

Exception Handling

Exception: An unwanted unexpected event that disturbs normal flow of the program is called exception.

Example:

- SleepingException
- TyrePuncturedException
- FileNotFoundException ...etc.

It is highly recommended to handle exceptions. The main objective of exception handling is graceful (normal) termination of the program.

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.

Dictionary Meaning: Exception is an abnormal condition.

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 the meaning of exception handling?

Exception handling doesn't mean repairing an exception. We have to define alternative way to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

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.

Ex:-

statement 1;

statement 2;

statement 3;

statement 4;

statement 5;//exception occurs

statement 6;

statement 7;

statement 8;

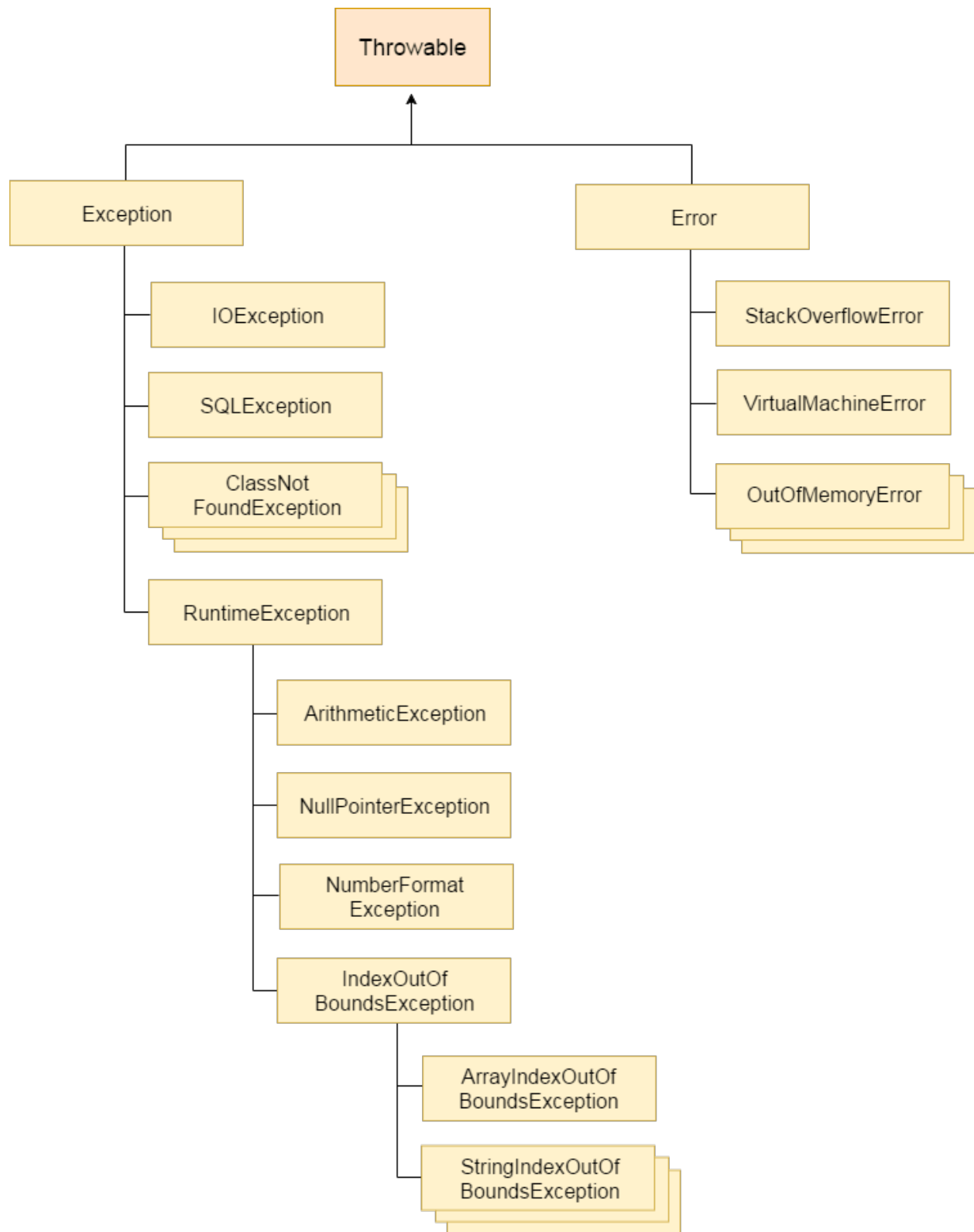
statement 9;

statement 10;

Suppose there are 10 statements in your program and there occurs an exception at statement 5, 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.

Hierarchy of Java Exception classes:-

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



For every thread JVM will create a separate stack at the time of Thread creation. All method calls performed by that thread will be stored in that stack. Each entry in the stack is called "Activation record" (or) "stack frame".

After completing every method call JVM removes the corresponding entry from the stack.

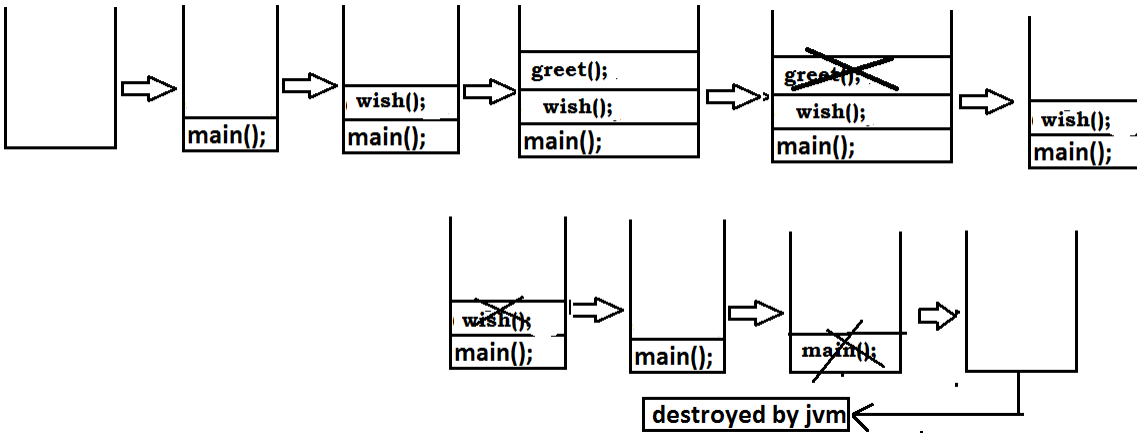
After completing all method calls JVM destroys the empty stack and terminates the program normally.

Example:

```
class Test{  
  
    public static void main(String[] args){  
  
        Wish();  
  
    }  
  
    public static void wish(){  
  
        greet();  
  
    }  
  
    public static void greet(){  
  
        System.out.println("Hello");  
  
    }  
  
}
```

Output:

Hello



Default Exception Handling in Java:

- If an exception raised inside any method then that method is responsible to create Exception object with the following information.
 1. Name of the exception.
 2. Description of the exception.
 3. Location of the exception.(StackTrace)
- After creating that Exception object, the method handovers that object to the JVM.
- JVM checks whether the method contains any exception handling code or not. If method won't contain any handling code then JVM terminates that method abnormally and removes corresponding entry form the stack.
- JVM identifies the caller method and checks whether the caller method contain any handling code or not. If the caller method also does not contain handling code then JVM terminates that caller method also abnormally and removes corresponding entry from the stack.

- This process will be continued until main() method and if the main() method also doesn't contain any exception handling code then JVM terminates main() method also and removes corresponding entry from the stack.
- Then JVM handovers the responsibility of exception handling to the ***default exception handler***.
- Default exception handler just print exception information to the console in the following format and terminates the program abnormally.

Exception in thread “xxx(main)” Name of exception: description

Location of exception (stack trace)

Example:

```
class Test{
public static void main(String[] args){
wish();
}
public static void wish(){
greet();
}
public static void greet(){
System.out.println(10/0);
}
}
```

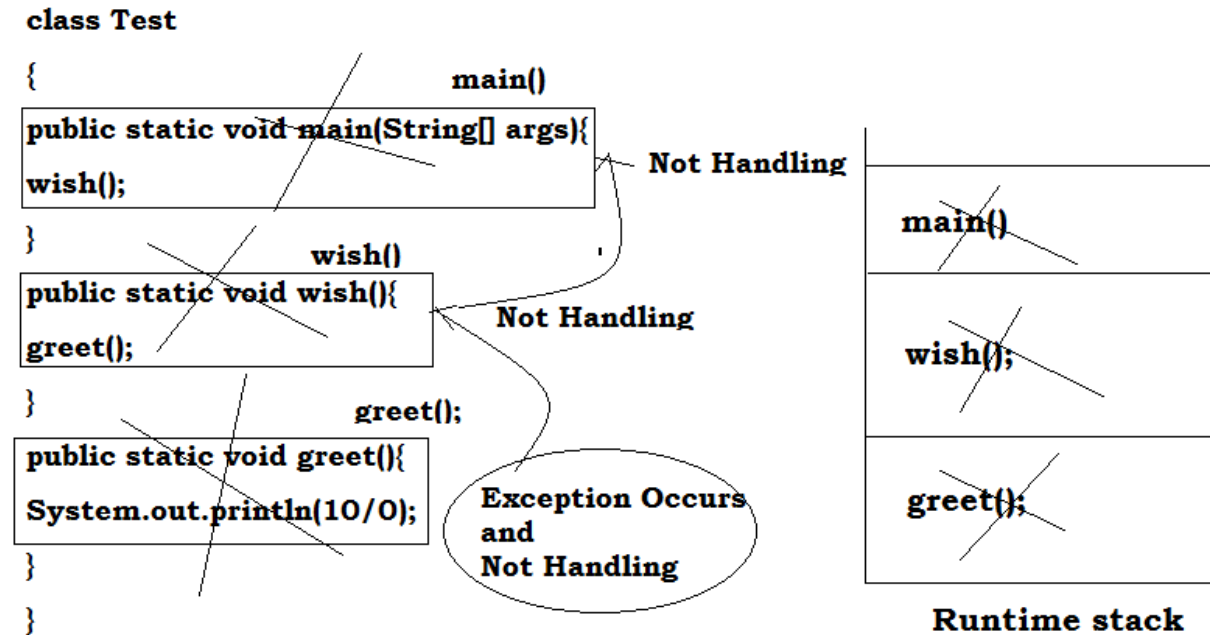
Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
atTest.greet(Test.java:10)
```

atTest.wish(Test.java:7)

atTest.main(Test.java:4)

Daigram:



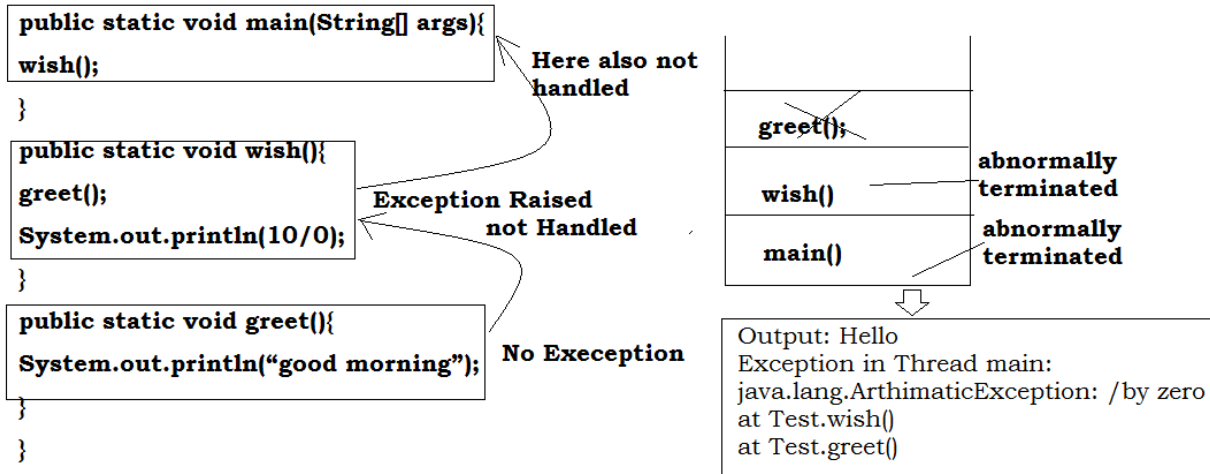
Example:

```
class Test {  
    public static void main(String[] args) {  
        wish();  
    }  
    public static void wish() {  
        greet();  
        System.out.println(10/0);  
    }  
    public static void greet() {  
        System.out.println("good morning");  
    }  
}
```



```
}
```

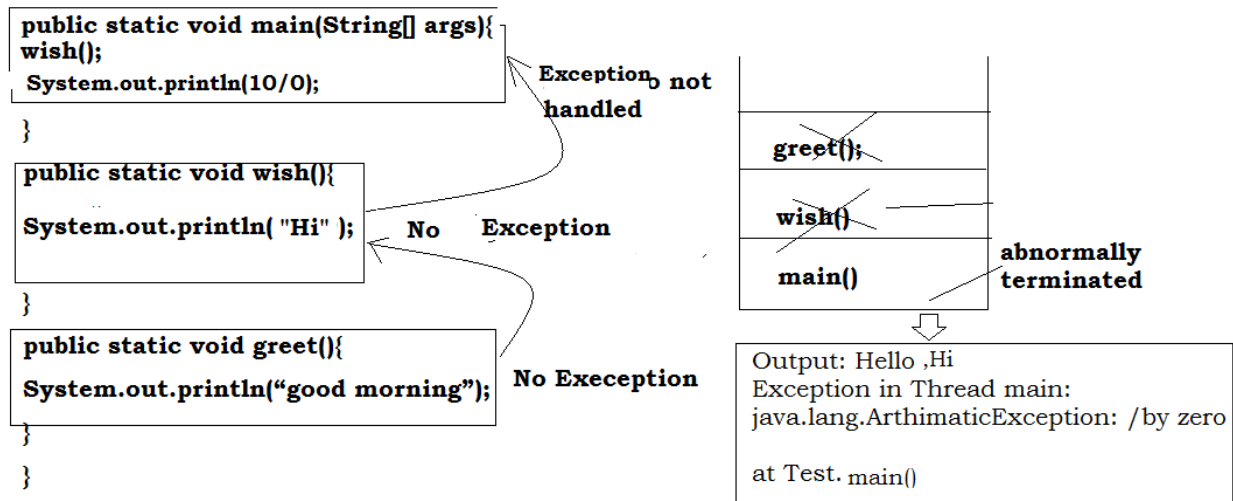
```
}
```



Example:

```
class Test{  
    public static void main(String[] args){  
        wish();  
        System.out.println(10/0);  
    }  
}
```

```
    public static void wish(){  
        System.out.println("Hi");  
    }  
    public static void greet(){  
        System.out.println("good morning");  
    }  
}
```

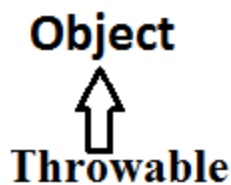


Note: In our program, if at least one method terminated abnormally then that program termination is abnormal termination.

If all methods terminated normally then only the program termination is normal termination.

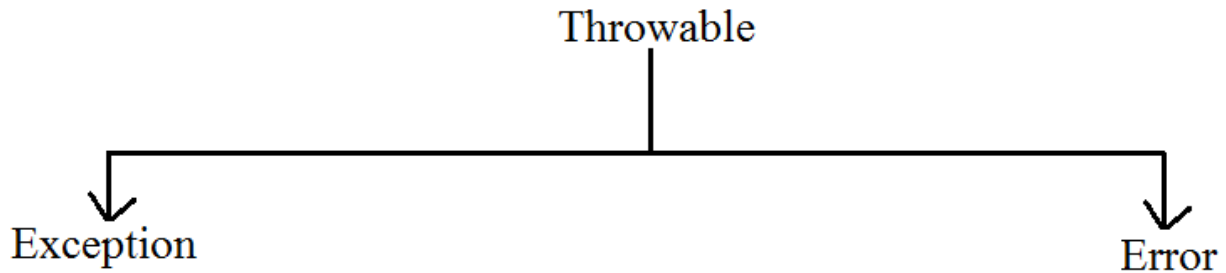
Hierarchy of Java Exception classes:

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



`Throwable` acts as a root for exception hierarchy.

`Throwable` class contains the following two child classes.



Exception:

Most of the cases exceptions are caused by our program and these are recoverable.

Ex: If `FileNotFoundException` occurs then we can use local file and we can continue rest of the program execution normally.

Error:

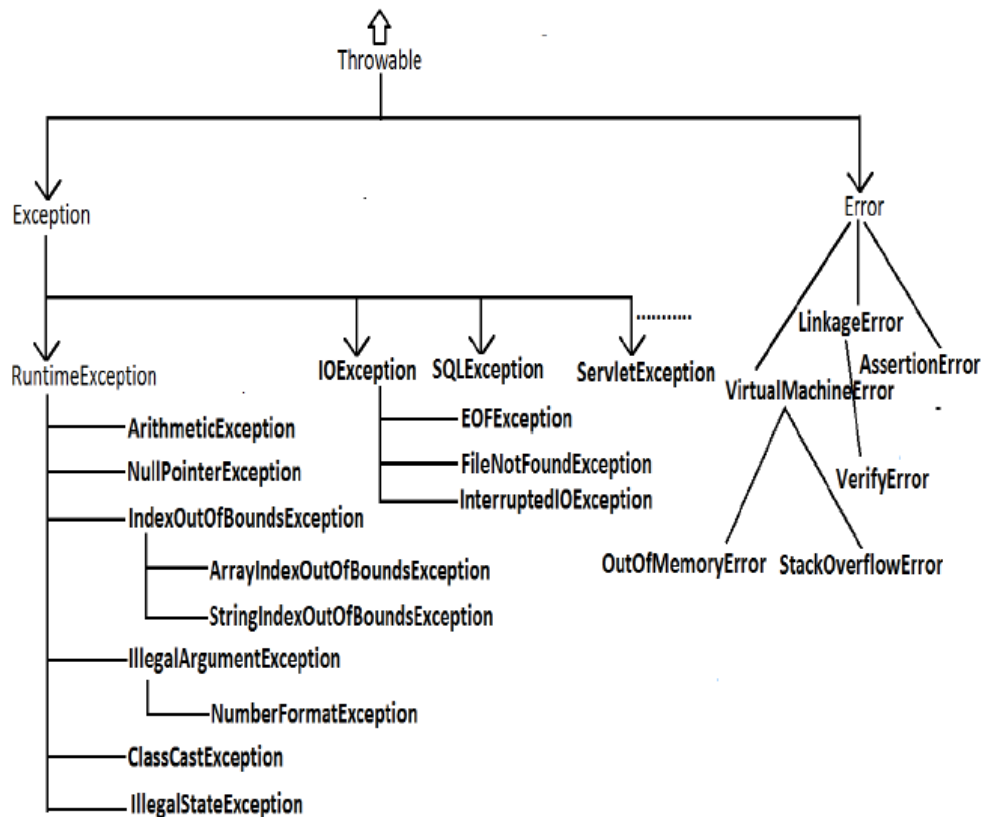
Most of the cases errors are not caused by our program these are due to lack of system resources and these are non-recoverable.

Ex: If `OutOfMemoryError` occurs being a programmer we can't do anything the program will be terminated abnormally. System Admin or Server Admin is responsible to raise/increase heap memory.

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:

1. Checked Exception
2. Unchecked Exception
3. Error



Checked Vs Unchecked Exceptions:

Checked exceptions: The exceptions which are checked by the compiler whether programmer handling or not, for smooth execution of the program at runtime, are called checked exceptions.

1. HallTicketMissingException
2. PenNotWorkingException
3. FileNotFoundException

Compiler will check whether we are handling checked exception or not.

If we are not handling then we will get compile time error.

Ex:

```
Import java.lang.*;  
class Test{  
    PrintWriter pw=new PrintWriter(abc.text);  
    pw.println("hello");  
}
```

Compile time: Unreported Exception java.io.FileNotFoundException; must be caught or declared to thrown.

Note: In our program if there is chance of raising Checked Exception then compulsory we should handle that checked exception either by using try-catch or throws keyword. Otherwise we will get compile time error.

Unchecked exceptions: The exceptions which are not checked by the compiler whether programmer handling or not, are called unchecked exceptions.

1. BombBlastException
2. ArithmeticException
3. NullPointerException

Note: RuntimeException and its child classes, Error and its child classes are unchecked and all the remaining are considered as checked exceptions.

Note: Whether exception is checked or unchecked compulsory it should occurs at runtime only and there is no chance of occurring any exception at compile time.

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

Ex:-. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions

Ex: - ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

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

Fully checked Vs Partially checked:

Fully checked: A checked exception is said to be fully checked if and only if all its child classes are also checked.

Example:

1) IOException

2) InterruptedException

Partially checked: A checked exception is said to be partially checked if and only if some of its child classes are unchecked.

Example:

Exception

Note: The only possible partially checked exceptions in java are:

1. Throwable.
2. Exception.

Q: Describe behavior of following exceptions?

1. RuntimeException-----unchecked
2. Error-----unchecked
3. IOException-----fully checked
4. Exception-----partially checked
5. InterruptedException-----fully checked
6. Throwable-----partially checked
7. ArithmeticException ----- unchecked
8. NullPointerException ----- unchecked
9. FileNotFoundException ----- fully checked

Java Exception Keywords:

There are 5 keywords which are used in handling exceptions in Java.

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.

Customized Exception Handling by using try-catch

- It is highly recommended to handle exceptions.
- In our program the code which may raise exception is called risky code; we have to place risky code inside try block and the corresponding handling code inside catch block.

Example:

```
try
{
    Risky code
}
catch(Exception e)
{
    Handling code
}
```

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

Syntax of Java try-catch

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

Syntax of try-finally block

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

Java catch block

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.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Without try catch / Problem without exception handling:

```
class Test{  
    public static void main(String[] args){  
        System.out.println("statement1");  
        System.out.println(10/0);  
        System.out.println("statement3");  
    }  
}
```

```
}
```

Output:

statement1

RE:AE:/by zero

at Test.main()

Abnormal termination.

There can be 100 lines of code after exception. So all the code after exception will not be executed.

With try catch:

```
class Test{  
    public static void main(String[] args){  
        System.out.println("statement1");  
        try{  
            System.out.println(10/0);  
        }  
        catch(ArithmeticException e){  
            System.out.println(10/2);  
        }  
        System.out.println("statement3");  
    }  
}
```

Output:

statement1

5

statement3

Normal termination.

Ex-2:

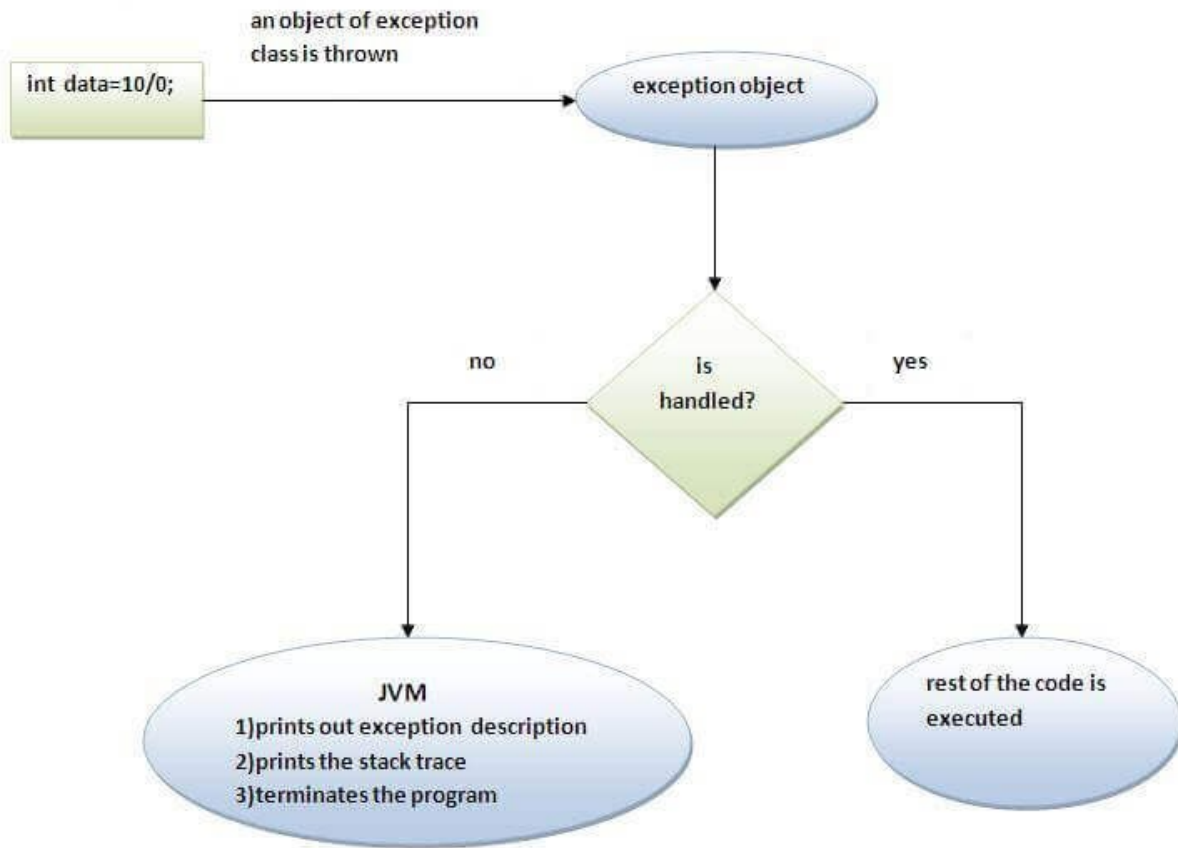
```
public class JavaExceptionExample{  
    public static void main(String args[]){  
        try{  
            //code that may raise exception  
            int data=100/0;  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.println(e);  
        }  
        //rest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

In the above example, 10/0 raises an ArithmeticException which is handled by a try-catch block.

Internal working of java try-catch block



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 exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

Note: If we kept the code in a try block that will not throw an exception.

```

public class Example3 {
    public static void main(String[] args) {
        try
        {
            int data=50/0; //may throw exception
            // if exception occurs, the remaining statement will not execute
            System.out.println("rest of the code");
        }
        // handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
    }
}

```

Output:

java.lang.ArithmeticException: / by zero

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

Note: We handle the exception using the parent class exception.

```

public class Example4 {
    public static void main(String[] args) {
        try
        {
            int data=50/0; //may throw exception
        }
    }
}

```

```
// handling the exception by using Exception class
catch(Exception e)
{
    System.out.println(e);
}
System.out.println("rest of the code");
}
}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Note: We can print a custom message on exception.

```
public class Example5 {
    public static void main(String[] args) {
        try
        {
            int data=50/0; //may throw exception
        }
        // handling the exception
        catch(Exception e)
        {
            // displaying the custom message
            System.out.println("Can't divided by zero");
        }
    }
}
```

Output:

Can't divided by zero

Note: We can resolve the exception in a catch block to get normal termination.

```
public class Example6 {  
    public static void main(String[] args) {  
        int i=50;  
        int j=0;  
        int data;  
        try  
        {  
            data=i/j; //may throw exception  
        }  
        // handling the exception  
        catch(Exception e)  
        {  
            // resolving the exception in catch block  
            System.out.println(i/(j+2));  
        }  
    }  
}
```

Output:

25

Note: We kept risky code along with try block, we also enclose exception code in a catch block.

```
public class Example7 {  
    public static void main(String[] args) {  
  
        try  
        {  
            int data1=50/0; //may throw exception  
  
        }  
        // handling the exception  
        catch(Exception e)  
        {  
            // generating the exception in catch block  
            int data2=50/0; //may throw exception  
        }  
        System.out.println("rest of the code");  
    }  
}
```

Output:

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

Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

Note: We handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

```

public class Example8 {
    public static void main(String[] args) {
        try
        {
            int data=50/0; //may throw exception
        }

        //try to handle the ArithmeticException using
        //ArrayIndexOutOfBoundsException
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }
}

```

Output:

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

Note: Example to handle another unchecked exception.

```

public class Example9 {
    public static void main(String[] args) {
        try
        {
            int arr[]={1,3,5,7};
            System.out.println(arr[10]); //may throw exception
        }
        // handling the array exception
    }
}

```

```

        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }
}

```

Output:

```

java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code

```

Note: an example to handle checked exception.

```

import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class Example10 {
    public static void main(String[] args) {
        PrintWriter pw;
        try {
            pw = new PrintWriter("jtp.txt"); //may throw exception
            pw.println("saved");
        }
        // providing the checked exception handler
        catch (FileNotFoundException e)
        {
            System.out.println(e);
        }
        System.out.println("File saved successfully");
    }
}

```

```
}  
}
```

Output:

File saved successfully

Control flow in try catch:

```
try{  
    statement-1;  
    statement-2;  
    statement-3;  
}  
catch(X e) {  
    statement-4;  
}  
statement5;
```

Case 1: If there is no exception.

1, 2, 3, 5 normal termination.

```
public class JavaExceptionExample{  
    public static void main(String args[])  
    {  
        System.out.println("start of the code...");  
        Try  
    {  
        System.out.println("s1...");  
        System.out.println("s2");  
        System.out.println("s3");  
    }  
    }  
}
```

```

        //code that may raise exception
        //int data=100/0;
    }
    catch(ArithmeticException e)
    {
        System.out.println("s4");
    }
    //rest code of the program
    System.out.println("s5");
}
}

```

Case 2: if an exception raised at statement 2 and corresponding catch block matched.

1, 4, 5 normal termination.

```

public class JavaExceptionExample{
    public static void main(String args[])
    {
        System.out.println("start of the code...");
        Try
    {
        System.out.println("s1...");
        System.out.println(10/0);
        System.out.println("s3");
        //code that may raise exception
        //int data=100/0;
    }
}

```

```

    }
    catch(ArithmeticException e)
    {
        System.out.println(e);
    }
    //rest code of the program
    System.out.println("s5");
    }
}

```

Case 3: if an exception raised at statement 2 but the corresponding catch block not matched.

1 followed by abnormal termination.

```

public class JavaExceptionExample{
    public static void main(String args[])
    {
        System.out.println("start of the code...");
        try{
            System.out.println("s1...");
            System.out.println(10/0);
            System.out.println("s3");
            //code that may raise exception
            //int data=100/0;
        }
        catch(NullPointerException e)
        {
            System.out.println(e);
        }
    }
}

```

```

}
//rest code of the program
System.out.println("s5");
}
}

```

Case 4: if an exception raised at statement 4 or statement 5 then it's always abnormal termination of the program.

```

public class JavaExceptionExample{
    public static void main(String args[])
    {
System.out.println("start  of the code...");
        try{
            System.out.println("s1...");
            System.out.println(10/0);
            System.out.println("s3");
            //code that may raise exception
            //int data=100/0;
        }
catch(ArithmeticException e)
{
System.out.println(e);
System.out.println(10/0);
}
//rest code of the program
System.out.println("s5");
}
}

```

}

Note:

- Within the try block if anywhere an exception raised then rest of the try block won't be executed even though we handled that exception. Hence we have to place/take only risk code inside try block and length of the try block should be as less as possible.
- If any statement which raises an exception and it is not part of any try block then it is always abnormal termination of the program.
- There may be a chance of raising an exception inside catch and finally blocks also in addition to try block.

Various methods to print exception information:

Throwable class defines the following methods to print exception information to the console.

printStackTrace():

This method prints exception information in the following format.

Name of the exception: description of exception

Stack trace

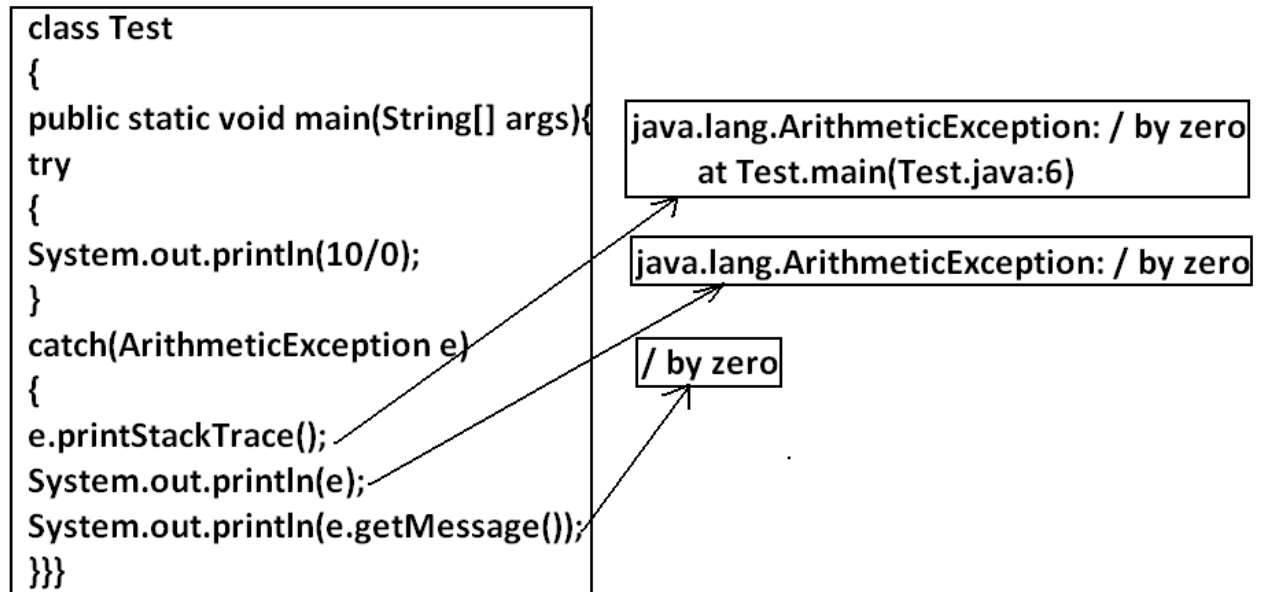
toString(): This method prints exception information in the following format.

Name of the exception: description of exception

getMessage(): This method returns only description of the exception.

Description

Example:



Note: Default exception handler internally uses `printStackTrace()` method to print exception information to the console.

Try with multiple catch blocks:

A try block can be followed by one or more catch blocks.

The way of handling an exception is varied from exception to exception. Hence for every exception type it is recommended to take a separate catch block. That is try with multiple catch blocks is possible and recommended to use.

Example:

Example:

```
try
```

```
{
```

```
}  
  
catch(Exception e)  
  
{  
  
default handler  
  
}
```

Ex:

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

Note: This approach is not recommended because for any type of Exception we are using the same catch block.

With multiple catch:

```
try
{
.
.
.
catch(FileNotFoundException e)
{
use local file
}
catch(ArithmeticException e)
{
perform these Arithmetic operations
}
catch(SQLException e)
{
don't use oracle db, use mysqldb
}
catch(Exception e)
{
```

default handler

}

Ex:

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

```
}
```

```
}
```

Note: This approach is highly recommended because for any exception raise we are defining a separate catch block.

Note: At a time only one Exception is occurred and at a time only one catch block is executed.

Rule: If try with multiple catch blocks present then order of catch blocks is very important. It should be from child to parent by mistake if we are taking from parent to child then we will get Compile time error saying

"exception xxx has already been caught"

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

Ex-Order of Parent to child:

```
class Test
```

```
{
```

```
public static void main(String[] args)
```

```
{
```

```
try
```

```
{
```

```
System.out.println(10/0);
```

```
}  
  
catch(Exception e)  
{  
    e.printStackTrace();  
}  
  
catch(ArithmeticException e)  
{  
    e.printStackTrace();  
}}}
```

CE: Exception

java.lang.ArithmeticException has already been caught

Ex- Order of Child to Parent:

```
class Test  
{  
    public static void  
    main(String[] args)  
    {  
        try  
        {  
            System.out.println(10/0);
```

```
}  
  
catch(ArithmeticException e)  
  
{  
  
e.printStackTrace();  
  
}  
  
catch(Exception e)  
  
{  
  
e.printStackTrace();  
  
}}}
```

Output:

Compile successfully.

Note: if we are trying to take multiple catch blocks same type exceptions then we will get compile time error.

Ex:

```
try  
  
{  
  
}  
  
catch(AE e)  
  
{  
  
}
```

```
catch(AE e)
```

```
{
```

```
}
```

Example 1

Let's see a simple example of java multi-catch block.

```
public class MultipleCatchBlock1 {
```

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

```
        try
```

```
{
```

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

```
    a[5]=30/0;
```

```
}
```

```
    catch(ArithmeticException e)
```

```
{
```

```
    System.out.println("Arithmetic Exception occurs");
```

```
}
```

```
    catch(ArrayIndexOutOfBoundsException e)
```

```
    {
```

```
    System.out.println("ArrayIndexOutOfBoundsException 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

Example 2

```
public class MultipleCatchBlock2 {
    public static void main(String[] args) {
        try{
            int a[]=new int[5];
            System.out.println(a[10]);
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic Exception occurs");
        }
    }
}
```

```

    }

    catch(ArrayIndexOutOfBoundsException e)

    {

        System.out.println("ArrayIndexOutOfBoundsException occurs");

    }

    catch(Exception e)

    {

        System.out.println("Parent Exception occurs");

    }

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

}

}

```

Output:

ArrayIndexOutOfBoundsException occurs

rest of the code

Example 3

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is invoked.

```

public class MultipleCatchBlock3 {

```

```
public static void main(String[] args) {  
    try{  
        int a[]=new int[5];  
        a[5]=30/0;  
        System.out.println(a[10]);  
    }  
    catch(ArithmeticException e)  
    {  
        System.out.println("Arithmetic Exception occurs");  
    }  
    catch(ArrayIndexOutOfBoundsException e)  
    {  
        System.out.println("ArrayIndexOutOfBoundsException 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

Example 4

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 MultipleCatchBlock4 {  
    public static void main(String[] args) {  
        try{  
            String s=null;  
            System.out.println(s.length());  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.println("Arithmetic Exception occurs");  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {
```

```
        System.out.println("ArrayIndexOutOfBoundsException 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

Example 5

Let's see an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

```
class MultipleCatchBlock5 {
    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...");

}

}
```

Output:

Compile-time error

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

Let's see a simple example of java nested try block.

```
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);  
            }  
  
            try{  
                int a[]=new int[5];  
                a[5]=4;  
            }  
        }  
    }  
}
```



```
catch(ArrayIndexOutOfBoundsException e){  
    System.out.println(e);  
}
```

```
    System.out.println("other statement");  
}
```

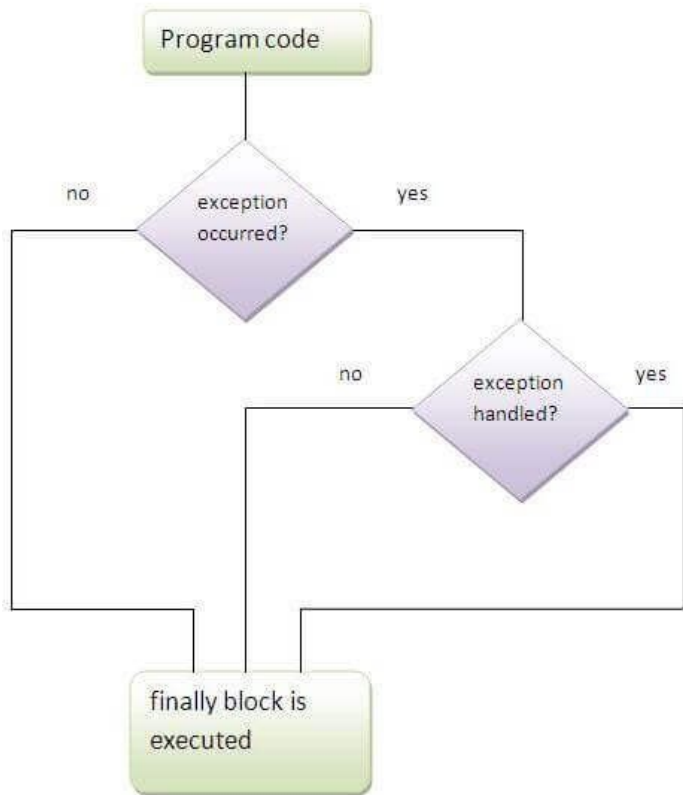
```
catch(Exception e){  
    System.out.println("handed");}  
    System.out.println("normal flow..");  
}  
}
```

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.



Why use java finally?

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

- It is not recommended to take clean up code inside try block because there is no guarantee for the execution of every statement inside a try.
- It is not recommended to place clean up code inside catch block because if there is no exception then catch block won't be executed.
- We require some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled. Such type of best place is nothing but finally block.
- Hence the main objective of finally block is to maintain cleanup code.

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

Example:

```
try
{
    risky code
}
catch(x e)
{
    handling code
}
finally
{
    cleanup code
}
```

The specialty of finally block is it will be executed always irrespective of whether the exception raised or not raised and whether handled or not handled.

Case-1: If there is no Exception:

```
class Test{

    public static void main(String[] args){

        try{

            System.out.println("try block executed");

        }

        catch(ArithmeticException e){
```

```
System.out.println("catch block executed");

}

finally{

System.out.println("finally block executed");

}

}

}
```

Output:

try block executed

Finally block executed

Ex:-

```
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...

Case-2: If an exception raised but the corresponding catch block matched:

```
class Test{  
public static void main(String[] args){  
try{  
System.out.println("try block executed");  
System.out.println(10/0);  
}  
catch(ArithmeticException e){  
System.out.println("catch block executed");  
}  
finally{  
System.out.println("finally block executed");  
}  
}  
}
```

Output:

Try block executed

Catch block executed

Finally block executed

3: If an exception raised but the corresponding catch block not matched:

```
class Test{  
    public static void main(String[] args){  
        try{  
            System.out.println("try block executed");  
            System.out.println(10/0);  
        }  
        catch(NullPointerException e){  
            System.out.println("catch block executed");  
        }  
        Finally{  
            System.out.println("finally block executed");  
        }  
    }  
}
```

Output:

Try block executed

Finally block executed

Exception in thread "main" java.lang.ArithmeticException: / by zero
at Test.main(Test.java:8)

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).

return Vs finally:

Even though return statement present in try or catch blocks first finally will be executed and after that only return statement will be considered. i.e. finally block dominates return statement.

Example:

```
class Test{
public static void main(String[] args){
try{
System.out.println("try block executed");
return;
}
catch(ArithmeticException e){
System.out.println("catch block executed");
}
finally{
System.out.println("finally block executed");
}
}}
```

Output:

try block executed

Finally block executed

Note:- If return statement present try, catch and finally blocks then finally block return statement will be considered.

Example:

```
class Test{
public static void main(String[] args){
System.out.println(m1());
}
public static int m1(){
try{
System.out.println(10/0);
return 777;
}
catch(ArithmeticException e){
return 888;
}
finally{
return 999;
}
}}
```

Output:

999

finally vs System.exit(0):

=====

There is only one situation where the finally block won't be executed is whenever we are using System.exit(0) method.

Whenever we are using System.exit(0) then JVM itself will be shutdown , in this case finally block won't be executed.

i.e., System.exit(0) dominates finally block.

Example:

```
class Test{  
    public static void main(String[] args){  
        try{  
            System.out.println("try");  
            System.exit(0);  
        }  
        catch(ArithmeticException e){  
            System.out.println("catch block executed");  
        }  
        finally{  
            System.out.println("finally block executed");  
        }  
    }  
}
```

Output:

Try

Note : System.exit(0);

- This argument acts as status code. Instead of zero, we can take any integer value
- Zero means normal termination , non-zero means abnormal termination
- This status code internally used by JVM, whether it is zero or non-zero there is no change in the result and effect is same wrt program.

Difference between final, finally, and finalize:

final:

- final is the modifier applicable for classes, methods and variables.
- If a class declared as the final then child class creation is not possible.
- If a method declared as the final then overriding of that method is not possible.
- If a variable declared as the final then reassignment is not possible.

finally:

finally is the block always associated with try-catch to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled.

finalize:

finalize is a method, always invoked by Garbage Collector just before destroying an object to perform cleanup activities.

Note:

- finally block meant for cleanup activities related to try block where as finalize() method meant for cleanup activities related to object.
- To maintain clean up code finally block is recommended over finalize() method because we can't expect exact behavior of GC.

Java final example

```
class FinalExample{  
  
    public static void main(String[] args){  
  
        final int x=100;  
  
        x=200;//Compile Time Error  
  
    }  
}
```

Java finally example

```
class FinallyExample{  
  
    public static void main(String[] args){  
  
        try{  
  
            int x=300;  
  
        }catch(Exception e){System.out.println(e);}  
  
        finally{System.out.println("finally block is executed");}  
  
    }  
}
```

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();  
  
}}
```

Control flow in try catch finally:

Ex:

```
{  
  try{  
    Stmt 1;  
    Stmt-2;  
    Stmt-3;  
  }  
  catch(Exception e){  
    Stmt-4;  
  }  
  finally{  
    stmt-5;  
  }  
  Stmt-6;  
}
```

Case 1: If there is no exception. 1, 2, 3, 5, 6 normal termination.

Case 2: if an exception raised at statement 2 and corresponding catch block matched. 1,4,5,6 normal terminations.

Case 3: if an exception raised at statement 2 and corresponding catch block is not matched. 1, 5 abnormal termination.

Case 4: if an exception raised at statement 4 then it's always abnormal termination but before the finally block will be executed.

Case 5: if an exception raised at statement 5 or statement 6 its always abnormal termination.

Control flow in Nested try-catch-finally:

```
try{  
    stmt-1;  
    stmt-2;  
    stmt-3;  
    try{  
        stmt-4;  
        stmt-5;  
        stmt-6;  
    }  
    catch (X e){  
        stmt-7;  
    }  
    finally{  
        stmt-8;  
    }  
    stmt-9;  
}  
catch (Y e){
```

```
stmt-10;  
}  
finally{  
  stmt-11;  
}  
stmt-12;
```

Case 1: if there is no exception. 1, 2, 3, 4, 5, 6, 8, 9, 11, 12 normal termination.

Case 2: if an exception raised at statement 2 and corresponding catch block matched 1,10,11,12 normal terminations.

Case 3: if an exception raised at statement 2 and corresponding catch block is not matched 1, 11 abnormal termination.

Case 4: if an exception raised at statement 5 and corresponding inner catch has matched 1, 2, 3, 4, 7, 8, 9, 11, 12 normal termination.

Case 5: if an exception raised at statement 5 and inner catch has not matched but outer catch block has matched. 1, 2, 3, 4, 8, 10, 11, 12 normal termination.

Case 6: if an exception raised at statement 5 and both inner and outer catch blocks are not matched. 1, 2, 3, 4, 8, 11 abnormal termination.

Case 7: if an exception raised at statement 7 and the corresponding catch block matched 1, 2, 3, 4, 5, 6, 8, 10, 11, 12 normal termination.

Case 8: if an exception raised at statement 7 and the corresponding catch block not matched 1, 2, 3, 4, 5, 6, 8, 11 abnormal terminations.

Case 9: if an exception raised at statement 8 and the corresponding catch block has matched 1, 2, 3, 4, 5, 6, 7, 10, 11,12 normal termination.

Case 10: if an exception raised at statement 8 and the corresponding catch block not matched 1, 2, 3, 4, 5, 6, 7, 11 abnormal terminations.

Case 11: if an exception raised at statement 9 and corresponding catch block matched 1, 2, 3, 4, 5, 6, 7, 8,10,11,12 normal termination.

Case 12: if an exception raised at statement 9 and corresponding catch block not matched 1, 2, 3, 4, 5, 6, 7, 8, 11 abnormal termination.

Case 13: if an exception raised at statement 10 is always abnormal termination but before that finally block 11 will be executed.

Case 14: if an exception raised at statement 11 or 12 is always abnormal termination.

Note:

- If we are not entering into the try block then the finally block won't be executed. Once we entered into the try block without executing finally block we can't come out.
- We can take try-catch inside try i.e., nested try-catch is possible
- The most specific exceptions can be handled by using inner try-catch and generalized exceptions can be handle by using outer try-catch.

Example:

```
class Test{  
  
public static void main(String[] args){  
  
try{  
  
System.out.println(10/0);
```

```
}  
  
catch(ArithmeticException e){  
  
System.out.println(10/0);  
  
}  
  
finally{  
  
String s=null;  
  
System.out.println(s.length());  
  
}  
  
}}
```

Output:

RE:NullPointerException

Note: Default exception handler can handle only one exception at a time and that is the most recently raised exception.

Various possible combinations of try catch finally:

- Whenever we are writing try block compulsory we should write either catch or finally. i.e., try without catch or finally is invalid.
- Whenever we are writing catch block compulsory we should write try. i.e., catch without try is invalid.
- Whenever we are writing finally block compulsory we should write try. i.e., finally without try is invalid.
- In try-catch-finally order is important.

- Within the try-catch -finally blocks we can take try-catch-finally. i.e., nesting of try-catch-finally is possible.
- For try-catch-finally blocks curly braces are mandatory.

```
try {}
catch (X e) {}
```

✓

```
try {}
catch (X e) {}
catch (Y e) {}
```

✓

```
try {}
catch (X e) {}
catch (X e) {} //CE:exception ArithmeticException has already been caught
```

X

```
try {}
catch (X e) {}
finally {}
```

✓

```
try {}
finally {}
```

✓

```
try {} //CE: 'try' without 'catch', 'finally' or resource declarations
```

X

```
catch (X e) {} //CE: 'catch' without 'try'
```

X

```
finally {} //CE: 'finally' without 'try'
```

X

```
try {} //CE: 'try' without 'catch', 'finally' or resource declarations  
System.out.println("Hello");  
catch {} //CE: 'catch' without 'try'
```

X

```
try {}  
catch (X e) {}  
System.out.println("Hello");  
catch (Y e) {} //CE: 'catch' without 'try'
```

```
try {}  
catch (X e) {}  
System.out.println("Hello");  
finally {} //CE: 'finally' without 'try'
```

X

```
try {}  
finally {}  
catch (X e) {} //CE: 'catch' without 'try'
```

X

```
try {}  
catch (X e) {}  
try {}  
finally {}
```

✓

```
try {}  
catch (X e) {}  
finally {}  
finally {} //CE: 'finally' without 'try'
```

X

```
try {}  
catch (X e) {  
  try {}  
  catch (Y e1) {}  
}
```

✓

```
try {}  
catch (X e) {}  
finally {  
  try {}  
  catch (Y e1) {}  
  finally {}  
}
```

✓

```
try {  
try {} //CE: 'try' without 'catch', 'finally' or resource declarations  
}  
catch (X e) {}
```

X

```
try //CE: '{' expected  
System.out.println("Hello");  
catch (X e1) {} //CE: 'catch' without 'try'
```

X

```
try {}  
catch (X e) //CE: '{' expected  
System.out.println("Hello");
```

X

```
try {}  
catch (NullPointerException e1) {}  
finally //CE: '{' expected  
System.out.println("Hello");
```

X

Java throw keyword / throw statement:-

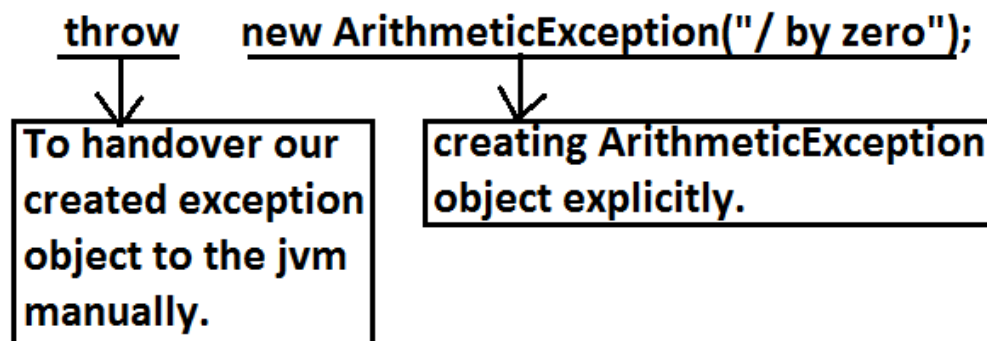
The Java throw keyword is used to explicitly throw an exception. It means sometimes we can create Exception object explicitly and we can hand over to the JVM manually by using throw keyword.

We can throw either checked or unchecked exception in Java by throw keyword. The throw keyword is mainly used to throw custom exception.

Syntax:

```
throw exception;
```

Ex-1:-



Ex:-

```
throw new IOException("sorry device error);
```

Ex: - without throw:-

```
class Test{  
    public static void main(String[] args){  
        System.out.println(10/0);  
    }  
}
```

```
}
```

In this case creation of `ArithmeticException` object and handover to the jvm will be performed automatically by the `main()` method.

Ex- With throw:-

```
class Test{  
    public static void main(String[]args){  
        throw new ArithmeticException("/by zero");  
    }  
}
```

In this case we are creating exception object explicitly and handover to the JVM manually.

Note: In general we can use `throw` keyword for customized exceptions but not for predefined exceptions.

java throw keyword example

We have created the `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 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

Case 1:



throw e;

If e refers null then we will get NullPointerException.

Ex:

```
class Test3 {  
    static ArithmeticException e = new ArithmeticException();  
    public static void main(String[] args) {  
        throw e;  
    }  
}
```

Output:

Runtime exception: Exception in thread "main" java.lang.ArithmeticException

Ex:

```
class Test3{  
    static ArithmeticException e;// not referring any object  
    public static void main(String[] args){  
        throw e;  
    }  
}
```

Output:

Exception in thread "main" java.lang.NullPointerException at
Test3.main(Test3.java:5)

Case 2:

After throw statement we can't take any statement directly otherwise we will get compile time error saying unreachable statement.

Ex:

```
class Test3{  
    public static void main(String[] args){  
        System.out.println(10/0);  
        System.out.println("hello");  
    }  
}
```

Output:

Runtime error: Exception in thread "main" java.lang.ArithmeticException: / by
zero at Test3.main(Test3.java:4)

Ex:

```
class Test3 {  
    public static void main(String[] args){  
        throw new ArithmeticException("/ by zero");  
        System.out.println("hello");  
    }  
}
```

Output:

Compile time error.

Test3.java:5: unreachable statement System.out.println("hello");

Case 3:

We can use throw keyword only for Throwable types otherwise we will get compile time error saying incomputable types.

Ex:

```
class Test3 {  
    public static void main(String[] args){  
        throw new Test3();  
    }  
}
```

Output:

Compile time error.

Test3.java:4: incompatible types found : Test3 required: java.lang.Throwable
throw new Test3();

Ex:

```
class Test3 extends RuntimeException{  
    public static void main(String[] args){  
        throw new Test3();  
    }  
}
```

Output:

Runtime error: Exception in thread "main" Test3 at Test3.main(Test3.java:4)

Throws statement:-

In our program if there is any chance of raising checked exception then compulsory we should handle either by try catch or by throws keyword otherwise the code won't compile.

Example:

```
import java.io.*;  
  
class Test3{  
  
    public static void main(String[] args){
```

```
PrintWriter out=new PrintWriter("abc.txt");

out.println("hello");

}

}
```

CE:

Unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown.

Example:

```
class Test3 {

public static void main(String[] args){

Thread.sleep(5000);

}

}
```

Unreported exception java.lang.InterruptedException; must be caught or declared to be thrown.

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.

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.

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.

We can handle this compile time error by using the following 2 ways.

1. By using try catch:

```
class Test3 {  
    public static void  
    main(String[] args) {  
        try {  
            Thread.sleep(5000);  
        }  
    }  
}
```

```
}  
catch(InterruptedException e){  
}  
}  
}
```

Output:

Compile and running

Successfully

By using throws keyword:

We can use throws keyword to delegate the responsibility of exception handling to the caller method. Then caller method is responsible to handle that exception.

Ex-2

```
class Test3 {  
  
    public static void main(String[] args) throws InterruptedException {  
  
        Thread.sleep(5000);  
  
    }  
  
}
```

Output:

Compile and running successfully

Ex:

```
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...

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

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.

Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

Ex:

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

Case2: You declare the exception

- 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 occurs, an exception will be thrown at runtime because throws does not handle the exception.

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...");
    }
}
```

Output: device operation performed

normal flow...

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

Note :

- Hence the main objective of "throws" keyword is to delegate the responsibility of exception handling to the caller method.
- "throws" keyword required only checked exceptions. Usage of throws for unchecked exception there is no use.

- "throws" keyword required only to convince compiler. Usage of throws keyword doesn't prevent abnormal termination of the program.
- Hence recommended to use try-catch over throws keyword.

Example:

```
class Test{  
    public static void main(String[] args)throws InterruptedException{  
        wish();  
    }  
    public static void wish()throws InterruptedException{  
        greet();  
    }  
    public static void greet()throws InterruptedException{  
        Thread.sleep(5000);  
    }  
}
```

Output:

Compile and running successfully.

In the above program if we are removing at least one throws keyword then the program won't compile.

Case 1:

We can use throws keyword only for Throwable types otherwise we will get compile time error saying incompatible types.

```
class Test3{  
    public static void main(String[] args)
```

throws Test3

```
{  
}  
}
```

Output:

Compile time error

Test3.java:2: incompatible types

found : Test3

required: java.lang.Throwable

public static void main(String[] args) throws Test3

Ex-2:

```
class Test3 extends RuntimeException{  
    public static void main(String[] args) throws Test3 {  
    }  
}
```

Output:

Compile and running successfully.

Case 2:Example:

```
class Test3 {  
    public static void main(String[] args) {  
        throw new Exception();  
    }  
}
```

```
}
```

Output:

Compile time error.

Test3.java:3: unreported exception java.lang.Exception; must be caught or declared to be thrown

Ex:

```
class Test3 {  
    public static void main(String[] args) {  
        throw new Error();  
    }  
}
```

Output:

Runtime error

Exception in thread "main" java.lang.Error

at Test3.main(Test3.java:3)

Case 3:

In our program within the try block, if there is no chance of rising an exception then we can't write catch block for that exception otherwise we will get compile time error saying exception XXX is never thrown in body of corresponding try statement. But this rule is applicable only for fully checked exception.

Example:

<pre> class Test { public static void main(String[] args){ try{ System.out.println("hello"); } catch(Exception e) {} <u>output:</u> } hello } partial checked </pre>	<pre> class Test { public static void main(String[] args){ try{ System.out.println("hello"); } catch(ArithmeticException e) {} <u>output:</u> } hello } unchecked </pre>	<pre> class Test { public static void main(String[] args){ try{ System.out.println("hello"); } catch(java.io.IOException e) {} <u>output:</u> } compile time error } fully checked </pre>
--	--	---

<pre> class Test { public static void main(String[] args){ try{ System.out.println("hello"); } catch(InterruptedException e) {} <u>output:</u> } compile time error } Fully checked </pre>	<pre> class Test { public static void main(String[] args){ try{ System.out.println("hello"); } catch(Error e) {} <u>output:</u> } compile successfully } unchecked </pre>
--	---

Case 4:

We can use throws keyword only for constructors and methods but not for classes.

Example:

```
class Test throws Exception    //invalid  
{  
    Test() throws Exception    //valid  
    { }  
    methodOne() throws Exception    //valid  
    { }  
}
```

Exception handling keywords summary:

1. try: To maintain risky code.
2. catch: To maintain handling code.
3. finally: To maintain cleanup code.
4. throw: To handover our created exception object to the JVM manually.
5. throws: To delegate responsibility of exception handling to the caller method.

Various possible compile time errors in exception handling:

1. Exception XXX has already been caught.
2. Unreported exception XXX must be caught or declared to be thrown.
3. Exception XXX is never thrown in body of corresponding try statement.
4. Try without catch or finally.
5. Catch without try.
6. Finally without try.
7. Incompatible types.

found:Test required:java.lang.Throwable;

8. Unreachable statement

Customized Exceptions (User defined Exceptions):

Sometimes we can create our own exception to meet our programming requirements. Such type of exceptions are called customized exceptions (user defined exceptions).

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.

Example:

1. InsufficientFundsException
2. TooYoungException
3. TooOldException

Ex:-

```
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 occured: "+m);  
        }  
        System.out.println("rest of the code...");  
    }  
}
```

Output:

Exception occured:InvalidAgeException:not valid

rest of the code...

Program:

```
class TooYoungException extends RuntimeException{

    TooYoungException(String s){

        super(s);

    }

}

class TooOldException extends RuntimeException{

    TooOldException(String s){

        super(s);

    }

}

class CustomizedExceptionDemo{

    public static void main(String[] args){

        int age=Integer.parseInt(args[0]);

        if(age<25){

            throw new TooYoungException("please wait some more time.... u will get best match");

        }

        else if(age>50){
```

```
throw new TooOldException("u r age already crossed....no chance of getting  
married");  
  
}  
  
else{  
  
System.out.println("you will get match details soon by e-mail");  
  
}}}
```

Output:

```
>java CustomizedExceptionDemo 61
```

Exception in thread "main" TooYoungException:

please wait some more time.... u will get best match

at CustomizedExceptionDemo.main(CustomizedExceptionDemo.java:21)

```
>java CustomizedExceptionDemo 27
```

You will get match details soon by e-mail

```
>java CustomizedExceptionDemo 9
```

Exception in thread "main" TooOldException:

u r age already crossed....no chance of getting married

at CustomizedExceptionDemo.main(CustomizedExceptionDemo.java:25)

Note: It is highly recommended to maintain our customized exceptions as unchecked by extending RuntimeException.

We can catch any Throwable type including Errors also.

Top-10 Exceptions:

Based on the person who is raising exception, all exceptions are divided into two types.

They are:

- 1) JVM Exceptions:
- 2) Programmatic exceptions:

JVM Exceptions:

The exceptions which are raised automatically by the jvm whenever a particular event occurs, are called JVM Exceptions.

Example:

- 1) `ArrayIndexOutOfBoundsException(AIOOBE)`
- 2) `NullPointerException (NPE)`.

Programmatic Exceptions:

The exceptions which are raised explicitly by the programmer (or) by the API developer are called programmatic exceptions.

Example:

- 1) `IllegalArgumentException(IAE)`.

Top 10 Exceptions:

1. ArrayIndexOutOfBoundsException:

It is the child class of RuntimeException and hence it is unchecked. Raised automatically by the JVM whenever we are trying to access array element with out of range index.

Example:

```
class Test{  
  
    public static void main(String[] args){  
  
        int[] x=new int[10];  
  
        System.out.println(x[0]);//valid  
  
        System.out.println(x[100]);//AIOOBE  
  
        System.out.println(x[-100]);//AIOOBE  
  
    }  
  
}
```

2. NullPointerException:

It is the child class of RuntimeException and hence it is unchecked. Raised automatically by the JVM, whenever we are trying to call any method on null.

Example:

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

```
String s=null;

System.out.println(s.length()); //R.E: NullPointerException

}

}
```

3. StackOverflowError:

It is the child class of Error and hence it is unchecked. Whenever we are trying to invoke recursive method call JVM will raise StackOverFloeError automatically.

Example:

```
class Test{

public static void m1(){

m2();

}

public static void m2(){

m1();

}

public static void main(String[] args){

M1();

}

}
```

Output:

Run time error: StackOverFloeError

4. NoClassDefFoundError:

It is the child class of Error and hence it is unchecked. JVM will raise this error automatically whenever it is unable to find required .class file.

Example: java

Test If Test.class is not available. Then we will get NoClassDefFound error.

5. ClassCastException:

It is the child class of RuntimeException and hence it is unchecked. Raised automatically by the JVM whenever we are trying to type cast parent object to child type.

<pre>class Test { public static void main(String[] args) { String s=new String("bhaskar"); Object o=(Object)s; } <u>output:</u> } <u>valid</u></pre>	<pre>class Test { public static void main(String[] args) { { Object o=new Object(); String s=(String)o; } <u>output:</u> } <u>Runtime exception:ClassCastException</u></pre>	<pre>class Test { public static void main(String[] args) { { Object o=new String("bhaskar"); String s=(String)o; } <u>output:</u> } <u>valid</u></pre>
--	--	--

6. ExceptionInInitializerError:

It is the child class of Error and it is unchecked. Raised automatically by the JVM, if any exception occurs while performing static variable initialization and static block execution.

Example 1:

```
class Test{

static int i=10/0;

}
```

Output:

Runtime exception:

Exception in thread "main" java.lang.ExceptionInInitializerError

Example 2:

```
class Test{  
  
    static {  
  
        String s=null;  
  
        System.out.println(s.length());  
  
    }  
}
```

Output:

Runtime exception:

Exception in thread "main" java.lang.ExceptionInInitializerError

7. IllegalArgumentException:

It is the child class of RuntimeException and hence it is unchecked. Raised explicitly by the programmer (or) by the API developer to indicate that a method has been invoked with inappropriate argument.

Example:

```
class Test{  
  
    public static void main(String[] args){  
  
        Thread t=new Thread();  
  
        t.setPriority(10);//valid
```

```
t.setPriority(100);//invalid---Thread priority in b/w 1to 10 only  
}}
```

Output:

Runtime exception

Exception in thread "main" java.lang.IllegalArgumentException.

8. NumberFormatException:

It is the child class of IllegalArgumentException and hence is unchecked. Raised explicitly by the programmer or by the API developer to indicate that we are attempting to convert string to the number. But the string is not properly formatted.

Example:

```
class Test{  
  
public static void main(String[] args){  
  
int i=Integer.parseInt("10");  
  
int j=Integer.parseInt("ten");  
  
}}
```

Output:

Runtime Exception

Exception in thread "main" java.lang.NumberFormatException: For input string:
"ten"

9. IllegalStateException:

It is the child class of RuntimeException and hence it is unchecked. Raised explicitly by the programmer or by the API developer to indicate that a method has been invoked at inappropriate time.

Ex:

```
HttpSession session=req.getSession();  
  
System.out.println(session.getId());  
  
session.invalidate();  
  
System.out.println(session.getId()); // illgalStateException
```

10. AssertionError:

It is the child class of Error and hence it is unchecked. Raised explicitly by the programmer or by API developer to indicate that Assert statement fails.

Example:

```
assert(false);
```

Exception/Error	Raised by
1. AIOOBE 2. NPE(NullPointerException) 3. StackOverFlowError 4. NoClassDefFoundError 5. CCE(ClassCastException)	Raised automatically by JVM(JVM Exceptions)

6. ExceptionInInitializerError	
1. IAE(IllegalArgumentException) 2. NFE(NumberFormatException) 3. ISE(IllegalStateException) 4. AE(AssertionError)	Raised explicitly either by programmer or by API developer (Programatic Exceptions).

ExceptionHandling with MethodOverriding in Java

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

If the superclass method does not declare an exception

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

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

If the superclass method does not declare an exception

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");
```

```

}

public static void main(String args[]){

Parent p=new TestExceptionChild();

p.msg();

}

}

```

Output: Compile Time Error

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");

```

```

}

public static void main(String args[]){

Parent p=new TestExceptionChild1();

p.msg();

}

}

```

Output:child

If the superclass method declares an exception

1) 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");

}

}

```

```
public static void main(String args[]){  
    Parent p=new TestExceptionChild2();  
    try{  
        p.msg();  
    }  
    catch(Exception e){  
    }  
}  
}
```

Output: Compile Time Error

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");  
  
}  
  
public static void main(String args[]){  
  
Parent p=new TestExceptionChild3();  
  
try{  
  
p.msg();  
  
}  
  
catch(Exception e){  
  
}  
  
}  
  
}
```

Output: child

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");

}

public static void main(String args[]){

Parent p=new TestExceptionChild4();

try{

p.msg();

}

catch(Exception e){

}

}

}

```

Output:child

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");  
    }  
    public static void main(String args[]){  
        Parent p=new TestExceptionChild5();  
        try{  
            p.msg();  
        }  
        catch(Exception e){  
        }  
    }  
}
```

Output:child