CSE 341, Spring 2011, Lecture 19 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

This dislaimer is especially true for this sort of introductory "get used to how Ruby feels" lecture.

This lecture is an introduction to Ruby. The corresponding code demonstrates many different language features, and the course textbook also provides a quite reasonable introduction to the language. Therefore, this summary will focus more on some of the concepts we considered rather than explaining every language feature we discussed. Our overlap with the textbook will be roughly chapters 1–9 of the 2nd, Ruby 1.8 edition (or chapters 1–10 of the 3rd edition, or chapters 1–8 of the free first edition), but we will not cover regexps or ranges. Also, our focus will be on the overall concepts of object-oriented programming, dynamic typing, blocks, and mixins, rather than small language details.

Ruby is:

- *Pure* object-orinted: *all* values are objects. In Java, some values that are not objects are `null`, `13`, `true`, and `4.0`. In Ruby, every expression evaluates to an object.
- Class-based: Every object is an instance of a class. An object's class determines what methods an object has. As in Java, you call a method "on" an object, e.g., obj.m(3,4) evaluates the variable `obj` to an object and calls its `m` method with arguments 3 and 4.
- Pure object-oriented (again): Most expressions are either variables (looking an object in the environment) or method calls. In fact, `3+4` is just syntactic sugar for `3.+(4)`, i.e., calling the `+` method on the 3 object with argument 4, which is itself an object.
- Dynamically typed: You can call any method on any object. If the object's class does not define such a method (or if the number of arguments is wrong), it is a run-time error, just as in Scheme many things are run-time errors. As such, we do not have types for method arguments, method results, fields, etc. Everything is an object.
- Convenient *reflection*: Various built-in methods make it easy to discover at run-time properties about objects. As examples, every object has a method `class` that returns the object's class, and a method `methods` that returns an array of the object's methods.
- A "scripting language": This term defies a precise definition. It means the language is engineered toward making it easy to write short programs, providing convenient access to manipulating files and strings. It also does not require that you declare variables before using them (more on this below). Also, there is no publicstaticvoidmain method or equivalent. A Ruby program is a sequence of statements (actually expressions), including method and class declarations, and these are executed in order.

Recall that after learning ML, Scheme, Ruby, and Java, you will have seen all four combinations of functional vs. object-oriented and statically vs. dynamically typed.

Our focus will be on Ruby's object-oriented nature, not on its benefits as a scripting language. We also won't discuss at all its support for building web applications, which is a main reason it is currently so popular. As an object-oriented language, Ruby shares much with Smalltalk, a language that has basically not changed since 1980. Ruby does have some nice additions we will study, such as mixins.

Ruby is also a large language with a "why not" attitude, especially with regard to syntax. ML and Scheme (and Smalltalk) adhere rather strictly to certain traditional programming-language principles, such as defining a small language with powerful features that programmers can then use to build large libraries. Ruby often takes the opposite view. For example, there are many different ways to write an if-expression.

Here are a few notes on syntax that can seem "unusual" after seeing Java, ML, and Scheme:

- You can call a zero-argument method with `e.m()` or `e.m`, i.e., the parentheses are optional.
- There are several forms of conditional expressions, including `e1 if e2` (all on one line), which evaluates `e1` only if `e2` is true (i.e., it reads right-to-left).
- Newlines are often significant. For example, you can write

```
if e1
  e2
else
  e3
end
```

  But if you want to put this all on one line you need to write `if e1 then e2 else e3 end`, or `if e1 : e2 else e3 end`. Note, however, indentation is never significant (only a matter of style).
- Field names always begin with the `@` character (e.g., `@foo`), class names begin with a capital letter, and method and variable names begin with a lower-case letter. Finally, class fields (same idea as Java's static fields) start with `@@`.
- You can define a method with a name that ends in `=`, for example:

```
def foo= x
  @blah = x * 2
end
```

  As expected, you can write `e.foo=(17)` to change `e`'s `foo` field to be 34. Better yet, you can adjust the parentheses and spacing to write `e.foo = 17`. This is just syntactic sugar. It "feels" like an assignment statement, but it is really a method call. Stylistically you do this for methods that mutate an object's state in some "simple" way (like setting a field).
- Where you write `this` in Java, you write `self` in Ruby.
- A class's methods do not all have to be defined in the same place. If you write `class Foo ... end` multiple times in a program, all the methods are part of class Foo. (Any repeated methods *replace* earlier definitions, even for instances of the class that have already been created.)

Some object basics:

- To create a new object, call `Foo.new` with any number of arguments where `Foo` is a class. This will create an object and then call its `initialize` method. That is, `initialize` is the one constructor for the object. The number of arguments it takes is the number that should be passed to `new` (though like all methods, you can define default arguments, allowing callers to pass fewer arguments).
- A class definition contains methods, but does not define what fields are in instances of the class. Any method of the class that assigns to a field makes `self` have that field, even if it didn't have it before. So different instances of a class can have different fields. Now, typically one does not use this flexibility. If `initialize` always assigns to some field, then all instances will have that field. And if all methods assign only to fields always assigned to by `initialize`, then all instances of a class will have the same fields.
- Methods can be `public`, `protected`, or `private`. Any code can invoke an object's public methods. Only other methods defined in the same class or a subclass (we'll see subclassing next lecture) can invoke a protected method (else it's a run-time error). Finally, private methods can be invoked only by other methods of the same object, not even other objects of the same class. Therefore, if `m` is private, then `e.m` should be allowed only if `e` evaluates to the `self` object. So `e` would always be redundant (since like in Java, if you omit a method call's *receiver*, then `self` is implicit). So it turns out you may *not* write `self.m` if `m` is private, you have to write just `m` (plus zero or more arguments, of course).

- Fields are always private. A method can only refer to fields of `self`, by writing `@foo` for field foo. Therefore, you end up writing a lot of getter and setter methods for fields that you do want other objects to be able to access. By convention, it makes sense for a getter method to have the same name as a field, e.g.,

```
def blah
  @blah
end
```

  and similarly a setter method could be:

```
def blah= x
  @blah = x
end
```

  Note it is often good style for these methods to be protected (or even private). Also, defining getters and setters is so common that there is a shorter syntax. To define a getter method named `foo` for field `@foo`, you can write `attr_reader :foo`, and similarly `attr_writer :foo` defines a setter.

Much like fields come into existence when you assign to them, local variables in methods come into existence when you assign to them. They exist until the method completes evaluating. For both fields and variables, this semantics leads to shorter programs, but it is even less eager than Scheme about detecting errors. For example, suppose you have a variable `blah` you want to set to 0, but you accidentally type `plah=0`. No error occurs, you simply created a new variable and `blah` still holds its previous value.

It is common to define some methods outside of any explicit class. These are like top-level functions in ML or Scheme. In actuality, they are in a class named `Main`, but it's fine to think of them as functions.

The value `nil` is used idiomatically like you use `null` in Java. However:

- Like in Scheme, any value (i.e., any object) can be a test for an if-expression. And like in Scheme, almost everything counts as true. However, `false` *and* `nil` count as false.
- `nil` is an object. Its class is `NilClass`. There are many methods defined on it because there are many methods that work for all objects.
- Like in ML or Scheme, every expression produces a value (there is no separate notion of statements). Sometimes when there is no other sensible value to produce, `nil` is the result.

One feature we mentioned in passing is that strings built with double-quotation marks can have expressions embedded within them. For example, `"The value of x concatenated with y is #{x+y}"` will produce the string `"The value of x concatenated with y is foobar"` if `x` is bound to `"foo"` and `y` is bound to `"bar"`. The expression between `#{` and `}` is simply evaluated (here looking up `x` and calling its `+` method with `y`) and the result is converted to a string (by calling the `to_s` method; this method is defined in class `String` just to return `self`). This sort of feature is a syntactic convenience, and you do not need to use it in this course. But we bring it up because it is exactly the same concept as Scheme's quasiquote and unquote applied to strings instead of lists.

Finally, let's consider how dynamic typing interacts with method calls a bit more. Recall we can pass any object to any method and call any method on any object (simply getting a run-time error if the object's class does not define the method or we pass the wrong number of arguments). So given a method like:

```
def foo x
  x.m + x.n
end
```

we can pass any object that has zero-argument methods `m` and `n` that are numbers right? Yes, but it is even more flexible than that. We just need that the object returned by `m` has a `+` method that takes one argument and the object returned by `n` has enough methods defined for the body of the `+` method that gets called.

This dynamic typing can lead to hard-to-diagnose errors (since calling `foo` with a different argument than you intended might still work), but can also be convenient. We considered an example where we defined a `+` method for class `PosRat` and then a doubling function "automatically" worked for instances of `PosRat`. This is essentially the concept of "duck typing," discussed in the next lecture.