Notes on Organizing Java Code: Packages, Visibility, and Scope

CS 112 – Wayne Snyder

Java programming in large measure is a process of defining entities (i.e., packages, classes, methods, or fields) by name and then using those names in various contexts, and the rules for how we go about this determine how we organize our code. In this set of notes, we will consider these rules in detail; in particular, we want to understand, for each entity, the following:

1) What is the scope of the definition (the locations in code that the definition can be used) and what mechanisms exist for extending or modifying that scope?
2) How do we refer to the entity, i.e., what is the syntax of the name when it is used?
3) What happens if we have multiple definitions of a name?
4) What is the lifetime of the definition, i.e., when does it begin to be visible and when does its visibility end?

The purpose of this study is first to understand the details of how we use class and class member definitions as we write lines of Java code, but secondly we want to understand how these rules allow us to structure large Java projects following the object-oriented philosophy.

Packages and the file system

The environment in which you run Java on your computer consists of files of Java code (ending in .java) containing class definitions, compiled Java classes (files ending in .class), and rules for how a particular piece of code will execute; most of these rules involve how to name classes and how to refer to one class from another class. In this set of notes we will consider broadly how code is organized on your computer, focussing in particular on the notion of a *package*.

The main folder where you store your Java code is sometimes referred to as the *workspace*, and it is organized hierarchically as follows:

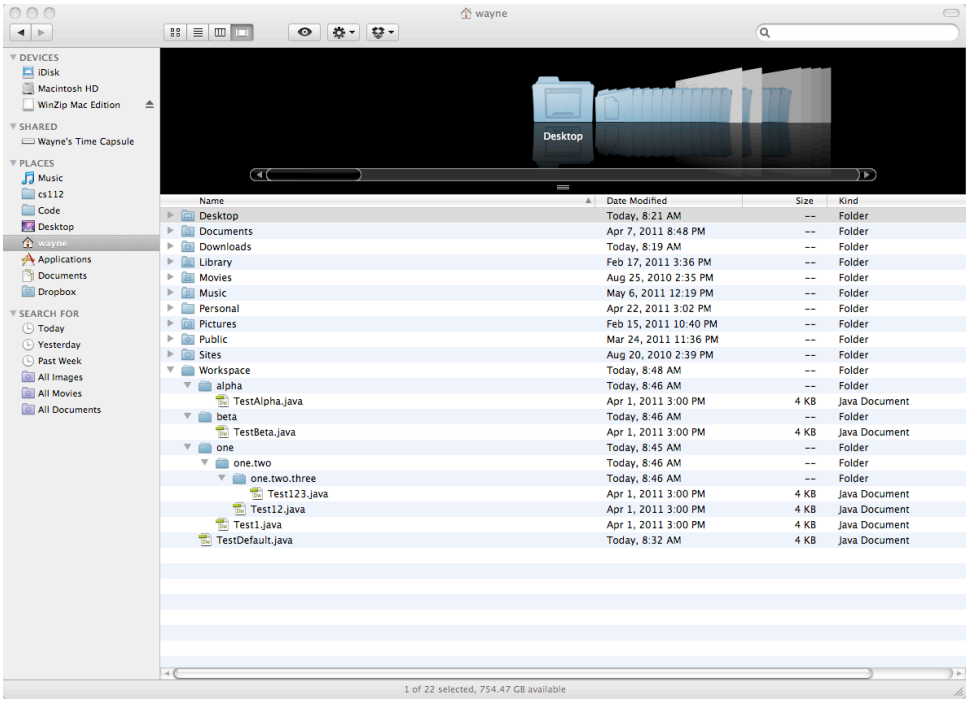
Workspace
    Packages (subfolders)
        Files
            Classes
                Members: fields, methods and nested classes

Your workspace (or working directory) may contain subfolders to organize your Java code, and these are any number of packages, and packages are typically organized in a hierarchical structure (the dotted names correspond to the folder structure) but each package is a separate entity and so this hierarchy is just for keeping your files organized in the file system: there is no concept of a "subpackage" and this hierarchy has no effect anything we discuss in these notes. If you don't specify a package when you create a class, it is put in the default or nameless package, as you can see with the class TestDefault below.

For example, consider the following files and folders in our workspace, which we will use as a running example:

Workspace
- TestDefault.java
- alpha
  - TestAlpha.java
- beta
  - TestBeta.java
- one
  - Test1.java
  - one.two
    - Test12.java
    - one.two.three
      - Test123.java

(where the folders are underlined). This might appear in your Mac as follows:



Packages: Making a file part of a package

A `.java` file contains one or more class definitions, and to be in a package it must be in the appropriate folder, as described above, and also must contain a package statement, e.g., if `Test1.java` is in package one, it must begin with the line:

```
package one;
```

and if Test123.java is in package one.two.three, it begins with the line:

```
package one.two.three;
```

A file with no package declaration is in the default package.

Packages: Using classes in the same package

When you want to refer to classes in the *same* package, you simply use their simple names (you have doubtless been doing this all along for files in the default package). However, when you want to use a public class in the same project, but in a *different* package, you have basically two choices:

1) You can use the full name with the package and the class name. For example, in the figure, to declare a `Test12` class instance from package `one.two` from inside `Test123`, you can write

   ```
   one.two.Test12 t2;
   ```

   This will work from any class in the project, including classes in the default package.[1]

2) You can import the class and then simply use the name of the class; for example, to create an instance of `Test1` inside `Test123`, you could write the following (as seen in the figure):

   ```
   import one.Test1;
   ```

   ```
   ….. Test1 t = new Test1();  ……
   ```

To import all the public classes in a package, you can use a wildcard in place of the class name, e.g., we could have imported all public classes in package one using:

```
import one.*;
```

(since there was only one class in any case, these would be equivalent).

Important note: When you import with the wildcard `*`, you get only the classes in that package/folder and no others; e.g., when you import `one.*` you do not

[1] However, there is no way to refer to classes in the default package from a different package, which is a good reason to avoid the default package!

automatically get the classes in `one.two` and `one.two.three`. You have to list all the packages separately you want to import. Again, the only reason for the hierarchical organization of packages is to organize your folders---there is no effect on the process of finding definitions of names.

Multiple definitions of a class and the buildpath

When you compile your Java code, the compile has to find the classes that you have referred to in your code, and to do this, it uses the buildpath, which is simply a list of folders to look in, one after the other, for classes. The first place it looks is in your workspace, and the second place is in the Java library that came with your compiler (this is where basic things like the Math class are stored); you can change the buildpath if you like, but that is outside the scope of these notes.

What happens when you have two classes with the same name? The rule is very simple: a class may not be redefined within its visibility, with the exception of multiple definitions in packages that are listed on the buildpath, in which case the Java compiler will use the first definition that it comes to in going down the buildpath. Thus, you may redefine the class `Math` and provide your own definition of the method `pow()`, and put it in your buildpath before the JRE, and this is the one that will be used by default. In every other case, the Java compiler will complain if you have multiple definitions for a name (as you have no doubt experienced!).

Access modifiers: public, private, and package access

Now we can understand the visibility of classes defined in files in a package. There are two possibilities:

1) A class is declared as `public`: there are no restrictions on its visibility; it can be used by any other class that can find it (e.g., in the same package, or in another package that has the appropriate buildpath).
2) A class is declared as `protected`: this has to do with subclasses, and will be dealt with late in the course when we discuss inheritance.
3) A class is declared without either of these access modifiers: in this case, the default is "package access," which means that it is visible only in its own package. The class can not be imported or named by its full package name in another package, and no modification of the buildpath will make it accessible from another project; for example, in the figure if we removed the `public` modifiers from the definitions of `Test1`, `Test12`, or `TestAlpha`, we would get a "`not visible`" error in `Test123`.

For members (i.e., fields, methods, and inner classes), we have four possibilities:

1) The member is declared as `public`: it is visible wherever the class is visible;
2) The member is delcared as `protected`: again, we shall deal with this later;

3) The member is declared as `private`, and it is visible inside the class only, and invisible outside the curly braces of the class;

4) The member is declared without a modifier, and has package access; this means it is visible inside the package only.

How to organize your classes in files

Finally, there is the matter of how classes are arranged in the files. Recall that a package is really a folder which contains files which contain definitions of classes. There can be more than one class definition per file, but we have the following restrictions on public classes:

1) There can be at most one public class per file;
2) A public class must have the same name as the file (with the .java suffix); and
3) Main methods can only occur in public classes.

It is a good idea to put any substantial class in its own file, and to have multiple classes in a file only when they are very closely related to each other; if they are only used in one class in that file, then they should be made into inner classes.

Visibility of class fields and local variables [Optional]

We will now consider the issue of visibility in the case of local variables, fields, and then the combination of these two.

Local Variables in Methods

The rule for local variables defined inside methods is actually quite simple: the scope of a local variable definition is from the point of the definition to the end of the closest enclosing pair of curly braces, and you may not redefine a variable within its scope. Definitions inside the heading of `for` loops, and parameters in methods, have scope from the point of definition to the end of the curly braces enclosing the loop or method. For example, here is the scope of the six local variables in a silly method; note carefully that `j` and `k` are both defined twice, but not redefined inside their scope, so there is no problem (e.g., the two `j`'s are different variables):

```
static void silly(int m) {
   int i = 4;

   for(int j=0; j<10; j++) {
      int k = 2;
      k = k + i + j;
   }

   for(int j=0; j<20; j++) {
      int k = 9;
      k = k + i - j;
```

```
m
m     i
m     i
m     i     j
m     i     j     k
m     i     j     k
m     i
m     i
m     i                 j
m     i                 j     k
m     i                 j     k
```

```
        }                               m     i
                                        m     i
}
```

A variable must be initialized before its first use (there is no default initialization for local variables).

Fields in Classes

The situation for fields is somewhat different, since the fields are definitions of the members available in a class, and are not so obviously part of an executable region of code. Here the rule is: a field definition has scope over the entire class, but may be redefined as part of a method or nested class; initializations (if they are done in the definition, and not in the constructor) take place in the order of the fields in the class.

Although the usual practice is to provide explicit initialization values in the constructor, we may do it in the definition in the class itself, in which case the initializations take place in the order of the fields (as if they were executable statements in a method). Thus, a method may refer to a field that is defined anywhere in the class, but field initialization may only use the definitions above it in the list of fields. Fields are initialized to `0` or `null` or `false` by default. For example, in the following class, `n` is initialized to `0`, `m` would be initialized to `4`, `k` to `4`, and `p` to `5`; the diagram shows the scope of each of the fields, and the underlined names show the "scope" of the sequential initialization. Thus, if we were to change the initialization of "`k = n + p`" it would be an error, since the definition of `p` occurs below the definition of `k`.

```
public class TestDefault {
    int n;                          n     m     k     p
    int m = 4;                      n     m     k     p
                                    n     m     k     p
    int sillyMethod(int q) {        n     m     k     p     q
       return q + n + m + k;        n     m     k     p     q
    }                               n     m     k     p
                                    n     m     k     p
    int k = n + m;                  n     m     k     p
    int p = m + 1;                  n     m     k     p
}
```

We saw above that we may not redefine a local variable in its scope. However, it is possible to redefine a field name inside its scope, by reusing the name as a local variable or a field in a nested class. This is sometimes useful, since the name may be descriptive in a way that is identical in two contexts (e.g., "i", "counter", "next"). Reusing common names may be easier and less confusing than coming up with new names. The basic idea when reusing field names, or when mixing fields and local variables, is that a use of a name refers to the closest enclosing definition. For example, in the following (contrived) example, there is a field i which is superceded

by a local variable in a method and a field inside an inner class, creating two holes in the outermost definition of i; the nested class field is in turn superceded by a local variable of the same name. Each column of i's showing the scope is a distinct variable:

```
public class TestDefault {
       int i = 1;           // field                    i
                                                        i
       class TestIt {                                   i
              int i = 2;    // field                          i
              // here i == 2                                   i
              int testItMethod() {                           i
                     int i = 3;    // local variable                  i
                     // here i == 3                                    i
                     return i;                                        i
              }
       }
                                                        i
       int testDefaultMethod() {                        i
              int i = 4;           // local variable                        i
              // here i == 4                                                 i
              TestIt t = new TestIt();                                       i
              return i;                 //  prints 4                        i
              //return this.i;              prints 1                        i
              //return t.i;                 prints 2                        i
              //return t.testItMethod();    prints 3                        i
       }
                                                        i
       public static void main (String [] cmd) {        i
              TestDefault d = new TestDefault();        i
              System.out.println(d.testDefaultMethod()); i
       }                                                i
}
```

A significant issue when redefinition is allowed is how we may refer to the various definitions. Of course, the default is to use the closest enclosing definition. However, we can also refer in some cases to other defintions, as shown in the comments attached to the various alternative return statements in `testMethod()` above:

1) `return i` yields 4 using the default rule;
2) `return this.i` yields 1, as the keyword `this` refers to the current instance of the class where this method was defined (just go outward until you hit a definition of a field `i`);
3) `return t.i` yields 2, as this is a field defined inside `t`, which is an instance of `TestIt`; and
4) `return t.testItMethod()` yields 3, as the method returns the value of its local variable.

Note that you may not refer to the field i in `TestDefault` from within the scope of the field `i` in `TestIt`, since `this` would refer to its own current instance; we will see later in the course that this will be possible when using static classes.

[End Optional]

Lifetime of Definitions: Static vs Instance Fields and Methods

Up to this point, we have not engaged with our fourth question with which we began, i.e., when do definitions come into being and when do they end? In Java this question is very simply answered:

1) When a member of a class is declared with the `static` keyword, there is exactly one instance of that member, which is created automatically before the program begins to run, and exists until the program terminates.
2) When member of a class is declared without the `static` keyword, then an instance of that member is created by the constructor for each instance of the class that is created; its lifetime is the lifetime of the instance.

In addition to the temporal issue of lifetime, it is very important to note that static members have only one instance, and although all instances of the class may use it, they all use that single instance.

Static Fields

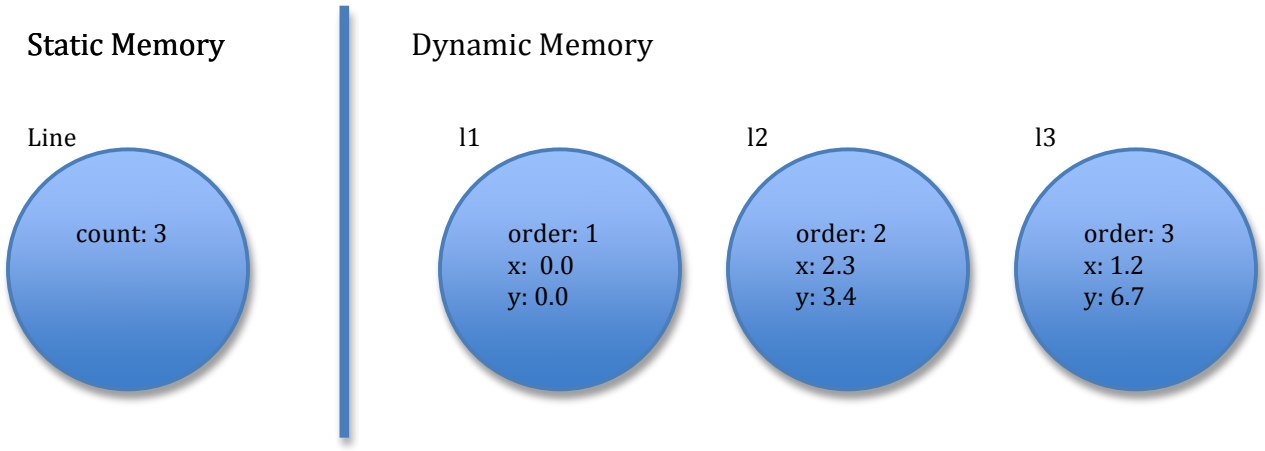
The best way to see how this works is with an example: the following class represents a line and contains a static field counting how many instances have been created:

```
class Line {
	static int count = 0;   // keeps track of how many lines exist
	public int order;       // when was this one created
	public double x, double y;

	public Line(double x, double y) {
		order = ++count;
		this.x = x; this.y = y;
	}
}

public class LineTest {
	public static void main(String[] args) {
		Line l1 = new Line(0.0, 0.0);
		Line l2 = new Line(2.3, 3.4);   // Now l1.count == 2 and l2.order == 2
		Line l3 = new Line(1.2, 6.7);   // Now l1.count == 3 and l2.order == 2
	}
}
```

We may illustrate this as follows, where we show a static region of memory which exists during the entire program lifetime, and a dynamic region of memory where (possibly multiple) instances of a class are created and (perhaps) destroyed:



When referring to members of classes, it is very important to realize that there are multiple instances of non-static members (one for each class instances), and that these must be referred either locally inside the methods of the class instances, or from outside through the name of the instance, e.g., `l1.order, l2.x`, etc. A static member may be referred in the same way as a non-static member BUT ALSO through the name of the class, e.g., `Line.count`.

Note that instance fields may be initialized by the constructor, but static fields have no explicit constructor, and must be initialized in the class definition itself (as we see with count above).

Static Methods

We may also define static methods, in which case there is exactly one instance of the method, with all its local variables (which are themselves static); for example, main() is always a static method. Again, any instance may refer to a static method by its simple name inside the class, by the instance from outside the class, and through the name of the class. However, there is an important restriction to keep in mind with static methods: a static method may not refer to a non-static member except through the name of the instance, because there may be no such instances, or multiple instances. A example will make this clear. Here is the Line class with a non-static method which calculates the length of the line:

```
class Line {
```

```
        static int count = 0;   // keeps track of how many lines exist
        public int order;   // when was this one created
        public double x, double y;

        public Line(double x, double y) {
                order = ++count;
                this.x = x; this.y = y;
        }

        double length() {
                return Math.sqrt(x*x + y*y); // sqrt is a static method of class Math!
        }
}

public class LineTest {
        public static void main(String[] args) {
                Line l1 = new Line(0.0, 0.0);
                Line l2 = new Line(2.3, 3.4);
                //   Now l2.length() would return 4.104875150354758
        }
}
```

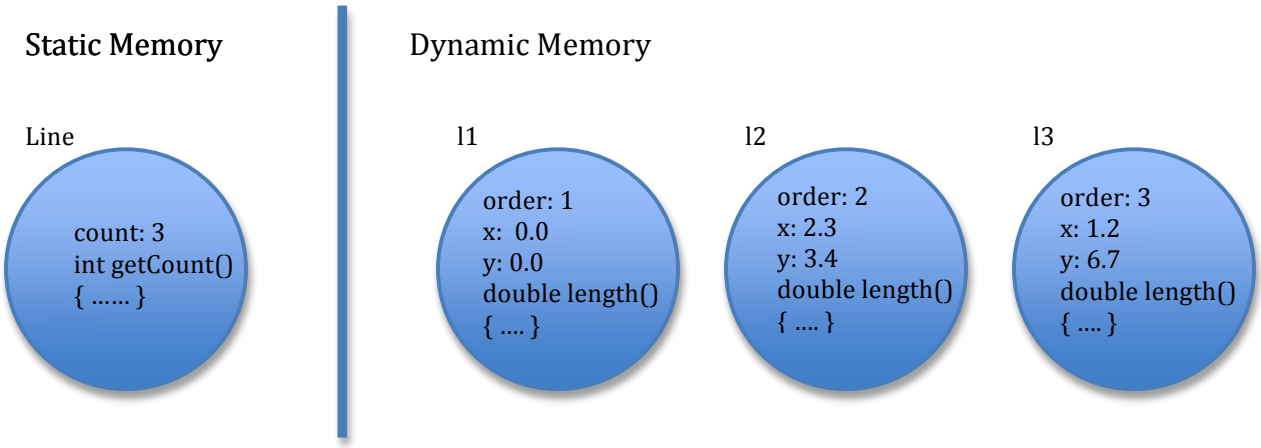
We could call `length()` inside another method of Line, or call it through the name of its instance. But what would happen if we made length() static?

```
        static double length() {
                return Math.sqrt(x*x + y*y);
        }
```

The answer is that we would get four compile-time errors "Can not make a static reference to the non-static field …, " one for each occurrence of x and y. Clearly, this definition makes no sense: which x and y are intended? However, a static method

```
        static int getCount() {
                return count;
        }
```

would generate no errors. We may imagine the code for static methods as living in the static memory instance of the class, and the non-static methods as living in each instance, and so a method may refer only to fields in its own instance:



Why are static fields and methods useful? Firstly, some utility methods, such as those in the Math class, have no need for any local data, and hence the class is just a container for a bunch of commonly-used functions; it is more efficient to have only one instance of this class rather than having to create an instance every time we want to use the Math methods. Second, sometimes, as with the field count above, it is useful to have some "global" information that all class instances can use.

Summary

1) The information hierarchy is workspace : projects : packages : files : classes : members. We may have a hierarchy of nested classes as fields inside classes or as local variables in methods.
2) Packages are stored in a hierarchical file system in their projects corresponding to the structure of their (dot-separated) compound names.
3) Files can contain any number of classes, but at most one public class, which has to have the same name as the file; public static main methods can only occur in such public classes.
4) Public entities are visible anywhere their definer is, private entities are visible only in their own class, and the default visibility is inside one's own package.
5) Classes can refer to classes in the same package by their simple names, can refer to classes in the same project but different packages by importing the package or prefixing the name of the package, and projects can refer to each other by extending the buildpath.
6) Classes may not be redefined within their visibility, with the exception of multiple definitions existing on the buildpath, in which case the compiler uses the first definition encountered.
7) The scope of a local variable is from the point of definition to the end of the closest enclosing right brace, or, in the case of method or loop parameters, to the end of the loop or method; local variables may not be redefined within their scope.
8) The scope of a field in a class is the entire class (except for holes-in-scope produced by redefinitions), but initializations take place in the field order.
9) Names of fields may be redefined by local variables in methods, or by fields in nested classes; one may refer to fields outside of their scope only by using the usual rules for referring to fields in instances of classes, the keyword `this` being used to refer to the current instance of the closest enclosing class.
10) Static fields have exactly one instance, which exists for the entire running time of the program, and which are accessible from inside any instance, and from outside through an instance name, or through the name of the class. Static methods may be referred to analogously, but may themselves only refer to static fields.