3.1: EXCEPTION HANDLING BASICS

Definition:

An Exception is an event that occurs during program execution which disrupts the normal flow of a program. It is an object which is thrown at runtime.

Occurrence of any kind of exception in java applications may result in an abrupt termination of the JVM or simply the JVM crashes.

In Java, an exception is an **object** that contains:

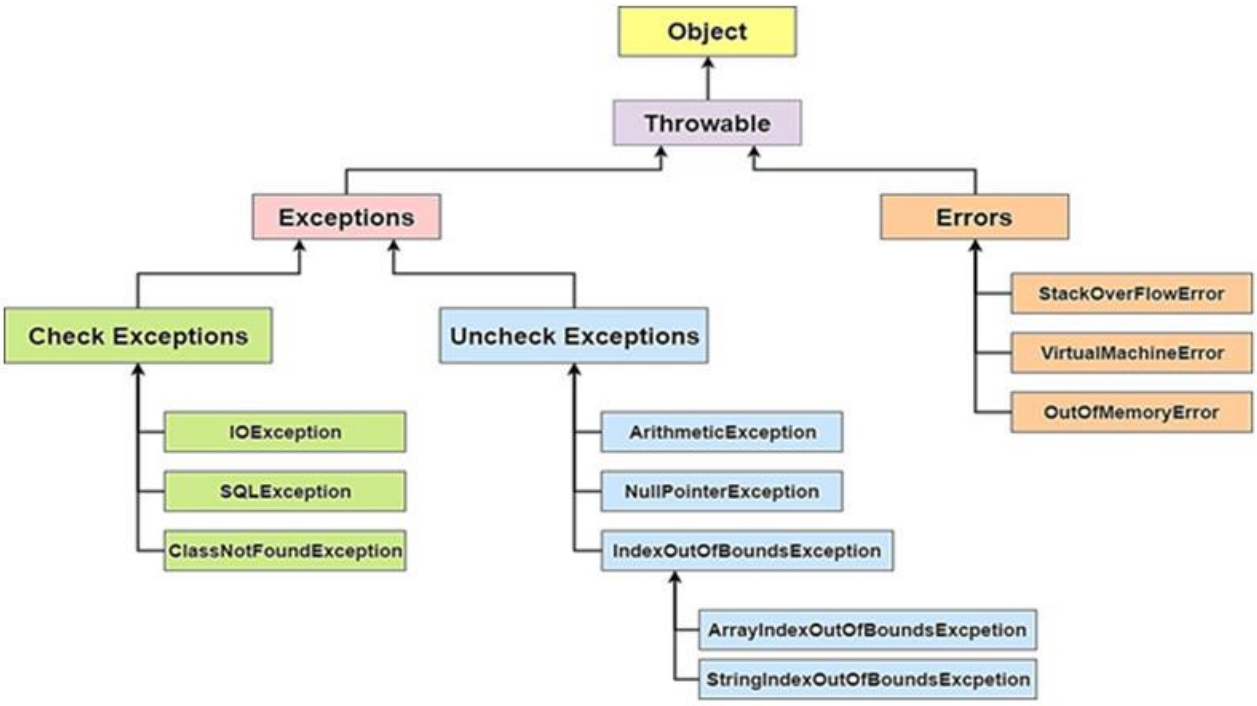
- Information about the error including its type
- The state of the program when the error occurred
- Optionally, other custom information

3.1.1: Exception Hierarchy

All exceptions and errors extend from a common java.lang.Throwable parent class.

The **Throwable** class is further divided into two classes:

1. **Exceptions** and
2. **Errors.**



Exceptions: Exceptions represents errors in the Java application program, written by the user. Because the error is in the program, exceptions are expected to be handled, either

- Try to recover it if possible
- Minimally, enact a safe and informative shutdown.

Sometimes it also happens that the exception could not be caught and the program may get terminated. Eg. **ArithmeticException**

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.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Errors: Errors represent internal errors of the Java run-time system which could not be handled easily. Eg. **OutOfMemoryError.**

DIFFERENCE BETWEEN EXCEPTION AND ERROR:

S.No.	Exception	Error
1.	Exceptions can be recovered	Errors cannot be recovered
2.	Exceptions are of type **java.lang.Exception**	Errors are of type **java.lang.Error**
3.	Exceptions can be classified into two types: a) Checked Exceptions b) Unchecked Exceptions	There is no such classification for errors. Errors are always unchecked.
4.	In case of Checked Exceptions, compiler will have knowledge of checked exceptions and force to keep try…catch block. Unchecked Exceptions are not known to compiler because they occur at run time.	In case of Errors, compiler won't have knowledge of errors. Because they happen at run time.
5.	Exceptions are mainly caused by the application itself.	Errors are mostly caused by the environment in which application is running.
6.	Examples: **Checked Exceptions:** SQLException, IOException **Unchecked Exceptions:** ArrayIndexOutOfBoundsException, NullPointerException	Examples: Java.lang.StackOverFlowError, java.lang.OutOfMemoryError

3.1.2: Exception Handling

What is exception handling?

Exception Handling is a mechanism to handle runtime errors, such as ClassNotFoundException, IOException, SQLException, RemoteException etc. by taking the necessary actions, so that normal flow of the application can be maintained.

Advantage of using Exceptions:

- Maintains the normal flow of execution of the application.
- Exceptions separate error handling code from regular code.
  - Benefit: Cleaner algorithms, less clutter
- Meaningful Error reporting.
- Exceptions standardize error handling.

JAVA EXCEPTION HANDLING KEYWORDS

Exception handling in java is managed using the following five keywords:

S.No.	Keyword	Description
1	try	A block of code that is to be monitored for exception.
2	catch	The catch block handles the specific type of exception along with the try block. For each corresponding try block there exists the catch block.
3	finally	It specifies the code that must be executed even though exception may or may not occur.
4	throw	This keyword is used to explicitly throw specific exception from the program code.
5	throws	It specifies the exceptions that can be thrown by a particular method.

- **try Block:**
  - ✓ The java code that might throw an exception is enclosed in try block. It must be used within the method and must be followed by either catch or finally block.
  - ✓ If an exception is generated within the try block, the remaining statements in the try block are not executed.

- **catch Block:**
  - ✓ Exceptions thrown during execution of the try block can be caught and handled in a catch block.
  - ✓ On exit from a catch block, normal execution continues and the finally block is executed.

- **finally Block:**

A finally block is always executed, regardless of the cause of exit from the try block, or whether any catch block was executed.

- ✓ Generally finally block is used for freeing resources, cleaning up, closing connections etc.
- ✓ Even though there is any exception in the try block, the statements assured by **finally** block are sure to execute.
- ✓ **Rule:**
  - **For each try block there can be zero or more catch blocks, but only one finally block.**
  - **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).**

The try-catch-finally structure(Syntax):

```
try {
   // Code block
}
catch (ExceptionType1 e1) {
   // Handle ExceptionType1 exceptions
}
catch (ExceptionType2 e2)  {
   // Handle ExceptionType2 exceptions
}
// ...
finally  {
   // Code always executed after the
   // try and any catch block
}
```

Rules for try, catch and finally Blocks:

1) Statements that might generate an exception are placed in a try block.
2) Not all statements in the try block will execute; the execution is interrupted if an exception occurs
3) For each try block there can be zero or more catch blocks, but only one finally block.
4) The try block is followed by
   i. one or more catch blocks
   ii. or, if a try block has no catch block, then it must have the finally block
5) A try block must be followed by either at least one catch block or one finally block.
6) A catch block specifies the type of exception it can catch. It contains the code known as exception handler

7) The catch blocks and finally block must always appear in conjunction with a try block.
8) The order of exception handlers in the catch block must be from the most specific exception

Program without Exception handling: (Default exception handler):

```
class Simple
{
 public static void main(String args[])
{
    int data=50/0;
    System.out.println("rest of the code...");
}
}
```

Output:

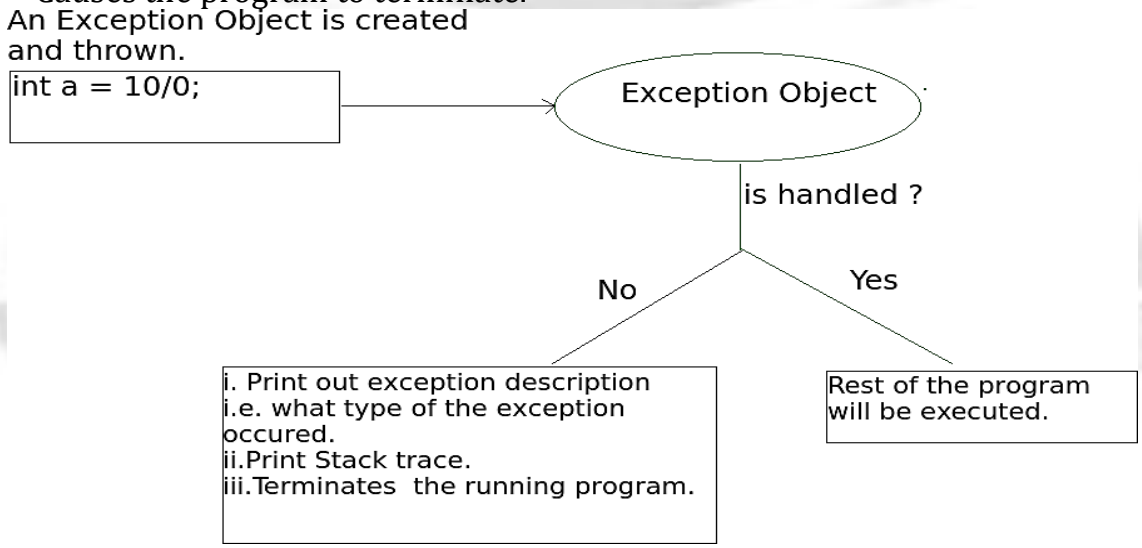
Exception in thread main java.lang.ArithmeticException:/ by zero

As displayed in the above example, rest of the code is not executed i.e. rest of the code... statement is not printed.

Program Explanation:

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.



Example:

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

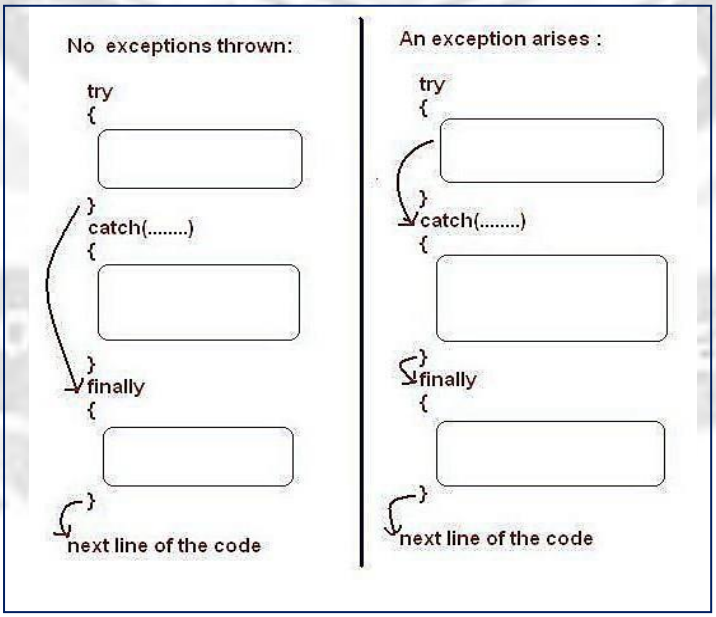
```
{



 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:

java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...



3.2: Multiple catch blocks

Multiple catch is used to handle many different kind of exceptions that may be generated while running the program. i.e more than one catch clause in a single try block can be used.

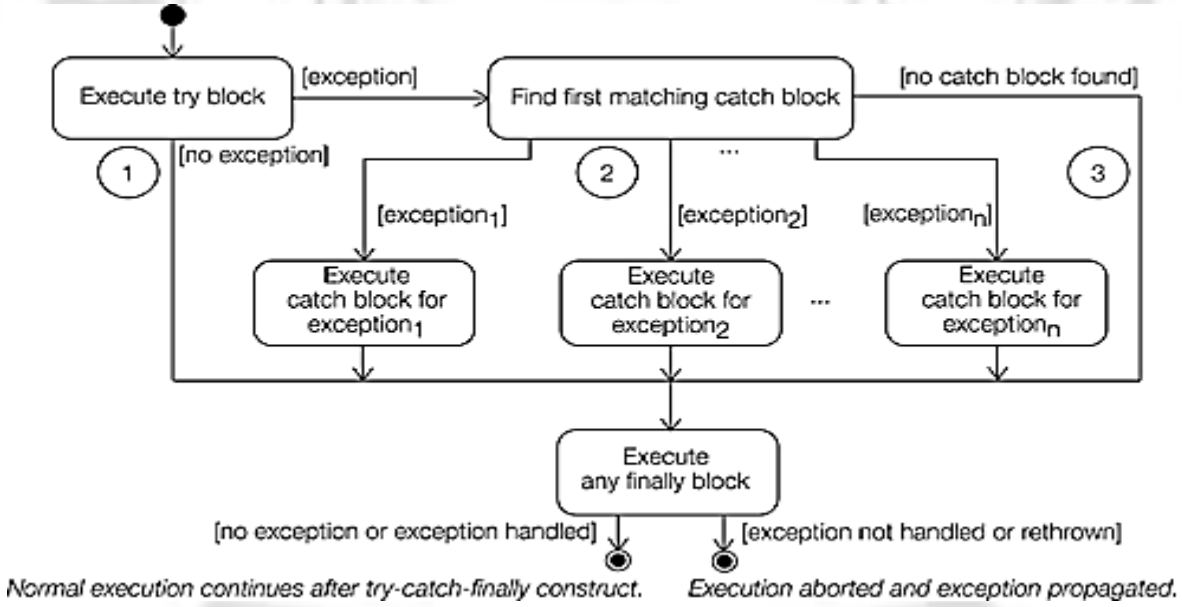
Rules:

- *At a time only one Exception can occur 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 forArithmetic Exception must come before catch for Exception.*

Syntax:

```
try {
   // Code block
}
catch (ExceptionType1 e1) {
   // Handle ExceptionType1 exceptions
}
catch (ExceptionType2 e2)  {
   // Handle ExceptionType2 exceptions
}
```



Example:

```
public class MultipleCatchBlock2 {
public static void main(String[] args) {
  try
{
    int a[]= {1,5,10,15,16};
    System.out.println("a[1] = "+a[1]);
    System.out.println("a[2]/a[3] = "+a[2]/a[3]);
    System.out.println("a[5] = "+a[5]);
  }

  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:

a[1] = 5
a[2]/a[3] = 0
ArrayIndexOutOfBounds Exception occurs
rest of the code

3.3: Nested Try Block

Definition: try block within a try block is known as nested try block.

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.
- ✓ If an inner try statement does not have a matching catch statement for a particular exception, the control is transferred to the next try statement's catch handlers that for a matching catch statement.
- ✓ If none of the catch statement match, then the Java run-time system will handle the exception.

Example:

```
class NestedExcep
{
 public static void main(String[] args)
 {
  try
  {
   int arr[]={1,5,4,10};
   try
   {
```

Syntax:

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

```
    int x=arr[3]/arr[1];
    System.out.println("Quotient = "+x);
   }
   catch(ArithmeticException ae)
   {
    System.out.println("divide by zero");
   }



    arr[4]=3;
   }
   catch(ArrayIndexOutOfBoundsException e)
   {
    System.out.println("array index out of bound exception");
   }
   System.out.println("...End of Program...");
  }
 }
```

Output:

Quotient = 2
array index out of bound exception
...End of Program...

3.4: THROWING AND CATCHING EXCEPTIONS

Before catching an exception, it is must to throw an exception first. This means that there should be a code somewhere in the program that could catch exception thrown in the try block.

An exception can be thrown explicitly

1. Using the **throw** statement
2. Using the **throws** statement

1: Using the throw statement

- ✓ A program can explicitly throw an exception using the throw statement besides the implicit exception thrown.
- ✓ We can throw either checked, uncheked exceptions or custom(user defined) exceptions
- ✓ When **throw** statement is called:

1) It causes the termination of the normal flow of control of the program code and stops the execution of the subsequent statements.
2) It transfers the control to the nearest catch block handling the type of exception object thrown
3) If no such catch block exists, then the program terminates.

The general format of the **throw** statement is as follows:

throw <exception reference>;

The Exception reference must be of type Throwable class or one of its subclasses. A detail

message can be passed to the constructor when the exception object is created.

Example:

```
1)  public class ThrowDemo
2)  {
3)  static void validate(int age)
4)  {
5)  if(age<18)
6)  throw new ArithmeticException("not valid");
7)  else
8)  System.out.println("welcome to vote");
9)  }
10) public static void main(String args[])
11) {
12) validate(13);
13) System.out.println("rest of the code...");
14) }
15) }
```

Output:

Exception in thread "main" java.lang.ArithmeticException: not valid
at ThrowDemo.validate(ThrowDemo.java:6)
at ThrowDemo.main(ThrowDemo.java:12)

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.

2: Using throws keyword:

- ✓ The **throws** statement is used by a method to specify the types of exceptions the method throws.
- ✓ If a method is capable of raising an exception that it does not handle, the method must specify that the exception have to be handled by the calling method.
- ✓ This is done using the throws clause. The throws clause lists the types of exceptions that a method might throw.

Syntax:

Return-type method_name(arg_list) **throws** *exception_list*
{
// method body

```
}
```

Example:

```
1.  import java.util.Scanner;
2.  public class ThrowsDemo
3.  {
4.  static void divide(int num, int din) throws ArithmeticException
5.  {
6.  int result=num/din;
7.  System.out.println("Result : "+result);
8.  }
9.  public static void main(String args[])
10. {
11. int n,d;
12. Scanner in=new Scanner(System.in);
13. System.out.println("Enter the Numerator : ");
14. n=in.nextInt();
15. System.out.println("Enter the Denominator : ");
16. d=in.nextInt();
17. try
18. {
19. divide(n,d);
20. }
21. catch(Exception e)
22. {
23. System.out.println(" Can't Handle : divide by zero ERROR");
24. }
25. System.out.println(" ** Continue with rest of the code ** ");
26. }
27. }
```

Output:

Enter the Numerator :
4

Enter the Denominator :
0

Can't Handle : divide by zero ERROR
**** Continue with rest of the code ****
Enter the Numerator :
6

Enter the Denominator :
2
Result : 3
**** Continue with rest of the code ****

Difference between throw and throws:

throw keyword	throws keyword
1) throw is used to explicitly throw an exception.	throws is used to declare an exception.
2) checked exception cannot be propagated without throws.	checked exception can be propagated with throws.
3) throw is followed by an instance.	throws is followed by class.
4) throw is used within the method.	throws is used with the method signature.
5)You cannot throw multiple exception	You can declare multiple exception e.g. **public void method()throws IOException,SQLException.**

3.5: Types of Exceptions