Exception Handling

Date __/__/____
Page ____

1. Introduction
2. Runtime stack mechanism
3. Default Exception Handling in Java
4. Exception Hierarchy
5. Customized Exception Handling by 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. difference b/w 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 keyword summary
17. various possible compile-time errors in exception handling
18. Customized (or) user-defined exceptions
19. Top-10 exceptions
20. 1.7 version enhancements
    (i) try-with resources
    (ii) multi-catch block

*** Checked Exception (vs) Unchecked Exception

Scanned with CamScanner

Introduction

⇝ An unexpected, unwanted event that disturbs the normal flow of program is called exception.

Ex: TyrePunctured Exception
SleepingException
FileNotFoundException

⇝ It is highly recommended to handle exceptions & the main objective of exception handling is graceful termination of program.

⇝ Exception handling doesn't mean repairing an exception, we have to provide an alternative way to continue rest of the program normally, is a concept of exception handling.

For example, our programming requirement is to read data from remote file locating at 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 the program normally, this way of providing alternative is known as exception handling

```
try
{
    Read data from remote file
    locating at london
}
catch (FileNotFoundException e)
{
    Use local file & continue
    rest of program normally
}
```

Runtime stack Mechanism

⇝ For every thread, JVM will create a runtime stack, each & every method call performed by that thread will be stored in corresponding stack.

- Each entry in stack is called Stack Frame / Activation Record

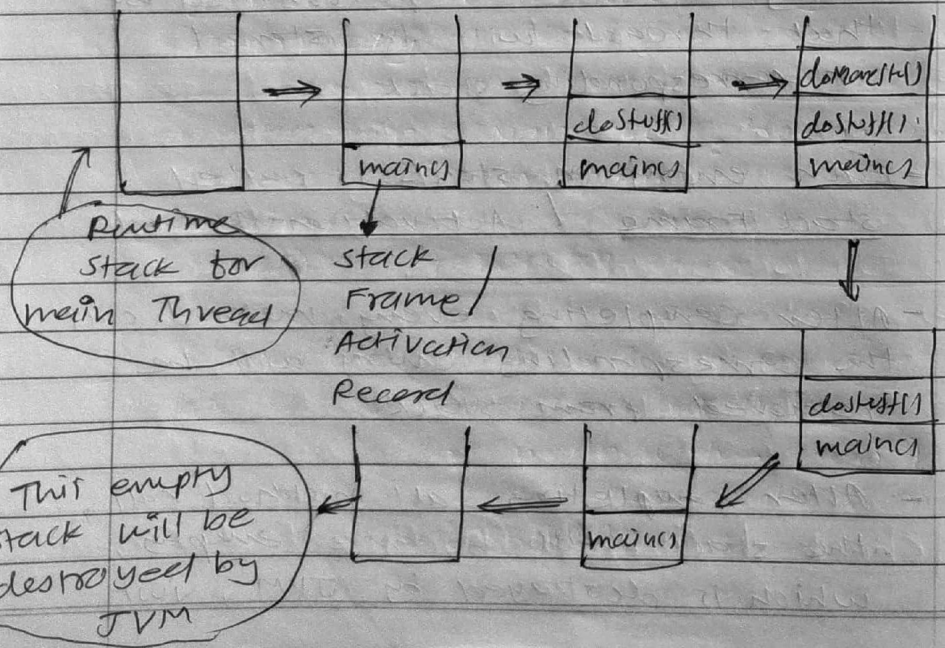
- After completing every method call, the corresponding entry will be removed from stack

- After completing all method calls the stack will became empty, which is destroyed by JVM, just

Scanned with CamScanner

before terminating the thread.

Ex.
```
class Test
{
    p s v main(String args)
    {
        doStuff();
    }
    p s v doStuff()
    {
        doMoreStuff();
    }
    p s v doMoreStuff()
    {
        S.O.P("Hello");
    }
}
```



⊛ Default Exception Handling in Java

↝ Inside a method if any exception occurs, the method in which it raised is responsible to create an object (Exception object) By including the following information:

① Name of exception
② Description of exception
③ location at which exception occurs (Stack-Trace)

↝ After creating exception object, method handovers that object to the JVM.

↝ JVM will check whether the method contains any exception handling code or not, if the method doesn't contains exception handling code then JVM terminates that method abnormally & removes corresponding entry from stack.

↝ Then JVM identifies caller method & checks whether caller method contains any handling code or not, if caller method doesn't contains handling code, then JVM terminates

Scanned with CamScanner

that caller method abnormally also & removes the entry from stack.

⇝ This process will be continued until main method & if the main method also doesn't contains handling code then JVM terminates that main method abnormally & removes the entry from stack

⇝ Then JVM hand-overs responsibility of exception handling to "default exception handler", which is the part of JVM.

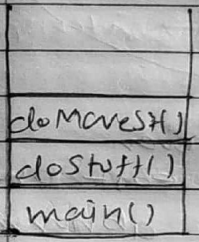
⇝ Default exception handler prints exception info in the following format & terminates program abnormally:-

Exception in thread main	: Name of exception	: Description	: stack Trace

Ex:
```
class Test
{
    p s v main(Strn args)
    {
        doStuff();
    }
    p s v doStuff()
    {
        doMoreStuff();
    }
    p s v doMoreStuff()
    {
        s.o.p(10/0);
    }
}
```

R.E:

```
Exception in thread main:
J.L.AE: / by zero
    at Test.doMoreStuff()
    at Test.doStuff()
    at Test.main()
```



Ex:
```
class Test
{
    p s v main(Strn args)
    {
        doStuff();
        s.o.p(10/0);
    }
    p s v doStuff()
    {
        doMoreStuff();
        s.o.p("Hii");
    }
}
```

REDMI NOTE 8 AI QUAD CAMERA
Scanned with CamScanner

```
p2 , v doMoreStuff()
        s.o.p("Hello");
    }
}
```

O/P :
Hello
Hii
Exception in thread main :
i.e. AE : / by zero
at Test.main()

Note : In a program, If even At least one method terminates abnormally then program termination is abnormal.

- If All method terminated normally then only program terminates normally.

Exception Hierarchy

~ Throwable (class) act as a (Root) for java exception hierarchy.

~ Throwable defines two child classes (i) Exception (ii) Errors.

~ Exception : Most of the times exceptions are caused by our programs & these are Recoverable.

For ex., if our programming requirment is to locate (Read) data from remote file locating at Lonelam ..... (that example) on previous page

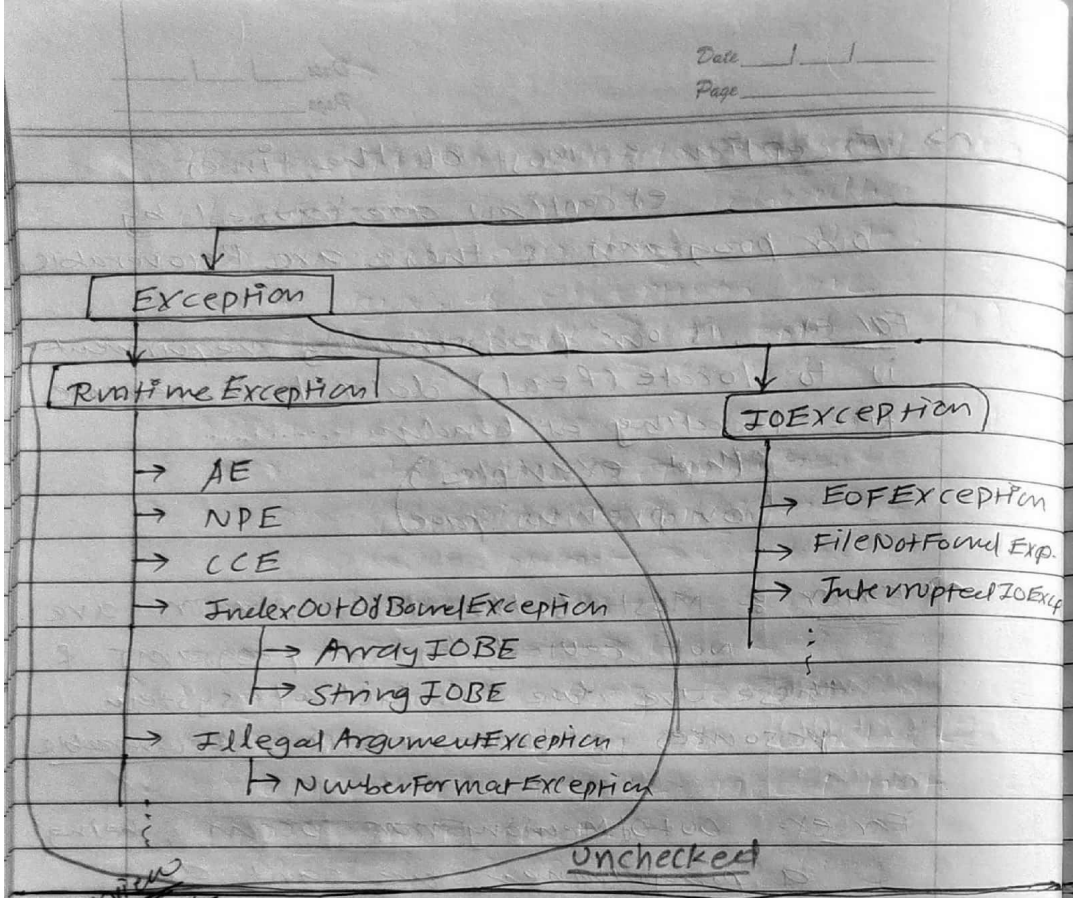
~ Error : Most of the times, errors are Not caused by our programs & these are due to lack of system resources & these are Non-Recoverable.

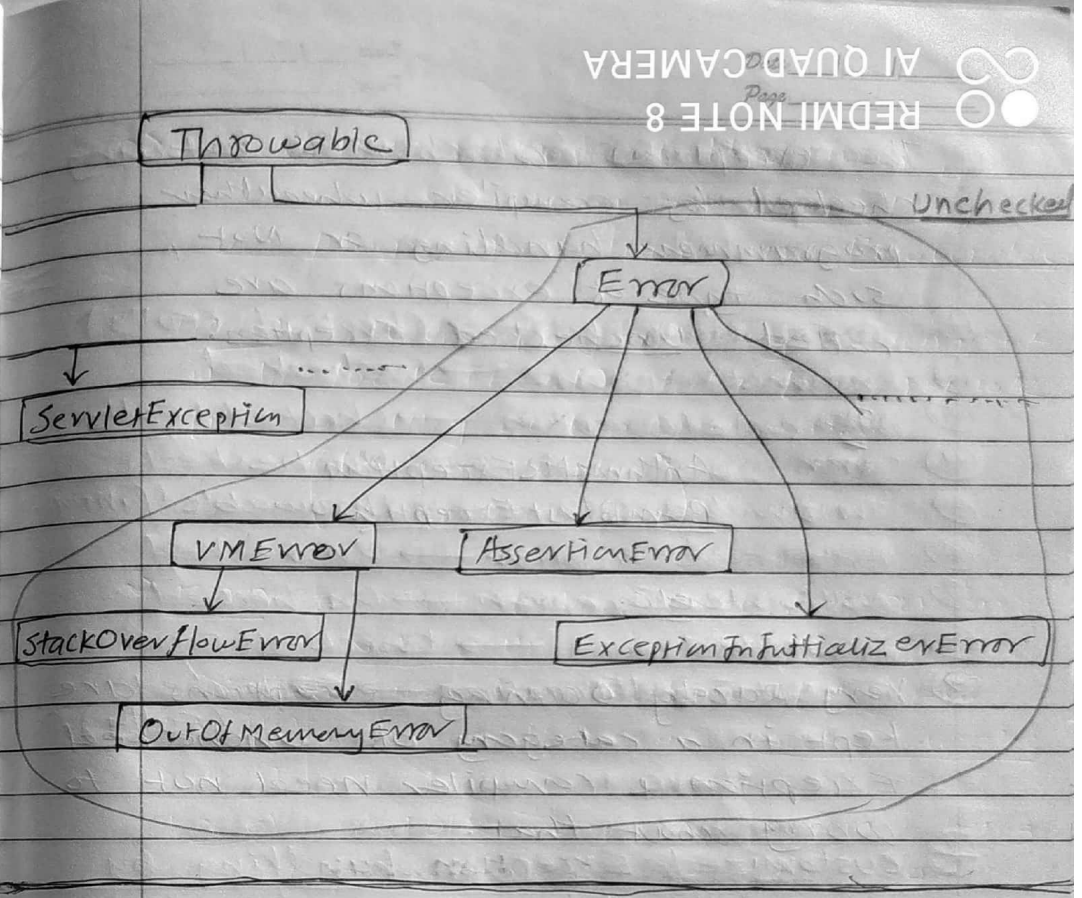
For ex. OutOfMemoryError occurs, being a programmer we can't do anything & program will be terminated abnormally.

System Admin / server Admin is responsible to increase heap memory.

~ partial hierarchy of Exceptions are as follows :

REDMI NOTE 8 AI QUAD CAMERA
Scanned with CamScanner





Interview ★★...

⊛ Checked Exception (VS) Unchecked Exception

⇝ The exceptions which are checked by compiler, for smooth execution of programs at runtime, are called checked Exception.

Ex: ⇒ HallTicketMissingException
PenNotWorkingException
;
FileNotFoundException
etc..

⇝ In our program, if there is a chance of rising checked Exception then compulsory we should handle that checked Exception. (Either by try-catch or throws keyword) otherwise we will get C.E.

⇝ Common occuring Exceptions are generally puted into the category of Checked Exceptions.

⇝ So, we can say that compiler is being pre-cautionary about checked Exception

Scanned with CamScanner

Date / /
Page

↝ The exceptions which are not checked by compiler whether programmer handling or Not, such type of exceptions are called Unchecked Exception.

Ex.

ArithmeticException
BombBlastException
↓
etc.

↝ Very Rarely occuring exceptions are kept in a category of Unchecked Exception & compiler need not to worry about that.

Note: Whether it is checked / Unchecked, Every Exception occurs at Runtime only, there is no chance of occuring any Exception at compile-Time.

→ Runtime Exceptions & its child classes
+
Error & its child classes are unchecked.
(Biz they occer rarely & it occers also, we can't do much)

↝ Except these all are Checked Exception.

↝ Fully Checked (vs) Partially Checked

→ A checked Exception is said to be fully checked, if & only if all its child classes are also checked.

Ex. IOException
InterruptedException

→ A checked Exception is said to be Partially checked, if & only if some of its child classes are Unchecked.

Ex. Exception
Throwable

★ Note: Note any 3 Partially Checked Exception in Java.

Ans: The only possible partially checked Exception in Java are:
Exception
Throwable
(2 only)

Scanned with CamScanner

Date / /
Page

↝ Describe the behaviour of following Exceptions :-

① IOException → Checked (Fully)
② RuntimeException → Unchecked
③ InterruptedException → Checked (Fully)
④ Error → Unchecked
⑤ Throwable → Throwable (Partially)
⑥ ArithmeticException → Unchecked
⑦ NullPointerException → Unchecked
⑧ Exception → Checked (Partially)
⑨ FileNotFoundException → Checked (Fully)

⊛ Customized Exception handling by using try-catch

↝ It is highly recommanded to handle exception.

↝ The code which may raise an exception is called ... Risky code & we have to define that code into 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
{
    p s v main(St- args)
    {
        s.o.p("stmt1");
        s.o.p(1/0);
        s.o.p("stmt3");
    }
}
```

Abnormal Termination

O/p : stmt1
RE : AE : / by zero

With try-catch

```
class Test
{
    p s v main(St- args)
    {
        s.o.p("stmt1");
        try
        {
            s.o.p(1/0);
        }
        catch(AE e)
        {
            s.o.p(1/2);
        }
        s.o.p("stmt3");
    }
}
```

Normal Termination

O/p :- stmt1, 5, stmt3

REDMI NOTE 8 AI QUAD CAMERA

Scanned with CamScanner

✱ Control flow in try-catch

```
try
{
    stmt1;
    stmt2;
    stmt3;
}
catch(X e)
{
    stmt4;
}
stmt5;
```

case 1 : If there is NO Exception
1, 2, 3, 5
Normal Termination

case 2 : If an Exception raised at stmt 2 and corresponding catch block matched
1, 4, 5
Normal Term

case 3 : If an Exception raised at stmt 2 & corr. catch block NOT matched
1.
Abnormal Ter.

case 4 : If Exception raised at stmt 4/stmt 5 then it is always abnormal termination.

Note : Within the try block if any where an 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

And length of try block should be as less as block.

~ In Addition to try block, there may be a chance of raising an exception inside catch & finally blocks

~ If any stmt, which is not part of try block & raises an exception, then it is always Abnormal termination

Scanned with CamScanner

Methods to print exception information

→ Throwable class defines the following methods to print exception info^n.

	Method	printable format
①	printStackTrace()	Name of Exception: Description Stack Trace
②	toString()	Name of Exception: Description
③	getMessage()	~~Name of Exception:~~ Description

Ex.

```
class Test
{
    p s v main (Strn args)
    {
        try
        {
            S.o.p(10/0);
        }
```

```
        catch (ArithmeticException e)
        {
            e.printStackTrace(); → j.l.AE: / at Test.m
            S.o.p(e);
            //S.o.p(e.toString()); → J.l.AE: / by
            s.o.p(e.getMessage()); / by ze
        }
    }
}
```

Note & ★ Internally, Default Exception Handler will use → printStackTrace()

try with Multiple catch blocks

→ The way of handling an exception is varied from exception to exception. Hence

For every exception type, it is highly recommanded to take separate catch block, i.e try-with Multiple catch blocks is always possible & recommanded to use.

Scanned with CamScanner

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

Worst prog. practice

```
try
{
    // Risky code
}
catch (AE e)
{
    // Perform AE specific handling
       & alternative op
}
catch (SQLE e)
{
    // Perform SQLE spec. handling
}
catch (FNFE e)
{
    // Perform FNFE spec. handling
       keep ready local file
}
catch (ExceptionE)
{
    // default handling
}
```

Good / Best prog. practice

⇝ If try-with-multiple catch blocks present, then order of catch blocks is very important, we have to take child first & then parent otherwise we will get C.E:

Exception xxx has already been caught

```
try
{
    // Risky code
}
catch (Exception e)
{
}
catch (AE e)
{
}
```

CE: Exception j.l.AE has already been caught

```
try
{
    // Risky code
}
catch (AE e)
{
}
catch (Exception e)
{
}
```

⇝ we can't declare two catch block for same exception

```
try
{ // Risky code
}
catch (AE e)
{
}
catch (AE e)
{
}
```

Scanned with CamScanner

Interview

Date / /
Page

★ ⊛ Difference b/w final, finally, finalize

⇒ final :-

- final is a modifier applicable for classes, methods & variables.
- If a class declared as final, then we can't extend that class means we can't create child class i.e inheritance is not possible
- If a method is final, then we can't override that method in child
- If a variable is final, then we can't perform reassignment for that

⇒ finally :-

- finally is a block always associated with try-catch to maintain clean-up code

```
try
{
    // Risky code
}
catch (Exception e)
{
    // Handling code
}
finally
{
    // cleanup code // Always executes
}
```

- The speciality of finally block is, it will be executed always irrespective of whether exception raised or not & whether handelled or not

⇒ finalize() :-

- finalize() is a method always invoked by Garbage Collector just before destroying an object to perform cleanup activities
- Once finalize() method completes immediately GC destroys that object.

Note: finally block is responsible to perform clean-up activities related to try block

Scanned with CamScanner

- Whatever resources we open as a part of try block will be closed inside finally block

- whereas finalize() method is responsible to perform clean-up activities related to Object

- Whatever resources associated with Object will be de-allocated before destroying Object by using finalize() method

⊛ various possible combinations of try-catch-finally

↝ In try-catch-finally order is important

↝ whenever we are writing try compulsory we should write either catch or finally, otherwise we will get C.E. i.e try without catch / finally is invalid.

↝ whenever we are writing catch block try block must be required. i.e catch without try is invalid

↝ whenever we are writing finally block, compulsory we should write try block, i.e finally without try is invalid.

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

★ ↝ for try-catch-finally blocks curly braces are mandatory.

①
```
try
{
}
catch(X e)
{
}
```
✓

②
```
try
{
}
catch(X e)
{
}
catch(Y e)
{
}
```
✓

③
```
try
{
}
catch(X e)
{
}
catch(X e)
{
}
```
C.E Exception X has already been caught

④
```
try
{
}
catch(X e)
{
}
finally
{
}
```
✓

⑤
```
try
{
}
finally
{
}
```
✓

⑥
```
try
{
}
catch(X e)
{
}
try
{
}
catch(Y e)
{
}
```
✓

Scanned with CamScanner

Date / /
Page

(7)
```
try
{ }
catch (X e)
{ }
try
{ }
finally
{ }
```
✓

(8)
```
try
{ }
```
C.E: try without catch (or) finally

(9)
```
catch (X e)
{ }
```
C.E: Catch without try

(10)
```
finally
{ }
```
C.E: finally without try

(11)
```
try
{ }
finally
{ }
catch (X e)
{ }
```
C.E: catch without try

(12)
```
try
{ }
S.O.P("Hi");
catch (X e)
{ }
```
C.E: try without catch or finally
C.E2: Catch without try

(13)
```
try
{ }
catch (X e)
{ }
S.O.P("HI");
catch (Y e)
{ }
```
C.E: catch without try

(14)
```
try
{ }
catch (X e)
{ }
S.O.P("Hi");
finally
{ }
```
C.E: finally without try

(15)
```
try
{
  try
  { }
  catch (X e)
  { }
}
catch (X e)
{ }
```
✓

(16)
```
try
{
  try
  { }
  catch (X e)
  { }
}
```
C.E: try without catch or finally

(17)
```
try
{
  try
  { }
  finally
  { }
}
catch (X e)
{ }
```
✓

(18)
```
try
{ }
catch (X e)
{
  try
  { }
  finally
  { }
}
```
✓

(19)
```
try
{ }
catch (X e)
{
  finally
  { }
}
```
C.E: finally without try

(20)
```
try
{ }
catch (X e)
{ }
finally
{
  try { }
  catch (X e)
  { }
}
```
✓

(21)
```
try { }
catch (X e)
{ }
finally { }
finally { }
```
C.E: finally without try

(22)
```
try { }
catch (X e)
{ }
finally
{
  finally { }
}
```
C.E: finally without try

Scanned with CamScanner

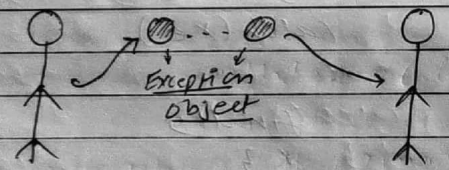
(23)
```
try
  s.o.p("try");
catch(X e)
{
  s.o.p("catch");
}
finally
{
}
```
✗

(24)
```
try{}
catch(X e)
  s.o.p("c");
finally
{}
```
✗

(25)
```
try
{}
catch(X e)
{}
finally
s.o.p("f");
```
✗

✱ throw keyword

→



↝ sometimes, we can create exception object explicitly & we can handover to the JVM manually, For this we have to use throw keyword

throw n.

```
throw new AE("/ by zero");
```

Hand-over our created exception object to the JVM manually

Creation of ArithmeticException object Explicitly

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

↝ Hence the result of following two program is exactly same.

```
class Test
{
  p s v m(sa args)
  {
    s.o.p(10/0)
  }
}
```

```
Exception in thread "main": j.l.AE: / by zero
    at Test.main()
```

```
class Test
{
  p s v m(sa args)
  {
    throw
    new AE("/ by zero");
  }
}
```

Scanned with CamScanner

In this case main() method is responsible to create exception object & handover to JVM

In this case, programmer creating exception object explicitly & handover to JVM manually

→ Note: Best use of throw keyword is for user defined Exceptions or customized Exceptions

Ex: At ATM

```
withdraw(double amount)
{
    if (amount > balance)
    {
        throw new
            InsufficientBalException();
    }
}
```

case 1: throw e;

If e refers null, then we will get NullPointerException

```
class Test
{
    static AE e=
        new AE();

    p s v m(Str args)
    {
        throw e;
    }
}
```

RE: AE

```
class Test
{
    static AE e;

    p s v m(Str args)
    {
        throw e;
    }
}
```

RE: NPE

case 2: After throw statement, we are not allowed to write any stmt directly, otherwise we will get C.E.

```
class Test
{
    p s v m(Str args)
    {
        s.o.p(10/0);

        s.o.p("Hello");
    }
}
```

RE: AE: / by zero

```
class Test
{
    p s v m(Str args)
    {
        throw new
            AE("/ by zero");

        s.o.p("Hello");
    }
}
```

C.E. unreachable statement

Scanned with CamScanner

Date / /
Page

case 3 : We can use throw keyword only for Throwable types, if we are trying to use for Normal Java Object, we will get compile time error C.E.

```
class Test
{
    p s v m(Sh args)
    {
        throw
            new TestH();
    }
}
```

C.E Incompatible types found: Test req: j.l. Throwable

```
class Test
extends RuntimeExceptn
{
    p s v m(Sh args)
    {
        throw
            new TestH();
    }
}
```

R.E Exception in thread "main": Test at Test.main();

⊛ throws Keyword

⇒ In our program, if there is a possibility of rising checked excp then compulsory we should handle that excp., or we will get C.E.

Ex.1

```
class Test
{
    p s v m(Sh args)
    {
        PrintWriter pw =
            new PrintWriter("abc.txt");
        pw.println("Hello");
    }
}
```

C.E. Unreported exception j.io.FNFException must be caught or declared to be thrown

Ex.2

```
class Test
{
    p s v m (Stn args)
    {
        Thread.sleep(10000);
    }
}
```

C.E. Unreported exception j.l.InterruptedExcp must be caught or declared to be thrown

⇒ we can handle this C.E, By using the following two way.

Scanned with CamScanner

① By using try-catch

```
try
{
    Thread.sleep(10000);
}
catch (InterruptedException e)
{
}
```

② By using throws keyword

⇒ We can use throws keyword to delegate (hand-over) the responsibility of exception handling to the caller, (It may be another method or JVM), then caller method is responsible to handle that exception

```
class Test
{
    p s v m(Sta args) throws
            InterruptedException
    {
        Thread.sleep(10000);
    }
}
```

⇒ Conclusions:

① throws keyword required only for checked Exception & usage of throws keyword for unchecked Exception is meaningless (makes no impact).

② throws keyword required only to convience compiler & usage of throws keyword doesn't prevent abnormal termination of program.

Ex:

```
class Test
{
    p s v m(Sta args) throws IE
    {
        doStuff();
    }
    p s v doStuff() throws IE
    {
        doMoreStuff();
    }
    p s v doMoreStuff() throws IE
    {
        Thread.sleep(10000);
    }
}
```

Scanned with CamScanner

In the above program if we remove atleast one throws statement then the won't compile

> Note : It is recommended to use try-catch over throws keyword

Case-1 :- We can use throws keyword for methods & constructors but not for classes

```
class Test throws Exception (X)
{
    Test() throws Exception (✓)
    {
    }
    p s v m() 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 C.E.

```
class Test
{
    p s v m() throws Test
    {
    }
}
```

C.E Incompatible types
found : Test
required : j.l.Throwable

```
class Test extends RuntimeException
{
    p s v m() throws Test
    {
    }
}
```

(✓)

Case-3 :-

```
class Test
{
    p s v m(String[] args)
    {
        throw new Exception();
    }
}
```

(checked)

C.E unreported exception j.l.Exception; must be caught or declared to be thrown

Scanned with CamScanner

```
class Test
{
    p s v m (Str args)
    {
        throw new Error();
    }
}
```
(unchecked)

R.E :- Exception in thread "main" : j.l.Error at Test.main()

Case 4 :- In our program, within the try block, if there is no chance of rising the exception then we can't write catch block for that exception, otherwise we will get C.E.

★

Exception xxx is never thrown in body of corresponding try statement

But this Rule is applicable for only Fully checked exceptions.

↓

Bcz, compiler only takes care of them fully on compile time & don't care about unchecked

```
class Test
{
    p s v m (Str args)
    {
        try
        {
            S.O.p("Hello");
        }
        catch(AE e)
        {
        }
    }
}
```
(unchecked)

O/P :- Hello

```
class Test
{
    p s v m (Str args)
    {
        try
        {
            S.O.p("Hello");
        }
        catch(Exception e)
        {
        }
    }
}
```
partially checked

O/P :- Hello

```
class Test
{
    p s v m (Str args)
    {
        try
        {
            S.O.p("Hello");
        }
        catch(IOException e)
        {
        }
    }
}
```
Fully checked

C.E: exception j.io.IOException is never thrown in body of corresponding try statement

```
class Test
{
    p s v m (Str args)
    {
        try
        {
            S.O.p("Hi");
        }
        catch(InterruptedException e)
        {
        }
    }
}
```
Fully checked

Scanned with CamScanner

```
class Test
{
    p s v m(String[] args)
    {
        try
        {
            s.o.p("H");
        }
        catch (Error e)
        {
        }
    }
}
```

unchecked

o/p : H

Exception handling keywords summary

- **try** → To maintain Risky code
- **catch** → To maintain exception handling code
- **finally** → To maintain cleanup code
- **throw** → To handover our created exception object to JVM manually
- **throws** → To delegate responsibilities of exception handling to caller.

Compile Time Errors in Exception Handling

1. Unreported exception xxx; must be caught or declared to be thrown
2. Exception xxx has already been caught
3. Exception xxx is never thrown in body of corresponding try
4. unreachable statement
5. incompatible types d:Test P:j.l.Throwable
6. try without catch or finally
7. catch without try
8. finally without try

Customized or User-defined Exception

→ Sometimes to meet programming requirement we can define our own exceptions, such exceptions are called customized / UD Exception.

Ex: TooYoungException
TooOldException

Scanned with CamScanner

InsufficientBalanceException

Ex.

Defining customized Exception

```
class TooYoungException extends
                    RuntimeException
{
    TooYoungException(String s)
    {
        super(s);        // To make description available to D-E.H
    }
}
class TooOldException extends
                    RuntimeException
{
    TooOldException(String s)
    {
        super(s);
    }
}
```

Using customized Exp

```
class CustExceptionDemo
{
    p s v m (String args[])
    {
        int age = Integer.
                parseInt(args[0]);
        if (age > 60)
        {
            throw new TooYoungException
("Too young to marriage");
        }
        else if (age < 18)
        {
            throw new TooOldException
("Too old to marriage");
        }
        else
        {
            S.O.P("You will get best match
                    by email !!");
        }
    }
}
```

Note: throw keyword is best suitable for UD / customized exception but not for predefined exception

Note: It is highly recommended to defined customized Exception as checked i.e we have to extends RuntimeException but not Exception or Throwable.

✱ Top-10 Exceptions in Java

→ Based on the person who is rising an exception, all exceptions are devided into two categories

Scanned with CamScanner

(1.) JVM Exceptions
(2.) programatic Exceptions

⇒ JVM Exceptions :

- The exceptions which are risedd automatically by JVM whenever a particular event occurs are called JVM exceptions.

Ex. AE,
NPE,
etc...

⇒ Programmatic Exceptions

- The exceptions which are risedd explicitly either by programmer / API developer (using throw keyword) to indicate that something goer wrong are called programmatic Exception

Ex. TooOldException,
IllegalArgumentException,
etc...

(1) ⇒ ArrayIndexOutOfBoundException

child of : RuntimeException
Type : Unchecked
Risedd By : JVM (Automatically)
Occurs : whenever we are trying to access array element with out of range index

Ex:

```
int[] n = new int[4];
s.o.p(n[0]);          ✓
s.o.p(n[4]);          } R.E: AIOBE
s.o.p(n[-1]);
```

(2) NullPointerException

child of : RuntimeException
Type : Unchecked
Risedd By : JVM (Auto->
Occurs : Whenever we are trying to perform any operation on n

Ex:

```
String s = null;
s.o.p(s.length());   R.E: NPE
```

(3) ClassCastException

child of : RuntimeException
Type : Unchecked
Risedd By : JVM (Auto..)

REDMI NOTE 8 AI QUAD CAMERA
Scanned with CamScanner

Date __/__/____
Page _____

Occurs : whenever we are trying to type-cast parent object to child object.

Ex:

- String s = new String("durga");
  Object o = (Object)s;

- Object o = new Object();
  String s = (String)o;

R.E. ClassCastException

(passes two compiler checks & fails runtime check)

- Object o = new String("durga");
  String s = (String)o;

④ StackOverflowError

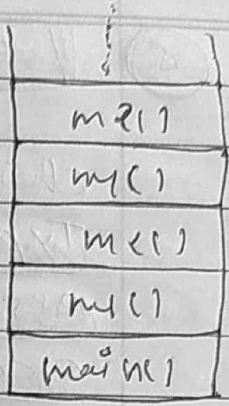
Child of : Error
Type : Unchecked
Raised By : JVM (Auto...)
Occurs : Whenever we are trying to perform recursive method call

Scanned with CamScanner

Date __/__/____
Page _____

Ex:

```
class Test
{
    p s v m1()
    {
        m2();
    }
    p s v m2()
    {
        m1();
    }
    p s v m(String args[])
    {
        m1();
    }
}
```



R.E. SOFE VIOR

(5) NoClassDefFoundError

Child of : Error
Type : Unchecked
Risedd By : JVM (Auto-)
Occurs : Whenever JVM unable to find requiredd .class file

Ex.

Java Test
(if Test.class not available then)

R.E. NoClassDefFoundError: Test

Scanned with CamScanner

Date / /
Page

(6) ExceptionInInitializerError

child of : Error
Type : Unchecked
Rised By : JVM (Auto.-)
occur : * If any Exception occurs while executing static variable assignment & static blocks

Ex:

```
class Test
{
  static int x = 10/0;
}
```

R.E. EIEError
caused by:
j.l.AE: / by zero

```
class Test
{
  static
  {
    String s=null;
    s.op(s.length());
  }
}
```

R.E. EIEError
Caused by:
j.l.NPE

(7) IllegalArgumentException

child of : RuntimeException
Type : Unchecked
Rised By : Programmer/API Deve (Explicitly)
occur : when method invocation is done with argument that is illegal.

Ex: The valid range of thread priorities is 1 to 10, if we trying to set a priority with any other value, then we will get R.E saying IllegalArgument

```
Thread t = new Thread();
t.setPriority(7);   (✓)
t.setPriority(15);  (✗)
```

R.E IllegalArgumentException

(8) NumberFormatException

child of : Direct(RE), Direct(IllegalAE)
Type : Unchecked
Rised By : Programmer/API Deve. (Explicitly)
Occur : When trying to convert string to number & string is not properly formatted.

RuntimeException
- IllegalArgumentEx.
  - NumberFormatEx.

Ex.

```
int i = Integer.parseInt("10");   (✓)
int i = Integer.parseInt("ten");
```

R.E. NumberFormatException

Scanned with CamScanner

Date / /
Page

(9) IllegalStateException

Child of : RuntimeException
Type : Unchecked
Raised By : Programmer / API Deve. (Explicitly)
occur : when method get invoked at runtime.

Ex: After starting of a thread, we are not allowed to restart the same thread once again, otherwise we will get R.E IllegalThreadStateEx.

```
Thread t = new Thread();
t.start(); (1)
:
t.start();
```

→ R.E : IllegalThreadStateException

(10) AssertionError

Child of : Error
Type : Unchecked
Raised By : Programmer/API Deve. (Explicitly)
occur : when Assert stmt fails.

Ex: assert (x>10); If x is not g.t 10 we will get

→ R.E AssertionError

Exception / Error	Raised By
(1) ArrayIndexOutOfBoundException	Raised by JVM (Automatically) and hence they are JVM Exception
(2) NullPointerException	
(3) ClassCastException	
(4) StackOverflowError	
(5) NoClassDefFoundError	
(6) ExceptionInInitializerError	
(7) IllegalArgumentException	Raised by programmer / API Deve. Explicitly & hence these are Programmatic Exception
(8) NumberFormatException	
(9) IllegalStateException	
(10) AssertionError	

✱ 1.7 V Enhancement in Exception handling.

→ As a part of 1.7 V in exception handling the following two concept introduced
(1.) try with resources
(2.) Multi-catch block

Scanned with CamScanner

Date / /
Page

⇒ try-with-Resources

→ Until 1.6V, it is highly recommended to write finally block to close resources which are open as a part of try block.

```
BR br = null;
try
{
    br = new BR(new FR("ip.txt"));

    // use br according to our requirement
}
catch(IOException e)
{
    // handling code
}
finally
{
    if(br != null)
    {
        br.close();
    }
}
```

→ The problems in this approach are:-

(i) Compulsory programmer is required to close the resources inside finally block, it increases complexity of programming.

(ii) We have to write finally block compulsory, hence it increases length of code & reduces readability.

→ To overcome these problems, some people introduce try-with-resources in 1.7V

→ The main advantage of try-with-resources is whatever resources we open as a part of try block will be closed automatically once control reaches end of try block either normally or abnormally & hence we are not required to closed explicitly & hence complexity of program will be reduced.

→ We are not required to write finally & hence length of code is reduced & readability will be increased.

Ex:

```
try(BR br = new BR(new FR("ip.txt")))
{
    // use br according to requirement
```

REDMI NOTE 8 AI QUAD CAMERA

Scanned with CamScanner

```
}
catch (IOException e)
{
    // handling code
}
```

⇒ Conclusions

(1) we can declare multiple resources, but there resources should be separated with **semi-colon (;)**

```
try (R1; R2; R3)
{
}
```

```
try (FileWriter fw = new FileWriter("op.txt");
     FileReader fr = new FileReader("ip.txt"))
```

(2) All resources should be **AutoCloseable** resources.

- A resource is said to be AutoCloseable if and only if corresponding class implements **java.lang.AutoCloseable (interface)**

- All IO related resources, DB related, NW related are already implements resource AutoCloseable interface

~ Being a programmer, we are not required to do anything just we should aware the point

~ AutoCloseable interface came in 1.7 v & it contains only 1 method

```
public void close();
```

(3) All resource reference variables are implicitly **final** & Hence within a try block we can't perform assignment otherwise we will get **C.E.**

Ex-
```
class TryWithRes
{
    psvm()
    {
        try(BR br = new
            BR(new FR("ip.txt")))
        {
            br = new BR
                 (new FR("op.txt"));
        }
    }
}
```

Scanned with CamScanner

C.E: Auto-closeable resource br may not be assigned

(4.) Untill 1.6V try should be associated with either catch or finally, but from 1.7V onwards we can take only try-with-resource without catch or finally

1.6V	1.7V onwards

try
{
}
finally
{
}
	try (R)
	{
	}

→ The main Advantage of try-with-resource is, we are not required to write finally block explicitly. bcz, we are not required to close explicitly.

So, untill 1.6V finally is hero but from 1.7V it is just dummy means zero.

→ Multi-catch block

→ Untill 1.6V even though, multiple different exceptions having same handling code, for every exception type we have to write a separate catch block which increases length of code & reduces readability.

Ex:
```
try
{
  --
}
catch (AE e)
{
    e.printStackTrace();
}
catch (IOException e)
{
    e.printStackTrace();
}
catch (NPE e)
{
  s.o.p(e.getMessage());
}
catch (InterruptedException e)
{
  s.o.p(e.getMessage());
}
```

Scanned with CamScanner

Date / /
Page

⇒ To overcome this problem, some people introduced multi-catch block in 1.7v.

⇒ According to this, we can write a single catch block that can handle different type of exception.

Ex. try
{
==
}
catch (AE | NPE Exception e)
{
e.printStackTrace();
}
catch (NPE | InterruptedEx e)
{
s.o.p(e.getMessage());
}

⇒ The main Advantage of this approach is length of code is reduced & readability will be improved.

Ex.
class MultiCatchBlock
{
p s v m(Str[])
{
try
{
s.o.p(10/0);
String s = null;
s.o.p(s.length());
}
catch (ArithmeticException | NullPointerException e)
{
s.o.p(e);
}
}
}

O/P : j.l.AE : / by zero

⇒ In multi-catch block, there should not be any relation b/w exception types (either child to parent / parent to child / same type) otherwise we will get C.E.

Ex. try
{
==
}

Scanned with CamScanner

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

C.E: Alternative in a multi-catch statement cannot be related by subclass.

✱ Exception Propagation

→ Inside a method if an exception raised & if we are not handling that excp, then excp object will be propagated to caller, then caller method is responsible to handle excp.

This process is called "Exception propagation."

✱ Rethrowing Exception

→ We can use this approach to convert one exception type to another exception type.

Ex:

```
try
{
    s.o.p(10/0);
}
catch(AE e)
{
    throw new NullPointerException();
}
```

End of Exception Handling

Start of Collections

Scanned with CamScanner