ECE155: Engineering Design with Embedded Systems Winter 2015

Lecture 2 — Introduction to Java

Jeff Zarnett, based on original by Patrick Lam

Java for C# programmers

The labs this semester will require you to write Java code for the Android platform, yet you learned C# in ECE 150. Fortunately, there are a lot of similarities between Java and C#, so you should have a smooth transition. As it says in the syllabus, if ECE 150 did not go well for you, please take some time at the beginning of the term to get caught up on Java. With that said, we're going to go over some basics, taken mostly from [Hub01].

The last page of these notes is intended as a quick lookup table that will hopefully serve you well in making the jump from C# to Java.

Java is an object-oriented programming language:

- Every piece of data is encapsulated in some object.
- Every executable statement is in some method.
- Every object is an instance of a class (or is an array).

Java Types

Java has eight primitive (basic) types. Every variable will be one of the primitive types or a reference to a Java object. A reference may be `null` or contain the address of an instance of an object. The eight basic types are:

1. `boolean` – can be either `true` or `false`
2. `char` – single unicode character (such as `'A'`)
3. `byte` – 8-bit signed integer
4. `short` – 16-bit signed integer
5. `int` – 32-bit signed integer
6. `long` – 64-bit signed integer
7. `float` – 32-bit signed floating point number
8. `double` – 64-bit signed floating point number

Note that in addition to their normal values, the floating point types have some extra weird values: `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, and `NaN` (Not a Number). These special values result from operations that go out of range or make no mathematical sense.

The most common types you are likely to use are `boolean`, `int`, and `double` (in the labs, `float` gets a fair amount of use).

You have no doubt noticed that `String` does not appear in the list. It's not a simple type; it's a reference to an array of characters. Hence you can declare a `String` as `null`, like this: `String s = null;`. Strings are also immutable: once created, they don't change. If you add to a String or replace characters in it, you get a new String back and not the old one. So if you did something like `string1.replaceAll('?', '.')`, (which replaces any question marks with periods) remember to assign that result somewhere (`string1 = string1.replaceAll('?', '.')`). The equality operator `==` can behave strangely on Strings. If you want to compare two Strings, the right way to do it is with the `equals` method: `if (string1.equals(string2) { ... }` .

Java also provides some wrapper classes for the primitive types that let you work with them as if they were regular objects. The wrapper class for `int`, for example, is `Integer` (mostly, just capitalize the first letter).

More complex objects exist, of course, and they are every other object you enounter in Java. Just like in C# you create them in classes, then when you want to create an instance of that class you use the `new` keyword. Example: `Integer example = new Integer(20);`. The `new` keyword invokes the class Constructor (a topic we'll come to later).

Java does not have `structs`; classes only. Frequently asked questions: there are no pointers or delegates in Java either.

Java uses *Garbage Collection*, so you don't have to worry about freeing up or de-allocating objects that are no longer needed. Java takes care of all of that for you, preventing the extremely common error of trying to use some object/memory that has been released.

Imperative Constructs All of the basic imperative constructs from ECE150 work the same way:

- Assignment: `x = y;`
- Math: `i = j + k`. (This includes the operators like `+=` etc.)
- Expressions: `z > 10 || ((c == 0) && (a == b))`
- If-Statements: `if(cond) { ... } else if (cond2) { ... } else { ... }`
- For Loops: `for (init; cond; expr2) { ... }`
- While Loops: `while (cond) { ... }`
- Do-While Loops: `do { ... } while (cond);`
- Switch-Case Statements: `switch (v) { case N: ...; break; case M: ...; break; default: ...; break; }`

However, C#'s `foreach (type t in c)` is instead `for (Type t : c)` in Java.

Methods work the same way also:

```
modifiers returnType methodName(param-list) {
  T1 t; returnType r;
  ...
  return r;
}
```

When you call a method on an object, you still use the `.` between the object and the method, such as `t.toString()`.

Caveat. Follow conventions. In C#, the method name convention is `UppercaseFirstLetter()`, while in Java, it is `lowercaseFirstLetter()`.

Example. A simple unit converter:

```
using System;

class FootConverter {
  static double ConvertFeetToMeters(double feet) {
    return feet / 3.28;
  }
  static void Main(string[] s) {
    Console.WriteLine("{0} ft is {1} m.", s[0],
      ConvertFeetToMeters(Convert.ToDouble(s[0])));
  }
}
```

```
class FootConverter {
  static double convertFeetToMeters(double feet) {
    return feet / 3.28;
  }
  public static void main(String[] s) {
    System.out.printf("%s ft is %.2f m.\n", s[0],
      convertFeetToMeters(Double.parseDouble(s[0])));
  }
}
```

Arrays & Collections: A Short Review

As in C#, Java has arrays. The simple array is created with the `[]` square brackets. Example: `int[] numbers = new int[10];`. You can have null elements in an array (say, of Strings) without this affecting the array length. An array is technically an object so you can assign it where a generic `Object` is expected. Multidimensional arrays are also allowed, but only for primitive types `int[][][] coordinates = new int[5][10][2];`. This is not a big restriction because you can just have an array of arrays. Unlike C# though, you can't specify a rectangular array.

This is great and all, but like the String the explicit array is of fixed size (and you could create a new, bigger one if you needed it and copy all the data to the bigger one... but that seems kind of wasteful...). It also isn't that nice to allocate an array of size 999 when we aren't sure how many we'll need. Wouldn't it be nice if we had a dynamic collection?

In Java, we do, and they're called, `Collection`s. The most common one you are likely to encounter or use is, unsurprisingly, the `List`. The list on its own doesn't do much or tell you much, but in modern Java, the type List takes a parameter in angle brackets to tell you what this is a list of. Example: `List<String>`. Note that you can't call `new List<String>()` because no constructor exists for just plain `List`. You will need to be a bit more specific about what kind of list you want to have, such as `ArrayList` (a very common one) or `LinkedList`.

There are 3 basic collections: the List (explained above), the Map (also in a later lecture), and the Set (it's like a list, but does not allow duplicates).

Your First Java Program

C#

```
using System;

class C {
  static void Main() {
    Console.WriteLine("Hello, world!");
  }
}
```

Java

```
class C {
  public static void main(String[] argv) {
    System.out.println("Hello, world!");
  }
}
```

Even though Java uses `main()` rather than `Main()`, this doesn't matter for Android programming, as you will observe in Lab 1.

Logging for Android Development

We'll finish with an Android development tip. `System.out.println()` is great for debugging console applications, but doesn't work on Android. Instead, use:

```
Log.d("tag", "i = "+i);
```

This writes out a debug (d) logging message, which appears e.g. in your Eclipse `LogCat` window. Instead of `d`, you can write `Log.d`, `.i`, `.v`, `.w` or `.wtf`. You can then filter out logging messages by level or tag, so that you only see the ones you're interested in.

References

[Hub01] John R. Hubbard. *Data Structures with Java.* McGraw Hill, 2001.

A short table to go over the differences between C# and Java at a glance.

C#	**Java**
Previous example:	
`UppercaseFirst()`	`lowercaseFirst()`
Main(...)	main(String[] argv)
`Console.WriteLine("{0}", s);`	`System.out.printf("%s", s);`
`Console.WriteLine("Text");`	`System.out.println("Text");`
`s = String.format("{0:.##}", f);`	`s = String.format("%.2f", f);`
`string`	`String`
`Convert.ToDouble`	`Double.parseDouble()`
Other fundamental language features:	
`const`	`final`
`bool`	`boolean`
both rectangular and jagged arrays	jagged arrays only
`array .Length`	`array .length`
`ref`, `out` (for method parameters)	no equivalent
pointers, `unsafe`, `fixed`	no equivalent
Object-oriented Features	
`class C : Parent, I`	`class A extends Parent implements I`
`struct`	`class`es only
`class C { public C(...) : base(...) {} }`	`class C { public C(...) { super(...); } }`
default visibility private	default visibility package
`x.GetType()`	`x.getClass()`
`is`	`instanceof`
`C cc = x as C`	`C cc = null;` `if (x instanceof C) cc = (C)x;`
`virtual`	(is mandatory default)
‘‘`new`’’ modifier	no equivalent
`override`	`@Override`
`IComparable`	`Comparable`
properties	manual getters and setters[1]
no equivalent	checked exceptions
using	import static (but don’t use it)
namespaces	packages