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. Various possible combinations of try catch finally

13. throw keyword

14. throws keyword

15. Exception handling keywords summary

16. Various possible compile time errors in exception handling

17. Customized exceptions

18. Top-10 exceptions

19. 1.7 Version Enhancements

1. try with resources

2. multi catch block

Introduction

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 remote file

locating at London. At runtime if London file is not available then 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

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.
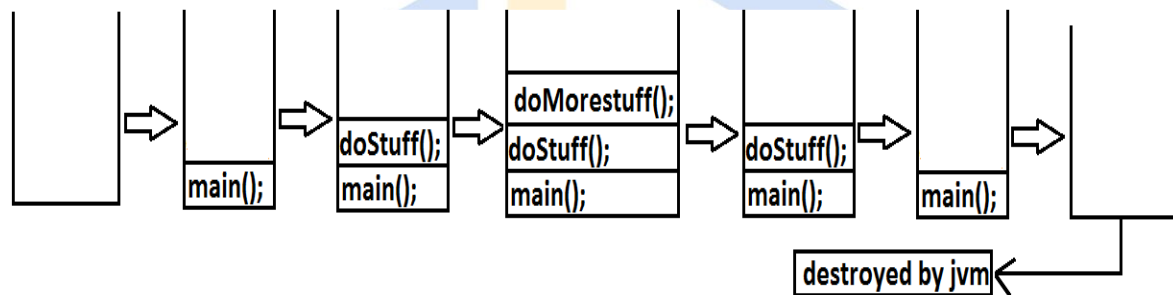
Example:

class Test

{

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

Output:

Hello

Diagram:

1. 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)

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 method also abnormally and 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 also 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 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){

doStuff();

}

public static void doStuff(){

doMoreStuff();

}

public static void doMoreStuff(){

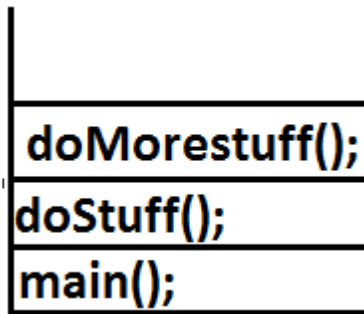System.out.println(10/0);

}}

Output:

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

atTest.doMoreStuff(Test.java:10)

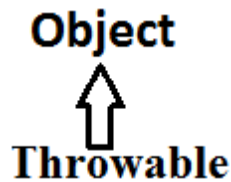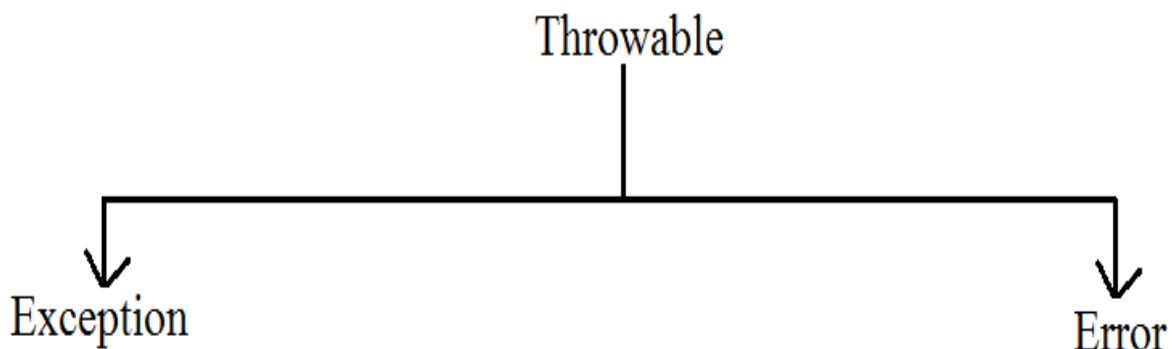atTest.doStuff(Test.java:7)

atTest.main(Test.java:4)

Diagram:

```
┌─────────────┐
│             │
├─────────────┤
│ doMorestuff();│
├─────────────┤
│doStuff();   │
├─────────────┤
│main();      │
└─────────────┘
```

Exception Hierarchy:

Object

⇑

Throwable

Throwable acts as a root for exception hierarchy.

Throwable class contains the following two child classes.

Throwable

Exception   Error

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.

Checked Vs Unchecked 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

 The exceptions which are not checked by the compiler whether programmer handing 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.

Fully checked Vs Partially 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

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

Diagram:
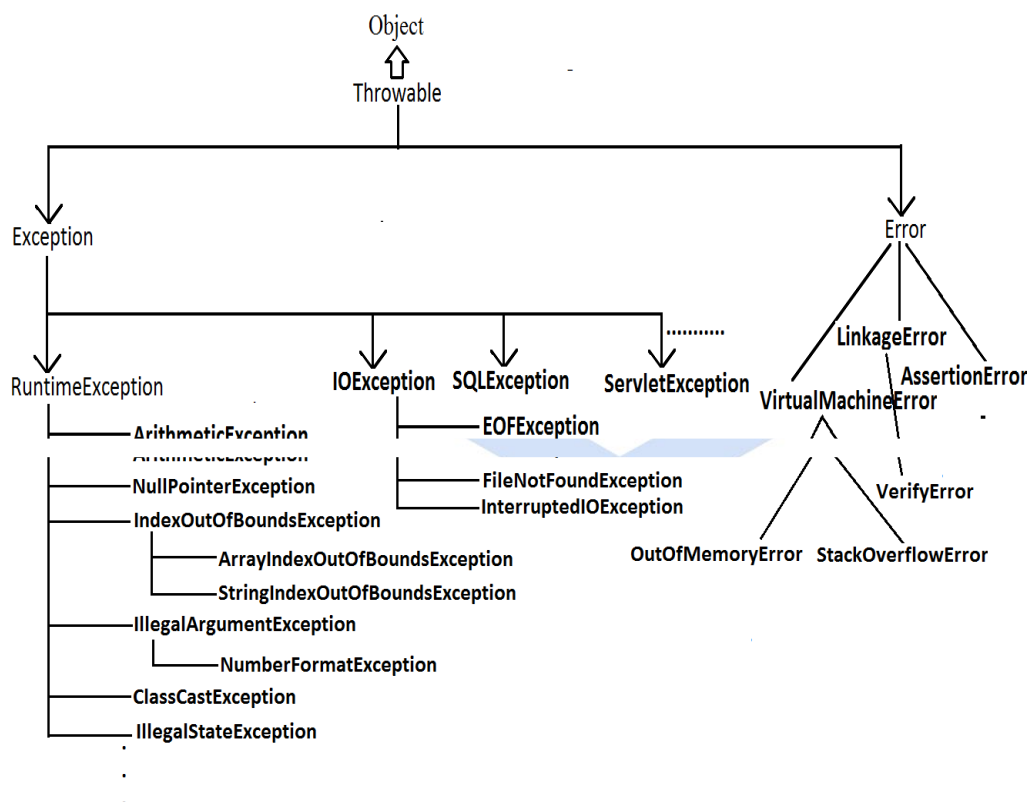
Customized Exception Handling by using try-catch:

 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

}

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");
}
}
output:
statement1
RE:AE:/by zero
at Test.main()

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:

try{

statement1;

statement2;

statement3;

}

catch(X e) {

statement4;

}

statement5;

☐ Case 1:If 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 block 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 type it is recommended to take a separate catch block. That is try with

multiple catch blocks is possible and recommended to use.

Example:

try { . . .	try { . . .

<table>
<tr><td>

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

</td><td>

```
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
}
```

</td></tr>
<tr><td>This approach is not recommended because for any type of Exception we are using the same catch block.</td><td>This approach is highly recommended because for any exception raise we are defining a separate catch block.</td></tr>
</table>

Finally block:

 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.

Example:

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

The speciality 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

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

Case-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

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.

When ever 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);

1. This argument acts as status code. Insteadof zero, we can take any integer value

2. zero means normal termination , non-zero means abnormal termination

3. 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.

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:

1. finally block meant for cleanup activities related to try block where as finalize()

method meant for cleanup activities related to object.

2. To maintain clean up code finally block is recommended over finalize() method

because we can't expect exact behavior of GC.

Example:

le:

```
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.

Note:

1.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.

2. We can take try-catch inside try i.e., nested try-catch is possible

3. 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:

1. Whenever we are writing try block compulsory we should write either catch or

finally. i.e., try without catch or finally is invalid.

2. Whenever we are writing catch block compulsory we should write try.

i.e., catch without try is invalid.

3. Whenever we are writing finally block compulsory we should write try.

i.e., finally without try is invalid.

4. In try-catch-finally order is important.

5. With in the try-catch -finally blocks we can take try-catch-finally.

i.e., nesting of try-catch-finally is possible.

6. 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
```
✗

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

```
try {}
finally {}
```
✓

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

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

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

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

```
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'
```
✗

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

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

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

```
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) {}
```
✗

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

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

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

throw statement:

Sometimes we can create Exception object explicitly and we can hand over to the JVM

manually by using throw keyword.

Example:

throw   **new ArithmeticException("/ by zero");**

↓ To handover our created exception object to the jvm manually.

↓ creating ArithmeticException object explicitly.

The result of following 2 programs is exactly same.

``java
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. | ```java
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.

Case 1:

```
throw e;
```

If e refers null then we will get NullPointerException.

Example:

```
class Test3
{
static ArithmeticException e=new
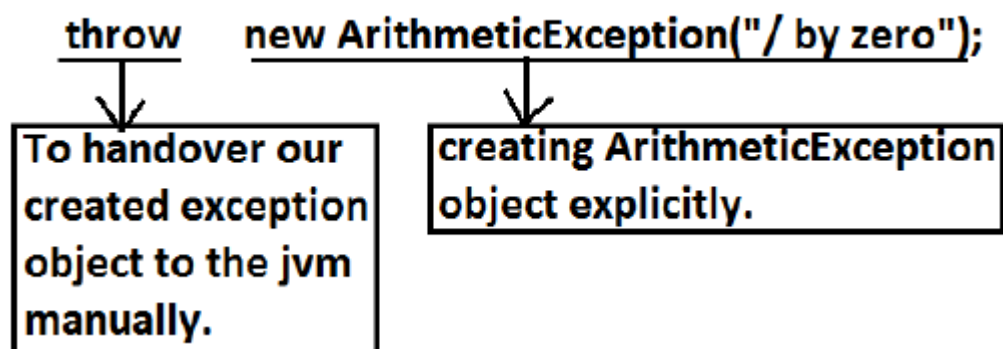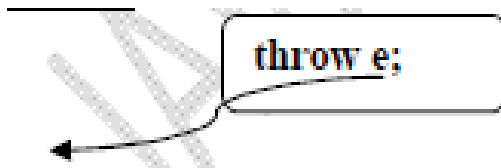
ArithmeticException();
public static void main(String[]
args){
throw e;
}
}
Output:
Runtime exception: Exception in thread
"main"

java.lang.ArithmeticException
```

```
class Test3
{
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	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)
```

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

```
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 compulsoryy we

should handle either by try catch or by throws keyword otherwise the code won't

compile.

class ExcTest7

{

    public static void main(String[] args)

    {

        Thread.sleep(1000);

    }

}

Unreported exception java.io.InterruptedExcption; 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.

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 delegate 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
``` |

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 complier. 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{
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:
Compile time error.
Test3.java:3: unreported exception
    java.lang.Exception;
must be caught or declared to be
thrown
```

```
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 with in the try block, 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

sayingexception 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
```

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
    { }
}
```

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.

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.

Sometimes we can create our own exception to meet our programing 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:

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

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.

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

1.7 Version Enhansements :

As part of 1.7 version enhancements in Exception Handling the following 2 concepts
Introduced

1. try with resources

2. multi catch block

1.try with resources

Untill 1.6 version it is highly recommended to write finally block to close all resources
which are open as part of try block.

```
BufferedReader br=null;
try{
br=new BufferedReader(new FileReader("abc.txt"));
//use br based on our requirements
}
catch(IOException e) {
// handling code
}
finally {
if(br != null)
br.close();
}
```

problems in this approach :

 Compulsory programmer is required to close all opened resources with increases

the complexity of the programming

 Compulsory we should write finally block explicitly which increases length of the

code and reviews readability.

To overcome these problems Sun People introduced "try with resources" in 1.7

version.

The main advantage of "try with resources" is

the resources which are opened as part of try block will be closed automatically

Once the control reaches end of the try block either normally or abnormally and hence

we are not required to close explicitly so that the complexity of programming will be

reduced.It is not required to write finally block explicitly and hence length of the code

will be reduced and readability will be improved.

try(BufferedReader br=new BufferedReader(new FileReader("abc.txt")))

{

use be based on our requirement, br will be closed automatically ,

Onec control reaches end of try either normally

or abnormally and we are not required to close explicitly

}

catch(IOException e) {

// handling code

}

Conclusions:

1. We can declare any no of resources but all these resources should be seperated with

;(semicolon)

try(R1 ; R2 ; R3)

{

}

2. All resources should be AutoCloseable resources. A resource is said to be auto

closable if and only if the corresponding class implements the java.lang.AutoCloseable

interface either directly or indirectly.

All database related, network related and file io related resources already

implemented AutoCloseable interface. Being a programmer we should aware and we

are not required to do anything extra.

3. All resource reference variables are implicitly final and hence we can't perform

reassignment with in the try block.

try(BufferedReader br=new BufferedReader(new FileReader("abc.txt"))) ;

{

br=new BufferedReader(new FileReader("abc.txt"));

}

output :

CE : Can't reassign a value to final variable br

4.Untill 1.6 version try should be followed by either catch or finally but 1.7 version we

can take only try with resource without catch or finally

try(R)

{ //valid

}

5.The main advantage of "try with resources" is finally block will become dummy

because we are not required to close resources of explicitly.

Multi catch block :

Until 1.6 version ,Eventhough Multiple Exceptions having same handling code we have

to write a separate catch block for every exceptions, it increases length of the code and

reviews readability

```
try{
----------------
----------------
}
catch(ArithmeticException e) {
e.printStackTrace();
}
catch(NullPointerException e) {
e.printStackTrace();
}
catch(ClassCastException e) {
System.out.println(e.getMessage());
}
catch(IOException e) {
System.out.println(e.getMessage());
}
```

To overcome this problem Sun People introduced "Multi catch block" concept in 1.7 version.

The main advantage of multi catch block is we can write a single catch block , which can handle multiple different exceptions

```
try{
----------------
----------------
}
catch(ArithmeticException | NullPointerException e) {
e.printStackTrace();
}
catch(ClassCastException | IOException e) {
System.out.println(e.getMessage());
}
```

In multi catch block, there should not be any relation between Exception types(either

child to parent Or parent to child Or same type , otherwise we will get Compile time

error)

Example:

```
try {

    ------
}
catch(AE | Exception  e)
       c         p
{
  e.printStackTrace();
}
```


invalid

Exception Propagation :

With in a method if an exception raised and if that method doesn't handle that exception, then Exception object will be propagated to the caller then caller method is

responsible to handle that exceptions. This process is called Exception Propagation.

Rethrowing an Exception :

To convert the one exception type to another exception type , we can use rethrowing

exception concept.

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

```
}
}
```
output:

RE:NPE