# Chapter 5 : Package & Access Specifiers

Java provides ***access specifiers*** to allow the library creator to say what is available to the client programmer and what is not.

Try to keep everything as “private” as possible, and expose only the methods that you want the client programmer to use.

## 5.1 package: the library unit

```
import java.util.*;
```

For example, the class **ArrayList** is in **java.util**, you can now either specify the full name **java.util.ArrayList** (which you can do without the **import** statement), or you can simply say **ArrayList.**

The reason for all this importing is to provide a mechanism to manage "name spaces." The names of all your class members are insulated from each other. But what about the class names?

Java must be able to create a completely unique name regardless of the constraints of the Internet.

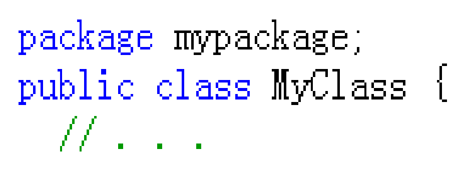
When you create a source-code file for Java, it's commonly called a *compilation unit* (*translation unit*). Each compilation unit must have a name ending in **.java**, and inside the compilation unit there can be a **public** class having the same name as the file.

The rest of the classes in that compilation unit are hidden from the world outside that package.

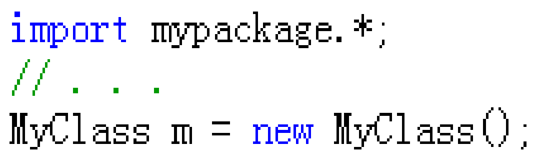
When you compile a **.java** file you get an output file with exactly the same name but an extension of **.class** *for each class in the* **.java** file. A working program is a bunch of **.class** files, which can be packaged and compressed into a JAR file.

A library is also a bunch of these class files. Each file has one class that is **public**, so there’s one component for each file. If you want to say that all these components belong together ---- package .

Note : the convention for Java package names is to use all lowercase letters.

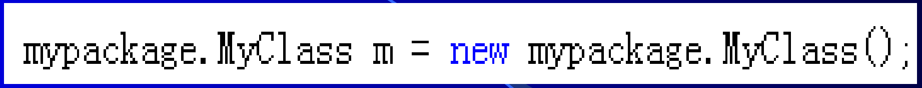


```
package mypackage;
public class MyClass {
  // . . .
```



```
import mypackage.*;
// . . .
MyClass m = new MyClass();
```

otherwise



```
mypackage.MyClass m = new mypackage.MyClass();
```

**package** and **import** keywords divide up the single global name space so you won't have clashing names.

## (1) Creating unique package names

A logical thing to do is to place all the **.class** files for a particular package into a single directory.

Questions : How to create unique package names? How to find those classes that might be buried in a directory structure?

1. Encode the path of the location of the **.class** file into the name of the **package**. (By convention, the first part of the **package** name is the reversed Internet domain name of the creator of the class. )

E:\Libraries\com\bruceeckel\simple → com.bruceeckel.simple

2. Resolve the **package** name into a directory on your machine.

- CLASSPATH contains one or more directories that are used as roots for a search for **.class** files.
- Java interpreter will take the package name and replace each dot with a slash to generate a path name :

  **foo.bar.baz** → foo\bar\baz

- It is concatenated to the various entries in the CLASSPATH.

- Java interpreter also searches some standard directories relative to its location.

Sample : Vector.java List.java

www.bruceeckel.com → com.bruceeckel.simple

e:\libraries\com\bruceeckel\simple

CLASSPATH=.; d:\j2sdk1.4.1_02\lib\tools.jar; e:\libraries

There’s a variation when using JAR files. You must put the name of the JAR file in the classpath, not just the path.

## Sample : LibTest.java (in e:\jexamples & e:\libraries)

When the compiler encounters the **import** statement, it begins searching at the directories specified by CLASSPATH, looking for subdirectory com\bruceeckel\simple, then seeking the compiled files of the appropriate names (**Vector.class** for **Vector** and **List.class** for **List**).

Note : both the classes and the desired methods in **Vector** and **List** must be **public**.

## (2) A custom tool library

With this knowledge, you can now create your own libraries of tools to reduce duplicate code.

Sample : P.java ToolTest.java

Notice : all objects can easily be forced into **String** representations by putting them in a **String** expression .

However, If you call **System.out.println(100)**, it works without casting it to a **String**.

## (3) Using imports to change behavior

*conditional compilation*

C's ***conditional compilation*** allows you to change a switch and get different behavior without changing any other code.

```
#ifdef   WIN32

/*

 * Win32 definitions

 */

#endif   /* WIN32 */
```

Since Java is intended to be automatically cross-platform, such a feature should not be necessary.

However, there are other valuable uses for conditional compilation. For example: debugging. The debugging features are enabled during development, and disabled in the final product.

Using packages to mimic conditional compilation in Java.

## Sample : Assert.java TestAssert.java

class Assert simply encapsulates Boolean tests, which print error messages if they fail. The output is printed to the console *standard error* stream.

By changing the imported **package**, we change our code from the debug version to the production version. This technique can be used for any kind of conditional code.

# (4) Package caveat

Anytime you create a package, you implicitly specify a directory structure when you give the package a name. The package *must* live in the directory indicated by its name.

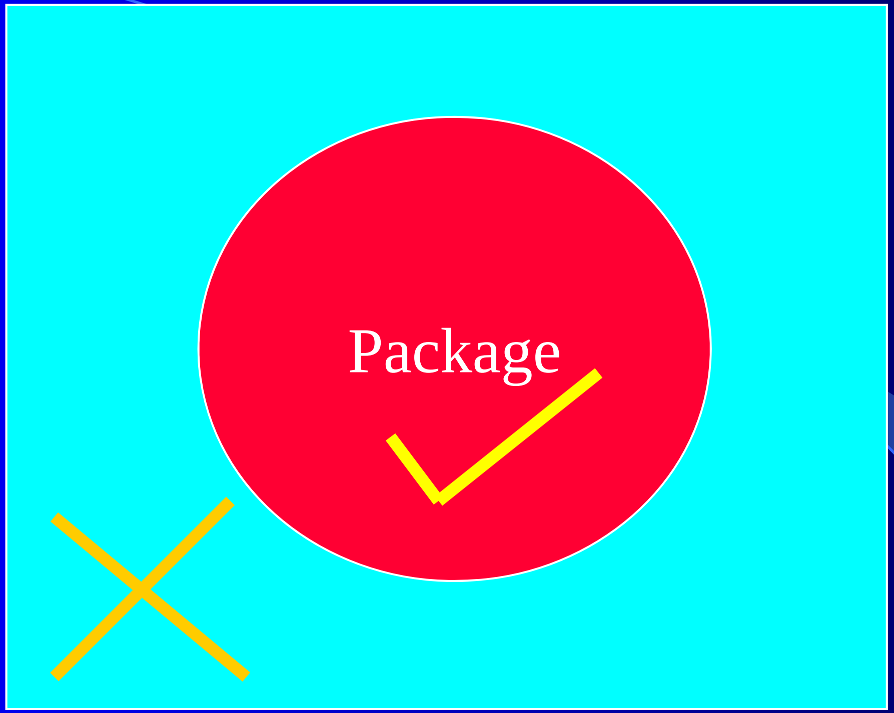
# 5.2 Java access specifiers

Java access specifiers **public**, **protected** and **private** are placed in front of each definition for each member in the class, whether it's a field or a method. Each access specifier controls the access for only that particular definition.

In C++ the access specifier controls all the definitions following it until another access specifier comes along.

## (1) "Friendly"

The default access has no keyword ---- "friendly."



All the other classes in the current package have access to the friendly member, but to all the classes outside of this package the member appears to be **private**.

All the classes within a single compilation unit are automatically friendly with each other. ---- *package access*

Friendly access allows you to group related classes together in a package so that they can easily interact with each other.

Only code you own should have friendly access to other code you own.

The only way to grant access to a member is to:

1. Make the member **public**.

2. Make the member friendly, and put the other classes in the same package.

3. An inherited class can access a **protected** member as well as a **public** member (but not **private** members).

4. Provide "accessor/mutator" methods (also known as "get/set" methods) ---- fundamental to JavaBeans

# (2) public: interface access

It means that the member declaration that immediately follows **public** is available to everyone, in particular to the client programmer.

Sample : Cookie.java Dinner.java

Note : Don’t think that Java will always look at the current directory as one of the starting points for searching “.class” files. If you don’t have a ‘**.**’ as one of the paths in your CLASSPATH, Java won’t look there.

You can create a **Cookie** object, since its constructor is **public** and the class is **public**. However, the **bite( )** member is inaccessible inside **Dinner.java** since **bite( )** is friendly.

## The default package :

Sample : Cake.java Pie.java

**Pie** and **f( )** are friendly. They are available in **Cake.java** because the two files are in the same directory and have no explicit package name. Java treats files like this as implicitly part of the "default package" for that directory.

## (3) private

The **private** keyword means that no one can access that member except that particular class. Other classes in the same package cannot access **private** members.

**private** allows you to freely change a member without concern that it will affect another class in the same package.

The default "friendly" package access often provides an adequate amount of hiding. It's tolerable to get away without private. ---- a distinct contrast with C++

However, the consistent use of **private** is very important, especially in multithreading.

Sample : IceCream.java

You might want to control how an object is created and prevent someone from directly accessing a particular constructor.

Any method that you're certain is only a "helper" method can be made **private**. Making a method private guarantees that you retain the option to change or remove it.

Unless you must expose the underlying implementation, you should make all fields **private**.

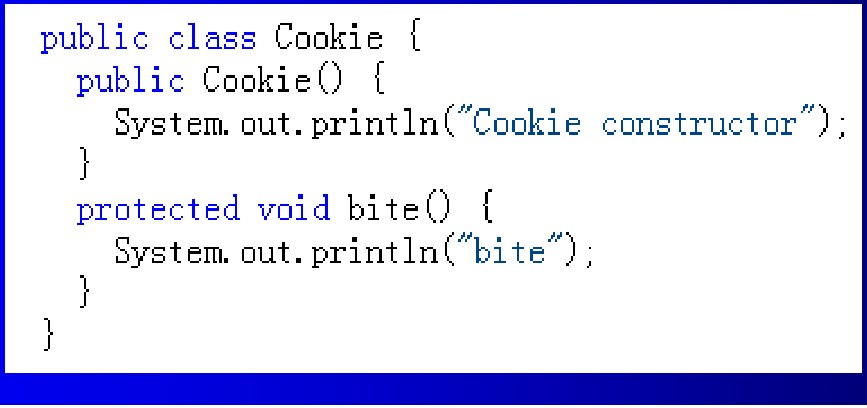
## (4) protected

The **protected** keyword deals with a concept called *inheritance*, which takes an existing class and adds new members to that class without touching the existing class. You can also change the behavior of existing members of the class.

Sometimes the creator of the base class would like to take a particular member and grant access to derived classes but not the world in general. **---- protected**

Sample : ChocolateChip.java

If a method **bite( )** exists in class **Cookie**, then it also exists in any class inherited from **Cookie**. But since **bite( )** is “friendly” in a foreign package, it’s unavailable.



```
public class Cookie {
  public Cookie() {
    System.out.println("Cookie constructor");
  }
  protected void bite() {
    System.out.println("bite");
  }
}
```

**bite( )** still has “friendly” access within package **dessert**, but it is also accessible to anyone inheriting from **Cookie**.

## 5.3 Interface and implementation

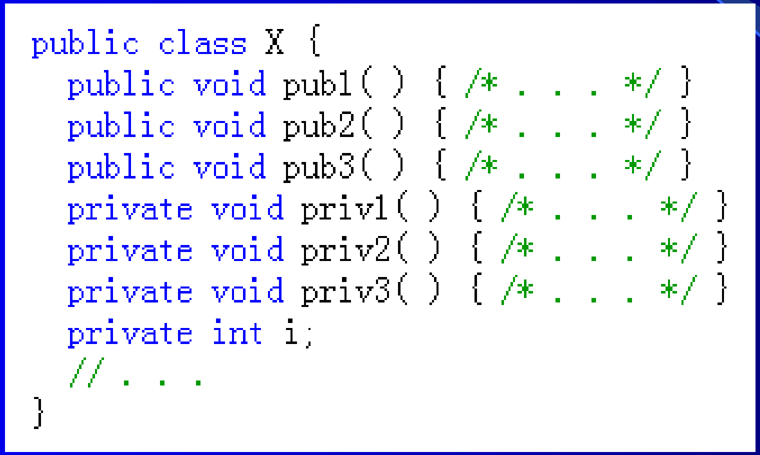
Wrapping data and methods within classes in combination with implementation hiding is often called *encapsulation*.

Access control exists for two important reasons :

1. Establish what the client programmers can and can’t use.

2. Separate the interface from the implementation. Then you can change anything that’s *not* **public** without requiring modifications to client code.

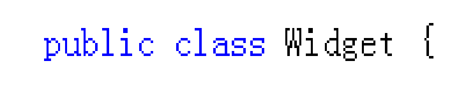
For clarity, you might prefer a style of creating classes that puts the **public** members at the beginning, followed by the **protected**, friendly, and **private** members.



```
public class X {
  public void pub1( ) { /* . . . */ }
  public void pub2( ) { /* . . . */ }
  public void pub3( ) { /* . . . */ }
  private void priv1( ) { /* . . . */ }
  private void priv2( ) { /* . . . */ }
  private void priv3( ) { /* . . . */ }
  private int i;
  // . . .
}
```

## 5.4 Class access

In Java, the access specifiers can also be used to determine which classes *within* a library will be available to the users of that library.



```
public class Widget {
```

Extra constraints:

1. There can be only one **public** class per compilation unit (file).

2. The name of the **public** class must exactly match the name of the file containing the compilation unit, including capitalization.

3. It is possible to have a compilation unit with no **public** class at all. In this case, you can name the file whatever you like.

Note :A class cannot be **private** or **protected**. So you have only two choices for class access: “friendly” or **public**.

If you don’t want anyone else to have direct access to a class, you can make all the constructors **private.**

Sample : Lunch.java

First, a **static** method is created that creates a new **Soup** and returns a reference to it.

Second, a *design pattern* named “singleton”: It allows only a single object to ever be created. And you can’t get at it except through the **public** method **access( )**.

If you don’t put an access specifier for class access it defaults to “friendly”. This means that an object of that class can be created by any other class in the package, but not outside the package.

# 5.5 Summary

When you have the ability to change the underlying implementation, you can not only improve your design later, but you also have the freedom to make mistakes. ---- private

The public interface to a class is the most important part of the class to get “right” during analysis and design. If you don’t get the interface right the first time, you can *add* more methods to fix it.