Odds and Ends

In this lecture, I want to touch on a number of topics that are intrinsically useful, but that we do not have time to cover. These topics are important because chances are you will actually encounter them in upcoming years.

Assertions

Let's think about testing some. We have advocated testing in this course, and in particular blackbox unit testing.

However, there are times when whitebox testing, that is, testing where the tests or checks are put *inside* the class to be tested, is useful, and more natural.

Java does provide some help in doing that.

An *assertion* takes the following form:

`assert` *BooleanExpression*;

or

`assert` *BooleanExpression* : *ValueExpression*;

When Java encounters an `assert` during execution, it does one of two things, depending on whether or not assertion checking is turned on or off. (It is off by default.)

- If assertion checking is turned on, it will evaluate the *BooleanExpression* in the assertion; if it evaluates to true, then execution proceeds to whatever statement follows the assertion; if it evaluates to false, then an exception `AssertError` is thrown (with a value given by *ValueExpression*, if one is provided).
- If assertion checking is turned off, then the assertion is skipped altogether.

Thus, assertions provide a way to put in "sanity checks" within your own code, that you can enable by running the code with assertion checking turned on, with the property that if you run the code with assertion checking turned off, it behaves just like you never had put in any assertions in the first place. (There is no runtime penalty, either.)

To execute a Java program with assertion checking turned on, use:

```
java -ea Program
```

(assuming `Program.class` is the compiled class containing the `main` method).

I will refer you to the course web page for a particularly good tutorial about assertions, from the Java documentation. It describes cases in which assertions are useful.

Generally speaking, assertions are meant to check invariants that *must* be true in order for the code to work correctly. This means, in particular, that whenever possible, you should have assertions on when you ship your code out. (There are ways to make sure that Java or whatever language you use executes your code with assertions turned on. Please refer to the documentation.) Many programmers do not enable assertions on production code, for fear generally of the run-time cost of checking all those assertions. Unless your assertions are very heavy, however, I would recomment you keep assertions on. The benefit of having this extra layer of checks for the important invariants your code usually outweighs the minor inconvenience of a slightly slower execution. (And, let's face it, if you are programming in Java, you are alreadu more or less admitting that executing programs as fast as possible is not your priority.)

JUnit

JUnit is a product (see `http://junit.org/`) to automate the unit blackbox testing of software. The basic idea is to write tests in your code as special classes, and the framework will automatically discover those tests and run them. Here is an example. (This is not the latest version of JUnit, but it illustrates the point anyways.)

```
public class HelloWorld extends TestCase {
  public void testMultiplication()
  {
    // Testing if 3*2=6:
    assertEquals ("Multiplication", 6, 3*2);
  }
}
```

This describes a class that implements tests (because it extends `TestCase`, a JUnit class), and that test performs a simple check. How does JUnit know what tests to run? It uses *reflection* (see below) to discover at runtime all the classes that extend `TestCase`, and all the methods that those classes implement, and run all those test methods.

More recent versions of JUnit do not rely on reflection, but on a new feature of Java called annotations. I will let you discover what those are for yourself.

Reflection

As I said, JUnit used to rely on reflection. Reflection is a strange beast in the Java world. For most guarantees that Java provides, reflection offers a backdoor that invalidates those guarantees. Because of that, it is a dangerous toy. It is also an inefficient toy, in that using reflection can slow down programs. However, there are things that can only be done cleanly using reflection.

Very roughly speaking, reflection gives you access to (a representation of) the source code of your program from within your program.

Consider the following example. Suppose we have a class `Foo` with a method `bar` that takes no arguments. Creating an object of type `Foo` and calling its `bar` method is simple enough:

```
Foo foo = new Foo ();
foo.bar();
```

However, suppose that `Foo` has several different methods, all taking no arguments, and the programmer does not know a priori which method will be invoked. How can that be? Well, as an example, imagine that the program simply asks the user for which method to invoke on object `foo`.[1]

Assume we have a class `Input` with a static method `getInput`. We might want to try something like this:

```
Foo foo = new Foo ();
String in = Input.getInput();
foo.in();
```

but that clearly doesn't work, as it tries to invoke method `in` on object `foo`, not the method named after the string contained in variable `in`. There's no easy way around that.

Enter reflection. Here's how to reproduce the above simple invocation of `bar` with reflection. The reflection classes are in package `java.lang.reflect`, so you will need to import its content. First, get a representation of the source code for class `Foo`:

```
Class<Foo> cls = Class.forName("Foo");
```

This returns an object of type `Class` which represents the source code of class `Foo`, given as a string argument. Next, we create a new instance of the class:

[1] We can come up with other examples of not knowing in advance which method to invoke for Homework 6. Suppose we wanted to support a notion of "game extension", or "game plugin": a piece of code that anyone can write that you load dynamically into the game to extend it, without having to recompile, perhaps via a verb `LOAD` in the game itself. Java has some facilities to add a class to an executing program (via the so-called class loader). But what method do you invoke on the loaded class? Your code a priori does not know about that class, so you cannot tell in advance what to invoke. So you may need, as an argument to the `LOAD` verb, to get the name of the method to invoke to launch the plugin, for instance.

```
Object foo = cls.newInstance();
```

This is essentially equivalent to performing a `new`, except that it does it indirectly, through the representation of the class we got our hands on earlier. Now that we have an instance, let's invoke method `bar` on it. First, we need to get a representation of the method to invoke by querying the source code of the class.

```
Method method = cls.getMethod("bar",null);
```

This gives us back a representation of the method `bar` in class `Foo`. We can now invoke that method on object `foo`:

```
method.invoke(foo,null);
```

There. Invoking `foo.bar()` ,the long way around.

What's interesting is that both the class name and the method name to invoke are just strings. Meaning that we can actually use anything that evaluates to a string in their place. Thus, we can solve our problem above:

```
Class cls = Class.forName("Foo");
Object foo = cls.newInstance();
String in = Input.getInput();
Method method = cls.getMethod(in,null);
method.invoke(foo,null);
```

Voilà. Of course, if the user inputs a method name that is actually not implemented by class `Foo`, the call to `cls.getMethod()` will fail with an exception.

Final Review

Here are the main topics we have seen in class. I expect you to be able to answer basic questions about each, including definitions, relationship between various notions. I also expect you to be able to recognize any of the notions below, and to use them in code fragments that I may ask you to write.

- ADT and specification: signatures, algebraic specifications, reasoning with algebraic specifications, implementing specifications, public versus private members of a class, static methods for creators, dynamic methods for operations (understand the difference), code against an ADT to allow substituting implementations, defining equality for ADTs, implicit specifications for equality.

- Errors and testing: classification of errors, exceptions, use of the type system to guarantee some classes of errors do not occur, exceptions, black-box versus white-box testing, unit testing versus integration testing, tests generation, testing via a specification.
- Subclassing: definition of subclass, abstract classes, concrete classes, recipe to derive an implementation from an ADT specification, nested classes, tree subclassing hierarchies, non-tree subclassing hierarchies, interfaces, casting, dynamic dispatch.
- Generics: generic interfaces, generic classes, generic methods.
- Inheritance: definition of inheritance, differences with subclassing, inheritance and subclassing in Java, multiple inheritance, protected members of a class.
- Design patterns: Iterator design pattern, distinction between functional iterators and Java iterators, Adapter design pattern, adapters for functional iterators, adapters for Java iterators, Visitor design pattern, Observer design pattern, higher-order programming in object-oriented languages, lazy structures.
- Mutation: memory model of Java, static fields, sharing, shallow copy, deep copy.