SE Semester-III

Object Oriented Programming with Java

Unit-4

Exception Handling & JAVAFX UI controls and multimedia:

Subject Teacher : Dr. K. V. Metre

Contents :

- Introduction to Exception Handling
- Exception vs Error
- Use of try, catch, throw, throws, and finally
- Built-in Exceptions
- Custom Exceptions
- Throwable Class
- A

Introduction

- Rarely does the program run successful at its very first attempt.
- It is common to make the mistakes while developing as well as typing a program.
- A mistake might lead to an error causing to program to produce unexpected results.
- Errors are the wrongs that can make a program go wrong.
- An error may produce an incorrect result or may terminate the execution of the program abruptly or even may cause the system to crash.

Types of Errors

- **Errors may be broadly classified into two types.**
- Compile time Errors
- Run Time Errors

- **Syntax Errors/ Compile time errors:**
  Syntax errors arise because the rules of the language have not been followed. They are detected by the compiler.
- **Runtime errors** occur while the program is running if the environment detects an operation that is impossible to carry out.
- **Logical errors** occur when a program doesn't perform the way it was intended to.

- **Compile Time Errors**

```
/* This Program Contains an Error
class Error1
{
      public static void main(String args[])
       {
            System.out.println(“Hello Java”)
           }
}
```

```
Error1.java :7: ‘ : ’ expected
System.out.println(“Hello java”)
^1 error
```

Compile Time Errors

Most compile-time errors are due to typing mistakes.

The most common problems are :

- Missing semicolon
- Missing (or mismatch of) brackets in classes and methods.
- Misspelling of identifiers and keywords
- Missing double quotes in strings.
- Use of undeclared variables
- Incompatible types in assignments /initialization
- Bad reference to objects
- Use of = in place of ==operator
- And so on

Run Time Errors

- Several time program may compile successfully and compiler creates the .class file of the program.

- At the time of running the program, it shows the error and that type of error called run time error.

- Such program may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow.

Run Time Errors

Most Common run time errors are :

- Dividing an integer by Zero
- Accessing an element that is out of the bounds of an array.
- Trying to store a value into an array of an incompatible class or type.
- Trying to cast an instance of class to one of its subclasses.
- Passing a parameter that is not a valid range or value for a method.
- Attempting to use a negative size for an array.
- And many more.

- Run Time Errors

```
class errors2
{
    public static void main(String args[])
    {
    int a=10,b=5,c=5;
    int x=a/(b-c);
            System.out.println("X= " +x)
    int y=a/(b+c);
            System.out.println("Y= " +y)
    }
}
```

Output :

```
Java.lang.ArithmeticException: / by zero
at Error2.main(Error2.java:8
```

What is an exception?

- Dictionary Meaning: Exception is an abnormal condition.
- **Definition**: Mechanism to handle runtime errors, ensuring the program's normal flow.
- An Exception is a condition that is caused by a run- time error in the program.

- When java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it.(i.e. informs us that error has occurred.)

What is an exception?

- An Exception is an unwanted event that interrupts the normal flow of the program.
- When an exception occurs program execution gets terminated.
- In such cases we get a system generated error message.
- The good thing about exceptions is that they can be handled in Java.
- By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

When an exception can occur?

Exception can occur at runtime (known as runtime exceptions) as well as at compile-time (known Compile-time exceptions).

Reasons for Exceptions

There can be several reasons for an exception.

For example, following situations can cause an exception -

- Opening a non-existing file,
- Network connection problem,
- Operands being manipulated are out of prescribed ranges,
- class file missing which was supposed to be loaded and so on.

Why to handle exception?

- If an exception is raised, which has not been handled by programmer then program execution can get terminated and system prints a non user friendly error message.

- Ex:-Take a look at the below system generated exceptionException in thread "main" java.lang.ArithmeticException: / by zero at ExceptionDemo.main(ExceptionDemo.java:5)

- For a novice user the above message won't be easy to understand.
- In order to let them know that what went wrong we use exception handling in java program.

- We handle such conditions and then prints a user friendly warning message to user.

- **Types of exceptions**
  There are two types of exceptions

1) Checked exceptions

2) Unchecked exceptions

Exception vs Error :

Exception: Indicates conditions that a program might want to catch.
Recoverable (e.g., IOException, SQLException).

Error : Indicates serious problems that a program should not try to catch.
Unrecoverable (e.g., OutOfMemoryError, StackOverflowError).

Checked exceptions

- All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not.

- If these exceptions are not handled/declared in the program, it will give compilation error.
- Examples of Checked Exceptions :- ClassNotFoundException, IllegalAccessException NoSuchFieldException EOFException etc.

Unchecked Exceptions

- Runtime Exceptions are also known as Unchecked Exceptions as the compiler do not check whether the programmer has handled them or not but it's the duty of the programmer to handle these exceptions and provide a safe exit.

- These exceptions need not be included in any method's throws list because compiler does not check to see if a method handles or throws these exceptions.

Examples of Unchecked Exceptions:-

- ArithmeticException
- ArrayIndexOutOfBoundsException
- NullPointerException
- NegativeArraySizeException etc.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur.

1) A scenario where ArithmeticException occurs

   If we divide any number by zero, there occurs an ArithmeticException.

```
int a = 50/0;        //ArithmeticException
```

2) A scenario where NullPointerException occurs

   If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s = null;

System.out.println(s.length());  //NullPointerException
```

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

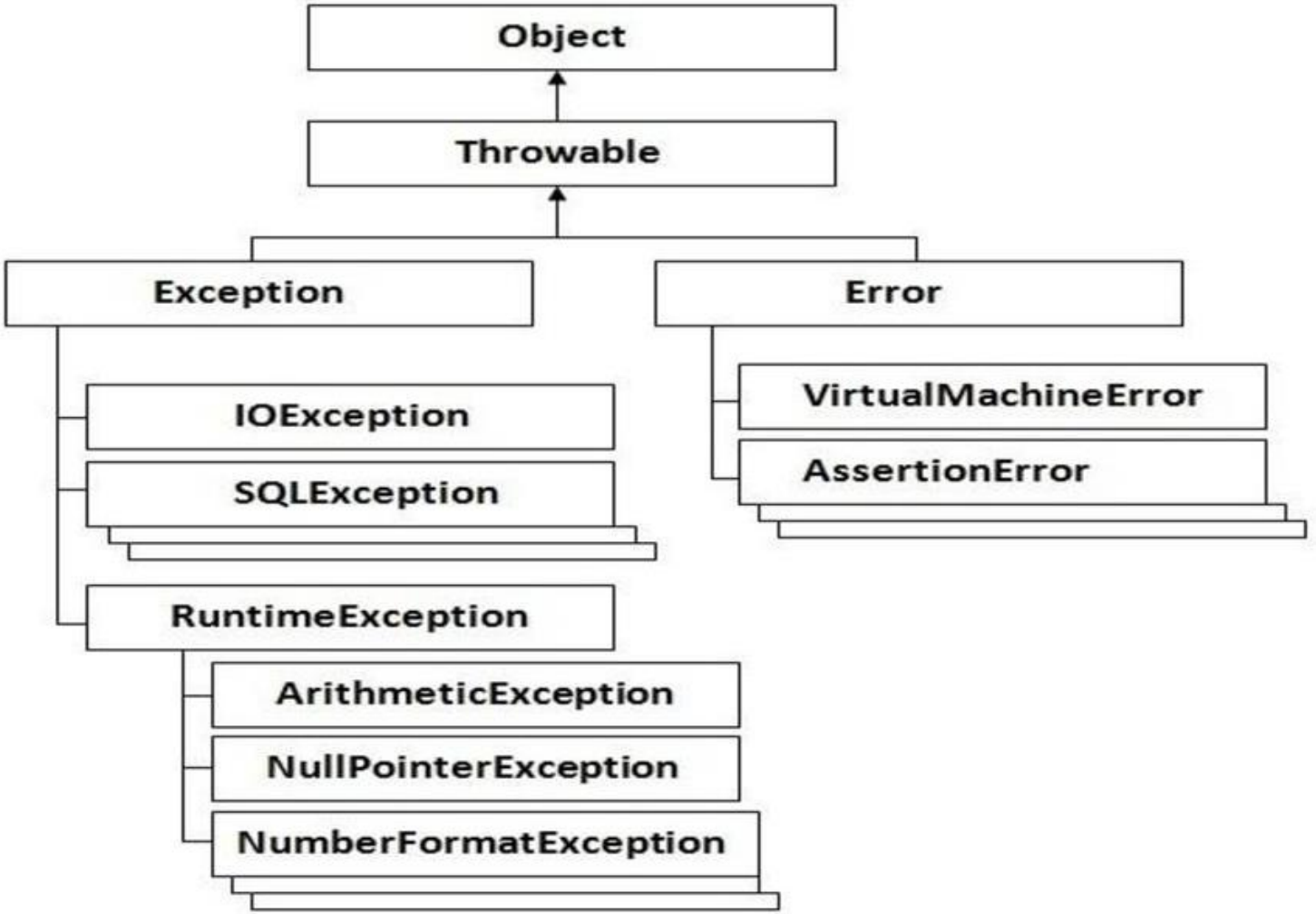
```
String s = “abc";
int i=Integer.parseInt(s); //NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Hierarchy of Exception classes



Exception Handling in Java:

- Java allows Exception handling mechanism to handle various exceptional conditions.

- When an exceptional condition occurs, an exception is thrown.
- For continuing the program execution, the user should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions.

- This task is known as Exception handling

Exception Handling in Java :

This mechanism consists of :

- Find the problem (Hit the Exception)
- Inform that an error has occurred(Throw the Exception)
- Receive the error Information(Catch the Exception)
- Take corrective actions(Handle the Exception)

- **Exception Handling in Java**

  In Java Exception handling is managed by 5 keywords:
- Try
- Catch
- Throw
- Throws
- finally

Exception Handling :Use of try and catch

- **try**: Block where exceptions may occur.

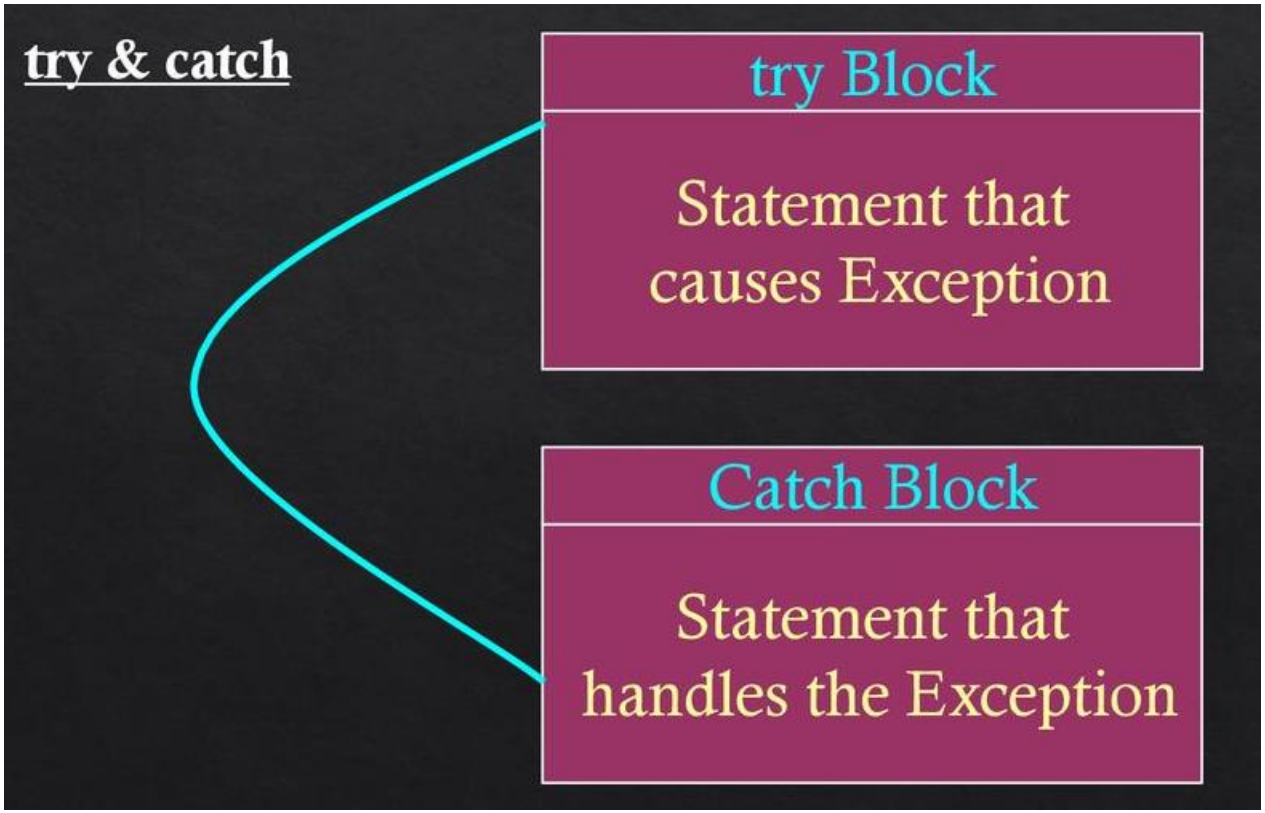
```
try {
      // Code that might throw an exception
 }
```

- **catch**: Block to handle the exception.

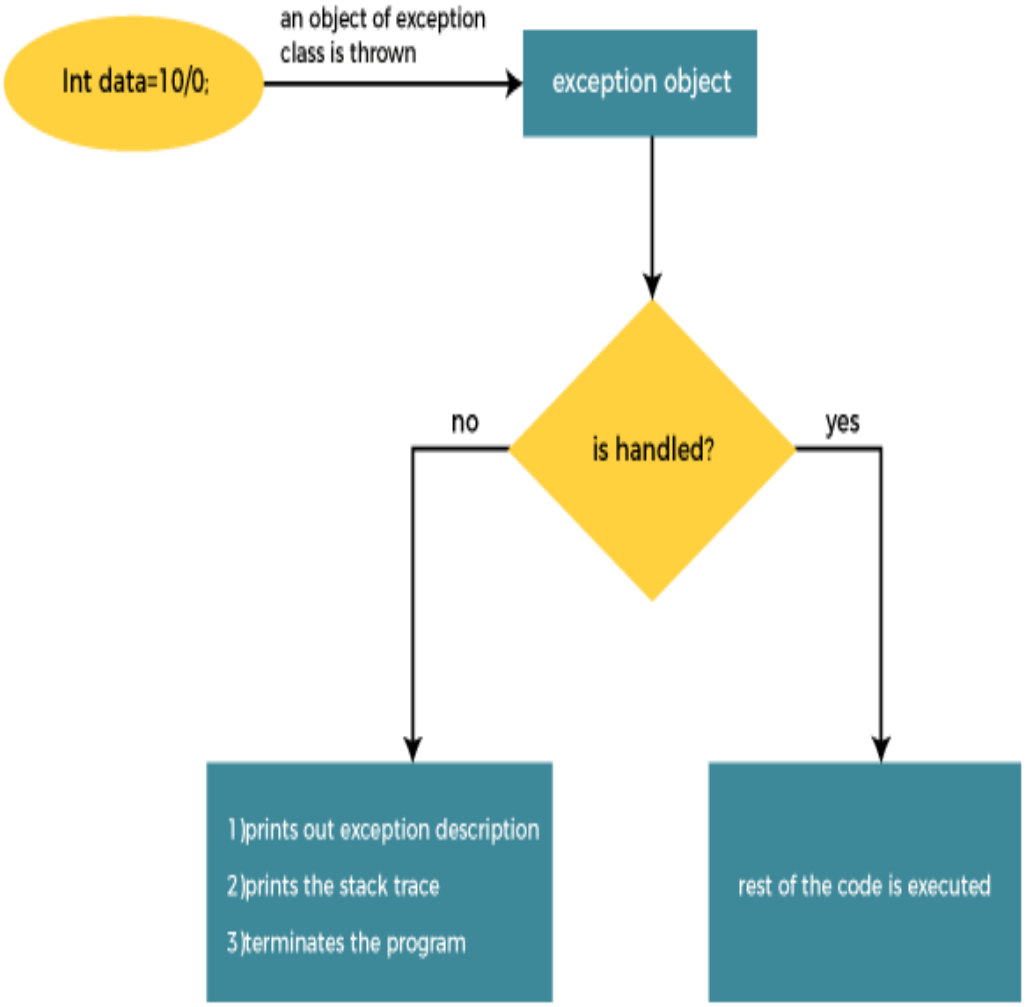
```
catch (ExceptionType e)
 {
     // Code to handle exception
    }
```

Syntax of Exception Handling Code

The basic concept of exception handling are throwing an exception and catching it.



- Internal Working of Java try-catch block



- 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.
- But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

```
try {
        // block of code to monitor for errors
      }
 catch (ExceptionType1 exOb)
  {
        // exception handler for Exception Type1
   }
 catch (ExceptionType2 exOb)
 {
    // exception handler for Exception Type2
 }

 // …
finally {
   // block of code to be executed before try block ends.
 }
```

`catch` clauses attempted in order; first match wins!

- Each **`catch`** handler can have only a single parameter
- Logical error to catch the same type in two different **`catch`** handlers following a single **`try`** block

Termination model of exception handling :

- **`try`** block *expires* when an exception occurs
  - Local variables in try block go out of scope
- Code within the matching catch handler executes
- Control resumes with the first statement after the last catch handler following the try block

- Java Exception Keywords
- Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
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 necessary 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 specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Program without exception handling:

```
public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=100/0;   //may throw exception

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

    }

}
```

Output:

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

the **rest of the code** is not executed (the **rest of the code** statement is not printed).

Program with exception handling:

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
                    //code that may raise exception
                    int data=100/0;
      }
catch(ArithmeticException e)
          {     System.out.println(“ Division by 0 error”);    }
   //rest code of the program
     System.out.println("rest of the code...");
  } }
```

Output:

Division by 0 error
rest of the code...

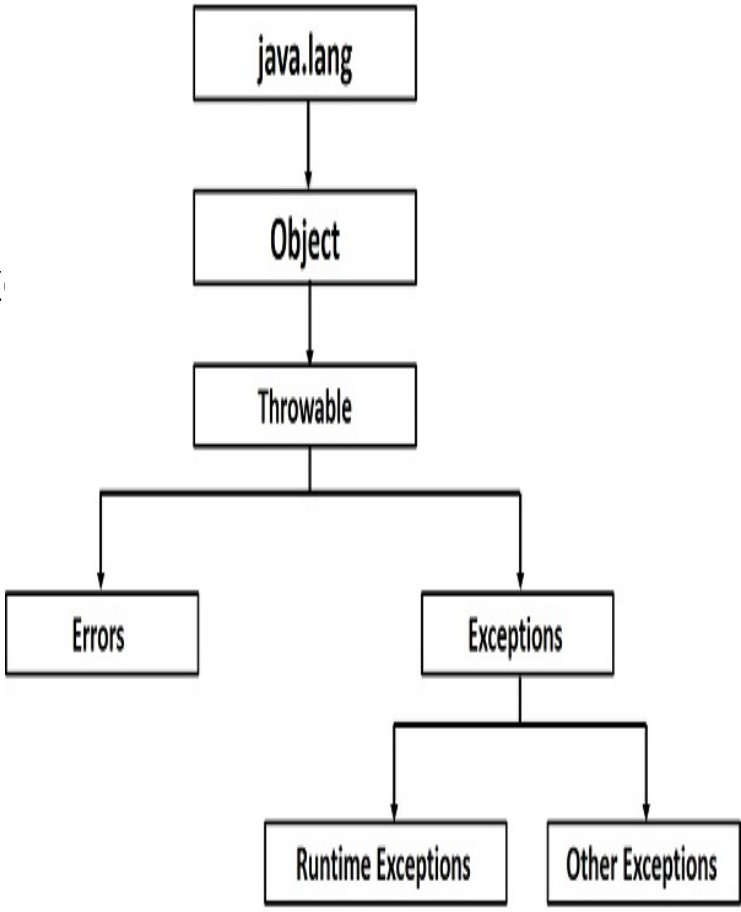
```
public class TryCatchExample3 {
    public static void main(String[] args) {
     try
     {
     int data=50/0; //may throw exception
            // if exception occurs, the remaining statement will not exceute

     System.out.println("rest of the code");
     }
        // handling the exception
     catch(ArithmeticException e)
     {
       System.out.println(“ Division by zero”);
     }
   }
}
```

Output:

Division by zero

Exception using the parent class exception.

```
public class TryCatchExample4 {
    public static void main(String[] args) {
      try  {
      int data=50/0; //may throw exception
      }
         // handling the exception by using Ex
      catch(Exception e)       {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }
}
```



Output:

java.lang.ArithmeticException: / by zero rest of the code

- **example to resolve the exception in a catch block.**

TryCatchExample6.java

```
public class TryCatchExample6 {

   public static void main(String[] args) {
      int i=50;
      int j=0;
      int data;
      try
      {
      data=i/j; //may throw exception
      }
         // handling the exception
      catch(Exception e)
      {
          // resolving the exception in catch block
         System.out.println(i/(j+2));
      }
   }
}
```

Output:

25

along with try block, we also enclose exception code in a catch block.

```
public class TryCatchExample7 {
    public static void main(String[] args) {
      try   {
            int data1=50/0; //may throw exception
        }
      catch(Exception e)    // handling the exception
      {             // generating the exception in catch block
         int data2=50/0; //may throw exception
        }
      System.out.println("rest of the code");
   }
}
```

Output:

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

- Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

- **In this example, we handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).**

```
public class TryCatchExample{
    public static void main(String[] args) {
      try   {
         int data=50/0; //may throw exception
       }
         // try to handle the ArithmeticException using ArrayIndexOutOfBoundsE
   xception
    catch(ArrayIndexOutOfBoundsException e)
     {
        System.out.println(e);
     }
     System.out.println("rest of the code");
  } }
```

Output:

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

Example to handle another unchecked exception.

```
public class TryCatchExample {
    public static void main(String[] args) {
     try   {
       int arr[]= {1,3,5,7};
        System.out.println(arr[10]); //may throw exception
        }
    catch(ArrayIndexOutOfBoundsException e)   // handling the array exception
     {
        System.out.println(e);
     }
     System.out.println("rest of the code");
  }      }
```

Output:

```
java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code
```

Multiple Catch Blocks

- A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following −

Syntax

```
try {
        // Protected code
         }
catch (ExceptionType1 e1)
        { // Catch block }
catch (ExceptionType2 e2)
        { // Catch block }
catch (ExceptionType3 e3)
        { // Catch block }
```

Nested try block

- In Java, using a try block inside another try block is permitted. It is called as nested try block.
- Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.
- the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

- **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.

```
//main try block
try
{
   statement 1;
   statement 2;
//try catch block within another try block

   try
   {
      statement 3;
      statement 4;
//try catch block within nested try block
      try
      {
         statement 5;
         statement 6;
   }
 catch(Exception e2)
      {
//exception message
      }

   }
   catch(Exception e1)
   {
    //exception message
   }
}
//catch block of parent (outer) try block

catch(Exception e3)
{
//exception message
}
....
```

```
public class NestedTryBlock{
 public static void main(String args[]){
//outer try block
 try {
 //inner try block 1
  try {
   System.out.println("going to divide by 0");

   int b =39/0;
  }
   //catch block of inner try block 1
   catch(ArithmeticException e)
   {
    System.out.println(" Division by ZERO
     Error");
   }
     //inner try block 2
   try {
   int a[]=new int[5];

   //assigning the value out of array bounds
    a[5]=4;
    }
   //catch block of inner try block 2
     catch(ArrayIndexOutOfBoundsException e
       )
     {
       System.out.println(" Array index is out of
       range");
     }
     System.out.println("other statement");

  }
  //catch block of outer try block
  catch(Exception e)
  {
   System.out.println("handled the exceptio
     n in outer catch");
  }

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

Output :

going to divide by 0
Division by ZERO Error
Array index is out of range
other statement
normal flow..

- When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

- If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

- try block within nested try block (inner try block 2) do not handle the exception. The control is then transferred to its parent try block (inner try block 1).
- If it does not handle the exception, then the control is transferred to the main try block (outer try block) where the appropriate catch block handles the exception.
- It is termed as nesting.

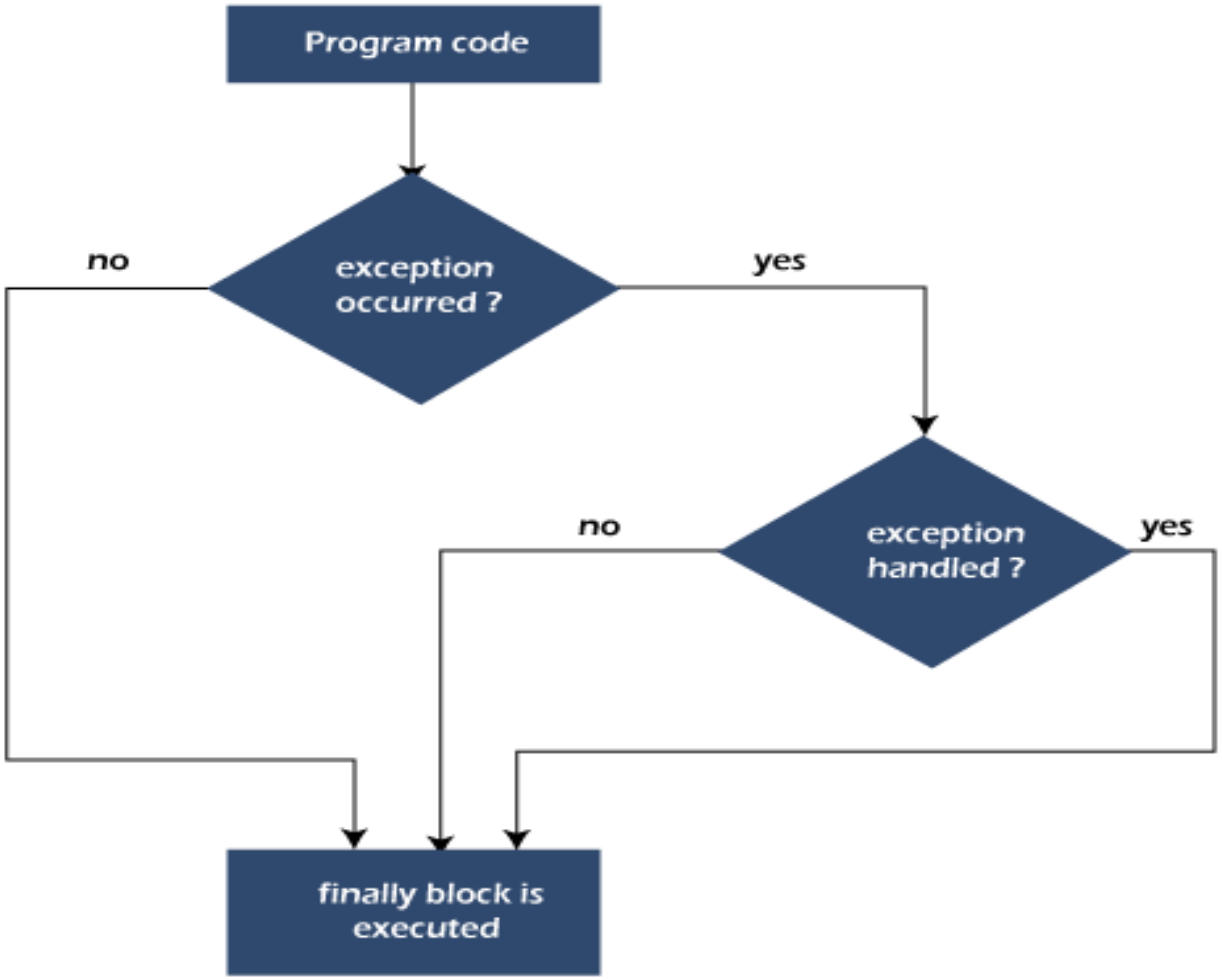
Java finally block :

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

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

- Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

- The finally block follows the try-catch block.



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

Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

Case 1: When an exception does not occur

The Java program does not throw any exception, and the finally block is executed after the try block.

```
class TestFinallyBlock {
  public static void main(String arg
    s[]){
  try{
//below code do not throw any ex
    ception
   int data=25/5;
   System.out.println(data);
  }
//catch won't be executed
  catch(ArithmeticException e){
System.out.println(e);
}
```

```
//executed regardless of exception occurred or not
 finally {
     System.out.println("finally block is al
     ways executed");
    }
System.out.println("rest of the code...")

  }
}
```

Output:

```
5
finally block is always executed
rest of the code...
```

Case 2: When an exception occur but not handled by the catch block

the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

```
public class TestFinallyBlock {
  public static void main(String args[]){
   try {
     System.out.println("Inside the try block");

    //below code throws divide by zero exception
     int data=25/0;
     System.out.println(data);
    }
    //cannot handle Arithmetic type exception
   //can only accept Null Pointer type exception
    catch(NullPointerException e){
     System.out.println(e);
    }
```

```
//executes regardless of exception occured or not

    finally {
     System.out.println("finally block is
                    always executed");
    }
     System.out.println("rest of the cod
e...");
    }     }
```

Output:

finally block is always executed

Exception in thread "main" java.lang.
ArithmeticException: / by zero at
TestFinallyBlock.main(TestFinallyBloc
k.java:5)

Case 3: When an exception occurs and is handled by the catch block

Example:

Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

```
public class TestFinallyBlock2{
    public static void main(String args[]){
     try {
       System.out.println("Inside try block");
//below code throws divide by zero exception
     int data=25/0;
     System.out.println(data);
    }
     //handles the Arithmetic Exception / Divide by zero
       exception
    catch(ArithmeticException e){
     System.out.println("Exception handled");
            System.out.println(e);
    }
//executes regardless of exception occured or n
     finally {
      System.out.println("finally block is a
      ways executed");
    }
      System.out.println("rest of the code
      .");
    }    }
```

Output :

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

- 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 the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

Java throw Exception :

- In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

Java throw keyword

- Java throw keyword is used to throw an exception explicitly.
- We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.
- We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section
- We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

- throw Instance i.e.,

  throw new exception_class("error message");

- example of throw IOException.

  throw new IOException("sorry device error");

- Where the Instance must be of type Throwable or subclass of Throwable.

Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

throw keyword Example

Example 1: Throwing Unchecked Exception

we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow {
  //function to check if person is eligible to vote or not
  public static void validate(int age) {
    if(age < 18) {
      //throw Arithmetic exception if not eligible to vote
        throw new ArithmeticException("Person is not eligible to vote");
    }
    else {
        System.out.println("Person is eligible to vote!!");
    }
  }
  //main method
   public static void main(String args[]){
      //calling the function
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

Output :

```
Exception in thread "main"
    java.lang.ArithmeticException:
    Person is not eligible to vote at
    TestThrow1.validate(TestThrow1.java:6) at
    TestThrow1.main(TestThrow1.java:15)
```

Throws Keywords:

- If a method does not handle an exception, the method must declare it using the **throws** keyword.
- The throws keyword appears at the end of a method's signature.
- You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.
- Try to understand the difference between throws and throw keywords, ***throws*** is used to postpone the handling of an exception and ***throw*** is used to invoke/call an exception explicitly.
- The following method declares that it throws a RemoteException

It provides information to the caller of the method about the exception.

```
return_type method_name() throws exception_class_name
{
    //method code
  }
```

```
import java.io.IOException;
class Test{
  void get() throws IOException
{
    throw new IOException("device error");//exception
  }
    void show(){
     try{
         get();
       }
   catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   Test obj=new Test();
   obj.show();
   System.out.println("normal flow...");
  }  }
```

Output:

```
exception handled
normal flow...
```

Built-in Exceptions :

- **Common Built-in Exceptions**:
  - NullPointerException
  - ArrayIndexOutOfBoundsException
  - ArithmeticException
  - IOException

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero
ArrayIndexOutOfBoundsException	Array index is out-of-bounds
ArrayStoreException	Assigning an element of an incompatible type
ClassCastException	Invalid cast
EnumConstantNotPresentException	Attempt made to use undefined enumeration value
IllegalArgumentException	Illegal argument used to invoke a method
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread
IllegalStateException	Environment or application is in incorrect state
IllegalThreadStateException	Requested operation not compatible with current thread state
IndexOutOfBoundsException	Some type of index is out-of-bounds
NegativeArraySizeException	Array created with a negative size

Exception	Meaning
NullPointerException	Invalid use of a null reference
NumberFormatException	Invalid conversion of a string to a numeric format
SecurityException	Attempt to violate security
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string
TypeNotPresentException	Type not fount
UnsupportedOperationException	An unsupported operation was encountered
ClassNotFoundException	Class not found
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface
IllegalAccessException	Access to a class is denied
InstantiationException	Attempt to create an object of an abstract class or interface
InterruptedException	One thread has been interrupted by another thread
NoSuchFieldException	A requested field does not exist
NoSuchMethodException	A requested method does not exist

Custom Exceptions

- **Why Custom Exceptions?**
  To handle application-specific issues in a more meaningful way.

- **How to Create a Custom Exception**:

```
class MyException extends Exception {
    public MyException(String message) {
     super(message); }
}
```

Custom Exceptions Example

```
class MyException extends Exception {
public MyException(String message) {
        super(message); }
 }
public class Test {
 public static void main(String[] args) {
try {
      throw new MyException("Custom Exception"); }
catch (MyException e) {
       System.out.println(e.getMessage()); }
} }
```

Output:

Custom Exception

Example 3: Throwing User-defined Exception

```
// class represents user-
    defined exception
class UserDefinedException exten
    ds Exception
{
   public UserDefinedException(St
    ring str)
   {
      // Calling constructor of pare
    nt Exception
      super(str);
   }
}
```

```
// Class that uses above MyException
public class TestThrow{
   public static void main(String args[])
   {
      try    {
         // throw an object of user defined exception
         throw new UserDefinedException("This is user-
      defined exception");
      }
      catch (UserDefinedException u1)
      {
         System.out.println("Caught the exception");
         // Print the message from MyException object
         System.out.println(u1.getMessage());
      }
   } }
```

Output :

Caught the exception

This is user-defined exception

Throwable Class

- **Throwable Class**:

Throwable class is the superclass of every error and exception in the Java language.

Hierarchy:

- Throwable
  - Exception
  - Error

public class Throwable extends Object

implements Serializable

- **Key Methods**:
  - getMessage()
  - printStackTrace()

Conclusion :

- Exception Handling ensures robust and error-free programs.
- Differentiate between exceptions and errors.
- Use try-catch-finally wisely.
- Create custom exceptions for better error management.