Java Regex

The **Java Regex** or Regular Expression is an API to *define a pattern for searching or manipulating strings*.

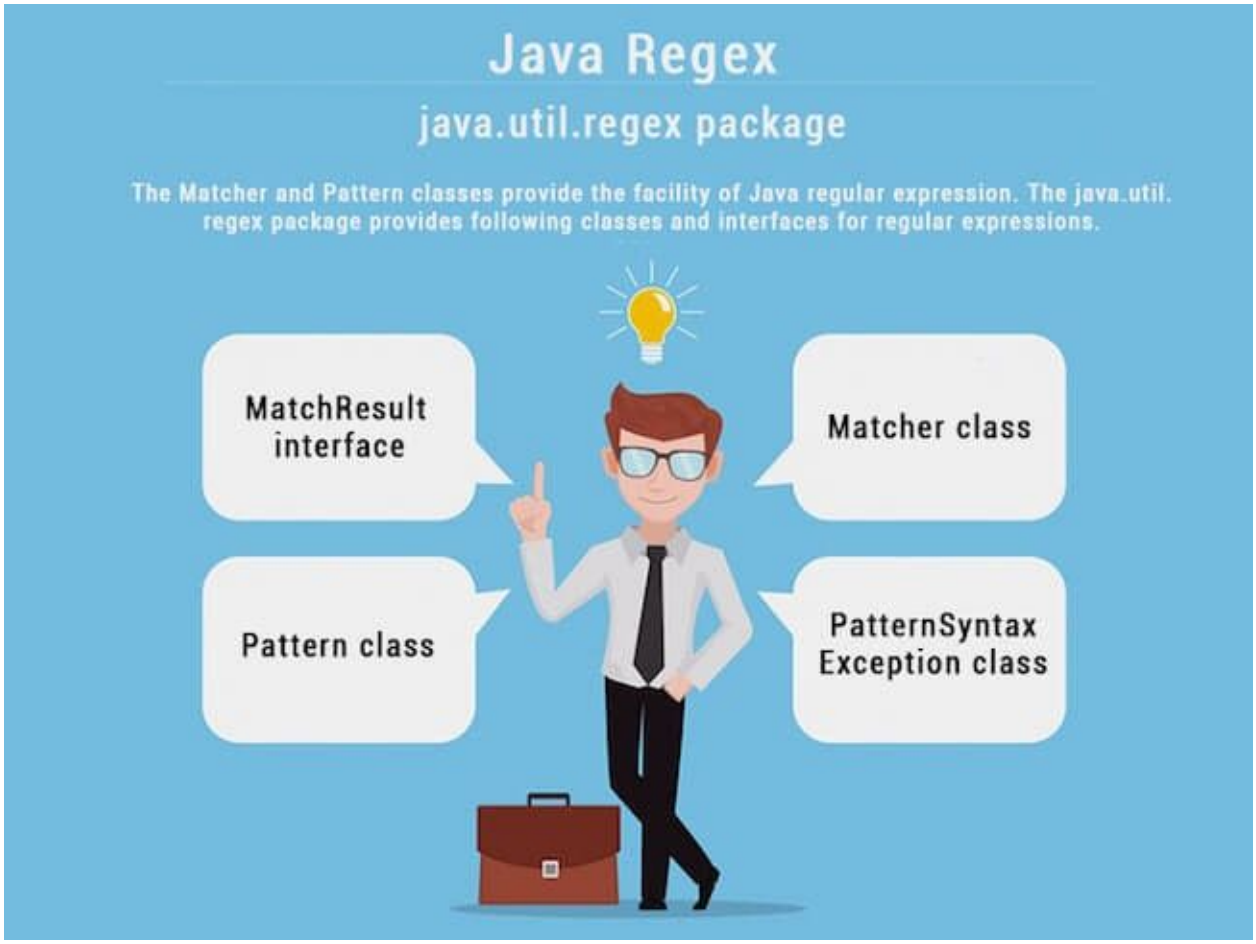
It is widely used to define the constraint on strings such as password and email validation. After learning Java regex tutorial, you will be able to test your regular expressions by the Java Regex Tester Tool.

Java Regex API provides 1 interface and 3 classes in **java.util.regex** package.

java.util.regex package

The Matcher and Pattern classes provide the facility of Java regular expression. The java.util.regex package provides following classes and interfaces for regular expressions.

1. MatchResult interface
2. Matcher class
3. Pattern class
4. PatternSyntaxException class



Matcher class

It implements the **MatchResult** interface. It is a *regex engine* which is used to perform match operations on a character sequence.

No.	Method	Description
1	boolean matches()	test whether the regular expression matches the pattern.
2	boolean find()	finds the next expression that matches the pattern.

3	boolean find(int start)	finds the next expression that matches the pattern from the given start number.
4	String group()	returns the matched subsequence.
5	int start()	returns the starting index of the matched subsequence.
6	int end()	returns the ending index of the matched subsequence.
7	int groupCount()	returns the total number of the matched subsequence.

Pattern class

It is the *compiled version of a regular expression*. It is used to define a pattern for the regex engine.

No.	Method	Description
1	static Pattern compile(String regex)	compiles the given regex and returns the instance of the Pattern.
2	Matcher matcher(CharSequence input)	creates a matcher that matches the given input with the pattern.
3	static boolean matches(String regex, CharSequence input)	It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern.

4	String[] split(CharSequence input)	splits the given input string around matches of given pattern.
5	String pattern()	returns the regex pattern.

Example of Java Regular Expressions

There are three ways to write the regex example in Java.

```
1. import java.util.regex.*;
2. public class RegexExample1{
3. public static void main(String args[]){
4. //1st way
5. Pattern p = Pattern.compile(".s");//. represents single character
6. Matcher m = p.matcher("as");
7. boolean b = m.matches();
8.
9. //2nd way
10. boolean b2=Pattern.compile(".s").matcher("as").matches();
11.
12. //3rd way
13. boolean b3 = Pattern.matches(".s", "as");
14.
15. System.out.println(b+" "+b2+" "+b3);
16. }}
```

Test it Now

Output

true true true

Regular Expression . Example

The . (dot) represents a single character.

```
1. import java.util.regex.*;
2. class RegexExample2{
3. public static void main(String args[]){
4. System.out.println(Pattern.matches(".s", "as"));//true (2nd char is s)
5. System.out.println(Pattern.matches(".s", "mk"));//false (2nd char is not s)
6. System.out.println(Pattern.matches(".s", "mst"));//false (has more than 2 char)
7. System.out.println(Pattern.matches(".s", "amms"));//false (has more than 2
   char)
8. System.out.println(Pattern.matches("..s", "mas"));//true (3rd char is s)
9. }}
```

Test it Now

Regex Character classes

No.	Character Class	Description
1	[abc]	a, b, or c (simple class)
2	[^abc]	Any character except a, b, or c (negation)
3	[a-zA-Z]	a through z or A through Z, inclusive (range)
4	[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
5	[a-z&&[def]]	d, e, or f (intersection)

6	[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
7	[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)

Regular Expression Character classes Example

```
1. import java.util.regex.*;
2. class RegexExample3{
3. public static void main(String args[]){
4. System.out.println(Pattern.matches("[amn]", "abcd"));//false (not a or m or n)
5. System.out.println(Pattern.matches("[amn]", "a"));//true (among a or m or n)
6. System.out.println(Pattern.matches("[amn]", "ammmna"));//false (m and a
   comes more than once)
7. }}
```

Test it Now

Regex Quantifiers

The quantifiers specify the number of occurrences of a character.

Regex	Description
X?	X occurs once or not at all
X+	X occurs once or more times
X*	X occurs zero or more times
X{n}	X occurs n times only

X{n,}	X occurs n or more times
X{y,z}	X occurs at least y times but less than z times

Regular Expression Character classes and Quantifiers Example

```
1. import java.util.regex.*;
2. class RegexExample4{
3. public static void main(String args[]){
4. System.out.println("? quantifier ....");
5. System.out.println(Pattern.matches("[amn]?", "a"));//true (a or m or n comes one time)
6. System.out.println(Pattern.matches("[amn]?", "aaa"));//false (a comes more than one time)
7. System.out.println(Pattern.matches("[amn]?", "aammmnn"));//false (a m and n comes more than one time)
8. System.out.println(Pattern.matches("[amn]?", "aazzta"));//false (a comes more than one time)
9. System.out.println(Pattern.matches("[amn]?", "am"));//false (a or m or n must come one time)
10.
11. System.out.println("+ quantifier ....");
12. System.out.println(Pattern.matches("[amn]+", "a"));//true (a or m or n once or more times)
13. System.out.println(Pattern.matches("[amn]+", "aaa"));//true (a comes more than one time)
```

```
14. System.out.println(Pattern.matches("[amn]+", "aammmnn"));//true (a or m or
    n comes more than once)
15. System.out.println(Pattern.matches("[amn]+", "aazzta"));//false (z and t are not
    matching pattern)
16.
17. System.out.println("* quantifier ....");
18. System.out.println(Pattern.matches("[amn]*", "ammmna"));//true (a or m or n
    may come zero or more times)
19.
20. }}
```

Test it Now

Regex Metacharacters

The regular expression metacharacters work as shortcodes.

Regex	Description
.	Any character (may or may not match terminator)
\d	Any digits, short of [0-9]
\D	Any non-digit, short for [^0-9]
\s	Any whitespace character, short for [\t\n\x0B\f\r]
\S	Any non-whitespace character, short for [^\s]
\w	Any word character, short for [a-zA-Z_0-9]
\W	Any non-word character, short for [^\w]

\b	A word boundary
\B	A non word boundary

Regular Expression Metacharacters Example

```
1. import java.util.regex.*;
2. class RegexExample5{
3. public static void main(String args[]){
4. System.out.println("metacharacters d....");\\d means digit
5.
6. System.out.println(Pattern.matches("\\d", "abc"));//false (non-digit)
7. System.out.println(Pattern.matches("\\d", "1"));//true (digit and comes once)
8. System.out.println(Pattern.matches("\\d", "4443"));//false (digit but comes
   more than once)
9. System.out.println(Pattern.matches("\\d", "323abc"));//false (digit and char)
10.
11. System.out.println("metacharacters D....");\\D means non-digit
12.
13. System.out.println(Pattern.matches("\\D", "abc"));//false (non-digit but comes
   more than once)
14. System.out.println(Pattern.matches("\\D", "1"));//false (digit)
15. System.out.println(Pattern.matches("\\D", "4443"));//false (digit)
16. System.out.println(Pattern.matches("\\D", "323abc"));//false (digit and char)
17. System.out.println(Pattern.matches("\\D", "m"));//true (non-digit and comes
   once)
18.
19. System.out.println("metacharacters D with quantifier....");
```

```
20. System.out.println(Pattern.matches("\\D*", "mak"));//true (non-digit and may
    come 0 or more times)
21.
22. }}
```

Test it Now

Regular Expression Question 1

```
1. /*Create a regular expression that accepts alphanumeric characters only.
2. Its length must be six characters long only.*/
3.
4. import java.util.regex.*;
5. class RegexExample6{
6. public static void main(String args[]){
7. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun32"));//true
8. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "kkvarun32"));//false
   (more than 6 char)
9. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "JA2Uk2"));//true
10. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun$2"));//false ($ is
    not matched)
11. }}
```

Test it Now

Regular Expression Question 2

```
1. /*Create a regular expression that accepts 10 digit numeric characters
2.  starting with 7, 8 or 9 only.*/
3.
```

```
4. import java.util.regex.*;
5. class RegexExample7{
6. public static void main(String args[]){
7. System.out.println("by character classes and quantifiers ...");
8. System.out.println(Pattern.matches("[789]{1}[0-9]{9}", "9953038949"));//true
9. System.out.println(Pattern.matches("[789][0-9]{9}", "9953038949"));//true
10.
11. System.out.println(Pattern.matches("[789][0-9]{9}", "99530389490"));//false
    (11 characters)
12. System.out.println(Pattern.matches("[789][0-9]{9}", "6953038949"));//false
    (starts from 6)
13. System.out.println(Pattern.matches("[789][0-9]{9}", "8853038949"));//true
14.
15. System.out.println("by metacharacters ...");
16. System.out.println(Pattern.matches("[789]{1}\\d{9}", "8853038949"));//true
17. System.out.println(Pattern.matches("[789]{1}\\d{9}", "3853038949"));//false
    (starts from 3)
18.
19. }}
```

Test it Now

Java Regex Finder Example

```
1. import java.util.regex.Pattern;
2. import java.util.Scanner;
3. import java.util.regex.Matcher;
4. public class RegexExample8{
5.     public static void main(String[] args){
6.         Scanner sc=new Scanner(System.in);
```

```
7.          while (true) {
8.              System.out.println("Enter regex pattern:");
9.              Pattern pattern = Pattern.compile(sc.nextLine());
10.             System.out.println("Enter text:");
11.             Matcher matcher = pattern.matcher(sc.nextLine());
12.             boolean found = false;
13.             while (matcher.find()) {
14.                 System.out.println("I found the text "+matcher.group()+" starting at
    index "+
15.                  matcher.start()+" and ending at index "+matcher.end());
16.                 found = true;
17.             }
18.             if(!found){
19.                 System.out.println("No match found.");
20.             }
21.         }
22.     }
23.}
```

Output:

```
Enter regex pattern: java
Enter text: this is java, do you know java
I found the text java starting at index 8 and ending at index 12
I found the text java starting at index 26 and ending at index 30
```

Exception Handling in Java

1. Exception Handling
2. Advantage of Exception Handling
3. Hierarchy of Exception classes
4. Types of Exception
5. Exception Example

6. Scenarios where an exception may occur

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

In this page, we will learn about Java exceptions, its type and the difference between checked and unchecked exceptions.

What is Exception in Java

Dictionary Meaning: Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is Exception Handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

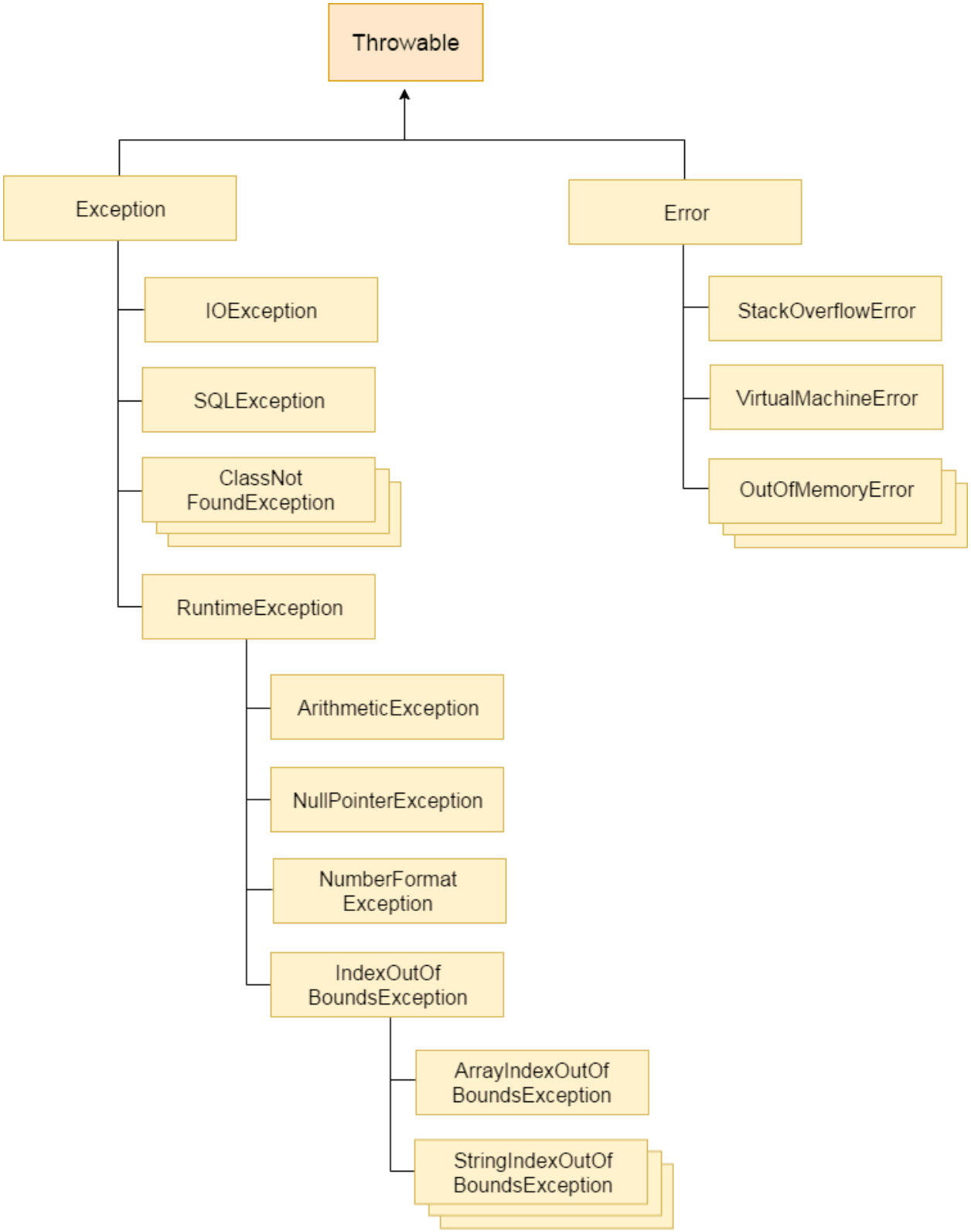
Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

Do You Know?

- What is the difference between checked and unchecked exceptions?
- What happens behind the code int data=50/0;?
- Why use multiple catch block?
- Is there any possibility when finally block is not executed?
- What is exception propagation?
- What is the difference between throw and throws keyword?
- What are the 4 rules for using exception handling with method overriding?

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:


Throwable
Exception
Error
IOException
SQLException
ClassNot FoundException
RuntimeException
StackOverflowError
VirtualMachineError
OutOfMemoryError
ArithmeticException
NullPointerException
NumberFormat Exception
IndexOutOf BoundsException
ArrayIndexOutOf BoundsException
StringIndexOutOf BoundsException

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Key word	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.

throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

```
1. public class JavaExceptionExample{
2.   public static void main(String args[]){
3.    try{
4.       //code that may raise exception
5.       int data=100/0;
6.    }catch(ArithmeticException e){System.out.println(e);}
7.    //rest code of the program
8.    System.out.println("rest of the code...");
9.   }
10.}
```

Test it Now

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
1. int a=50/0;//ArithmeticException
```

---

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
1. String s=null;
2. System.out.println(s.length());//NullPointerException
```

---

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

```
1. String s="abc";
2. int i=Integer.parseInt(s);//NumberFormatException
```

---

4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

```
1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException
```

Java try-catch block

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
1. try{
2. //code that may throw an exception
3. }catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
1. try{
2. //code that may throw an exception
3. }finally{}
```

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (

i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.

Example 1

```
1. public class TryCatchExample1 {
2.
3.     public static void main(String[] args) {
4.
5.         int data=50/0; //may throw exception
6.
7.         System.out.println("rest of the code");
8.
9.     }
10.
11.}
```

Test it Now

Output:

Exception in thread "main" java.lang.ArithmeticException: / by zero

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

Let's see the solution of the above problem by a java try-catch block.

Example 2

```
1.  public class TryCatchExample2 {
2.
3.      public static void main(String[] args) {
4.          try
5.          {
6.          int data=50/0; //may throw exception
7.          }
8.              //handling the exception
9.          catch(ArithmeticException e)
10.         {
11.             System.out.println(e);
12.         }
13.         System.out.println("rest of the code");
14.     }
15.
16.}
```

Test it Now

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Now, as displayed in the above example, the **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

Example 3

In this example, we also kept the code in a try block that will not throw an exception.

```
1.  public class TryCatchExample3 {
2.
3.      public static void main(String[] args) {
4.          try
5.          {
6.          int data=50/0; //may throw exception
7.                          // if exception occurs, the remaining statement will not
    exceute
8.          System.out.println("rest of the code");
9.          }
10.             // handling the exception
11.         catch(ArithmeticException e)
12.         {
13.             System.out.println(e);
14.         }
15.
```

```
16.    }
17.
18.}
```

Test it Now

Output:

```
java.lang.ArithmeticException: / by zero
```

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

Example 4

Here, we handle the exception using the parent class exception.

```
1.  public class TryCatchExample4 {
2.
3.      public static void main(String[] args) {
4.          try
5.          {
6.          int data=50/0; //may throw exception
7.          }
8.              // handling the exception by using Exception class
9.          catch(Exception e)
10.         {
11.             System.out.println(e);
12.         }
13.         System.out.println("rest of the code");
14.    }
```

```
15.
16.}
```

Test it Now

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Example 5

Let's see an example to print a custom message on exception.

```
1.  public class TryCatchExample5 {
2.
3.      public static void main(String[] args) {
4.          try
5.          {
6.          int data=50/0; //may throw exception
7.          }
8.              // handling the exception
9.          catch(Exception e)
10.         {
11.                 // displaying the custom message
12.             System.out.println("Can't divided by zero");
13.         }
14.     }
15.
16.}
```

Test it Now

Output:

Can't divided by zero

Example 6

Let's see an example to resolve the exception in a catch block.

```
1.  public class TryCatchExample6 {
2.
3.      public static void main(String[] args) {
4.          int i=50;
5.          int j=0;
6.          int data;
7.          try
8.          {
9.          data=i/j; //may throw exception
10.         }
11.             // handling the exception
12.         catch(Exception e)
13.         {
14.              // resolving the exception in catch block
15.             System.out.println(i/(j+2));
16.         }
17.     }
18.}
```

Test it Now

Output:

25

Example 7

In this example, along with try block, we also enclose exception code in a catch block.

```
1.  public class TryCatchExample7 {
2.
3.      public static void main(String[] args) {
4.
5.          try
6.          {
7.          int data1=50/0; //may throw exception
8.
9.          }
10.             // handling the exception
11.         catch(Exception e)
12.         {
13.             // generating the exception in catch block
14.         int data2=50/0; //may throw exception
15.
16.         }
17.     System.out.println("rest of the code");
18.     }
19.}
```

Test it Now

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

Example 8

In this example, we handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

```
1.  public class TryCatchExample8 {
2.
3.      public static void main(String[] args) {
4.          try
5.          {
6.          int data=50/0; //may throw exception
7.
8.          }
9.              // try to handle the ArithmeticException using
    ArrayIndexOutOfBoundsException
10.         catch(ArrayIndexOutOfBoundsException e)
11.         {
12.             System.out.println(e);
13.         }
14.         System.out.println("rest of the code");
15.     }
16.
```

```
17.}
```

Test it Now

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Example 9

Let's see an example to handle another unchecked exception.

```
1.  public class TryCatchExample9 {
2.
3.      public static void main(String[] args) {
4.          try
5.          {
6.          int arr[]= {1,3,5,7};
7.          System.out.println(arr[10]); //may throw exception
8.          }
9.              // handling the array exception
10.         catch(ArrayIndexOutOfBoundsException e)
11.         {
12.             System.out.println(e);
13.         }
14.         System.out.println("rest of the code");
15.     }
16.
17.}
```

Test it Now

Output:

```
java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code
```

Example 10

Let's see an example to handle checked exception.

```
1.  import java.io.FileNotFoundException;
2.  import java.io.PrintWriter;
3.
4.  public class TryCatchExample10 {
5.
6.      public static void main(String[] args) {
7.
8.
9.          PrintWriter pw;
10.         try {
11.             pw = new PrintWriter("jtp.txt"); //may throw exception
12.             pw.println("saved");
13.         }
14.// providing the checked exception handler
15. catch (FileNotFoundException e) {
16.
17.             System.out.println(e);
18.         }
19.     System.out.println("File saved successfully");
20.     }
```

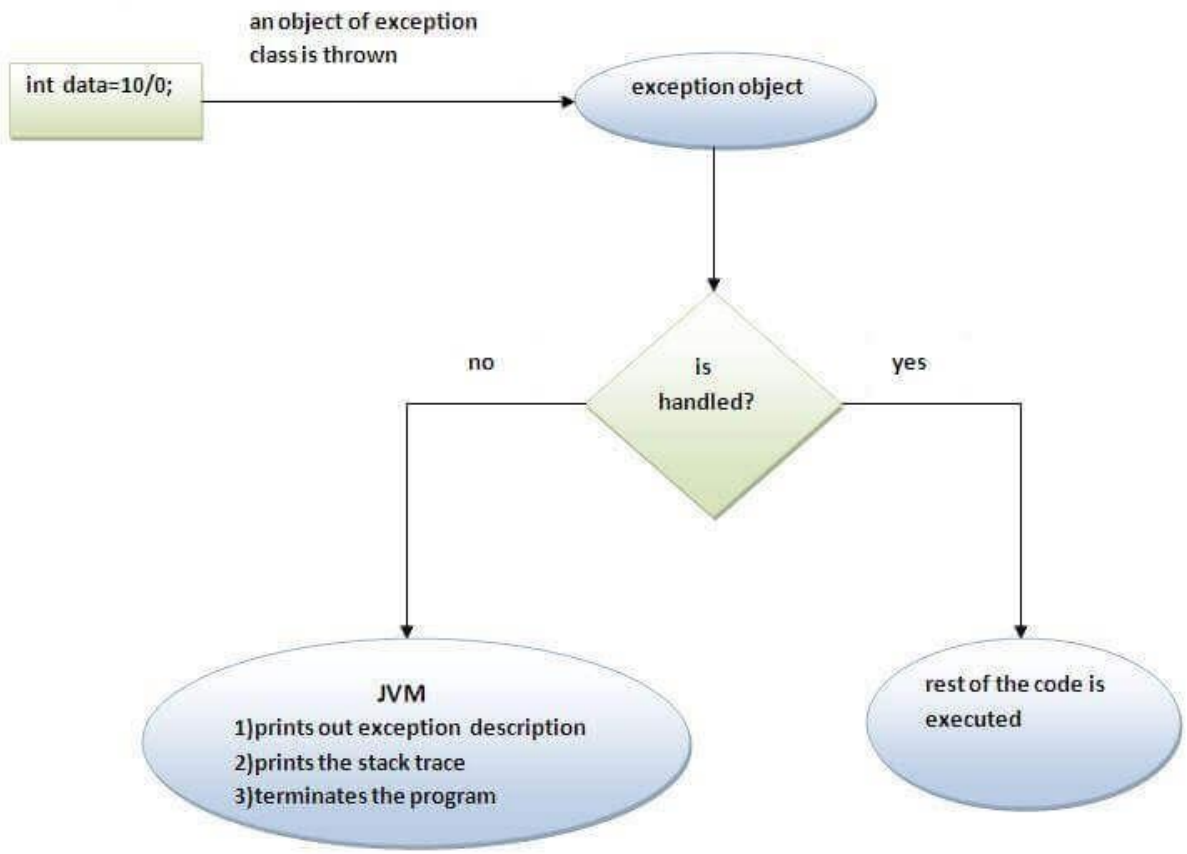
```
21.}
```

Test it Now

Output:

```
File saved successfully
```

Internal working of java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

Java catch multiple exceptions

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Example 1

Let's see a simple example of java multi-catch block.

```
1. public class MultipleCatchBlock1 {
2.
3.     public static void main(String[] args) {
4.
```

```
5.          try{
6.              int a[]=new int[5];
7.              a[5]=30/0;
8.             }
9.             catch(ArithmeticException e)
10.              {
11.               System.out.println("Arithmetic Exception occurs");
12.              }
13.            catch(ArrayIndexOutOfBoundsException e)
14.              {
15.               System.out.println("ArrayIndexOutOfBounds Exception occurs");
16.              }
17.            catch(Exception e)
18.              {
19.               System.out.println("Parent Exception occurs");
20.              }
21.            System.out.println("rest of the code");
22.    }
23.}
```

Test it Now

Output:

```
Arithmetic Exception occurs
rest of the code
```

Example 2

```
1.  public class MultipleCatchBlock2 {
2.
3.      public static void main(String[] args) {
```

```
4.
5.          try{
6.              int a[]=new int[5];
7.
8.              System.out.println(a[10]);
9.             }
10.            catch(ArithmeticException e)
11.              {
12.               System.out.println("Arithmetic Exception occurs");
13.              }
14.            catch(ArrayIndexOutOfBoundsException e)
15.              {
16.               System.out.println("ArrayIndexOutOfBounds Exception occurs");
17.              }
18.            catch(Exception e)
19.              {
20.               System.out.println("Parent Exception occurs");
21.              }
22.            System.out.println("rest of the code");
23.    }
24.}
```

Test it Now

Output:

```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

Example 3

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is invoked.

```
1.  public class MultipleCatchBlock3 {
2.
3.      public static void main(String[] args) {
4.
5.          try{
6.              int a[]=new int[5];
7.              a[5]=30/0;
8.              System.out.println(a[10]);
9.             }
10.            catch(ArithmeticException e)
11.              {
12.               System.out.println("Arithmetic Exception occurs");
13.              }
14.            catch(ArrayIndexOutOfBoundsException e)
15.              {
16.               System.out.println("ArrayIndexOutOfBounds Exception occurs");
17.              }
18.            catch(Exception e)
19.              {
20.               System.out.println("Parent Exception occurs");
21.              }
22.            System.out.println("rest of the code");
23.   }
24.}
```

Test it Now

Output:

```
Arithmetic Exception occurs
rest of the code
```

Example 4

In this example, we generate NullPointerException, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class **Exception** will invoked.

```
1.  public class MultipleCatchBlock4 {
2.
3.      public static void main(String[] args) {
4.
5.          try{
6.              String s=null;
7.              System.out.println(s.length());
8.             }
9.             catch(ArithmeticException e)
10.              {
11.               System.out.println("Arithmetic Exception occurs");
12.              }
13.            catch(ArrayIndexOutOfBoundsException e)
14.              {
15.               System.out.println("ArrayIndexOutOfBounds Exception occurs");
16.              }
17.            catch(Exception e)
18.              {
19.               System.out.println("Parent Exception occurs");
20.              }
21.            System.out.println("rest of the code");
22.   }
```

```
23.}
```

Test it Now

Output:

```
Parent Exception occurs
rest of the code
```

Example 5

Let's see an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

```
1. class MultipleCatchBlock5{
2.  public static void main(String args[]){
3.   try{
4.    int a[]=new int[5];
5.    a[5]=30/0;
6.   }
7.   catch(Exception e){System.out.println("common task completed");}
8.   catch(ArithmeticException e){System.out.println("task1 is completed");}
9.   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2
   completed");}
10.  System.out.println("rest of the code...");
11. }
12.}
```

Test it Now

Output:

```
Compile-time error
```

Java Nested try block

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
....
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
.....
```

Java nested try example

Let's see a simple example of java nested try block.

```
class Excep6{
```

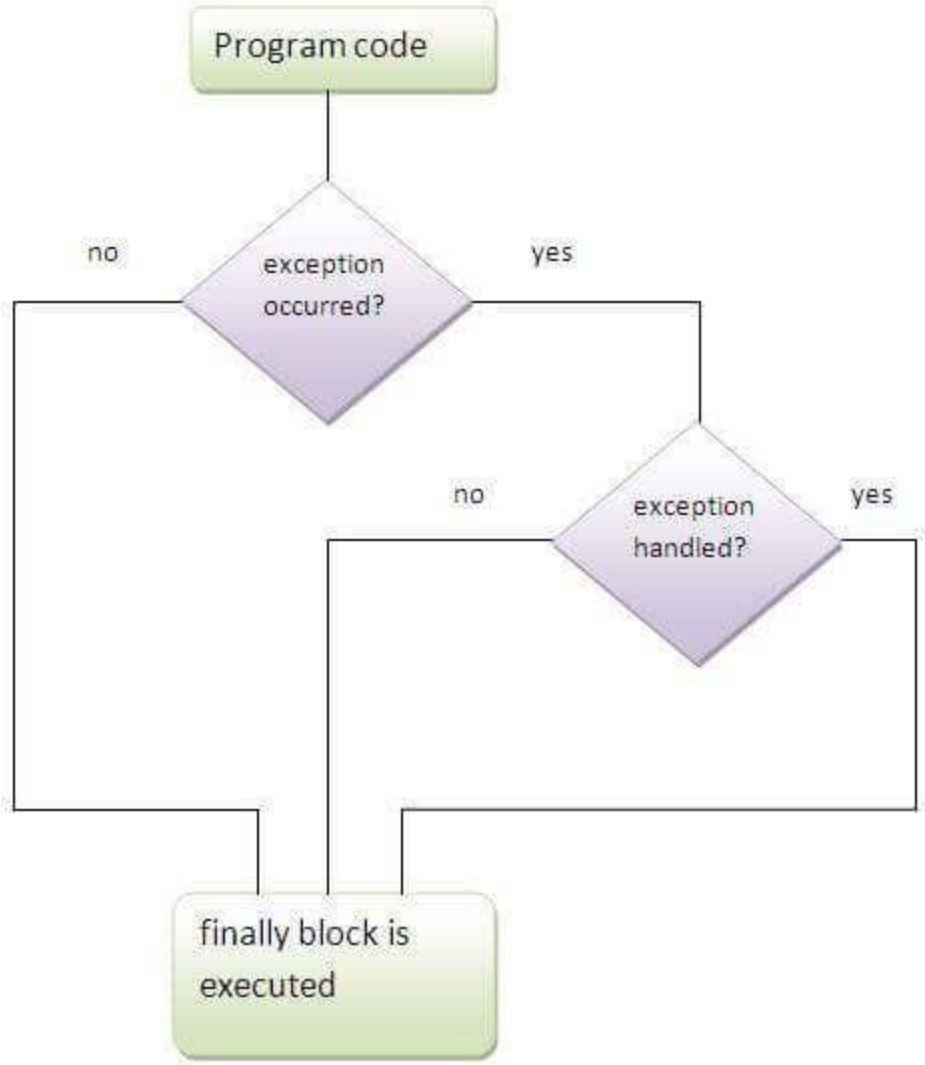
```
2.  public static void main(String args[]){
3.   try{
4.     try{
5.      System.out.println("going to divide");
6.      int b =39/0;
7.     }catch(ArithmeticException e){System.out.println(e);}
8.
9.     try{
10.    int a[]=new int[5];
11.    a[5]=4;
12.    }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
13.
14.    System.out.println("other statement);
15.  }catch(Exception e){System.out.println("handeled");}
16.
17.  System.out.println("normal flow..");
18. }
19.}
20.
```

Java finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.



Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally

Let's see the different cases where java finally block can be used.

Case 1

Let's see the java finally example where **exception doesn't occur**.

```
1.  class TestFinallyBlock{
2.    public static void main(String args[]){
3.    try{
4.     int data=25/5;
5.     System.out.println(data);
6.    }
7.    catch(NullPointerException e){System.out.println(e);}
8.    finally{System.out.println("finally block is always executed");}
9.    System.out.println("rest of the code...");
10.   }
11.}
```

Test it Now

Output:5
finally block is always executed
rest of the code...

Case 2

Let's see the java finally example where **exception occurs and not handled**.

```
1.  class TestFinallyBlock1{
2.    public static void main(String args[]){
3.    try{
4.     int data=25/0;
5.     System.out.println(data);
```

```
6.    }
7.    catch(NullPointerException e){System.out.println(e);}
8.    finally{System.out.println("finally block is always executed");}
9.    System.out.println("rest of the code...");
10.   }
11.}
```

Test it Now

Output:finally block is always executed
Exception in thread main java.lang.ArithmeticException:/ by zero

Case 3

Let's see the java finally example where **exception occurs and handled**.

```
1.  public class TestFinallyBlock2{
2.    public static void main(String args[]){
3.    try{
4.     int data=25/0;
5.     System.out.println(data);
6.    }
7.    catch(ArithmeticException e){System.out.println(e);}
8.    finally{System.out.println("finally block is always executed");}
9.    System.out.println("rest of the code...");
10.   }
11.}
```

Test it Now

Output:Exception in thread main java.lang.ArithmeticException:/ by zero
finally block is always executed
rest of the code...

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

Java throw exception

Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

```
1. throw exception;
```

Let's see the example of throw IOException.

```
1. throw new IOException("sorry device error);
```

java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
1. public class TestThrow1{
2.    static void validate(int age){
3.      if(age<18)
4.       throw new ArithmeticException("not valid");
5.      else
```

```
6.       System.out.println("welcome to vote");
7.    }
8.    public static void main(String args[]){
9.       validate(13);
10.      System.out.println("rest of the code...");
11.   }
12.}
```

Test it Now

Output:

```
Exception in thread main java.lang.ArithmeticException:not valid
```

Java Exception propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.This is called exception propagation.

> Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

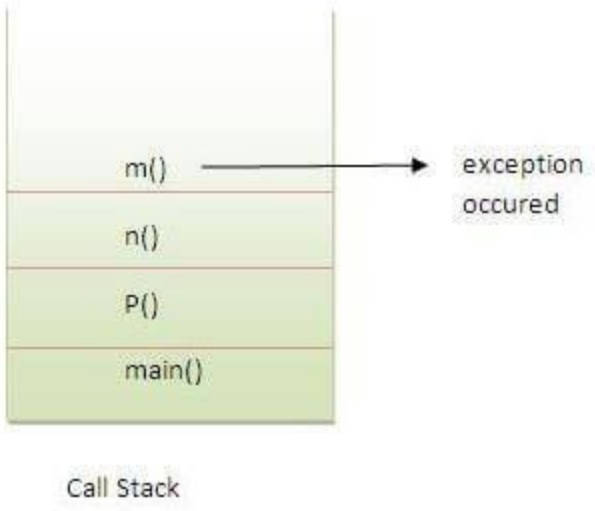
Program of Exception Propagation

```
1.  class TestExceptionPropagation1{
2.    void m(){
3.      int data=50/0;
4.    }
5.    void n(){
6.      m();
7.    }
8.    void p(){
```

```
9.    try{
10.    n();
11.   }catch(Exception e){System.out.println("exception handled");}
12.  }
13.  public static void main(String args[]){
14.   TestExceptionPropagation1 obj=new TestExceptionPropagation1();
15.   obj.p();
16.   System.out.println("normal flow...");
17.  }
18.}
```

Test it Now

Output:exception handled
normal flow...



In the above example exception occurs in m() method where it is not handled,so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

> Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Program which describes that checked exceptions are not propagated

```
1.  class TestExceptionPropagation2{
2.    void m(){
3.      throw new java.io.IOException("device error");//checked exception
4.    }
5.    void n(){
6.      m();
7.    }
8.    void p(){
9.     try{
10.     n();
11.    }catch(Exception e){System.out.println("exception handeled");}
12.  }
13.  public static void main(String args[]){
14.   TestExceptionPropagation2 obj=new TestExceptionPropagation2();
15.   obj.p();
16.   System.out.println("normal flow");
17.  }
18.}
```

Test it Now

Output:Compile Time Error

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

```
1. return_type method_name() throws exception_class_name{
2. //method code
3. }
```

Which exception should be declared

Ans) checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
1. import java.io.IOException;
2. class Testthrows1{
3.   void m()throws IOException{
4.     throw new IOException("device error");//checked exception
```

```
5.    }
6.    void n()throws IOException{
7.      m();
8.    }
9.    void p(){
10.    try{
11.     n();
12.    }catch(Exception e){System.out.println("exception handled");}
13.   }
14.   public static void main(String args[]){
15.    Testthrows1 obj=new Testthrows1();
16.    obj.p();
17.    System.out.println("normal flow...");
18.   }
19.}
```

Test it Now

Output:

```
exception handled
normal flow...
```

> **Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.**

There are two cases:

1. **Case1:**You caught the exception i.e. handle the exception using try/catch.

2. **Case2:**You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
1. import java.io.*;
2. class M{
3.  void method()throws IOException{
4.   throw new IOException("device error");
5.  }
6. }
7. public class Testthrows2{
8.    public static void main(String args[]){
9.     try{
10.     M m=new M();
11.     m.method();
12.    }catch(Exception e){System.out.println("exception handled");}
13.
14.    System.out.println("normal flow...");
15.  }
16.}
```

Test it Now

Output:exception handled
normal flow...

Case2: You declare the exception

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
1. import java.io.*;
2. class M{
3.  void method()throws IOException{
4.   System.out.println("device operation performed");
5.  }
6. }
7. class Testthrows3{
8.    public static void main(String args[])throws IOException{//declare
   exception
9.      M m=new M();
10.     m.method();
11.
12.   System.out.println("normal flow...");
13. }
14.}
```

Test it Now

Output:device operation performed
normal flow...

B)Program if exception occurs

```
1. import java.io.*;
2. class M{
3.  void method()throws IOException{
```

```
4.    throw new IOException("device error");
5.   }
6.  }
7.  class Testthrows4{
8.     public static void main(String args[])throws IOException{//declare
    exception
9.       M m=new M();
10.      m.method();
11.
12.    System.out.println("normal flow...");
13.  }
14.}
```

Test it Now

Output:Runtime Exception

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.

4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Java throw example

```
1. void m(){
2. throw new ArithmeticException("sorry");
3. }
```

Java throws example

```
1. void m()throws ArithmeticException{
2. //method code
3. }
```

Java throw and throws example

```
1. void m()throws ArithmeticException{
2. throw new ArithmeticException("sorry");
3. }
4.
```

Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Java final example

```
1. class FinalExample{
2. public static void main(String[] args){
3. final int x=100;
4. x=200;//Compile Time Error
5. }}
```

Java finally example

```
1. class FinallyExample{
2. public static void main(String[] args){
3. try{
4. int x=300;
5. }catch(Exception e){System.out.println(e);}
6. finally{System.out.println("finally block is executed");}
```

```
7. }}
```

Java finalize example

```
1. class FinalizeExample{
2. public void finalize(){System.out.println("finalize called");}
3. public static void main(String[] args){
4. FinalizeExample f1=new FinalizeExample();
5. FinalizeExample f2=new FinalizeExample();
6. f1=null;
7. f2=null;
8. System.gc();
9. }}
```

ExceptionHandling with MethodOverriding in Java

There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:

- **If the superclass method does not declare an exception**
    - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- **If the superclass method declares an exception**
    - If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

If the superclass method does not declare an exception

1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

```
1. import java.io.*;
2. class Parent{
3.   void msg(){System.out.println("parent");}
4. }
5.
6. class TestExceptionChild extends Parent{
7.   void msg()throws IOException{
8.     System.out.println("TestExceptionChild");
9.   }
10.  public static void main(String args[]){
11.   Parent p=new TestExceptionChild();
12.   p.msg();
13.  }
14.}
```

Test it Now

Output:Compile Time Error

---

2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

```
1. import java.io.*;
2. class Parent{
3.   void msg(){System.out.println("parent");}
```

```
4.  }
5.
6.  class TestExceptionChild1 extends Parent{
7.    void msg()throws ArithmeticException{
8.      System.out.println("child");
9.    }
10.  public static void main(String args[]){
11.   Parent p=new TestExceptionChild1();
12.   p.msg();
13.  }
14.}
```

Test it Now

Output:child

If the superclass method declares an exception

1) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

Example in case subclass overridden method declares parent exception

```
1.  import java.io.*;
2.  class Parent{
3.    void msg()throws ArithmeticException{System.out.println("parent");}
4.  }
5.
6.  class TestExceptionChild2 extends Parent{
```

```
7.    void msg()throws Exception{System.out.println("child");}
8.
9.    public static void main(String args[]){
10.    Parent p=new TestExceptionChild2();
11.    try{
12.    p.msg();
13.    }catch(Exception e){}
14.   }
15.}
```

Test it Now

Output:Compile Time Error

Example in case subclass overridden method declares same exception

```
1.  import java.io.*;
2.  class Parent{
3.    void msg()throws Exception{System.out.println("parent");}
4.  }
5.
6.  class TestExceptionChild3 extends Parent{
7.    void msg()throws Exception{System.out.println("child");}
8.
9.    public static void main(String args[]){
10.    Parent p=new TestExceptionChild3();
11.    try{
12.    p.msg();
13.    }catch(Exception e){}
```

```
14.  }
15.}
```

Test it Now

Output:child

Example in case subclass overridden method declares subclass exception

```
1.  import java.io.*;
2.  class Parent{
3.    void msg()throws Exception{System.out.println("parent");}
4.  }
5.
6.  class TestExceptionChild4 extends Parent{
7.    void msg()throws ArithmeticException{System.out.println("child");}
8.
9.    public static void main(String args[]){
10.   Parent p=new TestExceptionChild4();
11.   try{
12.   p.msg();
13.   }catch(Exception e){}
14.  }
15.}
```

Test it Now

Output:child

Example in case subclass overridden method declares no exception

```
1. import java.io.*;
2. class Parent{
3.   void msg()throws Exception{System.out.println("parent");}
4. }
5.
6. class TestExceptionChild5 extends Parent{
7.   void msg(){System.out.println("child");}
8.
9.   public static void main(String args[]){
10.  Parent p=new TestExceptionChild5();
11.  try{
12.  p.msg();
13.  }catch(Exception e){}
14. }
15.}
```

Test it Now

Output:child

Java Custom Exception

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

Let's see a simple example of java custom exception.

```
1. class InvalidAgeException extends Exception{
2.  InvalidAgeException(String s){
```

```
3.    super(s);
4.   }
5.  }
1.  class TestCustomException1{
2.
3.     static void validate(int age)throws InvalidAgeException{
4.       if(age<18)
5.        throw new InvalidAgeException("not valid");
6.       else
7.        System.out.println("welcome to vote");
8.     }
9.
10.    public static void main(String args[]){
11.      try{
12.      validate(13);
13.      }catch(Exception m){System.out.println("Exception occured: "+m);}
14.
15.      System.out.println("rest of the code...");
16.  }
17.}
```

Test it Now

Output:Exception occured: InvalidAgeException:not valid

```
    rest of the code...
```

Java Inner Classes

1. Java Inner classes
2. Advantage of Inner class
3. Difference between nested class and inner class
4. Types of Nested classes

Java inner class or nested class is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

Syntax of Inner class

```
1. class Java_Outer_class{
2.  //code
3.  class Java_Inner_class{
4.   //code
5.  }
6. }
```

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization**: It requires less code to write.

Do You Know

- What is the internal code generated by the compiler for member inner class ?
- What are the two ways to create annonymous inner class ?
- Can we access the non-final local variable inside the local inner class ?
- How to access the static nested class ?
- Can we define an interface within the class ?

- Can we define a class within the interface ?

Difference between nested class and inner class in Java

Inner class is a part of nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
    1. Member inner class
    2. Anonymous inner class
    3. Local inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.

Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

Java Member inner class

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

```
1. class Outer{
2.  //code
3.  class Inner{
4.   //code
5.  }
6. }
```

Java Member inner class example

In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

```
1. class TestMemberOuter1{
2.  private int data=30;
3.  class Inner{
4.   void msg(){System.out.println("data is "+data);}
5.  }
6.  public static void main(String args[]){
7.   TestMemberOuter1 obj=new TestMemberOuter1();
8.   TestMemberOuter1.Inner in=obj.new Inner();
```

```
9.    in.msg();
10. }
11.}
```

Test it Now

Output:

```
data is 30
```

Internal working of Java member inner class

The java compiler creates two class files in case of inner class. The class file name of inner class is "Outer$Inner". If you want to instantiate inner class, you must have to create the instance of outer class. In such case, instance of inner class is created inside the instance of outer class.

Internal code generated by the compiler

The java compiler creates a class file named Outer$Inner in this case. The Member inner class have the reference of Outer class that is why it can access all the data members of Outer class including private.

```
1.  import java.io.PrintStream;
2.  class Outer$Inner
3.  {
4.     final Outer this$0;
5.     Outer$Inner()
6.     {   super();
7.        this$0 = Outer.this;
8.     }
9.     void msg()
10.    {
11.       System.out.println((new StringBuilder()).append("data is ")
```

```
12.                 .append(Outer.access$000(Outer.this)).toString());
13.    }
14.}
```

Java Anonymous inner class

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

Java anonymous inner class example using class

```
1. abstract class Person{
2.   abstract void eat();
3. }
4. class TestAnonymousInner{
5.  public static void main(String args[]){
6.   Person p=new Person(){
7.   void eat(){System.out.println("nice fruits");}
8.   };
9.   p.eat();
10. }
11.}
```

Test it Now

Output:

nice fruits

Internal working of given code

```
1. Person p=new Person(){
2. void eat(){System.out.println("nice fruits");}
3. };
```

1. A class is created but its name is decided by the compiler which extends the Person class and provides the implementation of the eat() method.
2. An object of Anonymous class is created that is referred by p reference variable of Person type.

Internal class generated by the compiler

```
1. import java.io.PrintStream;
2. static class TestAnonymousInner$1 extends Person
3. {
4.    TestAnonymousInner$1(){}
5.    void eat()
6.    {
7.        System.out.println("nice fruits");
8.    }
9. }
```

Java anonymous inner class example using interface

```
1. interface Eatable{
2.  void eat();
3. }
4. class TestAnnonymousInner1{
5.  public static void main(String args[]){
6.  Eatable e=new Eatable(){
```

```
7.    public void eat(){System.out.println("nice fruits");}
8.   };
9.   e.eat();
10. }
11.}
```

Test it Now

Output:

nice fruits

Internal working of given code

It performs two main tasks behind this code:

```
1. Eatable p=new Eatable(){
2. void eat(){System.out.println("nice fruits");}
3. };
```

1. A class is created but its name is decided by the compiler which implements the Eatable interface and provides the implementation of the eat() method.
2. An object of Anonymous class is created that is referred by p reference variable of Eatable type.

Internal class generated by the compiler

```
1. import java.io.PrintStream;
2. static class TestAnonymousInner1$1 implements Eatable
3. {
4. TestAnonymousInner1$1(){}
5. void eat(){System.out.println("nice fruits");}
6. }
7.
```

Java Local inner class

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Java local inner class example

```
1. public class localInner1{
2.  private int data=30;//instance variable
3.  void display(){
4.   class Local{
5.    void msg(){System.out.println(data);}
6.   }
7.   Local l=new Local();
8.   l.msg();
9.  }
10. public static void main(String args[]){
11.  localInner1 obj=new localInner1();
12.  obj.display();
13. }
14.}
```

Test it Now

Output:

```
30
```

Internal class generated by the compiler

In such case, compiler creates a class named Simple$1Local that have the reference of the outer class.

```
1. import java.io.PrintStream;
2. class localInner1$Local
3. {
4.     final localInner1 this$0;
5.     localInner1$Local()
6.     {
7.         super();
8.         this$0 = Simple.this;
9.     }
10.    void msg()
11.    {
12.        System.out.println(localInner1.access$000(localInner1.this));
13.    }
14.}
```

Rule: Local variable can't be private, public or protected.

Rules for Java Local Inner class

1) Local inner class cannot be invoked from outside the method.

2) Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in local inner class.

Example of local inner class with local variable

```
1.  class localInner2{
2.   private int data=30;//instance variable
3.   void display(){
4.    int value=50;//local variable must be final till jdk 1.7 only
5.    class Local{
6.     void msg(){System.out.println(value);}
7.    }
8.    Local l=new Local();
9.    l.msg();
10. }
11. public static void main(String args[]){
12.  localInner2 obj=new localInner2();
13.  obj.display();
14. }
15.}
```

Test it Now

Output:

```
50
```

Java static nested class

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

Java static nested class example with instance method

```
1. class TestOuter1{
2.   static int data=30;
3.   static class Inner{
4.    void msg(){System.out.println("data is "+data);}
5.   }
6.   public static void main(String args[]){
7.   TestOuter1.Inner obj=new TestOuter1.Inner();
8.   obj.msg();
9.   }
10.}
```

Test it Now

Output:

```
data is 30
```

In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of Outer class because nested class is static and static properties, methods or classes can be accessed without object.

Internal class generated by the compiler

```
1. import java.io.PrintStream;
2. static class TestOuter1$Inner
```

```
3. {
4. TestOuter1$Inner(){}
5. void msg(){
6. System.out.println((new StringBuilder()).append("data is ")
7. .append(TestOuter1.data).toString());
8. }
9. }
```

Java static nested class example with static method

If you have the static member inside static nested class, you don't need to create instance of static nested class.

```
1. class TestOuter2{
2.   static int data=30;
3.   static class Inner{
4.    static void msg(){System.out.println("data is "+data);}
5.   }
6.   public static void main(String args[]){
7.   TestOuter2.Inner.msg();//no need to create the instance of static nested class
8.   }
9. }
```

Test it Now

Output:

```
data is 30
```

ava Nested Interface

An interface i.e. declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitely.

Syntax of nested interface which is declared within the interface

```
1. interface interface_name{
2.  ...
3.  interface nested_interface_name{
4.   ...
5.  }
6. }
```

Syntax of nested interface which is declared within the class

```
1. class class_name{
2.  ...
3.  interface nested_interface_name{
4.   ...
5.  }
6. }
```

Example of nested interface which is declared within the interface

In this example, we are going to learn how to declare the nested interface and how we can access it.

```
1. interface Showable{
2.   void show();
3.   interface Message{
4.    void msg();
5.   }
6. }
7. class TestNestedInterface1 implements Showable.Message{
8.  public void msg(){System.out.println("Hello nested interface");}
9.
10. public static void main(String args[]){
11.  Showable.Message message=new TestNestedInterface1();//upcasting here
12.  message.msg();
13. }
14.}
```

Test it Now

download the example of nested interface

Output:hello nested interface

As you can see in the above example, we are acessing the Message interface by its outer interface Showable because it cannot be accessed directly. It is just like almirah inside the room, we cannot access the almirah directly because we must enter the room first. In collection frameword, sun microsystem has provided a nested interface Entry. Entry is the subinterface of Map i.e. accessed by Map.Entry.

Internal code generated by the java compiler for nested interface Message

The java compiler internally creates public and static interface as displayed below:.

```
1. public static interface Showable$Message
2. {
3.   public abstract void msg();
4. }
```

Example of nested interface which is declared within the class

Let's see how can we define an interface inside the class and how can we access it.

```
1. class A{
2.   interface Message{
3.    void msg();
4.   }
5. }
6.
7. class TestNestedInterface2 implements A.Message{
8.  public void msg(){System.out.println("Hello nested interface");}
9.
10. public static void main(String args[]){
11.  A.Message message=new TestNestedInterface2();//upcasting here
12.  message.msg();
13. }
14.}
```

Test it Now

Output:hello nested interface

Can we define a class inside the interface?

Yes, If we define a class inside the interface, java compiler creates a static nested class. Let's see how can we define a class within the interface:

```
1. interface M{
2.   class A{}
3. }
4.
```

Multithreading in Java

1. Multithreading
2. Multitasking
3. Process-based multitasking
4. Thread-based multitasking
5. What is Thread

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

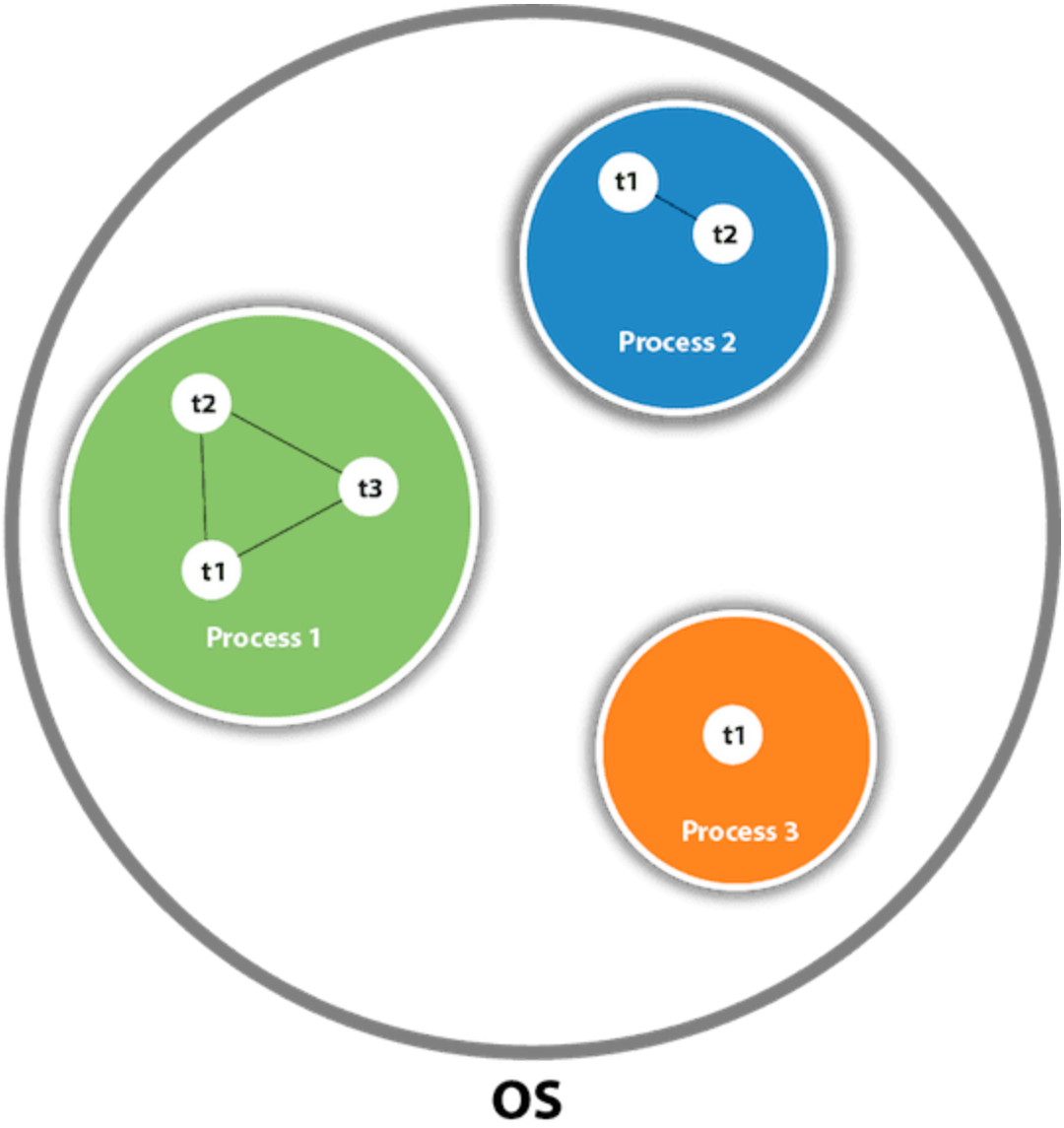
- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

> Note: At least one process is required for each thread.

What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

> Note: At a time one thread is executed only.

Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Java Thread Methods

S. N.	Modifier and Type	Method	Description
1)	void	start()	It is used to start the execution of the thread.
2)	void	run()	It is used to do an action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the currently executing thread object.
5)	void	join()	It waits for a thread to die.
6)	int	getPriority()	It returns the priority of the thread.
7)	void	setPriority()	It changes the priority of the thread.
8)	String	getName()	It returns the name of the thread.
9)	void	setName()	It changes the name of the thread.

10)	long	getId()	It returns the id of the thread.
11)	boolean	isAlive()	It tests if the thread is alive.
12)	static void	yield()	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
13)	void	suspend()	It is used to suspend the thread.
14)	void	resume()	It is used to resume the suspended thread.
15)	void	stop()	It is used to stop the thread.
16)	void	destroy()	It is used to destroy the thread group and all of its subgroups.
17)	boolean	isDaemon()	It tests if the thread is a daemon thread.
18)	void	setDaemon()	It marks the thread as daemon or user thread.
19)	void	interrupt()	It interrupts the thread.

20)	boolean	isinterrupted()	It tests whether the thread has been interrupted.
21)	static boolean	interrupted()	It tests whether the current thread has been interrupted.
22)	static int	activeCount()	It returns the number of active threads in the current thread's thread group.
23)	void	checkAccess()	It determines if the currently running thread has permission to modify the thread.
24)	static boolean	holdLock()	It returns true if and only if the current thread holds the monitor lock on the specified object.
25)	static void	dumpStack()	It is used to print a stack trace of the current thread to the standard error stream.
26)	StackTraceElement[]	getStackTrace()	It returns an array of stack trace elements representing the stack dump of the thread.
27)	static int	enumerate()	It is used to copy every active thread's thread group and its

			subgroup into the specified array.
28)	Thread.State	getState()	It is used to return the state of the thread.
29)	ThreadGroup	getThreadGroup()	It is used to return the thread group to which this thread belongs
30)	String	toString()	It is used to return a string representation of this thread, including the thread's name, priority, and thread group.
31)	void	notify()	It is used to give the notification for only one thread which is waiting for a particular object.
32)	void	notifyAll()	It is used to give the notification to all waiting threads of a particular object.
33)	void	setContextClassLoader()	It sets the context ClassLoader for the Thread.
34)	ClassLoader	getContextClassLoader()	It returns the context ClassLoader for the thread.

35)	static Thread.UncaughtExceptionHandler	getDefaultUncaughtExceptionHandler()	It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
36)	static void	setDefaultUncaughtExceptionHandler()	It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception.

Do You Know

- How to perform two tasks by two threads?
- How to perform multithreading by anonymous class?
- What is the Thread Scheduler and what is the difference between preemptive scheduling and time slicing?
- What happens if we start a thread twice?
- What happens if we call the run() method instead of start() method?
- What is the purpose of join method?
- Why JVM terminates the daemon thread if no user threads are remaining?
- What is the shutdown hook?
- What is garbage collection?
- What is the purpose of finalize() method?
- What does the gc() method?
- What is synchronization and why use synchronization?
- What is the difference between synchronized method and synchronized block?
- What are the two ways to perform static synchronization?

- What is deadlock and when it can occur?
- What is interthread-communication or cooperation?

What will we learn in Multithreading

- Multithreading
- Life Cycle of a Thread
- Two ways to create a Thread
- How to perform multiple tasks by multiple threads
- Thread Scheduler
- Sleeping a thread
- Can we start a thread twice?
- What happens if we call the run() method instead of start() method?
- Joining a thread
- Naming a thread
- Priority of a thread
- Daemon Thread
- ShutdownHook
- Garbage collection
- Synchronization with synchronized method
- Synchronized block
- Static synchronization
- Deadlock
- Inter-thread communication

Life cycle of a Thread (Thread States)

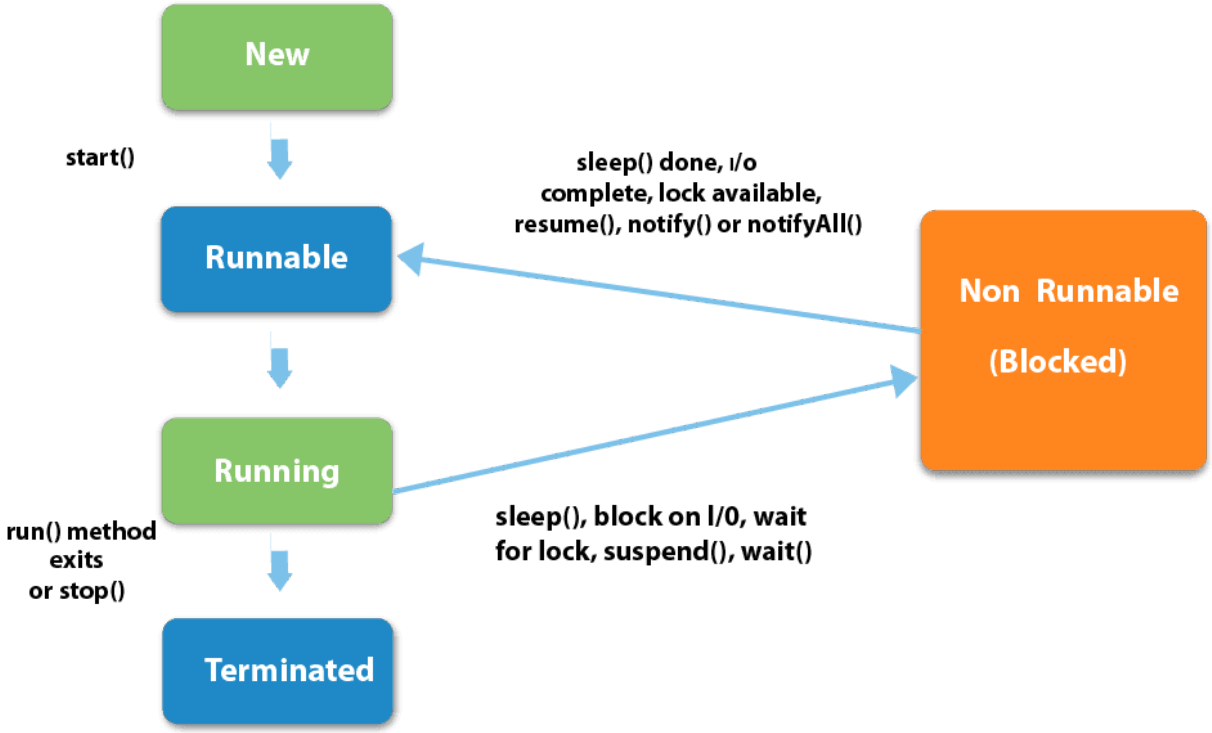
1. Life cycle of a thread
    1. New
    2. Runnable
    3. Running
    4. Non-Runnable (Blocked)
    5. Terminated

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.

20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

---

1) Java Thread Example by extending Thread class

```
1. class Multi extends Thread{
2. public void run(){
3. System.out.println("thread is running...");
```

```
4. }
5. public static void main(String args[]){
6. Multi t1=new Multi();
7. t1.start();
8.  }
9. }
```

Output:thread is running...

---

2) Java Thread Example by implementing Runnable interface

```
1. class Multi3 implements Runnable{
2. public void run(){
3. System.out.println("thread is running...");
4. }
5.
6. public static void main(String args[]){
7. Multi3 m1=new Multi3();
8. Thread t1 =new Thread(m1);
9. t1.start();
10. }
11.}
```

Output:thread is running...

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

Thread Scheduler in Java

Thread scheduler in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Difference between preemptive scheduling and time slicing

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

Sleep method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

- public static void sleep(long miliseconds)throws InterruptedException
- public static void sleep(long miliseconds, int nanos)throws InterruptedException

Example of sleep method in java

```
1. class TestSleepMethod1 extends Thread{
```

```
2.   public void run(){
3.    for(int i=1;i<5;i++){
4.      try{Thread.sleep(500);}catch(InterruptedException
      e){System.out.println(e);}
5.      System.out.println(i);
6.    }
7.   }
8.   public static void main(String args[]){
9.    TestSleepMethod1 t1=new TestSleepMethod1();
10.   TestSleepMethod1 t2=new TestSleepMethod1();
11.
12.   t1.start();
13.   t2.start();
14. }
15.}
```

Output:

```
 1
  1
  2
  2
  3
  3
  4
  4
```

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time,the thread shedular picks up another thread and so on.

Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

```
1. public class TestThreadTwice1 extends Thread{
2.  public void run(){
3.    System.out.println("running...");
4.  }
5.  public static void main(String args[]){
6.   TestThreadTwice1 t1=new TestThreadTwice1();
7.   t1.start();
8.   t1.start();
9.  }
10.}
```

Test it Now

```
running
   Exception in thread "main" java.lang.IllegalThreadStateException
```

What if we call run() method directly instead start() method?

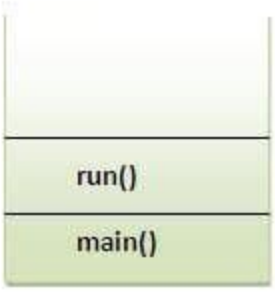
- Each thread starts in a separate call stack.
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

```
1. class TestCallRun1 extends Thread{
2.  public void run(){
3.    System.out.println("running...");
4.  }
```

```
5.   public static void main(String args[]){
6.    TestCallRun1 t1=new TestCallRun1();
7.    t1.run();//fine, but does not start a separate call stack
8.   }
9.  }
```

Test it Now

Output:running...



Stack
(main thread)

Problem if you direct call run() method

```
1.  class TestCallRun2 extends Thread{
2.   public void run(){
3.    for(int i=1;i<5;i++){
4.      try{Thread.sleep(500);}catch(InterruptedException
    e){System.out.println(e);}
5.      System.out.println(i);
6.    }
7.   }
8.   public static void main(String args[]){
9.    TestCallRun2 t1=new TestCallRun2();
10.   TestCallRun2 t2=new TestCallRun2();
11.
```

```
12.  t1.run();
13.  t2.run();
14. }
15.}
```

Test it Now

Output:1
2
3
4
5
1
2
3
4
5

As you can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object.

The join() method

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

Syntax:

```
public void join()throws InterruptedException

public void join(long milliseconds)throws InterruptedException
```

Example of join() method

```
1.  class TestJoinMethod1 extends Thread{
2.   public void run(){
3.    for(int i=1;i<=5;i++){
4.     try{
5.      Thread.sleep(500);
```

```
6.     }catch(Exception e){System.out.println(e);}
7.    System.out.println(i);
8.    }
9.   }
10. public static void main(String args[]){
11. TestJoinMethod1 t1=new TestJoinMethod1();
12. TestJoinMethod1 t2=new TestJoinMethod1();
13. TestJoinMethod1 t3=new TestJoinMethod1();
14. t1.start();
15. try{
16.  t1.join();
17. }catch(Exception e){System.out.println(e);}
18.
19. t2.start();
20. t3.start();
21. }
22.}
```

Test it Now

Output:1

```
2
3
4
5
1
1
2
2
3
3
4
4
5
5
```

As you can see in the above example,when t1 completes its task then t2 and t3 starts executing.

Example of join(long miliseconds) method

```
1.  class TestJoinMethod2 extends Thread{
2.   public void run(){
3.    for(int i=1;i<=5;i++){
4.     try{
5.      Thread.sleep(500);
6.     }catch(Exception e){System.out.println(e);}
7.    System.out.println(i);
8.    }
9.   }
10. public static void main(String args[]){
11. TestJoinMethod2 t1=new TestJoinMethod2();
12. TestJoinMethod2 t2=new TestJoinMethod2();
13. TestJoinMethod2 t3=new TestJoinMethod2();
14. t1.start();
15. try{
16.  t1.join(1500);
17. }catch(Exception e){System.out.println(e);}
18.
19. t2.start();
20. t3.start();
21. }
22.}
```

Test it Now

Output:1
2

```
3
1
4
1
2
5
2
3
3
4
4
5
5
```

In the above example,when t1 is completes its task for 1500 miliseconds(3 times) then t2 and t3 starts executing.

---

getName(),setName(String) and getId() method:

public String getName()

public void setName(String name)

public long getId()

```
1.  class TestJoinMethod3 extends Thread{
2.   public void run(){
3.    System.out.println("running...");
4.   }
5.  public static void main(String args[]){
6.   TestJoinMethod3 t1=new TestJoinMethod3();
7.   TestJoinMethod3 t2=new TestJoinMethod3();
8.   System.out.println("Name of t1:"+t1.getName());
9.   System.out.println("Name of t2:"+t2.getName());
```

```
10.  System.out.println("id of t1:"+t1.getId());
11.
12.  t1.start();
13.  t2.start();
14.
15.  t1.setName("Sonoo Jaiswal");
16.  System.out.println("After changing name of t1:"+t1.getName());
17. }
18.}
```

Test it Now

```
Output:Name of t1:Thread-0
       Name of t2:Thread-1
       id of t1:8
       running...
       After changling name of t1:Sonoo Jaiswal
       running...
```

The currentThread() method:

The currentThread() method returns a reference to the currently executing thread object.

Syntax:

```
public static Thread currentThread()
```

Example of currentThread() method

```
1.  class TestJoinMethod4 extends Thread{
2.   public void run(){
3.    System.out.println(Thread.currentThread().getName());
4.   }
```

```
5.  }
6.  public static void main(String args[]){
7.   TestJoinMethod4 t1=new TestJoinMethod4();
8.   TestJoinMethod4 t2=new TestJoinMethod4();
9.
10.  t1.start();
11.  t2.start();
12. }
13.}
```

Test it Now

Output:Thread-0

```
    Thread-1
```

Naming Thread and Current Thread

Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using setName() method. The syntax of setName() and getName() methods are given below:

1. **public String getName():** is used to return the name of a thread.
2. **public void setName(String name):** is used to change the name of a thread.

Example of naming a thread

```
1.  class TestMultiNaming1 extends Thread{
2.   public void run(){
3.    System.out.println("running...");
4.   }
```

```
5.   public static void main(String args[]){
6.    TestMultiNaming1 t1=new TestMultiNaming1();
7.    TestMultiNaming1 t2=new TestMultiNaming1();
8.    System.out.println("Name of t1:"+t1.getName());
9.    System.out.println("Name of t2:"+t2.getName());
10.
11.   t1.start();
12.   t2.start();
13.
14.   t1.setName("Sonoo Jaiswal");
15.   System.out.println("After changing name of t1:"+t1.getName());
16. }
17.}
```

Test it Now

Output:Name of t1:Thread-0
Name of t2:Thread-1
id of t1:8
running...
After changeling name of t1:Sonoo Jaiswal
running...

Current Thread

The currentThread() method returns a reference of currently executing thread.

```
1.  public static Thread currentThread()
```

Example of currentThread() method

```
1.  class TestMultiNaming2 extends Thread{
2.   public void run(){
```

```
3.    System.out.println(Thread.currentThread().getName());
4.   }
5.   public static void main(String args[]){
6.    TestMultiNaming2 t1=new TestMultiNaming2();
7.    TestMultiNaming2 t2=new TestMultiNaming2();
8.
9.    t1.start();
10.   t2.start();
11. }
12.}
```

Test it Now

Output:Thread-0

```
    Thread-1
```

Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular 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.

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:

```
1. class TestMultiPriority1 extends Thread{
2.  public void run(){
3.    System.out.println("running thread name
   is:"+Thread.currentThread().getName());
4.    System.out.println("running thread priority
   is:"+Thread.currentThread().getPriority());
5.
6.   }
7.  public static void main(String args[]){
8.   TestMultiPriority1 m1=new TestMultiPriority1();
9.   TestMultiPriority1 m2=new TestMultiPriority1();
10.  m1.setPriority(Thread.MIN_PRIORITY);
11.  m2.setPriority(Thread.MAX_PRIORITY);
12.  m1.start();
13.  m2.start();
14.
15. }
16.}
```

Test it Now

Output:running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1

Daemon Thread in Java

Daemon thread in java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

Points to remember for Daemon Thread in Java

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Why JVM terminates the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

No.	Method	Description
1)	public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.

2)	public boolean isDaemon()	is used to check that current is daemon.

Simple example of Daemon thread in java

File: MyThread.java

```
1. public class TestDaemonThread1 extends Thread{
2.  public void run(){
3.   if(Thread.currentThread().isDaemon()){//checking for daemon thread
4.    System.out.println("daemon thread work");
5.   }
6.   else{
7.   System.out.println("user thread work");
8.  }
9.  }
10. public static void main(String[] args){
11.  TestDaemonThread1 t1=new TestDaemonThread1();//creating thread
12.  TestDaemonThread1 t2=new TestDaemonThread1();
13.  TestDaemonThread1 t3=new TestDaemonThread1();
14.
15.  t1.setDaemon(true);//now t1 is daemon thread
16.
17.  t1.start();//starting threads
18.  t2.start();
19.  t3.start();
20. }
21.}
```

Test it Now

Output

```
daemon thread work
user thread work
user thread work
```

> Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

File: MyThread.java

```
1. class TestDaemonThread2 extends Thread{
2.  public void run(){
3.   System.out.println("Name: "+Thread.currentThread().getName());
4.   System.out.println("Daemon: "+Thread.currentThread().isDaemon());
5.  }
6.
7.  public static void main(String[] args){
8.   TestDaemonThread2 t1=new TestDaemonThread2();
9.   TestDaemonThread2 t2=new TestDaemonThread2();
10.  t1.start();
11.  t1.setDaemon(true);//will throw exception here
12.  t2.start();
13. }
14.}
```

Test it Now

```
Output:exception in thread main: java.lang.IllegalThreadStateException
```

Java Thread Pool

Java Thread pool represents a group of worker threads that are waiting for the job and reuse many times.

In case of thread pool, a group of fixed size threads are created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.

Advantage of Java Thread Pool

Better performance It saves time because there is no need to create new thread.

Real time usage

It is used in Servlet and JSP where container creates a thread pool to process the request.

Example of Java Thread Pool

Let's see a simple example of java thread pool using ExecutorService and Executors.

File: WorkerThread.java

```
1. import java.util.concurrent.ExecutorService;
2. import java.util.concurrent.Executors;
3. class WorkerThread implements Runnable {
4.     private String message;
5.     public WorkerThread(String s){
6.         this.message=s;
7.     }
8.      public void run() {
9.         System.out.println(Thread.currentThread().getName()+" (Start) message
   = "+message);
10.        processmessage();//call processmessage method that sleeps the thread for
   2 seconds
```

```
11.         System.out.println(Thread.currentThread().getName()+" (End)");//prints
    thread name
12.     }
13.     private void processmessage() {
14.         try {  Thread.sleep(2000);  } catch (InterruptedException e) {
    e.printStackTrace(); }
15.     }
16.}
```

File: JavaThreadPoolExample.java

```
1.  public class TestThreadPool {
2.       public static void main(String[] args) {
3.          ExecutorService executor = Executors.newFixedThreadPool(5);//creating a
    pool of 5 threads
4.          for (int i = 0; i < 10; i++) {
5.              Runnable worker = new WorkerThread("" + i);
6.              executor.execute(worker);//calling execute method of ExecutorService
7.            }
8.          executor.shutdown();
9.          while (!executor.isTerminated()) {   }
10.
11.         System.out.println("Finished all threads");
12.     }
13. }
```

download this example

Output:

```
pool-1-thread-1 (Start) message = 0
pool-1-thread-2 (Start) message = 1
pool-1-thread-3 (Start) message = 2
```

```
pool-1-thread-5 (Start) message = 4
pool-1-thread-4 (Start) message = 3
pool-1-thread-2 (End)
pool-1-thread-2 (Start) message = 5
pool-1-thread-1 (End)
pool-1-thread-1 (Start) message = 6
pool-1-thread-3 (End)
pool-1-thread-3 (Start) message = 7
pool-1-thread-4 (End)
pool-1-thread-4 (Start) message = 8
pool-1-thread-5 (End)
pool-1-thread-5 (Start) message = 9
pool-1-thread-2 (End)
pool-1-thread-1 (End)
pool-1-thread-4 (End)
pool-1-thread-3 (End)
pool-1-thread-5 (End)
Finished all threads
```

ThreadGroup in Java

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Note: Now suspend(), resume() and stop() methods are deprecated.

Java thread group is implemented by *java.lang.ThreadGroup* class.

A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.

A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

No.	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with given name.

2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with given parent group and name.

Methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of ThreadGroup methods are given below.

S.N.	Modifier and Type	Method	Description
1)	void	checkAccess()	This method determines if the currently running thread has permission to modify the thread group.

2)	int	activeCount()	This method returns an estimate of the number of active threads in the thread group and its subgroups.
3)	int	activeGroupCount()	This method returns an estimate of the number of active groups in the thread group and its subgroups.
4)	void	destroy()	This method destroys the thread group and all of its subgroups.
5)	int	enumerate(Thread[] list)	This method copies into the specified array every active thread in the thread group and its subgroups.

6)	int	getMaxPriority()	This method returns the maximum priority of the thread group.
7)	String	getName()	This method returns the name of the thread group.
8)	ThreadGroup	getParent()	This method returns the parent of the thread group.
9)	void	interrupt()	This method interrupts all threads in the thread group.
10)	boolean	isDaemon()	This method tests if the thread group is a daemon thread group.

11)	void	setDaemon(boolean daemon)	This method changes the daemon status of the thread group.
12)	boolean	isDestroyed()	This method tests if this thread group has been destroyed.
13)	void	list()	This method prints information about the thread group to the standard output.
14)	boolean	parentOf(ThreadGroup g	This method tests if the thread group is either the thread group argument or one of its ancestor thread groups.
15)	void	suspend()	This method is used to suspend

			all threads in the thread group.
16)	void	resume()	This method is used to resume all threads in the thread group which was suspended using suspend() method.
17)	void	setMaxPriority(int pri)	This method sets the maximum priority of the group.
18)	void	stop()	This method is used to stop all threads in the thread group.
19)	String	toString()	This method returns a string representation of the Thread group.

Let's see a code to group multiple threads.

```
1. ThreadGroup tg1 = new ThreadGroup("Group A");
2. Thread t1 = new Thread(tg1,new MyRunnable(),"one");
```

```
3. Thread t2 = new Thread(tg1,new MyRunnable(),"two");
4. Thread t3 = new Thread(tg1,new MyRunnable(),"three");
```

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

```
1. Thread.currentThread().getThreadGroup().interrupt();
```

ThreadGroup Example

File: ThreadGroupDemo.java

```
1.  public class ThreadGroupDemo implements Runnable{
2.      public void run() {
3.          System.out.println(Thread.currentThread().getName());
4.      }
5.     public static void main(String[] args) {
6.        ThreadGroupDemo runnable = new ThreadGroupDemo();
7.          ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");
8.
9.          Thread t1 = new Thread(tg1, runnable,"one");
10.         t1.start();
11.         Thread t2 = new Thread(tg1, runnable,"two");
12.         t2.start();
13.         Thread t3 = new Thread(tg1, runnable,"three");
14.         t3.start();
15.
```

```
16.          System.out.println("Thread Group Name: "+tg1.getName());
17.         tg1.list();
18.
19.     }
20.    }
```

Output:

```
one
two
three
Thread Group Name: Parent ThreadGroup
java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]
    Thread[one,5,Parent ThreadGroup]
    Thread[two,5,Parent ThreadGroup]
      Thread[three,5,Parent ThreadGroup]
```

Java Shutdown Hook

The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly. Performing clean resource means closing log file, sending some alerts or something else. So if you want to execute some code before JVM shuts down, use shutdown hook.

When does the JVM shut down?

The JVM shuts down when:

- user presses ctrl+c on the command prompt
- System.exit(int) method is invoked
- user logoff
- user shutdown etc.

The addShutdownHook(Thread hook) method

The addShutdownHook() method of Runtime class is used to register the thread with the Virtual Machine. Syntax:

```
1. public void addShutdownHook(Thread hook){}
```

The object of Runtime class can be obtained by calling the static factory method getRuntime(). For example:

Runtime r = Runtime.getRuntime();

Factory method

The method that returns the instance of a class is known as factory method.

Simple example of Shutdown Hook

```
1. class MyThread extends Thread{
2.     public void run(){
3.         System.out.println("shut down hook task completed..");
4.     }
5. }
6.
7. public class TestShutdown1{
8. public static void main(String[] args)throws Exception {
9.
10. Runtime r=Runtime.getRuntime();
11. r.addShutdownHook(new MyThread());
12.
13. System.out.println("Now main sleeping... press ctrl+c to exit");
14. try{Thread.sleep(3000);}catch (Exception e) {}
15. }
16. }
```

Output:Now main sleeping... press ctrl+c to exit
shut down hook task completed..

Note: The shutdown sequence can be stopped by invoking the halt(int) method of Runtime class.

Same example of Shutdown Hook by annonymous class:

```
1. public class TestShutdown2{
2. public static void main(String[] args)throws Exception {
3.
4. Runtime r=Runtime.getRuntime();
5.
6. r.addShutdownHook(new Thread(){
7. public void run(){
8.     System.out.println("shut down hook task completed..");
9.     }
10.}
11.);
12.
13.System.out.println("Now main sleeping... press ctrl+c to exit");
14.try{Thread.sleep(3000);}catch (Exception e) {}
15.}
16.}
```

Output:Now main sleeping... press ctrl+c to exit
shut down hook task completed..

How to perform single task by multiple threads?

If you have to perform single task by many threads, have only one run() method.For example:

Program of performing single task by multiple threads

```
1. class TestMultitasking1 extends Thread{
2.  public void run(){
3.    System.out.println("task one");
4.  }
5.  public static void main(String args[]){
6.   TestMultitasking1 t1=new TestMultitasking1();
7.   TestMultitasking1 t2=new TestMultitasking1();
8.   TestMultitasking1 t3=new TestMultitasking1();
9.
10.  t1.start();
11.  t2.start();
12.  t3.start();
13. }
14.}
```

Test it Now

Output:task one

task one

task one

Program of performing single task by multiple threads

```
1. class TestMultitasking2 implements Runnable{
2. public void run(){
3. System.out.println("task one");
4. }
5.
6. public static void main(String args[]){
```

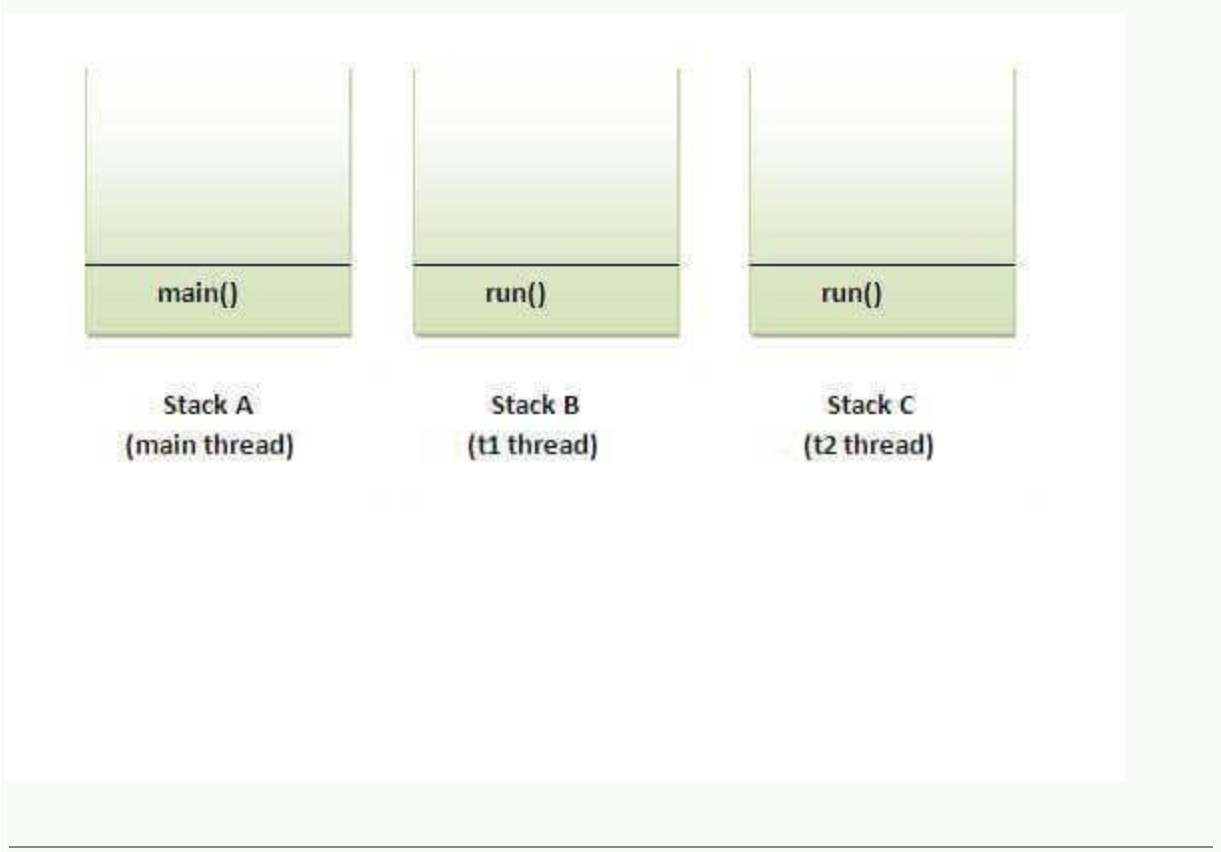
```
7.  Thread t1 =new Thread(new TestMultitasking2());//passing annonymous
    object of TestMultitasking2 class
8.  Thread t2 =new Thread(new TestMultitasking2());
9.
10.t1.start();
11.t2.start();
12.
13. }
14.}
```

Test it Now

Output:task one

task one

Note: Each thread run in a separate callstack.



How to perform multiple tasks by multiple threads (multitasking in multithreading)?

If you have to perform multiple tasks by multiple threads,have multiple run() methods.For example:

Program of performing two tasks by two threads

```
1. class Simple1 extends Thread{
2.  public void run(){
3.    System.out.println("task one");
4.  }
5. }
6.
```

```
7.  class Simple2 extends Thread{
8.   public void run(){
9.    System.out.println("task two");
10. }
11.}
12.
13. class TestMultitasking3{
14. public static void main(String args[]){
15.  Simple1 t1=new Simple1();
16.  Simple2 t2=new Simple2();
17.
18.  t1.start();
19.  t2.start();
20. }
21.}
```

Test it Now

Output:task one

task two

Same example as above by annonymous class that extends Thread class:

Program of performing two tasks by two threads

```
1.  class TestMultitasking4{
2.   public static void main(String args[]){
3.    Thread t1=new Thread(){
4.     public void run(){
```

```
5.        System.out.println("task one");
6.      }
7.    };
8.    Thread t2=new Thread(){
9.      public void run(){
10.       System.out.println("task two");
11.     }
12.   };
13.
14.
15.   t1.start();
16.   t2.start();
17. }
18.}
```

Test it Now

Output:task one

task two

Same example as above by annonymous class that implements Runnable interface:

Program of performing two tasks by two threads

```
1.  class TestMultitasking5{
2.   public static void main(String args[]){
3.    Runnable r1=new Runnable(){
4.      public void run(){
5.        System.out.println("task one");
```

```
6.      }
7.    };
8.
9.    Runnable r2=new Runnable(){
10.     public void run(){
11.       System.out.println("task two");
12.     }
13.   };
14.
15.   Thread t1=new Thread(r1);
16.   Thread t2=new Thread(r2);
17.
18.   t1.start();
19.   t2.start();
20. }
21.}
```

Test it Now

Output:task one

task two

Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

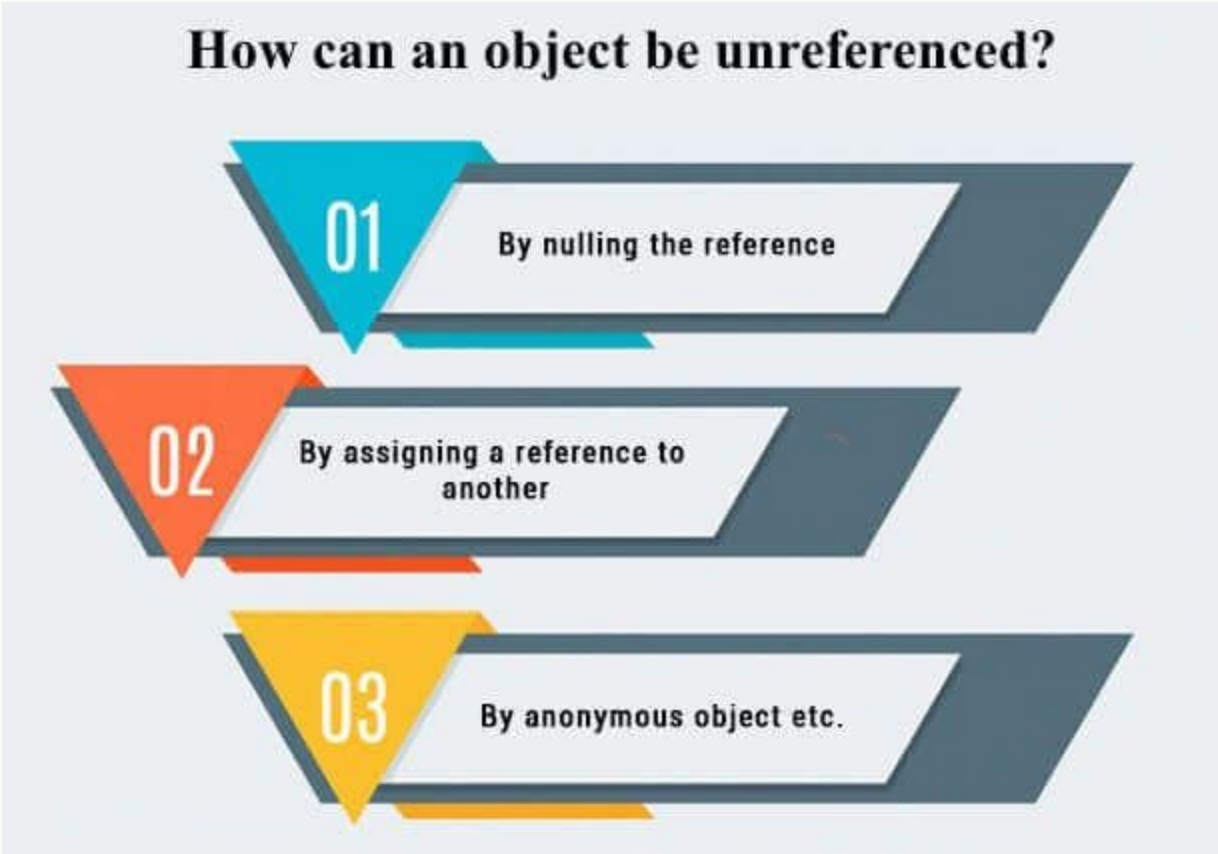
Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.



1) By nulling a reference:

```
1. Employee e=new Employee();
2. e=null;
```

2) By assigning a reference to another:

```
1. Employee e1=new Employee();
2. Employee e2=new Employee();
3. e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By anonymous object:

```
1. new Employee();
```

---

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

```
1. protected void finalize(){}
```

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
1. public static void gc(){}
```

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Simple Example of garbage collection in java

```
1. public class TestGarbage1{
2.  public void finalize(){System.out.println("object is garbage collected");}
3.  public static void main(String args[]){
4.   TestGarbage1 s1=new TestGarbage1();
```

```
5.    TestGarbage1 s2=new TestGarbage1();
6.    s1=null;
7.    s2=null;
8.    System.gc();
9.   }
10.}
```

Test it Now

```
object is garbage collected
object is garbage collected
```

Note: Neither finalization nor garbage collection is guaranteed.

Java Runtime class

Java Runtime class is used *to interact with java runtime environment*. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc. There is only one instance of java.lang.Runtime class is available for one java application.

The **Runtime.getRuntime()** method returns the singleton instance of Runtime class.

Important methods of Java Runtime class

No.	Method	Description
1)	public static Runtime getRuntime()	returns the instance of Runtime class.
2)	public void exit(int status)	terminates the current virtual machine.
3)	public void addShutdownHook(Thread hook)	registers new hook thread.
4)	public Process exec(String command)throws IOException	executes given command in a separate process.
5)	public int availableProcessors()	returns no. of available processors.
6)	public long freeMemory()	returns amount of free memory in JVM.
7)	public long totalMemory()	returns amount of total memory in JVM.

Java Runtime exec() method

```
1. public class Runtime1{
2.  public static void main(String args[])throws Exception{
```

```
3.    Runtime.getRuntime().exec("notepad");//will open a new notepad
4.   }
5.  }
```

How to shutdown system in Java

You can use *shutdown -s* command to shutdown system. For windows OS, you need to provide full path of shutdown command e.g. c:\\Windows\\System32\\shutdown.

Here you can use -s switch to shutdown system, -r switch to restart system and -t switch to specify time delay.

```
1.  public class Runtime2{
2.   public static void main(String args[])throws Exception{
3.    Runtime.getRuntime().exec("shutdown -s -t 0");
4.   }
5.  }
```

How to shutdown windows system in Java

```
1.  public class Runtime2{
2.   public static void main(String args[])throws Exception{
3.    Runtime.getRuntime().exec("c:\\Windows\\System32\\shutdown -s -t 0");
4.   }
5.  }
```

How to restart system in Java

```
1.  public class Runtime3{
2.   public static void main(String args[])throws Exception{
3.    Runtime.getRuntime().exec("shutdown -r -t 0");
```

```
4.  }
5. }
```

Java Runtime availableProcessors()

```
1. public class Runtime4{
2.  public static void main(String args[])throws Exception{
3.   System.out.println(Runtime.getRuntime().availableProcessors());
4.  }
5. }
```

Java Runtime freeMemory() and totalMemory() method

In the given program, after creating 10000 instance, free memory will be less than the previous free memory. But after gc() call, you will get more free memory.

```
1. public class MemoryTest{
2.  public static void main(String args[])throws Exception{
3.   Runtime r=Runtime.getRuntime();
4.   System.out.println("Total Memory: "+r.totalMemory());
5.   System.out.println("Free Memory: "+r.freeMemory());
6.
7.   for(int i=0;i<10000;i++){
8.    new MemoryTest();
9.   }
10.  System.out.println("After creating 10000 instance, Free Memory:
    "+r.freeMemory());
11.  System.gc();
```

```
12.  System.out.println("After gc(), Free Memory: "+r.freeMemory());
13. }
14.}
```

```
Total Memory: 100139008

Free Memory: 99474824

After creating 10000 instance, Free Memory: 99310552

After gc(), Free Memory: 100182832
```