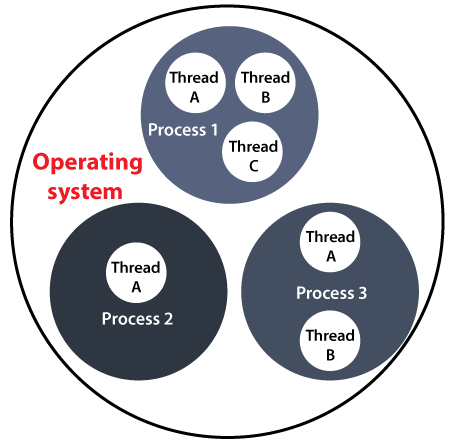
Thread Concept in Java

Before introducing the **thread concept**, we were unable to run more than one task in parallel. It was a drawback, and to remove that drawback, **Thread Concept** was introduced.

A **Thread** is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.

In order to perform complicated tasks in the background, we used the **Thread concept in Java**. All the tasks are executed without affecting the main program. In a program or process, all the threads have their own separate path for execution, so each thread of a process is independent.



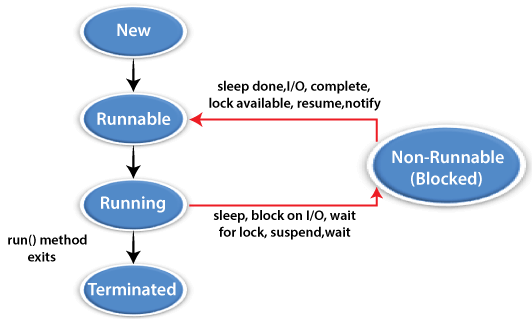
Another benefit of using **thread** is that if a thread gets an exception or an error at the time of its execution, it doesn't affect the execution of the other threads. All the threads share a common memory and have their own stack, local variables and program counter. When multiple threads are executed in parallel at the same time, this process is known as [**Multithreading**](https://www.javatpoint.com/multithreading-in-java).

In a simple way, a Thread is a:

* Feature through which we can perform multiple activities within a single process.
* Lightweight process.
* Series of executed statements.
* Nested sequence of method calls.

Thread Model

Just like a process, a thread exists in several states. These states are as follows:



**1) New (Ready to run)**

A thread is in **New** when it gets CPU time.

**2) Running**

A thread is in **a Running** state when it is under execution.

**3) Suspended**

A thread is in the **Suspended** state when it is temporarily inactive or under execution.

**4) Blocked**

A thread is in the **Blocked** state when it is waiting for resources.

**5) Terminated**

A thread comes in this state when at any given time, it halts its execution immediately.

Creating Thread

A thread is created either by "creating or implementing" the **Runnable Interface** or by extending the **Thread class**. These are the only two ways through which we can create a thread.

Let's dive into details of both these way of creating a thread:

Thread Class

A **Thread class** has several methods and constructors which allow us to perform various operations on a thread. The Thread class extends the **Object** class. The **Object** class implements the **Runnable** interface. The thread class has the following constructors that are used to perform various operations.

* **Thread()**
* **Thread(Runnable, String name)**
* **Thread(Runnable target)**
* **Thread(ThreadGroup group, Runnable target, String name)**
* **Thread(ThreadGroup group, Runnable target)**
* **Thread(ThreadGroup group, String name)**
* **Thread(ThreadGroup group, Runnable target, String name, long stackSize)**

Runnable Interface(run() method)

The Runnable interface is required to be implemented by that class whose instances are intended to be executed by a thread. The runnable interface gives us the **run()** method to perform an action for the thread.

start() method

The method is used for starting a thread that we have newly created. It starts a new thread with a new callstack. After executing the **start()** method, the thread changes the state from New to Runnable. It executes the **run() method** when the thread gets the correct time to execute it.

Let's take an example to understand how we can create a [Java](https://www.javatpoint.com/java-tutorial) thread by extending the Thread class:

**ThreadExample1.java**

// Implementing runnable interface by extending Thread class

**public** **class** ThreadExample1 **extends** Thread {

     // run() method to perform action for thread.

**public** **void** run()

     {

**int** a= 10;

**int** b=12;

**int** result = a+b;

        System.out.println("Thread started running..");

        System.out.println("Sum of two numbers is: "+ result);

     }

**public** **static** **void** main( String args[] )

     {

      // Creating instance of the class extend Thread class

        ThreadExample1 t1 = **new**  ThreadExample1();

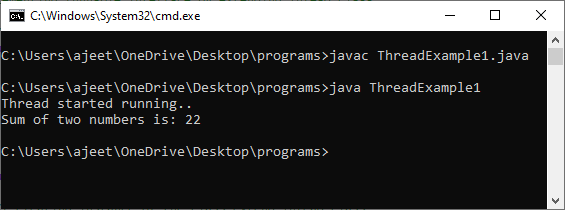
        //calling start method to execute the run() method of the Thread class

        t1.start();

     }

}

**Output:**



Creating thread by implementing the runnable interface

In Java, we can also create a thread by implementing the runnable interface. The runnable interface provides us both the run() method and the start() method.

Let's takes an example to understand how we can create, start and run the thread using the runnable interface.

**ThreadExample2.java**

**class** NewThread **implements** Runnable {

    String name;

    Thread thread;

    NewThread (String name){

**this**.name = name;

        thread = **new** Thread(**this**, name);

        System.out.println( "A New thread: " + thread+ "is created\n" );

        thread.start();

    }

**public** **void** run() {

**try** {

**for**(**int** j = 5; j > 0; j--) {

            System.out.println(name + ": " + j);

            Thread.sleep(1000);

        }

    }**catch** (InterruptedException e) {

        System.out.println(name + " thread Interrupted");

    }

     System.out.println(name + " thread exiting.");

    }

}

**class** ThreadExample2 {

**public** **static** **void** main(String args[]) {

**new** NewThread("1st");

**new** NewThread("2nd");

**new** NewThread("3rd");

**try** {

            Thread.sleep(8000);

        } **catch** (InterruptedException excetion) {

            System.out.println("Inturruption occurs in Main Thread");

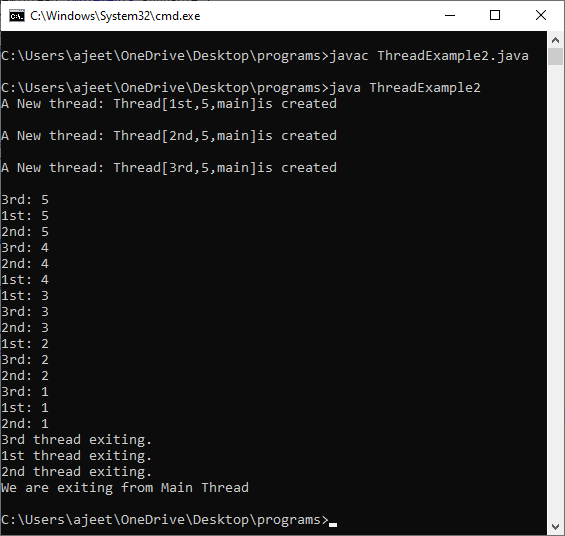
        }

        System.out.println("We are exiting from Main Thread");

    }

}

**Output:**



Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

**public final int getPriority():** The java.lang.Thread.getPriority() method returns the priority of the given thread.

**public final void setPriority(int newPriority):** The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. public static int MIN\_PRIORITY
2. public static int NORM\_PRIORITY
3. public static int MAX\_PRIORITY

Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.

Example of priority of a Thread:

**FileName:** ThreadPriorityExample.java

// Importing the required classes

**import** java.lang.\*;

**public** **class** ThreadPriorityExample **extends** Thread

{

// Method 1

// Whenever the start() method is called by a thread

// the run() method is invoked

**public** **void** run()

{

// the print statement

System.out.println("Inside the run() method");

}

// the main method

**public** **static** **void** main(String argvs[])

{

// Creating threads with the help of ThreadPriorityExample class

ThreadPriorityExample th1 = **new** ThreadPriorityExample();

ThreadPriorityExample th2 = **new** ThreadPriorityExample();

ThreadPriorityExample th3 = **new** ThreadPriorityExample();

// We did not mention the priority of the thread.

// Therefore, the priorities of the thread is 5, the default value

// 1st Thread

// Displaying the priority of the thread

// using the getPriority() method

System.out.println("Priority of the thread th1 is : " + th1.getPriority());

// 2nd Thread

// Display the priority of the thread

System.out.println("Priority of the thread th2 is : " + th2.getPriority());

// 3rd Thread

// // Display the priority of the thread

System.out.println("Priority of the thread th2 is : " + th2.getPriority());

// Setting priorities of above threads by

// passing integer arguments

th1.setPriority(6);

th2.setPriority(3);

th3.setPriority(9);

// 6

System.out.println("Priority of the thread th1 is : " + th1.getPriority());

// 3

System.out.println("Priority of the thread th2 is : " + th2.getPriority());

// 9

System.out.println("Priority of the thread th3 is : " + th3.getPriority());

// Main thread

// Displaying name of the currently executing thread

System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName());

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

// Priority of the main thread is 10 now

Thread.currentThread().setPriority(10);

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

}

}

**Output:**

Priority of the thread th1 is : 5

Priority of the thread th2 is : 5

Priority of the thread th2 is : 5

Priority of the thread th1 is : 6

Priority of the thread th2 is : 3

Priority of the thread th3 is : 9

Currently Executing The Thread : main

Priority of the main thread is : 5

Priority of the main thread is : 10

We know that a thread with high priority will get preference over lower priority threads when it comes to the execution of threads. However, there can be other scenarios where two threads can have the same priority. All of the processing, in order to look after the threads, is done by the Java thread scheduler. Refer to the following example to comprehend what will happen if two threads have the same priority.

**FileName:** ThreadPriorityExample1.java

// importing the java.lang package

**import** java.lang.\*;

**public** **class** ThreadPriorityExample1 **extends** Thread

{

// Method 1

// Whenever the start() method is called by a thread

// the run() method is invoked

**public** **void** run()

{

// the print statement

System.out.println("Inside the run() method");

}

// the main method

**public** **static** **void** main(String argvs[])

{

// Now, priority of the main thread is set to 7

Thread.currentThread().setPriority(7);

// the current thread is retrieved

// using the currentThread() method

// displaying the main thread priority

// using the getPriority() method of the Thread class

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

// creating a thread by creating an object of the class ThreadPriorityExample1

ThreadPriorityExample1 th1 = **new** ThreadPriorityExample1();

// th1 thread is the child of the main thread

// therefore, the th1 thread also gets the priority 7

// Displaying the priority of the current thread

System.out.println("Priority of the thread th1 is : " + th1.getPriority());

}

}

**Output:**

Priority of the main thread is : 7

Priority of the thread th1 is : 7

**Explanation:** If there are two threads that have the same priority, then one can not predict which thread will get the chance to execute first. The execution then is dependent on the thread scheduler's algorithm (First Come First Serve, Round-Robin, etc.)

Example of IllegalArgumentException

We know that if the value of the parameter *newPriority* of the method getPriority() goes out of the range (1 to 10), then we get the IllegalArgumentException. Let's observe the same with the help of an example.

**FileName:** IllegalArgumentException.java

// importing the java.lang package

**import** java.lang.\*;

**public** **class** IllegalArgumentException **extends** Thread

{

// the main method

**public** **static** **void** main(String argvs[])

{

// Now, priority of the main thread is set to 17, which is greater than 10

Thread.currentThread().setPriority(17);

// The current thread is retrieved

// using the currentThread() method

// displaying the main thread priority

// using the getPriority() method of the Thread class

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

}

}

When we execute the above program, we get the following exception:

Exception in thread "main" java.lang.IllegalArgumentException

at java.base/java.lang.Thread.setPriority(Thread.java:1141)

at IllegalArgumentException.main(IllegalArgumentException.java:12)

Package Naming Conversion in Java

In Java, [package](https://www.javatpoint.com/package) plays an important role in preventing naming conflicts, controlling access, and making searching and usage of classes, enumeration, interfaces, and annotation easier.

In order to group classes, interfaces, and sub-packages that are related to each other, we use packages. By using packages:

* It is very easy to find related types because they contain types that are logically related.
* We can uniquely identify the class name and avoid naming conflicts.
* We can easily control the access of types by combining access modifiers and packages.

Now, let's understand how we can create packages and their naming conventions one by one.

Creating a Package

We use the **package** keyword as a very first line of code in the file for creating a package in Java4sz. We create a package by adding the package keyword as the very first line of code in a file. The syntax of creating a package in Java is as follows:

1. **package** com.javatpoint.packages

We highly recommend you to add only a unique type in a package. If we don't place the defined type in a package, they will be placed in the default or unnamed package.

These are the following disadvantages of putting types into unnamed packages:

* There are no benefits of having a package structure.
* The sub-packages are not possible.
* Importing the types in the default package from the other packages is not possible.
* There is no use of protected and package-private access scopes.

So, we should have to avoid the use of unnamed or default packages in RWA(Real World Applications). To learn more about packages, go through the following link:

Naming Conventions

For avoiding unwanted package names, we have some following naming conventions which we use in creating a package.

* The name should always be in the lower case.
* They should be period-delimited.
* The names should be based on the company or organization name.

In order to define a package name based on an organization, we'll first reverse the company URL. After that, we define it by the company and include division names and project names.

How to use package members?

First, we define a class Test in a sub-package name examples:

// sub-package

**package** com.javatpoint.packages.examples;

// create class Test in examples package

**public** **class** Test {

**private** Long userId;

**private** String password;

    // standard getters and setters

}

Now, if we want to use Test class from outside the package, we will import it in the following way:

1. **import** com.javatpoint.packages.examples.Test;

In Java, we have some pre-defined classes and package available. We import the classes available in the pre-defined packages in the same way as we import classes in our own packages.

If we want to import ArrayList and List classes from the **util** package, we will import it in the following way:

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **public** **class** TestList {
4. **private** List<String> names;
5. **public** **void** addNames(String name) {
6. **if** (names == **null**) {
7. names = **new** ArrayList<String>();
8. }
9. names.add(name);
10. }
11. }

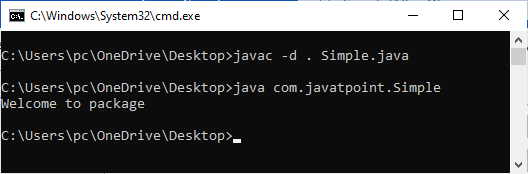
Let's take an example to understand how we can create a package, run it and compile it using Java.

**Simple.java**

Learn more

1. //save as Simple.java
2. **package** com.javatpoint;
3. // create Simple class in mypackage
4. **public** **class** Simple{
5. // main() method of Simple class
6. **public** **static** **void** main(String args[]){
7. // print package statement
8. System.out.println("Welcome to package");
9. }
10. }

**Output:**



# Java Package

1. [Java Package](https://www.javatpoint.com/package)
2. [Example of package](https://www.javatpoint.com/package#packageex)
3. [Accessing package](https://www.javatpoint.com/package#packageaccess)
   1. [By import packagename.\*](https://www.javatpoint.com/package#packageaccess1)
   2. [By import packagename.classname](https://www.javatpoint.com/package#packageaccess2)
   3. [By fully qualified name](https://www.javatpoint.com/package#packageaccess3)
4. [Subpackage](https://www.javatpoint.com/package#packagesub)
5. [Sending class file to another directory](https://www.javatpoint.com/package#packageanotherdirectory)
6. [-classpath switch](https://www.javatpoint.com/package#packageclasspathswitch)
7. [4 ways to load the class file or jar file](https://www.javatpoint.com/package#packagewaystoload)
8. [How to put two public class in a package](https://www.javatpoint.com/package#packagetwopublic)
9. [Static Import](https://www.javatpoint.com/package#packagestaticimport)
10. [Package class](https://www.javatpoint.com/package-class)

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

## Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



## Simple example of java package

The **package keyword** is used to create a package in java.

1. //save as Simple.java
2. **package** mypack;
3. **public** **class** Simple{
4. **public** **static** **void** main(String args[]){
5. System.out.println("Welcome to package");
6. }
7. }

## How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

## How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

Learn more

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java mypack.Simple |

Output:Welcome to package

|  |
| --- |
| The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder. |

## How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.\*;
2. import package.classname;
3. fully qualified name.

#### 1) Using packagename.\*

If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

## Example of package that import the packagename.\*

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **public** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
10. **class** B{
11. **public** **static** **void** main(String args[]){
12. A obj = **new** A();
13. obj.msg();
14. }
15. }

Output:Hello

#### 2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

## Example of package by import package.classname

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
8. **package** mypack;
9. **import** pack.A;
11. **class** B{
12. **public** **static** **void** main(String args[]){
13. A obj = **new** A();
14. obj.msg();
15. }
16. }

Output:Hello

#### 3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

Pause

Unmute

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

## Example of package by import fully qualified name

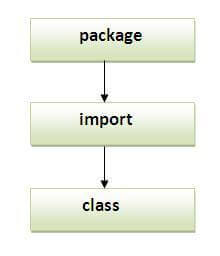
1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **public** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **class** B{
9. **public** **static** **void** main(String args[]){
10. pack.A obj = **new** pack.A();//using fully qualified name
11. obj.msg();
12. }
13. }

Output:Hello

#### Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

#### Note: Sequence of the program must be package then import then class.



## Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

#### The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

### Example of Subpackage

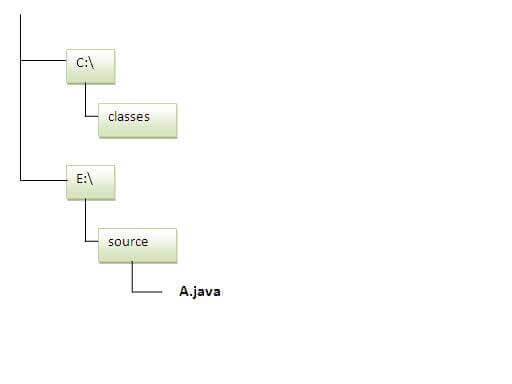
1. **package** com.javatpoint.core;
2. **class** Simple{
3. **public** **static** **void** main(String args[]){
4. System.out.println("Hello subpackage");
5. }
6. }

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java com.javatpoint.core.Simple |

Output:Hello subpackage

## How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:



1. //save as Simple.java
2. **package** mypack;
3. **public** **class** Simple{
4. **public** **static** **void** main(String args[]){
5. System.out.println("Welcome to package");
6. }
7. }

### To Compile:

**e:\sources> javac -d c:\classes Simple.java**

### To Run:

|  |
| --- |
| To run this program from e:\source directory, you need to set classpath of the directory where the class file resides. |
| **e:\sources> set classpath=c:\classes;.;** |
| **e:\sources> java mypack.Simple** |

### Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

**e:\sources> java -classpath c:\classes mypack.Simple**

Output:Welcome to package

### Ways to load the class files or jar files

|  |
| --- |
| There are two ways to load the class files temporary and permanent. |

* Temporary
  + By setting the classpath in the command prompt
  + By -classpath switch
* Permanent
  + By setting the classpath in the environment variables
  + By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

#### Rule: There can be only one public class in a java source file and it must be saved by the public class name.

1. //save as C.java otherwise Compilte Time Error
3. **class** A{}
4. **class** B{}
5. **public** **class** C{}

### How to put two public classes in a package?

|  |
| --- |
| If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example: |

1. //save as A.java
3. **package** javatpoint;
4. **public** **class** A{}
5. //save as B.java
7. **package** javatpoint;
8. **public** **class** B{}

Package class

The package class provides methods to get information about the specification and implementation of a package. It provides methods such as getName(), getImplementationTitle(), getImplementationVendor(), getImplementationVersion() etc.

Example of Package class

In this example, we are printing the details of java.lang package by invoking the methods of package class.

1. **class** PackageInfo{
2. **public** **static** **void** main(String args[]){
4. Package p=Package.getPackage("java.lang");
6. System.out.println("package name: "+p.getName());
8. System.out.println("Specification Title: "+p.getSpecificationTitle());
9. System.out.println("Specification Vendor: "+p.getSpecificationVendor());
10. System.out.println("Specification Version: "+p.getSpecificationVersion());
12. System.out.println("Implementaion Title: "+p.getImplementationTitle());
13. System.out.println("Implementation Vendor: "+p.getImplementationVendor());
14. System.out.println("Implementation Version: "+p.getImplementationVersion());
15. System.out.println("Is sealed: "+p.isSealed());

18. }
19. }

**Output:**package name: java.lang

Specification Title: Java Plateform API Specification

Specification Vendor: Sun Microsystems, Inc.

Specification Version: 1.6

Implemenation Title: Java Runtime Environment

Implemenation Vendor: Sun Microsystems, Inc.

Implemenation Version: 1.6.0\_30

IS sealed: false