Exception Handling

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.

Built-in Exceptions

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. 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.

12. **StringIndexOutOfBoundsException**

It is thrown by String class methods to indicate that an index is either negative than the size of the string

Examples of Built-in Exception:

- **Arithmetic exception**

```
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        try {
            int a = 30, b = 0;
            int c = a/b;  // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by 0");
        }
    }
}
```

Output:

```
Can't divide a number by 0
```

- **Input Mismatch Exeption:**

```
 class Mismatch
{
    public static void main(String args[])
    {
          System.out.println("Enter the number of cars you own: ");
          Scanner input = new Scanner(System.in);
          try {
               int numberOfCars = input.nextInt();
              }
          catch (InputMismatchException e) {
               System.out.println("Enter an integer");
              }

    }
}
```

Output:

```
"Enter an integer" If user enters a number with decimal point
```

- **NullPointer Exception**

```
//Java program to demonstrate NullPointerException
class NullPointer_Demo
{
    public static void main(String args[])
    {
        try {
            String a = null; //null value
            System.out.println(a.charAt(0));
        } catch(NullPointerException e) {
            System.out.println("NullPointerException..");
        }
    }
}
```

Output:

```
NullPointerException..
```

- **StringIndexOutOfBound Exception**

```
// Java program to demonstrate StringIndexOutOfBoundsException
class StringIndexOutOfBound_Demo
{
    public static void main(String args[])
    {
        try {
            String a = "This is like chipping "; // length is 22
            char c = a.charAt(24); // accessing 25th element
            System.out.println(c);
        }
        catch(StringIndexOutOfBoundsException e) {
            System.out.println("StringIndexOutOfBoundsException");
        }
    }
}
```

Output:

```
StringIndexOutOfBoundsException
```

- **FileNotFound Exception**

```
//Java program to demonstrate FileNotFoundException
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
 class File_notFound_Demo {

    public static void main(String args[])  {
        try {
```

```
            // Following file does not exist
            File file = new File("E://file.txt");

            FileReader fr = new FileReader(file);
        } catch (FileNotFoundException e) {
           System.out.println("File does not exist");
        }
    }
}
```

Output:

```
File does not exist
```

- **NumberFormat Exception**

```
// Java program to demonstrate NumberFormatException
class NumberFormat_Demo
{
    public static void main(String args[])
    {
        try {
            // "akki" is not a number
            int num = Integer.parseInt ("akki") ;

            System.out.println(num);
        } catch(NumberFormatException e) {
            System.out.println("Number format exception");
        }
    }
}
```

Output:

```
Number format exception
```

- **ArrayIndexOutOfBounds Exception**

```
// Java program to demonstrate ArrayIndexOutOfBoundException
class ArrayIndexOutOfBound_Demo
{
    public static void main(String args[])
    {
        try{
            int a[] = new int[5];
            a[6] = 9; // accessing 7th element in an array of
                      // size 5
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println ("Array Index is Out Of Bounds");
        }
    }
}
```

Output:

```
Array Index is Out Of Bounds
```

User-Defined Exceptions

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, user can also create exceptions which are called ‘user-defined Exceptions’. Following steps are followed for the creation of user-defined Exception.

- The user should create an exception class as a subclass of Exception class. Since all the exceptions are subclasses of Exception class, the user should also make his class a subclass of it. This is done as:
-
```
class UserException extends Exception
```
- We can write a default constructor in his own exception class.

```
UserException(){}
```
- We can also create a parameterized constructor with a string as a parameter. We can use this to store exception details. We can call super class(Exception) constructor from this and send the string there.

```
UserException(String str)
{
   super(str);
}
```
- To raise exception of user-defined type, we need to create an object to his exception class and throw it using throw clause, as:

```
UserException myException = new MyException(“Exception details”);
throw myException;
```

```
class UserException extends Exception
{

    // default constructor
    UserException() {    }

    // parametrized constructor
    UserException(String str) { super(str); }

    // write main()
    public static void main(String[] args)
    {
        try  {

                if (GPA < 0.0)
```

```
                    {
                        UserException myException =
                           new UserException("GPA less than zero");
                        throw myException;
                    }
                }
            } //end of try

            catch (UserException e) {
                e.printStackTrace();
            }
        }
    }
```

Exception Propagation in Unchecked Exceptions

When an exception happens, Propagation is a process in which the exception is being dropped from to the top to the bottom of the stack. If not caught once, the exception again drops down to the previous method and so on until it gets caught or until it reach the very bottom of the call stack. This is called exception propagation and this happens in case of Unchecked Exceptions.
In the example below, 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.

Note : By default, Unchecked Exceptions are forwarded in calling chain (propagated).

```
// Java program to illustrate
// unchecked exception propagation
// without using throws keyword
class Simple {
    void m()
    {
        int data = 50 / 0; // unchecked exception occurred
        // exception propagated to n()
    }

    void n()
    {
        m();
        // exception propagated to p()
    }

    void p()
    {
        try {
            n(); // exception handled
        }
```

```
        catch (Exception e) {
            System.out.println("Exception handled");
        }
    }

    public static void main(String args[])
    {
        Simple obj = new Simple();
        obj.p();
        System.out.println("Normal flow...");
    }
}
```

Output:

```
Exception handled

Normal flow...
```

Exception Propagation in Checked Exceptions

Unlike Unchecked Exceptions, the propagation of exception **does not happen** in case of Checked Exception and its mandatory to use throw keyword here. Only unchecked exceptions are propagated. **Checked exceptions throw compilation error.**
In example below, If we omit the throws keyword from the m() and n() functions, the compiler will generate compile time error. Because unlike in the case of unchecked exceptions, the checked exceptions cannot propagate without using throws keyword.

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

```
// Java program to illustrate exception propagation
// in checked exceptions  and it can be propagated
// by throws keyword ONLY
import java.io.IOException;
class Simple {

    // exception propagated to n()
    void m() throws IOException
    {
        // checked exception occurred
        throw new IOException("device error");
    }

    // exception propagated to p()
    void n() throws IOException
    {
        m();
    }
    void p()
    {
        try {

            // exception handled
```

```
            n();
        }
        catch (Exception e) {
            System.out.println("exception handled");
        }
    }

    public static void main(String args[])
    {
        Simple obj = new Simple();
        obj.p();
        System.out.println("normal flow...");
    }
}
```

Output:

```
exception handled

normal flow...
```