CSE 341, Autumn 2015, Ruby Introduction Summary

Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

Introducing Ruby

This lecture is an introduction to Ruby. The corresponding code demonstrates many different language features. Because the book *Programming Ruby, 2nd Edition* by Dave Thomas and various free online tutorials are more than sufficient, the lecture materials may not describe in full detail every language feature we use.

The course website provides installation and basic usage instructions for Ruby. Note in particular that officially we will be using version 2.2.3 of the language. You will need the tk graphics library for Assignment 7.

Ruby Features Most Interesting for a PL Course

Ruby is a large, modern programming language with various features that make it popular. Some of these features are useful for a course on programming-language features and semantics, whereas others are not useful for our purposes even though they may be very useful in day-to-day programming. Our focus will be on object-oriented programming, dynamic typing, blocks (which are almost closures), and mixins. We briefly describe these features and some other things that distinguish Ruby here:

- Ruby is a *pure object-oriented* language, which means *all* values in the language 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.
- Ruby is *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. Not all object-oriented languages are class-based; see, for example, Javascript.
- Ruby has *mixins*: A later lecture will describe mixins, which strike a reasonable compromise between C++'s multiple inheritance and Java's interfaces. Like Java, every Ruby class has one superclass, but it can include any number of mixins, which, unlike interfaces, can define methods (not just require their existence).
- Ruby is *dynamically typed*: Just as Racket allowed calling any function with any argument, Ruby allows calling any method on any object with any arguments. If the *receiver* (the object on which we call the method) does not define the method, we get a dynamic error.
- Ruby has *many dynamic features*: In addition to dynamic typing, Ruby allows instance variables (Java's fields) to be added and removed from objects and it allows methods to be added and removed from classes while a program executes.
- Ruby has *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.
- Ruby has *blocks* and *closures*: Blocks are almost like closures and are used throughout Ruby libraries for convenient higher-order programming. Indeed, it is rare in Ruby to use an explicit loop since collection classes like `Array` define so many useful iterators. Ruby also has fully-powerful closures for when you need them.

- Ruby is a *scripting language*: There is no precise definition of a what makes a language a scripting language. It means the language is engineered toward making it easy to write short programs, providing convenient access to manipulating files and strings (topics we won't discuss), and having less concern for performance. Like many scripting languages, Ruby does not require that you declare variables before using them and there are often many ways to say the same thing.
- Ruby is *popular for web applications*: The Ruby on Rails framework is a popular choice for developing the server side modern web-sites.

Recall that, taken together, Haskell, Racket, Ruby, and Java, cover 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, such as mixins.

Ruby is also a large language with a "why not" attitude, especially with regard to syntax. Haskell and Racket (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.

Objects, Classes, Methods, Variables, Etc.

The code associated with this lecture contains an example class definition for rational numbers, which is a useful complement to the general rules described here.

Class and method definitions

Since every *object* has a *class*, we need to define classes and then create *instances* of them (an object of class `C` is an instance of `C`). (Of course, Ruby also predefines many classes in its language and standard library.) The basic syntax (we will add features as we go) for creating a class `Foo` with *methods* `m1`, `m2`, ... `mn` can be:

```
class Foo
   def m1
     ...
   end

   def m2 (x,y)
     ...
   end

   ...

   def mn z
     ...
   end
end
```

Class names must be capitalized. They include method definitions. A method can take any number of arguments, including 0, and we have a variable for each argument. In the example above, `m1` takes 0 arguments, `m2` takes two arguments, and `mn` takes 1 argument. Not shown here are method bodies. Like Haskell and Racket functions, a method implicitly returns its last expression. Like Java, you can use an explicit `return` statement to return immediately when helpful. (It is bad style to have a return at the end of your method since it can be implicit there.)

Method arguments can have defaults in which case a caller can pass fewer actual arguments and the remaining ones are filled in with defaults. If a method argument has a default, then all arguments to its right must also have a default. An example is:

```
def myMethod (x,y,z=0,w="hi")
   ...
end
```

Instance variables

An object has a class, which defines its methods. It also has *instance variables*, which hold values (i.e., objects). Many languages, including Java, use the term *fields* instead of instance variables for the same concept. Unlike Java, our class definition does not indicate what instance variables an instance of the class will have. To add an instance variable to an object, you just assign to it: if the instance variable does not already exist, it is created. All instance variables start with an `@`, e.g., `@foo`, to distinguish them from variables local to a method. (Ruby also has class variables, which are like Java's static fields. They are written `@@foo`.)

Each object has its own instance variables. Instance variables are mutable. An expression (in a method body) can read an instance variable with an expression like `@foo` and write an instance variable with an expression `@foo = newValue`.

Instance variables are private to an object. There is no way to directly access an instance variable of any other object.

Calling methods

The method call `e0.m(e1, ..., en)` evaluates `e0`, `e1`, ..., `en` to objects. It then calls the method `m` in the result of `e0` (as determined by the class of the result of `e0`), passing the results of `e1`, ..., `en` as arguments. As for syntax, the parentheses are optional. In particular, a zero-argument call is usually written `e0.m`, though `e0.m()` also works.

To call another method on the same object as the currently executing method, you can write `self.m(...)` or just `m(...)`. (Java works the same way except it uses the keyword `this` instead of `self`.)

In OOP, another common name for a method call is a *message send*. So we can say `e0.m e1` sends the result of `e0` the message `m` with the argument that is the result of `e1`. This terminology is "more object-oriented" — as a client, we do not care how the receiver (of the message) is implemented (e.g., with a method named `m`) as long as it can handle the message.

Constructing an object

To create a new instance of class `Foo`, you write `Foo.new (...)` where `(...)` holds some number of arguments (where, as with all method calls, the parentheses are optional and when there are zero or one arguments it is preferred to omit them). The call to `Foo.new` will create a new instance of `Foo` and then, before `Foo.new` returns, call the new object's `initialize` method with all the arguments passed to `Foo.new`. That is, the method `initialize` is special and serves the same role as Java's constructors.

Typical behavior for `initialize` is to create and initialize instance variables. In fact, the normal approach is for `initialize` always to create the same instance variables and for no other methods in the class to create instance variables. But Ruby does not require this and it may be useful on occasion to violate these conventions.

It is a run-time error to call `Foo.new` with a number of arguments that the `initialize` method for the class cannot handle.

Expressions and Local Variables

Most expressions in Ruby are actually method calls. Even `e1 + e2` is just syntactic sugar for `e1.+ e2`, i.e., call the `+` method on the result of `e1` with the result of `e2`. Another example is `puts e`, which prints the result of `e` (after calling its `to_s` method to convert it to a string) and then a newline. It turns out `puts` is a method in all objects (it is defined in class `Object` and all classes are subclasses of `Object` — the next lecture discusses subclasses), so `puts e` is just `self.puts e`.

Not every expression is a method call. The most common other expression is some form of conditional. There are various ways to write conditionals; see the example code. As the next lecture discusses, loop expressions are rare in Ruby code.

Like instance variables, variables local to a method do not have to be declared: The first time you assign to `x` in a method will create the variable.

Everything is an Object

Everything is an object, including numbers, booleans, and `nil` (which is often used like `null` in Java). For example, `-42.abs` evaluates to `42` because the `Fixnum` class defines the method `abs` to compute the absolute value and `-42` is an instance of `Fixnum`. (Of course, this is a silly expression, but `x.abs` where `x` currently holds `-42` is reasonable.)

All objects have a `nil?` method, which the class of `nil` defines to return `true` but other classes define to return `false`. Like in Haskell and Racket, every expression produces a result, but when no particular result makes sense, `nil` is preferred style (much like Haskell's `()` and Racket's void-object). That said, it is often convenient for methods to return `self` so that subsequent method calls to the same object can be put together. For example, if the `foo` method returns `self`, then you can write `x.foo(14).bar("hi")` instead of

```
x.foo(14)
x.bar("hi")
```

Some Syntax, Semantics, and Scoping To Get Used To

Ruby has a fair number of quirks that are often convenient for quickly writing useful programs but may take some getting used to. Here are some examples; you will surely discover more.

- 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`. Note, however, indentation is never significant (only a matter of style).
- Conditionals can operate on any object and treat every object as "true" with *two* exceptions: `false` and `nil`.
- 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.
- The methods of a class 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.)
- Remember variables (local, instance, or class) get automatically created by assignment, so if you misspell a variable in an assignment, you end up just creating a different variable.

Visibility, Getters/Setters

As mentioned above, instance variables are private to an object: only method calls with *that object* as the receiver can read the fields. As a result, the syntax is `@foo` and the self-object is implied. The syntax `self.@foo` is not allowed since it is redundant and `x.@foo` would break the privacy rules. Notice even other instances of the same class cannot access the instance variables. This is quite object-oriented: you can interact with another object only by sending it messages.

Methods can have different *visibilities*. The default is `public`, which means any object can call the method. There is also `private`, which like with instance variables, allows only the object itself to call the method (from other methods in the object). In-between is `protected`: A protected method can be called by any object that is an instance of the same class or any subclass of the class.

There are various ways to specify the visibility of a method. Perhaps the simplest is within the class definition you can put `public`, `private`, or `protected` between method definitions. Reading top-down, the most recent visibility specified holds for all methods until the next visibility is specified. There is an implicit `public` before the first method in the class.

To make the contents of an instance variable available and/or mutable, we can easily define getter and setter methods, which by convention we can give the same name as the instance variable. For example:

```
def foo
  @foo
end

def foo= x
  @foo = x
end
```

If these methods are public, now any code can access the instance variable `@foo` indirectly, by calling `foo` or `foo=` (and, as noted above, calls to the latter can be written as `e1.foo = e2`). It often makes sense to instead make these methods `protected`. The `Rational` class in the associated code uses protected getter methods to good effect: The getters are needed to implement addition by another instance of `Rational`, but we do not make the numerator and denominator publicly available.

The advantage of the getter/setter approach is it remains an implementation detail that these methods are implemented as getting and setting an instance variable. We, or a subclass implementer, could change this decision later without clients knowing. We can also omit the setter to ensure an instance variable is not mutated except perhaps by a method of the object.

Because getter and setter methods are so common, there is shorter syntax for defining them. For example, to define getters for instance variables `@x`, `@y`, and `@z` and a setter for `@x`, the class definition can just include:

```
attr_reader :x, :y, :z
attr_writer :x
```

Top-Level

You can define methods, variables, etc. outside of an explicit class definition. The methods are implicitly added to class `Object`, which makes them available from within any object's methods.

Top-level expressions are evaluated in order when the program runs. So instead of Ruby specifying a main class and method with a special name (like `main`), you can just create an object and call a method on it at top-level.

Dynamic Class Definitions

A Ruby program (or a user of the REPL) can change class definitions while a Ruby program is running. Naturally this affects all users of the class. Perhaps surprisingly, it even affects instances of the class that have already been created. That is, if you create an instance of `Foo` and then add or delete methods in `Foo`, then the already-created object "sees" the changes to its behavior.

This is usually dubious style, but it leads to a simpler language definition: defining classes and changing their definitions is just a run-time operation like everything else. It can certainly break programs: If I change or delete the `+` method on numbers, I would not expect many programs to keep working correctly. It can be useful to add methods to existing classes, especially if the designer of the class did not think of a useful helper method.

Duck Typing

Duck typing refers to the expression, "If it walks like a duck and quacks like a duck, then it's a duck" though a better conclusion might be, "then there is no reason to concern yourself with the possibility that it might not be a duck." In Ruby, this refers to the idea that the class of an object (e.g., "Duck") passed to a method is not important so long as the object can respond to all the messages it is expected to (e.g., "walk to x" or "quackNow").

For example, consider this method:

```
def mirror_update pt
  pt.x = pt.x * -1
end
```

It is natural to view this as a method that must take an instance of a particular class `Point` (not shown here) since it uses methods `x` and `x=` defined in it. And the `x` getter must return a number since the result of `pt.x` is sent the `*` message with `-1` for multiplication.

But this method is more generally useful. It is not necessary for `pt` to be an instance of `Point` provided it has methods `x` and `x=`.

Moreover, the `x` and `x=` methods need not be a getter and setter for an instance variable `@x`.

Even more generally, we do not need the `x` method to return a number. It just has to return some object that can respond to the `*` message with argument `-1`.

Duck typing can make code more reusable, allowing clients to make "fake ducks" and still use your code. In Ruby, duck typing basically "comes for free" as long you do not explicitly check that arguments are instances of particular classes using methods like `instance_of?` or `is_a?` (see next lecture).

Duck typing has disadvantages. The most lenient specification of how to use a method ends up describing the whole implementation of a method, in particular what messages it sends to what objects. If our specification reveals all that, then almost no variant of the implementation will be equivalent. For example, if we know `i` is a number (and ignoring clients redefining methods in the classes for numbers), then we can replace `i+i` with `i*2` or `2*i`. But if we just assume `i` can receive the `+` message with itself as an argument, then we cannot do these replacements since `i` may not have a `*` method (breaking `i*2`) or it may not be the sort of object that `2` expects as an argument to `*` (breaking `2*i`).