shortcut

Rubyisms in Rails

Jacob Harris

Addison-Wesley
Pearson Education

www.awprofessional.com/ruby

Visit us on the Web: www.awprofessional.com

ISBN 0-321-47407-4

First release, July 2006

LISTING 3 A Simple Bird-Watching Class

```
class BirdWatcher
  include Enumerable

  def initialize
    @birds = {}
  end

  def saw_bird(bird)
    if @birds[bird].nil?
      @birds[bird] = 1
    else
      @birds[bird] += 1
    end
  end
end
```

Yet, nothing happens when we interpret that class declaration in `irb`. It is even possible to create and use the object without any errors being thrown. In fact, an exception is thrown only when you attempt to call one of the methods from `Enumerable` that uses `each`:

```
irb> b = BirdWatcher.new
irb> b.saw_bird("Robin")
irb> b.saw_bird("Sparrow")
… # 23 sparrows later
irb> b.saw_bird("Sparrow")
irb> common_birds = b.select {¦bird , count¦ count > 10 }
NoMethodError: undefined method 'each' for #<BirdWatcher:0x356950 @birds={"Sparrow"=>24,
"Robin"=>1}>
        from (irb):11:in 'select'
        from (irb):11
        from :0
```

In this example, we use the `Enumerable#select` method to obtain an array of common birds out of our overall sighting statistics. Don't worry about that weird syntax after `select` in there (we get to that later). Instead, notice that the error is thrown only for the first time we use the `select` method, when it in turn tries to call `each`. Interestingly enough, this is easily fixed within `irb` if we define an `each` function after the fact:

```
irb> class BirdWatcher
irb> def each
irb> @birds.each {¦bird, count¦ yield  bird, count }
irb> end
irb> end
=> nil
irb> common_birds = b.select {¦name, count¦ count > 3 }
=> [["Sparrow", 4]]
```

See whether you can understand what we are doing here in the context of the previous section. (Again, do not worry about the syntax of that `each` statement yet.) We are just extending an existing class with `irb` in the same way that Rails adds new methods to `Numeric`. Let's pose a different question then. How does Ruby figure out what methods an object can call? Why doesn't Ruby get upset that we have added some methods when the underlying method they call is undefined? What type system does Ruby use anyway?

In a strongly typed language such as Java, type is paramount. An object's type is its class, and only variables of the same class can use its methods. Variables whose type is a superclass or interface can also use the object, but they can only call methods defined at or above their level, an information-hiding principle generally known as **abstraction**. Thanks to C++ and Java, this approach has largely become canon. But why do we need types in those languages? In the early days of OOP, types were needed so that the compiler could determine exactly which methods would be invoked where; this in turn allows the compiler to optimize method calls because it has all the information it needs at compilation time. In most modern statically typed object-oriented languages, however, method resolution is performed at runtime anyway. Instead of being an optimization technique, static type checking is a security mechanism screening callers: A caller can use an object only if it matches the type he is expecting. Otherwise, the program does not compile. This assurance is known as **type safety**, and the goal of it has been to build a web of types and expectations to ensure that nothing unexpected occurs in the program. This sounds great, except for all the ways in which it no longer applies.

As the saying goes, the road to hell is paved with good intentions. (That is one back road we won't be traveling down in this shortcut.) The goal of type safety is to eliminate unexpected behavior (that is, bugs) through the contracts of types, but this approach has several shortcomings. The first problem is the massive amount of bureaucracy required for static typing. For instance, consider an example of Java code we might need to get a `datetime` out of a generic collection of events generated by a system logger:

```
Datetime dt = (Datetime) ((Event) eventQueue.front).getDatetime();
```

Do you think that `dt` variable might possibly be a `datetime` there? Now repeat for every line of your program: It is a lot like writing a novel by focusing solely on the punctuation. Still, if type safety works well, this might seem like a fair cost. Unfortunately, however, this example does not even have that safety. If you look again, it is doing a dynamic runtime cast of an item retrieved from the top of `eventQueue` (which, in this example, is just a quickly named generic queue whose values are stored as type `Object`). Because this is a dynamic cast, the type coercion and checking happens at runtime as opposed to compilation time, thus bypassing compilation's type safety mechanics entirely. Of course, the smart Java programmer might say that I could mitigate this issue by creating an `EventQueue` object that encapsulates the generic queue with properly typed input and output functions. And that programmer is right, too, except that then the programmer is providing extra logic to correct a failing of the type safety net. In addition, many programmers would just leave it at the cast anyway. In any event, this problem gets worse when you read data from generic sources, be they distributed architectures such as DCOM/CORBA, database layers such as ADO/ODBC, or even certain web services. In those cases, you will likely have to do dynamic casting from a generic object or variant type. Meaning, the notion of all-encompassing type safety fails at the very place you want it most: protecting the integrity of your data.

True type safety is an illusion. Without type safety, however, are your programs destined to a buggy doom? Not really. Type safety is definitely good for eliminating a basic class of bugs, but the problem is that program errors are often more subtle than confusing an integer for a string. Type safety is like a broad contract; it can guarantee general conditions (that a field is a string, or an object is of a particular transaction type), but not specific uses. Unfortunately, correct programs often require more specific constraints—for example, it is not enough that a customer ID is typed as an integer; it must be an integer in a particular range that validates against internal checksums—that go beyond what can be expressed with mere types. In other words, static types can never replace full-fledged testing, but too many programmers act like it does. Compiling is not testing; *testing* is testing. Bugs happen in both statically typed and dynamically typed programs, and the best defense is unit testing or some other more-rigorous testing approach. Because it is not compiled and has no static types, Ruby might seem to be more dangerous to develop in; but in some sense, it can actually be safer to program in: Because developers have no illusions that type checking is a serious defense against bugs, they will not confuse compilation with correctness.

Ruby takes a more enlightened approach to type: Ruby just does not care. Of course, in Ruby, objects do have classes, and you can also easily inspect any object's class if that is important to you. If you program in the Ruby style, however, you never need to care. Classes are still useful in Ruby as a way of defining methods for objects, but they are not important as a typing framework. Ruby does not check that a variable has the right class type for a call; instead, the only important thing Ruby cares about is this: *Does this object have this method I'm calling at this instant*? Methods are all that matters; the class is not used at all. This approach is commonly referred to as **duck typing**, after the aphorism "If it talks like a duck and walks like a duck, it is a duck." The class of the object has no effect, and you can call a method without needing to cast the object appropriately. The result is cleaner, simpler, and more-understandable code. For instance, we could rewrite our sample Java `datetime` snippet in Ruby as follows:

```
dt = eventStack.front.getDatetime
```

Without all the type coercion, that code certainly becomes a lot simpler. When executing that line, Ruby merely checks that the `eventStack` variable contains a `front` method and the value popped contains a `getDatetime` method to run in turn. I am repeating this over and over—classes and types are irrelevant to how Ruby handles method calls, they only exist to define object methods—because I know at least a few of you were still trying to figure out how Ruby determines and validates the classes are correct in that call. The `eventStack` variable could have the class `EventStack` or the class `RandomStuffHappening` or the class `Giraffe`; as long as it has a method named `front`, Ruby is happy. Don't feel bad if you still have that preoccupation with classes and types; old habits die hard, and the importance of types has been drummed into your head for years by every computer science textbook. If you really want to program in the Ruby way, however, you need to get over that. Forget about figuring out object types. And when you get used to duck typing, you will find it hard to go back.

Symbols

When you start programming in Ruby on Rails, you see a lot of funny little constructions that look somewhat like strings but start with colons rather than quotation marks:

```
link_to "log in", :controller => "accounts", :action => "login" options = { :type => 'text/xml',
:disposition => 'inline' }
Category.find(:all, :order => :position, :limit => 1000)
```

You might wonder what that funny little colon is doing in the front of `:controller` or `:action`. "Those are symbols, and they are very important," the typical Rails tutorial replies, "but we have to move on." Okay, but what are symbols? A symbol in Ruby is basically an immutable string. This is a useful abstraction for representing enumerated values in terms of performance and memory consumption. Because symbols are immutable, multiple references can refer to a single spot of storage in memory. It does not matter whether there are one or one million variables with the value `:action` in your code; they all use a single shared location in memory. In addition, comparing whether two symbol references are the same is trivial, because you need merely check that they point to the same location in memory. Symbols are certainly efficient.