23 Odds and Ends

In this lecture, I want to touch on a few topics that we did not have time to cover.

23.1 Factory Methods

Through the course, most of the code that I wrote (and that I have had you write) used a static method to invoke the constructor of a class, instead of calling the constructor directly.

This is known in the literature as using a *factory method*. There are several reasons why using factory methods is a good idea, despite the fact that they require you to write more code.

(1) You can have different factory methods corresponding to different ways of creating an instance of a class, with reasonably informative names.

As an example, consider a `Point` class, without factory methods

```
public class Point {

    private double xpos;
    private double ypos;

    private Point (double x, double y) {
        xpos = x;
        ypos = y;
    }

    public double xPos () {
      return this.xpos;
    }

    public double yPos () {
        return this.ypos;
    }

    public Point move (double x, double y) {
```

```
        return new Point(this.xPos()+x, this.yPos()+y);
    }

    public String toString () {
        return "(" + xpos + "," + ypos + ")";
    }
}
```

Of course, as such, this is useless, because there is no way to create points, since the constructor is private. Now, there are two reasonable factory methods (creators) we can define: one that takes a pair of doubles as *cartesian coordinates* of the point we want to create, and one that takes a pair of doubles as *polar coordinates* of the point we want to create:

```
    public static Point createFromCartesian (double x, double y) {
        return new Point(x,y);
    }

    public static Point createFromPolar (double rho, double theta) {
        return new Point(rho*Math.sin(theta), rho*Math.cos(theta));
    }
```

The above is hard to do with only Java constructors — because note that even if we can overload constructors in Java, overloading is resolved using the types of the arguments, and the types for the above two ways of constructing points take the same types as arguments.

(2) Because factory methods abstract away from the whole constructors business, it makes it easier to replace code by equivalent implementations that use possibly a different class hierarchiy.

For example, recall the `List` class we defined earlier, using the design pattern to derive an implementation from a specification. That implementation uses two concrete subclasses of an abstract `List` class, that we can call `EmptyList` and `ConsList`. Using factory methods means that we can `List.empty()` and `List.cons()` to create lists without needing to know about the subclasses. It also means we can write a different implementation of `List` that does not use, say, subclasses, but delegation — such as:

```
import java.util.*;

public class List {

    private LinkedList<Integer> del;
```

```
    private List (LinkedList<Integer> d) {
        del = d;
    }

    public static List empty () {
        return new List(new LinkedList<Integer>());
    }

    public static List cons (int a, List l) {
        LinkedList<Integer> copy = new LinkedList<Integer>(l.del);
        copy.addFirst(a);
        return new List(copy);
    }

    public boolean isEmpty () {
        return del.isEmpty();
    }

    public int first () {
        return del.getFirst();
    }

    public List rest () {
        LinkedList<Integer> copy = new LinkedList<Integer>(del);
        copy.removeFirst();
        return new List(copy);
    }

    public String toString() {
        return del.toString();
    }
}
```

(3) Factory methods are also interesting when we do not know until runtime which of two implementations of a class we want. For instance, suppose we have two implementations of `Foo` with the same interface, call them `Foo1` and `Foo2`, with difference in terms of memory usage and efficiency. Then we may want to use a factory method that chooses between `Foo1` and `Foo2` based on various runtime conditions:

```
public abstract class Foo {

  public static Foo create () {
    if ( /* some conditions that make Foo1 a good choice */ ) {
      return new Foo1();
    } else {
      return new Foo2();
    }
  }

  public abstract void bar ();

}


public class Foo1 extends Foo {

  /* Code for the first implementation */

}


public class Foo2 extends Foo {

  /* Code for the second implementation */

}
```

As a concrete example, writing code that works on multiple platforms (e.g., Windows and Mac OS X) sometimes requires system-specific initialization and libraries, and factory methods can be used to chose the appropriate code to use during execution, when the system on which the code is being run can be checked, and the appropriate instance of a class created.

23.2 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:

```
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);
```

[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.

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, design pattern to derive an implementation from an ADT specification, nested classes, tree subclassing hierarchies, non-tree subclassing hierarchies, interfaces, casting, dynamic dispatch.
- Polymorphism (aka generics): polymorphic interfaces, polymorphic classes, polymorphic methods.
- Inheritance: definition of inheritance, differences with subclassing, inheritance and subclassing in Java, multiple inheritance, inheritance versus delegation, protected members of a class.
- Design patterns: Iterator design pattern, distinction between functional iterators and Java iterators, Adapter design pattern, adapters for functional iterators and Java iterators, Map and Reduce design patterns, Observer design pattern, laziness, MVC design pattern.
- Mutation: memory model of Java, static fields, sharing, shallow copy, deep copy.