

Lecture 0: Introduction

Quote of the day:

“You know what’s needed before we get good software? Cars in this country got better when Japan showed us that cars could be built better. Someone will have to show the industry that software can be built better.”

John Murray, FDA’s software quality guru
quoted in Software Conspiracy, Mark Minasi, McGraw Hill, 2000

0.1 What 6170’s About

Course is actually three courses in one:

- crash course in object-oriented programming
- software design in the medium
- studio course on team construction of software

Emphasis is on design. Programming is included because it’s a prerequisite; the project is included because you only really learn an idea when you try and use it.

You will learn:

- how to design software: powerful abstraction mechanisms; patterns that have been found to work well in practice; how to represent designs so you can communicate them and critique them
- how to implement in Java
- how to get it right: dependable, flexible software.

Not hacking

- how to be a designer, not just a low-level coder
- you don’t need an MIT education to learn how to code in Java

How many of you have programmed in Java?

0.2 Admin & Policies

Course staff intros:

- Lecturers: Daniel Jackson and Paul Fitzpatrick
- TAs: you’ll meet in recitation tomorrow – go to any
- LAs: you’ll meet in clusters
- Hours: see website. Lecturers don’t have fixed office hours but happy to talk to students: just send email or drop by

Materials:

- lecture notes: usually published the day of the lecture
- Liskov and Guttag: good overview of all course material
- 'Gang of Four' design patterns book: recommended
- 'Effective Java' by Bloch: recommended
- Java tutorials: esp. java.sun.com/docs/books/tutorial/

Online forum:

- for all questions that might interest other students
- problem sets, Java, design, lecture material, etc

Course organization:

- First half of term: lectures, weekly problem sets, recitations
- Second half of term: team project, reviews, mentoring

Typical week:

- Monday 12.01am: problem set due, electronic handin
- Monday: new problem set out
- Monday, Tuesday, Wednesday 2:00 - 3:00: lecture
- Thursday 2:00-3:00: recitation; brainstorming note due at start

Your responsibilities:

- problem sets and final project
- read and respond to online forum
- attending lectures, recitations, grading meetings

Grading:

- Weekly meeting with your TA for 15 mins
- Opportunity for feedback and mentoring
- No written comments

Collaboration and IP policy:

- see general info
- in short: you can discuss, but *all* written work must be your own; includes spec, design, code, tests, explanations
- in team project, can collaborate on everything

Grading:

- we compute number: 70% problem sets, 30% project (same for all in team)
- we adjust subjectively (brainstorming notes, participation in recitation, online forum)
- *no late work will be accepted*

What's new? 6170 is continually evolving. We work hard to keep the material fresh, and to respond to comments from students. Some experiments we're trying this term:

- Grading meetings: to increase face contact with TAs, to focus TA work

- Brainstorming notes: to encourage early work and making good use of staff
- Online forum: so all students can benefit from staff responses, opportunity to discuss issues

And in an attempt to tie the lecture material more closely to problem sets, and to bridge the gap between theory and practice:

- Exercises instead of quizzes (part of problem set)
- More realistic examples in lecture

0.3 Advice

Course strategy

- don't get behind: pace is fast!
- start early, and talk to colleagues and staff
- think in advance: don't rush to code
- de-sign, not de-bug
- attend lectures
- enjoy yourself

Can't emphasize enough importance of starting early and thinking in advance. Of course I don't expect you to finish your problem sets the day they're handed out. But you'll save yourself a lot of time in the long run, and you'll get much better results, if you make *some* start on your work early. First, you'll have the benefit of elapsed time: you'll be mulling problems over subconsciously. Second, you'll know what additional resources you need, and you'll be able to get hold of them while it's easy and in good time. In particular, take advantage of the course staff – we're here to help! We've scheduled LA cluster hours and TA office hours with the handin times in mind, but you can expect more help if it isn't the night before the problem set is due when everyone else wants it...

0.4 Parting Shots

Reminders:

- Signup online by 8pm tonight, with user names of team mates
- Section assignments will be posted by midnight
- Tomorrow, go to recitation room for your section as given by general info online
- Get started on learning Java now!
- Your 6170 directory will be set up on your Athena account by midnight tonight
- PS0 is due Monday

© 2004, Daniel Jackson

Lecture 1: Object Semantics

1.1 Context

What you'll learn: the basic semantics of Java: the heap model, objects and references, variables, assignments, mutability, etc.

Why you should learn this: you can't write Java without understand it (though some try); most notions common to all OO programming languages; first cut at some fundamental design notions (mutability, sharing, abstraction)

What I assume you already know: basic imperative programming (the idea of sequencing)

1.2 Problem

Each topic we discuss in lecture will be motivated by a particular realistic (although often small) problem. Today's problem is how to print error messages in an application such as a compiler that reports errors associated with line numbers in some input file. We want to write a procedure `reportError` that takes a string representing a reason, and an integer representing a line number, and prints them in a message to the console. For example, if the reason is "no such command", and the line number is 23, it should print:

```
no such command, line 23
```

1.3 Ingredients

I'll give you some ingredients and you figure it out.

1. A procedure is called a 'static method' in Java. You declare it like this:

```
class Reporter {  
    static void reportError (String reason, int line) {...}
```

2. A static method is called like this, giving class and method name:

```
Reporter.reportError ("no such command", 23);
```

3. To print a string `s` to the console, write

```
System.out.println (s);
```

4. For two strings `s` and `t`, you can write an expression `s.concat (t)`, which is the string representation of their concatenation.

5. The expression `String.valueOf (i)` gives the string representation of the integer `i`;
6. The expression `"hello"` evaluates to a string object; it's called a 'literal'.

1.4 A First Solution

Here's one way to do it using these ingredients:

```
class Reporter {
    static void reportError (String reason, int line) {
        String onString = ", line ";
        String lineString = String.valueOf (line);
        String msg = reason.concat (onString).concat(lineString);
        System.out.println (msg);
    }
}
```

What's going on here exactly?

Look at this statement:

```
String onString = ", line ";
```

The statement does double duty. First, it *declares* the variable `onString` as a `String`; this is information used by the compiler to check the program before you run it. Second, it actually causes some computation. The right-hand side is an expression that evaluates to a `String` object. The `=` sign, read "gets" not "equals", *assigns* the object to the variable. (Why not "equals"? We'll use that word for the `==` test.)

Let's be a bit more careful. The assignment actually binds the variable to a value, and that value is a *reference* to the string object. This will be important later.

The next line is similar; the expression `String.valueOf (line)` creates an object, and its reference gets assigned to `lineString`.

Now let's look at the more complicated statement:

```
String msg = (reason.concat (onString)).concat(lineString);
```

I've added some parens to show you how it's parsed. The expression `reason.concat (onString)` evaluates to a string object, which is obtained by applying the method `concat` to the arguments `reason` and `onString`.

How's this different from the kind of function call you've seen in Scheme, which might have been written like this?

```
(concat reason onString)
```

The difference is that `concat` is a method, not a function. There can be lots of methods called `concat` in the program. The right one is found by *dynamic dispatch*: the runtime system actually takes the object referred to by `reason`, looks up its *class*, which is `String`, and then calls the method called `concat` from the `String` class, passing `onString` as an argument. More on this in recitation tomorrow.

Note there's a fundamental asymmetry between arguments. The first argument is called the *receiver*.

[If you were at the lecture, you saw the diagrams I drew to explain the execution of this code; if you weren't, try and draw some yourself to check your understanding.]

How are arguments passed in Java? By value, but the value here is the object *reference*. So the variable `reason` inside the method gets bound to the object passed in. What about `line`? It's an integer, and integers aren't actually objects at all. Yes, this is awkward, and it makes a mess of the language in many ways. Other OO languages, such as CLU (developed by Barbara Liskov, author of your text) do away with this distinction and make all variables hold references to objects.

1.5 Immutability

What if I wrote this instead?

```
class Reporter {
    static void reportError (String reason, int line) {
        String onString = ", line ";
        String lineString = String.valueOf (line);
        String msg = reason;
        msg.concat (onString);
        msg.concat(lineString);
        System.out.println (msg);
    }
}
```

Prints "no such command". Why? What happened to the line number?

Nothing strange: the `concat` method produces a new object, but it gets thrown away. How do I know that? By checking the specification of `concat`. But I actually know that this cannot have worked without even looking at the *method* specification, because the *class* specification tells me that `String` is *immutable*. That means that `String` objects never change, so it would not be possible to write a `concat` that changes its argument.

Could it do it by assignment instead? No, because inside the method, there's no way to change the binding of the argument variable. You can't even reference it; it's out of *scope*.

1.6 Null References

What happens if you run this?

```
String reason;  
Reporter.reportError (reason, 23);
```

It crashes with a `NullPointerException`. The problem is that `reason` is *uninitialized*. The variable has the value `null`. Remember that variables' values are *references*. So `null` is a reference, not an object. This confuses a lot of people; they think `null` is a special object. When the runtime system looks up the class of the object bound to `reason` as it evaluates the expression `reason.concat (...)`, it fails, because there's no object there.

This is another example of the basic asymmetry of method call. The expression

```
x.m (y)
```

will always fail if `x` is `null`, but may not fail if `y` is `null` (as we'll see in a minute).

I lied to you. This error of the uninitialized variable is so egregious that the compiler won't even accept the program; you can't run it. You can actually write this:

```
String reason = null;  
Reporter.reportError (reason, 23);
```

and the compiler won't complain, because `reason` is no longer uninitialized. You might wonder why you should ever want to initialize a variable to `null`; it does turn out to be useful, but it's something you should be very careful with, and it's a sign of bad programming to do it often.

Here's a puzzle for you: find a way to invoke the method with the `reason` argument bound to `null`, without mentioning `null` explicitly.

1.7 Mutable Objects

There is actually a way to write the program with the accumulating concatenations:

```
class Reporter {  
    static void reportError (String reason, int line) {  
        String onString = ", line ";  
        String lineString = String.valueOf (line);  
        StringBuffer msg = new StringBuffer ();  
        msg.append (reason);  
        msg.append (onString);  
        msg.append (lineString);  
        System.out.println (msg);  
    }  
}
```

```
}
```

How does this work? This time `msg` is bound to a `StringBuffer`, not a `String`. A `StringBuffer` object is mutable, so there are methods that change objects of that class. The call to `new` creates an empty buffer – that’s special syntax for calling a method known as the *constructor*. The `append` method *mutates* the buffer object.

It actually returns something too: a reference to the same buffer object. This allows us to write:

```
class Reporter {
    static void reportError (String reason, int line) {
        String onString = ", line ";
        String lineString = String.valueOf (line);
        StringBuffer msg = new StringBuffer ().append (reason).append (onString).append (lineString);
        System.out.println (msg);
    }
}
```

Can you draw a diagram like the one I drew in lecture to explain how this procedure executes?

What happens if you pass `null` to this one for the `reason` argument?

It will actually print:

```
null, line 23
```

Why? Because `append` will accept a `null` reference as an argument. Its code says something like this:

```
StringBuffer append (String s) {
    if (s == null) s = "null";
    ...
}
```

This is the first time you’ve seen the equality test. The expression `s == null` evaluates to `TRUE` when the variable `s` holds a `null` reference.

1.8 Aliasing

Suppose you haven’t quite understood mutation, and you wrote this:

```
class Reporter {
    static void reportError (String reason, int line) {
        String onString = ", line ";
```

```

String lineString = String.valueOf (line);
StringBuffer msg1 = new StringBuffer ().append (reason);
StringBuffer msg2 = msg1.append (onString). append (lineString);
System.out.println (msg2);
}
}

```

Will it work? Yes, because there's no harm done in creating another variable (msg2) pointing to the same buffer object. It's just a bit inelegant.

But what if you made a mistake and wrote msg1 instead of msg2 in the last line? Will that still work? Again, yes, because the variables hold a reference to the same object. That would be just even more inelegant.

Finally, here's a real mess. What does it do?

```

class Reporter {
    static void reportError (String reason, int line) {
        String onString = ", line ";
        String lineString = String.valueOf (line);
        StringBuffer msg1 = new StringBuffer ().append (reason);
        StringBuffer msg2 = msg1;
        msg1.append (onString). append (lineString);
        System.out.println (msg2);
    }
}

```

Again, it still works: msg1 and msg2 reference the same object. This phenomenon, in which there are two names for an object at a single time is called *aliasing*. It's very useful, because it allows you to introduce a short name (ie, a variable) for something that until then has only a long name (eg, a complicated expression involving lots of field dereferences).

But it's confusing at first, and makes program analysis much harder. You might have expected that if a statement mentions variable x but not variable y, then the value of y cannot be affected by the statement. Is this true? Yes, if we're taking precisely, because the values of the variables are object *references*. But if you take 'value' to mean what people often mean colloquially – namely the contents of the object referred to, then it isn't true.

1.9 How You'd Really Write It

In practice, you'd write this:

```

class Reporter {

```

```

static void reportError (String reason, int line) {
    String msg = reason + ", line " + line;
    System.out.println (msg);
}
}

```

using some special Java syntax: the + operator. The expression

```
a + b + c
```

is short for

```
(new StringBuffer ().append (a). append (b). append (c)).toString ()
```

One of the nice features of append is that you can actually pass it all kinds of arguments, including primitive values; in our example, the second call passes an integer. How does this work? Roughly speaking, there's a separate version of the append method for each kind of argument. This isn't dynamic dispatch; the compiler figures this out at compile-time. This phenomenon is called *overloading*; look it up in a good Java text for an explanation, and check that you understand how it differs from dynamic dispatch.

1.10 A Design Question

Suppose that sometimes you want to report an error without a reason. What should you do? Here are some of your options:

- Pass in null as the reason;
- Pass in "" as the reason;
- Pass in "error";
- Pass in null or "" and have reportError convert it to "error".

None of these is very attractive. A better solution, which is usually adopted, is simply to make another version of the method. You can even exploit overloading and give it the same name. Then you can write

```

Reporter.error (23);
Reporter.error ("no such command", 24);

```

and the compiler will figure out that you're calling two different methods.

1.11 Summary

We looked at a small problem. I'm convinced that almost all real software is fractal, in the sense that when you look closely at something small, you see at least some of the complications of bigger things. This tiny example brought us in contact with some fundamental ideas – references, objects, assignment, mutability, nulls, dynamic dispatch;

it introduced us to some features of the Java language – checking for uninitialized variables, overloading; and we saw two built classes, `String` and `StringBuffer`, and some of their methods.

Later we'll come back to error reporting. We'll see other, deeper software design issues lurking there, in particular in the question of what to do if you want to print somewhere else (and not the console).

© 2004, Daniel Jackson