Week – 11

FILES AND EXCEPTION HANDLING

Files and I/O streams

Java I/O (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Let's see the code to print **output and an error** message to the console.

```
System.out.println("simple message");
System.err.println("error message");
```

Let's see the code to get **input** from console.

```
int i=System.in.read();//returns ASCII code of 1st character
System.out.println((char)i);//will print the character
```

OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

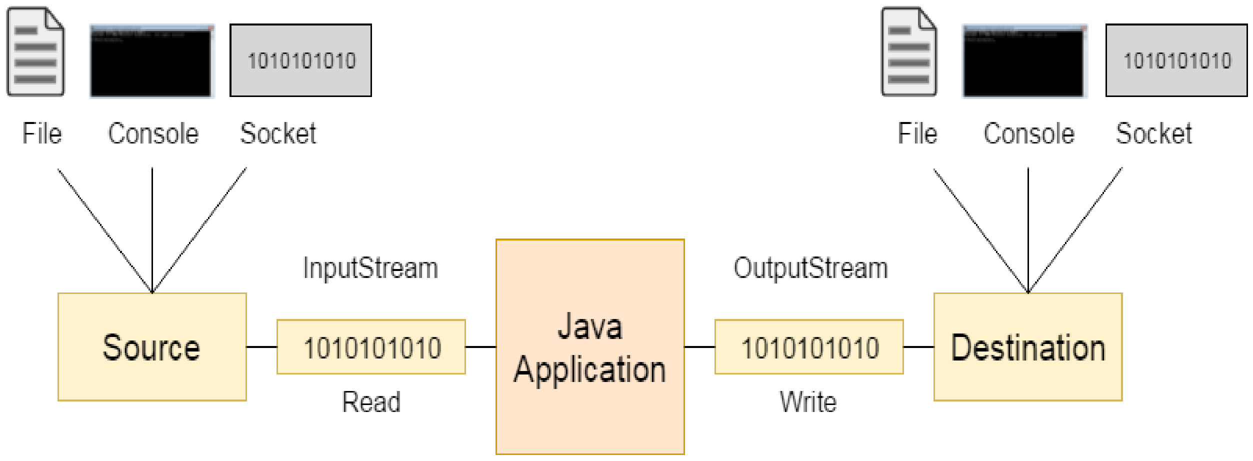
OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.



Java FileWriter and FileReader

FileWriter class in Java

The FileWriter class is used to write the data to a file of a given file name or full path string. The FileWriter class will throw an Exception named IOException or SecurityException so you need to handle the Exception in your code.

Creation of a FileWriter is not dependent on the file already existing. A FileWriter will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an IOException will be thrown.

There is commonly used constructor

FileWriter(String *path of file*)
In this constructor you give the path of a specific directory like as "D:\bhaskar\b.java".

FileWriter(String *pathoffile*, boolean *append*)
In this constructor there are two arguments, first *pathofthefile* and
second *append* if *append* is **true** then ouput is appended to the end of the file.

FileWriter(File *fileobject*)
In this constructor you can give the direct File type object that describes the file.

FileReader class in Java

The "FileReader" class is used for reading the data (contents) of a file. This class is a child class of the Reader class; their commonly used constructor is a follow:

Either can throw a FileNotFoundException or IOException.

FileReader(String *path*)

This type constructor takes the *path* = full path of the file which file you want to read.

FileReader(File objectfile)

In this constructor you can give the direct File type object that describe the file.

Example

The following example shows how a string can be written to a file with the help of the FileWriter class; this program first checks the current directory to determine if the particular file is available or not; if not then it creates a new file before writing the data to the file.

```
import java.io.*;
class MyFileWriter {
    public static void main(String arg[]) throws IOException {
        // first, we make the object of FileWriter class
        FileWriter fw = new FileWriter("bhaskar.txt");
        String s = "Good Morning, to all my 4th Sem Students.";
          // toCharArray() is method to convert a string in a character array
        char ch[] = s.toCharArray();
        for (int i = 0; i < ch.length; i++)
            fw.write(ch[i]);
        fw.close();
    }

}
```

Example

The following program shows how we can read a file's content from a given file.
First checks that the particular file exists or not; if not then it will throw an exception (that is FileNotFoundException).

```
import java.io.*;
class MyFileReader {
    public static void main(String arg[]) throws IOException {
        int i = 0;
        FileReader fr = new FileReader("bhaskar.txt");
        while ((i = fr.read()) != -1)
            System.out.print((char) i);
        fr.close();
    }
}
```

Exception concept

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

Or

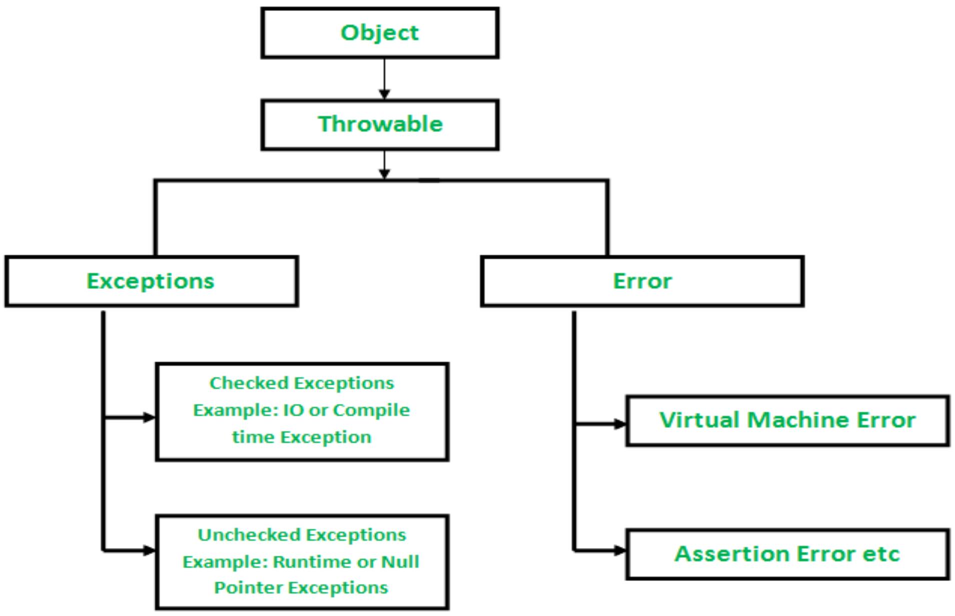
An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Classification of Exceptions:

1. Checked exceptions
2. Unchecked exceptions


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

Checked Exceptions

These are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using the *throws* keyword. In checked exception, there are two types: fully checked and partially checked exceptions. A fully checked exception is a checked exception where all its child classes are also checked, like IOException, InterruptedException. A partially checked exception is a checked exception where some of its child classes are unchecked, like Exception.

UnChecked Exceptions

In Java, exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.
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.

Some common unchecked exceptions in Java are *NullPointerException*, *ArrayIndexOutOfBoundsException* and *IllegalArgumentException*.

Exception Handling in Java

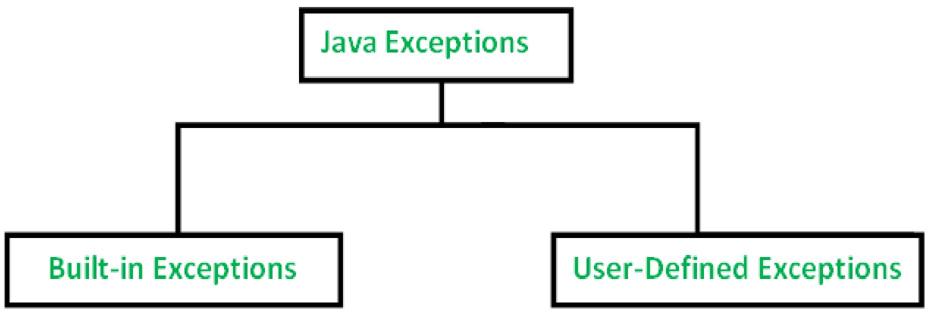
The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

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

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

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

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



Below is the list of important built-in exceptions in Java.

1. **ArithmeticException**
   It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException**
   It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException**
   This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException**
   This Exception is raised when a file is not accessible or does not open.
5. **IOException**
   It is thrown when an input-output operation failed or interrupted
6. **InterruptedException**
   It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.
7. **NoSuchFieldException**
   It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException**
   It is thrown when accessing a method which is not found.
9. **NullPointerException**
   This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException**
    This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException**
    This represents any exception which occurs during runtime.

Exception Handling mechanism

1. find the problem (**Hit** the exception)
2. inform that an error has occurred (**Throw** the exception)
3. receive the error information (**Catch** the exception)
4. take corrective actions (**Handle** the exception)

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

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.

Java try-catch block

Java try block

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

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

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

java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

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

Syntax of Java try-catch

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

Syntax of try-finally block

```
try{
//code that may throw an exception
} finally{}
```

Java Nested try block

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

Why use nested try block

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

Syntax:

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

Java nested try example

```
    class Excep6{
     public static void main(String args[]){
      try{
            try{
             System.out.println("going to divide");
             int b =39/0;
            }catch(ArithmeticException e){System.out.println(e);}

            try{
            int a[]=new int[5];
            a[5]=4;
       }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

        System.out.println("other statement);
      }catch(Exception e){System.out.println("handeled");}

      System.out.println("normal flow..");
     }
    }
```

Java catch multiple exceptions

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.

- 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.

```
public class MultipleCatchBlock1 {

   public static void main(String[] args) {

           try{
               int a[]=new int[5];
               a[5]=30/0;
              }
              catch(ArithmeticException e)
                {
                 System.out.println("Arithmetic Exception occurs");
                }
              catch(ArrayIndexOutOfBoundsException e)
                {
                 System.out.println("ArrayIndexOutOfBounds Exception occurs");
                }
              catch(Exception e)
                {
                 System.out.println("Parent Exception occurs");
                }
              System.out.println("rest of the code");
   }
}
```

Output

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

Java finally block

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

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

Java finally block follows try or catch block.

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

Why use java finally

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

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).

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

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

Throwing Our Own Exceptions

There may be times when we would like to throw our own exceptions. We can do this using the keyword **throw** as follows:

throw new **Throwable_subclass**;

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

Example:

```
public class TestThrow1{
   static void validate(int age)
   {
     if(age<18)
```

```
      throw new ArithmeticException("not valid");
     else
      System.out.println("welcome to vote");
   }
   public static void main(String args[])
   {
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

Output:

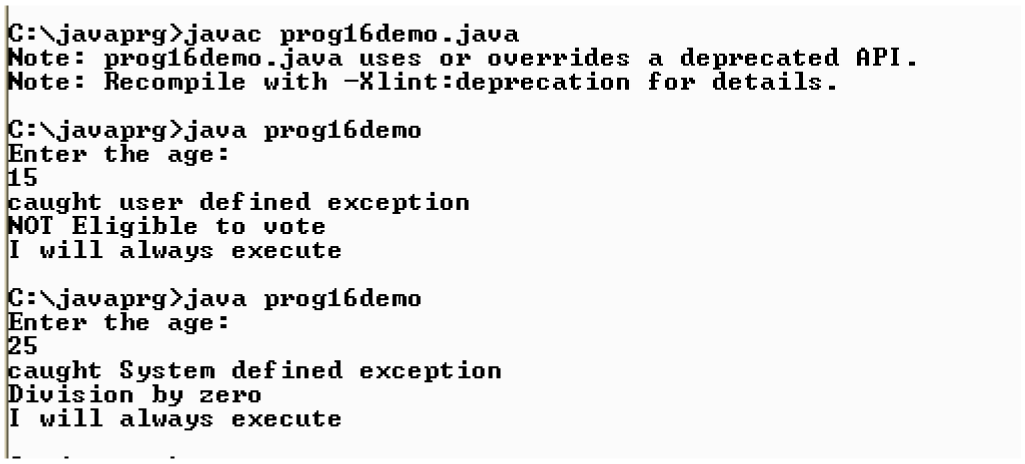
```
Exception in thread main java.lang.ArithmeticException:not valid
```

WAJP to implement the concept of Exception Handling

- using predefined exception.
- by creating user defined exception

```
import java.io.*;
class MyException extends Exception
{
   MyException(String msg)
   {
    super(msg);
   }
}

class prog16demo
{
   public static void main(String args[])
   {
     int age; int a=10;int b=5;int c=5;
     try
     {
      DataInputStream in= new DataInputStream(System.in);
      System.out.println("Enter the age:");
      String s=in.readLine();
      age=Integer.parseInt(s);
      if(age<18)
      {
       throw new MyException("NOT Eligible to vote");
      }
      int x=a/(b-c);
     }
     catch(IOException e)
     {
     System.out.println("IO error");
     }
     catch(MyException e)
     {
      System.out.println("caught user defined exception");
      System.out.println(e.getMessage());
     }
     catch(ArithmeticException e)
     {
      System.out.println("caught System defined exception");
      System.out.println("Division by zero");
     }
   finally
   {
    System.out.println("I will always execute");
   }
  }
}
```



```
C:\javaprg>javac prog16demo.java
Note: prog16demo.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\javaprg>java prog16demo
Enter the age:
15
caught user defined exception
NOT Eligible to vote
I will always execute

C:\javaprg>java prog16demo
Enter the age:
25
caught System defined exception
Division by zero
I will always execute
```