## UNIT III EXCEPTION HANDLING AND I/O 9

Exceptions - exception hierarchy - throwing and catching exceptions - built-in exceptions, creating own exceptions, Stack Trace Elements. Input / Output Basics - Streams - Byte streams and Character streams - Reading and Writing Console - Reading and Writing Files

### 3.1 EXCEPTIONS

A Java exception is an object that finds an exceptional condition occurs from a piece of code. An exception object is created and thrown to the method from the code where an exception is found.

**Types of Exceptions**

There are two types of exceptions

1. Predefined Exceptions-The Exceptions which are predefined are called predefined exceptions
2. Userdefined Exceptions- The Exceptions which are defined by the user are called userdefined exceptions

**Purpose of exception handling**

The main purpose of exception handling mechanism is used to detect and report an "exceptional circumstance" so that necessary action can be taken. It performs the following tasks

1. Find the problem(Hit the exception)
2. Inform that an error occurred(throw the exception).
3. Receive the error information(Catch the exception)
4. Take corrective actions(Handle the exception)

Java exception handling is managed via five keywords.

1. try
2. catch
3. throw
4. throws and
5. finally

**1.try block:**

Program statements that need to be monitored for exceptions are placed within a try block.

**2.catch block:**

If an exception occurs within a try block, it is thrown to the catch block. This block can catch this exception and handle it

**3.throw:**

Some exceptions are automatically thrown by the Java run time system. To throw the exceptions manually, we can use the keyword throw.

**4.throws:**
Any exception that is thrown out of a method is specified as such by a throws clause.

**5.finally:**
finally blocks contains any code that need to be executed after a try block completes .

This is the general form of an exception-handling block:

```
try
{
// block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
// exception handler for ExceptionType2
}
// ...
finally
{
// block of code to be executed after try block ends
}
```

**ExcetptionType is the type of exception that has occurred. exOb is an exception object.**

## 3.2 EXCEPTION HIERARCHY

Throwable is a superclass for all exception types. Thus, Throwable is at top of the exception hierarchy.
There are two subclasses under Throwable class.

1. Exception
2. Error

**1 . Exception:**
This class is used for exceptional conditions that user programs should catch. We can also subclass this class to create own custom exception types. The important subclass of this class is **RuntimeException**.
**RuntimeException**

Exceptions of this type are automatically defined for the programs that you write and include thing such as division by zero and invalid array indexing.

**2 .Error**

The another subclass is by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.

**Eg: Stack overflow is an example of such an error.**

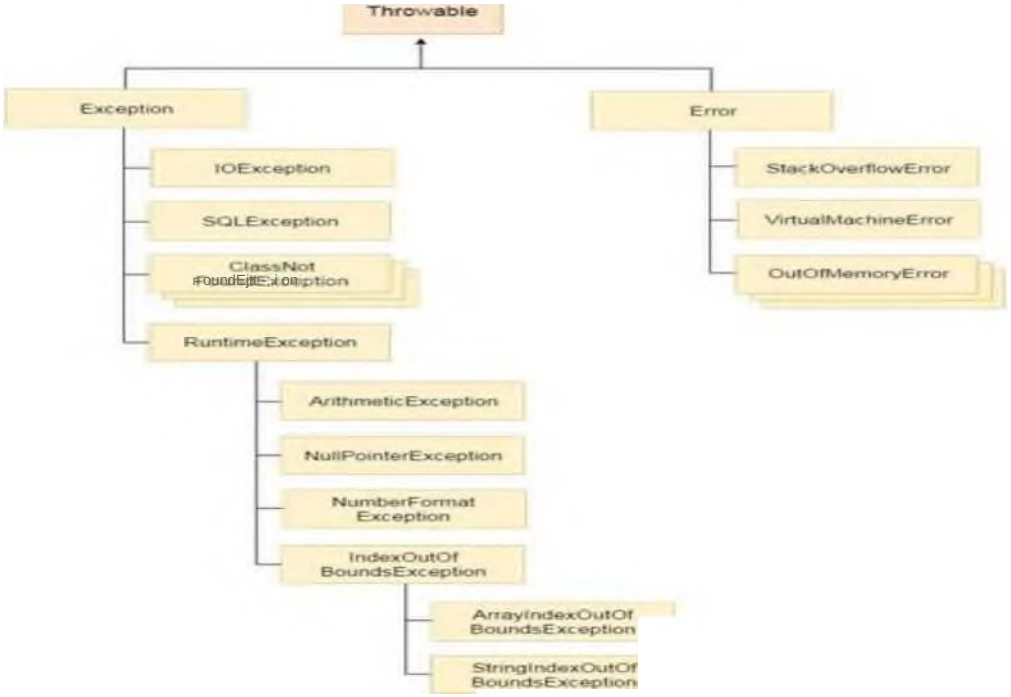


**Figure 3.1 Exception Hierarchy**

## 3.3 THROWING AND CATCHING EXCEPTIONS

### 3.3.1 Using try and catch

The try block allows us to fix the errors. Catch block prevents the program from automatically terminating. To handle a run-time error, enclose the code to be monitored inside a try block. After the try block, include a catch block that specifies the exception type that needs to be caught.

**Syntax:**

```
try
{
statement;
}
catch(Exception-type exOb)
{
statement;
}
```

The try block can have one or more statements that would generate an exception. If one statement of the try block generates an exception, the remaining statements in the block are not be executed and execution jumps to the catch that is placed next to the try block.

**Example Program:**

```
class Ex1
{
public static void main(String args[])
{
int a,b;
try
{
// monitor a block of code.
a = 0;
b = 42 / a;
System.out.println("This will not be printed.");
}
catch(ArithmeticException e)
{
// catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
```

```
}
}
```

**Output:**
Division by zero.
After catch statement.

**Example Program:**

```
class Ex2
{
public static void main(String[] args)
{
int numbers[] = { 1, 2, 3, 4, 5 };
try
{
for (int c = 1; c <= 5; c++)
{
System.out.println(numbers[c]);
}
}
catch (Exception e)
{
System.out.println(e);
}
}
}
```

**Output:**

```
2
3
4
5
java.lang.ArrayIndexOutOfBoundsException: 5
```

### 3.3.2 Multiple catch Clauses

In some situation, more than one exception can occur by a single piece of code. To handle this situation, we can use two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch block is executed in order, and the first one whose type matches that exception is executed. After one catch block executes, the others are bypassed, and continues after the try/catch block.

**Syntax:**

```
try
{
// block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
// exception handler for ExceptionType2
}
```

**Example Program:**

```
class Ex3
{
public static void main(String args[])
{
try
{
int a[]=new int[5];
a[5]=30/0;
}
catch(ArithmeticException e)
{
System.out.println("task1 is completed");
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("task 2 completed");
}
catch(Exception e)
{
System.out.println("common task completed");
}
System.out.println("rest of the code...");
}
}
```

**Output:**

```
task1 completed
rest of the code...
```

**Example Program:**

```
class Ex4
{
public static void main(String args[])
{
try
{
int s = args.length;
System.out.println("s = " + s);
int b = 42 / s;
int c[] = { 1 };
c[42] = 99;
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
```

**Output:**

```
s = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

java Ex4 TestArg a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

This program will cause a division-by-zero exception if it is started with no command line arguments, since **s** will equal zero. It will survive the division if you provide a command line argument, setting **s** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

### 1.1.3 Nested try Statements

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch**

handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested **try** statements:

**Syntax:**

```
//Main try block
try
{
statement 1;
statement 2;
//try-catch block inside another try block try
{
statement 3;
statement 4;
//try-catch block inside nested try block
try
{
statement 5;
statement 6;
}
catch(Exception e2)
{
//Exception Message
}
}
catch(Exception e1)
{
//Exception Message
}
}
//Catch of Main(parent) try block
catch(Exception e3)
{
//Exception Message
}
```

**Example Program:**

```
class Nest
{
public static void main(String args[]){
//Parent try block
try
{
```

```
//Child try block1
try
{
System.out.println("Inside block1");
int b =45/0;
System.out.println(b);
}
catch(ArithmeticException e1)
{
System.out.println("Exception: e1");
}
//Child try block2
try
{
System.out.println("Inside block2");
int b =45/0;
System.out.println(b);
}
catch(ArrayIndexOutOfBoundsException e2)
{
System.out.println("Exception: e2");
}
System.out.println("Just other statement");
}
catch(ArithmeticException e3)
{
System.out.println("Arithmetic Exception");
System.out.println("Inside parent try catch block");
}
catch(ArrayIndexOutOfBoundsException e4)
{
System.out.println("ArrayIndexOutOfBoundsException");
System.out.println("Inside parent try catch block");
}
catch(Exception e5)
{
System.out.println("Exception");
System.out.println("Inside parent try catch block");
}
System.out.println("Next statement..");
}
}
```

**Output:**

Inside block1

```
Exception: e1
Inside block2
Arithmetic Exception
Inside parent try catch block
Next statement..
```

### 1.1.4 throw statement

To throw an exception explicitly, we can use **throw** keyword. **Syntax:**

```
throw new exception_class("error message");
```

**For example:**

```
throw new ArithmeticException("dividing a number by 5 is not allowed in this program");
```

**Example Program:**

```
class ThrowDemo
{
static void demoproc()
{
try
{
throw new NullPointerException("demo");
}
catch(NullPointerException e)
{
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[])
{
try
{
demoproc();
}
catch(NullPointerException e)
{
System.out.println("Recaught: " + e);
}
}
}
```

**output:**

```
Caught inside demoproc.
```

Recaught: java.lang.NullPointerException: demo

### 1.1.5 throws clause

Using throws clause, We can list the types of exceptions that a method might throw.The exceptions which are thrown in a method might be using **throws** clause. If they are not, a compile-time error will result.

**Syntax:**

**type method-name(parameter-list) throws exception-list**
**{**
**// body of method**
**}**

**Exception-list is a comma-separated number of exceptions that a method can throw.**

**Example Program:**

```
class ThrowsDemo
{
static void throwOne() throws IllegalAccessException
{
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[])
{
try
{
throwOne();
}
catch (IllegalAccessException e)
{
System.out.println("Caught " + e);
}
}
}
```

**Output:**

inside throwOne
caught java.lang.IllegalAccessException: demo

### 1.1.6 finally

finally creates a block of code that is to be executed after a try/catch block has

completed its execution.The finally block will execute if an exception is thrown or not thrown. The finally clause is optional.Each try block requires either one catch or a finally clause**.**

**Syntax:**

```
try
{
   //Statements that may cause an exception
}
catch
{
  //Handling exception
}
finally
{
  //Statements to be executed
}
```

**Example Program:**

```
// Demonstrate finally.
class FinallyDemo
{
// Through an exception out of the method. static void procA()
{
try
{
System.out.println("inside procA");
throw new RuntimeException("demo");
}
finally
{
System.out.println("procA's finally");
}
}
// Return from within a try block.
static void procB()
{
try
{
System.out.println("inside procB");
return;
}
finally
{
```

```
System.out.println("procB's finally");
}
}
// Execute a try block normally.
static void procC()
{
try
{
System.out.println("inside procC");
}
finally
{
System.out.println("procC's finally");
}
}
public static void main(String args[])
{
try
{
procA();
}
catch (Exception e)
{
System.out.println("Exception caught");
}
procB();
procC();
}
}
```

**Output:**

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```