Packages & Java Naming Conventions

Java Naming Conventions

- Java naming conventions must be followed while developing software in java for good maintenance and readability of code.
- Java uses **CamelCase** as a practice for writing names of methods, variables, classes, packages and constants.
- Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.
- But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as **Sun Microsystems** and **Netscape**.
- All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion.

Java Naming Conventions

Class:

- It should start with the uppercase letter.
- It should be a noun such as Color, Button, System, Thread, etc.
- Use appropriate words, instead of acronyms.

Example: -

```
public class Employee
{
        //code snippet
}
```

Interface:

- It should start with the uppercase letter.
- It should be an adjective such as Runnable, Remote, ActionListener.
- Use appropriate words, instead of acronyms.

Example: -

```
interface Printable
{
        //code snippet
}
```

Java Naming Conventions

Method:

- It should start with lowercase letter.
- It should be a verb such as main(), print(), println().
- If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().

Example: -

```
class Employee
{
    //method
    void draw()
    {
        //code snippet
    }
}
```

Java Naming Conventions

Variable:

- It should start with a lowercase letter such as id, name.
- It should not start with the special characters like & (ampersand), $ (dollar), _ (underscore).
- If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.

Example: -

```
class Employee
{
    //variable
    int id;
    //code snippet
}
```

Java Naming Conventions

Package:

- It should be a lowercase letter such as java, lang.
- It should be separated by dots (.) such as java.util, java.lang.

Example: -

```
package com.javapack; //package
class Employee
{
        //code snippet
}
```

Java Naming Conventions

Constant:

- It should be in uppercase letters such as RED, YELLOW.
- If the name contains multiple words, it should be separated by an underscore(_) such as MAX_PRIORITY.
- It may contain digits but not as the first letter.

Example: -

```
package com.javapack; //package
class Employee
{
    //constant
     static final int MIN_AGE = 18;
    //code snippet
}
```

Good programming style

- Use good names for variables and methods
- Use indentation
- Add whitespaces
- Don't duplicate tests

Packages

- A Package can be defined as a grouping of related types classes, interfaces and sub-packages.
- Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces.
- Package in java can be categorized in two form, built-in (or existing) package and user-defined package.
- Some of the existing packages in Java are −
- java.lang − bundles the fundamental classes
- java.io − classes for input , output functions are bundled in this package

Defining a Package

- To create a package is quite easy: simply include a **package** command.
- The package statement defines a name space in which classes are stored.
- At most one package declaration can appear in a source file, and it must be the **first statement** in the source Java file.
- This is the general form of the package statement:

  package *pack*;

- Here, *pack* is the name of the package.
- If a package declaration is omitted in a compilation unit, the Java byte code for the declarations in the compilation unit will belong to an unnamed package (also called the default package), which is typically synonymous with the current working directory on the host system.

Defining a Package

- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a **period**.
- The general form of a multileveled package statement is:

```
package pack1[.pack2[.pack3]];
```

Using Packages

- The import facility in Java makes it easier to use the contents of packages.
- The accessibility of types (classes and interfaces) in a package determines their access from other packages.
- The contents of packages accessible from outside a package, using two ways.

1) Using the fully qualified name.
2) Using the import declaration.

Using Packages

1) Using the fully qualified name.

- The fully qualified name of a class is the name of the class prefixed with the package name.
- However, writing long names can become tedious.

2) Using the import declaration.

- The import declarations must be the first statement after any package declaration in a source file.

➢ single-type-import:

import <fully qualified type name>;

Example **import pkg.Test**;

➢ multiple-type-import:

import <fully qualified package name>.*;

Example **import pkg.***;

Using Packages

- Two classes that have the same name but are in different packages.
- To distinguish between the two classes, we can use their fully qualified names.
- A single-type-import declaration can be used to disambiguate a type name when access to the type is ambiguous.
- For example both, java.util and java.sql packages have a class named Date.

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

- then we will get a compile-time error.
- The compiler will not be able to figure out which Date class do we want.
- We need to use a full package name every time we declare a new object of that class.

```
java.util.Date deadLine = new java.util.Date();
java.sql.Date today = new java.sql.Date();
```

- Members of the **java.lang** package are imported automatically.
- More than one file can include the same package statement.

Compiling Code into Packages

- The location in the file system where the package directory should be created is specified using the **-d** option (d for destination) of the javac command.
- The compiler will create the package directory under the specified location, and place the Java byte code inside the package directory.

```
javac -d directory javafilename
```

- We can use . (dot), If you want to keep the package within the same directory.

```
javac -d . javafilename
```

Compiling Code into Packages

```
package packA;
public class A{
        public void test()
        {
                System.out.println("Hello I am in class A in package packA");
        }
}
   ------------------------------------------------------------------------------------------- -----
package packB;
import packA.*;
class B
{
        public static void main(String args[]){
        A obj = new A();
        obj.test();
        }
}
```

Compiling Code into Packages

```
package packA;
public class A{
        public void test()
        {
                System.out.println("Hello I am in class A in package packA");
        }
}
```

Compile:

```
javac -d . A.java
```

-- -----

```
package packB;
import packA.*;
class B
{
        public static void main(String args[]){
        A obj = new A();
        obj.test();
        }
}
```

Compile:

```
javac -d . B.java
```

Run:

```
java packB.B
```