring out when two objects are the same object. In fact, Ruby—in the form of the implementation of `equal?` found in `Object` class—is so good at this that there is no need for you to override `equal?`. Ever.

Double Equals for Everyday Use

Which brings us to the next version of Ruby equality, the == or double-equals operator. Since the == operator is the one that Ruby programmers habitually reach for when they are coding, this is probably the one we need to fix in our `DocumentIdentifier` class. Our problem is that the default implementation of ==, the one that `DocumentIdentifier` inherits from `Object`, does the same thing as `equal?`—it tests for object identity. For example, if you do this:

```
first_id = DocumentIdentifier.new( 'secret/plans', 'raygun.txt' )
second_id = DocumentIdentifier.new('secret/plans', 'raygun.txt' )

puts "They are equal!" if first_id == second_id
```

You will see, well, nothing. Left to its own devices, the == method will behave just like `equal?` and will only return true if the objects being compared are identically the same object. Unlike the `equal?` method, testing for object identity is just the default behavior for the == method: You are free to implement any kind of "same value" comparison that makes sense for your object. The way to do this is to implement your own == method:

```
class DocumentIdentifier
  attr_reader :folder, :name
  def initialize( folder, name )
    @folder = folder
    @name = name
  end

  def ==(other)
    return false unless other.instance_of?(self.class)
    folder == other.folder && name == other.name
  end
end
```

The logic behind the == method shown here is pretty mundane: First, it makes sure that the other object is in fact an instance of `DocumentIdentifier`. If it is, the method goes on to check that the other object has the same folder and name. Notice that there is no need for a special check for `nil`: Ruby's `nil` is just another object, in this case an object that will fail the `instance_of?` test.

One addition we might make is to check whether the other object is in fact identically the same as this object:

```
class DocumentIdentifier

  # ...

  def ==(other)
    return true if other.equal?(self)
    return false unless other.instance_of?(self.class)
    folder == other.folder && name == other.name
  end
end
```

This kind of check can speed up your equality comparisons quite a bit and might be worth doing if you think you are going to being using the == method a lot.

With our newly enhanced code, the most common of the Ruby equality methods now works for our document identifiers so that this:

```
puts "They are equal!" if first_id == second_id
```

will indeed print something.

Broadening the Appeal of the == Method

One problem with our current == implementation is that it takes a very narrow view of what it means to be equal. That `instance_of?` test at the beginning means that the other object must be an instance of a `DocumentIdentifier`—no subclasses allowed. We can loosen the rules a bit by using `kind_of?`, which will return true if the object is an instance of `DocumentIdentifier` or a subclass of `DocumentIdentifier`:

```
class DocumentIdentifier
  # ...

  def ==(other)
    return true if other.equal?(self)
    return false unless other.kind_of?(self.class)
    folder == other.folder && name == other.name
  end
end


class ContractIdentifier < DocumentIdentifier
end
```

Now, if we make an instance of each:

```
doc_id =  DocumentIdentifier.new( 'contracts', 'Book Deal' )
con_id =  ContractIdentifier.new( 'contracts', 'Book Deal' )
```

And then compare the two, we will find that `doc_id == con_id` is indeed true, since the two objects are carrying around the same data and `ContractIdentifier` is a subclass of `DocumentIdentifier`.

You may be feeling that all this worry about classes and subclasses has a distinctly un-Ruby feel to it. Didn't we say that Ruby programmers frown on attaching too much importance to the class of an object? Can't we do something a little more in the spirit of Ruby's dynamic typing?

We can indeed. The following code defines `DocumentPointer`, a class completely unrelated to `DocumentIdentifier`:

```
class DocumentPointer
  attr_reader :folder, :name

  def initialize( folder, name )
    @folder = folder
    @name = name
  end

  def ==(other)
    return false unless other.respond_to?(:folder)
    return false unless other.respond_to?(:name)
    folder == other.folder && name == other.name
  end
end
```

The `DocumentPointer` class dispenses with the class check and instead pays attention to whether the other object has the right methods: Using the `respond_to?` method, the `DocumentPointer` class asks if this other object has a `name` method and a `folder` method. If so, then it might be an equal. Using this approach, instances of `DocumentPointer` will accept an instance of a completely unrelated class—`DocumentIdentifier` for example—as an equal:

```
doc_id = DocumentIdentifier.new( 'secret/area51', 'phone list' )
pointer = DocumentPointer .new('secret/area51', 'phone list' )

pointer == doc_id   # True!!
```

Well-Behaved Equality

Unfortunately we still have some problems. Both the `kind_of?` as well as the `respond_to?` based implementations of == suffer from the "different classes may have different points of view" problem that we came across in the last chapter. For example, in the last section we were happy because by relying on `kind_of?`, we had managed to make the `DocumentIdentifier` recognize the equality of subclasses, so that we got some output from:

```
doc_id =  DocumentIdentifier.new( 'contracts', 'Book Deal' )
con_id =  ContractIdentifier.new( 'contracts', 'Book Deal' )

puts "They are equal!" if doc_id == con_id
```

The trouble starts when we try to reverse the order of the expression. This:

```
puts "They are equal!" if con_id == doc_id
```

Will print nothing. The problem is that although `ContractIdentifier` is a subclass of `DocumentIdentifier`, the reverse is not true, and when you say `con_id == doc_id`, you are really calling the == method on the `ContractIdentifier` instance.

We have a similar disconnect between `DocumentIdentifier` and the `DocumentPointer` classes. In the last section we put the `respond_to?` logic into `DocumentPointer` so that `pointer == identifier` comes out true. Unfortunately, since we didn't put the same smarts into the `DocumentIdentifier` class, `identifier != pointer`.

As I say, this sad situation is an example of exactly the kind of muddled operator definition that we talked about back in Chapter 11. The problem is that we have defined equality relationships that violate the principal of **symmetry**: People tend to expect that if `a == b` then `b == a`.

There really is no magic elixir that will fix an asymmetrical equality relationship: You can either change the == methods on both classes so that they agree, or you can simply live with (and perhaps document) a less-than-intuitive, asymmetrical equality.

Asymmetry is also not the extent of our woes. Another built-in assumption we have about equality is that if `a == b` and `b == c`, then surely `a == c`. This **transitive property** is another expectation that is all too easy to violate. To see how, imagine that our document management system starts storing multiple of versions of each document. We might then build a subclass of `DocumentIdentifier` that knows about the versions and tries to be smart doing equality comparisons:

```
class VersionedIdentifier < DocumentIdentifier
  attr_reader :version

  def initialize( folder, name, version )
    super( folder, name )
```

```
      @version = version
    end

    def ==(other)
      if other.instance_of? VersionedIdentifier
        other.folder == folder &&
        other.name == name &&
        other.version == version
      elsif other.instance_of? DocumentIdentifier
        other.folder == folder && other.name == name
      else
        false
      end
    end
  end
```

This == method does something that seems very sensible: If the other object is a `VersionedIdentifier`, then it does a full comparison including the version. Alternatively, if the other object is a plain old `DocumentIdentifier`, then the method looks only at the folder and name fields. The trouble is that this == method is not transitive. To see why, think about these objects:

```
versioned1 = VersionedIdentifier.new( 'specs', 'bfg9k', "V1" )
versioned2 = VersionedIdentifier.new( 'specs', 'bfg9k', "V2" )
unversioned = DocumentIdentifier .new('specs', 'bfg9k')
```

All three point to the same basic document, but the two versioned identifiers point at different versions of the document. Since the plain `DocumentIdentifier` class doesn't know about the versions, and the `VersionedIdentifier` class deliberately ignores the version when dealing with a plain identifier, both of these expressions are true:

```
versioned1 == unversioned     # True!
unversioned == versioned2     # True!
```

Unfortunately, this expression is very definitely not true:

```
versioned1 ==  versioned2     # Not true!
```

In our efforts to do something sensible we have managed to create a situation where `(a == b)` and `(b == c)` but `(a != c)`.

There are a couple of ways around this conundrum. One is to dispense with trying to deal with both `VersionedIdentifier` instances and regular `DocumentIdentifier` instances in the `VersionedIdentifier ==` method. Instead, add a method to `VersionedIdentifier` that returns the plain old document identifier:

```
def as_document_identifier
  DocumentIdentifier.new( folder, name )
end
```

You can then compare the resulting document identifier, serene in the knowledge that you know exactly what is going on.

The other way out is to ask yourself whether you really need to use the == operator for everything. It is easy enough to add your own specialized comparison method to `VersionedIdentifier`:

```
def is_same_document?(other)
  other.folder == folder && other.name == name
end
```

The `is_same_document?` method here simply ignores the versions and will return true if the two identifiers point at the same document.

Triple Equals for Case Statements

The next contestant in our equality sweepstakes is the triple equals operator ===. The main use for === is in `case` statements.

So why do we need yet another operator just for `case` systems? Why not just use == in `case` statements? It turns out that this is another example of Ruby pragmatism in pursuit of[2] clean and concise code. Take strings and regular expressions for example. Strings are not regular expressions and regular expressions are not strings, so we certainly would not want a string to be equal to a regular expression according to == even if they do match:

2. Wait for it!

```
/Roswell.*/ =~ 'Roswell' # Yes!
/Roswell.*/ == 'Roswell' # No!
```

However, pattern matching regular expressions against strings in `case` statements does make for tidy-looking code:

```
location = 'area 51'

case location
when /area.*/
  # ...
when /roswell.*/
  # ...
else
  # ...
end
```

To this end, the `Regexp` class has a === method that does pattern matching when confronted with a string.

By default, === calls the double equals method, so unless you specifically override ===, wherever you send ==, === is sure to follow. It's probably a good idea to leave === alone unless doing so results in really ugly case statements.

Hash Tables and the eql? Method

Finally, we have the `eql?` method. Since you don't typically call `eql?` directly, you might never know you need it—until you try to use your object as a key in a hash. To see where the trouble lies, imagine that you store a document in a hash table with a `DocumentIdentifier` as the key:

```
hash = {}
document = Document.new( 'cia', 'Roswell', 'story' )
first_id = DocumentIdentifier.new( 'public', 'CoverStory' )

hash[first_id] = document
```

Initially, things seem to work. You have no trouble getting your document out of the hash with the identifier, so `hash[first_id]` will indeed return your document.

The problem is that if you try fetching your document out of the hash with a second instance of `DocumentIdentifier`, like this:

```
second_id =  DocumentIdentifier.new( 'public', 'CoverStory' )
the_doc_again =  hash[second_id]
```

You will end up with `the_doc_again` set to `nil`. To see what the problem is, we need to dive into the workings of the `Hash` class.

The idea behind a hash is to build a thing that works a lot like an array, but an array on performance-enhancing drugs. The feature that makes hashes special is that they can take things other than just numbers as indexes. Now there is a spectacularly simple way to get this behavior: You build a class that stores all the keys and values in a simple list. When you need to find a value by its key, you simply do a linear search down that list, looking at each key until you find the one you want. The trouble with this simple implementation is that performance falls off in direct proportion to the number of entries in the table. If you have ten entries in your simple table, then on average you'll have to look at five entries[3] before you hit the key that you want, which isn't too bad. Unfortunately, if you have 1,000 entries you'll probably have to look at 500 entries before you get lucky, and if you have 10,000 entries . . . Well, you get the picture.

Real hash tables improve the performance of this simple model with a divide-and-conquer strategy. Instead of maintaining a single key/value list, a typical hash table implementation maintains a number of lists, or **buckets**. By spreading out the stuff that's stored in the table across a number of buckets, a hash table can do things dramatically faster. Take that 10,000 entry table: If you spread the 10,000 entries over 100 different buckets, instead of having to look at 5,000 entries to find the one you want, you only need to search through about 50.[4]

The challenging thing with this scheme is picking the right bucket: If you are saving a new key/value pair, which bucket do you use? Later, when you go looking for that same key, how do you know which bucket you picked originally? Answering the

3. Remember, on average you are only going to have to search down half the list before you find what you are looking for.

4. Starting in version 1.9, Ruby hashes have a list superimposed atop the whole bucket structure, a list that keeps track of the order of the hash keys, because the natural workings of a hash will randomize the keys very thoroughly.

"which bucket" question is where the hash part of a hash table comes in. The idea goes like this: You define a method on all of your objects, a method which returns a hash value. The **hash value** is a more or less random number somehow generated from the value of the object. When you need to store a key/value pair in your hash table, you pull the hash value from the key. You then use that number to pick a bucket—typically by using the modulo operator (hash_code % number_of_buckets)—and you store your key/value pair in that bucket. Later on when you are looking to retrieve the value associated with some key, you get the hash code for the key again and use it to pick the right bucket to search.

Hash codes need to have a couple of properties to make this all work. First, they need to be stable over time: If a key generates one hash code now and a different one later, we are inevitably going to end up looking in the wrong bucket. Hash codes also need to be consistent with the value of the key: If two keys are equal—if they should return the same value out of the hash table—then, when asked, they must return the same hash code.

Building a Well-Behaved Hash Key

All of this theory translates pretty directly into Ruby: The `Hash` class calls the aptly named `hash` method (another one of those methods that you inherit from `Object`) to get the hash code from its keys. The `Hash` class uses the `eql?` method to decide if two keys are in fact the same key. The default implementations of `hash` and `eql?` from the `Object class`, like the default implementations of == and ===, are based on object identity: The default `eql?` returns true only if the other object is identically the same as this object. The default hash method returns the `object_id` of the object, which is guaranteed to be unique. This is why, in the example, our second `DocumentIdentifier` instance failed to find anything in the hash.

There is, however, no rule saying that your class needs to accept the default implementation. As long as you follow the hash Prime Directive—that if `a.eql?(b)` then `a.hash == b.hash`—you are free to override these two methods. Devising an implementation of `eql?` and `hash` that will work together is really not that difficult. You just need to make sure that any field that has a vote in the hash code also has a vote in equality.

Going back to the `DocumentIdentifier` example, our `eql?` should obviously take both the `folder` and the `name` fields into account. Given this, we want to make sure that both fields have a say in the `hash` code. So here is what we do:

```
class DocumentIdentifier

  # Code omitted...

  def hash
    folder.hash ^ name.hash
  end

  def eql?(other)
    return false unless other.instance_of?(self.class)
    folder == other.folder && name == other.name
  end
end
```

In the code above, the `hash` method combines the hash codes from the two fields using the exclusive or operator `^`: Doing this is a simple way of creating a very thorough mishmash of the two numbers, which is exactly what we are looking for in a hash. The other thing to notice in this code is that the `eql?` method takes a very restrictive view of equality. According to the `eql?`, only other instances of `DocumentIdentifier` can be equal: Getting your objects to work as hash table keys is no time to be imaginative about cross-class equality.

Staying Out of Trouble

Making equality work can be painful. There are, however, things that you can do to minimize the pain. The best thing is to avoid the suffering all together. Given how easy it is to screw up when you start messing with object equality, your first rule should be: If it ain't broke—or used—don't fix it. Many objects will never find themselves in the middle of an equality expression or be called upon to be a hash key. If you have an object like this, then just leave its equality methods alone.

Even if you do need to implement some or all of the equality methods, you might be able to lean on someone else's work.[5] If you're defining a class that is mostly a wrapper for some other object, consider borrowing the equality methods from that other object. For example, if you are building a class that wraps an array, you just might be able to delegate to the array's equality methods:

5. According to the American screenwriter Aaron Sorkin, good writers borrow from other writers. Great writers steal from them outright. Apparently Sorkin stole this line from Oscar Wilde.

```
class DisArray

  attr_reader :my_array

  def initialize
    @my_array = []
  end

  def ==(other)
    return false unless other.kind_of?(DisArray)
    @my_array == other.my_array
  end

  def eql?(other)
    return false unless other.kind_of?(DisArray)
    @my_array.eql?( other.my_array )
  end

  def hash
    @my_array.hash
  end

  # Rest of the class omitted...
end
```

Finally, if you do need to write your own equality methods, do the simplest thing that will work. If you don't have to support equality across different classes, then don't. A limited, but working implementation is better than an elaborate and subtly broken one.

In the Wild

Ruby's built-in numeric classes do a bit of equality slight of hand right under your nose. A little exploration will show that integers (that is, instances of `Fixnum` or `Bignum`) will accept instances of `Float` as equals, at least according to the == method. Thus, if you run:

```
puts 1 == 1.0     # A Fixnum and a Float
```

You will see that `1` is in fact == to `1.0`. Ruby does this by converting the `Fixnum` to a `Float` before doing the comparison.

Classes like `Float` and `Fixnum`, classes whose instances have a natural ordering, can add one additional twist to the equality saga in the form of the <=> operator. The expression:

```
a <=> b
```

Should evaluate to `-1` if `a` is less than `b`, `0` if they are equal, and `1` if `a` is greater than `b`. The <=> is a boon when you are trying to sort a collection of objects. If you find yourself needing to implement <=>, keep in mind that <=> should be consistent with ==. That is, if `a <=> b` evaluates to zero, then `a == b` should be true. The good news is that Ruby actually supplies a mixin module (see Chapter 16 for more on mixin modules) to help you keep all of this straight. If you define a <=> operator for your class, and include `Comparable`, like this:

```
class RomanNumerals
  include Comparable


  # Actual guts of the class omitted...

  def <=>(other)
    # Return -1, 0, or 1...
  end
end
```

Then `Comparable` will not only add a == method to your class, but also <, <=, >=, and >, all of which will rely on your <=> method to come up with the right answer.

Ruby classes—those objects that are instances of `Class`—have their own twist on the triple equality method: Classes treat the === method as an alias for `kind_of?`. This is so that you can pick out the class of an object with a `case` statement, like so:

```
the_object = 3.14159

case the_object
when String
  puts "it's a string"
```

```
    when Float
      puts "It's a float"

    when Fixnum
      puts "It's a fixnum"

    else
      puts "Dunno!"
    end
```

This is yet another asymmetric relationship: Although `Float === 1.0` is true, `1.0 === Float` is not.

Wrapping Up

I sometimes think that you can divide software engineers into two classes: the optimists and those who have tangled with object equality. In my more optimistic moments I realize that while getting object equality right can be trying, it is certainly doable. The key to getting it right is to keep in mind that Ruby has a fairly fine-grained model of equality—we have the `equal?` method, strictly for object identity. We have the everyday equality method, ==, and we also have the === method, which comes out mostly for `case` statements. We also have `eql?`, and its friend `hash`, to cope with hash tables. Getting object equality right is all about understanding the differences between all those methods and overriding the right ones.