Exception Handling

Exceptions in Java

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

- An exception is an abnormal condition that arises in a code sequence at run time.

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.

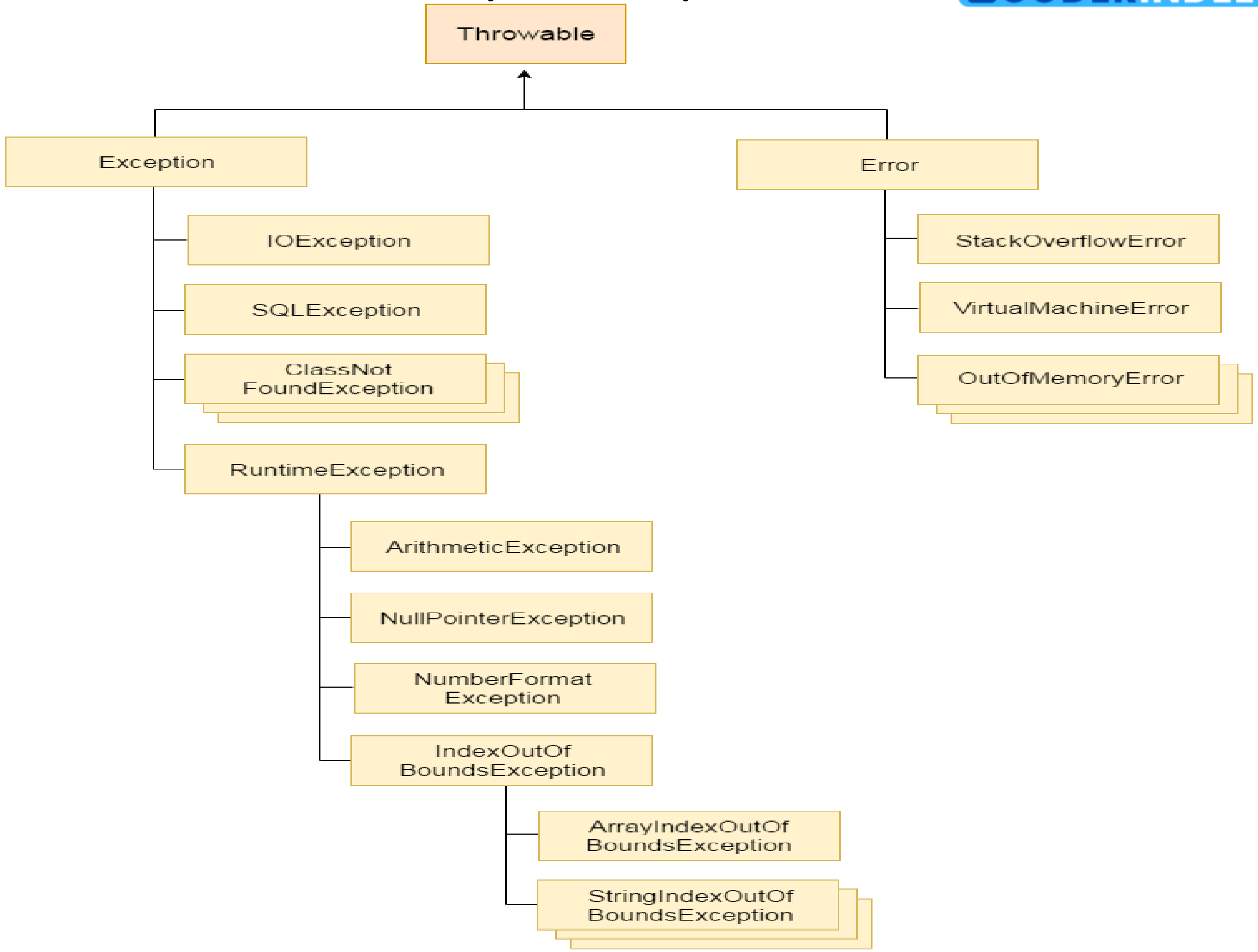
- In other words, “An exception is a run-time error.”

- An Exception is a run-time error that can be handled programmatically in the application and does not result in abnormal program termination.

- Exception handling is a mechanism that facilitates programmatic handling of run-time errors.

- In java, each run-time error is represented by an object.

Hierarchy of exception classes



- At the root of the class hierarchy, there is a class named *‘Throwable’* which represents the basic features of run-time errors.

- There are two non-abstract sub-classes of Throwable.

• Exception	: can be handled
• Error	: can’t be handled

- *RuntimeException* is the sub-class of Exception.

- Each exception is a run-time error but all run-time errors are not exceptions.

Types of Exceptions: Checked and Unchecked

Checked Exception

- 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 *throws* keyword.
- Checked Exceptions are those, that have to be either caught or declared to be thrown in the method in which they are raised.
- Examples: FileNotFoundException, IOException,

  SQLException, ClassNotFoundException

Example in the next slide:

•

Checked Exception

```
import java.io.*;
public class checked {
        public static void main(String[] args) {
        FileReader file = new FileReader("C:\\Users\\abc\\a.txt");
                BufferedReader fileInput = new BufferedReader(file);
                for (int counter = 0; counter < 3; counter++)
                System.out.println(fileInput.readLine());
                fileInput.close();
        }
}
```

The program doesn’t compile, because the function main() uses FileReader() and FileReader() throws a checked exception *FileNotFoundException*. It also uses readLine() and close() methods, and these methods also throw checked exception *IOException*

Method throwing Checked Exception

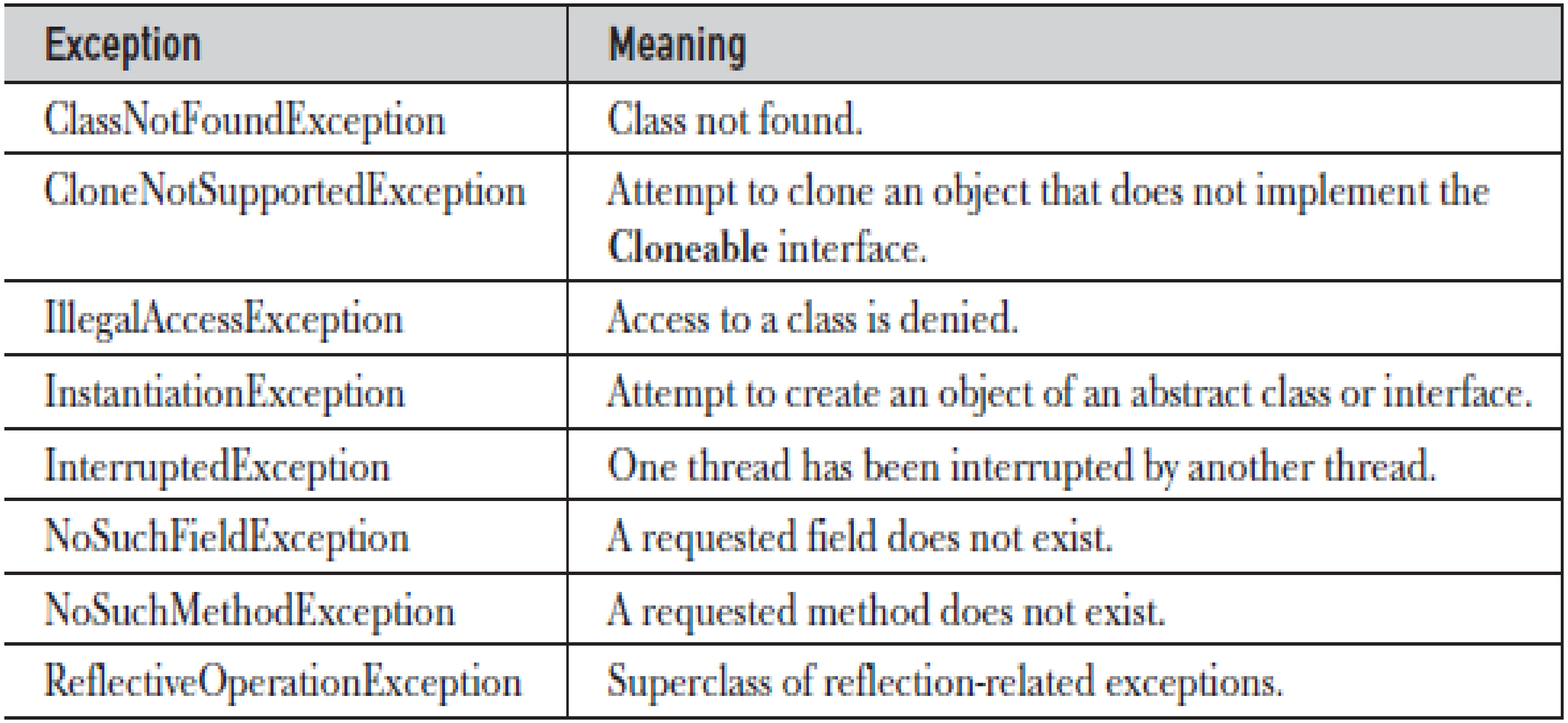
```
import java.io.*;
public class checkedcorrect {
		public static void main(String[] args) throws IOException {
				FileReader file = new FileReader("C:\\Users\\a.txt");
				BufferedReader fileInput = new BufferedReader(file);
				for (int counter = 0; counter < 3; counter++)
				System.out.println(fileInput.readLine());
				fileInput.close();
		}
}
```

We have used throws in the below program. Since *FileNotFoundException* is a subclass of *IOException*, we can just specify *IOException* in the throws list and make the above program compiler-error-free.

Output:

<First three lines a.txt will be printed>

Java’s Checked Exceptions defined in java.lang package



Exception	Meaning
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.
ReflectiveOperationException	Superclass of reflection-related exceptions.

IOException
SQLException
FileNotFoundException[Sub-class of IOException]

Unchecked Exceptions

- Unchecked Exceptions are those that are not forced by the compiler either to be caught or to be thrown.
- Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn’t handle/declare that exception, the program won’t give a compilation error.
- Most of the times these exception occurs due to the bad data provided by user during the user-program interaction. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately.
- In Java exceptions/or classes under *Error* and *RuntimeException* classes are unchecked exceptions

Unchecked Exceptions[Subclasses of RuntimeException class defined in java.lang package

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an 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.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

How a default exception is thrown by JVM??

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.

```
public class Main {
public static void main(String args[]) {
        int x = 0;
        int y = 10;
        int z = y/x;
}
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at Main.main(Main.java:5) Java Result: 1
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception.
- once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.

Key points from the previous example

- In the previous example, we haven’t supplied any exception handlers of our own.
- So the exception is caught by the default handler provided by the Java run-time system.
- Any exception that is not caught by your program will ultimately be processed by the default handler.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

CODERINDEED

Why Exception handling?

- When the default exception handler is provided by the Java run-time system , why Exception Handling?

- Exception Handling is needed because:
  - it allows to fix the error, customize the message .
  - it prevents the program from automatically terminating

Customized Exception Handling

Java uses following keywords for handling exceptions:

- try
- catch
- throw
- throws
- finally

Keywords for Exception Handling

try

- used to execute the statements whose execution may result in an exception.

```
try  {
        Statements whose execution may cause an exception
      }
```

Note: try block is always used either with catch or finally or with both.

catch

- catch is used to define a handler.

- It contains statements that are to be executed when the exception represented by the try block is generated.

- If program executes normally, then the statements of catch block will not executed.

- If no catch block is found in program, exception is caught by JVM and program is terminated.

Basic example of try-catch

```
public class Main
{
public static void main(String args[])
{
int d, a;
try
{ // monitor a block of code.
d = 0;
a = 42 / d;
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
```

Key points from the last example

If try catch block is not used, then default exception handler will run, and exception will be handled by Java runtime system automatically, but where the exception arises the program will terminate there only and next part of program will not run.

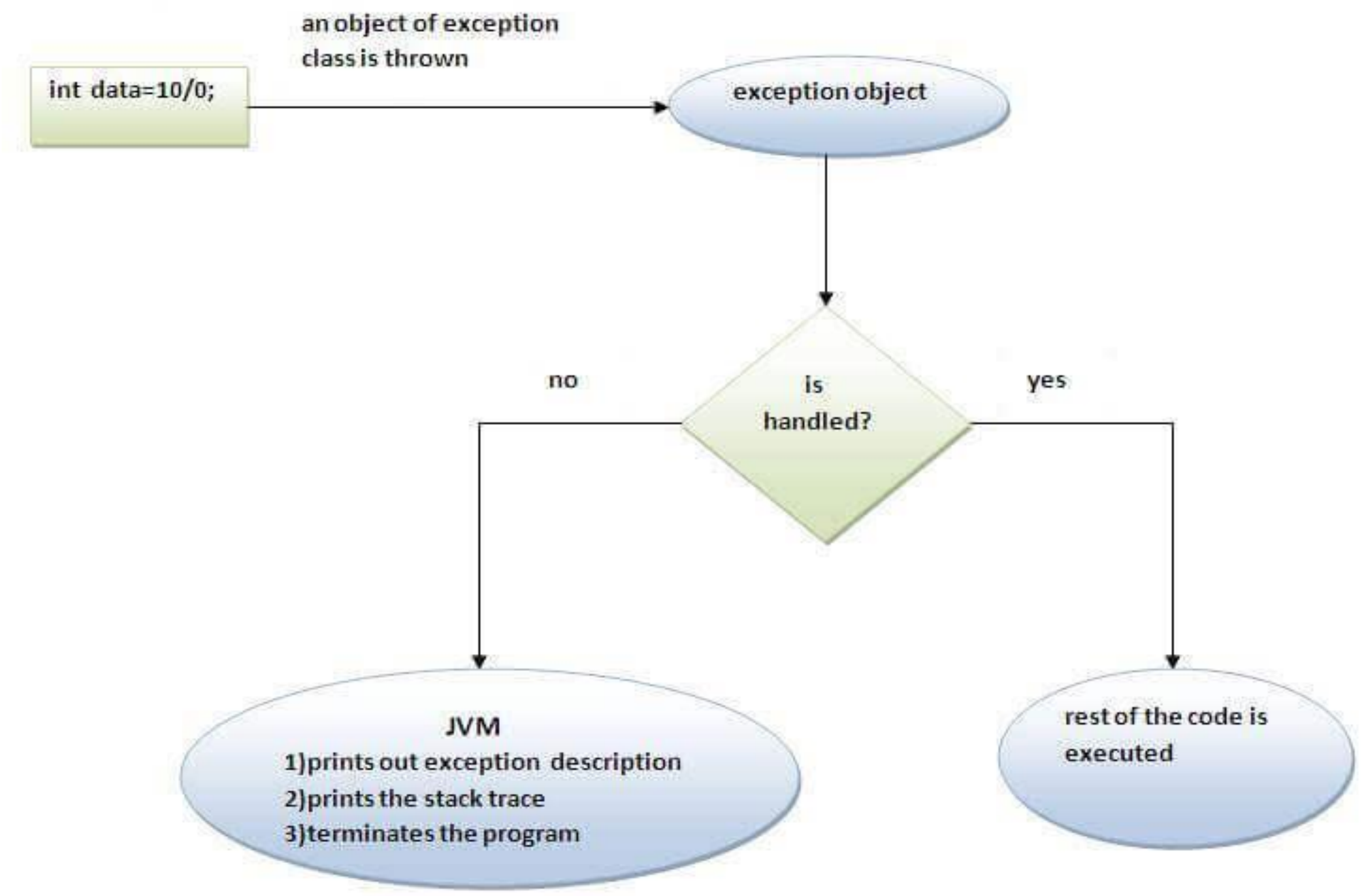
If we remove try catch from last example, then it will be like:

```
public class Main
{
public static void main(String args[])
{
int d, a;
d = 0;
a = 42 / d;//Exception will be raised here and message after this will not be printed, as program
will be terminated
System.out.println("Statement after expression");
}
}
```

Exception raised by JVM:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at Main.main(Main.java:7)
```

Internal working of java try catch block


int data=10/0;
an object of exception class is thrown
exception object
is handled?
no
yes
JVM
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

Multiple catch clauses

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one **catch** statement executes, the others are bypassed, and execution continues after the **try /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, Example: catch for ArithmeticException must come before catch for Exception.

Example

```
public class Main {
   public static void main(String[] args) {
          try{
              String x="ABC";
               int a[]=new int[5];
                a[0]=30/0;//It will raise ArithmeticException
               //System.out.println(a[7]);//It will raise ArrayIndexOutOfBoundsException
               //System.out.println(Integer.parseInt(x));//It can raise NumberFormatException, but its specific catch is not
written so catch with Exception class will work
              }
              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");
   }
}
```

Multi-catch feature introduced from JDK 7 onwards

```
// Demonstrate the multi-catch feature.
public class Main {
public static void main(String args[]) {
int a=10, b=0;
int vals[] = { 1, 2, 3 };
try {
int result = a / b; // generate an ArithmeticException
 //vals[10] = 19; // generate an ArrayIndexOutOfBoundsException
// This catch clause catches both exceptions.
} catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
System.out.println("Exception caught: " + e);
}
System.out.println("After multi-catch.");
}
}
```

Output:

```
Exception caught: java.lang.ArithmeticException: / by zero
After multi-catch.
```

Example, where compilation error could come

```
public class Main {
   public static void main(String[] args) {
         try{
            String x="ABC";
             int a[]=new int[5];
              a[0]=30/0;//It will raise ArithmeticException
             //System.out.println(a[7]);//It will raise ArrayIndexOutOfBoundsException
             //System.out.println(Integer.parseInt(x));//It can raise NumberFormatException, but its
specific catch is not written so catch with Exception class will work
            }
            catch(Exception e)
               {
                System.out.println("Parent Exception occurs");
               }
            catch(ArithmeticException e)
               {
                System.out.println("Arithmetic Exception occurs");
               }
            catch(ArrayIndexOutOfBoundsException e)
               {
                System.out.println("ArrayIndexOutOfBounds Exception occurs");
               }
            System.out.println("rest of the code");
   }
}
```

Here compilation error will come, as general catch handler with super class is written first and then the subclass catches are written

Nested try statements

- The try block within a try block is known as nested try block in java.
- 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)
{
}
....
```

Example

```
public class Main{
 public static void main(String args[]){
  try{
      int arr[]=new int[5];
    try{
     System.out.println("going to divide");
     int b =39/0;
    }catch(ArithmeticException e){System.out.println(e+" inner");}

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

    System.out.println(arr[6]);
  }catch(Exception e){System.out.println(e+" outer");}
  System.out.println("After try catch");
 }
}
```

Output:

```
going to divide
java.lang.ArithmeticException: / by zero inner
java.lang.ArrayIndexOutOfBoundsException: 5 inner
java.lang.ArrayIndexOutOfBoundsException: 6 outer
After try catch
```

Defining Generalized Exception Handler

- A generalized exception handler is one that can handle the exceptions of all types.

- If a class has a generalized as well as specific exception handler, then the generalized exception handler must be the last one.

Example

```
public class Main{
 public static void main(String args[]){
  try {
             String x=null;
            //int c = 12/0;//It will raise ArithmeticException
            //System.out.println(c);
             System.out.println(x.length());//It will raise NullPointerException
       }
catch (Throwable e) //Generalized exception handler, which can handle both
     exceptions[one at one time]
      {
     System.out.println(e);
      }
 }
}
```

Using throw keyword

The *throw* keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The *throw* keyword is mainly used to throw custom exceptions.

Syntax:

throw Instance

Example:

throw new ArithmeticException("/ by zero");

But this exception i.e, *Instance* must be of type **Throwable** or a subclass of **Throwable**.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing try block is checked to see if it has a catch statement that matches the type of exception.

If it finds a match, controlled is transferred to that statement otherwise next enclosing try block is checked and so on. If no matching catch is found then the default exception handler will halt the program.

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

Output:

```
Caught inside demo.
Recaught: java.lang.NullPointerException: demo
```

Using throws keyword

throws

- A throws clause lists the types of exceptions that a method might throw.

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

- This is necessary for all exceptions, except those of type Error or Runtime Exception, or any of their subclasses.

- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

Example of throws

```
import java.io.*;
public class abc {
  public static void findFile() throws IOException {
    // code that may produce IOException
    File newFile=new File("test.txt");
    FileInputStream stream=new FileInputStream(newFile);
  }
  public static void main(String[] args) {
    findFile();//try catch block must be used as method throws checked exception
  }
}
```

Output:

```
C:\Codes>javac abc.java
abc.java:9: error: unreported exception IOException; must be caught or declared to be thrown
    findFile();
            ^
1 error
```

Enclosing method call inside try-catch which throws the exception

```
import java.io.*;
public class abc {
  public static void findFile() throws IOException {
    // code that may produce IOException
    File newFile=new File("test.txt");
    FileInputStream stream=new FileInputStream(newFile);
  }
  public static void main(String[] args) {
    try{
      findFile();
    } catch(IOException e){
      System.out.println(e);
    }
  }
}
```

Output:

```
java.io.FileNotFoundException: test.txt (No such file or directory)
```

throw vs throws

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Throw is followed by an instance.	Throws is followed by class.
3)	Throw is used within the method.	Throws is used with the method signature.
4)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Finally

- finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.

- The finally block will execute whether or not an exception is thrown.

- If an exception is thrown, the finally block will execute even if no catch statement matches the exception.

- If a finally block is associated with a try, the finally block will be executed upon conclusion of the try.
- The finally clause is optional. However, each try statement requires at least one catch or a finally clause.
- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Example 1

```
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...
```

Example 2

```
// Demonstrate finally.
public class Main {
// Throw 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
```

Propagation of Exceptions

- If an exception is not caught and handled where it is thrown, the control is passed to the method that has invoked the method where the exception was thrown.

- The propagation continues until the exception is caught, or the control passes to the main method, which terminates the program and produces an error message.

Example

```
public class Main{
                public void first(){int data=50/0;  }
                public void second(){first();}
                public void third(){try{second();}
                                                catch(Exception e){
System.out.println("Exception occurred");}
                                }
                public static void main(String [] args){
                Main ob = new Main();
                ob.third();
                System.out.println("Thank You");
                                }
}
```

Output:

```
Exception occurred
Thank You
```

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.

- **ArithmeticException**
  It is thrown when an exceptional condition has occurred in an arithmetic operation.
- **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.
- **ClassNotFoundException**
  This Exception is raised when we try to access a class whose definition is not found
- **FileNotFoundException**
  This Exception is raised when a file is not accessible or does not open.
- **IOException**
  It is thrown when an input-output operation failed or interrupted
- **InterruptedException**
  It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.

- **NoSuchFieldException**
  It is thrown when a class does not contain the field (or variable) specified
- **NoSuchMethodException**
  It is thrown when accessing a method which is not found.
- **NullPointerException**
  This exception is raised when referring to the members of a null object. Null represents nothing
- **NumberFormatException**
  This exception is raised when a method could not convert a string into a numeric format.
- **RuntimeException**
  This represents any exception which occurs during runtime.
- **StringIndexOutOfBoundsException**
  It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string

Few Examples
ArithmeticException

```
// Java program to demonstrate ArithmeticException
class Main
{
	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");
		}

		System.out.println("Code will execute after the exception handled");
	}
}
```

Output:

```
Can't divide a number by 0
Code will execute after the exception handled
```

If we use try catch block to handle the exceptions then code will be executed even after the exception is thrown which is not possible without handling the exception. If we do not handle the exception then JVM will give built in exception message and terminate the program.

NullPointerException

```
//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..");
		}
	}
}
```

ArrayIndexOutOfBounds Exception

```
public class Main {
public static void main(String args[]) {
int a[]=new int[5];
try {
    System.out.println(a[7]);
}
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println(e);
}
System.out.println("After try catch...");
}
}
```

Output:

```
java.lang.ArrayIndexOutOfBoundsException: 7
After try catch...
```

StringIndexOutOfBoundsException

```
public class Main {
public static void main(String args[]) {
String x="Testing";
try {
   System.out.println(x.charAt(7));
}
catch(StringIndexOutOfBoundsException e)
{
   System.out.println(e);
}
System.out.println("After try catch...");
}
}
```

Output:

```
java.lang.StringIndexOutOfBoundsException: String index out of range: 7
After try catch...
```

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.

Step 1:

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 MyException extends Exception

- We can write a default constructor in his own exception class.
  MyException(){}
- 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.
  MyException(String str) { super(str); }

Step 2:

To raise exception of user-defined type, we need to create an object to his exception class and throw it using throw clause, as:

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

Example

```
// Java program to demonstrate user defined
exception
class MyException extends Exception
{
private static int accno[] = {1001, 1002, 1003,
1004,1005};
private static String name[] = {"Nish", "Shubh",
"Sush", "Abhi", "Akash"};
private static double bal[] = {10000.00,
12000.00, 5600.0, 999.00, 1100.55};
MyException() { }
MyException(String str) { super(str);
 }
-----------------------------------------------------------
```

Output:

```
1001    Nish    10000.0
1002    Shubh   12000.0
1003    Sush    5600.0
1004    Abhi    999.0
Main: Balance is less than 1000
        at Main.main(Main.java:18)
Program is running fine after handlng the
exception.
```

```
public static void main(String[] args)
{
try {
for (int i = 0; i < 5 ; i++)
{
System.out.println(accno[i] + "\t" + name[i] +"\t" + bal[i]);
if (bal[i] < 1000)
{
MyException me = new MyException("Balance is less
than 1000");
throw me;
}
}
}
catch (MyException e) {
e.printStackTrace();
}
System.out.println("Program is running fine after handlng
the exception.");
}
}
```

Methods of Throwable class

printStackTrace():

It is a method of Java’s **throwable class** which prints the throwable along with other details like the line number and class name where the exception occurred.

getMessage():

The getMessage() method of Java Throwable class is used to get a detailed message of the Throwable[or Exception thrown].

Example 1

```
public class Main
{
        public static void main(String[] args) {
                try
                {
                   int a=12,b=0;
                   System.out.println(a/b);
                }
                catch(ArithmeticException e)
                {
                   System.out.println(e);
                   e.printStackTrace();
                   System.out.println(e.getMessage());
                }
        }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
java.lang.ArithmeticException: / by zero
	at Main.main(Main.java:8)
/ by zero
```

Example 2

```
public class Main
{
        public static void main(String[] args) {
                try
                {
                   throw new ArithmeticException("Wrong denominator");
                }
                catch(ArithmeticException e)
                {
                   System.out.println(e);
                   e.printStackTrace();
                   System.out.println(e.getMessage());
                }
        }
}
```

Output:

```
java.lang.ArithmeticException: Wrong denominator
java.lang.ArithmeticException: Wrong denominator
        at Main.main(Main.java:7)
Wrong denominator
```

Example 3

```
public class Main
{
		public static void main(String[] args) {
				try
				{
				   throw new ArithmeticException();
				}
				catch(ArithmeticException e)
				{
				   System.out.println(e);
				   e.printStackTrace();
				   System.out.println(e.getMessage());
				}
		}
}
```

Output:

```
java.lang.ArithmeticException
java.lang.ArithmeticException
	at Main.main(Main.java:7)
null
```

Q1
Predict the output of following Java program

```
public class Main {
  public static void main(String args[]) {
    try {
      throw 10;
    }
    catch(int e) {
      System.out.println("Got the  Exception " + e);
    }
  }
}
```

A. Got the Exception 10
B. Got the Exception 0
C. Compiler Error
D. Blank output

Q2:What will be the output of following code?

```
class Test extends Exception { }
public class Main {
  public static void main(String args[]) {
    try {
      throw new Test();
    }
    catch(Test t) {
      System.out.println("Got the Test Exception");
    }
    finally {
      System.out.println("Inside finally block ");
    }
  }
}
```

A. Got the Test Exception
   Inside finally block
B. Got the Test Exception
C. Inside finally block
D. Compiler Error

Q3:What will be the output of following Java program?

```
public class Main {
  public static void main(String args[]) {
    int x = 0;
    int y = 10;
    int z = y/x;
  }
}
```

A. Compiler Error

B. Compiles and runs fine

C. Compiles fine but throws ArithmeticException exception

D. None of these

What will be the output of following code?[Q4]

```
public class Test
{
    public static void main (String[] args)
    {
        try
        {
            int a = 0;
            System.out.println ("a = " + a);
            int b = 20 / a;
            System.out.println ("b = " + b);
        }

        catch(ArithmeticException e)
        {
            System.out.println ("Divide by zero error");
        }

        finally
        {
            System.out.println ("inside the finally block");
        }
    }
}
```

A. Compile error
B. Divide by zero error
C. a = 0
   Divide by zero error
   inside the finally block
D. a = 0

What will be the output of following code?[Q5]

```
public class Test
{
    public static void main(String[] args)
    {
        try
        {
            int a[]= {1, 2, 3, 4};
            for (int i = 1; i <= 4; i++)
            {
                System.out.println ("a[" + i + "]=" + a[i] + "n");
            }
        }
        catch (Exception e)
        {
            System.out.println ("error = " + e);
        }

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

A. Compiler error
B. Run time error
C. ArrayIndexOutOfBoundsException
D. Array elements are printed

Predict the output of the following program.[Q6]

```
public class Test
{   int count = 0;
   void A() throws Exception
   {
      try
      {
         count++;
         try
         {
            count++;

            try
            {
               count++;
               throw new Exception();
            }
            catch(Exception ex)
            {
               count++;
               throw new Exception();
            }
         }
         catch(Exception ex)
         {
            count++;
         }
      }
      catch(Exception ex)
      {
         count++;
      }
   }
void display()
   {
      System.out.println(count);
   }
   public static void main(String[] args)
throws Exception
   {
      Test obj = new Test();
      obj.A();
      obj.display();
   }
}
```

A. 4

B. 5

C. 6

D. Compilation error

Which of these is a super class of all errors and exceptions in the Java language?[Q7]

A. RunTimeExceptions

B. Throwable

C. Catchable

D. None of the above

Output??[Q8]

```
public class Test
{
        public static void main(String[] args)
        {
                try
                {
                        System.out.printf("1");
                        int sum = 9 / 0;
                        System.out.printf("2");
                }
                catch(ArithmeticException e)
                {
                        System.out.printf("3");
                }
                catch(Exception e)
                {
                        System.out.printf("4");
                }
                finally
                {
                        System.out.printf("5");
                }
        }
}
```

a) 1325
b) 1345
c) 1342
d) 135

Output??[Q9]

```
public class Test
{
                public static void main(String[] args)
                {
                                try
                                {
                                                System.out.printf("1");
                                                int data = 5 / 0;
                                }
                                catch(ArithmeticException e)
                                {
                                                System.out.printf("2");
                                                System.exit(0);
                                }
                                finally
                                {
                                                System.out.printf("3");
                                }
                                System.out.printf("4");
                }
}
```

a) 12
b) 1234
c) 124
d) 123

Output??[Q10]

```
public class Test
{
        public static void main(String[] args)
        {
                try
                {
                        System.out.printf("1");
                        int data = 5 / 0;
                }
                catch(ArithmeticException e)
                {
                        Throwable obj = new Throwable("Sample");
                        try
                        {
                                throw obj;
                        }
                        catch (Throwable e1)
                        {
                                System.out.printf("8");
                        }
                }
                finally
                {
                        System.out.printf("3");
                }
                System.out.printf("4");
        }
}
```

a) Compilation error
b) Runtime error
c) 1834
d) 134

Output??[Q11]

```
import java.io.EOFException;
import java.io.IOException;

public class Test
{
        public static void main(String[] args)
        {
                try
                {
                        System.out.printf("1");
                        int value = 10 / 0;
                        throw new IOException();
                }
                catch(EOFException e)
                {
                        System.out.printf("2");
                }
                catch(ArithmeticException e)
                {
                        System.out.printf("3");
                }
                catch(NullPointerException e)
                {
                        System.out.printf("4");
                }
                catch(IOException e)
                {
                        System.out.printf("5");
                }
                catch(Exception e)
                {
                        System.out.printf("6");
                }
        }
}
```

a) 1346

b) 136726

c) 136

d) 13

Output??[Q12]

```
public class Main
{
                public static void main(String[] args) {
                try
    {
    System.out.print("Hello" + " " + 1 / 0);
    }
    catch(ArithmeticException e)
    {
                System.out.print("World");
            }
                }
}
```

a) Hello
b) World
c) HelloWorld
d) Hello World

Output??[Q13]

What will be the output of the following Java program?

```
public class exception_handling
{
    public static void main(String args[])
    {
        try
       {
            int a, b;
            b = 0;
            a = 5 / b;
            System.out.print("A");
        }
        catch(ArithmeticException e)
        {
             System.out.print("B");
        }
    }
}
```

a) A

b) B

c) Compilation Error

d) Runtime Error

Output??[Q14]

```
// Demonstrate the multi-catch feature.
public class Main {
public static void main(String args[]) {
int sum=10;
try
{
int i;
for (i = -1; i < 3 ;++i)
sum = (sum / i);
}
catch(ArithmeticException e)
{
System.out.print("0 ");
}
System.out.print(sum);
}
}
```

A. 0 -10

B. 0 10

C. Compile time error

D. 0