Visibility Modifiers

You should be familiar with defining classes to be `public` and defining methods and fields to be `public` or `private`. You learned, for example, that fields in a class usually are defined to be `private`, and so someone using the class has to access or modify the information in the fields using getter and setter methods. Later today I will discuss why this is good practice. [Note that in the assignments we often made fields and methods public, some of which normally should be private. This was done to simplify the tester code so that we could have access to these fields and methods.]

The question of `public` vs. `private` gets a bit more complicated when you consider inheritance relationships. Packages come into this too, because they can determine whether classes can extend each other, and whether they can reference and invoke methods from each other. Today I'll go over what visibility modifiers do, and then introduce a few other important modifiers that you may have seen but have not yet learned about.

This lecture has many examples, which I do not reproduce here. You will need to go through the slides, in addition to reading these lecture notes.

Packages

To understand visibility modifiers, we first need to understand what packages are. A *package* is a particular set of classes. Go to the Java API

`http://docs.oracle.com/javase/7/docs/api/`.

and notice that in the upper left corner of that page is a listing of dozens of packages. A few of them that you are familiar with are:

- `java.lang` - familar classes such as `Object, String, Math, ...`
- `java.util` - has many of the data structure classes that we use in this course, such as `LinkedList`, `ArrayList`, `HashMap`

The full name of a class is the name of the package following by the name of the class, for example `java.lang.Object` or `java.util.LinkedList`. You can have packages within packages e.g. `java.lang` is a package within the `java` package.

As you know from your assignments, you can make your own packages. For me, the classes `Beagle` and `Dog` belong to package `comp250.lectures`, and so the full name of these classes would be `comp250.lectures.Beagle` and `comp250.lectures.Dog`. My Assignment 3 package is `comp250.assignments2017.a3`

Packages correspond to file directories, and as such packages have a tree structure. The classes within a package are leaves. Packages typically are internal nodes of a directory tree (the only exception being if you have an empty package in which case it would be a leaf).

The `package` directory in the first line of the `.java` file specifies which directory the file should be in. When the `.java` file gets compiled into a `.class` file, the `package` definition specifies the directory where this `.class` file goes. Usually the `.java` file goes in a `src/` directory and `.class` files go in a `bin/` directory. Your computer finds the packages using a CLASSPATH environment variable `https://en.wikipedia.org/wiki/Classpath_(Java)`.

 ©Michael Langer

Visibility modifiers for classes

If you want a class `A` to be visible to all other classes, regardless of which package the other classes are in, declare `public class A {...}` . If you want a class `A` to be visible only to classes in the same package, then do not put a visibility modifier in the declaration of the class, i.e. just use `class A {...}` That is, the default is that the class is visible to other classes in teh same package. Note that there is no reserved word `package`.

Suppose you want a class `B` to be visible only to class `A`. For this, you define `B` inside `A` – so that `B` is an *inner class* or *nested class* of `A` – and declare class `A` to be `private`. This is only way it makes sense to have a private class. An example of such as class `B` is a `Node` class for a linked list or tree or graph, or an `Iterator` class.

Class visibility modifiers also determine whether a class can extend another. If a class `A` is declared without a modifier (package visibility) then this class can be extended only by a class `B` within the same package. If you want a class `B` to extend a class `A` from a different package, then this is only possible if `A` is defined as `public`. [1]

Visibility modifiers for fields and methods

Let's next consider modifiers for class fields and methods. Let's deal with methods first. Methods contain instructions which invoke other methods. Suppose that method `mA` is within class `A` and method `mB` is within class `B` and suppose that class `A` is visible to class `B`. [2] For an instruction in method `mB` to invoke method `mA`, method `mA` needs to be visible to class `B`. For this, it is necessary that class `A` is visible to class `B`. But this is not sufficient, since we also require that the method `mA` itself is visible to class `B`. So we have a situation as follows.

```
_____  void   mB( ){   //  visibility is left unspecified since it doesn't matter here
      A   v;
        :
      v.mA();           //  Does the compiler allow this?
}
```

The left column of the table below shows three modifiers for method `mA` in class `A`. The other two columns indicate whether the compiler allows `mB` to invoke method `mA`, as in the code above.

mA modifier	A&B in same package	A and B in diff packages
public	yes	yes
(package)	yes	no
private	no	no

[1] There is one other visibility modifier – `protected` – which allows `A` to be visible to a class in the same package and also allows `A` to be extended by classes in the same or other packages. I will not discuss the `protected` modifier further in these lecture notes and you don't need to know about this modifier for the final exam.

[2] Visiblity is *not* a symmetric relationship. It is possible for class `A` to be visible to class `B` but for `B` to not be visible to `A`, e.g. `A` might be public and `B` might be package, and `A` and `B` might be in different packages.

Encapsulation – getters and setters

If you look at various classes the Java API, you hardly ever see class fields listed – even though you know from your experience making your own classes that it is almost impossible to do anything interesting in a class without having fields. The reason that fields are not listed in the Java API for most classes is that the fields are typically private and so they are not part of the interface.

To access the information in class fields, one generally uses getter and setter methods. The reason why is best seen for the setter methods. Often there are restrictions on values that the fields can take. A field may have type `String` but the author really doesn't want to allow *any* String. For example, someone's `firstName` should not be allowed to be the String "`9*@!+`", and for a McGill Student ID (say it is `String`), a string such as `150912664` should be allowed, but the string "`1234troll`" should not.

Getter methods e.g. `getFirstName()` are examples of "accessors", and setter methods `setFirstName(Stri` `s)` are examples of "mutators". Accessors retrieve information but don't change the values of any fields. Mutators change the values of fields, but don't return anything. You can define methods that are both accessors and mutators e.g. `HashTable.put(K,V)` .

Why don't method variables have a visibility modifier ?

Often methods have local variables. These are not given a visibility modifier. Why not? Methods themselves have a visibility, and so do the classes in which they are defined. However, the variables that are defined within a method are not, and the reason is that these variable (names) have no meaning in code outside the method. So it is impossible for another method in another class to use these variabl names.

While we are talking about these variables, remember where they are in the running program. They are in the stack frame of the method which is on the call stack. See illustration in the slides.

Visibility and Inheritance

[ADDED Dec. 14: In the lecture (and recording), the slides discussing visibility and inheritance contained an error. I posted corrected slides on Dec. 2 following the lecture. Here I provide a bit more information that should help you understand the slides, in particular, slides 29-31.]

How do visibility and inheritance interact? Rathe than writing down a set of rules, let's just give one basic example which illustrates a key idea: *a subclass will inherit the fields and methods of its superclass regardless of the visiblity of these fields and methods.* For example, the `Dog` class has a `String serialNumber` field and a `Person owner` field. These fields will be `private`, and will be accessible only through `public` getter and setter methods. So, the `Beagle` class and other sub-`Dog` classes will inherit the `serialNumber` and `owner` fields. But to access these fields, a `Beagle` object will need to use `getSerialNumber()` and `setSerialNumber( String )`, etc. This is just the same way that any other class would access the private fields of a `Dog` object (or that any other class would access the fields of a `Beagle` object).

In the modified lecture slides, I gave an example with `int` type fields `x,y,z` and classes `A, B`. The claims made there about `private` variable `z` are the same as for the serioal number above.

 ©Michael Langer

"Use" modifiers

`static`

The modifier `static` specifies that a variable or method is associated with the class, not with particular instances. (It is possible to define a `static` class, but only in the special case that the class is an inner class, so let's just deal with static methods and variables here.) Static methods and fields are often used, for example, to keep track of the number of instances of a class.

```
static  int  numberOfDogs = 0;

static  int  getNumberOfDogs(){        //  this variable.
    return numberOfDogs;
}

Dog(){
   numberOfDogs ++;
}
```

Static fields are not stored in objects (instances) of the class. Rather, static fields are stored in the class descriptor for that class, which is an object of type `Class`. One can have make static variables in a class. For example, if Dog objects were organized into a monarchy, there could be a `queenDog` and `kingDog` which would be static variables in the `Dog` class.

Another example of a `static` method is `main()`. As an aside, note that you invoke this method when you "run the class" and you do this without instantiating the class. How exactly this works in the JVM is outside the scope of the present discussion.

A `static` method or field is often called a "class method" or "class field". Examples of static methods are `sin()` or `exp()` in the `java.lang.Math` class. Note that to use such methods, in your class definition you would specify that you are using the package `java.lang` and then you would invoke the method just by stating the classes name and the method e.g. `Math.cos( 0.5 )`. The class name is used instead of a reference variable (to an object) which is why we say that `static` methods are "class methods" rather than "instance methods".

`final`

The `final` modifier means different things, depending on whether it is used for a class, a method, or a variable.

- `public final class A` - means that the class cannot be extended. (A compiler error results if you write `B extends A`.)
- `final double PI = 3.1415`. The value cannot be changed. Note that you need to assign a value for this to make sense.

  The example I gave in class is that when I was child and teenager I had a beagle named "Buddy". After he passed away, I swore I would never get another dog. It was as if I declared:

```
final Dog  myDog = new Beagle(``Buddy'');
```

But life is not a computer program. Many years later, my wife and kids overruled me, so then:

```
myDog = new Poodle(``Willie'');
```

In a computer program, this instruction would have yielded a compiler (not runtime) error, since the `final` modifier for `myDog` only allows one assignment for that variable.

- `final int myMethod()` - means that the method (which happens to return an `int`) cannot be overridden in a subclass.

  `String` and `Math` are example of a `final` class.

[ASIDE: Please see the lecture slides for examples of what was discussed today. For this lecture, in particular, I would ignore the lecture recordings as some of the slides were messed up, and others were redundant. I cleaned up the slides from Sec. 001 to 002, and then made a few small further changes before posting them to the public web page.]