Exception Handling

1. Introduction
2. Runtime stack mechanism
3. Default exception handling in java
4. Exception hierarchy
5. Customized exception handling by try catch
6. Control flow in try catch
7. Methods to print exception information
8. Try with multiple catch blocks
9. Finally
10. Difference between final, finally, finalize
11. Control flow in try catch finally
12. Control flow in nested try catch finally
13. Various possible combinations of try catch finally
14. throw keyword
15. throws keyword
16. Exception handling keywords summary
17. Various possible compile time errors in exception handling
18. Customized exceptions
19. Top-10 exceptions

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

Example:

SleepingException
TyrePunchuredException
FileNotFoundException…..etc

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

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".

Example: Suppose our programming requirement is to read data from London file at runtime if London file is not available our program should not be terminated abnormally. We have to provide a local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

Example:

```
try
{
read data from london file
}
catch(FileNotFoundException e)
{
use local file and continue rest of the program normally
}
}
.
.
.
```

Runtime stack mechanism: For every thread JVM will create a separate stack all method calls performed by the thread will be stored in that stack. Each entry in the stack is called "one 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.

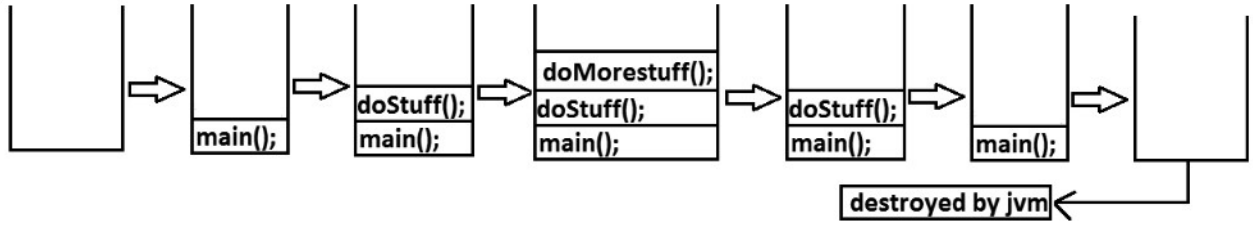
Example:

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

Output:

Hello

Diagram:



Default exception handling in java:

1) If an exception raised inside any method then the 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.
2) After creating that Exception object the method handovers that object to the JVM.
3) 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.
4) 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 also abnormally and the removes corresponding entry from the stack.
5) 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 and removes corresponding entry from the stack.
6) Then JVM handovers the responsibility of exception handling to the default exception handler.
7) Default exception handler just print exception information to the console in the following formats and terminates the program abnormally.

    Name of exception: description
    Location of exception (stack trace)

Example:

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

Output:

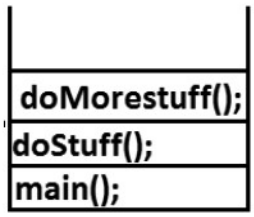
```
Runtime error
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

at Test.doMoreStuff(Test.java:10)

at Test.doStuff(Test.java:7)

at Test.main(Test.java:4)

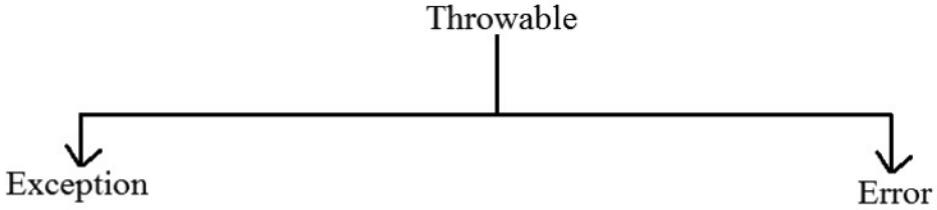
Diagram:



Exception hierarchy:

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.

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.

Checked Vs Unchecked Exceptions:

- The exceptions which are checked by the compiler for smooth execution of the program at runtime are called checked exceptions.
    1) HallTicketMissingException
    2) PenNotWorkingException
    3) FileNotFoundException
- The exceptions which are not checked by the compiler are called unchecked exceptions.
    1) BombBlaustException
    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 occur at runtime only there is no chance of occurring any exception at compile time.

Partially checked Vs 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

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

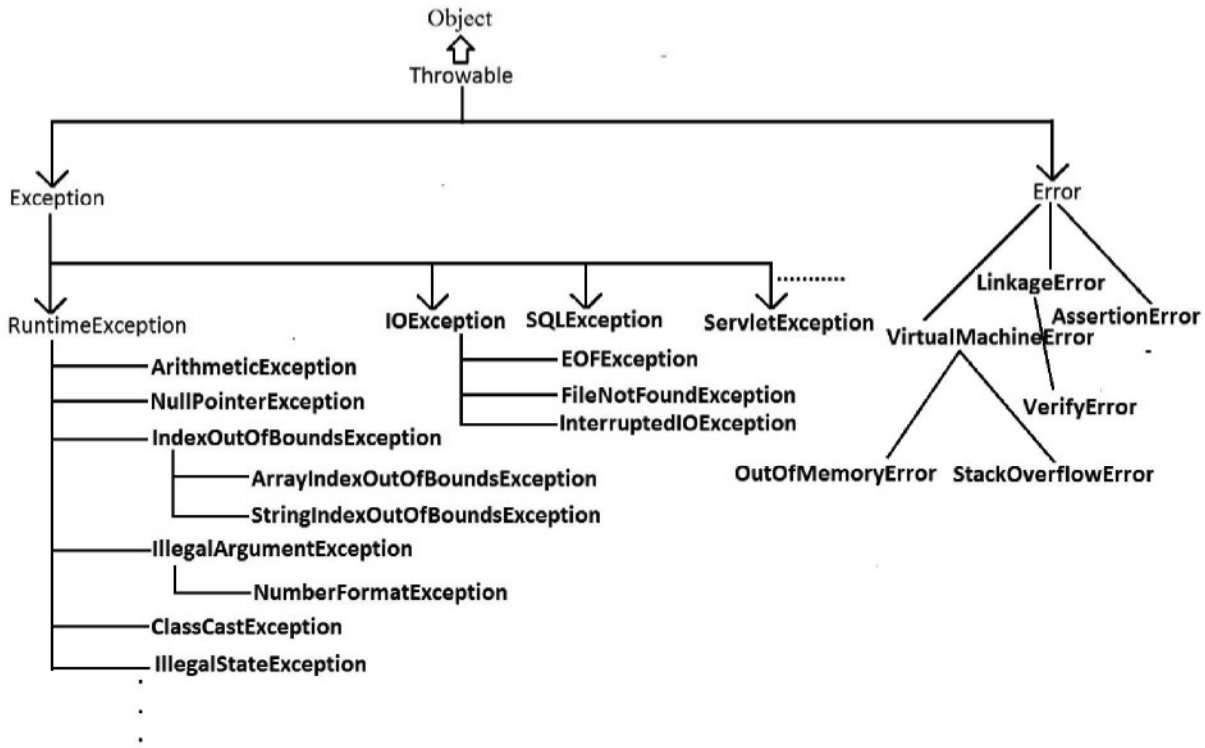
Example: Exception

- The only partially checked exceptions available in java are:
  1. Throwable.
  2. Exception.

Which of the following are checked?

1. RuntimeException-----unchecked
2. Error-----unchecked
3. IOException-----fully checked
4. Exception-----partially checked
5. InterruptedException-----fully checked
6. Throwable------partially checked

Diagram:



Customized exception handling by try catch:

- It is highly recommended to handle exceptions.
- In our program the code which may cause an 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
}
```

Without try catch	With try catch

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

Control flow in try catch:

Case 1: There is no exception.

1, 2, 3, 5 normal termination.

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

Case 3: if an exception raised at statement 2 but the corresponding catch block not matched 1 followed by abnormal termination.

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

Note:

1. 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 and **length** of the try block should be as **less** as possible.
2. 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.
3. 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:

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

```
java.lang.ArithmeticException: / by zero
        at Test.main(Test.java:6)
java.lang.ArithmeticException: / by zero
/ by zero
```

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

Try with multiple catch blocks: The way of handling an exception is varied from exception to exception hence for every exception raise a separate catch block is required that is try with multiple catch blocks is possible and recommended to use.

Example:

try { .	try { .

.
.
.
}
catch(Exception e)
{
default handler
}
 	
● This approach is not recommended because for any type of Exception we are using the same catch block.
	.
	.
	.
	catch(FileNotFoundException e)
	{
	use local file
	}
	catch(ArithmeticException e)
	{
	perform these Arithmetic operations
	}
	catch(SQLException e)
	{
	don't use oracle db, use mysql db
	}
	catch(Exception e)
	{
	default handler
	}
	
	● This approach is highly recommended because for any exception raise we are defining a separate catch block.

- If try with multiple catch blocks presents 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".

Example:

class Test
{
public static void main(String[] args)
{
try
{
System.out.println(10/0);
}
catch(Exception e)
{
e.printStackTrace();
}
catch(ArithmeticException e)
{
e.printStackTrace();
}}}
	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 time error. Test.java:13: exception java.lang.ArithmeticException has already been caught catch(ArithmeticException e)	**Output:** Compile successfully.

Finally block:

- It is never 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 never 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 place is nothing but finally block.
- Hence the main objective of finally block is to maintain cleanup code.

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.

Example 1:

```
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
```

Example 2:

```
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
```

Example 3:

```
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)
```

Return Vs Finally:

- Even though return present in try or catch blocks first finally will be executed and after that only return statement will be considered that is 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

- 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(methodOne());
}
public static int methodOne(){
try
{
System.out.println(10/0);
return 777;
}
catch(ArithmeticException e)
{
return 888;
}
finally{
return 999;
}}}
```

Output:

999

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

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

Difference between final, finally, and finalize:

Final:

- Final is the modifier applicable for class, 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:

- It 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:

- It is a method which should be called by garbage collector always just before destroying an object to perform cleanup activities.

Note:

- To maintain clean up code faunally block is recommended over finalize() method because we can't expert exact behavior of GC.

Control flow in try catch finally:

Example:

```
class Test
{
public static void main(String[] args){
try{
System.out.println("statement1");
System.out.println("statement2");
System.out.println("statement3");
}
```

```
catch(Exception e){
System.out.println("statement4");
}
finally
{
System.out.println("statement5");
}
System.out.println("statement6");
}
}
```

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:

Example:

```
class Test
{
public static void main(String[] args){
try{
System.out.println("statement1");
System.out.println("statement2");
System.out.println("statement3");
        try{
             System.out.println("statement4");
             System.out.println("statement5");
             System.out.println("statement6");
            }
            catch(ArithmeticException  e){
            System.out.println("statement7");
            }
           finally
           {
           System.out.println("statement8");
```

```
            }
            System.out.println("statement9");
    }
catch(Exception e)
{
System.out.println("statement10");
}
finally
{
System.out.println("statement11");
}
System.out.println("statement12");
}
}
```

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,.,.,., 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,.,.,.,8,11 abnormal terminations.

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

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

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

Case 12: if an exception raised at statement 9 and corresponding catch block not matched 1, 2, 3,.,.,.,., 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.

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());
}}}
```

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:

Example 1:

```
class Test1{
public static void main(String[] args){
try
{}
catch(ArithmeticException e)
{}
}}
```

Output:

Compile and running successfully.

Example 2:

```
class Test1{
public static void main(String[] args){
try
{}
```

```
catch(ArithmeticException e)
{}
catch(NullPointerException e)
{}
}
}
```

Output:

Compile and running successfully.

Example 3:

```
class Test1{
public static void main(String[] args){
try
{}
catch(ArithmeticException e)
{}
catch(ArithmeticException e)
{}
}
}
```

Output:

Compile time error.

Test1.java:7: exception java.lang.ArithmeticException has already been caught

```
catch(ArithmeticException e)
```

Example 4:

```
class Test1{
public static void main(String[] args){
try
{}
}
}
```

Output:

Compile time error

Test1.java:3: 'try' without 'catch' or 'finally'

```
try
```

Example 5:

```
class Test1{
public static void main(String[] args){
catch(Exception e)
```

```
{}
}
}
```

Output:

```
Compile time error.
Test1.java:3: 'catch' without 'try'
catch(Exception e)
```

Example 6:

```
class Test1{
public static void main(String[] args){
try
{}
System.out.println("hello");
catch(Exception e)
{}
}
}
```

Output:

```
Compile time error.
Test1.java:3: 'try' without 'catch' or 'finally'
Try
```

Example 7:

```
class Test1{
public static void main(String[] args){
try
{}
catch(Exception e)
{}
finally
{}
}
}
```

Output:

```
Compile and running successfully.
```

Example 8:

```
class Test1{
public static void main(String[] args){
try
```

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

Output:

Compile and running successfully.

Example 9:

```
class Test1{
public static void main(String[] args){
try
{}
finally
{}
finally
{}
}
}
```

Output:

Compile time error.

Test1.java:7: 'finally' without 'try'

Finally

Example 10:

```
class Test1{
public static void main(String[] args){
try
{}
catch(Exception e)
{}
System.out.println("hello");
finally
{}
}
}
```

Output:

Compile time error.

Test1.java:8: 'finally' without 'try'

Finally

Example 11:

```
class Test1{
public static void main(String[] args){
try
{}
finally
{}
catch(Exception e)
{}
}
}
```

Output:

```
Compile time error.
Test1.java:7: 'catch' without 'try'
catch(Exception e)
```

Example 12:

```
class Test1{
public static void main(String[] args){
finally
{}
}
}
```

Output:

```
Test1.java:3: 'finally' without 'try'
Finally
```

Example 13:

```
class Test1{
public static void main(String[] args){
try
{  try{}
   catch(Exception e){}
}
catch(Exception e)
{}
}
}
```

Output:

```
Compile and running successfully.
```

Example 14:

```
class Test1{
public static void main(String[] args){
try
{ }
catch(Exception e)
{
  try{}
   finally{}
}
}
}
```

Output:

Compile and running successfully.

Example 15:

```
class Test1{
public static void main(String[] args){
try
{ }
catch(Exception e)
{
  try{}
  catch(Exception e){}
}
finally{
   finally{}
    }
}
}
```

Output:

Compile time error.

```
Test1.java:11: 'finally' without 'try'
   finally{}
```

Example 16:

```
class Test1{
public static void main(String[] args){
finally{}
try{ }
```

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

Output:

```
Compile time error.
Test1.java:3: 'finally' without 'try'
finally{}
```

Example 17:

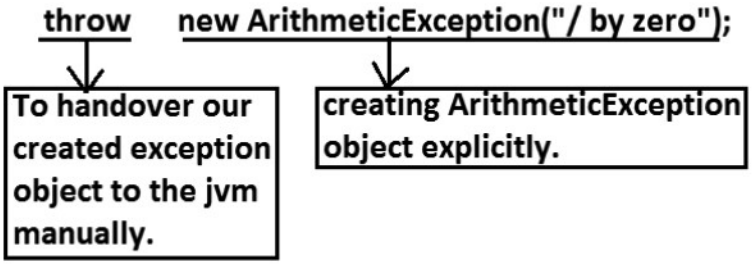
```
class Test1{
public static void main(String[] args){
try{ }
catch(Exception e){}
finally
{
try{}
catch(Exception e){}
finally{}
}
}
}
```

Output:

```
Compile and running successfully.
```

Throw statement: Sometime we can create exception object explicitly and we can hand over to the JVM manually by using throw keyword.

Example:



- The result of following 2 programs is exactly same.

class Test
{
public static void main(String[] args){
System.out.println(10/0);
}}
• In this case creation of ArithmeticException object and
	class Test
	{
	public static void main(String[] args){
	throw new ArithmeticException("/ by zero");
	}}
	• In this case we are creating exception

handover to the jvm will be performed automatically by the main() method.	object explicitly and handover to the JVM manually.

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

Case 1: throw e;

- If e refers null then we will get NullPointerException.

Example:

```
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
```

```
class Test
{
static ArithmeticException e;
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.

Example:

```
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)
```

```
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.

Example:

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

Output:

```
Compile time error.
```

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

Output:

Test3.java:4: incompatible types found : Test3 required: java.lang.Throwable throw new Test3();	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 compulsory we should handle either by try catch or by throws keyword otherwise the code won't compile.

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

Example:

By using try catch	By using throws keyword

class Test3
{
public static void main(String[] args){
try{
Thread.sleep(5000);
}
catch(InterruptedException e){}
}
}
Output:
Compile and running successfully
	• We can use throws keyword to delicate the responsibility of exception handling to the caller method. Then caller method is responsible to handle that exception.
	class Test3
	{
	public static void main(String[]
	args)throws InterruptedException{
	Thread.sleep(5000);
	}
	}
	Output:
	Compile and running successfully

- Hence the main objective of "throws" keyword is to delicate 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 convenes complier. Usage of throws keyword doesn't prevent abnormal termination of the program.

Example:

```
class Test
{
public static void main(String[] args)throws InterruptedException{
doStuff();
}
public static void doStuff()throws InterruptedException{
doMoreStuff();
}
public static void doMoreStuff()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.

Example:

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

Compile time error. Test3.java:3: unreported exception java.lang.Exception; must be caught or declared to be thrown	Runtime error Exception in thread "main" java.lang.Error at Test3.main(Test3.java:3)

Case 3:

- In our program if there is no chance of rising an exception then we can't right 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:

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(Exception e)
{}
}
}
```
output:
hello
partial checked

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(ArithmeticException e)
{}
}
}
```
output:
hello
unchecked

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(java.io.IOException e)
{}
}
}
```
output:
compile time error
fully checked

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(InterruptedException e)
{}
}
}
```
output:
compile time error
Fully checked

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(Error e)
{}
}
}
```
output:
compile successfully
unchecked

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
    - Requried: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).

Example:

1) InSufficientFundsException
2) TooYoungException
3) TooOldException

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>60)
{
throw new TooYoungException("please wait some more time.... u will get best match");
}
else if(age<18)
{
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:

1) E:\scjp>java CustomizedExceptionDemo 61
   Exception in thread "main" TooYoungException: please wait some more time.... u will get best match
   at CustomizedExceptionDemo.main(CustomizedExceptionDemo.java:21)
2) E:\scjp>java CustomizedExceptionDemo 27
   You will get match details soon by e-mail
3) E:\scjp>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.

Example:

```
try
{}
catch(Error e)
{}
```
valid

Top-10 Exceptions:

- Exceptions are divided into two types. They are:
  1) JVM Exceptions:
  2) Programatic exceptions:

JVM Exceptions:

- The exceptions which are raised automatically by the jvm whenever a particular event occurs.

Example:

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

Programatic Exceptions:

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

Example:

1) IllegalArgumentException(IAE).

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 methodOne()
{
methodTwo();
}
public static void methodTwo()
{
methodOne();
}
public static void main(String[] args)
{
```

```
methodOne();
}
}
```

Output:

Run time error: StackOverFloeError

4) NoClassDefFound:

- 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 typecast parent object to child type.

Example:

```
class Test
{
public static void main(String[] args)
{
String s=new String("bhaskar");
Object o=(Object)s;
}
}
```

output:
valid

```
class Test
{
public static void main(String[] args)
{
Object o=new Object();
String s=(String)o;
}
```

output:
Runtime exception:ClassCastException

```
class Test
{
public static void main(String[] args)
{
Object o=new String("bhaskar");
String s=(String)o;
}
}
```

output:
valid

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
}}
```

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.

Example:

- Once session expires we can't call any method on the session object otherwise we will get IllegalStateException

```
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) 6) ExceptionInInitializerError	Raised automatically by JVM(JVM Exceptions)
7) IAE(IllegalArgumentException) 8) NFE(NumberFormatException) 9) ISE(IllegalStateException) 10) AE(AssertionError)	Raised explicitly either by programmer or by API developer (Programatic Exceptions).