

Java modifiers

Some of the material today is taken from the Readings BackgroundJava.pdf which I posted early in the course. I have added further details that involve concepts from inheritance.

Packages

A *package* is a particular set of classes. Go to the Java API

<http://java.sun.com/j2se/1.5.0/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` - familiar 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.

You can also make your own packages, of course. For example, the classes `Beagle` and `Dog` might belong to package `comp250.lectureExamples`, and so the full name of the classes would be `comp250.lectureExamples.Beagle` and `comp250.lectureExamples.Dog`.

A package has a tree structure. Packages are directories, typically internal nodes¹). The classes within a package are leaves. The issue of how file systems and packages are organized is important in practice, but I did not discuss it in class. If you want to learn about package trees and how they correspond to file directories (and to `www` URLs) then you read through the following:

<http://cs.armstrong.edu/liang/intro6e/supplement/packages.pdf>]

Visibility (or access) Modifiers

You are 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 are usually defined to be `private`, and the fields are accessed using `public` getter and setter methods.

The question of `public` vs. `private` gets a bit more complicated when you consider inheritance relationships and packages. Packages come into this too, because they can determine whether classes can reference and invoke methods of other classes.

Visibility modifiers for classes

If you want a class to be visible to all other classes, declare it `public`. If you want a class to be visible only to other classes within the same package, then do not put a visibility modifier. This gives the class “package” visibility. If you define a nested class `B` within class `A` and you want class

¹the only exception being if you have an empty package in which case it would be a leaf

B to be visible only within A, then you can declare class B to be `private`. (This is only way it makes sense to have a private class.)

The visibility modifiers also determine whether a class can extend another one. If a class is declared without a modifier (package visibility) then this class can only be extended by classes within the same package. For a class (B) to extend a class (A) from a different package, the class A can be defined as `public`. However, if you only want to allow it to be visible to subclasses (or classes within the same package) then there is a special modifier for this, namely `protected`. (I will not discuss this modifier any further).

For example, our class `Dog` and `Beagle` extend class `java.lang.Object`. This is fine since the `Object` class is `public`, and so it doesn't matter that `Dog` and `Beagle` are not in the same package as `java.lang.Object`. (Rather they are in some other unspecified package. For example, maybe I have a package `examples250` and so the full path to the `Beagle` class would be `examples250.Dog.Beagle`)

Visibility modifiers for variables and methods

Let's next consider modifiers for variables and methods. Here the issue is not inheritance like with visibility modifiers for classes, but rather the issue is which methods can invoke which methods (or see which fields).

Let's deal with methods first. Methods contain instructions which invoke methods. Suppose that method `mA` is within class A and `mB` is within class B and that class A is visible to class B, e.g. A is `public`, or A has package visibility and B are in the same package, or they are in different classes but A is `protected` and B extends A.

For an instruction in method `mB` to invoke method `mA`, method `mA` in class A needs to be visible. It is necessary that class A is visible to class B, but this is not sufficient. We also require that the method `mA` itself is visible. So we have a situation like this:

```
... void mB( A v ){ // unspecified visibility of mB
    v.mA();
}
```

The left column of the table below shows the four possible (method) visibility modifiers for `mA`. The other columns consider whether `mB` can invoke method `mA`, as in the code above. A 1 means yes and 0 means no.

mA modifier	mB & mA		mB & mA	
	same class same package	diff classes same package	B extends A diff package	B not extends A diff package
-----	-----	-----	-----	-----
<code>public</code>	1	1	1	1
<code>protected</code>	1	1	1	0
(package)	1	1	0	0
<code>private</code>	1	0	0	0

Use modifiers

final

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

- `final double PI = 3.1415`. The value cannot be changed. Note that you need to assign a value for this to make sense.
- `final int myMethod()` - means that the method (which happens to return an `int`) cannot be overridden in a subclass.
- `public final ClassA` - means that the class cannot be extended (a compiler error results if you write `ClassB extends classA`.)

`String` is an example of a `final` class.

static

The modifier `static` specifies that a variable or method is associated with the class, not with particular instances (objects). It is used, for example, to keep track of the number of instances of a class, or the number of times that a method is invoked.

```
static int  numberOfObjects;           // A constructor would increment
static int  getNumberOfObjects(){     // this variable.
    return numberOfObjects;
}
```

Another example of a `static` method is `main()`.

A `static` method or field is often called a “class method” or field. A `static` method cannot contain statements that refer to instance fields or that invoke instance methods. The reason is that these instance fields and methods belong to particular instances of the class.

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)`. Note that 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”.

At the end of the lecture I spent 10 minutes reviewing the big picture of where classes, objects, methods, fields, and variables reside in the memory of the Java Virtual Machine. See the lecture slides. This discussion is not part of the official course curriculum for COMP 250, so I am not dwelling on it. But I do want you to be aware – in particular, you must understand that everything you manipulate in your Java programs exists somewhere in the memory of the JVM, and there are basically three areas of memory: the class descriptor area, the object area, and the call stack.