CHAPTER-4

INPUT/OUTPUT and EXCEPTION HANDLING

FILE HANDLING

The data stored in computer memory in two ways.

1) Temporary storage.

RAM is temporary storage. Whenever were executing java program that memory is created, when program completes memory destroyed. This type of memory is called volatile memory.

2) Permanent storage.

When we write a program and save it in hard disk, that type of memory is called permanent storage. It is also known as non-volatile memory.

When we work with stored files, we need to follow following task.

1) Determine whether the file is there or not.
2) Open a file.
3) Read the data from the file.
4) Writing information to file.
5) Closing file.

Stream :-

Stream is a channel it support continuous flow of data from one place to another place.

Java.io is a package that contains number of classes. By using that classes, we are able to send the data from one place to another place.

java
|------→io
|--→FileInputStream(class)
|--→FileOutputStream(class)
|--→FileReader(class)
|--→FileWriter(class)
|--→Serializable(interface)

In java language we are transferring the data in the form of two ways:-

1. Byte format
2. Character format

Stream/channel:-

It is acting as medium by using stream or channel we are able to send particular data from one place to the another place.

Streams are two types:-

1. Byte oriented stream.(supports byte formatted data to transfer)
2. Character oriented stream.(supports character formatted data to transfer)

Byte oriented streams:-

Java.io.FileInputStream

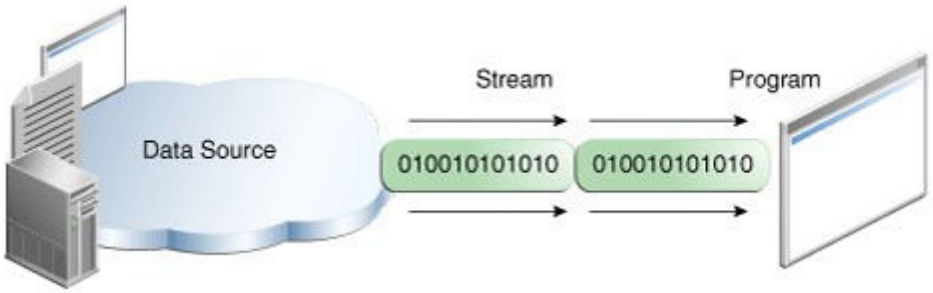
byte channel:-

1)**FileInputStream**

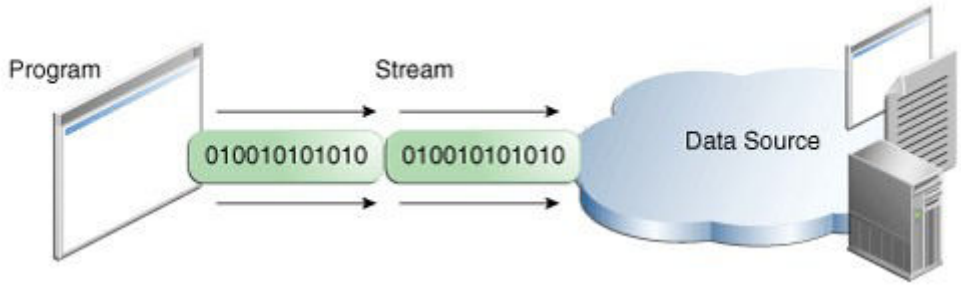
public native int read() throws java.io.IOException;
public void close() throws java.io.IOException;

2)**FileOutputStream**
public native void write(int) throws java.io.IOException;
public void close() throws java.io.IOException;

To read the data from the destination file to the java application, we have to use FileInputSream class.
To read the data from the .txt file, we have to use read() method.



Java.io.FileOutputStream:-
To write the data to the destination file, we have to use the FileOutputStream class.
To write the data to the destination file, we have to use write() method.



Ex:- It will supports one **character** at a time.

```
import java.io.*;
class Test
{
        public static void main(String[] args)throws Exception
        {
                //Byte oriented channel
        FileInputStream fis = new FileInputStream("abc.txt");//read data from source file
        FileOutputStream fos = new FileOutputStream("xyz.txt");//write data to target file
        int c;
        while((c=fis.read())!=-1)/                  /read and checking operations
        {
                System.out.print((char)c);          //printing data of the file
                fos.write(c);                       //writing data to target file
        }
        System.out.println("read() & write operatoins are completed");
        fis.close();                                //stream closing operations
        fos.close();
        }
}
```

File:

File f=new File("GameFile.txt");

➔This line first checks whether GameFile.txt file is already available (or) not.

➔If it is already available then "f" simply refers that file.

➔If it is not already available then it won't create any physical file, just creates a java File object represents name of the file.

Example:

```
import java.io.*;
class  FileDemo
{
        public static void main(String[] args) throws IOException
        {
                File f=new File("cricket.txt");          //no file yet
                System.out.println(f.exists());          //false
                f.createNewFile();          //make a file,  "cricket.txt" which is assigned to f
                System.out.println(f.exists());  //true
        }
}
```

output :

1st run :

false

true

2nd run :

true

true

A java File object can represent a directory also.

Example:

```
import java.io.*;
class  FileDemo
{
        public static void main(String[ ] args) throws IOException
        {
                File f=new File("cricket123");          //no directory yet
                System.out.println(f.exists());          //false
                f.mkdir();                               //create an actual directory
                System.out.println(f.exists());          //true
        }
}
```

Note: in UNIX everything is a file, java "file IO" is based on UNIX operating system hence in java also we can represent both files and directories by File object only.

File class constructors:

1. File f=new File(String name);
   Creates a java File object to represent name of the file or directory in current working directory.
2. File f=new File(String subdirname , String name);
   Creates a java File object to represent name of the file or directory present in specified sub directory.(some other location)

3. File f=new File(File subdir, String name);

Requirement: Write code to create a file named with demo.txt in current working directory.

Program:

```
import java.io.*;
class  FileDemo
{
        public static void main(String[] args)throws IOException
        {
                File f=new File("demo.txt");
                f.createNewFile();
        }
}
```

Requirement: Write code to create a directory named with Ramu123 in current working directory and create a file named with abc.txt in that directory.

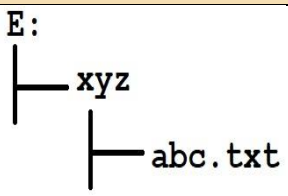
Program:

```
import java.io.*;
class  FileDemo
{
        public static void main(String[] args) throws IOException
        {
                File f1=new File("Ramu123"); //create an object
                f1.mkdir( );                 //create an actual directory
                File f2=new File("Ramu123","abc.txt");
                f2.createNewFile( );         //make a file, “Ramu123” which is assigned to f2
        }
}
```

Requirement: Write code to create a file named with abc.txt present in E:\xyz folder.

Program:

```
import java.io.*;
class  FileDemo
{
        public static void main(String[] args) throws IOException
        {
                File f=new File("E:\\xyz","abc.txt");
                f.createNewFile();
        }

}
```



Import methods of File class:

Methods	Description
boolean exists( );	Returns true if the specified file or directory available.
boolean createNewFile( );	First This method will check whether the specified file is already available or not. If it is already available then this method simply returns false without creating any physical file. If this file is not already available then it will create a new file and returns true
boolean mkdir( );	First This method will check whether the directory is already available or not. If it is already available then this method simply returns false without creating any directory. If this directory is not already available then it will create a new directory and returns true
boolean isFile( );	Returns true if the specified File object represents a physical file.
String[ ] list( );	It returns the names of all files and subdirectories present in the specified directory.
long length();	Returns the no of characters present in the file.
boolean isDirectory( );	Returns true if the File object represents a directory.
boolean delete( );	To delete a file or directory.

Requirement: Write a program to display the names of all files and directories present in c:\\ramu_classes.

```
import java.io.File;
import java.io.IOException;

public class  FileDemo
{
        public static void main(String[] args)throws IOException
        {
                int count=0;
                File f=new File("c:\\ramu_classes");
                String[ ] s=f.list( );          //list all files and directories in this directory

                for(String s1:s)                //for each string s1 in s
                {
                        count++;
                        System.out.println(s1);
                }

                System.out.println("total number : "+count);
        }
```

```
}
```

Requirement: Write a program to display only file names

```
import java.io.*;
class  FileDemo
{
        public static void main(String[] args) throws IOException
        {
                int count=0;
                File f=new File("c:\\ramu_classes");
                String[ ] s=f.list( );

                for(String s1:s)
                {
                        File f1=new File(f,s1);
                        if(f1.isFile( ))
                        {
                                count++;
                                System.out.println(s1);
                        }
                }
                System.out.println("total number : "+count);

        }
}
```

Requirement: Write a program to display only directory names

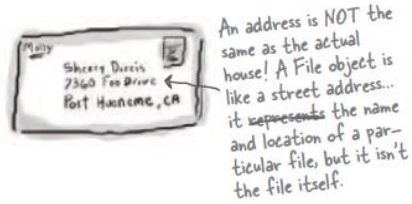
```
import java.io.*;
class  FileDemo
{
        public static void main(String[ ] args) throws IOException
        {
                int count=0;
                File f=new File("c:\\ramu_classes");
                String[ ] s=f.list( );

                for(String s1:s)
                {
                        File f1=new File(f,s1);
                        if(f1.isDirectory())
                        {
                                count++;
                                System.out.println(s1);
                        }
                }
                System.out.println("total number : "+count);

        }
}
```

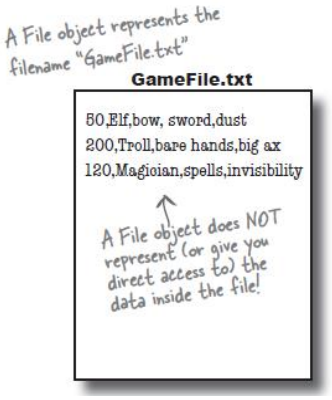
Important points:

➔**A file object represents the name and path of a file or directory on disk.**

➔**For example: /Users/Krishna/Data/GameFile.txt**

➔**But it does NOT represent, or give us access to, the data in the file!**





FileReader:

We can use FileReader object to read character data from the file.

Constructors:

1. **FileReader fr=new FileReader(String file);**
2. **FileReader fr=new FileReader (File f);**

Methods:

1) int read( );
It attempts to read next character from the file and return its ASCII value. If the next character is not available, then we will get -1.

```
int i=fr.read( );
System.out.println((char)i);
```

As this method returns ASCII value, while printing we have to perform type casting.

2) int read(char[ ] ch);
It attempts to read enough characters from the file into char[ ] array and returns the no of characters copied from the file into char[] array.

```
File f=new File("abc.txt");
Char[ ] ch=new Char[(int)f.length( )];
```

3) void close( );

It is used to close the FileReader class.

```
Approach 1:
import java.io.*;
class FileReaderDemo
{
        public static void main(String[] args)throws IOException
        {
                FileReader fr=new FileReader("cricket.txt");                    //just an object
                int i=fr.read( );           //more amount of data
                while(i != -1)
                {
                        System.out.print((char)i);      //typecasting mandatory
                        i=fr.read();
                }
        }
}
Output:
```

Charan
Software solutions
ABC

Approach 2:

```
import java.io.*;
class FileReaderDemo
{
        public static void main(String[] args) throws IOException
        {
                File f=new File("cricket.txt");                 //just an object
                FileReader fr=new FileReader(f);                //create a FileReader object
                char[ ] ch=new char[(int)f.length( )];          //small amount of data
                fr.read(ch);                                    //read the whole file!
                for(char ch1:ch)                                //print the array
                {
                        System.out.print(ch1);
                }
        }
}
```

Output:

XYZ
Software solutions.

Usage of *FileWriter* and *FileReader* is not recommended because :

1. While writing data by FileWriter compulsory we should insert line separator(\n) manually which is a bigger headache to the programmer.
2. While reading data by FileReader we have to read character by character instead of line by line which is not convenient to the programmer.
3. To overcome these limitations we should go for BufferedWriter and BufferedReader concepts.

FileWriter:

By using FileWriter object, we can write character data to the file.

Constructors:

```
FileWriter fw=new FileWriter(String fname);
```

➔ It creates a new file. It gets file name in string.

```
FileWriter fw=new FileWriter(File f);
```

➔ It creates a new file. It gets file name in File object.

The above 2 constructors meant for overriding.

Instead of overriding, if we want append operation then we should go for the following 2 constructors.

```
FileWriter fw=new FileWriter(String file, boolean append);
FileWriter fw=new FileWriter(File file, boolean append);
```

If the specified physical file is not already available, then these constructors will create that file.

Methods:

Method	Description
write(int ch);	To write a single character to the file.
write(char[ ] ch);	To write an array of characters to the file.
write(String text);	To write a String to the file.
flush( );	To give the guarantee the total data include last character also written to the file.
close( );	To close the stream.

Example:

```
import java.io.*;
class FileWriterDemo
{
        public static void main(String[ ] args) throws IOException
        {
                FileWriter fw=new FileWriter("cricket.txt",true); //create a FileWriter object
```

```
                    fw.write(99);                    //adding a single character
                    fw.write("charan\nsoftware solutions");   //write characters to the file
                    fw.write("\n");
                    char[ ] ch={'a','b','c'};
                    fw.write(ch);
                    fw.write("\n");
                    fw.flush( );                     //flush before closing
                    fw.close( );                     //close file when done always.
          }
}
Output:
charan
software solutions
abc
```

Note :

- The main problem with FileWriter is "we have to insert line separator manually", which is difficult to the programmer. ('\n')
- And even line separator varying from system to system.

BufferedWriter:

By using BufferedWriter object we can write character data to the file.

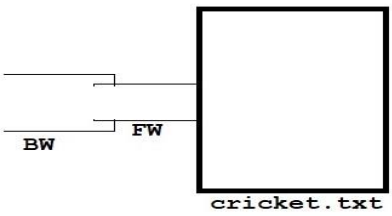
Constructors:

BufferedWriter bw=new BufferedWriter(writer w);
BufferedWriter bw=new BufferedWriter(writer w, int buffersize);

Note: BufferedWriter can't communicate directly with the file it can communicate via some writer object.

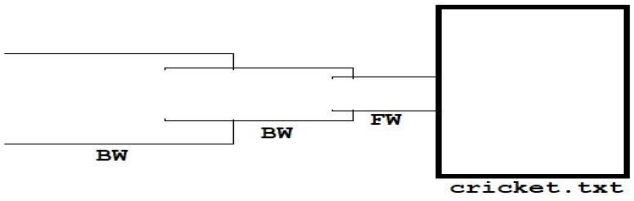
Which of the following declarations are valid?

1. BufferedWriter bw=new BufferedWriter("cricket.txt"); (invalid)
2. BufferedWriter bw=new BufferedWriter (new File("cricket.txt")); (invalid)
3. BufferedWriter bw=new BufferedWriter (new FileWriter("cricket.txt")); (valid)



BUFFERING

4)BufferedWriter bw=new BufferedWriter(new BufferedWriter(new FileWriter("cricket.txt"))); (valid)
Two level buffering



TWO LEVEL BUFFERING

Methods:

1. write(int ch); //To write a single character to the file
2. write(char[ ] ch); //To write n array of characters
3. write(String s); //To write string to the file
4. flush( );
5. close( );
6. newline( ); //To insert a line seperator

When compared with FileWriter which of the following capability is available extra as method form in BufferedWriter.

1. Writing data to the file.
2. Closing the file.
3. Flushing the file.
4. Inserting new line character.

Ans : 4

Example:

```
import java.io.*;
class BufferedWriterDemo
{
        public static void main(String[] args)throws IOException
        {
                FileWriter fw=new FileWriter("cricket.txt");
                BufferedWriter bw=new BufferedWriter(fw);
                bw.write(100);
                bw.newLine();
                char[ ] ch={'a','b','c','d'};
                bw.write(ch);
                bw.newLine();
                bw.write("Ramu");
                bw.newLine();
                bw.write("software solutions");
                bw.flush();
                bw.close();
                }
}
```

Output:

Note : Whenever we are closing BufferedWriter, automatically internal Filewriter will be closed and we are not required to close explicitly.

bw.close( )	fw.close( )	bw.close( ) fw.close( )
✓	✗	✗

BufferedReader:

We can use BufferedReader to read character data from the file.

The main advantage of BufferedReader when compared with FileReader is we can read data line by line in addition to character by character.

Constructors:

BufferedReader br=new BufferedReader(Reader r);
BufferedReader br=new BufferedReader(Reader r, int buffersize);

Note: BufferedReader can't communicate directly with the File and it can communicate via some Reader object.

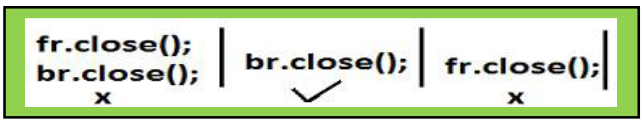
Methods:

1. int read(); //to read a single character
2. int read(char[] ch); //to read n array of characters
3. void close();
4. **String readLine();**
   It attempts to read next line and return it , from the File. if the next line is not available then this method returns null.

Example:

```
import java.io.*;
class BufferedReaderDemo
{
        public static void main(String[] args) throws IOException
        {
                FileReader fr=new FileReader("cricket.txt");
                BufferedReader br=new BufferedReader(fr);
                String line=br.readLine( );
                while(line!=null)
                {
                        System.out.println(line);
                        line=br.readLine( );
                }
                br.close( );
        }
}
```

Note: Whenever we are closing BufferedReader, automatically underlying FileReader will be closed, it is not required to close explicitly. Even this rule is applicable for BufferedWriter also.



The most enhanced Reader to read character data from the file is BufferedReader.

PrintWriter:

- This is the most enhanced Writer to write character data to the file.
- The main advantage of PrintWriter over FileWriter and BufferedWriter is
  ➔By using FileWriter and BufferedWriter we can write only character data to the File but
  ➔By using PrintWriter we can write any type of data to the File.

Constructors:

1) PrintWriter pw=new PrintWriter(String fname);
2) PrintWriter pw=new PrintWriter(File f);
3) PrintWriter pw=new PrintWriter(Writer w);

Note: PrintWriter can communicate directly with the File and can communicate via some Writer object also.

Methods:

1. write(int ch);
2. write (char[] ch);
3. write(String s);

4. flush();
5. close();

6. print(char ch);
7. print (int i);
8. print (double d);
9. print (boolean b);
10. print (String s);

11. println(char ch);
12. println (int i);
13. println(double d);
14. println(boolean b);
15. println(String s);

Example:

```
import java.io.*;
class PrintWriterDemo {
  public static void main(String[] args)throws IOException
            {
                        FileWriter fw=new FileWriter("cricket.txt");
                        PrintWriter out=new PrintWriter(fw);
```

```
                    out.write(100);
                    out.println(100);
                    out.println(true);
                    out.println('c');
                    out.println("SaiCharan");
                    out.flush();
                    out.close();
            }
}
```

Output:

d100

true

c

SaiCharan

What is the difference between write(100) and print(100)?

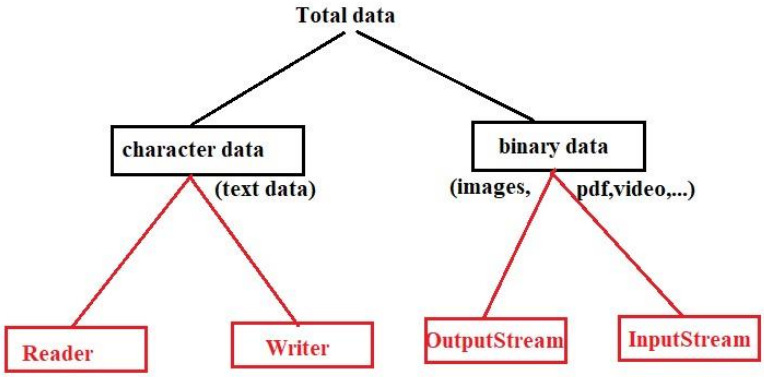
In the case of write(100) the corresponding character **"d"** will be added to the File but in the case of print(100) **"100" value** will be added directly to the File.

Note 1:

1. The most enhanced Reader to read character data from the File is **BufferedReader**.
2. The most enhanced Writer to write character data to the File is **PrintWriter.**

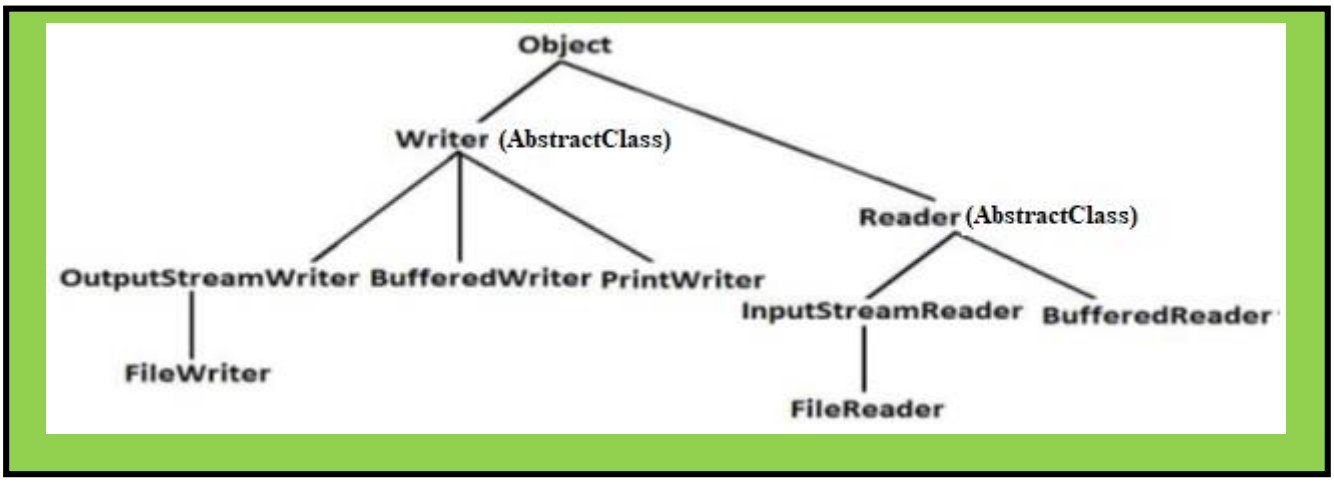
Note 2:

1. we can use **Readers and Writers** to handle **character data**.
   we can use **InputStreams and OutputStreams** to handle **binary data** (like images, audio files, video files etc).



2. We can use OutputStream to **write binary data** to the File and
   we can use InputStream to **read binary data** from the File.

Diagram:


Object
Writer (AbstractClass)
Reader (AbstractClass)
OutputStreamWriter
BufferedWriter
PrintWriter
InputStreamReader
BufferedReader
FileWriter
FileReader

EXCEPTION HANDLING

Agenda

1. Introduction to Exception Handling.
2. Runtime stack mechanism
3. Default exception handling in java

Introduction

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

Example:
SleepingException
TyrePunchuredException
FileNotFoundException ...etc

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

What is the meaning of exception handling?

Exception handling doesn't mean repairing an exception. We have to define alternative way to continue rest of the program normally this way of "defining alternative is nothing but exception handling".

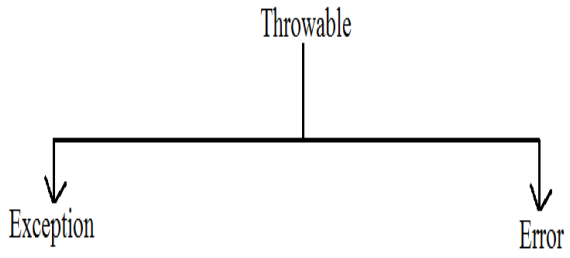
Example: Suppose our programming requirement is to read data from remote file locating at London at runtime if London file is not available our program should not be terminated abnormally.

We have to provide a local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

```
Example:

try
{
        read data from london file
}
catch(FileNotFoundException e)
{
        use local file and continue rest of the program normally
}
...
```

Exception hierarchy:

- Throwable acts as a root for exception hierarchy.
- Throwable class contains the following two child classes.



1) Exception: Most of the cases, exceptions are caused by our program and these are recoverable.
Ex :
If FileNotFoundException occurs, we can use local file and we can continue rest of the program execution normally.

2) 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.
Ex :
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 Vs Unchecked Exceptions:

➔The exceptions which are checked by the compiler for smooth execution of the program at runtime are called checked exceptions.

1. HallTicketMissingException
2. PenNotWorkingException
3. FileNotFoundException

➔The exceptions which are not checked by the compiler are called unchecked exceptions.

1. BombBlaustException
2. ArithmeticException
3. NullPointerException

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

Note: Whether exception is checked or unchecked compulsory it should occur at runtime only there is no chance of occurring any exception at compile time.

Partially checked vs fully checked :

➔A checked exception is said to be fully checked, if and only if all its child classes are also checked.
Example:
1) IOException
2) InterruptedException

➔A checked exception is said to be partially checked, if and only if some of its child classes are unchecked.

Example:
Exception

Note: The only partially checked exceptions available in java are:

1. Throwable.
2. Exception.

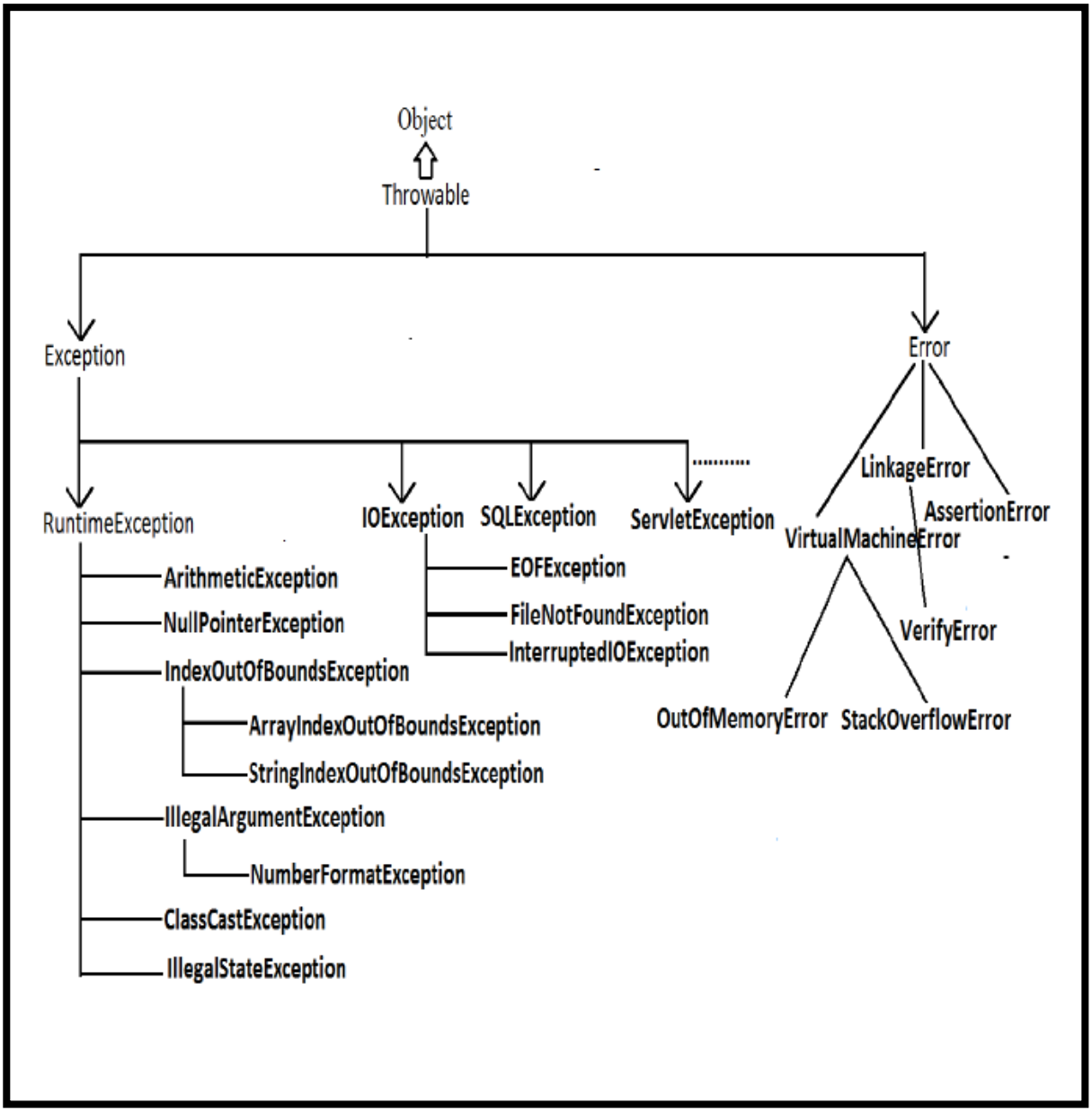


Figure: Exception Hierarchy

Q: Describe behavior of following exceptions ?

1. RuntimeException----------unchecked
2. Error----------unchecked
3. IOException----------fully checked
4. Exception----------partially checked
5. InterruptedException--------fully checked
6. Throwable----------partially checked
7. ArithmeticException -------unchecked
8. NullPointerException ------ unchecked
9. FileNotFoundException ----fully checked

Exception Handling

Agenda

1. **Customized exception handling by try catch**
2. **Control flow in try catch**
3. **Try with multiple catch blocks**
4. **Finally**
5. **Difference between final, finally, finalize**
6. **Control flow in try catch finally**
7. **Control flow in nested try catch finally**
8. **Various possible combinations of try catch finally**

1.Customized exception handling by try catch:

- It is highly recommended to handle exceptions.
- In our program the code which may cause an 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
{
psvm(String[] args)
{
S.o.p("statement1");
S.o.p(10/0);
S.o.p("statement3");
}
}
output:
statement1
RE:AE:/by zero
at Test.main()
	class Test
	{
	psvm(String[] args)
	{
	S.o.p("statement1");
	try
	{
	S.o.p(10/0);
	}
	catch(ArithmeticException e)
	{
	S.o.p(10/2);
	}

```
Abnormal termination.
```

```
    S.o.p("statement3");
}
}
Output:
statement1
5
statement3

Normal termination.
```

2.Control flow in try catch:

```
try
{
        statement1;
        statement2;
        statement3;
}
catch(X e)
{
        statement4;
}
statement5;
```

- **Case 1:** There is no exception.
  1, 2, 3, 5 normal termination.
- **Case 2:** if an exception raised at statement 2 and corresponding catch block matched 1, 4, 5 normal termination.
- **Case 3**: if an exception raised at statement 2 but the corresponding catch block not matched , 1 followed by abnormal termination.
- **Case 4:** if an exception raised at statement 4 or statement 5 then it's always abnormal termination of the program.

Note:

1. Within the try block if anywhere an exception raised then rest of the try block won't be executed even though we handled that exception. Hence we have to place/take only risk code inside try and length of the try block should be as less as possible.
2. If any statement which raises an exception and it is not part of any try block then it is always abnormal termination of the program.
3. There may be a chance of raising an exception inside catch and finally blocks also in addition to try block.

3.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.

Example:

```
try
{
.
.
}
catch(Exception e)
{
    default handler
}
```

```
try
{
.
.
}
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.

If try with multiple catch blocks present then order of catch blocks is very important. It should be from child to parent by mistake if we are taking from parent to child then we will get Compile time error saying, "exception xxx has already been caught".

Example:

```
class Test
{
      psvm(String[] args)
      {
            try
            {
            System.out.println(10/0);
            }
            catch(Exception e)
            {
            e.printStackTrace();
            }
            catch(ArithmeticException e)
            {
            e.printStackTrace();
            }
      }
}
```

CE:exception java.lang.ArithmeticException has already been caught

```
class Test
{
psvm(String[] args)
{
try
{
      System.out.println(10/0);
}
catch(ArithmeticException e)
{
      e.printStackTrace();
}
catch(Exception e)
{
      e.printStackTrace();
}}}
Output:
Compile successfully.
```

4.Finally block:

- It is never 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 never 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 place is nothing but finally block.
- Hence the main objective of finally block is to maintain cleanup code.

```
Example:

try
{
	risky code
}
catch(x e)
{
	handling code
}
finally
{
	cleanup code
}
```

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

```
Example 1:
class Test
{
	public static void main(String[] args)
	{
		try
		{
			System.out.println("try block executed");
		}
		catch(ArithmeticException e)
		{
			System.out.println("catch block executed");
		}
		finally
		{
			System.out.println("finally block executed");
		}
	}
}
Output:
Try block executed
Finally block executed

Example 2:
class Test
{
	public static void main(String[] args)
	{
		try
		{
			System.out.println("try block executed");
			System.out.println(10/0);
		}
		catch(ArithmeticException e)
```

```
		{
			System.out.println("catch block executed");
		}
		finally
		{
			System.out.println("finally block executed");
		}
	}
}
Output:
Try block executed
Catch block executed
Finally block executed
Example 3:
class Test
{
	public static void main(String[] args)
	{
		try
		{
			System.out.println("try block executed");
			System.out.println(10/0);
		}
		catch(NullPointerException e)
		{
			System.out.println("catch block executed");
		}
		finally
		{
			System.out.println("finally block executed");
		}
	}
}
Output:
Try block executed
Finally block executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at Test.main(Test.java:8)
```

5. Difference between final, finally, and finalize:

Final:

- Final is the modifier applicable for class, 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:

- It 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.
- There is only one situation where the finally block won't be executed is whenever we are using **System.exit(0)**method.

Finalize:

It is a method which should be called by garbage collector always just before destroying an object to perform cleanup activities.

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

6.Control flow in try catch finally:

Example:

```
class Test
{
        public static void main(String[] args)
        {
                Try
                {
                        System.out.println("statement1");
                        System.out.println("statement2");
                        System.out.println("statement3");
                }
                catch(Exception e)
                {
                        System.out.println("statement4");
                }
                finally
                {
                        System.out.println("statement5");
                }
                System.out.println("statement6");
        }
}
```

- **Case 1:** If there is no exception. 1, 2, 3, 5, 6 normal termination.
- **Case 2:** if an exception raised at statement 2 and corresponding catch block matched. 1,4,5,6 normal terminations.
- **Case 3:** if an exception raised at statement 2 and corresponding catch block is not matched. 1,5 abnormal termination.
- **Case 4:** if an exception raised at statement 4 then it's always abnormal termination but before the finally block will be executed.

- **Case 5:** if an exception raised at statement 5 or statement 6 its always abnormal termination.

7.Control flow in nested try catch finally:

```
Example:
class Test
{
	public static void main(String[] args)
	{
	try
	{
		System.out.println("statement1");
		System.out.println("statement2");
		System.out.println("statement3");
		try
		{
			System.out.println("statement4");
			System.out.println("statement5");
			System.out.println("statement6");
		}
		catch(ArithmeticException  e)
		{
			System.out.println("statement7");
		}
		finally
		{
			System.out.println("statement8");
		}
		System.out.println("statement9");
	}
	catch(Exception e)
	{
		System.out.println("statement10");
	}
	finally
	{
		System.out.println("statement11");
	}
	System.out.println("statement12");
	}
}
```

- **Case 1:** if there is no exception. 1, 2, 3, 4, 5, 6, 8, 9, 11, 12 normal termination.
- **Case 2:** if an exception raised at statement 2 and corresponding catch block matched 1,10,11,12 normal terminations.
- **Case 3**: if an exception raised at statement 2 and corresponding catch block is not matched 1, 11 abnormal termination.
- **Case 4:** if an exception raised at statement 5 and corresponding inner catch has matched 1, 2, 3, 4, 7, 8, 9, 11, 12 normal termination.

- **Case 5:** if an exception raised at statement 5 and inner catch has not matched but outer catch block has matched. 1, 2, 3, 4, 8, 10, 11, 12 normal termination.
- **Case 6:** if an exception raised at statement 5 and both inner and outer catch blocks are not matched. 1, 2, 3, 4, 8, 11 abnormal termination.
- **Case 7:** if an exception raised at statement 7 and the corresponding catch block matched 1, 2, 3, 4, 5, 6, 8, 10, 11, 12 normal termination.
- **Case 8:** if an exception raised at statement 7 and the corresponding catch block not matched 1, 2, 3, 4, 5, 6, 8, 11 abnormal terminations.
- **Case 9:** if an exception raised at statement 8 and the corresponding catch block has matched 1, 2, 3, 4, 5, 6, 7, 10, 11,12 normal termination.
- **Case 10:** if an exception raised at statement 8 and the corresponding catch block not matched 1, 2, 3, 4, 5, 6, 7, 11 abnormal terminations.
- **Case 11:** if an exception raised at statement 9 and corresponding catch block matched 1, 2, 3, 4, 5, 6, 7, 8,10,11,12 normal termination.
- **Case 12:** if an exception raised at statement 9 and corresponding catch block not matched 1, 2, 3, 4, 5, 6, 7, 8, 11 abnormal termination.
- **Case 13:** if an exception raised at statement 10 is always abnormal termination but before that finally block 11 will be executed.
- **Case 14:** if an exception raised at statement 11 or 12 is always abnormal termination.

Note: if we are not entering into the try block then the finally block won't be executed. Once we entered into the try block without executing finally block we can't come out.

We can take try-catch inside try i.e., nested try-catch is possible
The most specific exceptions can be handled by using inner try-catch and generalized exceptions can be handle by using outer try-catch.

```
Example:
class Test
{
	public static void main(String[] args)
	{
		try
		{
			System.out.println(10/0);
		}
		catch(ArithmeticException e)
		{
			System.out.println(10/0);
		}
		finally{
		String s=null;
		System.out.println(s.length());
		}
	}
}
output :
RE:NullPointerException
```

Note: Default exception handler can handle only one exception at a time and that is the most recently raised exception.

8.Various possible combinations of try catch finally:

1. Whenever we are writing try block compulsory we should write either catch or finally.
   i.e., try without catch or finally is invalid.
2. Whenever we are writing catch block compulsory we should write try.
   i.e., catch without try is invalid.
3. Whenever we are writing finally block compulsory we should write try.
   i.e., finally without try is invalid.
4. In try-catch-finally order is important.
5. With in the try-catch -finally blocks we can take try-catch-finally.
   i.e., nesting of try-catch-finally is possible.
6. For try-catch-finally blocks curly braces are mandatory.

```
try {}
catch (X e) {}
```
✓

```
try {}
catch (X e) {}
catch (Y e) {}
```
✓

```
try {}
catch (X e) {}
catch (X e) {} //CE:exception ArithmeticException has already been caught
```
✗

```
try {}
catch (X e) {}
finally {}
```
✓

```
try {}
finally {}
```
✓

```
try {} //CE: 'try' without 'catch', 'finally' or resource declarations
```
✗

```
catch (X e) {} //CE: 'catch' without 'try'
```
✗

```
finally {} //CE: 'finally' without 'try'
```
✗

```
try {} //CE: 'try' without 'catch', 'finally' or resource declarations
System.out.println("Hello");
catch {} //CE: 'catch' without 'try'
```

✗

```
try {}
catch (X e) {}
System.out.println("Hello");
catch (Y e) {} //CE: 'catch' without 'try'
```

✗

```
try {}
catch (X e) {}
System.out.println("Hello");
finally {} //CE: 'finally' without 'try'
```

✗

```
try {}
finally {}
catch (X e) {} //CE: 'catch' without 'try'
```

✗

```
try {}
catch (X e) {}
try {}
finally {}
```

✓

```
try {}
catch (X e) {}
finally {}
finally {} //CE: 'finally' without 'try'
```

✗

```
try {}
catch (X e) {
try {}
catch (Y e1) {}
}
```

✓

```
try {
try {} //CE: 'try' without 'catch', 'finally' or resource declarations
}
catch (X e) {}
```

✗

```
try //CE: '{' expected
System.out.println("Hello");
catch (X e1) {} //CE: 'catch' without 'try'
```

✗

```
try {}
catch (X e) //CE:'{' expected
System.out.println("Hello");
```

✗

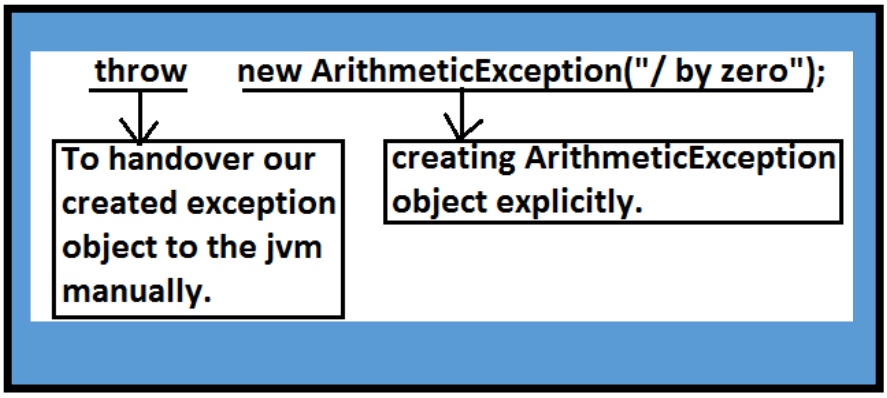
```
try {}
catch (NullPointerException e1) {}
finally //CE: '{' expected
System.out.println("Hello");
```

✗

Throw statement:

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

Example:



The result of following 2 programs is exactly same.

```
class Test
{
      p s v main(String[] args)
      {
          S.o.p(10/0);
      }
}
```

In this case creation of arithmeticException object and handover to the jvm will be performed automatically by the main() method.

```
class Test
{
   public static void main(String[] args)
   {
        throw new ArithmeticException("/ by zero");
   }
}
```

In this case we are creating exception object explicitly and handover to the JVM manually.

Note: In general we can use throw keyword for customized exceptions but not for predefined exceptions.

Case 1:

throw e;

If e refers null then we will get NullPointerException.

Example:

```
class Test3
{
  static ArithmeticException e=new ArithmeticException( );
  public static void main(String[] args)
  {
     throw e;
   }
}
Output:
Runtime exception: Exception in thread "main"
          java.lang.ArithmeticException
```

```
class Test3
{
     static ArithmeticException e;
      p s v main(String[] args)
     {
          throw e;
      }
}

Output:
Exception in thread "main"
   java.lang.NullPointerException
       at Test3.main(Test3.java:5)
```

Case 2:

After throw statement we can't take any statement directly otherwise we will get compile time error saying unreachable statement.

Example:

```
class Test3
{
public static void main(String[] args){
System.out.println(10/0);
System.out.println("hello");
}
}
Output:
Runtime error: Exception in thread "main"
  java.lang.ArithmeticException: / by zero
       at Test3.main(Test3.java:4)
```

```
class Test3
{
public static void main(String[] args){
throw new ArithmeticException("/ by zero");
System.out.println("hello");
}
}
Output:
Compile time error.
Test3.java:5: unreachable statement
System.out.println("hello");
```

Case 3:

We can use throw keyword only for Throwable types otherwise we will get compile time error saying incomputable types.

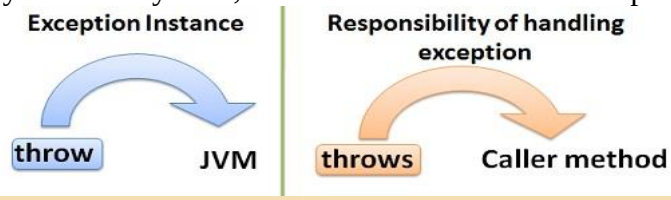
Example:

```
class Test3
{
   public static void main(String[] args)
  {
     throw new Test3();
   }
}
Output:
Compile time error.
Test3.java:4: incompatible types
found   : Test3
required: java.lang.Throwable
throw new Test3();
```

```
class Test3 extends RuntimeException
{
      public static void main(String[] args)
     {
           throw new Test3();
      }
}
Output:
Runtime error: Exception in thread "main"
Test3
        at Test3.main(Test3.java:4)
```

Throws statement:

In our program if there is any chance of raising checked exception, 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.

Example:

By using try catch	By using throws keyword

class Test3
{
psvm(String[] args)
{
try
{
Thread.sleep(5000);
}
catch(InterruptedException e){}
}
}
Output:
Compile and running successfully
	We can use throws keyword to delicate the responsibility of exception handling to the caller method. Then caller method is responsible to handle that exception.
	class Test3
	{
	psvm(String[] args) throws InterruptedException
	{
	Thread.sleep(5000);
	}
	}
	Output:
	Compile and running successfully

Note :

- Hence the main objective of "throws" keyword is to delicate 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 convenes complier. Usage of throws keyword doesn't prevent abnormal termination of the program.
  Hence recommended to use try-catch over throws keyword.

```
Example:
class Test
{
        public static void main(String[] args)throws InterruptedException
        {
                doStuff();
        }
        public static void doStuff()throws InterruptedException
        {
                doMoreStuff();
        }
        public static void doMoreStuff()throws InterruptedException
        {
                Thread.sleep(5000);
        }
}
Output:
Compile and running successfully.
```

In the above program, if we are removing at least one throws keyword, then the program won't compile.

Case 1:
we can use throws keyword only for Throwable types otherwise we will get compile time error saying "incompatible types".

Example:

```
class Test3
{
      psvm(String[] args) throws Test3
     {
      }
}
Output:
Compile time error
Test3.java:2: incompatible types
found   : Test3
```

```
class Test3 extends RuntimeException
{
     psvm(String[] args) throws Test3
     {
      }
}
Output:
Compile and running successfully.
```

```
required: java.lang.Throwable
public static void main(String[] args)
                    throws Test3
```

Case 2: Example:

```
class Test3
{
   public static void main(String[] args)
   {
     throw new Exception();
    }
}
Output:
Compile time error.
Test3.java:3: unreported exception
    java.lang.Exception;
must be caught or declared to be thrown
```

```
class Test3
{
      public static void main(String[] args)
     {
           throw new Error();
     }
}
Output:
Runtime error
Exception in thread "main" java.lang.Error
       at Test3.main(Test3.java:3)
```

Case 3:

In our program with in the try block, if there is no chance of rising an exception then we can't write catch block for that exception. otherwise we will get compile time error saying **"exception XXX is never thrown in body of corresponding try statement".** But this rule is applicable only for fully checked exception.

Example:

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(Exception e)
{}   output:
}    hello
}    partial checked
```

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(ArithmeticException e)
{}  output:
}   hello
}   unchecked
```

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(java.io.IOException e)
{} output:
}  compile time error
}  fully checked
```

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(InterruptedException e)
{}  output:
}   compile time error
}   Fully checked
```

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(Error e)
{} output:
}  compile successfully
}  unchecked
```

Case 4:

We can use throws keyword only for constructors and methods but not for classes.

Example:

```
class Test  throws Exception      //invalid
{
  Test() throws Exception                 //valid
   { }
   methodOne() throws Exception      //valid
    { }
}
```

Exception handling keywords 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.

Various possible compile time errors in exception handling:

1. Exception XXX has already been caught.
2. Unreported exception XXX must be caught or declared to be thrown.
3. Exception XXX is never thrown in body of corresponding try statement.
4. Try without catch or finally.
5. Catch without try.
6. Finally without try.
7. Incompatible types.
   Found:test
   Requried:java.lang.Throwable;

8. Unreachable statement.

DIFFERENCES BETWEEN THROW AND THROWS:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Throw is followed by an instance.	Throws is followed by class.
3)	Throw is used within the method.	Throws is used with the method signature.
4)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

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:\Sai>java CustomizedExceptionDemo 61
Exception in thread "main" TooYoungException:
please wait some more time.... u will get best match
at CustomizedExceptionDemo.main(CustomizedExceptionDemo.java:21)



2)E:\Sai>java CustomizedExceptionDemo 27
You will get match details soon by e-mail

3)E:\Sai>java CustomizedExceptionDemo 9
Exception in thread "main" TooOldException:
u r age already crossed....no chance of getting married
at CustomizedExceptionDemo.main(CustomizedExceptionDemo.java:25)
```

Note: It is highly recommended to maintain our customized exceptions as unchecked by extending RuntimeException.
We can catch any Throwable type including Errors also.

Example:

```
try
{}
catch(Error e)
{}
```

valid