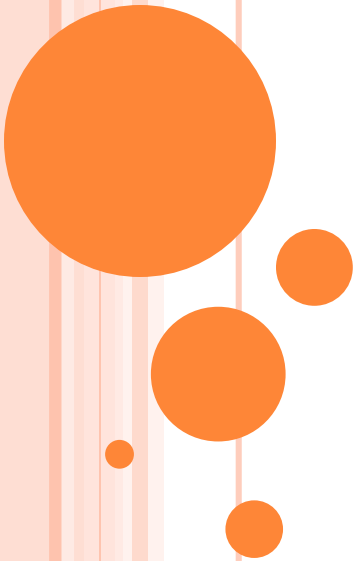


MODULE IV

Part I

When to use exception handling, Java exception hierarchy, finally block, Stack unwinding, Chained exceptions, Declaring new exception types

EXCEPTION

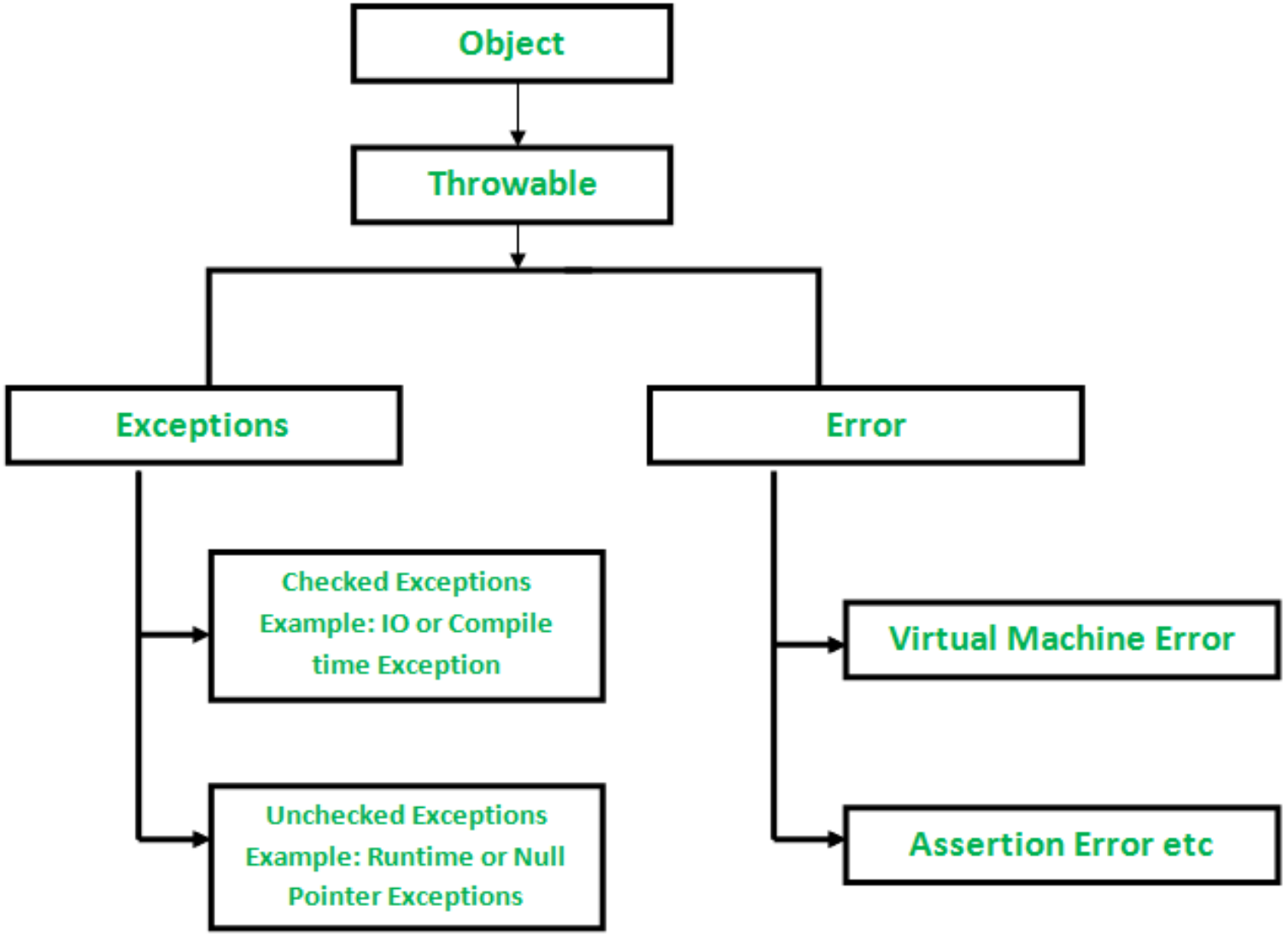
- An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.
- **[Error:** An Error indicates serious problem that a reasonable application should not try to catch.]

- **Exception:** Exception indicates conditions that a reasonable application might try to catch.
- It is an object which is thrown at runtime.

Exception Hierarchy

- All exception and errors types are sub classes of class **Throwable**, which is base class of hierarchy.
- One branch is headed by **Exception**.
  - This class is used for exceptional conditions that user programs should catch.
  - NullPointerException is an example of such an exception.
- Another branch,**Error**
  - Are used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE).
  - StackOverflowError is an example of such an error.

Exception hierarchy



How JVM handle an Exception?

- **Default Exception Handling :**
- Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system(JVM).
- The exception object contains name and description of the exception, and current state of the program where exception has occurred.
- Creating the Exception Object and handling it to the run-time system is called **throwing** an Exception.
- There might be the list of the methods that had been called to get to the method where exception was occurred. This ordered list of the methods is called **Call Stack**.

- Now the following procedure will happen.
  - The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called **Exception handler**.
  - The run-time system starts searching from the method in which exception occurred, proceeds through call stack in the reverse order in which methods were called.
  - If it finds appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
  - If run-time system searches all the methods on call stack and couldn't have found the appropriate handler then run-time system handover the Exception Object to **default exception handler** , which is part of run-time system. This handler prints the exception information in the following format and terminates program **abnormally**.

Types of Java Exceptions

- In Java, there are two types of exceptions:
- **1) Checked Exception**
  - Checked exceptions are checked at compile-time.
  - It means if a method is throwing a checked exception then it should handle the exception using **try-catch block** or it should declare the exception using **throws keyword**, otherwise the program will give a compilation error.
  - e.g. IOException, SQLException etc.
- **2) Unchecked Exception**
  - The classes which inherit RuntimeException are known as unchecked exceptions
  - e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.
  - Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Some Examples

- Example1:

```
public class Main{
    public static void main (String[] args) {
            // array of size 4.
        int[] arr = new int[4];
      // this statement causes an exception because index is from 0 to 3
        int i = arr[4];
        // the following statement will never execute
        System.out.println("Hi, I want to execute");
    }
}
```

- Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBound
sException: 4 at Main.main(Main.java:6)
```

- **Example 2**

```
public class Main
{
public static void main(String args[])
{
int d = 0;
  int n = 20;
 int fraction = n/d; //Exception(division by 0)
 System.out.println("End Of Main");
 }}
```

- **Output**

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

- **Example 3:**

```
public class Main
{
public static void main(String args[])
 {
String a = null; //null value
System.out.println(a.charAt(0)); //Exception(String
//is initialized to null
System.out.println("Completed");
 }}
```

- **Output:**

```
Exception in thread "main" java.lang.NullPointerExcep
tion at Main.main(Main.java:6)
```

Exception handling

- Exception handling ensures that the flow of the program doesn't break when an exception occurs.
- For example, if a program has bunch of statements and an exception occurs mid way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly.
- By handling we make sure that all the statements execute and the flow of program doesn't break.

How Programmer handles an exception?

- **Customized Exception Handling :**
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Program statements that you think can raise exceptions are contained within a **try block**.
- If an exception occurs within the try block, it is thrown.
- Your code can catch this exception (using **catch block**) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the **keyword throw**
- Any exception that is thrown out of a method must be specified as such by a **throws clause**.
- Any code that absolutely must be executed after a try block completes is put in a **finally block**.

Some Examples

- Example1

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

- Output:

  Can't divide a number by 0

- Example 2

```
public class Main
{
    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..");
        }
        System.out.println("Program Completed");
    }
}
```

- Output:

```
NullPointerException..

Program Completed
```

- Example3:

```
class  NumberFormat_Demo
{
    public static void main(String args[])
    {
        try {
            // “xyz" is not a number
            int num = Integer.parseInt (“xyz") ;
            System.out.println(num);
        }
catch(NumberFormatException e) {
            System.out.println("Number format exception");
        }
    }
}
```

- Output:

Number format exception

- Example 4:

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

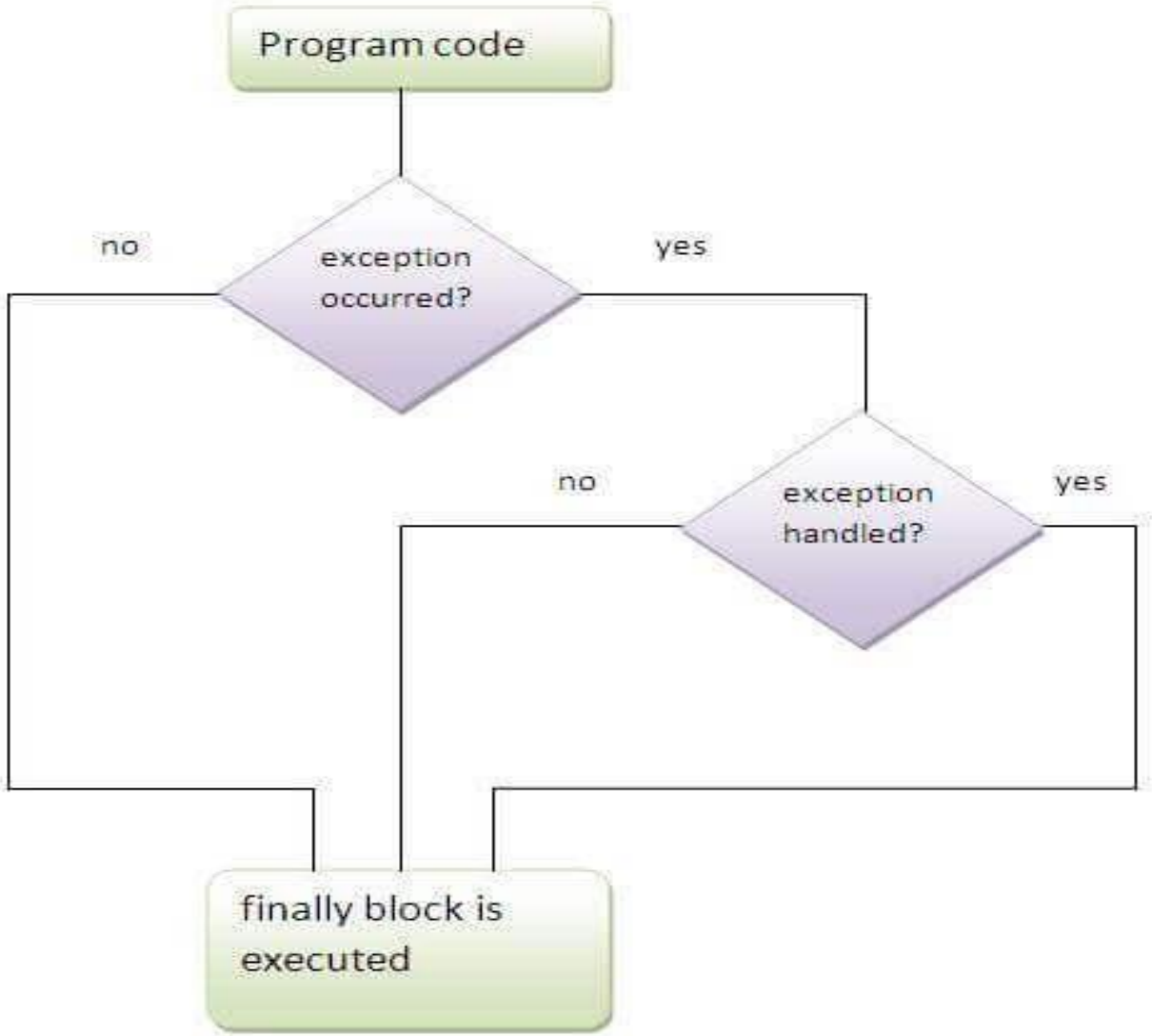
- Output:

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

Java finally block

- **Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.

FINALLY BLOCK



- Example1:

```
public class Main {
  public static void main(String args[]){
  try{
   int data=25/5;  //no exception occured
   System.out.println(data);
  }
  catch(NullPointerException e)
     {
System.out.println(e);
}
  finally
{
 System.out.println("finally block is executed");
}
  System.out.println("rest of the code...");
  }
}
```

- Output:

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

- Example 2-Exception not handled because the given type in catch block is not matching

```
public class main {
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(NullPointerException e)
{System.out.println(e);
}
  finally
{System.out.println("finally block is always executed");
}
  System.out.println("rest of the code...");
  }
}
```

- Output:

  finally block is always executed
  Exception in thread main java.lang.ArithmeticException:/ by zero

- Example 3

```
public class Main {
  public static void main(String args[]){
  try{
   int data=25/0;  //exception occured
   System.out.println(data);
  }
  catch(ArithmeticException e)
{
System.out.println(“Exception handled”);
}
  finally
{
System.out.println("finally block is always executed");
}
  System.out.println("rest of the code...");
  }
}
```

- Output:

```
Exception handled
finally block is always executed
rest of the code...
```

Stack unwinding

- The process of removing function entries from function call stack at run time is called Stack Unwinding.
- Stack Unwinding is generally related to Exception Handling.
- when an exception occurs, the function call stack is linearly searched for the exception handler, and all the entries before the function with exception handler are removed from the function call stack.
- So exception handling involves Stack Unwinding if exception is not handled in same function (where it is thrown).
- **Note :** By default, Unchecked Exceptions are forwarded in calling chain (propagated).

EXAMPLE

```
class Simple {
    void m()
    {
        int data = 50 / 0; // unchecked exception occurred
        // exception propagated to n()
    }
    void n()
    {
        m();
        // exception propagated to p()
    }
    void p()
    {
        try {
            n(); // exception handled
        }
        catch (Exception e) {
            System.out.println("Exception handled");
        }
    }
    public static void main(String args[])
    {
        Simple obj = new Simple();
        obj.p();
        System.out.println("Normal flow...");
    }
}
```

- Output:

  Exception handled

  Normal flow...

Java Multi-catch block

- A try block can be followed by one or more catch blocks.
- Each catch block must contain a different exception handler.
- So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.
- **Points to remember**
- 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, i.e. catch for ArithmeticException must come before catch for Exception.

```
public class Main {

   public static void main(String[] args) {

         try{
            int a[]=new int[5];

            System.out.println(a[10]);
           }
           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:**

ArrayIndexOutOfBounds Exception occurs
rest of the code

```
public class MultipleCatchBlock1 {

   public static void main(String[] args) {

        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
           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:**

Arithmetic Exception occurs
rest of the code

Nested try blocks

- In Java, we can use a try block within a try block.
- Each time a try statement is entered, the context of that exception is pushed on to a stack.
- Given below[next slide] is an example of a nested try.
  In this example, inner try block (or try-block2) is used to handle ArithmeticException, i.e., division by zero.
- After that, the outer try block (or try-block) handles the ArrayIndexOutOfBoundsException.

```
class NestedTry {

    // main method
    public static void main(String args[])
    {
        // Main try block
        try {
            // initializing array
            int a[] = { 1, 2, 3, 4, 5 };
            // trying to print element at index 5
            System.out.println(a[5]);
            // try-block2 inside another try block
            try {
                // performing division by zero
                int x = a[2] / 0;
            }
            catch (ArithmeticException e2) {
                System.out.println("division by zero is not possible");
            }
        }
        catch (ArrayIndexOutOfBoundsException e1) {
            System.out.println("ArrayIndexOutOfBoundsException");
            System.out.println("Element at such index does not exists");
        }
    }
    // end of main method
}
```

- Output:

  ArrayIndexOutOfBoundsException

  Element at such index does not exists

- Whenever a try block does not have a catch block for a particular exception, then the catch blocks of parent try block are inspected for that exception, and if a match is found then that catch block is executed.
- If none of the catch blocks handles the exception then the Java run-time system will handle the exception and a system generated message would be shown for the exception.

```
class Nesting {
    // main method
    public static void main(String args[])
    {
        // main try-block
        try {
            // try-block2
            try {
                // try-block3
                try {
                    int arr[] = { 1, 2, 3, 4 };
                    System.out.println(arr[10]);
                }
                catch (ArithmeticException e)
                    System.out.println(" try-block3");
                }
            }
            catch (ArithmeticException e) {
                System.out.println(" try-block2");
            }
        }
        catch (ArrayIndexOutOfBoundsException e4) {
            System.out.println(" main tryblock");
        }
    }
}
```

- Output:

  main tryblock

Chained Exceptions in Java

- Chained Exceptions allows to relate one exception with another exception, i.e one exception describes cause of another exception.
- For example, consider a situation in which a method throws an ArithmeticException because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero.
- The method will throw only ArithmeticException to the caller.
- So the caller would not come to know about the actual cause of exception.
- Chained Exception is used in such type of situations.

Constructors Of Throwable class Which support chained exceptions in java :

- Throwable(Throwable cause) :- Where cause is the exception that causes the current exception.
- Throwable(String msg, Throwable cause) :- Where msg is the exception message and cause is the exception that causes the current exception.

- **Methods** Of Throwable class Which support chained exceptions in java :
  - getCause() method :- This method returns actual cause of an exception.
  - initCause(Throwable cause) method :- This method sets the cause for the calling exception.

```
public class ExceptionHandling
{
    public static void main(String[] args)
    {
        try
        {
            // Creating an exception
            NumberFormatException ex =
                    new NumberFormatException("Exception");

            // Setting a cause of the exception
            ex.initCause(new NullPointerException(
                    "This is actual cause of the exception"));

            // Throwing an exception with cause.
            throw ex;
        }

        catch(NumberFormatException ex)
        {
            // displaying the exception
            System.out.println(ex);

            // Getting the actual cause of the exception
            System.out.println(ex.getCause());
        }
    }
}
```

- Output:

  java.lang.NumberFormatException: Exception
  java.lang.NullPointerException: This is actual cause of the exception

JAVA.LANG.THROWABLE CLASS

- The **java.lang.Throwable** class is the superclass of all errors and exceptions in the Java language.
- Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.

Sr.No.	Constructor & Description
1	**Throwable()** This constructs a new throwable with null as its detail message.
2	**Throwable(String message)** This constructs a new throwable with the specified detail message.
3	**Throwable(String message, Throwable cause)** This constructs a new throwable with the specified detail message and cause.
4	**Throwable(Throwable cause)** This constructs a new throwable with the specified cause

Sr.No.	Method & Description
1	**Throwable getCause():** This method returns the cause of this throwable or null if the cause is nonexistent or unknown.
2	**String getMessage():** This method returns the detail message string of this throwable.
3	**StackTraceElement[] getStackTrace():** This method provides programmatic access to the stack trace information printed by printStackTrace().
4	**Throwable initCause(Throwable cause):** This method initializes the cause of this throwable to the specified value.
5	**void printStackTrace():** This method prints this throwable and its backtrace to the standard error stream.
6	**String toString():** This method returns a short description of this throwable.

GETMESSAGE()

```
public class Main {
    public static void main(String[] args)throws Throwable
{
   try{
      int i=10/0;
   }catch(Throwable t){
      System.out.println(t.getMessage());
   }
}
}
```

- **Output:**

/ by zero

PRINTSTACKTRACE()

```
public class Program {
    public static void foo() {
      try {
        int num1 = 5/0;
      }
      catch (Throwable e) {
          e.printStackTrace();
      }
    }

    public static void main( String args[] ) {
      foo();
    }
}
```

- Output:

```
java.lang.ArithmeticException: / by zero
at Program.foo(main.java:4)
at Program.main(main.java:12)
```

[Note that the top-most function in the stack trace is the one that was executed last, hence, that is the function where the exception occurred.]

- Last two slides includes the important constructors and methods of Throwable class
- If you want to refer more use the following link:
- https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html

THROW AND THROWS IN JAVA

- **throw**
- 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.

```
// Java program that demonstrates the use of throw
class ThrowExcep
{
    static void fun()
    {
        try
        {
            throw new NullPointerException("demo");
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught inside fun().");
            throw e; // rethrowing the exception
        }
    }

    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught in main.");
        }
    }
}
```

- Output:

  Caught inside fun().

  Caught in main.

- **throws**
- throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions.
- The caller to these methods has to handle the exception using a try-catch block.
- We can use throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM) then caller method is responsible to handle that exception.

- // Java program to demonstrate working of throws

```
class ThrowsExecp
{
    static void fun() throws ArithmeticException
    {
        System.out.println("Inside fun(). ");
        throw new ArithmeticException();
    }
    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(ArithmeticException e)
        {
            System.out.println("caught in main.");
        }
    }
}
```

- Output:

```
Inside fun().
caught in main.
```

Difference between throw and 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)	Checked exception cannot be propagated using throw only.	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 exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

User-defined Custom Exception in Java

- If you are creating your own Exception that is known as custom exception or user-defined exception.
- Java custom exceptions are used to customize the exception according to user need.
- By the help of custom exception, you can have your own exception and message.
- Let's see a simple example of java custom exception[next slides].

```
// A Class that represents use-defined expception
class MyException extends Exception
{
    public MyException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}

// A Class that uses above MyException
public class Main
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException(“My user defined Exception");
        }
        catch (MyException ex)
        {
            System.out.println("Caught");

            // Print the message from MyException object
            System.out.println(ex.getMessage());
        }
    }
}
```

- Output:

  Caught

  My user defined Exception

- In the above code, constructor of MyException requires a string as its argument.
- The string is passed to parent class Exception's constructor using super().
- The constructor of Exception class can also be called without a parameter and call to super is not mandatory.[next example]

```
// A Class that represents use-defined expception
class MyException extends Exception
{

}

// A Class that uses above MyException
public class setText
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException();
        }
        catch (MyException ex)
        {
            System.out.println("Caught");
            System.out.println(ex.getMessage());
        }
    }
}
```

- Output:

  Caught

  null

- **//Another Example**

```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 }
}
public class Main{

   static void validate(int age)throws InvalidAgeException{
     if(age<18)
      throw new InvalidAgeException("not valid");
     else
      System.out.println("welcome to vote");
   }

   public static void main(String args[]){
      try{
      validate(13);
      }
catch(Exception m)
{
System.out.println("Exception occured: "+m);
}
      System.out.println("rest of the code...");
  }
}
```

- Output:

  Exception occured: InvalidAgeException:not valid

  rest of the code...