Exception Handling in Java

Exception Handling in Java :

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 in Java ?

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

Exception Handling in Java

What is Exception Handling ?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

Advantage of Exception Handling

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:

Exception Handling in Java

Exception Handling Example

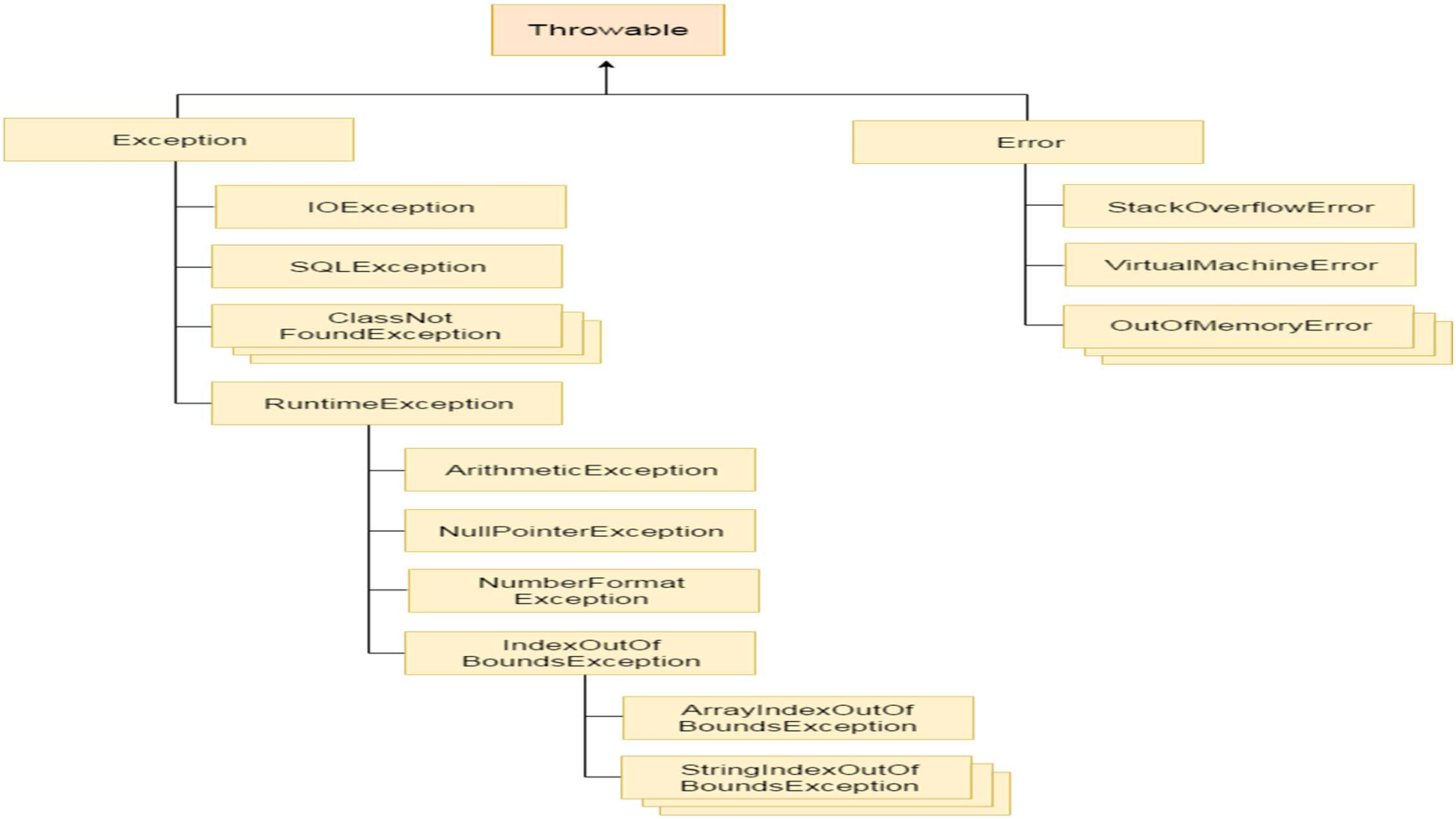
```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Exception Handling Example

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

Exception Handling in Java

Hierarchy of Java Exception classes :



Exception Handling in Java

Types of Java Exceptions :

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

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

Exception Handling in Java

Difference between Checked and Unchecked Exceptions

1) 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.

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.

Difference between Checked and Unchecked Exceptions

3) Error :

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Exception Handling in Java

Java Exception Keywords :

Keyword	Description
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.
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.

Exception Handling in Java

Java Exception Handling Example

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

Output:

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

Exception Handling in Java

Common Scenarios of Java Exceptions :

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

1) A scenario where ArithmeticException occurs

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

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

Exception Handling in Java

Common Scenarios of Java Exceptions :

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

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

Exception Handling in Java

Common Scenarios of Java Exceptions :

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

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

Exception Handling in Java

Common Scenarios of Java Exceptions :

4) A scenario where ArrayIndexOutOfBoundsException occurs

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

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

Exception Handling in Java

Java try-catch :

Java try block :

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 java try-catch

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

Exception Handling in Java

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.

You can use multiple catch block with a single try.

Exception Handling in Java

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

Exception Handling in Java

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...");
}}
```

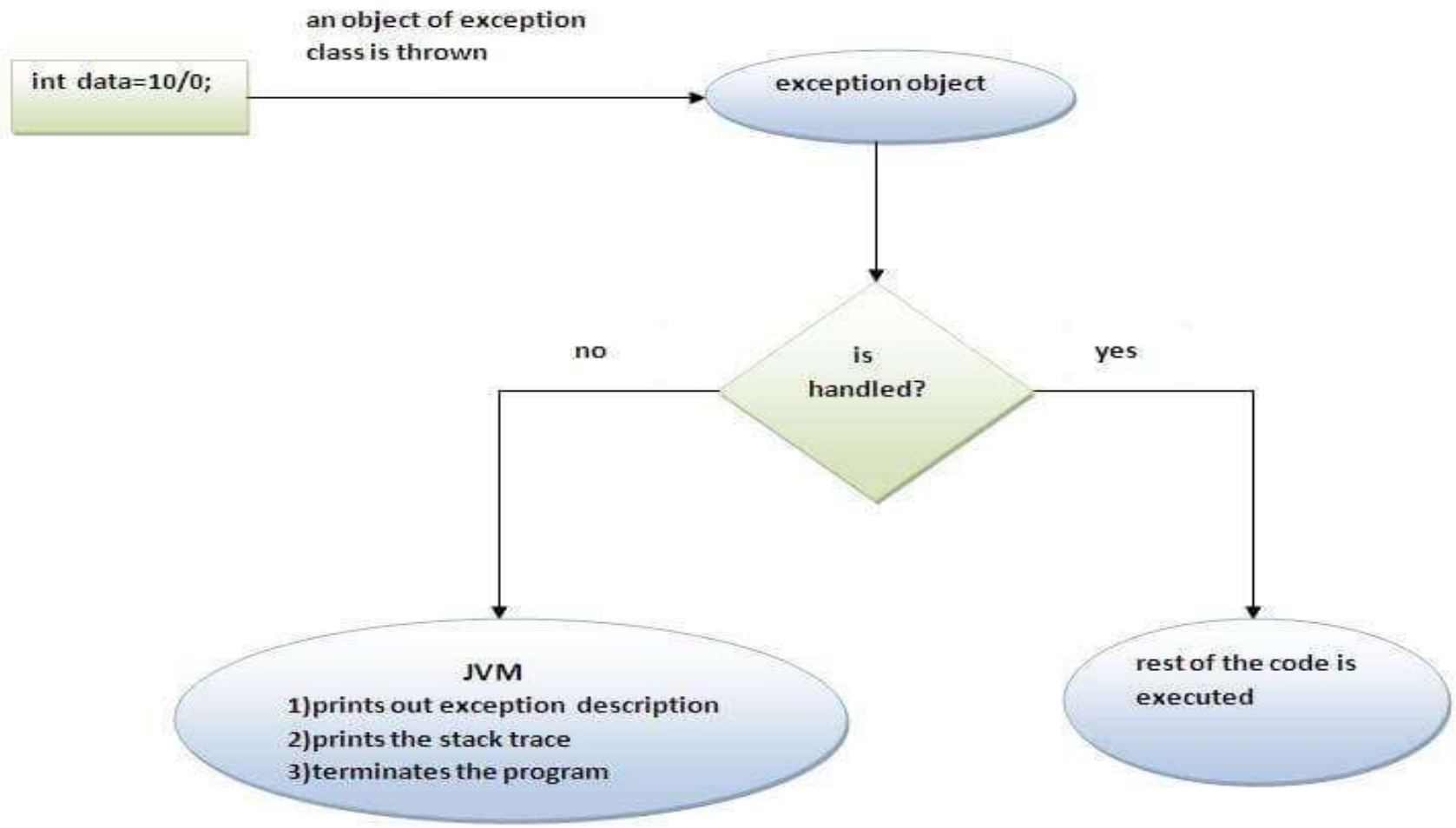
Output:

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

rest of the code...

Exception Handling in Java

Internal working of java try-catch block :



Exception Handling in Java

Java Multi catch block :

```
public class TestMultipleCatchBlock{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=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..."); }}
```

Exception Handling in Java

Java Multi catch block :

Output:task1 completed
rest of the code...

Rule: At a time only one Exception is occured and at a time only one catch block is executed.

Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

Exception Handling in Java

Java Multi catch block :

```
class TestMultipleCatchBlock1
{
  public static void main(String args[])
  {
   try
   {
    int a[]=new int[5];
    a[5]=30/0;
   }
```

Exception Handling in Java

Java Multi catch block :

```
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..."); } }
```

Output:

Compile-time error

Exception Handling in Java

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.

Exception Handling in Java

Syntax:

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

Exception Handling in Java

Java nested try example :

```
class Excep6{
 public static void main(String args[]){
  try{
    try{
     System.out.println("going to divide");
     int b =39/0;
    }catch(ArithmeticException e){System.out.println(e);}
```

Exception Handling in Java

Java nested try example :

```
 try{
    int a[]=new int[5];
    a[5]=4;
    }catch(ArrayIndexOutOfBoundsException
e){System.out.println(e);}

    System.out.println("other statement);
  }catch(Exception e){System.out.println("handeled");}

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

Exception Handling in Java

Java finally block :

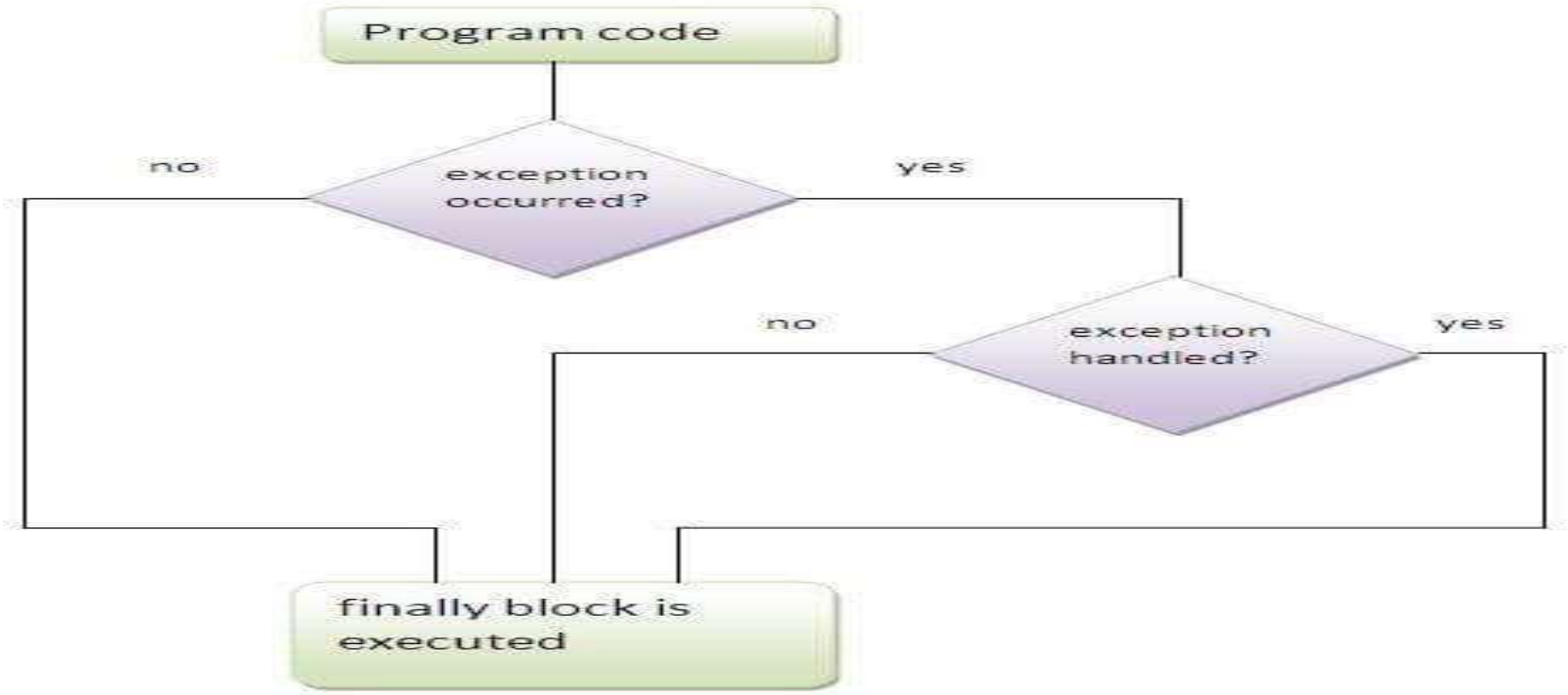
Java finally block is a block that is used to execute important code such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.

Exception Handling in Java

Java finally block :



Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

Exception Handling in Java

Why use java finally ?

Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally :

```
class TestFinallyBlock{
    public static void main(String args[]){
    try{
     int data=25/5;
     System.out.println(data);
    }
```

Exception Handling in Java

Usage of Java finally :

```
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...

Exception Handling in Java

Usage of Java finally :

```
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

Exception Handling in Java

Usage of Java finally :

```
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...");
  }
}
```

Exception Handling in Java

Usage of Java finally :

Output:

```
        Exception in thread main
java.lang.ArithmeticException:/ by zero
        finally block is always executed
        rest of the code...
```

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

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

Exception Handling in Java

Java throw keyword :

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked 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;
```

Exception Handling in Java

Java throw keyword :

```
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...");
 } }
```

Exception Handling in Java

Java throw keyword :

Output:

Exception in thread main java.lang.ArithmeticException:not valid

Exception Handling in Java

Java 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.

Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

Exception Handling in Java

Java Exception propagation :

```
class TestExceptionPropagation1{
  void m(){
    int data=50/0;
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e)
    {System.out.println("exception handled");}
  }
```

Exception Handling in Java

Java Exception propagation :

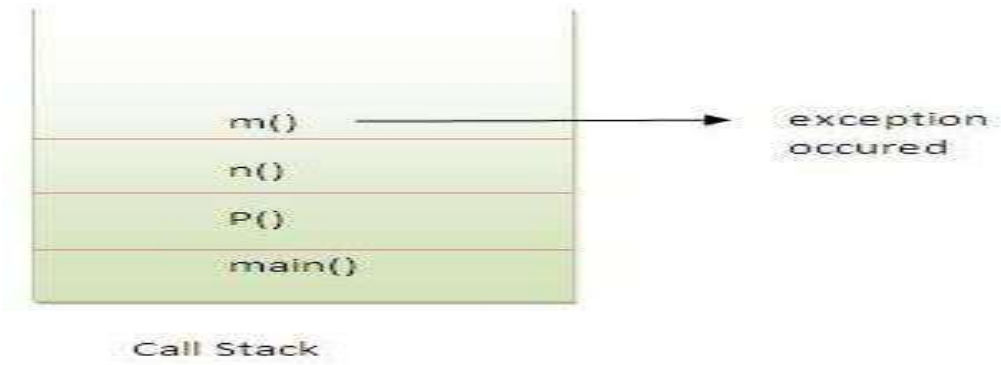
```
public static void main(String args[]){
  TestExceptionPropagation1 obj=new
TestExceptionPropagation1();
  obj.p();
  System.out.println("normal flow...");
 }
}
```

Output:

```
exception handled
normal flow...
```

Exception Handling in Java

Java Exception propagation :



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

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

Exception Handling in Java

Java Exception propagation :

Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

```
class TestExceptionPropagation2{
  void m(){
    throw new java.io.IOException("device error");//checked exception
  }
  void n(){
    m();
  }
```

Exception Handling in Java

Java Exception propagation :

```
 void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception
handeled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation2 obj=new
TestExceptionPropagation2();
   obj.p();
   System.out.println("normal flow"); }}
```

Output:Compile Time Error

Exception Handling in Java

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.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Exception Handling in Java

Syntax of java throws :

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

Which exception should be declared ?

Ans) checked exception only, because:

- **unchecked Exception: under your control so correct your code.**
- **error: beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.**

Exception Handling in Java

Advantage of Java throws keyword :

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

Exception Handling in Java

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");}}
```

Exception Handling in Java

Java throws example :

```
public static void main(String args[]){
  Testthrows1 obj=new Testthrows1();
  obj.p();
  System.out.println("normal flow...");
 }
}
```

Output:

```
exception handled
normal flow...
```

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

Exception Handling in Java

Java throws example :

There are two cases:

Case1:You caught the exception i.e. handle the exception using try/catch.

Case2:You declare the exception i.e. specifying throws with the method.

Exception Handling in Java

Case1: You handle the exception

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
public class Testthrows2{
   public static void main(String args[]){
```

Exception Handling in Java

Case1: You handle the exception

```
try{
    M m=new M();
    m.method();
   }catch(Exception e){System.out.println("exception
handled");}

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

Output:exception handled
normal flow...

Exception Handling in Java

Case2: You declare the exception

- **In case you declare the exception, if exception does not occur, the code will be executed fine.**

- **In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.**

Exception Handling in Java

Program if exception does not occur

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.

- B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

Exception Handling in Java

A)Program if exception does not occur :

```
import java.io.*;
class M{
 void method()throws IOException{
  System.out.println("device operation performed");
 }
}
class Testthrows3{
public static void main(String args[])throws IOException{
    //declare exception
    M m=new M();
    m.method();
   System.out.println("normal flow...");
  } }
```

Exception Handling in Java

A)Program if exception does not occur :

Output:

```
device operation performed
normal flow...
```

Exception Handling in Java

B)Program if exception occurs :

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])throws IOException{
     //declare exception
     M m=new M();
     m.method();
    System.out.println("normal flow...");
  }  }
```

Exception Handling in Java

B)Program if exception occurs :

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }}
class Testthrows4{
   public static void main(String args[])throws IOException{
     //declare exception
     M m=new M();
     m.method();
    System.out.println("normal flow...");
  }  }
```

Output:Runtime Exception

Exception Handling in Java

Can we rethrow an exception ?

Yes, by throwing same exception in catch block.

Exception Handling in Java

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)	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

Exception Handling in Java

Difference between final, finally and finalize

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method

Exception Handling in Java

Java finalize example :

```
class FinalizeExample{
public void finalize(){System.out.println("finalize called");}
public static void main(String[] args){
FinalizeExample f1=new FinalizeExample();
FinalizeExample f2=new FinalizeExample();
f1=null;
f2=null;
System.gc();
}}
```

Exception Handling in Java

ExceptionHandling with MethodOverriding in Java

There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:

- **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, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.**

Exception Handling in Java

1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

```
import java.io.*;
class Parent{
  void msg(){System.out.println("parent");}
}

class TestExceptionChild extends Parent{
  void msg()throws IOException{
    System.out.println("TestExceptionChild");
  }
```

Exception Handling in Java

1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

```
public static void main(String args[]){
   Parent p=new TestExceptionChild();
   p.msg();
  }
}
```

Output:Compile Time Error

Exception Handling in Java

2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

```
import java.io.*;
class Parent{
  void msg(){System.out.println("parent");}
}

class TestExceptionChild1 extends Parent{
  void msg()throws ArithmeticException{
    System.out.println("child");
  }
```

Exception Handling in Java

2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

```
 public static void main(String args[]){
   Parent p=new TestExceptionChild1();
   p.msg();
  }
}
```

Output:child

Exception Handling in Java

3) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

```
//Example in case subclass overridden method declares parent exception
import java.io.*;
class Parent{
  void msg()throws ArithmeticException{System.out.println("parent");}
}
class TestExceptionChild2 extends Parent{
  void msg()throws Exception{System.out.println("child");}
```

Exception Handling in Java

3) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

```
//Example in case subclass overridden method declares parent exception
   public static void main(String args[]){
     Parent p=new TestExceptionChild2();
     try{
     p.msg();
     }catch(Exception e){}
    }  }
```

Output:Compile Time Error

Exception Handling in Java

3) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

```
//Example in case subclass overridden method declares same exception
import java.io.*;
class Parent{
  void msg()throws Exception{System.out.println("parent");}
}
class TestExceptionChild3 extends Parent{
  void msg()throws Exception{System.out.println("child");}
```

Exception Handling in Java

3) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

```
//Example in case subclass overridden method declares same exception
    public static void main(String args[]){
    Parent p=new TestExceptionChild3();
    try{
    p.msg();
    }catch(Exception e){}
   }  }
```

Output:child

Exception Handling in Java

3) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

```
//Example in case subclass overridden method declares subclass exception
import java.io.*;
class Parent{
  void msg()throws Exception{System.out.println("parent");} }
class TestExceptionChild4 extends Parent{
void msg()throws
ArithmeticException{System.out.println("child");}
```

Exception Handling in Java

3) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

```
//Example in case subclass overridden method declares subclass exception
   public static void main(String args[]){
     Parent p=new TestExceptionChild4();
     try{
     p.msg();
     }catch(Exception e){}
    }  }
```

Output:child

Exception Handling in Java

3) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

```
//Example in case subclass overridden method declares no exception
import java.io.*;
class Parent{
  void msg()throws Exception{System.out.println("parent");}
}
class TestExceptionChild5 extends Parent{
  void msg(){System.out.println("child");}
```

Exception Handling in Java

3) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

```
//Example in case subclass overridden method declares no exception
   public static void main(String args[]){
     Parent p=new TestExceptionChild5();
     try{
     p.msg();
     }catch(Exception e){}
    }  }
```

Output:child

Java Custom 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.

By the help of custom exception, you can have your own exception and message.

Exception Handling in Java

Java Custom Exception 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");
  }
```

Exception Handling in Java

Java Custom Exception Example

```
    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...