Exception Handling

Definition: An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

✓ The term exception is shorthand for the phrase "exceptional event."

✓ When an error occurs within a method, the method creates an object and hands it off to the run-time system (JVM).

✓ The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred.

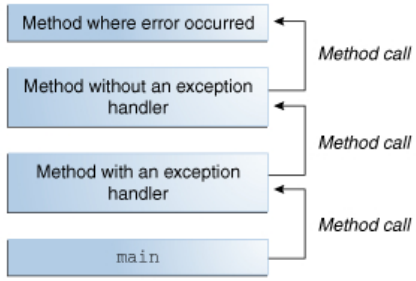
✓ Creating an exception object and handing it to the run-time system is called **throwing an exception**.

✓ After a method throws an exception, the run-time system attempts to **find something to handle it**.

Exception Handling

✓ The set of possible ”somethings” to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred.

✓ The list of methods is known as the **call stack**.



The call stack.

Exception Handling

✓ The run-time system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an **exception handler**.

✓ The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called.

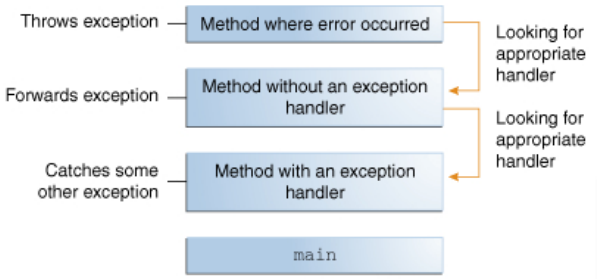
✓ When an appropriate handler is found, the run-time system passes the exception to the handler.

✓ An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

✓ The exception handler chosen is said to **catch the exception**.

✓ **If the run-time system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the run-time system (and, consequently, the program) terminates abnormally.**

Exception Handling



Searching the call stack for the exception handler.

✓ This handler prints the exception information in the following format and terminates program abnormally.

```
1 Exception in thread "xx" Name of Exception : Description
2 ... ..... .. // Call Stack
```

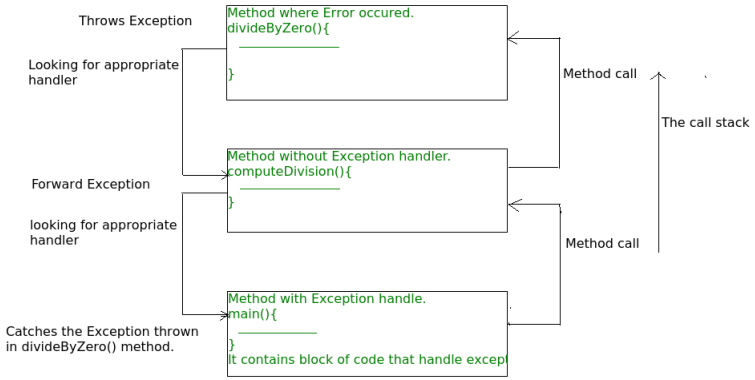
Exception Handling

Example:

```
1  // Java program to demonstrate how exception is thrown.
2  public class ThrowsExecp
3  {
4          public static void main(String args[])
5          {
6                  String str = null;
7                  System.out.println(str.length());
8          }
9  }
10 //Output : Exception in thread "main"
11 java.lang.NullPointerException at
12 ThrowsExecp.main(ThrowsExecp.java:6)
```

Exception Handling

The below diagram to understand the flow of the call stack.



The call stack and searching the call stack for exception handler.

Exception Handling

✓ An example that illustrate how run-time system searches appropriate exception handling code on the call stack :

```
1  //Java program to demonstrate exception is thrown how the runTime system
2  searches th call stack to find appropriate exception handler.
3  class ExceptionThrown
4  {
5      // It throws the Exception(ArithmeticException). Appropriate Exception handler is not found within this
       method.
6      static int divideByZero(int a, int b)
7      {
8          // this statement will cause ArithmeticException(/ by zero)
9          int i = a/b;
10         return i;
11     }
12     //The runTime System searches the appropriate Exception handler in this method also but couldn't have
        found. So looking forward on the call stack.
13     static int computeDivision(int a, int b)
14     {
15         int res =0;
```

```
16          try
17          {
18              res = divideByZero(a,b);
19          }
20          //doesn't matches with ArithmeticException
21          catch(NumberFormatException ex)
22          {
23              System.out.println("NumberFormatException is occured");
24          }
25          return res;
26      }
27      // In this method found appropriate Exception handler. i.e. matching catch block.
28      public static void main(String args[])
29      {
30          int a = 1;
31          int b = 0;
32          try
33          {
34              int i = computeDivision(a,b);
```

```
35              }
36              // matching ArithmeticException
37              catch(ArithmeticException ex)
38              {
39                  // getMessage will print description of exception(here / by zero)
40                  System.out.println(ex.getMessage());
41              }
42      }
43  }
44  Output :
45  / by zero.
```

Advantage of Exception Handling:

✓ The core advantage of exception handling is to maintain the normal flow of the application.

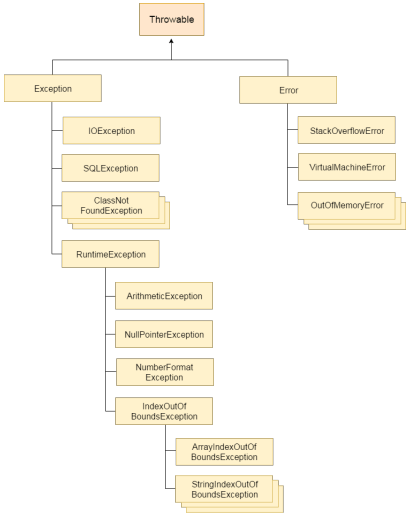
✓ An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
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;
```

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:

Exception Handling



Exception Handling

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:

- Checked Exception
- Unchecked Exception
- Error

Error vs Exception

Error: An Error indicates serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

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.
   ✓ *For example, consider the following Java program that opens file at location "C:\test\a.txt" and prints the first three lines of it. The program doesn't compile, because the function main() uses FileReader() and FileReader() throws a checked exception FileNotFoundException. It also uses readLine() and close() methods, and these methods also throw checked exception IOException*

Exception Handling

```
1  import java.io.*;
2  class Main
3  {
4      public static void main(String[] args)
5      {
6          FileReader file = new FileReader("C:\test\a.txt");
7          BufferedReader fileInput = new BufferedReader(file);
8
9          // Print first 3 lines of file "C:\test\a.txt"
10         for (int counter = 0; counter < 3; counter++)
11         System.out.println(fileInput.readLine());
12
13         fileInput.close();
14     }
15 }
16 Output:
17
18 Exception in thread "main" java.lang.RuntimeException: Uncompilable source code -
19 unreported exception java.io.FileNotFoundException;
20 must be caught or declared to be thrown at Main.main(Main.java:5)
```

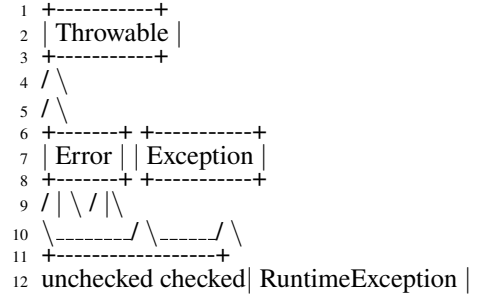
Exception Handling

✓ To fix the above program, we either need to specify list of exceptions using throws, or we need to use try-catch block. Since FileNotFoundException is a subclass of IOException, we can just specify IOException in the throws list and make the above program compiler-error-free.

```
1  import java.io.*;
2  class Main
3  {
4      public static void main(String[] args) throws IOException
5      {
6          FileReader file = new FileReader("C:\test\a.txt");
7          BufferedReader fileInput = new BufferedReader(file);
8          // Print first 3 lines of file "C:\test\a.txt"
9          for (int counter = 0; counter < 3; counter++)
10         System.out.println(fileInput.readLine());
11         fileInput.close();
12     }
13 }
```

Output: First three lines of file "C:\test\a.txt"

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 run-time*.



```
+------------------+
/ || \
\_________________/
unchecked
```

✓ Consider the following Java program. It compiles fine, but it throws ArithmeticException when run. The compiler allows it to compile, because ArithmeticException is an unchecked exception.

```
class Main
{
    public static void main(String args[])
    {
        int x = 0;
        int y = 10;
        int z = y/x;
    }
}
Output:
```

```
11
12 Exception in thread "main" java.lang.ArithmeticException: / by zero at Main.main(Main.java:5)
13 Java Result: 1
```

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

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.

Exception Handling

```
1 // Java program to demonstrate how exception is thrown.
2 class ThrowsExecp
3 {
4       public static void main(String args[])
5       {
6             String str = null;
7             System.out.println(str.length()); //NullPointerException
8       }
9 }
```

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.
   ✓ Since NumberFormatException occurs due to the inappropriate format of string for the corresponding argument of the method which is throwing the exception, there can be various ways of it. A few of them are mentioned as follows-

Exception Handling

- The input string provided might be null-
  Example- Integer.parseInt(null);
- The input string might be empty-
  Example- Integer.parseInt("");
- The input string might be having trailing space-
  Example- Integer.parseInt("123 ");
- The input string might be having a leading space-
  Example- Integer.parseInt(" 123");
- The input string may be alphanumeric-
  Example- Long.parseLong("b2");

Exception Handling

- The input string may have an input which might exceed the range of the datatype storing the parsed string-
  Example- Integer.parseInt(”135”); The maximum possible value of integer can be 127, but the value in the string is 135 which is out of range, so this will throw the exception.
- There may be a mismatch between the input string and the type of the method which is being used for parsing. If you provide the input string like ”1.0” and you try to convert this string into an integer value, it will throw a NumberFormatException exception.
  Example- Integer.parseInt(”1..0”);

```
1 String s=”abc”;
2 int i=Integer.parseInt(s);
```

Exception Handling

```
1 public class Example
2 {
3       public static void main(String[] args)
4       {
5             int a = Integer.parseInt(null); //throws Exception as //the input string is of illegal format for
        parsing as it is null.
6       }
7 }
```

Exception Handling

How Programmer handles an exception?

- Customized Exception Handling : Java exception handling is managed via five keywords: **try, catch, throw, throws, and finally**.
- Program statements that you think can raise exceptions are contained within a **try block**.
- If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword throw.
- Any exception that is thrown out of a method must be specified as such by a throws clause.

Exception Handling

- *Any code that absolutely must be executed after a try block completes is put in a finally block*.

Java Exception Keywords:

Keyword	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.

Need of try-catch clause(Customized Exception Handling):

```
1  // java program to demonstrate
2  // need of try-catch clause
3  class GFG
4  {
5      public static void main (String[] args)
6      {
7          // array of size 4.
8          int[] arr = new int[4];
9          // this statement causes an exception
10         int i = arr[4];
11         // the following statement will never execute
12         System.out.println("Hi, I want to execute");
13     }
14 }
```

How to use try-catch clause

```
try
{
    // block of code to monitor for errors
    // the code you think can raise an exception
}
catch (ExceptionType1 exOb)
{
    // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
    // exception handler for ExceptionType2
}
// optional
finally
{
    // block of code to be executed after try block ends
}
```

Java Exception Handling Example:

```
1  public class JavaExceptionExample
2  {
3      public static void main(String args[])
4      {
5          try
6          {
7              //code that may raise exception
8              int data=100/0;
9          }
10         catch(ArithmeticException e)
11         {
12             System.out.println(e);
13         }
14         //rest code of the program
15         System.out.println("rest of the code...");
16     }
17 }
18 //In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.
```

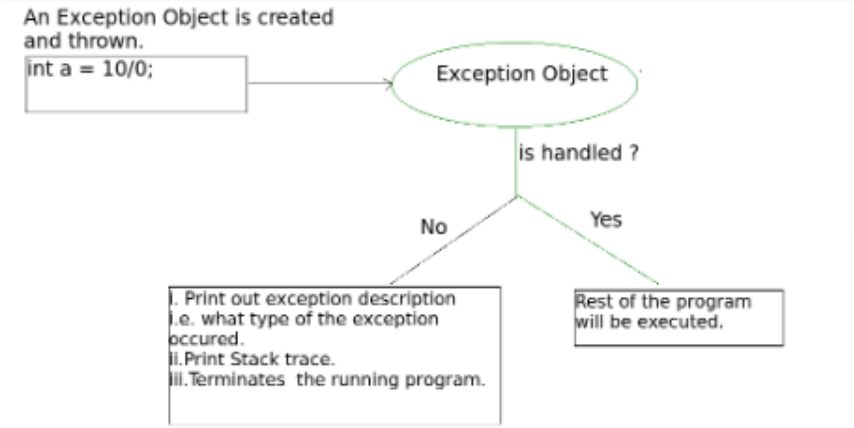
Points to remember :

- In a method, there can be more than one statements that might throw exception, So put all these statements within its own try block and provide separate exception handler within own catch block for each of them.
- If an exception occurs within the try block, that exception is handled by the exception handler associated with it.
- To associate exception handler, we must put catch block after it.
- There can be more than one exception handlers. Each catch block is an exception handler that handles the exception of the type indicated by its argument.
- The argument, ExceptionType declares the type of the exception that it can handle and must be the name of the class that inherits from Throwable class.

Exception Handling

- For each try block there can be zero or more catch blocks, but only one finally block.
- The finally block is optional. It always gets executed whether an exception occurred in try block or not .
- If exception occurs, then it will be executed after try and catch blocks. And if exception does not occur then it will be executed after the try block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

Exception Handling



Example 1:

```
1  public class TryCatchExample1
2  {
3      public static void main(String[] args)
4      {
5          int data=50/0; //may throw exception
6          System.out.println("rest of the code");
7      }
8  }
9
10 // Solution:
11 public class TryCatchExample2
12 {
13     public static void main(String[] args)
14     {
15         try
16         {
17             int data=50/0; //may throw exception
18         }
19         //handling the exception
20         catch(ArithmeticException e)
21         {
22             System.out.println(e);
23         }
24         System.out.println("rest of the code");
25     }
26 }
```

Example 2:

✓ 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      {
5          try
6          {
7              int data=50/0; //may throw exception
8              // if exception occurs, the remaining statement will not exceute
9              System.out.println("rest of the code");
10         }
11         // handling the exception
12         catch(ArithmeticException e)
13         {
14             System.out.println(e);
15         }
16     }
```

```
17 }
```

✓ Solution:

```
1  public class TryCatchExample4
2  {
3      public static void main(String[] args)
4      {
5          try
6          {
7              int data=50/0; //may throw exception
8          }
9          // handling the exception by using Exception class
10         catch(Exception e)
11         {
12             System.out.println(e);
13         }
14         System.out.println("rest of the code");
15     }
16 }
```

Example 3: an example to print a custom message on exception.

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

Exception Handling

✓ Solution: an example to resolve the exception in a catch block.

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

```
19 }
```

Example 4: 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          // handling the exception
10         catch(Exception e)
11         {
12             // generating the exception in catch block
13             int data2=50/0; //may throw exception
14         }
15         System.out.println("rest of the code");
16     }
17 }
```

Exception Handling

✓ Solution: 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      {
5          try
6          {
7              int data=50/0; //may throw exception
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 }
```

Example 5: to handle another unchecked exception.

```
1  public class TryCatchExample9
2  {
3      public static void main(String[] args)
4      {
5          try
6          {
7              int arr[]=
8              {
9                  1,3,5,7
10             }
11             ;
12             System.out.println(arr[10]); //may throw exception
13         }
14         // handling the array exception
15         catch(ArrayIndexOutOfBoundsException e)
16         {
17             System.out.println(e);
18         }
```

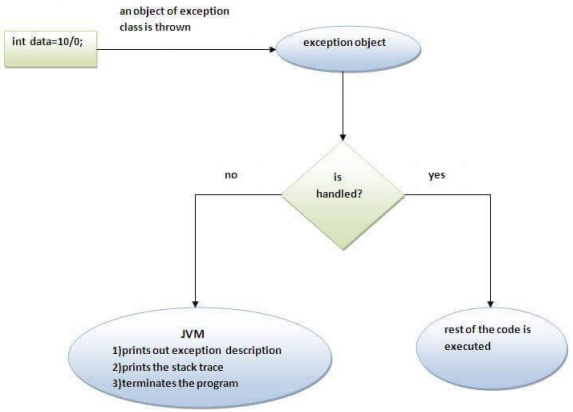
```
19              System.out.println("rest of the code");
20          }
21  }
```

Example 5: To handle checked exception.

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

```
19         }
20 }
```

Internal working of java try-catch block:



Exception Handling

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: java multi-catch block.

```
1  public class MultipleCatchBlock1
2  {
3      public static void main(String[] args)
4      {
5          try
6          {
7              int a[]=new int[5];
8              a[5]=30/0;
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  }
```

Example 2: java multi-catch block.

```
1  public class MultipleCatchBlock2
2  {
3      public static void main(String[] args)
4      {
5          try
6          {
7              int a[]=new int[5];
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  }
```

Example 3: 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          {
7              int a[]=new int[5];
8              a[5]=30/0;
9              System.out.println(a[10]);
10         }
11         catch(ArithmeticException e)
12         {
13             System.out.println("Arithmetic Exception occurs");
14         }
15         catch(ArrayIndexOutOfBoundsException e)
16         {
```

```
17                  System.out.println("ArrayIndexOutOfBounds Exception occurs");
18              }
19              catch(Exception e)
20              {
21                  System.out.println("Parent Exception occurs");
22              }
23              System.out.println("rest of the code");
24          }
25  }
```

Example 4: 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          {
7              String s=null;
8              System.out.println(s.length());
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  }
```

Example 5: to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

```
1  class MultipleCatchBlock5
2  {
3      public static void main(String args[])
4      {
5          try
6          {
7              int a[]=new int[5];
8              a[5]=30/0;
9          }
10         catch(Exception e)
11         {
12             System.out.println("common task completed");
13         }
14         catch(ArithmeticException e)
15         {
16             System.out.println("task1 is completed");
```

```
17              }
18              catch(ArrayIndexOutOfBoundsException e)
19              {
20                    System.out.println("task 2 completed");
21              }
22              System.out.println("rest of the code...");
23        }
24  }
```

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:

Exception Handling

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

Java nested try example:

```
1  class Excep6
2  {
3      public static void main(String args[])
4      {
5          try
6          {
7              try
8              {
9                  System.out.println(”going to divide”);
10                 int b =39/0;
11             }
12             catch(ArithmeticException e)
13             {
14                 System.out.println(e);
15             }
16
17             try
18             {
```

Exception Handling

```
19                  int a[]=new int[5];
20                  a[5]=4;
21              }
22              catch(ArrayIndexOutOfBoundsException e)
23              {
24                  System.out.println(e);
25              }
26
27              System.out.println("other statement");
28          }
29          catch(Exception e)
30          {
31              System.out.println("handeled");
32          }
33
34          System.out.println("normal flow..");
35      }
36  }
```

Java finally block

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

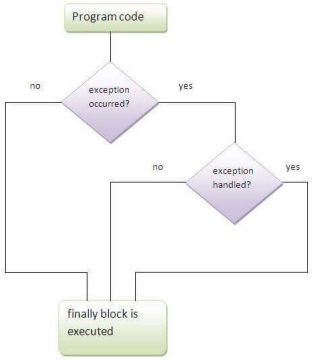
✓ Java finally block is always executed whether exception is handled or not.

✓ Java finally block follows try or catch block.

✓ The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break.

✓ Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

Exception Handling



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.

Example to see the different cases where java finally block can be used.

Case 1: the java finally example where exception doesn’t occur.

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

```
12                  System.out.println(e);
13              }
14              finally
15              {
16                  System.out.println("finally block is always executed");
17              }
18              System.out.println("rest of the code...");
19          }
20  }
```

Case 2: the java finally example where exception occurs and not handled.

Exception Handling

```
1  class TestFinallyBlock1
2  {
3      public static void main(String args[])
4      {
5          try
6          {
7              int data=25/0;
8              System.out.println(data);
9          }
10         catch(NullPointerException e)
11         {
12             System.out.println(e);
13         }
14         finally
15         {
16             System.out.println("finally block is always executed");
17         }
18         System.out.println("rest of the code...");
19     }
```

```
}
```

Case 3: the java finally example where exception occurs and handled.

```
public class TestFinallyBlock2
{
    public static void main(String args[])
    {
        try
        {
            int data=25/0;
            System.out.println(data);
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        finally
        {
            System.out.println("finally block is always executed");
```

Exception Handling

```
17              }
18              System.out.println("rest of the code...");
19          }
20  }
```

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).

Exception Handling

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.

✓ The syntax of java throw keyword is given below.

```
throw exception;

Example: throw new IOException("sorry device error);
```

java throw keyword example:

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

```
1  public class TestThrow1
2  {
3      static void validate(int age)
4      {
5          if(age<18)
6          throw new ArithmeticException("not valid");
7          else
8          System.out.println("welcome to vote");
9      }
10     public static void main(String args[])
11     {
12         validate(13);
13         System.out.println("rest of the code...");
```

```
14          }
15  }
```

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).

Exception Handling

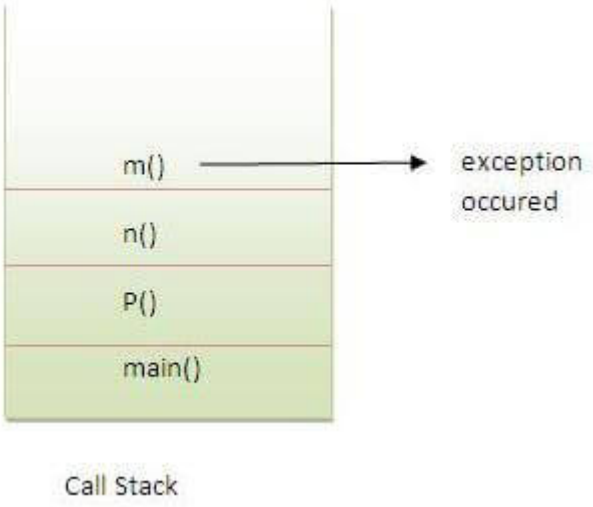
Program of Exception Propagation:

```
1  class TestExceptionPropagation1
2  {
3      void m()
4      {
5          int data=50/0;
6      }
7      void n()
8      {
9          m();
10     }
11     void p()
12     {
13         try
14         {
15             n();
16         }
17         catch(Exception e)
18         {
```

Exception Handling

```
19                  System.out.println(”exception handled”);
20              }
21      }
22      public static void main(String args[])
23      {
24              TestExceptionPropagation1 obj=new TestExceptionPropagation1();
25              obj.p();
26              System.out.println(”normal flow...”);
27      }
28  }
```

Exception Handling



Exception Handling

✓ 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).

Exception Handling

Program which describes that checked exceptions are not propagated:

```
1  class TestExceptionPropagation2
2  {
3      void m()
4      {
5          throw new java.io.IOException("device error"); //checked exception
6      }
7      void n()
8      {
9          m();
10     }
11     void p()
12     {
13         try
14         {
15             n();
16         }
17         catch(Exception e)
18         {
```

```
19                  System.out.println("exception handeled");
20              }
21      }
22      public static void main(String args[])
23      {
24              TestExceptionPropagation2 obj=new TestExceptionPropagation2();
25              obj.p();
26              System.out.println("normal flow");
27      }
28  }
```

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 {
3       //method code
4 }
```

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:

Example 1: throws clause which describes that checked exceptions can be propagated by throws keyword.

```
1  import java.io.IOException;
2  class Testthrows1
3  {
4      void m() throws IOException
5      {
6          throw new IOException("device error"); //checked exception
7      }
8      void n()throws IOException
9      {
10         m();
11     }
12     void p()
13     {
14         try
15         {
```

```
16                  n();
17              }
18              catch(Exception e)
19              {
20                  System.out.println("exception handled");
21              }
22      }
23      public static void main(String args[])
24      {
25          Testthrows1 obj=new Testthrows1();
26          obj.p();
27          System.out.println("normal flow...");
28      }
29  }
```

Exception Handling

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

There are two cases:

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

Exception Handling

```
1  import java.io.*;
2  class M
3  {
4      void method()throws IOException
5      {
6          throw new IOException("device error");
7      }
8  }
9  public class Testthrows2
10 {
11     public static void main(String args[])
12     {
13         try
14         {
15             M m=new M();
16             m.method();
17         }
18         catch(Exception e)
19         {
```

```
20                  System.out.println("exception handled");
21              }
22
23              System.out.println("normal flow...");
24          }
25  }
```

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 run-time because throws does not handle the exception.

Example A: Program if exception does not occur

Exception Handling

```
1  import java.io.*;
2  class M
3  {
4      void method()throws IOException
5      {
6          System.out.println("device operation performed");
7      }
8  }
9  class Testthrows3
10 {
11     public static void main(String args[])throws IOException
12     {
13         //declare exception
14         M m=new M();
15         m.method();
16
17         System.out.println("normal flow...");
18     }
19 }
```

Example B: Program if exception occurs

```
1  import java.io.*;
2  class M
3  {
4      void method()throws IOException
5      {
6          throw new IOException("device error");
7      }
8  }
9  class Testthrows4
10 {
11     public static void main(String args[])throws IOException
12     {
13         //declare exception
14         M m=new M();
15         m.method();
16
17         System.out.println("normal flow...");
18     }
```

```
19 }
```

Exception Handling

Difference between throw and throws in Java

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 {
3       throw new ArithmeticException("sorry");
4 }
```

Java throws example

```
1 void m()throws ArithmeticException
2 {
3       //method code
4 }
```

Java throw and throws example

```
1 void m()throws ArithmeticException
2 {
3       throw new ArithmeticException("sorry");
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 {
3       public static void main(String[] args)
4       {
5             final int x=100;
6             x=200; //Compile Time Error
7       }
8
9 }
```

Java finally example:

```
1  class FinallyExample
2  {
3          public static void main(String[] args)
4          {
5                  try
6                  {
7                          int x=300;
8                  }
9                  catch(Exception e)
10                 {
11                         System.out.println(e);
12                 }
13                 finally
14                 {
15                         System.out.println("finally block is executed");
16                 }
17         }
18
19 }
```

Java finalize example:

```
1  class FinalizeExample
2  {
3       public void finalize()
4       {
5            System.out.println("finalize called");
6       }
7       public static void main(String[] args)
8       {
9            FinalizeExample f1=new FinalizeExample();
10           FinalizeExample f2=new FinalizeExample();
11           f1=null;
12           f2=null;
13           System.gc();
14      }
15 }
```

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.

ExceptionHandling with MethodOverriding in Java

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  {
4      void msg()
5      {
6          System.out.println("parent");
7      }
8  }
9  class TestExceptionChild extends Parent
10 {
11     void msg()throws IOException
12     {
13         System.out.println("TestExceptionChild");
14     }
15     public static void main(String args[])
16     {
17         Parent p=new TestExceptionChild();
18         p.msg();
19     }
20 }
21 Output:Compile Time Error
```

ExceptionHandling with MethodOverriding in Java

- Checked exceptions represent errors that the compiler checks at compile time, requiring the programmer to handle them explicitly (e.g., IOException, SQLException).
- If the superclass method does not declare any checked exceptions in its throws clause, the compiler assumes that calling the method will not lead to any checked exceptions.
- When a subclass overrides this method, if it declares a checked exception, the overridden method might throw exceptions that callers of the superclass method aren't prepared to handle. This violates **Liskov Substitution Principle**, which requires that the subclass method should be interchangeable with the superclass method.

ExceptionHandling with MethodOverriding in Java

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  {
4      void msg()
5      {
6          System.out.println("parent");
7      }
8  }
9  class TestExceptionChild1 extends Parent
10 {
11     void msg()throws ArithmeticException
12     {
13         System.out.println("child");
14     }
15     public static void main(String args[])
16     {
17         Parent p=new TestExceptionChild1();
18         p.msg();
19     }
20 }
21 Output:child
```

ExceptionHandling with MethodOverriding in Java

1. Checked Exceptions Are Part of the Method's API Contract
   - When a superclass method does not declare a checked exception, it explicitly guarantees to the caller that invoking the method will not require handling checked exceptions.
   - If a subclass overrides this method and declares a checked exception, it would break this guarantee. This could lead to compile-time errors for code written against the superclass.
2. Unchecked Exceptions Are Not Part of the Method's API Contract
   - Unchecked exceptions (subclasses of RuntimeException) are not required to be declared or explicitly handled in the method signature.
   - Since they are not part of the method's API contract, declaring an unchecked exception in a subclass does not affect the compatibility or behavior of the overridden method.

ExceptionHandling with MethodOverriding in Java

- Unchecked exceptions usually represent programming errors or runtime anomalies (e.g., NullPointerException, ArithmeticException), which do not need to be explicitly declared.

ExceptionHandling with MethodOverriding in Java

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

```
1  class Parent
2  {
3          void msg()throws ArithmeticException
4          {
5                  System.out.println("parent");
6          }
7  }
8  class TestExceptionChild2 extends Parent
9  {
10         void msg()throws Exception
11         {
12                 System.out.println("child");
13         }
14
15         public static void main(String args[])
16         {
17                 Parent p=new TestExceptionChild2();
18                 try
19                 {
20                         p.msg();
21                 }
22                 catch(Exception e)
23                 {
24
25                 }
26         }
27 }
28 Output:Compile Time Error
```

This behavior ensures adherence to the **Liskov Substitution Principle** and maintains the integrity of polymorphism in Java.

1. Superclass Method Declares an Exception:
   - When a superclass method declares an exception in its throws clause, it communicates to the caller that the method may throw that specific exception (or its subclasses).
   - The subclass is allowed to override the method and modify the exception declaration, but only in a way that narrows or eliminates the exceptions declared.
2. Why Can a Subclass Declare the Same Exception?
   - Declaring the same exception keeps the method signature compatible with the superclass.
   - Callers of the superclass method can handle the exception as expected without any additional changes.

ExceptionHandling with MethodOverriding in Java

Example: in case subclass overridden method declares same exception

```
1  import java.io.*;
2  class Parent
3  {
4      void msg()throws Exception
5      {
6          System.out.println("parent");
7      }
8  }
9  class TestExceptionChild3 extends Parent
10 {
11     void msg()throws Exception
12     {
13         System.out.println("child");
14     }
15
16     public static void main(String args[])
17     {
18         Parent p=new TestExceptionChild3();
```

ExceptionHandling with MethodOverriding in Java

```
19              try
20              {
21                      p.msg();
22              }
23              catch(Exception e)
24              {
25
26              }
27      }
28  }
29  Output:child
```

ExceptionHandling with MethodOverriding in Java

Example: in case subclass overridden method declares subclass exception

```
1  import java.io.*;
2  class Parent
3  {
4      void msg()throws Exception
5      {
6          System.out.println("parent");
7      }
8  }
9  class TestExceptionChild4 extends Parent
10 {
11     void msg()throws ArithmeticException
12     {
13         System.out.println("child");
14     }
15
16     public static void main(String args[])
17     {
18         Parent p=new TestExceptionChild4();
```

ExceptionHandling with MethodOverriding in Java

```
19              try
20              {
21                      p.msg();
22              }
23              catch(Exception e)
24              {
25
26              }
27      }
28  }
29  Output:child
```

ExceptionHandling with MethodOverriding in Java

Example: in case subclass overridden method declares no exception

```
1  import java.io.*;
2  class Parent
3  {
4      void msg()throws Exception
5      {
6          System.out.println("parent");
7      }
8  }
9  class TestExceptionChild5 extends Parent
10 {
11     void msg()
12     {
13         System.out.println("child");
14     }
15
16     public static void main(String args[])
17     {
18         Parent p=new TestExceptionChild5();
```

ExceptionHandling with MethodOverriding in Java

```
19              try
20              {
21                      p.msg();
22              }
23              catch(Exception e)
24              {
25
26              }
27      }
28  }
29  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.

ExceptionHandling with MethodOverriding in Java

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

```
class InvalidAgeException extends Exception
{
    InvalidAgeException(String s)
    {
        super(s);
    }
}
```

```
class TestCustomException1
{

    static void validate(int age)throws InvalidAgeException
    {
        if(age<18)
        throw new InvalidAgeException("not valid");
        else
        System.out.println("welcome to vote");
    }

```

ExceptionHandling with MethodOverriding in Java

```
12      public static void main(String args[])
13      {
14          try
15          {
16              validate(13);
17          }
18          catch(Exception m)
19          {
20              System.out.println("Exception occured: "+m);
21          }
22
23          System.out.println("rest of the code...");
24      }
25  }
26  Output:Exception occured: InvalidAgeException:not valid
27  rest of the code...
```

Practice Questions

1. What is wrong with the following code? Why it is showing compilation error?

```
1  public class JavaExceptionHandlingQuiz
2  {
3      public static void main(String[] args)
4      {
5          try
6          {
7              System.out.println("Try Block");
8          }
9
10         System.out.println("-----");
11
12         catch (Exception e)
13         {
14             System.out.println("Catch Block");
15         }
16     }
17 }
18 //Output: There should not be any other statements in between try and catch blocks.
```

Practice Questions

❷ What will be the output of the following program?

```
1  public class JavaExceptionHandlingQuiz
2  {
3      public static void main(String[] args)
4      {
5          int i = 1;
6
7          try
8          {
9              i++;
10         }
11         catch (Exception e)
12         {
13             i++;
14         }
15         finally
16         {
17             i++;
18         }
19
```

```
20              System.out.println(i);
21          }
22  }
23  //output: 3
```

Practice Questions

❸ What will be the output of the following program?

```
1  public class JavaExceptionHandlingQuiz
2  {
3      public static void main(String[] args)
4      {
5          try
6          {
7              System.out.println(1);
8
9              int i = 100 / 0;
10
11             System.out.println(2);
12         }
13         catch (Exception e)
14         {
15             System.out.println(3);
16         }
17     }
18 }
19 //Output: 1 3
```

Practice Questions

4. What will be the output of the following program?

```
1  public class JavaExceptionHandlingQuiz
2  {
3      public static void main(String[] args)
4      {
5          try
6          {
7              System.out.println(1);
8          }
9          catch (Exception e)
10         {
11             System.out.println(2);
12         }
13
14         System.out.println(3);
15
16         finally
17         {
18             System.out.println(4);
19         }
```

```
20          }
21  }
22  //Output: Compile time error. try, catch and finally blocks together form one unit. There should not be an
        other statements in between try-catch-finally blocks.
```

Practice Questions

5. What will be the output of the following program?

```
1  public class JavaExceptionHandlingQuiz
2  {
3      public static void main(String[] args)
4      {
5          try
6          {
7              System.out.println(1);
8
9              int i = Integer.parseInt("ABC");
10
11             System.out.println(2);
12         }
13         catch (NumberFormatException e)
14         {
15             System.out.println(3);
16         }
17         catch (Exception e)
18         {
19             System.out.println(4);
```

```
20              }
21          }
22  }
23  //Output: 1 3
```

Practice Questions

6. What will be the output of the following program?

```
1  public class JavaExceptionHandlingQuiz
2  {
3      public static void main(String[] args)
4      {
5          try
6          {
7              int[] a =
8              {
9                  1, 2, 3, 4
10             }
11             ;
12
13             int i = a[4];
14         }
15         catch (NumberFormatException e)
16         {
17             System.out.println(1);
18         }
19         catch (NullPointerException e)
```

```
20              {
21                  System.out.println(2);
22              }
23              catch (ArrayIndexOutOfBoundsException e)
24              {
25                  System.out.println(3);
26              }
27      }
28  }
```

Practice Questions

7. What will be the outcome of the following program?

```
1  public class JavaExceptionHandlingQuiz
2  {
3      public static void main(String[] args)
4      {
5          try
6          {
7              String s = null;
8              int i = s.length();
9          }
10         catch (Exception e)
11         {
12             System.out.println(1);
13         }
14         catch (NullPointerException e)
15         {
16             System.out.println(2);
17         }
18     }
```

```
19 }
20 // Output: Compile Time Error : Unreachable catch block for NullPointerException. It is already handle
       the catch block for Exception.
```

Practice Questions

Java Exception Handling Quiz

https://www.javatpoint.com/directload.jsp?val=89
https://www.javatpoint.com/directload.jsp?val=96
https://www.javatpoint.com/directload.jsp?val=101