Exception Handling

1. Introduction
2. Runtime Stack Mechanism
3. Default exception Handling in java
4. Exception Hierarchy
5. Customized exception handling using try catch
6. Control flow in try catch
7. Methods to print exception information
8. try with multiple catch blocks.
9. finally block
10. diff. btw final, finally & finalize.
11. control flow in try-catch-finally
12. " " " in nested " " "
13. Various possible combinations of try, catch
14. throw
15. throws } Keyword
16. Exception handling summary
17. Various possible compile time errors in EH.
18. Top 10 - exceptions.
19. 1.7v enhancements
    1. try with resources
    2. multi-catch block.

Introduction

→ An unexpected, unwanted event that disturbs normal flow of the program is called exception.
eg → Tired Punctured Exception,
Sleeping Exception, etc.

→ It is highly recommended to handle exceptions & the main objective of E.H is graceful termination of the program.

→ Exception Handling doesn't mean repairing an exception.
We have to provide alternative way to continue the rest of program normally, it is called E.H.

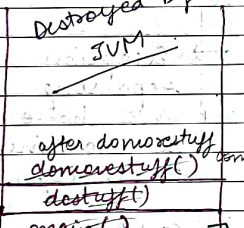
eg → Our programmer need is to read data from file locating in london At Runtime, if london file is not available our program should not be terminated abnormally.
We have to provide some local file to continue rest of program normally.
This way of defining alternative way is E.H.

```
try
{
    Read data from Remote
    file in london
}
catch ( FileNotFoundException )
{
    use local file & continue
    rest of program
}
```

Runtime Stack Mechnaism

eg →
```
class test
{
    psvm
    {
        dostuff();
    }
    ps v   dostuff()
    {
        domorestuff();
    }
    ps v   domorestuff()
    {  Sop ln ("Hello");
    }
}
```

o/p → Hello



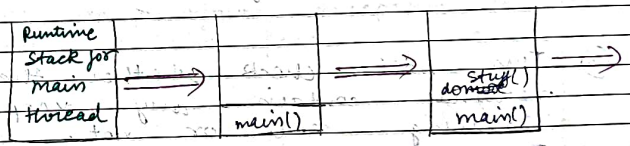
→ for every thread, JVM will create a Runtime stack.

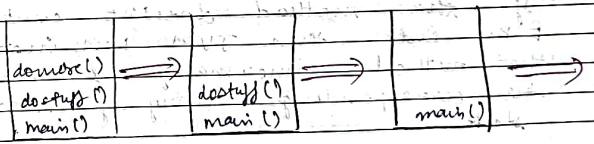
→ Each & every method call performed by that thread will be stored in the corresponding stack.

→ Each entry in the stack is called Stack Frame or Activation Record.

→ After completing every method call the corresponding entry from the stack will be removed.

→ After completing all methods calls the stack will become empty & that empty stack will be destroyed by JVM just before terminating the thread.





→ this empty stack will be destroyed by JVM

Default Exception Handling

(1) Inside a method if any exception occurs, the method in which it arise is responsible to create exception object by including the following information —

① Name of exception

② Description of exception

③ Location at which exception occure [Stack trace]

(2) After creating exception object method handovers that object to the JVM.

(3) JVM will check whether the methonds contain any exception handling code or not.
If the method doesn't contain E.H. code then JVM terminate that method abnormally & removes corresponding entry from the stack.

(4) Then, JVM identifies caller method & checks whether caller method contains any handling code or not.
If the caller method doesn't contain handling code then JVM terminates that caller method abnormally & removes corresponding entry from stack.

(5) This process will be continued until main method & if the main method also doesn't contain handling code then, JVM terminates main method also abnormally. & removes corr. entry from the stack.

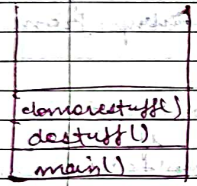
(6) Then JVM handover responsibility of E.H to default exception Handler, which is the part of JVM.

→ DEH prints exception information in the following format & termina-tes program abnormally.

exception in thread "xxx" Name of exception : Description
Stack Trace

eg → (1)

```
class Test
{
    psvmain
    {
        dostuff();
    }
    ps v dostuff()
    {
        domorestuff();
    }
    ps v domorestuff()
    {
        sopln(10/0);
    }
}
```



→ Exception in thread "main"
java.lang.Arithmetic Exception
Division by 0 — at test.domorestuff
at test.dostuff()
at test.main()

(2)

```
class Test
{
    ps vm
    {
        dostuff();
        Sopln(10/0);
    }
    ps v dostuff()
    {
        domorestuff()
        Sopln("H");
    }
    ps v domorestuff()
    {
        Sopln("Hello");
    }
}
```

O/P → Hello
Exception in thread "main"
at Test.main()

NOTE → In a program if atleast one method terminates abnormally then the program termination is abnormal termination.

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

Exception Hierarchy

→ Throwable class acts as Root for java exception hierarchy.

→ Throwable class defines two child classes —
1. Exception
2. Error

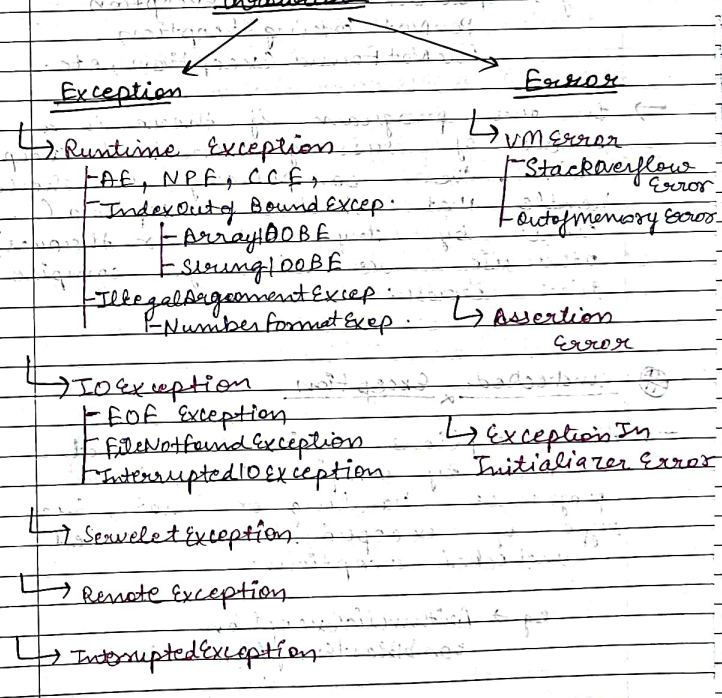
Exception → Most of the times, exception are caused by our program and these are recoverable.

eg →
```
try
{
    Read data from Remote file locating in london
}
catch(FileNotfound Exception)
{
    use local file & continue rest of the program
}
```

Error → Most of the times, errors are not caused by our program and these are non-coverable.

↳ & caused because of lack of system resources.

eg → If OutofMemoryOccurs during a programmer we can't do anything & the program will be terminated abnormally.
System Admin or Server Admin is responsible to increase heap memory.



Checked VS Unchecked Exception

Checked Exception

→ The exception which are checked by compiler for smooth execution of the program.

eg → HallTicketMissing Exception
PenNotWorking Exception
FileNotFound Exception, etc.

→ In our program if there is a chance of raising checked exception, then compulsary we should handle that checked exception (either by try, catch or throwes) otherwise we will get compile time error.

Unchecked Exception

→ The exceptions which are not checked by compiler whether programer handling or not, such type of exceptions are called unchecked exceptions.

eg → ArithmeticException
BombBlastException, etc.

→ NOTE →

① whether it is checked / unchecked every exception occurs at Runtime only.
There is no chance of occuring exception at compile time.

② Runtime exception & its child classes, Error & its child classes are unchecked.
Except these remaining are checked.

fully checked VS partially checked

A checked exception is said to be fully checked if & only if all its child classes also checked.
eg — IO exception
InterruptedException

A checked exception is said to be partially checked if & only if some of its child classes are unchecked.
eg — Exception
Throwable

NOTE →

the only possible partially checked exception in java are —

1) Exception
2) Throwable

Qus → Describe the behaviour of following Exceptions —

IOException	fully checked
RuntimeException	unchecked
InterruptedException	fully checked
Error	unchecked
Throwable	Partially
AE	unchecked
NPE	unchecked
Exception	Partially checked
FileNotFoundException	fully checked

Customised Exception Handling by using — try, catch

It is highly recommended to handle exceptions.

The code which may raise an exception is called Risky code. & we have to define that code inside try block. & corresponding handling code we have to define inside catch block.

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

→ without try, catch →

```
class Test
{
    psv main
    {
        Sopln ("hi");
        Sopln (10/0);
        Sopln ("bye");
    }
}
```

O/P → hi
RE: / by zero

Abnormal Termination

→ With try, catch

```
class Test
{
  psv main
  {
    Sop("Hi");
    try
    {
      Sop(10/2);
    }
    catch (ArithmeticException)
    {
      Sop(10/2);
    }
    Sop("Go");
  }
}
```

O/P →
Hi
5
Go
Normal Termination

→ Control flow in try, catch

```
try
{
  Stat 1;
  Stat 2;
  Stat 3;
}
catch (X e)
{
  Stat 4;
}
  Stat 5;
```

Case 1 → If there is no exception.

O/P → 1, 2, 3, 5 (Normal Termi.)

Case 2 → If an exception raised at statement 2 & corresponding catch block matched

O/P → 1, 4, 5, NT

Case 3 → If an exception raised at statement 2 & corresponding catch block not matched

O/P → 1, Abnormal Termination

Case 4 → If an exception raised at statement 4 or statement 5 then it is always abnormal termination.

* NOTE → ① within the try block, if anywhere exception raised then rest of the try block won't be executed even though we handled that exception.
Hence, within the try block we have to take only risky code.

→ Length of try block as less as possible.

(2) In additional to try block there may be a chance of rising an exception inside catch & finally blocks.
If any statement which is not part of try block & rises an exception then it is always abnormal exception.

Methods to print exception information

Throwable class defines the following methods to print exception info —

	Method	Printable format
(1)	printStackTrace()	Name of exception: Description Stack Trace
(2)	toString()	Name of exception: Description
(3)	getMessage()	Description

eg →
```
class Test
{
  psvm(...)
  {
    Try
    { Sopln(10/0); }
    catch (Arithemtic exception e)
    {
      e.printStackTrace();
      Sopln(e) or Sopln(e.toString());
      Sopln(e.getmessage());
    }
  }
}
```

→ java.lang.Af :/ by zero
at Test.main()

java.lang.Af :/ by zero

/ by zero

NOTE →
Internally, default exceptional handler will use 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 it is highly recommended to take separate catch block i.e try with multiple catch block is always possible & recommended to use.

eg →

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

worst practice

Best / Good practice :→

```
try
  {
    Risky code;
  }
catch (AF e)
  {
    perform alternate operation;
  }
catch (SQL Exception e)
  {
    use mySQL DB instead of oracle;
  }
catch ( FileNotFound Exception e)
  {
    use local file;
  }
catch (exception e)
{ // default + exception-handling;
}
```

→ If try with multiple catch block presents then, the order of catch blocks is very important we have to take child first & then parent otherwise we will get compile time error saying Exception xxx has already been caught.

eg →

```
try
  {
    Risky code;
  }
catch (exception e);
  { . . . }
catch (AF e)
  { . . . }
```

CE: exception java.AF has already been caught.

otherwise ↑

```
try
  {
    Risky code;
  }
catch (AF e)
  {
  }
catch (AF e)
  {
  }
```

we can't declare two catch blocks for the same exception.

```
try
  {
    Risky code;
  }
catch (AF e)
  {
  }
catch (exception e)
  {
  }
```

✓

final –

→ It is a modifier applicable for classes, methods & variables.

→ If a class declared as final then, we can't extend that class i.e we can't create child class for that class it means inheritance is not possible for final classes.

→ If a method is final, then we can't override in the child class.

→ If a variable declared as final then we can't perform reassignment for that variable.

finally()

→ It is a block always associated with try catch to main cleanup code.

eg →
```
try
{ Risky code;
}
catch (exception e)
{
}
finally
{ cleanup code;
}
```

→ The speciality of finally block is – it will be executed always irrespective of whether exception is raised or not raised & whether handled / not handled.

finalise()

→ finalise is a method always invoked by garbage collector just before destroying an object to perform cleanup activities.

→ Once finalize methods completes, immediately garbage collector destroy that object.

NOTE →

→ finally block is responsible to perform cleanup activities related to try block i.e whatever resource we open at the part of try block will be closed inside finally block.

→ Whereas finalise method is responsible to perform cleanup activities related to object i.e whatever resource we open for object will be deallocated before destroying an object by using finalise.

Various possible combinations of try catch finally —

1. In try, catch finally — order is important

2. Whenever we are writing try compulsory we should write either catch or finally otherwise we will get CE: try without catch or finally is invalid.

3. Whenever we are writing catch block compulsory we should write try block otherwise CE: catch without try is invalid.

4. Whenever we are writing finally compulsory we should write try block otherwise CE: finally without try is invalid.

5. Inside try, catch & finally blocks we can declare try, catch & finally blocks i.e nesting of try, catch, finally is allowed.

6. Curly brackets are mandatory for try, catch & finally blocks.

1.
```
try
{
}
catch(x e)
{
}
```

2.
```
try
{
}
catch(x e)
{ }
catch(y e)
{ }
```

3.
```
try {
}
catch(x e)
{ }
catch(x e)
{ }
```
CE: exception e has already been declared.

4.
```
try { }
catch(x e)
{ }
finally
{ }
```

5.
```
try
{ }
finally
{ }
```

6.
```
try { }
catch(x e)
{ }
try { }
catch(y e)
{ }
```

7.
```
try { }
catch(x e)
{ }
try { }
finally
{ }
```

8.
```
try { }
```

9.
```
catch(x e)
{ }
```

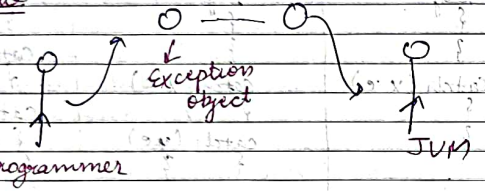
CE: try/catch/finally without try/catch/finally

10.
```
finally
{ }
```

11.
```
try
{ }
finally
{ }
catch(x e)
{ }
```

12.
```
try {
try { }
catch(x e)
{
}
catch(x e)
{
}
```

Throw



Sometimes, we can create exception explicitly and handover to the JVM manually for this we have to use throw Keyword.

→ throw new ~~AE~~ Exception("/ by zero");

[creation of AE explicitly]

handover our created exception object to the JVM manually

Hence, the main objective of throw keyword is to handover our created exception to the JVM manually.

The result of following two programs are same —

①
```
class Test
{
  psv main
  {
    sopln(10/0);
  }
}
```
Exception in thread "main" j.l.AE: / by zero
at Test.main()

②
```
class Test
{
  psv main
  {
    throw new AE("/ by 0");
  }
}
```
Exception in thread "main" j.l.AE: / by zero
at Test.main()

In 1st case, main method is responsible to create exception object & handover to the JVM.

In 2nd case, programmer creating exception object explicitly & handover to the JVM manually

NOTE →
Best use of throw keyword is for user defined exceptions or customised exceptions.

Case 1 → throw e;

If e refers null then we will get NullPointerException.

```
class Test{
Static AE e= new AE();
 psvm
  {
    throw e;
  }
}
```

RE: AE

```
class Test
{
 static AE e;
  psvm
   {
     throw e;
   }
}
```

RE: NPE

Case 2 → After throw, we are not allowed to write any statement directly otherwise we will CE saying unreachable statement.

```
class Test
{
 psvm
 {
  sopln("10/0");
  Sopln ("Hi");
 }
}
```

RE: AE / by zero

```
class Test
 {
   psvm
   {
     throw new
  AE ("/by zero");
   Sopln ("Hi")
   }
 }
```

CE: unreachable statement

Case 3 → We can use throw keyword only for throwable types.

If we are trying to use normal java objects we will get CE saying incompatible types.

```
class Test
{
psvm
 {
 throw new Test();
 }
}
```

CE: incompatible types
found: Test
required: java.lang.Throwable

```
class Test extends RE
 {
  psvm
   {
   throw new Test();
   }
 }
```

RE: Exception in thread "main" Test
at Test.main()

throws

In our program, if there is a possibility of rising checked excep. then, compulsory we should handle that exception otherwise we will get CE : unreported exception ...

eg → ① class Test
{
psvm ...
{
PrintWriter pw = new PrintWriter("abc.txt");
pw.println("Hello");
}
}
CE: unreported exception java.io.FileNotFoundEx must be caught or declared to be thrown.

② class Test
{
psvm ...
{
Thread.sleep(10000);
}
}
CE: unreported exception java.lang.Interrupted must be caught or declared to be thrown.

We can handle this CE by two ways -

First way → By using try catch

eg → class Test
{
psvm main
{
try
{
Thread.sleep(10000);
}
catch(InterruptedException)
{
}
}
}

Second way → By using throws keyword

eg → class Test
{
psv main [String [] args] throws IE
{
Thread.sleep(10000);
}
}

→ We can use throws to delicate responsibility of EH to the caller (it may another method or JVM) then caller method is responsible to handle that exception.

→ Throws keyword required only for checked exception & usage of throws keyword unchecked exception — there is no use or impact.

→ Throws required only to convince compiler & usage of throws does not prevent abnormal termination of program.

eg →
```
class Test
{
  psv main(String[] args) throws IE
  {
    dostuff();
  }
  psv dostuff() throws IE
  {
    domorestuff();
  }
  psv domorestuff() throws IE
  {
    Thread.sleep(10000);
  }
}
```

E: unreported exce. j.t IE must be caught or thrown

In the above program, if we remove at least one throws statement then the code won't compile.

We can use to delegate responsibilities of EH to the caller

Throws

It is required only for ✓ exceptions & usage of throws for unchecked, there is no use

Required only to convince compiler & usage of throws does not prevent abnormal termination of program.

Case 1 → We can use throws for methods & constructors but not for classes.

eg →
```
class Test throws exception ✗
{
  Test() throws exception ✓
  {
  }
  psv m1() throws exception ✓
  {
  }
}
```

case 2 → We can use throws keyword only for throwable types if we are trying to use for normal java classes then we will get CE saying incompatible types.

eg → ① class Test
{
public void m1() throws Test
{
}
}

CE: incompatible types
found: Test
required: j.l.Throwable

② class Test extends RuntimeException
{
public void m1() throws Test
{
}
}
✓

case 3 → ① class Test
{
ps v main
{
throw Exception(); → checked
}
}

CE: unreported exception j.l.Exception must be caught or declared to be thrown.

② class Test
{
ps v main
{
throw Error(); → unchecked
}
}

RE: Exception in thread "main" j.l.Error
at Test.main()

case 4 → 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 CE: Exception XXX is never thrown in body of corresponding try statement.

But this rule is applicable only for fully checked exceptions.

eg → ① class Test
{
ps v main
{
try
{ sopln("Hello") }
catch(AE e) unchecked
{ }
}
}
✓

o/p → Hello

There is no such as "throwon" keyword in java.

(2)
```
class test
{
    psvm                          o/p → crazy
    {
      try
      { sopln("crazy"); }
      catch(exception e)          partially checked
        { }
    }
}
```

(3)
```
import java.util.io;
class test
{
  psvm
  {
    try
    { sopln("Hi"); }              → fully checked
    catch(IOException e)
    { }
  }
}
```
CE: exception in java.io.IOException

(4)
```
class test
{ psvm {
    try
    { sopln("Hi"); ; }            fully checked
    catch(Unreputedexception e)
    { }
  }
}
```
CE: exception in j.l.IE

Customized Exception (Summary)

① try → to maintain Risky code

② catch → to maintain exception 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 EH. to the caller.

Various possible compile time error in EH →

→ unreported exception xxx; must be caught or declared to be thrown

→ exception xxx has already been caught

→ exception xxx is never thrown in body of corresponding try statement

→ unreachable statement

→ incompatible types

→ try/catch/finally without catch/finally/try.

Customized / User Defined Exception

Sometimes, to meet programming requirements, we can define our own exceptions, Such type of exceptions are called customized or user defined exceptions.

eg → TooYoung, TooOld, InsufficientFunds → Exceptions

eg →
```
class TooYoungException extends RuntimeException
{
    TooYoungException(String s)
    {
        Sop ln(s);
    }
}
```
→ to make description available to default exception handler

```
class TooOldException extends RE
{
    TooOldException(String s)
    {
        Sop ln (s);
    }
}
class custException
{
    psvm ... {
    int age = Integer.parseInt(args[0]);
    if (age > 60)
    {
        throw new TooYoungException("wait");
    }
    Else if (age < 18)
    {
        throw new TooOldException("age passed");
    }
    else
    {
        throw Sop ln(" soon ");
    }
}
}
```

NOTE →

→ throw is best suitable for user defined or customised exceptions. But not for pre-defined exceptions.

→ It is highly recommended to defined customised exceptions as unchecked, i.e we have to extends RE but not Exception.

→ We use Sop ln in customised exception to make description available to default Exception Handler to print statement.