Day9: Regular Expression, Pattern Matchers, Exception Handling

Regular Expression:

A regular expression is a special sequence of characters that helps us to match or find other string or set of strings using a specialized syntax held in a pattern . it can be used to search, edit, and manipulate text and data.

Example:

- We can write a Regular Expression to represent all valid mail ids.
- We can write a Regular Expression to represent all valid mobile numbers.

The main important application areas of Regular Expression are:

- To implement validation logic.
- To develop Pattern matching applications.
- To develop translators like compilers, interpreters etc.

To represent and use Regular Expressions in Java applications, Java has provided a predefined library in the form of a package "**java.util.regex**".

The **Pattern** and **Matcher** class of this **java.util.regex** package provides the facility of java regular expression.

A Pattern object represents "compiled version of Regular Expression".

To create Regular Expression in the form of Pattern object we have to use the following method from **java.util.regex.Pattern** class.

public static Pattern compile(String regex);

Example:

```
Pattern p=Pattern.compile("ab");
```

A Matcher object can be used to match character sequences against a Regular Expression.

We can create a Matcher object by using the matcher() method of the Pattern class.

public Matcher matcher(String target);

Example:

```
Matcher m=p.matcher("abbbabbaba");
```

To get details about the Matches identified in the Matcher object we have to use the following methods of the Matcher object:

- **public boolean find():** it attempts to find the next match and returns true if it is available otherwise returns false.
- **public int start():** returns the starting index of the matched subsequence.
- **public int end():** returns the ending index of the matched subsequence.
- **public String group():** returns the matched subsequence.

Example:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

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

```
        int count = 0;

        Pattern p = Pattern.compile("ab");
        Matcher m = p.matcher("abbbabbaba");

        while (m.find()) {
            count++;
            System.out.println(m.start() + "------" + m.end() + "------" + m.group());
        }

        System.out.println("The no of occurences:" + count);
    }
}


output:
0------2------ab
4------6------ab
7------9------ab
The no of occurrences: 3
```

To prepare Regular Expressions, we have to use the following elements.

- Character Classes
- Quantifiers

Character Classes:

Character classes are used to specify alphabets and digits in Regular Expressions.

- [abc]----------Either 'a' or 'b' or 'c'
- [^abc] ----------Except 'a' and 'b' and 'c'
- [a-z] ----------Any lower case alphabet symbol
- [A-Z] ----------Any upper case alphabet symbol
- [a-zA-Z] ----------Any alphabet symbol
- [0-9] ----------Any digit from 0 to 9
- [a-zA-Z0-9] --------Any alphanumeric character
- [^a-zA-Z0-9] ------Any special character
- [a-z&&[^bc]] ------ a through z, except for b and c

- [a-z&&[^m-p]] ------ a through z, and not m through p

Regex Metacharacters:

The regular expression metacharacters work as shortcodes.

\s----space character
\d----Any digit from o to 9 [o-9]
\w----Any word character [a-zA-Z0-9]
. ----Any character including special characters.
\S----any character except for space character
\D----any character except for digit
\W----any character except for word character (special character)

Regex Quantifiers:

The quantifiers specify the number of occurrences of a character.

re* -- 0 or any number of occurrences of the preceding expression.

re+ -- 1 or more(at least) number of occurrences of the preceding expression.

re? -- 0 or 1(at most 1)number of occurrences of the preceding expression.

re{n} -- exactly n number of occurrence of the preceding expression.

re{n,} -- n or more occurrence of the preceding expression.

re{n, m} -- at least n and at most m.

a | b -- either a or b.

Example:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

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

        Pattern p = Pattern.compile("x");
        Matcher m = p.matcher("a7b@z#9");
        while (m.find()) {
```

```
            System.out.println(m.start() + "-------" + m.group());
        }
    }
}

The output of the above application with respect to the different values of x.

if x = [abc]

then output:
0.....a
2.....b

if x = [^abc]

then output:
1.....7
3.....@
4.....z
5.....#
6.....9

if x = [a-z]

then output:
0....a
2....b
4....z

if x = [0-9]

then output:
1......7
6......9


if x = [^a-zA-Z0-9]

then output:
3.....@
5.....#
```

Example 2:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

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

        Pattern p = Pattern.compile("x");
        Matcher m = p.matcher("a7b k@9");
        while (m.find()) {
```

```
            System.out.println(m.start() + "-------" + m.group());
        }
    }
}

The output of the above application with respect to the different values of x.

if x = \\s

then output:
3.... //space

if x = \\S

then output:
0.......a
1.......7
2.......b
4.......k
5.......@
6.......9

if x = .

then output:
0.......a
1.......7
2.......b
3.......
4.......k
5.......@
6.......9
```

Example 3:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

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

        Pattern p = Pattern.compile("x");
        Matcher m = p.matcher("abaabaaab");
        while (m.find()) {
            System.out.println(m.start() + "-------" + m.group());
        }
    }
}

The output of the above application with respect to the different values of x.

if x = a+ (at least one a or more a treated as a single match)

then output:
```

```
0.....a
2.....aa
5.....aaa


if x = a* (any number of a including zero number also)

then output:
0.....a
1.....
2.....aa
4.....
5.....aaa
8.....
9.....

if x = a? (at most one a either zero number or one a)

then output:
0.....a
1.....
2.....a
3.....a
4.....
5.....a
6.....a
7.....a
8.....
9.....
```

Pattern class matches method:

matches method of Pattern class is a static method. it works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern.

public static boolean matches(String regex, CharSequence input):

Example:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

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

      System.out.println(Pattern.matches("[amn]", "abcd"));//false (not a or m or n)
      System.out.println(Pattern.matches("[amn]", "a"));//true (among a or m or n)
```

```
      System.out.println(Pattern.matches("[amn]", "ammmna"));//false (m and a comes more than once)
    }
}
```

Example: Create a regular expression that accepts 10 digit numeric characters starting with 7, 8 or 9 only.

```
System.out.println(Pattern.matches("[789]{1}[0-9]{9}", "9953038949"));//true
System.out.println(Pattern.matches("[789][0-9]{9}", "9953038949"));//true

System.out.println(Pattern.matches("[789][0-9]{9}", "99530389490"));//false (11 characters)
System.out.println(Pattern.matches("[789][0-9]{9}", "6953038949"));//false (starts from 6)
System.out.println(Pattern.matches("[789][0-9]{9}", "8853038949"));//true

System.out.println("by metacharacters ...");
System.out.println(Pattern.matches("[789]{1}\\d{9}", "8853038949"));//true
System.out.println(Pattern.matches("[789]{1}\\d{9}", "3853038949"));//false (starts from 3)
```

Pattern class split method:

Pattern class contains split() method to split the given string against a regular expression.

Example:

```
import java.util.regex.Pattern;

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

        Pattern p = Pattern.compile("\\s");
        String[] s = p.split("Hello how are you");
        for(String s1:s) {
            System.out.println(s1);
        }
    }
}

output:
Hello
how
are
you
```

String Class split() Method:

The String class also contains split() method to split the given string against a regular expression.

Example:

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

        String str = "Hello how are you";
        String[] s = str.split("\\s");
        for(String s1:s) {
            System.out.println(s1);
        }
    }
}

output:
Hello
how
are
you
```

Exception Handling In Java:

The **Exception Handling in Java** is one of the powerful *mechanisms to handle the runtime errors*
so that the normal flow of the application can be maintained.

Exception In Java:

In Java, 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.

Technically in Java, an exception is an object of some java classes that will be created by the JVM at runtime whenever any Logical error occurs in our java application.

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 need to handle exceptions.

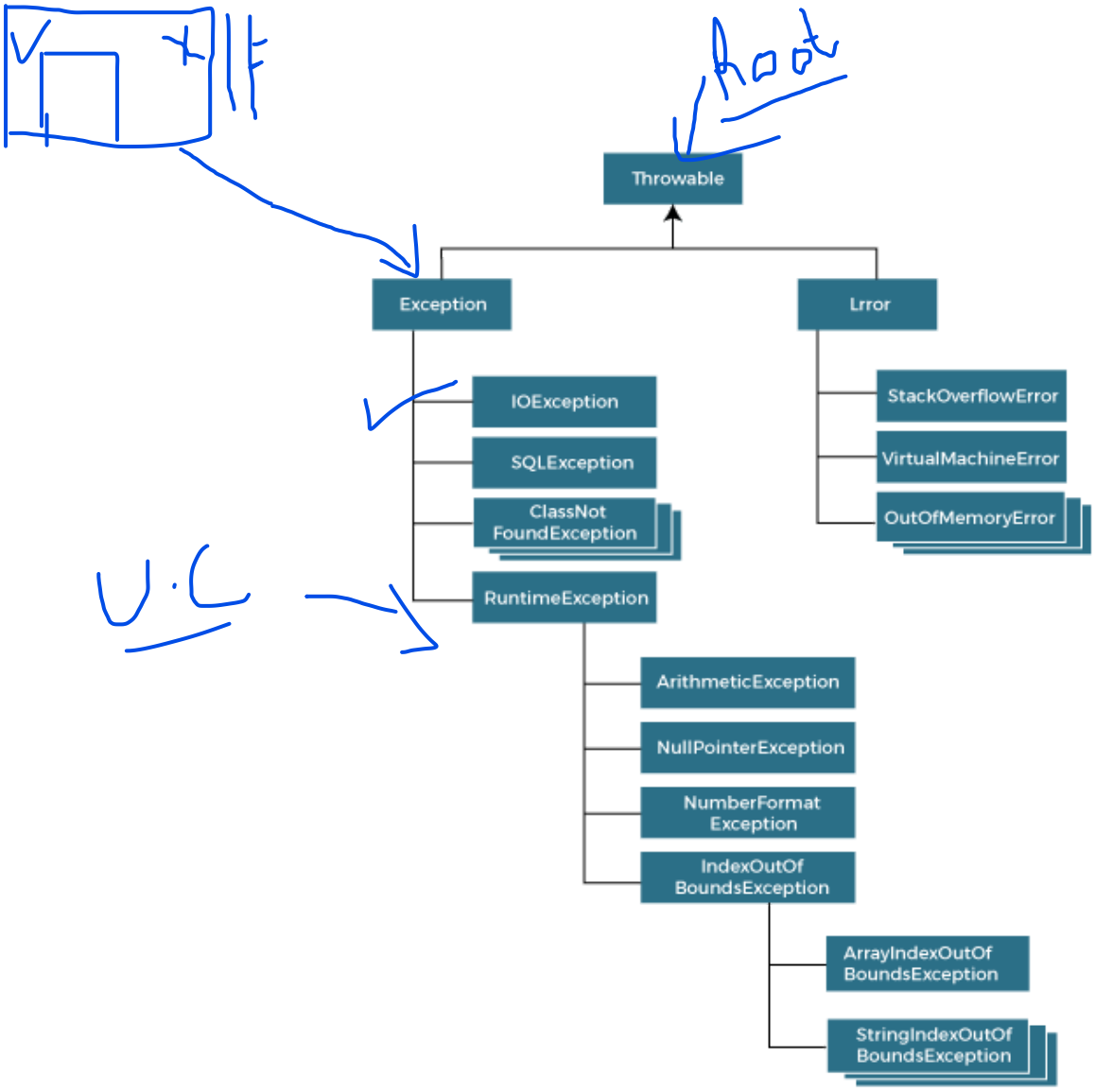
Example:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs a[10]=13; // size=5
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Suppose there are 10 statements in a Java program and an exception occurs at statement 5. the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



Error vs Exception:

- E**rror:** An Error indicates a serious problem that a reasonable application should not try to catch. Error is irrecoverable.
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.

Types of Exception:

There are mainly two types of exceptions: **checked** and **unchecked**. An **Error** is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception

2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions:

1. **Checked Exception:** The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
2. **Unchecked Exception:** The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

   The *RuntimeException* class is the superclass of all unchecked exceptions, so we can create a custom unchecked exception by extending *RuntimeException*:

```
public class NullOrEmptyException extends RuntimeException {
public NullOrEmptyException(String errorMessage) {
        super(errorMessage);
    }
}
```

Guidance on when to use checked exceptions and unchecked exceptions:

"If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception."

Java Exception Keywords:

Java provides five keywords that are used to handle the exception.

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.

Example: Without exception handling

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

      //code that may raise exception
      int data=100/0;
      System.out.println(data);
      //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

In the above application, at the line

```
int data = 100/0;
```

an exception occur at runtime and our application will be terminated abnormally in the middle.

In order to avoid this abnormal termination, we need to apply the concept of exception handling using **try and catch.**

Example: With exception handling

```
public class Main{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
      System.out.println(data);
```

```
     }catch(ArithmeticException e){
        System.out.println(e);
     }
    //rest code of the program
    System.out.println("rest of the code...");
   }
}
```

Common Scenarios of Java Exceptions:

There are given some scenarios where unchecked exceptions may occur. They are as follows:

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

   Example : **int** a=100/0;

   +,-,*./

2. **NullPointerException:** If we have a null value in any reference variable, performing any operation on that variable throws a NullPointerException.

   Example:

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

3. **NumberFormatException:** 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.

   Example:

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

4. **ArrayIndexOutOfBoundsException:** When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException.

   Example:

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

try-catch block:

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java **catch** block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

Note: Java try block must be followed by either catch or finally block.

Syntax:

```
try{
  //code that may throw an exception
}catch(Exception_class_Name ref){
  //alternate logic which we want to execute if exception occurs
}
```

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.

Handling the Exception using the parent class reference:

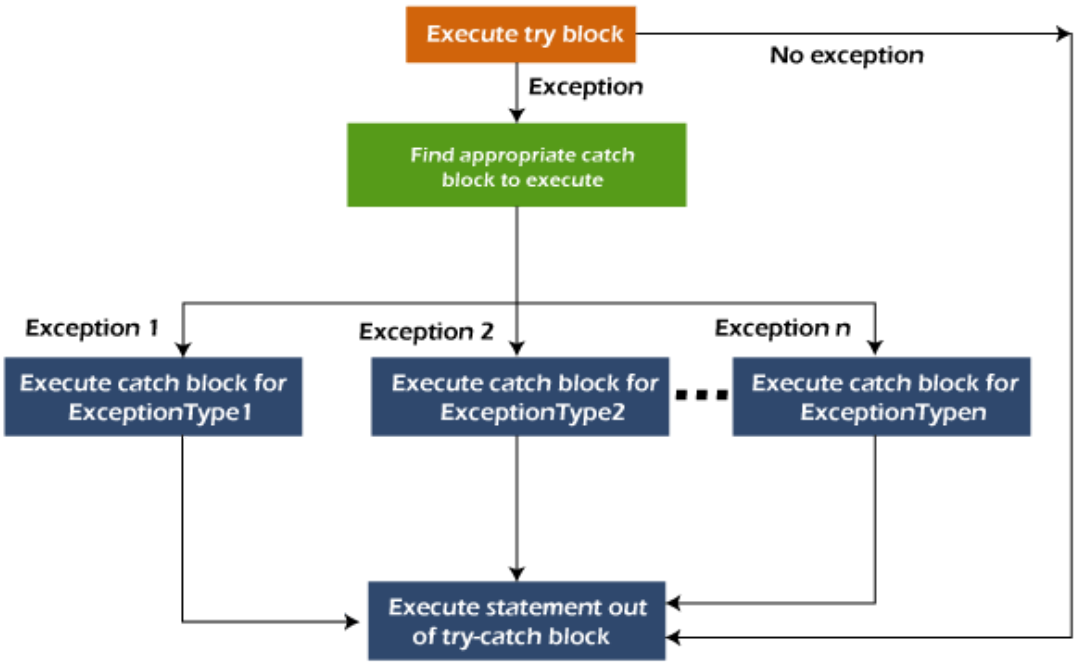
```
public class Main{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
      System.out.println(data);
     }catch(Exception e){     //Exception class is the parent class of ArithmaticException class
        System.out.println(e);
    }
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

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.

- 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.

  Siblings exception classes can be in any order, but the parent exception class must be at last.



Example:

```
public class Main{

    public static void main(String[] args) {

                try{
                 int a[]=new int[5];
                 a[6]=30/0;
                 System.out.println("End of try");
                }
                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
```

Example2:

```
public class Main{

    public static void main(String[] args) {

                try{
                 int a[]=new int[5];
                 a[6]=30/2;
                 System.out.println("End of try");
                }
                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
```

Example:

In this example, we generate NullPointerException but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class **Exception**
will be invoked.

```
public class Main{

    public static void main(String[] args) {

                try{

                 String s=null;
                 System.out.println(s.length());

                 System.out.println("End of try");
                }
                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:
Parent Exception occurs
rest of the code
```

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.

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.

For example, the **inner try block** can be used to
handle **ArrayIndexOutOfBoundsException**
while the **outer try block** can handle the **ArithemeticException** (division by zero).

Example:

```
//main try block: Level 1
try
{
    statement 1;
    statement 2;
    //try catch block within another try block: Level 2
    try
    {
        statement 3;
        statement 4;
        //try catch block within nested try block: Level 3
        try
        {
            statement 5;
            statement 6;
        }
        catch(Exception e1) //catch block of level 3
        {
          //exception message
        }

    }
    catch(Exception e2)  //catch block of level 2
    {
      //exception message
    }
}
//catch block of parent (outer) try block: lovel 3
catch(Exception e3)
{
//exception message
}
```

I Problem:

Let's consider the following example. Here the try block within the 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.

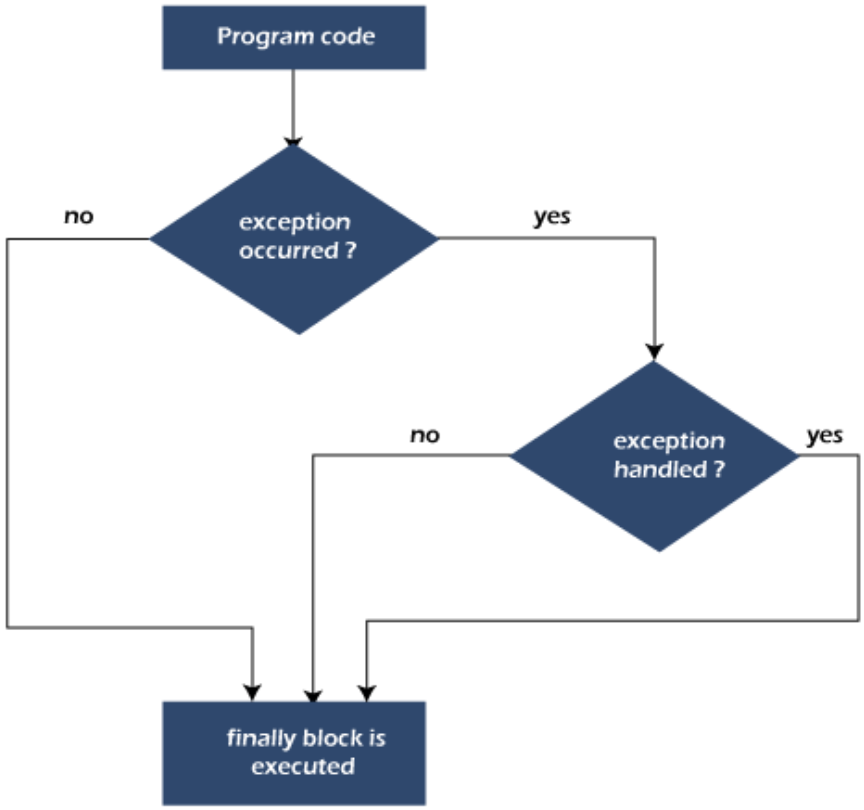
```
public class Main{

  public static void main(String args[]){

        try {  // outer (main) try block
```

```
            try {  //inner try block 1

                try {  // inner try block 2
                    int arr[] = { 1, 2, 3, 4 };

                    //printing the array element out of its bounds
                    System.out.println(arr[10]);
                }

                // to handles ArithmeticException
                catch (ArithmeticException e) {
                    System.out.println("Arithmetic exception");
                    System.out.println(" inner catch block 2");
                }
            }

            // to handle ArithmeticException
            catch (ArithmeticException e) {
                System.out.println("Arithmetic exception");
                System.out.println("inner catch block 1");
            }
        }

        // to handle ArrayIndexOutOfBoundsException
        catch (ArrayIndexOutOfBoundsException e4) {
            System.out.println(e4);
            System.out.println(" outer (main) try block");
        }
        catch (Exception e5) {
            System.out.println("Exception");
            System.out.println(" handled in main try-block");
        }

  }
}


Output:
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 4
 outer (main) try block
```

finally block:

The finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be executed regardless of the exception occurs or not.

- 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.



Example1: When Exception does not occurred

```
public class Main {
    public static void main(String args[]) {
        try {
            //below code do not throw any exception
            int data = 25 / 5;
            System.out.println(data);
        }
        //catch won't be executed
        catch (NullPointerException e) {
            System.out.println(e);
        }
        //executed regardless of exception occurred or not
        finally {
            System.out.println("finally block is always executed");
        }

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

Output:
5
```

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

Example1: When Exception occurred.

```
public class Main {
    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 occurred or not
            finally {
                System.out.println("finally block is always executed");
            }

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

Output:
Inside the try block
Exception in thread "main" java.lang.ArithmeticException: / by zero
finally block is always executed
```

Example 3: When exception is handled

```
public class Main {
    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 occurred or not
        finally {
            System.out.println("finally block is always executed");
        }

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

Output:
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

Note: For each try block there can be zero or more catch blocks, but only one finally block.

Finally not executed:

```
public class FinallyBlock {
   public static void main(String args[]) {
      try {
         System.out.println("I am in try block");
         System.exit(1);
      } catch(Exception ex){
         ex.printStackTrace();
      } finally {
         System.out.println("I am in finally block");
      }
   }
}
```

In the above example, the finally block will not execute due to the System.exit(1) condition in the try block.

A finally block will not execute due to other conditions like when JVM runs out of memory when our java process is killed forcefully from task manager or console when our machine shuts down due to power failure and deadlock condition in our try block.

throw keyword :

The **throw** keyword in java 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.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mostly used to throw a custom exception.

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.

Syntax:

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

We Problem:

Example 1: **Throwing Unchecked Exception**

Let’s create a validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class Main {

   //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 Main.validate(Main.java:10)
  at Main.main(Main.java:19)
```

Note: If we throw an unchecked exception from a method, it is not mandatory to handle the exception or declare in **throws** clause. whereas If we throw a checked exception using **throw** keyword, it is must to handle the exception using **try-catch block** or the method must declare it using **throws** declaration.

Example 2: Throwing Checked Exception

```
import java.io.IOException;
public class Main {

    //function to check if person is eligible to vote or not
    public static void validate(int age)throws IOException {
        if(age<18) {
            //throw IOException if not eligible to vote
            throw new IOException("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
        try {
            validate(13);
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("rest of the code...");
    }
}
```

Exception Propagation:

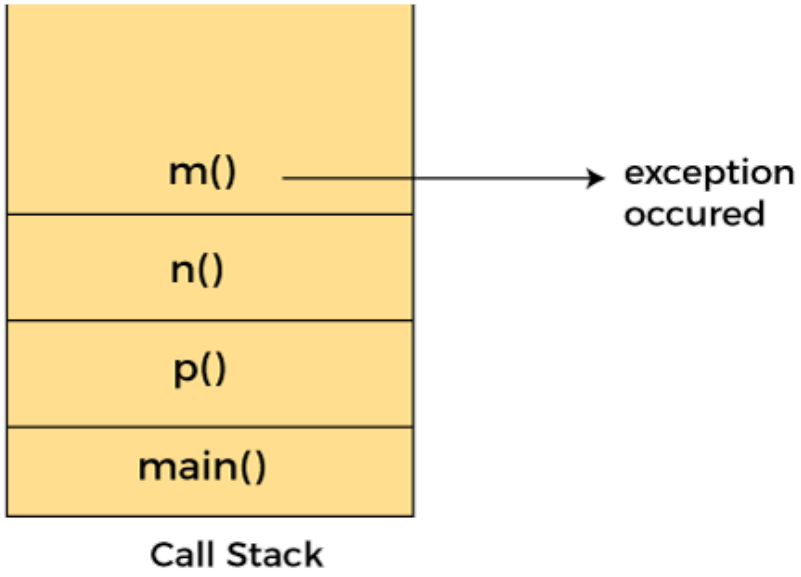
An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Example:

```
class Main {
    void m() {
        int data = 50 / 0;
    }

    void n() {
        m();
    }

    void p() {
        try {
            n();
        } catch (Exception e) {
            System.out.println("exception handled");
        }
    }

    public static void main(String args[]) {
        Main obj = new Main();
        obj.p();
        System.out.println("normal flow...");
    }
}

output:
exception handled
normal flow...
```

In the above example exception occurs in the m() method where it is not handled, so it is propagated to the previous n() method where it is not handled, again it is propagated to the p() method where exception is handled.

Exception can be handled in any method in call stack either in the main() method, p() method, n() method or m() method.



The throws keyword in java:

The **Java throws keyword** is used to declare an exception with the method signature . It gives information to the programmer/caller that there may occur an exception inside the method. So, it is better for the caller of that method to provide the exception handling code so that the normal flow of the program can be maintained.

It’s the responsibility of the calling method to act upon the exception. The thing is the called method will not do any try-catch or any exception handling.

Syntax :

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

Example:

Let's see the example of the Java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Main{
```

```
    void m()throws IOException{
        throw new IOException("device error");//checked exception
    }

    void n()throws IOException{
        m();
    }
    void p(){
        try{
            n();
        }catch(Exception e){
            System.out.println("exception handled");
        }
    }
    public static void main(String args[]){
        Main obj=new Main();
        obj.p();
        System.out.println("normal flow...");
    }
}

Output:
exception handled
normal flow...
```

Note: If we are calling a method that declares an exception, we must either catch or declare the exception.

Case 1 Example: Handle Exception Using try-catch block

```
import java.io.IOException;
class Main{
 void method()throws IOException{
  throw new IOException("device error");
 }
   public static void main(String args[]){
    try{
     Main m=new Main();
     m.method();
    }catch(Exception e){
      System.out.println("exception handled");
    }

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

Output:
exception handled
normal flow...
```

Case 2 Example: Declare Exception

- In case we declare the exception, if exception does not occur, the code will be executed fine.
- In case we declare the exception and the exception occurs, it will be thrown at runtime because **throws** does not handle the exception.

```
import java.io.IOException;
class Main{
 void method()throws IOException{
  System.out.println("device operation performed");
  // throw new IOException("device error");
 }
   public static void main(String args[])throws IOException{

     Main m=new Main();
     m.method();

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

Output:
device operation performed
normal flow...
```

Difference between throw and throws:

The **throw** keyword throws the exception explicitly from a method or a block of code whereas the **throws** keyword is used in the signature of the method.

There are many differences between **throw** and **throws** keywords:

throw	**throws**
Throw is a keyword which is used to throw an exception explicitly in the program inside a function or inside a block of code.	Throws is a keyword used in the method signature used to declare an exception which might get thrown by the function while executing the code.
Internally throw is implemented as it is allowed to throw only a single exception object at a time i.e we cannot throw multiple exception objects with throw keyword.	On other hand we can declare multiple exceptions with throws keyword that could get thrown by the function where throws keyword is used.
With throw keyword we can propagate only unchecked exception i.e checked exception cannot be propagated using throw.	On other hand with throws keyword both checked and unchecked exceptions can be declared and for the propagation checked exception must use throws keyword followed by specific exception class name.

The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.

You Problem:

- Write a class Main, in which define a static method called getEligibility.
- the return type of this method should be void and it takes one parameter as int age.
- If the supplied age is less than 18 then throw an ArithmeticException class object with the message “Invalid age”.
- If the age is greater than 18 then print the message “You are Eligible”.
- from the main method of Main class, prompt the user to Enter the age using Scanner class.
- call the getEligibility method inside the try-catch block from the main method by passing user-entered age

Done till here ..... @Ratan Lal Gupta

Method overriding rule with exception handling:

- **If the superclass method does not declare an exception using “throws”**
  - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- **If the superclass method declares an exception using “throws”**
  - If the superclass method declares an exception, the subclass overridden method can declare same, subclass exception of declared exception or no exception but cannot declare parent exception of declared exception.

User-defined or custom exception in java:

Java provides almost all the general types of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

Till now we have used predefined exception classes whose objects were created by the JVM because of predefined logical error.

In a similar manner, we can define our own exception classes whose objects will be created when user-defined logical error occurs.

strictly speaking the user defined logical error is not a logical error from the JVM's point of view.

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as a custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Example:

```
InvalidFileNameException.java

public class InvalidFileNameException extends Exception {

}
```

In order to make the InvalidFileNameException class more flexible we can define 2 overloaded constructors also.

Example:

```
InvalidFileNameException.java

public class InvalidFileNameException extends Exception {

    public InvalidFileNameException(String errorMessage) {
    super(errorMessage);
    }
    public InvalidFileNameException() {
    }
}
```

Note: If our User-defined exception class extends from the j**ava.lang.Exception** class then our custom exception class will become the **checked-exception** class.

whereas if our user-defined exception class extends from the **java.lang.RuntimeException** class then our custom exception class will become the **unchecked-exception.**

Example:

```
InvalidAgeException.java:

public class InvalidAgeException extends Exception{

    InvalidAgeException(String errorMessage){
        super(errorMessage);
    }

}

Employee.java:

import java.util.Scanner;

class Employee{
    public int getPension(int age,int sal)throws Exception
    {
        int pension = 0;
        if(age>45 && age <100){
            pension=(age * sal)/100;
        }
        else{
            InvalidAgeException ae=new InvalidAgeException("Invalid Age");
            throw ae;
        }
        return pension;
    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter age");
        int age = sc.nextInt();

        System.out.println("Enter salary");
        int salary = sc.nextInt();

        Employee emp = new Employee();
        try {
            emp.getPension(age,salary);
        } catch (Exception e) {
            //e.printStackTrace();
            System.out.println(e.getMessage());
        }


    }
}
```

References:

https://www.javatpoint.com

https://www.geeksforgeeks.org/

https://www.baeldung.com/java-checked-unchecked-exceptions