

The Java package has one more interesting characteristic; the package name corresponds where the actual file is stored on the file system. And that is actually how the compiler and the class loader find the Java files on the file system. For example, the class `com.yourcompany.yourapplication.yourmodule.YourClass`, is stored on the file system in the corresponding directory : `com/yourcompany/yourapplication/yourmodule/YourClass`. Because of this, package names should be lowercase, since in some operating systems the directory names are not case sensitive.

Wildcard imports

It is possible to import an entire package, using an asterisk:

```
import javax.swing.*;
```

While it may seem convenient, it may cause problems if you make a typographical error. For example, if you use the above import to use `JFrame`, but then type `JFram frame=new JFrame();`, the Java compiler will report an error similar to "Cannot find symbol: `JFram`". Even though it seems as if it was imported, the compiler is giving the error report at the first mention of `JFram`, which is half-way through your code, instead of the point where you imported `JFrame` along with everything else in `javax.swing`.

If you change this to `import javax.swing.JFrame;` the error will be at the import instead of within your code.

Furthermore, if you import `javax.swing.*;` and `import java.util.*;`, and `javax.swing.Queue` is later added in a future version of Java, your code that uses `Queue` (`java.util`) will fail to compile. This particular example is fairly unlikely, but if you are working with non-Sun libraries, it may be more likely to happen.

Importing packages from .jar files

If you are importing library packages or classes that reside in a .jar file, you must ensure that the file is in the current classpath (both at compile- and execution-time). Apart from this requirement, importing these packages and classes is the same as if they were in their full, expanded, directory structure.



Example:

To compile and run a class from a project's top directory (that contains the two directories `/source` and `/libraries`) you could use the following command:

```
javac -classpath libraries/lib.jar source/MainClass.java
```

And then to run it, similarly:

```
java -classpath libraries/lib.jar source/MainClass
```

(The above is simplified, and demands that `MainClass` be in the default package, or a package called 'source', which isn't very desirable.)

Class Loading / Name Space

A fully qualified class name

consist of the package name plus the class name.

For example, the fully qualified class name of `HashMap` is `java.util.HashMap`.

Sometime is can happen that two class has the same name, but it can not have on the same package, otherwise it would be the same class.

It can be said that the two class with the same name is in different name space. In the above example, the `HashMap` class is in the `java.util` name space.

Let be two `Customer` class with different name space (in different package).

- `com.bluecompany.Customer`
- `com.redcompany.Customer`

When we need to use both class in the same program file, we can use the **import** keyword only for one of the class. For the other we need to use the fully qualified name.

The runtime identity of a class in Java 2

is defined by the fully qualified class name and its defining class loader. This means that the same class, loaded by two different class loaders, is seen by the Virtual Machine as two completely different types.

Nested Classes

Navigate **Language Fundamentals** topic:

In Java you can define a class inside an other class.

A class can be nested:

- inside another class,
- or inside a method

Nest a class inside a class

When a class is declared inside another class, the nested class' access modifier can be **public**, **private** or package(default).

```
public class OuterClass
{
    private String outerInstanceVar;

    public class InnerClass
    {
        public void printVars()
        {
            System.out.println( "Print Outer Class Instance Var.:" +
outerInstanceVar);
        }
    }
}
```

The inner class has access to the enclosing class instance's variables and methods, even private ones, as seen above. This makes it very different from the nested class in C++, which are equivalent to the "static" inner classes, see below.

An inner object has a reference to the outer object. The nested object can only be created with a reference to the 'outer' object. See below.

```
public void testInner()
{
    ...
    OuterClass outer = new OuterClass();
    OuterClass.InnerClass inner = outer.new OuterClass.InnerClass();
}
```

```
...  
}
```

(When in a non-static method of the outer class, you can directly use `new InnerClass()`, since the class instance is implied to be this.)

You can directly access the reference to the outer object from within an inner class with the syntax `OuterClass.this`; although this is usually unnecessary because you already have access to its fields and methods.

Inner classes compile to separate ".class" bytecode files, usually with the name of the enclosing class, followed by a "\$", followed by the name of the inner class. So for example, the above inner class would typically be compiled to a file named "OuterClass\$InnerClass.class".

Static inner class

An inner class can be declared *static*. A static inner class has no enclosing instance, and therefore cannot access instance variables and methods of the outer class. You do not specify an instance when creating a static inner class. This is equivalent to the inner classes in C++.

Nest a class inside a method

These inner classes, also called *local classes*, cannot have access modifiers, like local variables, since the class is 'private' to the method. The inner class can be only **abstract** or **final**.

```
public class OuterClass  
{  
    public void method()  
    {  
        class InnerClass  
        {  
  
        }  
    }  
}
```

In addition to instance variables of the enclosing class, local classes can also access local variables of the enclosing method, but only ones that are declared `final`. This is because the local class instance might outlive the invocation of the method, and so needs its own copy of the variable. To avoid problems with having two different copies of a mutable variable with the same name in the same scope, it is required to be *final*, so it cannot be changed.

Anonymous Classes

In Java a class definition and its instantiation can be combined into a single step. By doing that the class does not require a name. Those classes are called anonymous classes. An anonymous class can be defined and instantiated in contexts where a reference can be used, and it is a nested class to an existing class. Anonymous class is a special case of the local class to a method, above; and hence they also can use `final` local variables of the enclosing method.

Anonymous classes are most useful to subclass and upcast to an 'Adapter Class' or to an interface.

```
public interface ActionListener  
{  
    public void click();  
}
```