

Exception Handling

- The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.
- **What is exception?**
- Exception is an abnormal condition.
- **What is exception handling?**
- Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IO, SQL, Remote etc.

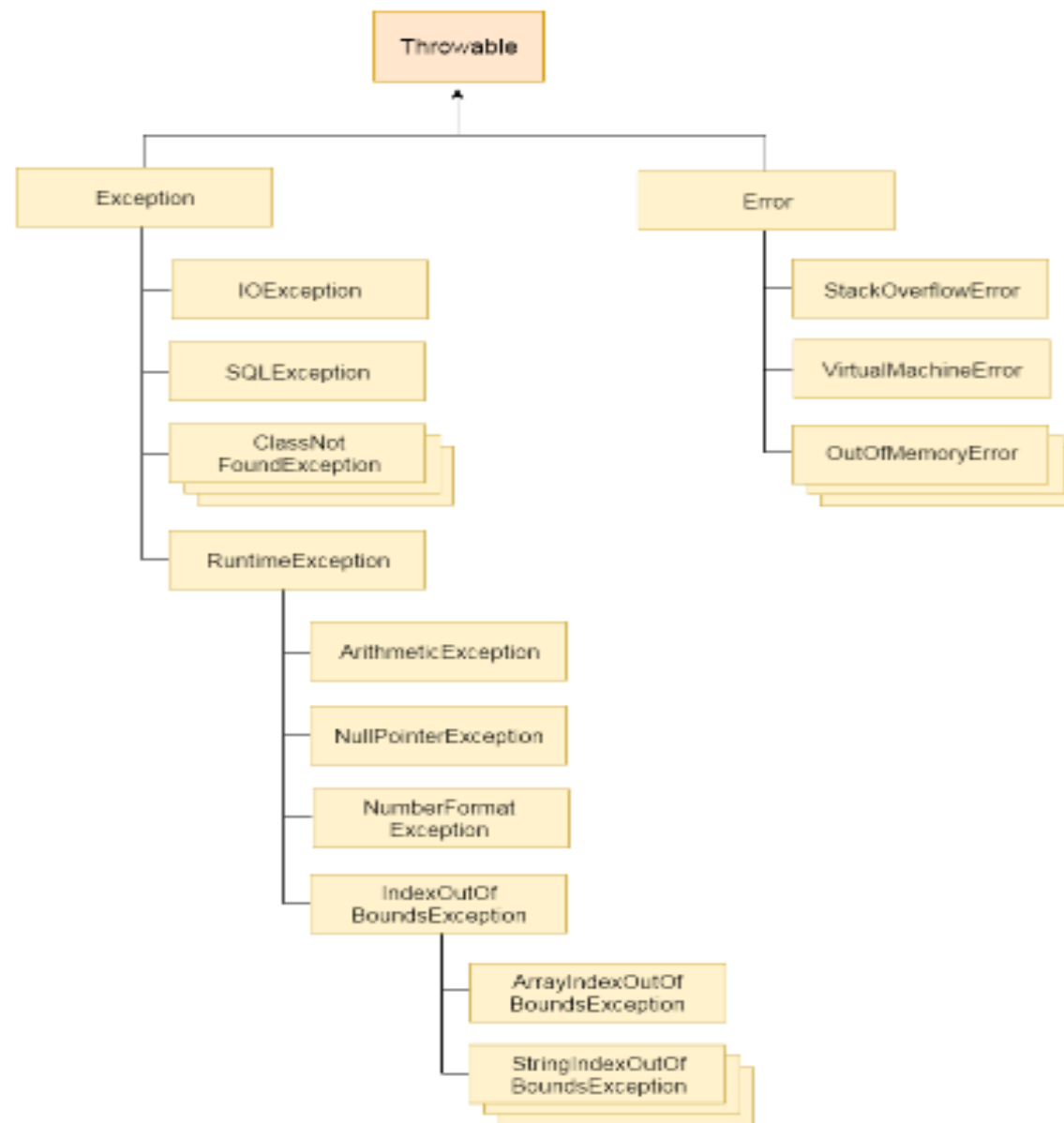
Advantage of Exception Handling

- The core advantage of exception handling is **to maintain the normal flow of the application.**
- Exception normally disrupts the normal flow of the application that is why we use exception handling.

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

Suppose there are 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 perform exception handling, rest of the statement will be executed.

Hierarchy of Java Exception classes



Types of Exception

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

➤ Checked Exception

- Checked exceptions are checked at compile-time.
- Checked exceptions are exceptions that the Java compiler requires us to handle.
- e.g: IOException, SQLException etc.

➤ Unchecked Exception

- Unchecked exceptions are not checked at compile-time rather they are checked at runtime.
- Unchecked exceptions are exceptions that the Java compiler does *not* require us to handle.
- e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Common 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

- If we have null value in any variable, performing any operation by the variable occurs an `NullPointerException`.
- `String s=null;`
- `System.out.println(s.length());//NullPointerException`

➤ 3) Scenario where **NumberFormatException** occurs

- The wrong formatting of any value, may occur **NumberFormatException**.
- Suppose if we have a string variable that have characters, converting this variable into digit will occur **NumberFormatException**.

```
String s="123";
```

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

➤ 4) Scenario where **ArrayIndexOutOfBoundsException** occurs

- If you are inserting any value in the wrong index, it would result **ArrayIndexOutOfBoundsException** as shown below:

```
int a[]=new int[5];
```

```
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Exception Handling Keywords

- There are 5 keywords used in java exception handling.
- try
- catch
- finally
- throw
- throws

Java try-catch

- Java try block is used to enclose the code that might throw an exception. It must be used within the method.
- Java try block must be followed by either catch or finally block.

Syntax of try-catch block

```
try{  
    //code that may throw exception  
}  
catch(Exception_class_Name ref)  
{  
}
```

Syntax of try-finally block

```
try{  
    //code that may throw  
    exception  
}  
finally  
{  
}
```

Java catch block

- Java catch block is used to handle the Exception. It must be used after the try block only.
- We can use multiple catch blocks with a single try.

Problem without exception handling

```
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 exception handling

```
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...

Java Multi catch block

```
public class TestMultipleCatchBlock{  
    public static void main(String args[]){  
        try{  
            int a[]=new int[5];  
            a[4]=30/0;  
        }  
        catch(ArithmeticException e){System.out.println("task1 is completed");}  
        catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}  
        catch(Exception e){System.out.println("common task completed");}  
  
        System.out.println("rest of the code...");  
    }  
}
```

Output:

task1 completed
rest of the code...

Note

- At a time only one Exception is occurred 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` .
- For each try block there can be zero or more catch blocks, but only one finally block.

Guess the output?

```
class TestMultipleCatchBlock1{
    public static void main(String args[]){
        try{
            int a[]=new int[5];
            a[5]=30/0;
        }
        catch(Exception e){System.out.println("common task completed");}
        catch(ArithmeticException e){System.out.println("task1 is completed");}
        catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}

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

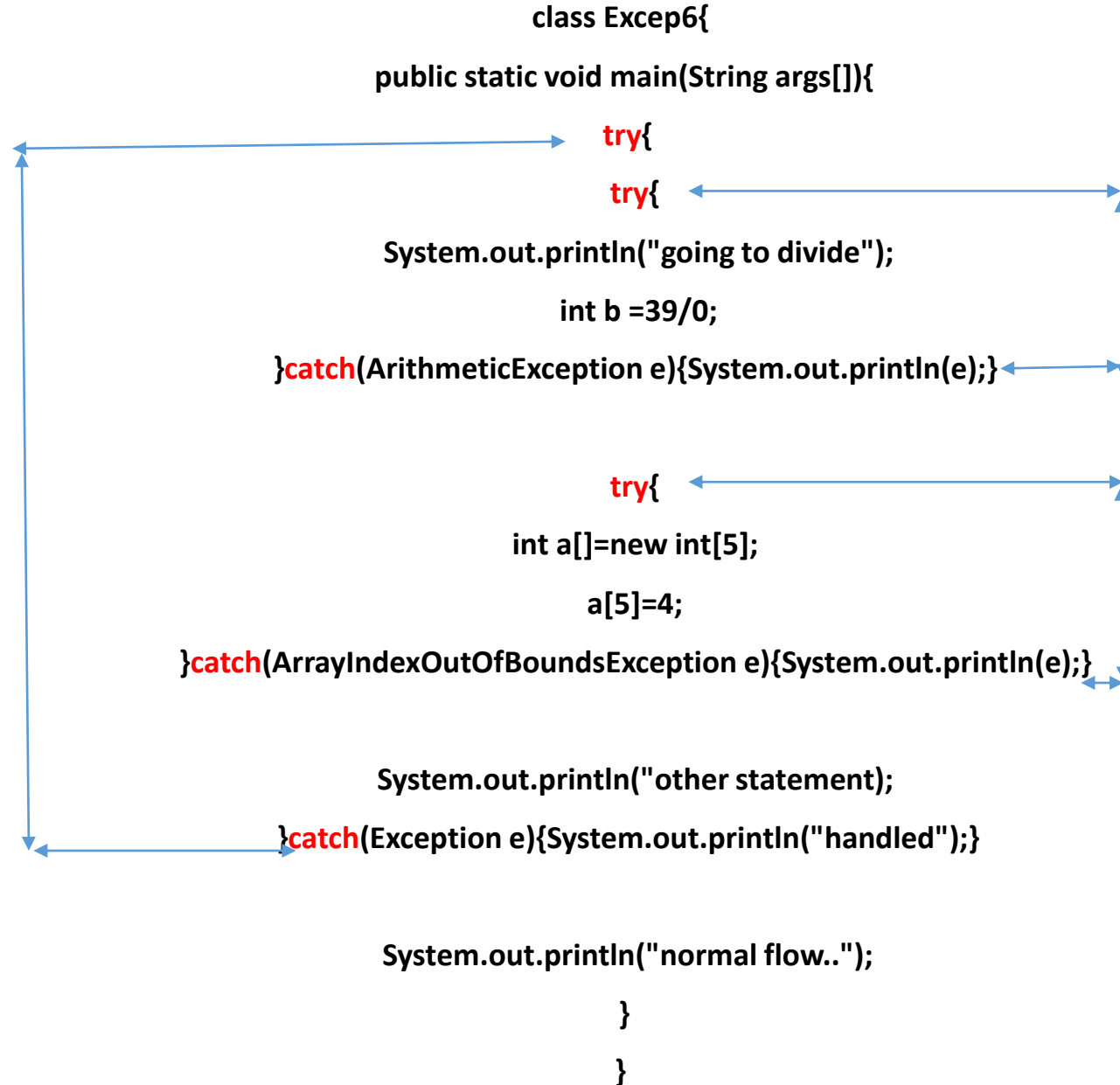
Java Nested try block

- The try block within a try block is known as nested try block in java.
- **Why use 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.

- **Syntax:**

```
....  
try  
{  
    statement 1;  
    statement 2;  
    try  
    {  
        statement 1;  
        statement 2;  
    }  
    catch(Exception e)  
    {  
    }  
}  
catch(Exception e)  
{  
}  
....
```


Java nested try example

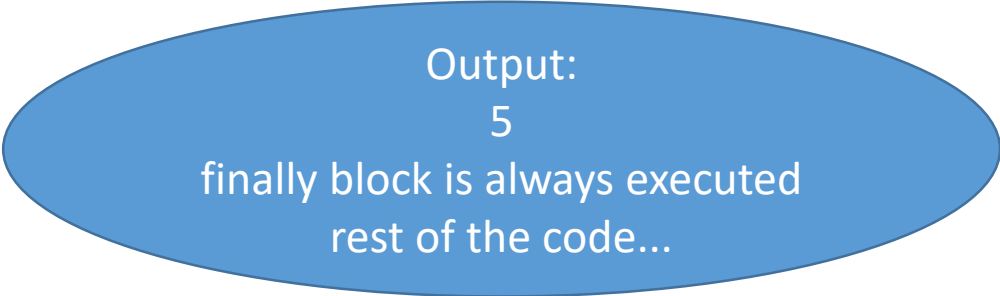


Java finally block

- **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.

Usage of Java finally

```
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...

Java throw keyword

- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or unchecked exception in java by throw keyword.
- The throw keyword is mainly used to throw custom exception.
- The syntax of java throw keyword is given below.
- **throw** exception;
- **throw new** IOException("sorry device error);

java throw keyword example

```
public class TestThrow1{  
    static void validate(int age){  
        if(age<18)  
            throw new ArithmeticException("not valid");  
        else  
            System.out.println("welcome to vote");  
    }  
    public static void main(String args[]){  
        validate(13);  
        System.out.println("rest of the code...");  
    }  
}
```

- **Output: Exception in thread main java.lang.ArithmeticException:not valid**

Java throws keyword

- 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.
- **Syntax of java throws**

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

Java throws example

```
import java.io.IOException;

class Testthrows1{
    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[]){
        Testthrows1 obj=new Testthrows1();
        obj.p();
        System.out.println("normal flow...");
    }
}
```

Output:
exception handled
normal flow...

Difference between throw and throws in Java

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

Java Custom Exception (User Defined Exception)

- 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.

Example

```
class InvalidAgeException extends Exception{  
    InvalidAgeException(String s){  
        super(s);  
    }  
}  
  
class TestCustomException1{  
    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 occurred: "+m);}  
        System.out.println("rest of the code...");  
    }  
}
```

**Output:Exception occurred:
InvalidAgeException:not valid
rest of the code...**