Lecture-9

Exceptions Handling

Java Exceptions Handling

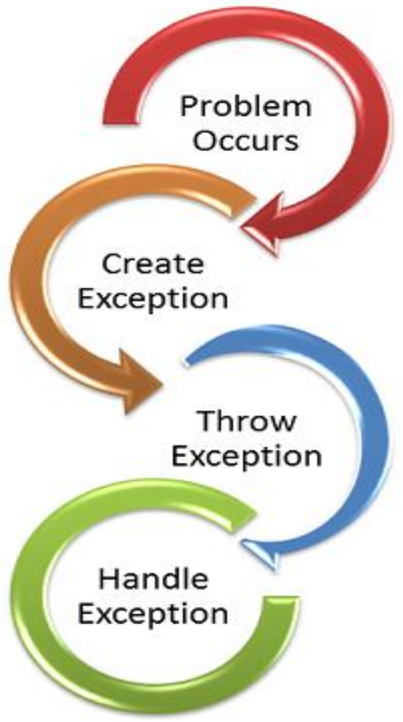
```
try{

}catch( Exception ){
    //Do nothing
}
```

Exception Handling

- Exception is an **abnormal condition**.
- An exception (or exceptional event) is a problem that arises **during the execution of a program**.
- In java, exception is an event that **disrupts the normal flow** of the program.
- It is an object which is thrown at **runtime**.
- Exception Handling is a **mechanism** to handle **runtime errors** such as:

  ClassNotFound, IO, SQL, Remote etc.



Java Exceptions

An exception can occur for many different reasons, below given are some scenarios where exception occurs.

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

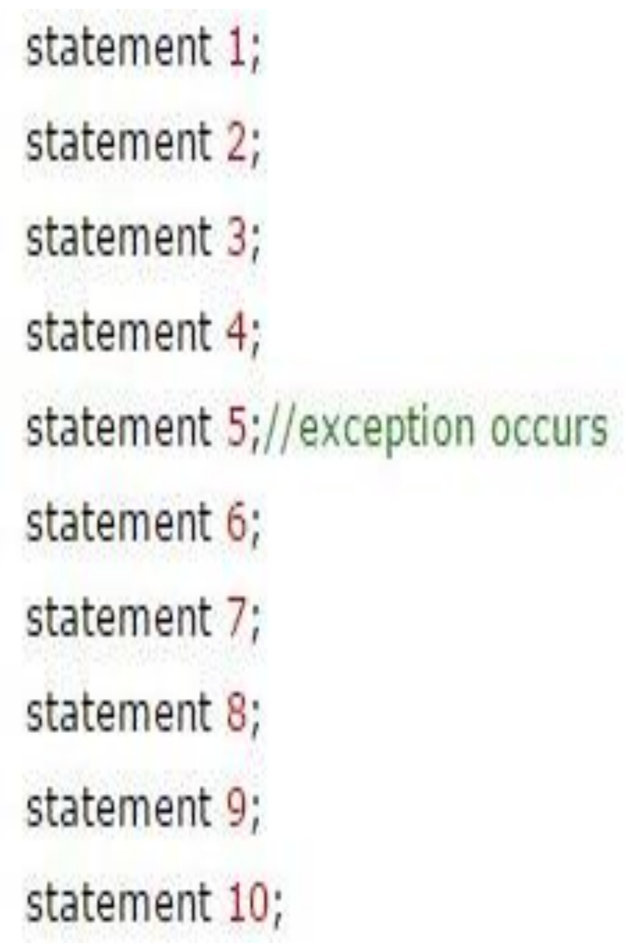
Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application.**

Let's take a scenario:

- Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run.
- If we include exception handling here, rest of the statement will be executed. That is why we use exception handling in java.



```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Types of Exception

There are mainly two types of exceptions: **checked** and **unchecked** where error is considered as unchecked exception.

Whereas the sun microsystem says there are **three types** of exceptions:

- **Checked Exception**
- **Unchecked Exception**
- **Error**

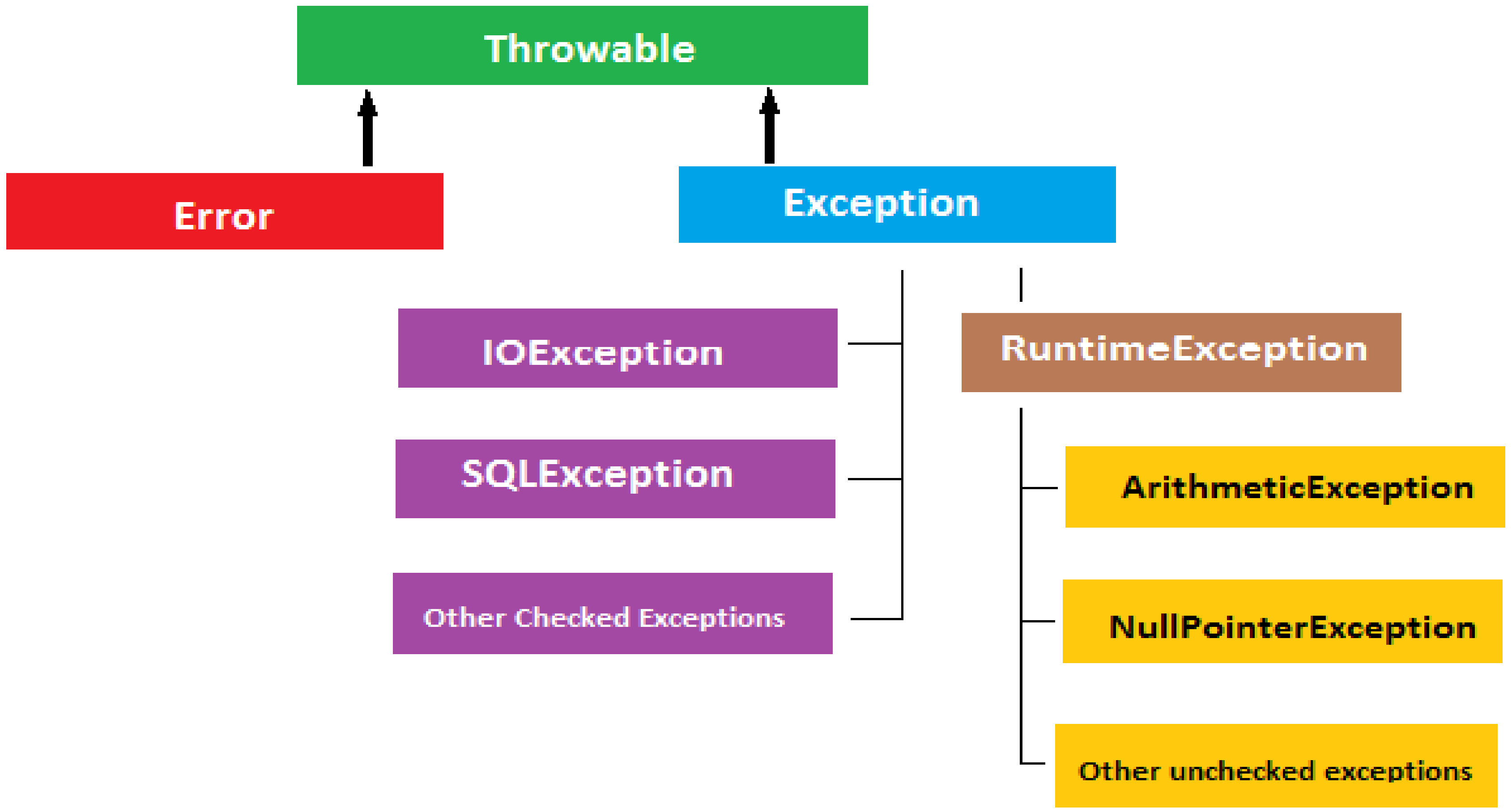
Checked Exceptions

- All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation.
- A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions.
- Some checked exceptions are as follows:
  - ClassNotFoundException
  - IllegalAccessException
  - NotSuchFieldException
  - EOFExceptionException

Unchecked Exceptions

- Runtime Exceptions are also known as Unchecked Exceptions as the compiler do not check whether the programmer has handled them or not.
- These exceptions not need to be included in any method's throws list because compiler does not check to see if a method handles or throws these exceptions.
- Some unchecked exceptions are as follows:
  - **ArithmaticException**
  - **ArrayIndexOutOfBoundException**
  - **NullPointerException**
  - **NegativeArraySizeException**

Exception Classes



Scenarios Where Exceptions May Occur

1. **Scenario Where ArithmeticException Occurs:**

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

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

2. **Scenario where NullPointerException occurs**

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

3. **Scenario where ArrayIndexOutOfBoundsException occurs**

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

Example 1: ArrayIndexOutOfBounds Exception

```
public class Unchecked_Demo {

   public static void main(String args[]) {
         int num[]={1,2,3,4};
         System.out.println(num[5]);
   }
}
```

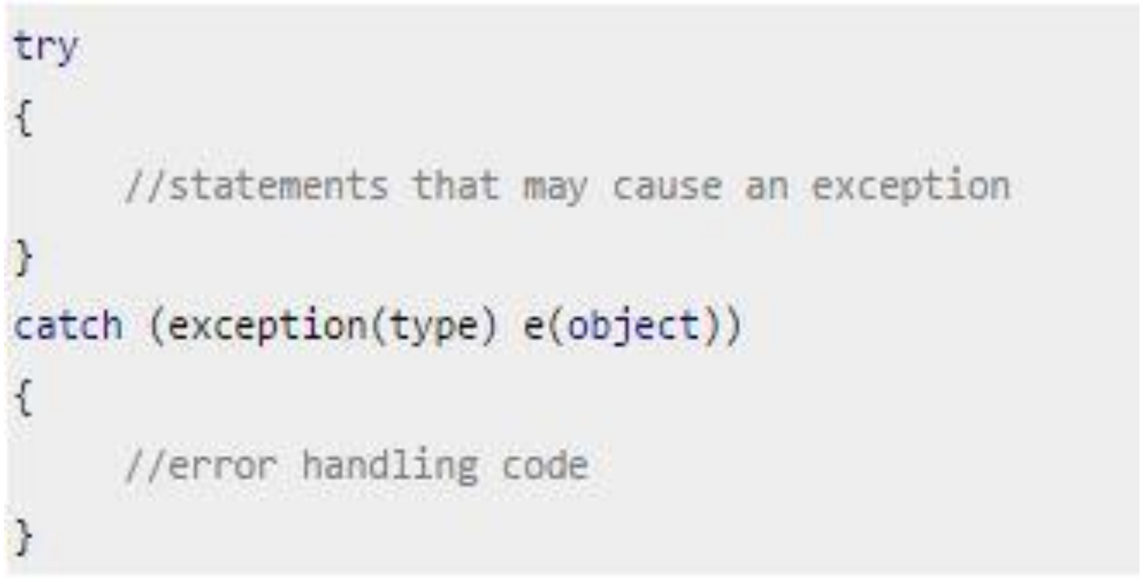
If you compile and execute the above program you will get exception as shown below.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
```

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

- try
- catch
- finally
- throw
- throws



```
try
{
    //statements that may cause an exception
}
catch (exception(type) e(object))
{
    //error handling code
}
```

Syntax of try catch in java

Problem Without Exception Handling

Let's try to understand the problem if we don't use try-catch block.

```
public class Testtrycatch1{
    public static void main(String args[]){
        int data=50/0;                    //may throw exception
        System.out.println("rest of the code...");
    }
}
```

Output:

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

Solution by Using Exception Handling

Let's see the solution of above problem by java **try-catch** block.

```
public class Testtrycatch2{
    public static void main(String args[]){
        try{
            int data=50/0;
          }
        catch (ArithmeticException e)
          {
            System.out.println(e);
          }
        System.out.println("rest of the code...");
          }
    }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

End of Part-1

Exceptions Handling
Part-2

Flow of Control in Try/Catch Blocks:

When exception doesn't occur

In the given example exception didn't occur in try block so catch block didn't run

```
int x = 10;
int y = 10;
try {
   int num= x/y;
   System.out.println("Inside try block");
}
catch (Exception ex)
 {
    System.out.println("Exception");
 }
System.out.println("next-statement: Outside of try-catch");
```

Output:

```
Inside try block
next-statement: Outside of try-catch
```

Flow of Control in Try/Catch Blocks:

When exception occur

There are two statements present inside try block. Since exception occurred, because of first statement (x/y), the second statement didn't execute.

Hence we can conclude that if an exception occurs then the rest of the try block doesn't execute and control passes to catch block.

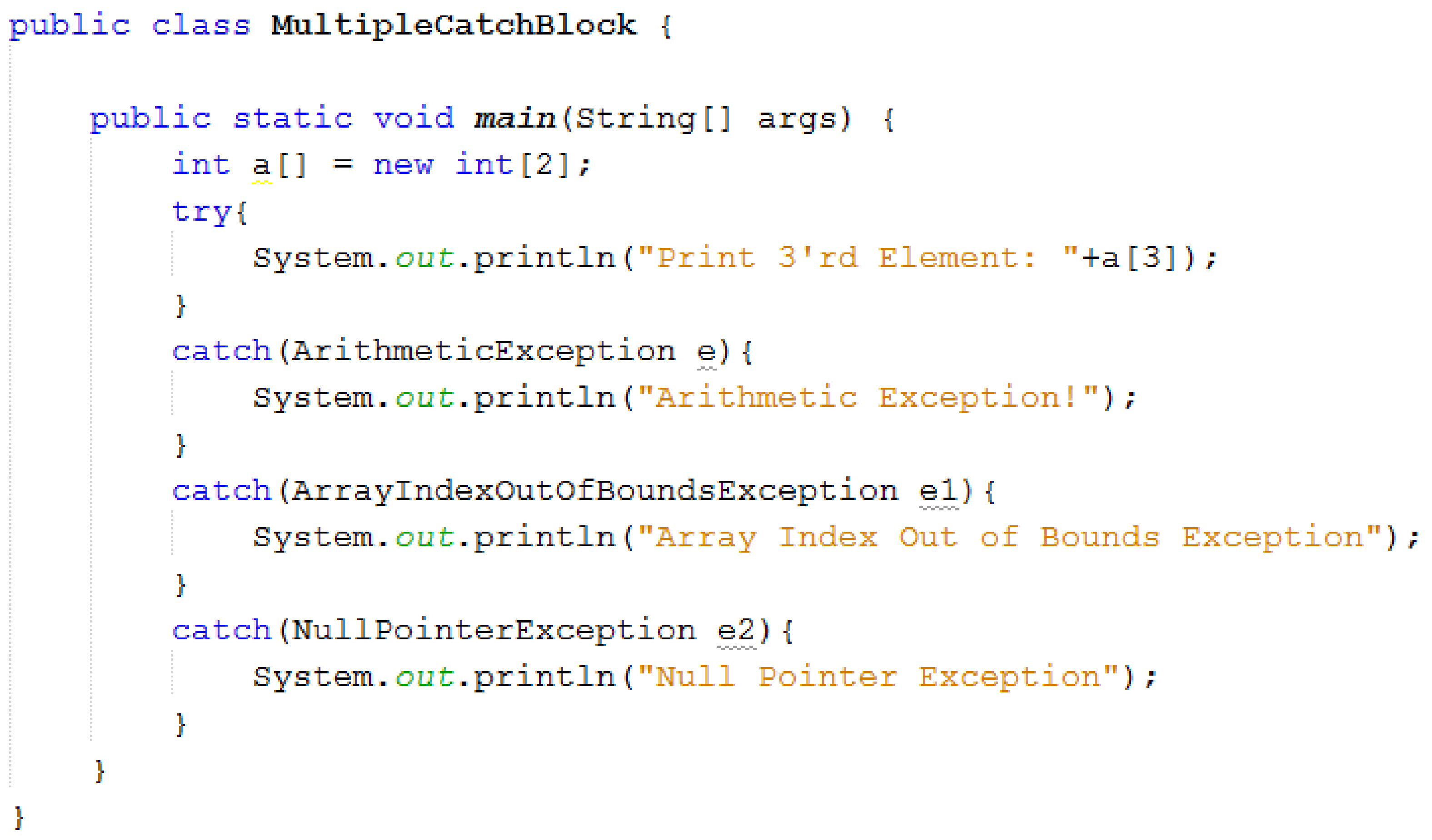
```
int x = 10;
int y = 0;
try {
   int num= x/y;
   System.out.println("Inside try block");
}
catch (Exception ex)
 {
    System.out.println("Exception Occurred");
 }
System.out.println("next-statement: Outside of
try-catch");
```

Output:

```
      Exception Occurred
      next-statement: Outside of try-catch
```

Multiple Catch Blocks

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



```java
public class MultipleCatchBlock {

    public static void main(String[] args) {
        int a[] = new int[2];
        try{
            System.out.println("Print 3'rd Element: "+a[3]);
        }
        catch(ArithmeticException e){
            System.out.println("Arithmetic Exception!");
        }
        catch(ArrayIndexOutOfBoundsException e1){
            System.out.println("Array Index Out of Bounds Exception");
        }
        catch(NullPointerException e2){
            System.out.println("Null Pointer Exception");
        }
    }
}
```

Output: Array Index Out Of Bounds Exception

The Finally Block

The finally block follows a try block or a catch block.

- A finally block of code always executes, instead the occurrence of an Exception.
- Using a finally block allows you to run any statements that you want to execute, no matter what happens in the protected code (in the try block).

Syntax of Using finally block

Use of finally block at the end of the try blocks

```
try
{
   //statements that may cause an exception
}
finally
{
  //statements to be executed
}
```

Use of finally block at the end of the catch blocks

```
try
{
  //statements that may cause an exception
}
catch(ExceptionType1 e1)
{
  //Catch block
}
finally
{
  //The finally block always executes.
}
```

Example 1: Use of Java finally

Let’s see the different cases where java finally block can be used.

Case 1:

Where Exception doesn’t occur

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   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:5
       finally block is always executed
       rest of the code...
```

Example 2: Use of Java finally

Case 2:

Where Exception occur but doesn't handle

```
class TestFinallyBlock1{
  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: Use of Java finally

Case 3:

Where Exception occur and handled

```
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...
```

Thank You