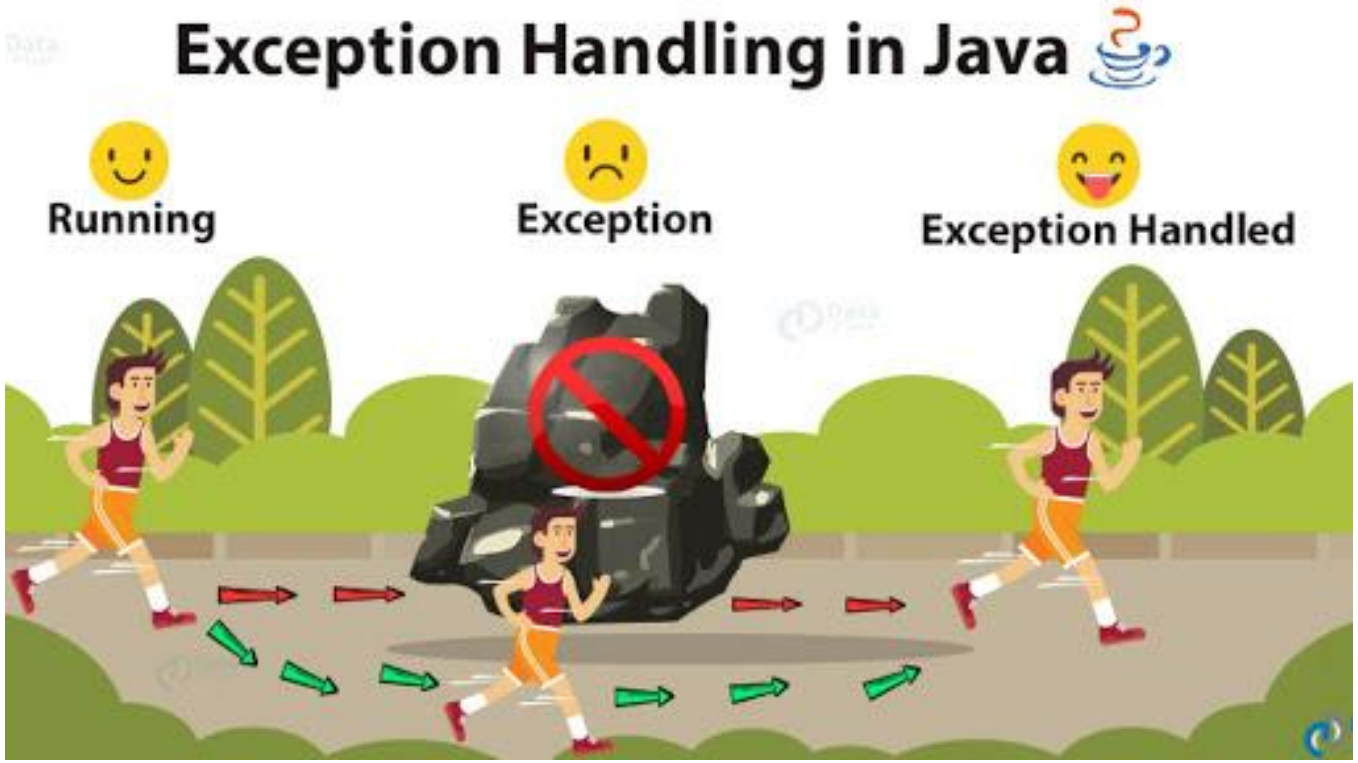
Exception Handling

A powerful mechanism to handle runtime errors

- **In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.**



What is exception?

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

When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.

- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
  - Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
  - Manually generated exceptions are typically used to report some error condition to the caller of a method.

Exception Handling

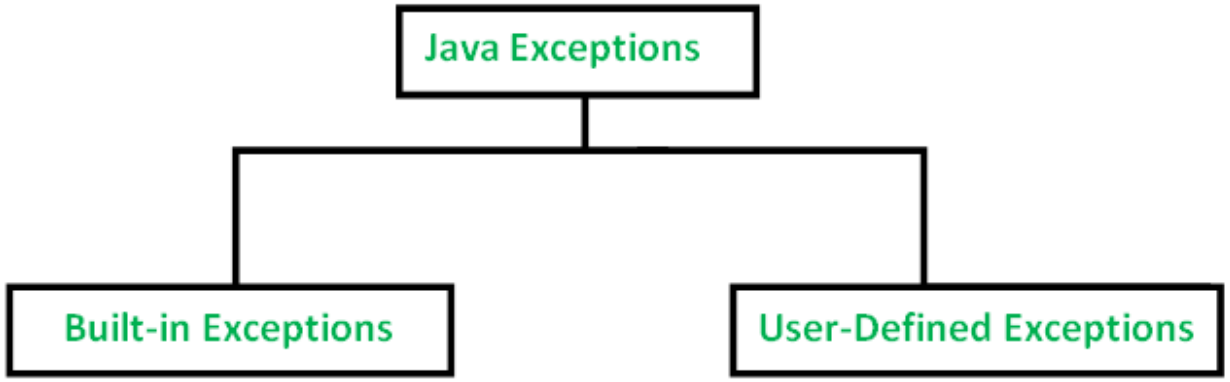
- Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.
- The core advantage of exception handling is to maintain the normal flow of the application.
- An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

What is exception –handling?

Java exception handling is managed via five keywords:

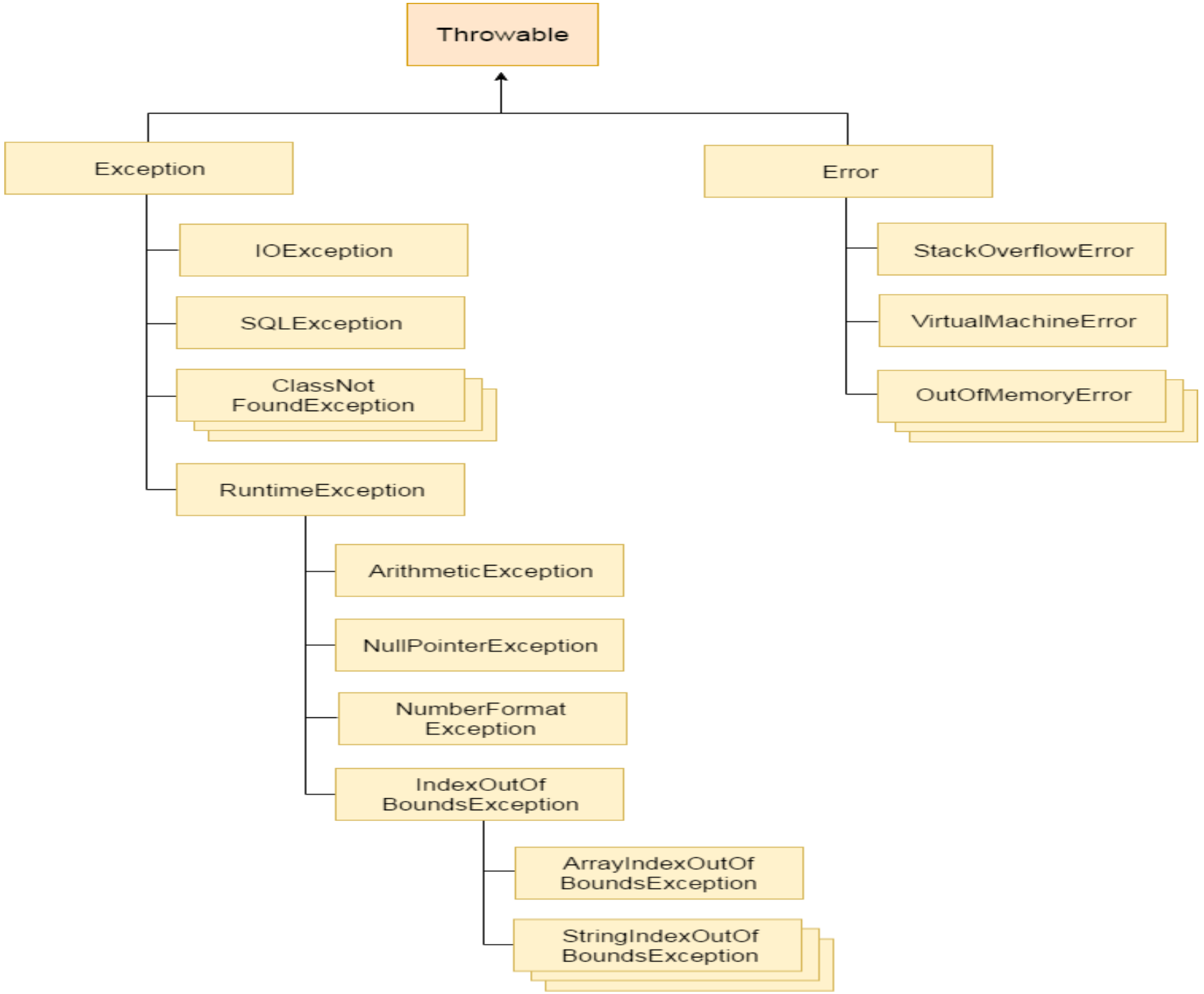
- **try**,
- **catch**,
- **throw**,
- **throws**,
- **finally**

Types of Exception in Java



Types of Exceptions

- **Built-in Exception**
  - The exceptions which are available in Java libraries.
  - These exceptions are suitable to explain certain error situations.
  - Example- ArithmeticException, IOException, etc.
- **User-defined Exception**
  - 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’.



Hierarchy of Java Exception classes

java.lang.Throwable

All exception types are subclasses of the built-in class **Throwable**

One branch of Throwable is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types

There is an important subclass of **Exception**, called **RuntimeException**

The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program.

Exceptions of type Error are used by the **Java run-time system** to indicate errors having to do with the run-time environment, itself.

Types of Java exceptions

- There are two types of exception
  - **Checked**
  - **Unchecked**

Checked and Unchecked Exception

Checked Exception

- The classes which directly inherit Throwable class **except RuntimeException and Error** are known as checked exceptions e.g. IOException, SQLException etc.
- Checked exceptions are **checked at compile-time**.

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.**

Error

- Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, Stackoverflow error, AssertionError etc.

Java Exception Keywords

- There are 5 keywords which are used in handling exceptions in Java.
  - **try**
  - **catch**
  - **finally**
  - **throw**
  - **throws**

Java Exception Keywords

- **try**
  - The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
- **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.

- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block.
- Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch

Java Exception Keywords

- **finally**
  - The "finally" block is used to execute the important 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 doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature

try-catch

- To guard against and handle a run-time error, enclose the code inside a try block.
- Immediately following a try block, include a catch clause by mentioning the exception that need to be handled.
- A try-catch form a unit where the catch clause is restricted to those statements specified by the immediately preceding try statement.

```
try {
  //  Block of code to try
}
catch(Exception e) {
  //  Block of code to handle errors
}
```

try-catch example

```
class Exc2 {
  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.");
  }
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

```
class Example{
   public static void main(String args[]){
      try{
         int arr[]=new int[7];
         arr[4]=30/0;
         System.out.println("Last Statement of try block");
      }
      catch(ArithmeticException e){
         System.out.println("You should not divide a number by zero");
      }
      catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Accessing array elements outside of the limit");
      }
      catch(Exception e){
         System.out.println("Some Other Exception");
      }
      System.out.println("Out of the try-catch block");
   }
}
```

```
You should not divide a number by zero
Out of the try-catch block
```

Multiple catch clauses

Certain cases more than one exception could be raised by a single piece of code. Here, two or more catch clauses, each catching a different type of exception.

The first catch block got executed because the code we have written in try block throws ArithmeticException (because we divided the number by zero).

```
class Example{
   public static void main(String args[]){
      try{
         int arr[]=new int[7];
         arr[10]=10/5;
         System.out.println("Last Statement of try block");
      }
      catch(ArithmeticException e){
         System.out.println("You should not divide a number by zero");
      }
      catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Accessing array elements outside of the limit");
      }
      catch(Exception e){
         System.out.println("Some Other Exception");
      }
      System.out.println("Out of the try-catch block");
   }
}
```

```
Accessing array elements outside of the limit
Out of the try-catch block
```

Multiple catch clauses

1. It is clear that when an exception occurs, the specific catch block (that declares that exception) executes. This is why in first example first block executed and in second example second catch.

2. Although I have not shown you above, but if an exception occurs in above code which is not Arithmetic and ArrayIndexOutOfBounds then the last generic catch handler would execute.

```java
class Example{
   public static void main(String args[]){
      try{
         int arr[]=new int[7];
         arr[10]=10/5;
         System.out.println("Last Statement of try block");
      }
      catch(Exception e){
         System.out.println("Some Other Exception");
      }
      catch(ArithmeticException e){
         System.out.println("You should not divide a number by zero");
      }
      catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Accessing array elements outside of the limit");
      }
      System.out.println("Out of the try-catch block");
   }
}
```

```
Compile time error: Exception in thread "main" java.lang.Error:
Unresolved compilation problems: Unreachable catch block for ArithmeticException.
It is already handled by the catch block for Exception Unreachable catch block
for ArrayIndexOutOfBoundsException. It is already handled by the catch block for
Exception at Example.main(Example1.java:11)
```

Why we got this error?

This is because we placed the generic exception catch block at the first place which means that none of the catch blocks placed after this block is reachable. You should always place this block at the end of all other specific exception catch blocks.

Nested try-catch block

- When a try catch block is present in another try block then it is called the **nested try catch block.**
- Each time a **try block does not have a catch handler for a particular exception, then the catch blocks of parent try block are inspected for that exception**, if match is found then that catch block executes.
- If neither catch block nor parent catch block handles exception then the system generated message would be shown for the exception, similar to what we see when we don't handle exception.

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

}
//Catch of Main(parent) try block
catch(Exception e3) {
      //Exception Message
}
....
```

Syntax of nested try

```
class NestingDemo{
   public static void main(String args[]){
      //main try-block
      try{
        //try-block2
        try{
            //try-block3
            try{
                int arr[]= {1,2,3,4};
                /* I'm trying to display the value of
                 * an element which doesn't exist. The
                 * code should throw an exception
                 */
                System.out.println(arr[10]);
            }catch(ArithmeticException e){
                System.out.print("Arithmetic Exception");
                System.out.println(" handled in try-block3");
            }
        }
        catch(ArithmeticException e){
           System.out.print("Arithmetic Exception");
           System.out.println(" handled in try-block2");
        }
    }
    catch(ArrayIndexOutOfBoundsException e4){
        System.out.print("ArrayIndexOutOfBoundsException");
        System.out.println(" handled in main try-block");
    }
    catch(Exception e5){
        System.out.print("Exception");
        System.out.println(" handled in main try-block");
     }
}
```

Each time a **try block does not have a catch handler for a particular exception, then the catch blocks of parent try block are inspected for that exception**

```
ArrayIndexOutOfBoundsException handled in main try-block
```

Finally, throw, throws- keywords we will see in the next video

Namah Shivaya!

Exception Handling

A powerful mechanism to handle runtime errors

finally

- A **finally block** contains all the **crucial statements** that must be executed whether **exception occurs or not**.
- The statements present in this **block will always execute** regardless of whether exception occurs in try block or not such as closing a connection, stream etc.

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

Example finally

```
class Example
{
   public static void main(String args[]) {
      try{
         int num=121/0;
         System.out.println(num);
      }
      catch(ArithmeticException e){
         System.out.println("Number should not be divided by zero");
      }
      /* Finally block will always execute
       * even if there is no exception in try block
       */
      finally{
         System.out.println("This is finally block");
      }
      System.out.println("Out of try-catch-finally");
   }
}
```

```
Number should not be divided by zero
This is finally block
Out of try-catch-finally
```

Important points in finally

1. A finally block must be associated with a try block, you cannot use finally without a try block. You should place those statements in this block that must be executed always.
2. Finally block is optional, however if you place a finally block then it will always run after the execution of try block.
3. In normal case when there is no exception in try block then the finally block is executed after try block. However if an exception occurs then the catch block is executed before finally block.
4. An exception in the finally block, behaves exactly like any other exception.
5. The statements present in the finally block execute even if the try block contains control transfer statements like return, break or continue.

```
class JavaFinally
{
   public static void main(String args[])
   {
      System.out.println(JavaFinally.myMethod());
   }
   public static int myMethod()
   {
      try {
        return 112;
      }
      finally {
        System.out.println("This is Finally block");
        System.out.println("Finally block ran even after return statement");
      }
   }
}
```

```
This is Finally block
Finally block ran even after return statement
112
```

Another example on finally

try-catch-finally

1. Either a try statement should be associated with a catch block or with finally.
2. Since catch performs exception handling and finally performs the clean up, the best approach is to use both of them.

```
try {
     //statements that may cause an exception
}
catch (…) {
     //error handling code
}
finally {
    //statements to be executed
}
```

Example try-catch-finally

```
class Example1{
  public static void main(String args[]){
    try{
       System.out.println("First statement of try block");
       int num=45/3;
       System.out.println(num);
    }
    catch(ArrayIndexOutOfBoundsException e){
       System.out.println("ArrayIndexOutOfBoundsException");
    }
    finally{
       System.out.println("finally block");
    }
    System.out.println("Out of try-catch-finally block");
  }
}
```

```
First statement of try block
15
finally block
Out of try-catch-finally block
```

throw

- We can define our own set of conditions or rules and throw an exception explicitly using throw keyword.
- Throw keyword can also be used for throwing custom exceptions,

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

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

```java
/* In this program we are checking the Student age
 * if the student age<12 and weight <40 then our program
 * should return that the student is not eligible for registration.
 */
public class ThrowExample {
   static void checkEligibilty(int stuage, int stuweight){
      if(stuage<12 && stuweight<40) {
         throw new ArithmeticException("Student is not eligible for registration");
      }
      else {
         System.out.println("Student Entry is Valid!!");
      }
   }

   public static void main(String args[]){
     System.out.println("Welcome to the Registration process!!");
     checkEligibilty(10, 39);
     System.out.println("Have a nice day..");
 }
}
```

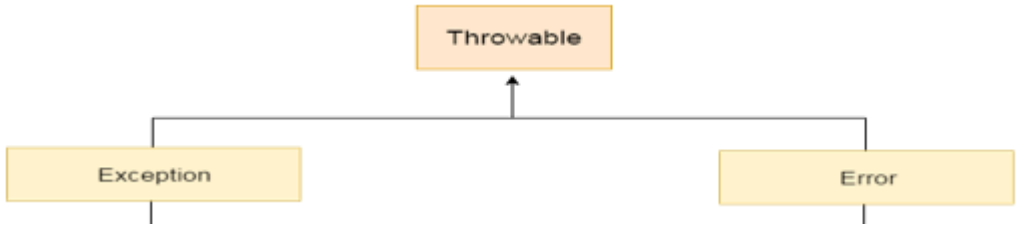
```
Welcome to the Registration process!!Exception in thread "main"
java.lang.ArithmeticException: Student is not eligible for registration
at beginnersbook.com.ThrowExample.checkEligibilty(ThrowExample.java:9)
at beginnersbook.com.ThrowExample.main(ThrowExample.java:18)
```

Example of throw

The custom exception gets called when the student age and weight is not within the expected limit

Creating user-defined exception

In java we can create our own exception class and throw that exception using throw keyword. These exceptions are known as user-defined or custom exceptions.




```
class MyException extends Exception{
   String str1;
   /* Constructor of custom exception class
    * here I am copying the message that we are passing while
    * throwing the exception to a string and then displaying
    * that string along with the message.
    */
   MyException(String str2) {
        str1=str2;
   }
   public String toString(){
        return ("MyException Occurred: "+str1) ;
   }
}
```

How to use the user-defined exception

1. User-defined exception must extend Exception class.
2. The exception is thrown using throw keyword.

```
class Example1{
   public static void main(String args[]){
        try{
                System.out.println("Starting of try block");
                // I'm throwing the custom exception using throw
                throw new MyException("This is My error Message");
        }
        catch(MyException exp){
                System.out.println("Catch Block") ;
                System.out.println(exp) ;
        }
   }
}
```

throws

- The Java **throws keyword is used to declare an exception**. It gives an **information to the programmer that there may occur an exception** so it is better for the programmer to provide the exception handling code so that normal flow can be maintained
- Using this approach is that **you will be forced to handle the exception** when you call this method, all the exceptions that are declared using throws, must be handled where you are calling this method else you will get compilation error.
- 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.

throws

```java
public void myMethod() throws ArithmeticException, NullPointerException
{
  // Statements that might throw an exception
}

public static void main(String args[]) {
  try {
    myMethod();
  }
  catch (ArithmeticException e) {
    // Exception handling statements
  }
  catch (NullPointerException e) {
    // Exception handling statements
  }
}
```

By using throws we can declare **multiple exceptions in one go**.

```
import java.io.*;
class ThrowExample {
  void myMethod(int num)throws IOException, ClassNotFoundException{
     if(num==1)
        throw new IOException("IOException Occurred");
     else
        throw new ClassNotFoundException("ClassNotFoundException");
  }
}

public class Example1{
  public static void main(String args[]){
   try{
     ThrowExample obj=new ThrowExample();
     obj.myMethod(1);
   }catch(Exception ex){
     System.out.println(ex);
    }
  }
}
```

```
java.io.IOException: IOException Occurred
```

Example throws

throw Vs throws

1. Throws clause is used to declare an exception, which means it works similar to the try-catch block.
2. syntax wise throws is followed by exception class names.
3. throws is used in method signature to declare the exceptions that can occur in the statements present in the method.
4. You can handle multiple exceptions by declaring them using throws keyword.

1. throw keyword is used to throw an exception explicitly.
2. Syntax wise throw is followed by an instance of Exception class
3. Throw keyword is used in the method body to throw an exception
4. You can throw one exception at a time

Namah Shivaya!