Chapter 23

Duck Typing

You'll have noticed that in Ruby we don't declare the types of variables or methods—everything is just some kind of object.

Now, it seems like folks react to this in two ways. Some like this kind of flexibility and feel comfortable writing code with dynamically typed variables and methods. If you're one of those people, you might want to skip to the section called "Classes Aren't Types" on the next page. Some, though, get nervous when they think about all those objects floating around unconstrained. If you've come to Ruby from a language such as C# or Java, where you're used to giving all your variables and methods a type, you may feel that Ruby is just too sloppy to use to write "real" applications.

It isn't.

We'd like to spend a couple of paragraphs trying to convince you that the lack of static typing is not a problem when it comes to writing reliable applications. We're not trying to criticize other languages here. Instead, we'd just like to contrast approaches.

The reality is that the static type systems in most mainstream languages don't really help that much in terms of program security. If Java's type system were reliable, for example, it wouldn't need to implement ClassCastException. The exception is necessary, though, because there is runtime type uncertainty in Java (as there is in C++, C#, and others). Static typing can be good for optimizing code, and it can help IDEs do clever things with tooltip help, but we haven't seen much evidence that it promotes more reliable code.

On the other hand, once you use Ruby for a while, you realize that dynamically typed variables actually add to your productivity in many ways. You'll also be surprised to discover that your fears about the type chaos were unfounded. Large, long-running, Ruby programs run significant applications and just don't throw any type-related errors. Why is this?

Partly, it's a question of common sense. If you coded in Java (pre–Java 1.5), all your containers were effectively untyped: everything in a container was just an Object, and you cast it to the required type when you extracted an element. And yet you probably never saw a ClassCastException when you ran these programs. The structure of the code just didn't permit it. You put Person objects in, and you later took Person objects out. You just don't write programs that would work in another way.

Well, it's the same in Ruby. If you use a variable for some purpose, chances are very good that you'll be using it for the same purpose when you access it again three lines later. The kind of chaos that *could* happen just doesn't happen.

On top of that, folks who code Ruby a lot tend to adopt a certain style of coding. They write lots of short methods and tend to test as they go along. The short methods mean that the scope of most variables is limited; there just isn't that much time for things to go wrong with their type. And the testing catches the silly errors when they happen; typos and the like just don't get a chance to propagate through the code.

The upshot is that the "safety" in "type safety" is often illusory and that coding in a more dynamic language such as Ruby is both safe and productive. So, if you're nervous about the lack of static typing in Ruby, we suggest you try to put those concerns on the back burner for a little while and give Ruby a try. We think you'll be surprised at how rarely you see errors because of type issues and at how much more productive you feel once you start to exploit the power of dynamic typing.

Classes Aren't Types

The issue of types is actually somewhat deeper than an ongoing debate between strong typing advocates and the hippie-freak dynamic typing crowd. The real issue is the question, what is a type in the first place?

If you've been coding in conventional typed languages, you've probably been taught that the *type* of an object is its *class*—all objects are instances of some class, and that class is the object's type. The class defines the operations (methods) the object can support, along with the state (instance variables) on which those methods operate. Let's look at some Java code:

```
Customer  c;
c = database.findCustomer("dave");    /* Java */
```

This fragment declares the variable c to be of type Customer and sets it to reference the customer object for Dave that we've created from some database record. So, the type of the object in c is Customer, right?

Maybe. However, even in Java, the issue is slightly deeper. Java supports the concept of *interfaces*, which are a kind of emasculated abstract base class. A Java class can be declared as implementing multiple interfaces. Using this facility, you may have defined your classes as follows:

```
public interface Customer {
  long  getID();
  Calendar getDateOfLastContact();
  // ...
}
public class Person
  implements Customer {
  public long getID() { ... }
  public Calendar getDateOfLastContact() { ... }
  // ...
}
```

So, even in Java, the class is not always the type—sometimes the type is a subset of the class, and sometimes objects implement multiple types.

In Ruby, the class is never (OK, almost never) the type. Instead, the type of an object is defined more by what that object can do. In Ruby, we call this *duck typing*. If an object walks like a duck and talks like a duck, then the interpreter is happy to treat it as if it were a duck.

Let's look at an example. Perhaps we've written a method to write our customer's name to the end of an open file:

Download **samples/ducktyping_3.rb**

```
class Customer
  def initialize(first_name, last_name)
    @first_name = first_name
    @last_name  = last_name
  end
  def append_name_to_file(file)
    file << @first_name << " " << @last_name
  end
end
```

Being good programmers, we'll write a unit test for this. Be warned, though—it's messy (and we'll improve on it shortly):

Download **samples/ducktyping_4.rb**

```
require 'test/unit'
require 'addcust'
class TestAddCustomer < Test::Unit::TestCase
  def test_add
    c = Customer.new("Ima", "Customer")
    f = File.open("tmpfile", "w") do |f|
      c.append_name_to_file(f)
    end
    f = File.open("tmpfile") do |f|
      assert_equal("Ima Customer", f.gets)
    end
  ensure
    File.delete("tmpfile") if File.exist?("tmpfile")
  end
end
```

produces:

```
Finished in 0.001084 seconds.
1 tests, 1 assertions, 0 failures, 0 errors, 0 skips
```

We have to do all that work to create a file to write to, then reopen it, and read in the contents to verify the correct string was written. We also have to delete the file when we've finished (but only if it exists).

Instead, though, we could rely on duck typing. All we need is something that walks like a file and talks like a file that we can pass in to the method under test. And all that means *in*

this circumstance is that we need an object that responds to the << method by appending something. Do we have something that does this? How about a humble String?

Download **samples/ducktyping_5.rb**

```
require 'test/unit'
require 'addcust'
class TestAddCustomer < Test::Unit::TestCase
  def test_add
    c = Customer.new("Ima", "Customer")
    f = ""
    c.append_name_to_file(f)
    assert_equal("Ima Customer", f)
  end
end
```

produces:

```
Finished in 0.000361 seconds.
1 tests, 1 assertions, 0 failures, 0 errors, 0 skips
```

The method under test thinks it's writing to a file, but instead it's just appending to a string. At the end, we can then just test that the content is correct.

We didn't have to use a string—for the object we're testing here, an array would work just as well:

Download **samples/ducktyping_6.rb**

```
require 'test/unit'
require 'addcust'
class TestAddCustomer < Test::Unit::TestCase
  def test_add
    c = Customer.new("Ima", "Customer")
    f = []
    c.append_name_to_file(f)
    assert_equal(["Ima", " ", "Customer"], f)
  end
end
```

produces:

```
Finished in 0.000405 seconds.
1 tests, 1 assertions, 0 failures, 0 errors, 0 skips
```

Indeed, this form may be more convenient if we wanted to check that the correct individual things were inserted.

So, duck typing is convenient for testing, but what about in the body of applications themselves? Well, it turns out that the same thing that made the tests easy in the previous example also makes it easy to write flexible application code.

In fact, Dave had an interesting experience where duck typing dug him (and a client) out of a hole. He'd written a large Ruby-based web application that (among other things) kept a

database table full of details of participants in a competition. The system provided a comma-separated value (CSV) download capability, allowing administrators to import this information into their local spreadsheets.

Just before competition time, the phone starts ringing. The download, which had been working fine up to this point, was now taking so long that requests were timing out. The pressure was intense, because the administrators had to use this information to build schedules and send out mailings.

A little experimentation showed that the problem was in the routine that took the results of the database query and generated the CSV download. The code looked something like this:

```
def csv_from_row(op, row)
  res = ""
  until row.empty?
    entry = row.shift.to_s
    if /[,"]/ =~ entry
      entry = entry.gsub(/"/, '""')
      res << '"' << entry << '"'
    else
      res << entry
    end
    res << "," unless row.empty?
  end
  op << res << CRLF
end
result = ""
query.each_row {|row|  csv_from_row(result, row)}
http.write result
```

When this code ran against moderate-size data sets, it performed fine. But at a certain input size, it suddenly slowed right down. The culprit? Garbage collection. The approach was generating thousands of intermediate strings and building one big result string, one line at a time. As the big string grew, it needed more space, and garbage collection was invoked, which necessitated scanning and removing all the intermediate strings.

The answer was simple and surprisingly effective. Rather than build the result string as it went along, the code was changed to store each CSV row as an element in an array. This meant that the intermediate lines were still referenced and hence were no longer garbage. It also meant that we were no longer building an ever-growing string that forced garbage collection. Thanks to duck typing, the change was trivial:

```
def csv_from_row(op, row)
  # as before
end
result = []
query.each_row {|row|  csv_from_row(result, row)}
http.write result.join
```

All that changed is that we passed an array into the csv_from_row method. Because it (implicitly) used duck typing, the method itself was not modified; it continued to append

the data it generated to its parameter, not caring what type that parameter was. After the method returned its result, we joined all those individual lines into one big string. This one change reduced the time to run from more than three minutes to a few seconds.

Coding like a Duck

1.9

If you want to write your programs using the duck typing philosophy, you really need to remember only one thing: an object's type is determined by what it can do, not by its class. (In fact, older versions of Ruby had a method Object#type that returned the class of an object. That has been removed in Ruby 1.9—the name type was misleading.)

What does this mean in practice? At one level, it simply means that there's often little value testing the class of an object.

For example, you may be writing a routine to add song information to a string. If you come from a C# or Java background, you may be tempted to write this:

```
def append_song(result, song)
  # test we're given the right parameters
  unless result.kind_of?(String)
    fail TypeError.new("String expected")
  end
  unless song.kind_of?(Song)
    fail TypeError.new("Song expected")
  end

  result << song.title << " (" << song.artist << ")"
end

result = ""
append_song(result, song)   # =>   "I Got Rhythm (Gene Kelly)"
```

Embrace Ruby's duck typing, and you'd write something far simpler:

```
def append_song(result, song)
  result << song.title << " (" << song.artist << ")"
end

result = ""
append_song(result, song)   # =>   "I Got Rhythm (Gene Kelly)"
```

You don't need to check the type of the arguments. If they support << (in the case of *result*) or title and artist (in the case of *song*), everything will just work. If they don't, your method will throw an exception anyway (just as it would have done if you'd checked the types). But without the check, your method is suddenly a lot more flexible. You could pass it an array, a string, a file, or any other object that appends using <<, and it would just work.

Now sometimes you may want more than this style of *laissez-faire* programming. You may have good reasons to check that a parameter can do what you need. Will you get thrown out

of the duck typing club if you check the parameter against a class? No, you won't.[1] But you may want to consider checking based on the object's capabilities, rather than its class:

Download samples/ducktyping_11.rb

```
def append_song(result, song)
  # test we're given the right parameters
  unless result.respond_to?(:<<)
    fail TypeError.new("'result' needs `<<' capability")
  end
  unless song.respond_to?(:artist) && song.respond_to?(:title)
    fail TypeError.new("'song' needs 'artist' and 'title'")
  end

  result << song.title << " (" << song.artist << ")"
end

result = ""
append_song(result, song)   # =>   "I Got Rhythm (Gene Kelly)"
```

However, before going down this path, make sure you're getting a real benefit—it's a lot of extra code to write and to maintain.

Standard Protocols and Coercions

Although not technically part of the language, the interpreter and standard library use various protocols to handle issues that other languages would deal with using types.

Some objects have more than one natural representation. For example, you may be writing a class to represent Roman numbers (I, II, III, IV, V, and so on). This class is not necessarily a subclass of Integer, because its objects are representations of numbers, not numbers in their own right. At the same time, they do have an integer-like quality. It would be nice to be able to use objects of our Roman number class wherever Ruby was expecting to see an integer.

To do this, Ruby has the concept of *conversion protocols*—an object may elect to have itself converted to an object of another class. Ruby has three standard ways of doing this.

We've already come across the first. Methods such as to_s and to_i convert their receiver into strings and integers. These conversion methods are not particularly strict. If an object has some kind of decent representation as a string, for example, it will probably have a to_s method. Our Roman class would probably implement to_s in order to return the string representation of a number (VII, for instance).

The second form of conversion function uses methods with names such as to_str and to_int. These are strict conversion functions. You implement them only if your object can naturally be used every place a string or an integer could be used. For example, our Roman number

1. The duck typing club doesn't check to see whether you're a member anyway....

objects have a clear representation as an integer and so should implement to_int. When it comes to stringiness, however, we have to think a bit harder.

Roman numbers clearly have a string representation, but are they strings? Should we be able to use them wherever we can use a string itself? No, probably not. Logically, they're a representation of a number. You can represent them as strings, but they aren't plug-compatible with strings. For this reason, a Roman number won't implement to_str—it isn't really a string. Just to drive this home: Roman numerals can be converted to strings using to_s, but they aren't inherently strings, so they don't implement to_str.

To see how this works in practice, let's look at opening a file. The first parameter to File.new can be either an existing file descriptor (represented by an integer) or a filename to open. However, Ruby doesn't simply look at the first parameter and check whether its type is Fixnum or String. Instead, it gives the object passed in the opportunity to represent itself as a number or a string. If it were written in Ruby, it may look something like this:

Download **samples/ducktyping_12.rb**

```
class File
  def File.new(file, *args)
    if file.respond_to?(:to_int)
      IO.new(file.to_int, *args)
    else
      name = file.to_str
      # call operating system to open file 'name'
    end
  end
end
```

So, let's see what happens if we want to pass a file descriptor integer stored as a Roman number into File.new. Because our class implements to_int, the first respond_to? test will succeed. We'll pass an integer representation of our number to IO.open, and the file descriptor will be returned, all wrapped up in a new IO object.

A small number of strict conversion functions are built into the standard library.

`to_ary` → Array

This is used when interpreter needs a parameter to a method to be an array, and when expanding parameters and assignments containing the *xyz syntax.

Download **samples/ducktyping_13.rb**

```
class OneTwo
  def to_ary
    [ 1, 2 ]
  end
end
ot = OneTwo.new
puts ot
```

produces:

```
1
2
```

`to_a` → Array

1.9 This is used when interpreter needs to convert an object into an array for parameter passing or multiple assignment.

Download samples/ducktyping_14.rb

```
class OneTwo
  def to_a
    [ 1, 2 ]
  end
end
ot = OneTwo.new
a, b = *ot
puts "a = #{a}, b = #{b}"
printf("%d -- %d\n", *ot)
```

produces:

```
a = 1, b = 2
1 -- 2
```

`to_enum` → Enumerator

1.9 This converts an object (presumably a collection) to an enumerator. It's never called internally by the interpreter.

`to_hash` → Hash

This is used when the interpreter expects to see Hash. (The only known use is the second parameter to Hash#replace.)

`to_int` → Integer

This is used when the interpreter expects to see an integer value (such as a file descriptor or as a parameter to Kernel.Integer).

`to_io` → IO

This is used when the interpreter is expecting I/O objects (for example, as parameters to IO#reopen or IO.select).

`to_open` → IO

This is called (if defined) on the first parameter to IO.open.

`to_path` → String

1.9 This is called by the interpreter when it is looking for a filename (for example, by File#open).

`to_proc` → Proc

This is used to convert an object prefixed with an ampersand in a method call.

```
class OneTwo
  def to_proc
    proc { "one-two" }
  end
end
def silly
  yield
end
```

Download samples/ducktyping_16.rb

```
ot = OneTwo.new
silly(&ot)   # =>   "one-two"
```

to_regexp → Regexp

1.9

This is invoked by Regexp#try_convert to convert its argument to a regular expression.

to_str → String

This is used pretty much any place the interpreter is looking for a String value.

Download samples/ducktyping_17.rb

```
class OneTwo
  def to_str
    "one-two"
  end
end
ot = OneTwo.new
puts("count: " + ot)
File.open(ot) rescue puts $!.message
```

produces:

```
count: one-two
No such file or directory - one-two
```

to_sym → Symbol

This expresses the receiver as a symbol. This is used by the interpreter when compiling instruction sequences, but it's probably not useful in user code.

One last point is that classes such as Integer and Fixnum implement the to_int method, and String implements to_str. That way you can call the strict conversion functions polymorphically:

```
# it doesn't matter if obj is a Fixnum or a
# Roman number, the conversion still succeeds
num = obj.to_int
```

The Symbol.to_proc Trick

1.9

Ruby 1.9 implements the to_proc for objects of class symbol. Say you want to convert an array of strings to uppercase. You could write this:

```
names = %{ant bee cat}
result = names.map {|name| name.upcase}
```

That's fairly concise, right? Return a new array where each element is the corresponding element in the original, converted to uppercase. But, as of Ruby 1.9, you can instead write this:

```
names = %{ant bee cat}
result = names.map(&:upcase)
```

Now that's concise: apply the upcase method to each element of names.

So, how does it work? It relies on Ruby's type coercions. Let's start at the top.

When you say names.map(&xxx), you're telling Ruby to pass the Proc object in xxx to the map method as a block. If xxx isn't already a Proc object, Ruby tries to coerce it into one by sending it a to_proc message.

Now :upcase isn't a Proc object—it's a symbol. So when Ruby sees names.map(&:upcase), the first thing it does is try to convert the symbol :upcase into a Proc by calling to_proc. And, by an incredible coincidence, Ruby implements just such a method. If it was written in Ruby, it would look something like this:

```
def to_proc
  proc { |obj, *args| obj.send(self, *args) }
end
```

This method creates a Proc, which, when called on an object, sends that object the symbol itself. So, when names.map(&:upcase) starts to iterate over the strings in names, it'll call the block, passing in the first name and invoking its upcase method.

It's an incredibly elegant use of coercion and of closures. However, it comes at a price. The use of dynamic method invocations mean that the version of our code that uses &:upcase is about half as fast as the more explicitly coded block. This doesn't worry me personally unless I happen to be in a performance-critical section of my code.

Numeric Coercion

Back on page 376 we said there were three types of conversion performed by the interpreter. We covered loose and strict conversion. The third is numeric coercion.

Here's the problem. When you write 1+2, Ruby knows to call the + on the object 1 (a Fixnum), passing it the Fixnum 2 as a parameter. However, when you write 1+2.3, the same + method now receives a Float parameter. How can it know what to do (particularly because checking the classes of your parameters is against the spirit of duck typing)?

The answer lies in Ruby's coercion protocol, based on the method coerce. The basic operation of coerce is simple. It takes two numbers (one as its receiver, the other as a parameter). It returns a two-element array containing representations of these two numbers (but with the parameter first, followed by the receiver). The coerce method guarantees that these two objects will have the same class and therefore that they can be added (or multiplied, compared, or whatever).

```
1.coerce(2)         # =>  [2, 1]
1.coerce(2.3)       # =>  [2.3, 1.0]
(4.5).coerce(2.3)   # =>  [2.3, 4.5]
(4.5).coerce(2)     # =>  [2.0, 4.5]
```

The trick is that the receiver calls the coerce method of its parameter to generate this array. This technique, called *double dispatch*, allows a method to change its behavior based not only on its class but also on the class of its parameter. In this case, we're letting the parameter decide exactly *what* classes of objects should get added (or multiplied, divided, and so on).

Let's say that we're writing a new class that's intended to take part in arithmetic. To participate in coercion, we need to implement a coerce method. This takes some other kind of number as a parameter and returns an array containing two objects of the same class, whose values are equivalent to its parameter and itself.

For our Roman number class, it's fairly easy. Internally, each Roman number object holds its real value as a Fixnum in an instance variable, @value. The coerce method checks to see whether the class of its parameter is also an Integer. If so, it returns its parameter and its internal value. If not, it first converts both to floating point.

Download **samples/ducktyping_23.rb**

```
class Roman
  def initialize(value)
    @value = value
  end

  def coerce(other)
    if  Integer === other
      [ other, @value ]
    else
      [ Float(other), Float(@value) ]
    end
  end

  # .. other Roman stuff
end

iv = Roman.new(4)
xi = Roman.new(11)

3 * iv     # =>   12
1.1 * xi   # =>   12.1
```

Of course, class Roman as implemented doesn't know how to do addition. You couldn't have written xi+3 in the previous example, because Roman doesn't have a + method. And that's probably as it should be. But let's go wild and implement addition for Roman numbers:

Download **samples/ducktyping_24.rb**

```
class Roman
  MAX_ROMAN = 4999
  attr_reader :value
  protected :value
  def initialize(value)
    if value <= 0 || value > MAX_ROMAN
      fail "Roman values must be > 0 and <= #{MAX_ROMAN}"
    end
    @value = value
  end
```

```
  def coerce(other)
    if  Integer === other
      [ other, @value ]
    else
      [ Float(other), Float(@value) ]
    end
  end
  def +(other)
    if Roman === other
      other = other.value
    end
    if Fixnum === other && (other + @value) < MAX_ROMAN
      Roman.new(@value + other)
    else
      x, y = other.coerce(@value)
      x + y
    end
  end
  FACTORS = [["m", 1000], ["cm", 900], ["d",  500], ["cd", 400],
             ["c",  100], ["xc",  90], ["l",   50], ["xl",  40],
             ["x",   10], ["ix",   9], ["v",    5], ["iv",   4],
             ["i",    1]]
  def to_s
    value = @value
    roman = ""
    for code, factor in FACTORS
      count, value = value.divmod(factor)
      roman << (code * count)
    end
    roman
  end
end
```

Download **samples/ducktyping_25.rb**

```
iv = Roman.new(4)
xi = Roman.new(11)

iv + 3         # =>   vii
iv + 3 + 4     # =>   xi
iv + 3.14159   # =>   7.14159
xi + 4900      # =>   mmmmcmxi
xi + 4990      # =>   5001
```

Finally, be careful with coerce—try always to coerce into a more general type, or you may end up generating coercion loops. This is a situation where A tries to coerce to B, and B tries to coerce back to A.

Walk the Walk, Talk the Talk

Duck typing can generate controversy. Every now and then a thread flares on the mailing lists or someone blogs for or against the concept. Many of the contributors to these discussions have some fairly extreme positions.

Ultimately, though, duck typing isn't a set of rules; it's just a style of programming. Design your programs to balance paranoia and flexibility. If you feel the need to constrain the types of objects that the users of a method pass in, ask yourself why. Try to determine what could go wrong if you were expecting a String and instead get an Array. Sometimes, the difference is crucially important. Often, though, it isn't. Try erring on the more permissive side for a while, and see whether bad things happen. If not, perhaps duck typing isn't just for the birds.