1 Object-oriented languages

We now begin to study language features used for object-oriented programming (OOP). Just like we used SML as a vehicle to study functional language features, we’ll use Ruby to study object-oriented (OO) features.

Why Ruby?

- 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. As a pure object-oriented language, Ruby shares much with Smalltalk, a language that has basically not changed since 1980.

- 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, method are called “on” objects. For example, obj.m(3,4) evaluates variable obj to an object then calls its m method with arguments 3 and 4. Not all object-oriented languages are class-based; see, for example, Javascript.

- Ruby has mixins: Mixins 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: 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, a run-time error occurs.

- Ruby has many dynamic features: Ruby allows instance variables (which are similar to Java fields) to be added and removed from objects, and it allows methods to be added and removed from classes during execution.

- 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 (called procs) when you need them.

There are other reasons Ruby is popular. These features won’t be our focus, even though they might be very useful in day-to-day programming.

- Ruby is a scripting language: It is engineered for small 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 a large language with a “why not” attitude, especially with regard to syntax. ML (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.
• Ruby is popular for web applications: The Ruby on Rails framework is a popular choice for developing the server side of modern web-sites.

Pragmatics. Because the book Programming Ruby 1.9 by Dave Thomas and various free online tutorials are more than sufficient, the lecture materials will 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 we will be using version 1.9 of the language, not the older Ruby 1.8.

2 Introduction to Ruby

The code provided with this lecture contains complementary information; make sure to study it, too.

Class and method definitions. Every object has a class. Ruby has many predefined classes, and we can define our own. An object of class \( C \) is an instance of \( C \). The basic syntax for creating a class \( \text{Foo} \) with methods \( \text{m1}, \text{m2}, \ldots, \text{mn} \) is:

```ruby
class Foo
  def m1
    ...
  end

  def m2 (x,y)
    ...
  end

  ...

  def mn z
    ...
  end
end
```

Class names must be capitalized. A method can take any number of arguments, including 0. In the example above, \( \text{m1} \) takes 0 arguments, \( \text{m2} \) takes two arguments, and \( \text{mn} \) takes 1 argument. Method bodies are not shown here.

A method implicitly returns the value of its last expression. Like Java, you can use an explicit \texttt{return} statement in the middle of a method to return immediately. (It is bad style to have a \texttt{return} at the end of a method since it should be implicit there.)

Method arguments can have default values, 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:

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

Instance variables. An object also has instance variables, which hold values (which are themselves objects). Many languages, including Java, use the term fields instead of instance variables for the same concept. But unlike Java, you do not declare instance variables in Ruby. That is, Ruby class definitions do not indicate what instance variables objects 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 @, e.g., @foo, to distinguish them from local variables inside the body of a method. Ruby also has class variables, which are like Java’s static fields. All class variables start with @@, e.g., @@foo.

Each object has its own instance variables that are distinct from all other objects’ instance variables. An expression (in a method body) can read an instance variable named @foo simply with expression @foo and write with expression @foo = e.

Instance variables are private to an object. There is no way from within an object to directly access an instance variable of any other object.[1]

Assignment is mutation. Assigning to an instance variable mutates it. That is, an instance variable is actually a reference to an object, and by assigning to the variable, you are changing the reference. So just as in Java, you must now worry about aliasing again. Have fun!

Calling methods. Method call e0.m(e1, ..., en) evaluates e0, ..., en to objects o0, ..., on. It then calls method m in o0 (as determined by o0’s class), passing o1, ..., on as arguments. In OOP, another common name for a method call is a message send. So we can say o0.m(o1) “sends message m” to object o0 with argument o1.

The parentheses in method calls are optional. In particular, a zero-argument call is usually written e0.m, though e0.m() also works. They’re even optional for multiple arguments: either e0.n(x,y) or e0.n x,y works.

To send a message to the same object that received the currently executing message, write self.m(...) or just m(...). Keyword self means the currently “executing” object. It’s analogous to Java’s keyword this.

It’s a common idiom for methods to return self so that subsequent method calls to the same object can be put together. For example, if foo returns self, then you can write x.foo(14).bar("hi") instead of having to write

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

Constructing objects. Expression Foo.new arglist creates a new instance of class Foo, where arglist is zero or more arguments, just like in a method call. The call to Foo.new will create a new instance of Foo and, before Foo.new returns, call the new object’s initialize method with all the arguments passed to Foo.new. Method initialize thus is analogous to Java constructors. It is a run-time error to call Foo.new with a number of arguments that the initialize method for the class cannot handle.

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.

Expressions and local variables. Most expressions in Ruby are actually just syntactic sugar for method calls. For example, e1 + e2 is just syntactic sugar for e1.+(e2)—that is, call the + method on the result of e1 with the result of e2. Another example is puts e, which is really just self.puts e. Method puts—which is defined in class Object, the superclass of all objects—calls e’s to_s method to convert it to a string then prints that string followed by newline.

[1] Actually, there is, but it’s considered very poor style.
But not every expression is a method call. Conditional (if) expression aren’t. 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 a method assigns to $x$, that variable will be created.

**Everything is an object.** Everything is an object, including numbers, booleans, and nil (which is often used like null in Java). For example, -42 is an object of class Fixnum. Methods can be invoked on it: -42.abs evaluates to 42, because Fixnum defines method abs to compute absolute values.

Like in ML, every expression produces a result, but when no particular result makes sense, nil is a possible return value (much like ML’s unit value ())). All objects have a nil? method. (Yes, the question mark is part of the name of the method. Methods so-named conventionally return booleans.) The only object for which nil? returns true is nil itself—unless you define a class that violates that convention, which isn’t normally a good idea.

### 3 Some Ruby quirks

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 $e_1$ if $e_2$ (all on one line), which evaluates $e_1$ only if $e_2$ is true (i.e., it reads right-to-left).
- Newlines are often significant. For example, you can write

```
if $e_1$
  $e_2$
else
  $e_3$
end
```

But if you want to put this all on one line you need to write if $e_1$ then $e_2$ else $e_3$ end. Note, however, indentation is never significant (even though it’s certainly a matter of style).

- Conditionals can take any object as a guard. Only two particular objects will be treated as false: the object false and the object nil.
- You can define a method with a name that ends in =, for example:

```
def foo= $x
  @bar = $x * 2
end
```

As expected, you can write $e$.foo=(17) to change $e$’s @bar instance variable 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). Of course, the most natural thing is to name the method after the instance variable:

```
def foo= $x
  @$foo = $x
end
```
That provides a way for code outside the class to mutate instance variables.

- 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 so-defined are part of class `Foo`. Any repeated methods replace earlier definitions—even for instances of the class that have already been created.
- Variables (whether local, instance, or class) are created by assignment, so if you misspell a variable in an assignment, you won’t get an error. Instead, you end up creating a different variable.
- 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.

4 Visibility, Getters, and Setters

As mentioned above, instance variables are accessible only from inside an object: only method calls with `that object` as the receiver can read the fields. For accessing instance variables, Ruby’s syntax is therefore just `@foo`. The syntax `self.@foo` is not allowed (it would be redundant), and syntax `x.@foo` is illegal, because it would break the privacy rules. Note that even other instances of the same class cannot access the instance variables. This is quite object-oriented: one object 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 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 readable or writable, we can easily define getter and setter methods. For example:

```ruby
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 read and write a particular instance variable. The original class or a subclass could change this decision later, without clients ever being aware. The setter could also be omitted to ensure an instance variable is not mutated from outside the object.

---

2 Whether we should is debatable: getters and setters can be a sign of poor OO design.
5 Dynamic class definitions

Because getter and setter methods are so common, there is special support for defining them. For example, to define getters for instance variables @x, @z, and @w, and setters for @y, @z, and @w, the class definition can simply include:

```ruby
attr_reader :x
attr_writer :y
attr_accessor :z, :w
```

Rather than being syntactic sugar, attr_reader, attr_writer, and attr_accessor are actually methods that automatically create the accessor methods and add them to the current class. So methods in Ruby can actually modify classes and create new methods!

Similarly, 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 class 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 you change or delete the + method on numbers, you would not expect many programs to keep working correctly. But 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.

6 Mutability in SML

Since we’ve introduced mutability in Ruby, we’ll finally admit that SML has support for mutation. Features for mutability are often called imperative features. Ruby, C, and Java are imperative languages. The antonym of “imperative” is “declarative”, of which functional programming is one example.

SML allows creation of mutable references, which are “containers” whose contents can be changed. Create a new reference with expression `ref e`. The initial contents of the reference are the result of evaluating `e`. To get the current contents of reference `r`, use expression `!r`. (Don’t confuse the exclamation point with negation as in Java or C). To change `r`’s contents, use expression `r := e`, which evaluates `e` to a value `v`, changes the contents of `r` to be `v`, and returns `v` as its value. The type of a reference that contains values of type `t` is `t ref`. Used in this way, `ref` is a type constructor, not a type itself. Note that keyword `ref` can actually be used in two distinct ways—as a type constructor and as a value constructor.