Return to Module Overview Page (https://onlinelearning.berkeley.edu/courses/665040/pages/module-four)

4.11: Packages

Packages are nothing more than groups of classes and other packages. Packages are therefore nested containers for classes. The classes we've written so far have been in the *default package*, which has no name. The Java API classes, such as java.lang.String, java.io.DataInputStream, and java.util.Stack, all appear in named packages. The name java.lang.String actually refers to the class named String in the package named lang in the package named java.

The default package (i.e., no package) should only be used for trivial test programs and other disposable fluff. Any real program should have its classes defined to be in a package. This is simple enough to do: You simply need to place a package statement as the first statement in e ach source file of the package:

```
package com.mycompany.myproject;
```

(The example shows the recommended scheme for naming packages.)
Unfortunately, using packages can make using some development environments a bit trickier. As long as your programs consist of only one source file, you might not notice any difference. If a project is spread across two or more source files, you will have to pay close attention to the proper usage of your compiler.

To use all the classes in a package (except for the java.lang package, which is special) from code inside another package without having to type the package name every time you refer to a class in it, simply use the *import* statement at the beginning of your source file:

```
import java.io.*;
```

This tells the compiler that any time you refer to, for example, DataInputStream, you mean java.io.DataInputStream. Classes in java.lang are special in that there is an implicit import java.lang.*; in every source file.

The classes in a package can have special access to one another's data. So far, I've talked about public and private class members. What if you specify neither? For example,

```
public class Circle
{
  // neither public nor private
  double x,y;
  // ...
}
```

Such members are said to have *package* protection. Package protection falls between public and private. Package-protected members are accessible to all code in the class in which they are declared, *plus* all other code in all classes in the same package. Code outside the package in which the class appears cannot access these variables. Thus you can create a group of interacting classes that can share data freely while still protecting that data from client code.

What does "public class . . ." mean? It's only slightly different than the meaning of public when applied to class members. Public classes are visible outside the package in which they are defined. Classes *not* marked public (there's no such thing as a "private" class) cannot be seen by code outside the package in which the class appears. Such classes can be considered implementation details of a package; they are subject to change at any time, which is fine because client code cannot access them.

There is an important rule regarding public and nonpublic classes and filenames: at most one public class can appear in a file of source code. If a public class appears, the file must take the name of the class; i.e., the source file

```
public class Foo() {}
class Bar() {}
class Baz() {}
```

must be named Foo.java.