**Wrapper Classes:**

The main objectives of wrapper classes are:

1. To wrap primitives into object form so that we can handle primitives also just like objects.

2. To define several utility functions which are required for the primitives.

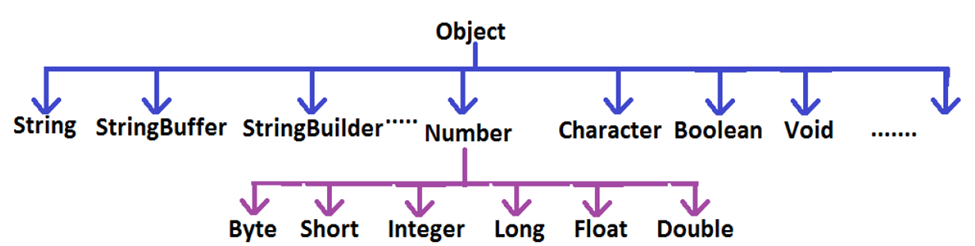
All most all wrapper classes define the following 2 constructors one can take

corresponding primitive as argument and the other can take String as argument.

**Example:**

**1) Integer i=new Integer(10);**

**2) Integer i=new Integer("10");**



**Note:**

1. String, StringBuffer, StringBuilder and all wrapper classes are final classes.

2. The wrapper classes which are not child class of Number are Boolean and Character.

3. The wrapper classes which are not direct class of Object are Byte, Short, Integer, Long, Float, Double.

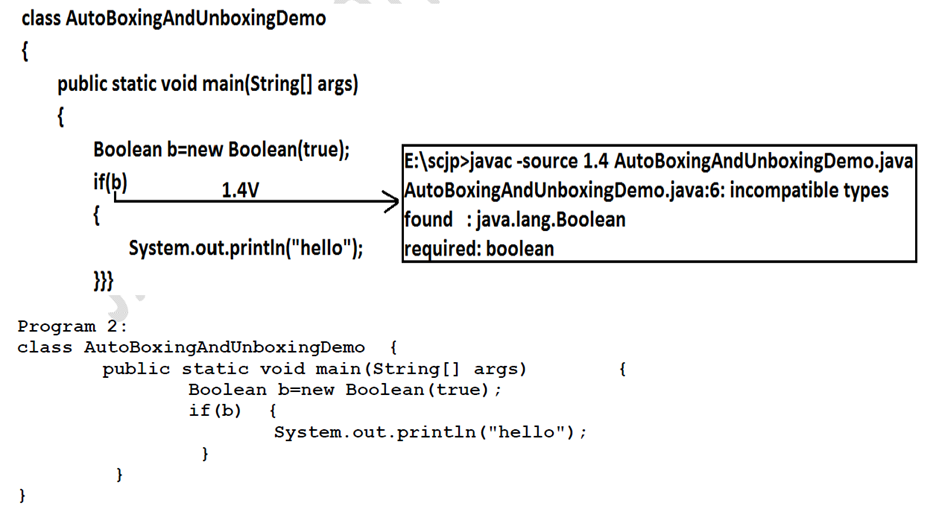
4. Sometimes we can consider Void is also as wrapper class.

5. In addition to String objects , all wrapper class objects also immutable in java

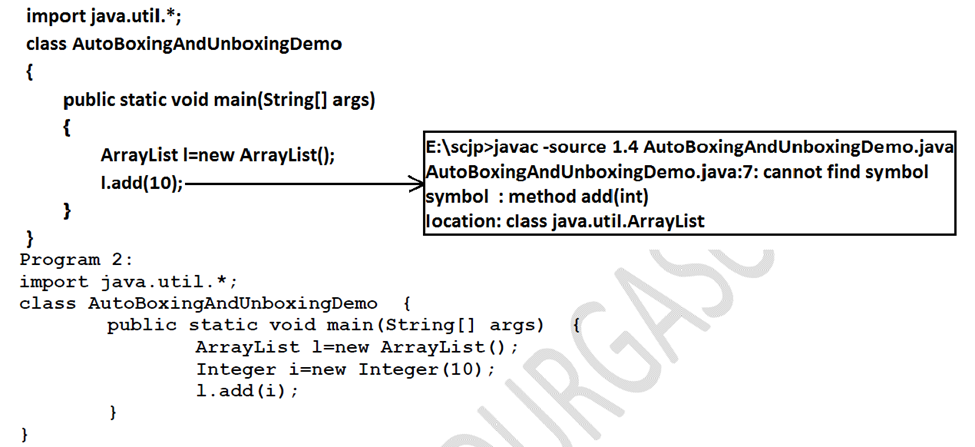
**Autoboxing and Auto-unboxing(1.5v):**

Until 1.4 version we can't provide wrapper object in the place of primitive and primitive in the place of wrapper object all the required conversions should be performed explicitly by the programmer.

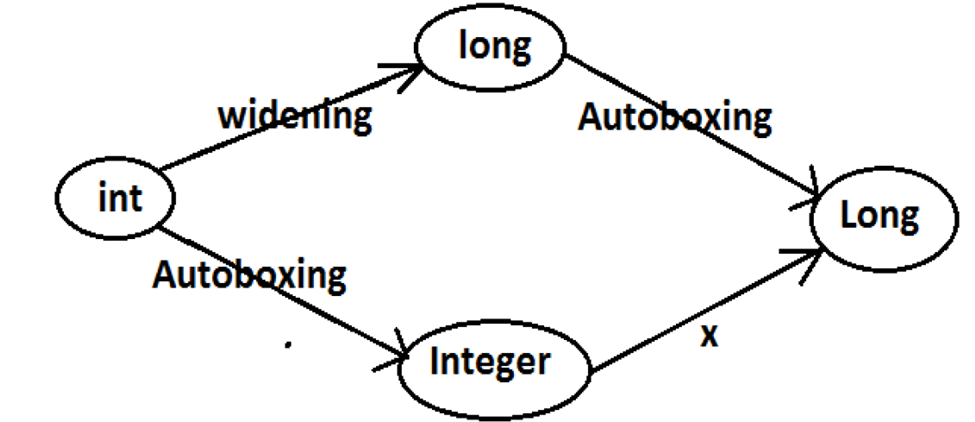
**Example 1:**

****

**Example 2:**

****

**Widening vs Autoboxing:**



**Autoboxing:** Automatic conversion of primitive to wrapper object by compiler is called

Autoboxing.

Example:

Integer i=10; [compiler converts “int” to “Integer” automatically by Autoboxing]

-> After compilation the above line will become.

Integer i=Integer.valueOf(10);

-> That is internally Autoboxing concept is implemented by using valueOf() method.

**Autounboxing:** automatic conversion of wrapper object to primitive by compiler is called

Autounboxing.

Example:

Integer i=new Integer(10);

Int i=I; [compiler converts “Integer” to “int” automatically by Autounboxing]

-> After compilation the above line will become.

Int i=I.intValue();

-> That is Autounboxing concept is internally implemented by using xxxValue() method.

**Exception Handling**

**Exception:** An unwanted unexpected event that disturbs normal flow of the program is called exception.

**Example:**

FileNotFoundException

**Note:** It is highly recommended to handle exceptions. The main objective of exception handling is graceful (normal) termination of the program.

**Example:**

**try**

**{**

**read data from SFTP server**

**}**

**catch(FileNotFoundException e)**

**{**

**use local file and continue rest of the program normally**

**}**

**Runtime Stack Mechanism:**

· For every thread JVM will create a separate stack at the time of Thread creation. All method calls performed by that thread will be stored in that stack. Each entry in the stack is called "Activation record" (or) "stack frame".

· After completing every method call JVM removes the corresponding entry from the stack.

· After completing all method calls JVM destroys the empty stack and terminates the program normally.

**Example:**

**class Test**

**{**

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

**doStuff();**

**}**

**public static void doStuff(){**

**doMoreStuff();**

**}**

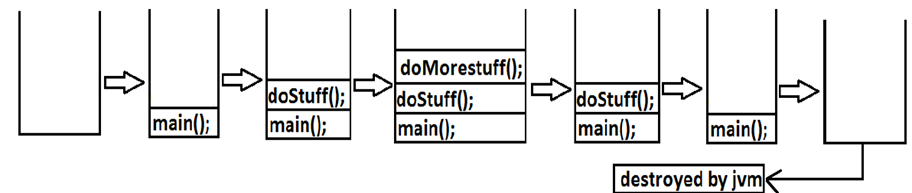
**public static void doMoreStuff(){**

**System.out.println("Hello");**

**}}**

**Output:**

**Hello**



**Default Exception Handling:**

1. If an exception raised inside any method then that method is responsible to create Exception object with the following information.

1. Name of the exception.

2. Description of the exception.

3. Location of the exception.(StackTrace)

2. After creating that Exception object, the method handovers that object to the JVM.

3. JVM checks whether the method contains any exception handling code or not. If method won't contain any handling code then JVM terminates that method abnormally and removes corresponding entry form the stack.

4. JVM identifies the caller method and checks whether the caller method contain any handling code or not. If the caller method also does not contain handling code then JVM terminates that caller method also abnormally and removes corresponding entry from the stack.

5. This process will be continued until main() method and if the main() method also doesn't contain any exception handling code then JVM terminates main() method also and removes corresponding entry from the stack.

6. Then JVM handovers the responsibility of exception handling to the default exception handler.

7. Default exception handler just print exception information to the console in the following format and terminates the program abnormally.

**Example:**

**class Test**

**{**

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

**doStuff();**

**}**

**public static void doStuff(){**

**doMoreStuff();**

**}**

**public static void doMoreStuff(){**

**System.out.println(10/0);**

**}}**

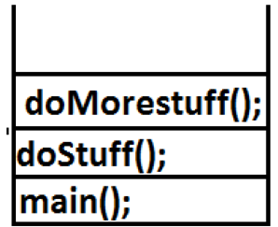
**Output:**

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

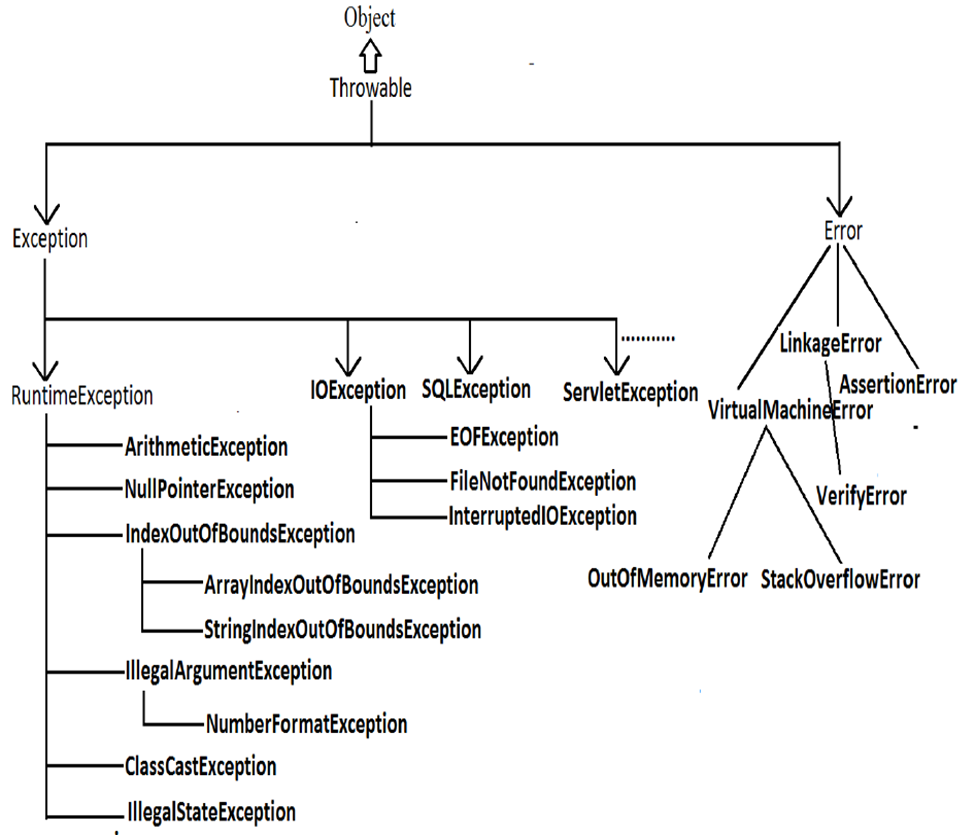
**atTest.doMoreStuff(Test.java:10)**

**atTest.doStuff(Test.java:7)**

**atTest.main(Test.java:4)**

****

**Exception Hierarchy:**

****

**Exception:**

Most of the cases exceptions are caused by our program and these are recoverable.

Example : If FileNotFoundException occurs then we can use local file and we can continue rest of the program execution normally.

**Error:**

Most of the cases errors are not caused by our program these are due to lack of system resources and these are non-recoverable.

Example :If OutOfMemoryError occurs being a programmer we can't do anything the program will be terminated abnormally.System Admin or Server Admin is responsible to raise/increase heap memory.

**Checked Exceptions vs Unchecked Exceptions:**

**Checked Exceptions :** The exceptions which are checked by the compiler whether programmer

handling or not, for smooth execution of the program at runtime, are called checked exceptions.

Example :

FileNotFoundException

**Unchecked Exceptions :** The exceptions which are not checked by the compiler whether programmer

handing or not, are called unchecked exceptions.

Example :

ArithmeticException

NullPointerException

**Note:**

· RuntimeException and its child classes, Error and its child classes are unchecked and all the remaining are considered as checked exceptions.

· Whether exception is checked or unchecked compulsory it should occurs atruntime only and there is no chance of occurring any exception at compile time.

**package** com.ttc;

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.util.Scanner;

**public** **class** CheckedAndUnchecked {

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

File f = **new** File("hello.txt");

**try** {

Scanner sc=**new** Scanner(f);

Thread.*sleep*(1000);

System.***out***.println(10/0);

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

**catch**(ArithmeticException e)

{

e.printStackTrace();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**Customized Exception Handling by using Try Catch:**

· It is highly recommended to handle exceptions.

· In our program the code which may raise exception is called risky code, we have to place risky code inside try block and the corresponding handling code inside catch block**.**

**Example:**

**try**

**{**

**Risky code**

**}**

**catch(Exception e)**

**{**

**Handling code**

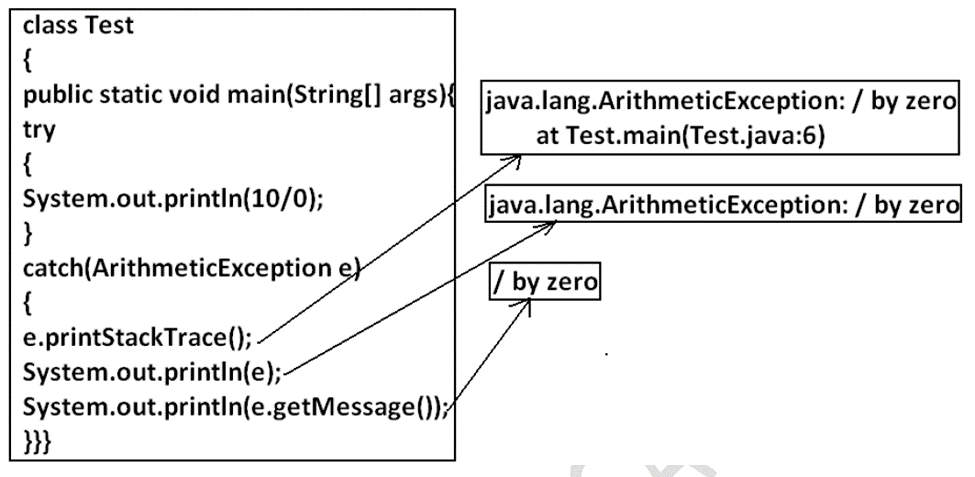
**}**

| **Without try catch** | **With try catch** |
| --- | --- |
| **class Test**  **{**  **public static void main(String[] args){**  **System.out.println("statement1");**  **System.out.println(10/0);**  **System.out.println("statement3");**  **}**  **}**  **output:**  **statement1**  **RE:AE:/by zero**  **at Test.main()**  **//Abnormal termination.** | **class Test{**  **public static void main(String[] args){**  **System.out.println("statement1");**  **try{**  **System.out.println(10/0);**  **}**  **catch(ArithmeticException e){**  **System.out.println(10/2);**  **}**  **System.out.println("statement3");**  **}}**  **Output:**  **statement1**  **5**  **statement3**  **//Normal termination.** |

**Methods to print exception information:**

Throwable class defines the following methods to print exception information to the console.

| printStackTrace(): | This method prints exception information in the following format.  Name of the exception: description of exception  Stack trace |
| --- | --- |
| toString(): | This method prints exception information in the following format.  Name of the exception: description of exception |
| getMessage(): | This method returns only description of the exception.  Description. |

****

**Try with multiple catch blocks:**

The way of handling an exception is varied from exception to exception. Hence for every exception type it is recommended to take a separate catch block. That is try with multiple catch blocks is possible and recommended to use.

**package** com.ttc;

**public** **class** MultipleCatchBlock {

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

**int**[] numbers = {1, 2, 3};

**try** {

System.***out***.println(numbers[5]);

}

**catch**(ArrayIndexOutOfBoundsException e)

{

System.***out***.println(e.getMessage());

}

**catch**(IndexOutOfBoundsException e)

{

System.***out***.println(e.getMessage());

}

}

}

| **try**  **{**  **.**  **.**  **.**  **.**  **}**  **catch(Exception e)**  **{**  **default handler**  **}** | **try**  **{**  .  .//fetching file  .//arthe  .  **catch(FileNotFoundException e)**  **{**  **use local file**  **}**  **catch(ArithmeticException e)**  **{**  **perform these Arithmetic operations**  **}**  **catch(SQLException e)**  **{**  **don't use oracle db, use mysqldb**  **}**  **catch(Exception e)**  **{**  **default handler**  **}** |
| --- | --- |
| This approach is not recommended because for any type of Exception we are using the same catch block. | This approach is highly recommended because for any exception raise we are defining a separate catch block. |

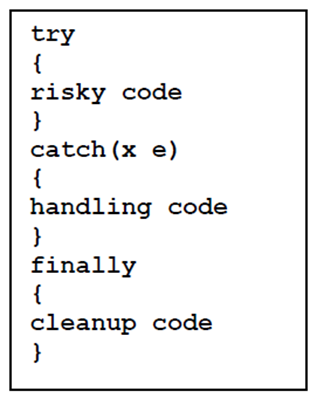
**Finally Block:**

· It is not recommended to take clean up code inside try block because there is no guarantee for the execution of every statement inside a try.

· It is not recommended to place clean up code inside catch block because if there is no exception then catch block won't be executed.

· We require some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled. Such type of best place is nothing but finally block.

· Hence the main objective of finally block is to maintain cleanup code.



**Note:**

The specialty of finally block is it will be executed always irrespective of whether the exception raised or not raised and whether handled or not handled.

**Difference between final, finally and finalize:**

**final:**

· final is the modifier applicable for classes, methods and variables.

· If a class declared as the final then child class creation is not possible.

· If a method declared as the final then overriding of that method is not possible.

· If a variable declared as the final then reassignment is not possible.

**finally:**

· finally is the block always associated with try-catch to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled.

**finalize:**

· finalize is a method, always invoked by Garbage Collector just before destroying

an object to perform cleanup activities**.**

**Note :**

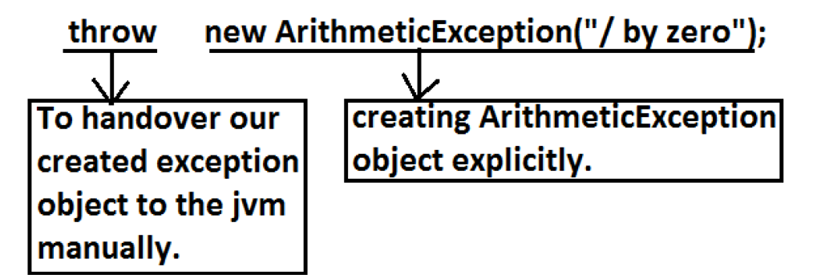
1. finally block meant for cleanup activities related to try block where as finalize() method meant for cleanup activities related to object.

2. To maintain clean up code finally block is recommended over finalize() method because we can't expect exact behavior of GC.

**Throw keyword, Throws Keyword:**

**Throw Keyword:**

Sometimes we can create Exception object explicitly and we can hand over to the JVM manually by using throw keyword.



| **class Test**  **{**  **public static void main(String[] args){**  **System.out.println(10/0);**  **}}** | **class Test**  **{**  **public static void main(String[]**  **args){**  **throw new ArithmeticException("/**  **by zero");**  **}}** |
| --- | --- |
| **In this case creation of ArithmeticException**  **object and handover to the jvm will be**  **performed automatically by the main() method.** | **In this case we are creating exception**  **object explicitly and handover to the**  **JVM manually.** |

**Throws Statement:**

In our program if there is any chance of raising checked exception then compulsory we should handle either by try catch or by throws keyword otherwise the code won't compile.

**Example:**

**import java.io.\*;**

**class Test3**

**{**

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

**PrinterWriter out=new PrintWriter("abc.txt");**

**out.println("hello");**

**}**

**}**

**CE :**

**Unreported exception java.io.FileNotFoundException; must be caught or**

**declared to be thrown.**

**Example:**

**class Test3**

**{**

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

**Thread.sleep(5000);**

**}**

**}**

**Unreported exception java.lang.InterruptedException; must be caught or**

**declared to be thrown.**

We can handle this compile time error by using the following 2 ways.

| **By using try catch** | **By using throws keyword** |
| --- | --- |
| **class Test3**  **{**  **public static void**  **main(String[] args){**  **try{**  **Thread.sleep(5000);**  **}**  **catch(InterruptedException**  **e){}**  **}**  **}**  **Output:**  **Compile and running**  **successfully** | **We can use throws keyword to delegate the**  **responsibility of exception handling to the caller**  **method. Then caller method is responsible to handle**  **that exception.**  **class Test3**  **{**  **public static void main(String[] args)throws**  **InterruptedException{**  **Thread.sleep(5000);**  **}**  **}**  **Output:**  **Compile and running successfully** |

**Note :**

· Hence the main objective of "throws" keyword is to delegate the responsibility of exception handling to the caller method.

· "throws" keyword required only checked exceptions. Usage of throws for unchecked exception there is no use.

· "throws" keyword required only to convince complier. Usage of throws keyword doesn't prevent abnormal termination of the program.

**Exception Handling Keyword Summary:**

1.try: To maintain risky code.

2. catch: To maintain handling code.

3. finally: To maintain cleanup code.

4. throw: To handover our created exception object to the JVM manually.

5. throws: To delegate responsibility of exception handling to the caller method.

**Customized Exceptions(User Defined Exceptions)**

Sometimes we can create our own exception to meet our programming requirements.

Such type of exceptions are called customized exceptions (user defined exceptions).

Example:

1. InSufficientFundsException

2. TooYoungException

3. TooOldException

**Program:**

**class TooYoungException extends RuntimeException**

**{**

**TooYoungException(String s)**

**{**

**super(s);**

**}**

**}**

**class TooOldException extends RuntimeException**

**{**

**TooOldException(String s)**

**{**

**super(s);**

**}**

**}**

**class CustomizedExceptionDemo**

**{**

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

**int age=Integer.parseInt(args[0]);**

**if(age>60)**

**{**

**throw new TooYoungException("please wait some more time.... u will get best**

**match");**

**}**

**else if(age<18)**

**{**

**throw new TooOldException("u r age already crossed....no chance of getting**

**married");**

**}**

**else**

**{**

**System.out.println("you will get match details soon by e-mail");**

**}}}**

**Output:**

**1)E:\scjp>java CustomizedExceptionDemo 61**

**Exception in thread "main" TooYoungException:**

**please wait some more time.... u will get best match**

**at CustomizedExceptionDemo.main(CustomizedExceptionDemo.java:21)**

**Top 5 Exceptions:**

**1. ArrayIndexOutOfBoundsException:**

**It is the child class of RuntimeException and hence it is unchecked. Raised**

**automatically by the JVM whenever we are trying to access array element with**

**out of range index. Example:**

**class Test{**

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

**int[] x=new int[10];**

**System.out.println(x[0]);//valid**

**System.out.println(x[100]);//AIOOBE**

**System.out.println(x[-100]);//AIOOBE**

**}**

**}**

**2. NullPointerException:**

**It is the child class of RuntimeException and hence it is unchecked. Raised**

**automatically by the JVM, whenever we are trying to call any method on null.**

**Example:**

**class Test{**

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

**String s=null;**

**System.out.println(s.length()); //R.E: NullPointerException**

**}**

**}**

**3. StackOverFlowError:**

**It is the child class of Error and hence it is unchecked. Whenever we are trying**

**to invoke recursive method call JVM will raise StackOverFloeError**

**automatically.**

**Example:**

**class Test**

**{**

**public static void methodOne()**

**{ methodTwo();**

**}**

**public static void methodTwo()**

**{**

**methodOne();**

**}**

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

**{**

**methodOne();**

**}**

**}**

**Output:**

**Run time error: StackOverFloeError**

**4. NoClassDefFoundError:**

**It is the child class of Error and hence it is unchecked. JVM will raise this error**

**automatically whenever it is unable to find required .class file. Example: java**

**Test If Test.class is not available. Then we will get NoClassDefFound error.**

**5.ClassCastException:**

**It is the child class of RuntimeException and hence it is unchecked. Raised**

**automatically by the JVM whenever we are trying to type cast parent object to**

**child type.**

