ECE155: Engineering Design with Embedded Systems Winter 2015

Lecture 3 — Java II

Jeff Zarnett & Mahesh Tripunitara

Java, Continued

Now that you've got a short introduction to Java, it's time to move on to some more complex object-oriented bits of the language.

Classes, Superclasses, Subclasses

In Java, a `class` is an implementation of a data type, and we use them to create objects (by creating new instances, or "instantiating", them). The class definition specified the kind of data that its objects maintain (data members, or fields) and the operations that the object performs (method members). A field is a variable that holds data; a method performs operations. Classes can also have other classes inside of them (called "inner class"), but we'll get to them later [Hub01].

Classes can be related to one another, of course: Java, like C#, has *inheritance*. We say that a class B inherits from A if B `extends` A by using (at least some of) its methods and fields in addition to any of its own. We say that B is a "subclass" or "child" of A; and that A is a "superclass" or "parent" of B. In this class we'll use the subclass/superclass terminology.

In Java, all classes have a single inheritance hierarchy: each class has exactly one superclass. (Other languages like C++ allow multiple inheritance, where a class may have more than one superclass... which can be a very big headache.) The keyword in Java for declaring a class as a subclass of another is `extends`. For example: `public class Book extends Document` declares the class `Book`, with superclass `Document`; we can also say that `Book` is a subclass of the class `Document`. Every object in Java eventually descends from `Object`; if you do not declare a specific superclass with the `extends` keyword, the superclass will be `Object`.

An obvious advantage is avoiding repetition: we don't have to re-implement code that is common; it is simply re-used. Similarly, we don't have to copy-and-paste code, which is a potential source of error.

Another aspect of inheritance that makes it useful is *polymorphism* – the ability of a class to look like another. If you have a method that expects a `Rectangle`, you can give in a `Rectangle` or `Square` and it will work if `Square extends Rectangle`.

On (or inside of) a subclass we can call a method of the parent class. Given the `Book` example from earlier, we know that book is a subclass of `Document` and somewhere in the hierarchy `Document` descends from `Object`, either directly or indirectly. Given that `toString()` is defined on `Object` we can call the `toString()` method on a `Book`.

Also from within a subclass, we can use an implementation of a method from a parent class with the keyword `super`. The most common situation where this happens is in the constructor. When writing the constructor for `Book` it might make sense to do all the actions of the `Document` constructor, plus some `Book`-specific actions. In that case, the first line of the `Book` constructor is a call to `super()` which executes the constructor of `Document` first. In the constructor of `Book` you can overwrite things that were set in the constructor of `Document`; the superclass constructor runs first and then the rest of the statements. See the example below.

```
public class Book extends Document {

  public Book() {
    super();
    setPageWidthInCm(20);
    setPageHeightInCm(30);
  }

  ...
}
public class Document implements Readable

  public Document() {
    setRead(false);
    setTitle(``Untitled'');
    setAuthor(Environment.getUserID());
  }

  ...
}
```

The Keyword `static`

When you see the keyword `static`, it is a modifier that means there is only one copy of the thing. By now you have probably seen the `main()` method in Java is declared as `static` and that you have to declare variables as `static` as well if you want to reference them in `main`. Let us take a moment to examine the semantics of this keyword in Java.

If `static` is applied to a variable, it means that variable is common to all instances of a class. Assume you have an integer `i` in class A and there are two instances of the class called A_1 and A_2. The value of `i` starts at 0; if you increment that variable in A_1 it will be 1; it's shared between all instances so if A_2 increments the variable, `i` will be 2. Because *i* is static, it will be 2 whether you access it in A_1, A_2, or elsewhere using `A.getI()`. A common variable could be used, for example, to keep a running total of the number of times a method has been called.

If `static` is applied to a method, it means the method is shared between all instances of the class. A static method cannot access any of the instance variables or methods of the class; only methods and variables that share the static modifier.

Note that `static` cannot be applied to a class. There are a number of useful Java classes which offer their functionality only in `static` methods, such as the `Math` class. Using `Math` you can perform such operations as square root which are accessed in a static way (`Math.squareRoot(...)`. It doesn't make logical sense to instantiate `new Math()` to perform such an operation.

Interfaces

An `interface` is a way of specifying in Java what is effectively a contract. The interface specifies some number of methods which any class that wants to implement this interface must have. A simple example:

```
public interface Drawable {
  void draw();
}
```

Any class which implements `Drawable` (which is declared by putting `implements Drawable` in the declaration of the class) must contain an implementation of the method `draw()`.

Methods that are declared in an interface are always public, so it is not necessary to put that modifier before them (although you can; it does no harm). Other modifiers are not allowed.

Interfaces can extend other interfaces, but may never contain implementation. The closest we get is that an interface may declare some constants (always static, never changeable). Accordingly, an interface can never be instantiated. A class can implement as many interfaces as desired (whereas it can have only one superclass). This is the Java sort-of-equivalent of multiple inheritance, but with slightly less headache.

Why are interfaces useful? They allow us to interact with an object without knowing or caring about what it is underneath. The very common interface `Comparable` allows the Java runtime environment to sort objects of some type. If you want to sort a list of something using Java's built-in sorting routines, the thing to sort must implement `Comparable` (otherwise Java has no idea how to order them). To implement `Comparable`, only one method (`CompareTo`) is required. If the objects we are looking at represent soccer (football) players and we want to sort them by their uniform numbers ascending, we implement that logic in `CompareTo`. Then we can use something like `Collections.sort()` without the Java runtime environment having to know anything in advance about the soccer players.

Abstract Classes

Halfway between an interface and a regular class is the abstract class. The Java developer guidelines will tell you that an abstract class is declared using the `abstract` keyword and may (but does not have to) contain abstract methods. However, any class that contains methods marked `abstract` must also be marked as `abstract`. An abstract class cannot be instantiated, but may be subclassed. Unlike an interface, an abstract class can have as much or as little implementation as desired.

To declare an abstract method, we put the method signature as we would in an interface, but with the keyword `abstract` attached: `public abstract void draw(int parameter);`.

Abstract classes may have static fields and methods. You can make use of one of these in the standard way, e.g. `AbstractClass.method()`.

Abstract Classes vs. Interfaces

Given that there are a lot of similarities between abstract classes and interfaces, when should each of these be used? According to the developer guidelines [Ora14]:

Consider using abstract classes if any of these statements apply to your situation:

- You want to share code among several closely related classes.
- You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
- You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

Consider using interfaces if any of these statements apply to your situation:

- You expect that unrelated classes would implement your interface. For example, the interfaces `Comparable` and `Cloneable` are implemented by many unrelated classes.

- You want to specify the behaviour of a particular data type, but not concerned about who implements its behaviour.
- You want to take advantage of multiple inheritance of type (i.e. implement multiple interfaces on one class)

Of course, there is always the third option of "why not both?". Another common strategy is to have an interface which is partly implemented by an abstract class. The abstract class may implement some or all of the methods of the interface. The concrete class that extends the abstract class is responsible for implementing any of the methods of the interface that its superclass does not. Imagine interface K has four methods, and there is an abstract class L declared as `implements K`. If L implements three of the four methods of K, when a subclass P of L is declared, then P must implement that last remaining method that L did not (or else be abstract itself).

Visibility Modifiers

There are four options for visibility (accessibility) of something (a variable, method, class, etc) declared outside of a function.

- `public`: The field/method/class may be accessed from anywhere, by any piece of code.
- `private`: The field/method/class can only be accessed from within the class where it is declared. A top level class cannot be declared as private (makes no sense since nobody could use it / call it).
- `protected`: The field/method can only be accessed from within the class where it is declared and its subclasses, all the way down the hierarchy.
- *(No Modifier)* If no modifier is specified, the result is "package private": the field/method/class may be accessed from within the same Java package, so it's sort of public. Don't use this.

Could one in theory just declare everything `public` and not care about this? Probably, but it is very poor programming practice. In Java, convention upholds the principles of encapsulation and information hiding: the internal state of objects should not be visible to the world. That might seem like an academic distinction, but in the real world anything that is `public` will be accessed by other programmers and they may come to depend on a particular implementation detail (a problem that Microsoft has in abundance). Consider carefully if a method should be accessible from outside the given class; if the answer is no, then `private` is the right answer for the modifier, or `protected` if it may be used in a subclass.

With Finality

There's another keyword we haven't mentioned yet: `final`. This can be applied to three things, with slightly different meanings:

- **Field**: The field cannot be modified (it's a constant).
- **Method**: The method cannot be overridden in a subclass.
- **Class**: No subclass of this class is permitted.

We cannot apply `final` to an interface, because we will obviously have to implement it somehow. Similarly, we cannot make an abstract class final, because it is meant to be subclassed.

Example: Shapes

Suppose we define a *Shape* as something that has an area and a perimeter. And a *Quadrilateral* as something with four sides. Consider the following structure for the interface `Shape`, abstract class `Quadrilateral`, and classes `Rectangle` and `Square`.

Note that the field `side` in `Quadrilateral`, which is an integer array, is declared to be `protected`, which means that it is visible to subclasses.

Complete the code where indicated with "`/* Your code here */`." There are three places where it appears — the constructor for `Quadrilateral`, the method perimeter() in `Quadrilateral`, and the method area() in `Rectangle`.

```
public interface Shape {
    public int area();
    public int perimeter();
}
```

```
public abstract class Quadrilateral
                    implements Shape {

    protected int[] side;

    public Quadrilateral(int side1, int side2,
                         int side3, int side4) {

        /* Your code here */

    }

    public int perimeter() {

        /* Your code here */

    }
}
```

```
public class Rectangle
            extends Quadrilateral {

    public Rectangle(int side1,
                     int side2) {
        super(side1, side2,
              side1, side2);
    }

    @Override
    public int area() {

        /* Your code here */

    }
}
```

```
public class Square extends Rectangle {
    public Square(int s) {
        super(s, s);
    }

    @Override
    public int area() {
        return super.area();
    }

    @Override
    public int perimeter() {
        return super.perimeter();
    }
}
```

Solution

```
public abstract class Quadrilateral
                    implements Shape {

    protected int[] side;

    public Quadrilateral(int side1, int side2,
                         int side3, int side4) {
        side = new int[4];
        side[0] = side1; side[1] = side2;
        side[2] = side3; side[3] = side4;
    }

    public int perimeter() {
        int result = 0;

        for(int i = 0; i < 4; i++) {
            result += side[i];
        }

        return result;
    }
}
```

```
public class Rectangle
             extends Quadrilateral {
    ...

    @Override
    public int area() {
        return side[0] * side[1];
    }
}
```

Summing It All Up

This table from [Hub01] sums up the different modifiers in Java.

Modifier	Interface	Class	Nested Class	Field	Method
`public`	Accessible from any class.				
`private`	Accessible only from this class.				
`protected`	Accessible only from this class and its subclasses.				
(No modifier)	Accessible from any class within the same package. Don't use this.				
`abstract`	N/A	Contains at least one `abstract` method; cannot be instantiated.		N/A	Its implementation is not defined; only signature & return type declared.
`final`	N/A	Cannot be subclassed.		Its value cannot be changed.	It cannot be overridden by a subclass.
`static`	N/A	N/A	Not an inner class.	Exactly one instance exists for all objects of the class.	Exactly one instance exists for all objects of the class.

References

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

[Ora14] Oracle. Abstract methods and classes. `http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html`, 1995-2014. Online; accessed 21-April-2014.