6. Exception Handling

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. 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 enhancement with w.r.t. Exception handling
    - i) try with resources
    - ii) Multi-catch block.

1. Introduction:—

Exception:— An unwanted, unexpected event that disturbs normal flow of program is called Exception.

Ex: SleepingException, TyrePunchuredException, FileNotFoundException etc.

→ It is highly recommended to handle Exceptions.

→ The main objective of Exception Handling is graceful termination of the program.

*** Q: What is the meaning of Exception Handling?

Ans: Exception Handling doesn't mean repair an exception. we have to define an alternative way to continue the rest of the program normally is called Exception Handling.

→ For example, if our programming requirement is to read data from remote file locating at London.

At runtime, if London file is not available then the program should be terminated abnormally.

We have to provide a local file to continue rest of the program normally this way of defining an alternative is nothing but Exception Handling.

Ex:
```
try
{
  Read data from
  London file
}
catch(FileNotFoundException e)
{
  Use local file to continue
  rest of the program normally
}
```

2. Runtime Stack Mechanism:—

→ For every thread JVM will create a runtime stack.

→ All method calls performed by that thread will be stored in the corresponding stack.

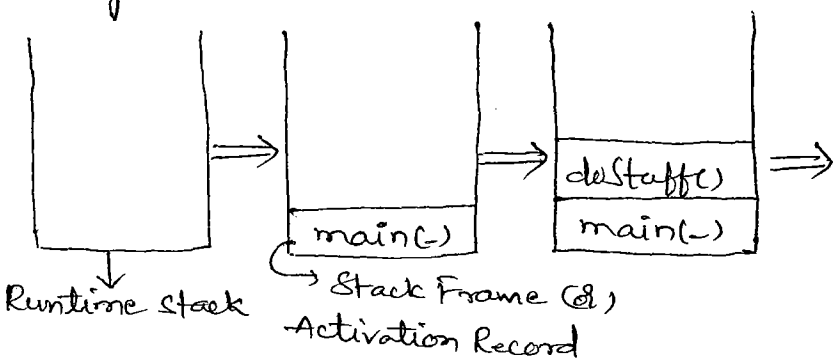
→ Each entry in the stack is called Activation Record or Stack Frame.

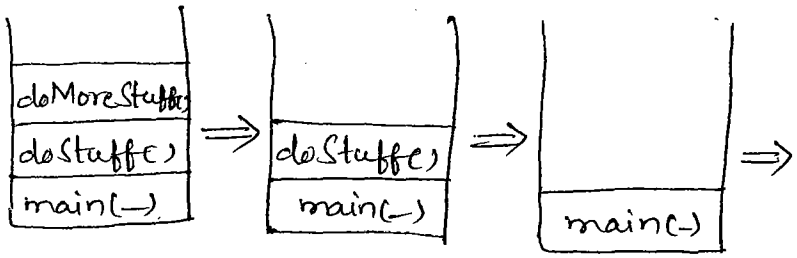
→ After completing every method call JVM removes the corresponding entry from the stack.

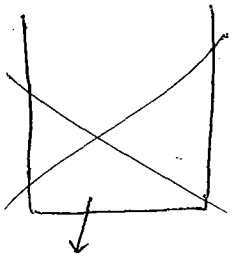
→ After completing all method calls just before terminating thread JVM destroys the corresponding stack.

Ex:
```
class Test
{
  P S V m(-)
  {
    doStuff();
  }
  P S V doStuff()
  {
    doMoreStuff();
  }
  P S V doMoreStuff()
  {
    S.o.p("Hello");
  }
}
```

O/P : Hello.







JVM will destroy this empty stack

3. Default Exception Handling in Java :—

→ In our program, if any where an Exception raised the method in which it is raised is responsible to create Exception object by including the following information.

1) Name of the Exception.
2) Description
3) Location (Stack Trace)

→ After creating Exception object the Method handovers that object to the JVM.

→ JVM will check whether the corresponding method having any Exception handling code or not.

→ If the method having any Exception handling code then it will be executed, o.w. JVM terminates that method abnormally & removes corresponding from the stack.

→ JVM identifies Caller method and will check whether caller method contains handling code or not. If the caller method doesn't contain Exception handling code then JVM terminates caller method also abnormally & removes corresponding entry from the stack.

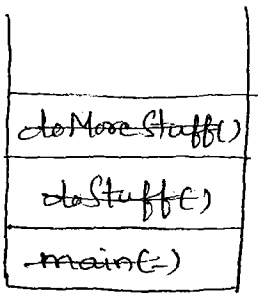
→ This process will be continued until main(-) method & if the main(-) method also doesn't contain Exception handling code then JVM terminates main(-) method also abnormally & removes corresponding entry from the stack.

→ Then JVM handovers Exception object to the Default Exception Handler & it is part of JVM.

→ Default Exception Handler just print Exception information to the console in the following format & terminates program abnormally.

Name of Exception : Description
Stack Trace

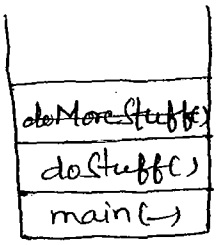
Ex:-
```
class Test
{
  p s v main(-)
  {
    doStuff();
  }
  p s v doStuff()
  {
    doMoreStuff();
  }
  p s v doMoreStuff()
  {
    S.o.p(10/0); → RE
  }
}
```



Runtime Stack.

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

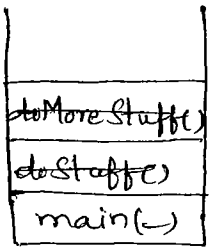
Ex ①:
```
class Test
{
  p s v main(-)
  {
    doStuff();
  }
  p s v doStuff()
  {
    doMoreStuff();
    S.o.p(10/0);
  }
  p s v doMoreStuff()
  {
    S.o.p("Hello");
  }
}
```



Runtime Stack

o/p: Hello
Exception in thread "main" : j.l.AE : / by zero
at Test.doStuff()
at Test.main(-)

Eg②:
```
class Test
{
  p s v main(-)
  {
    doStuff();
    S.o.p(10/0);
  }
  p s v doStuff()
  {
    doMoreStuff();
    S.o.p("Hi");
  }
  p s v doMoreStuff()
  {
    S.o.p("Hello");
  }
}
```



O/P: Hello
Hi
Exception in thread "main": j.l.AE: / by zero
at Test.main()

Note:- In our program, if atleast one method terminated abnormally then the program termination is Abnormal termination.

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

4. Exception Hierarchy :-

→ Throwable class acts as a root for Exception hierarchy

→ Throwable class contains 2 child classes

1) Exception
2) Error.

1) Exception :- Most of the cases Exceptions are caused by our program & these are recoverable.

For Example, if our programming requirement is to read data from London file. At runtime if London file is not available then we will get RE saying FileNotFoundException

If FileNotFoundException occurs we can provide a local file to continue rest of the program normally.

2. Error :-

→ Most of the cases Errors are not caused by our program & these are due to lack of system resources.

→ Errors are non-recoverable.

For Example, if OutOfMemoryError occurs being a programmer we can't do anything then the program will be terminated abnormally.

System or Server Admin is responsible to increase Heap memory.

Checked Vs Unchecked Exception :-

Checked Exception :-

→ The Exceptions which are checked by compiler for smooth execution of the program at runtime are called checked Exceptions.

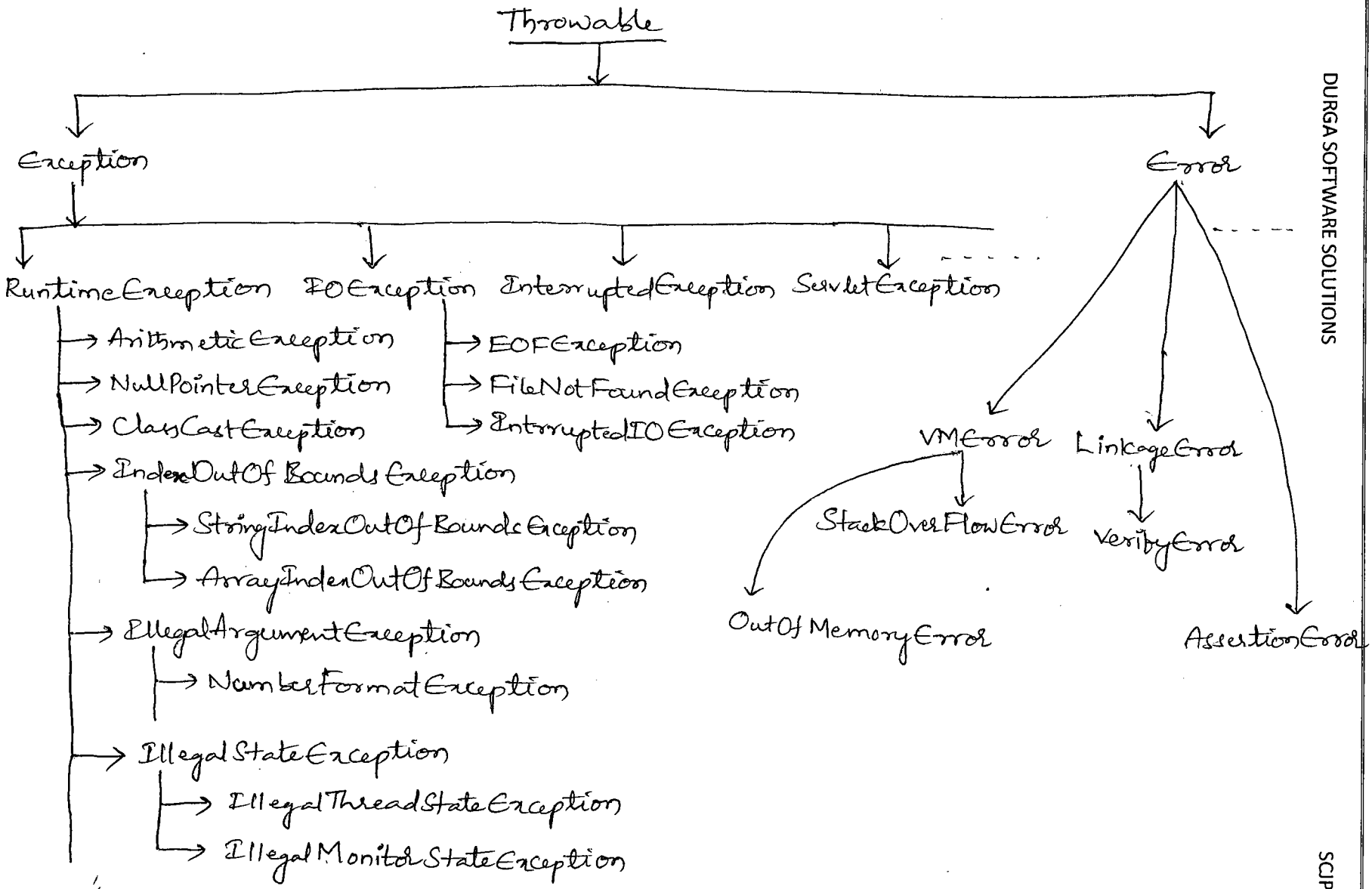
Ex: HallTicketMissingException, PenNotWorkingException, InsufficientDinnerException, FileNotFoundException etc.

→ Compiler will check whether we are handling checked Exception or not.

→ If we are not handling then we will get Compile time error.

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

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



PLOT NO.56, HARSHA CHAMBERS, SAI NAGAR,NEAR BIG C MOBILES, MADHAPUR-500034, MOBILE:9505905786

Unchecked Exception:—

→ The Exceptions which are not checked by compiler whether the programmer handling or not are called Unchecked Exceptions.

Ex:- BombBlastException, ShortCircuitException, ArithmeticException, NullPointerException etc.

*** Note①: Whether Exception is checked or unchecked compulsory every Exception should occurs at runtime only & there is no chance of occurring any Exception at compile time.

*** ② RuntimeException and its child classes, Error and its child classes are unchecked. Except these the remaining are checked Exceptions.

Fully checked & Partially checked Exceptions:—

Fully checked Exceptions:—

→ A checked Exception is said to be fully checked iff all its child classes also checked.

Ex:- IOException, InterruptedException etc.

Partially Checked Exceptions:—

→ A checked Exception is said to be partially checked iff some of its child classes are unchecked.

Ex: Exception, Throwable.

Note:— The only available partially checked Exceptions in Java are 1) Throwable
2) Exception

Q: Describe the behaviour of following Exceptions?

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

5. Customized Exception Handling by using try-catch :-

→ It is highly recommended to handle exceptions.

→ The code which may raise Exception is called Risky code, we have to place risky code inside try block and the corresponding handling code we have to place inside catch block.

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

Without try-catch

```
class Test
{
  p s v m(-)
  {
    S.o.p("Stmt1");
    S.o.p(10/0);
    S.o.p("Stmt3");
  }
}
```

With try-catch

```
class Test
{
  p s v m(-)
  {
    S.o.p("Stmt1");
```

o/p : stmt1
Exception in thread "main" : j.l.AE : / by zero
at Test.main()

Abnormal termination

```
try
{
 S.o.p(10/0);
}
catch(AE e)
{
 S.o.p(10/2);
}
S.o.p("stmt3");
}}
```

o/p : stmt1
5
stmt3

Normal termination

6. Control Flow in try-catch :—

```
try
{
 stmt1;
 stmt2;
 stmt3;
}
catch(X e)
{
  stmt4;
}
 statement5;
```

Case(i):

→ If there is no Exception.

1, 2, 3, 5, Normal Termination.

Case(ii): If an Exception raised at stmt2 and the corresponding catch block matched.

1, 4, 5, NT

Case(iii): If an Exception raised at stmt2 and the corresponding catch block not matched.

1, Abnormal Termination

Case(iv): If an Exception raised at stmt4 (or) stmt5 it is always abnormal termination.

Note: ① Within the try block if any where an Exception raised rest of the try block won't be executed eventhough we handled that Exception.

Hence length of try block should as less as possible and we have to take only risky code within the try block, but not normal Java code.

② In addition to try block there may be a chance of raising Exception inside catch & finally blocks also.

③ If any statement raises an Exception & if it is not part of try block then it is always Abnormal termination of the program.

7. Methods to print Exception information:—

→ Throwable class defines the following methods to print Exception information.

Method	Printable Format
1. printStackTrace()	Name of Exception : Description Stack Trace
2. toString()	Name of Exception : Description
3. getMessage()	Description.

Ex: class Test
{
p s v m(-)
{

```
try
{
  S.o.p (10/0);
}
catch(AE e)
{
  e.printStackTrace();
  S.o.p(e); ⇒ e.toString();
  S.o.p(e.getMessage());
}
}
```

j.l.AE : / by zero
at Test.main()

j.l.AE : / by zero

/ by zero

***Note:- Default Exception Handler always print Exception information by using printStackTrace() method.

***8. try with multiple catch blocks:—

→ The way of handling an Exception is varied from Exception to Exception.

→ Hence for every Exception type is recommended to take separate catch block.

→ Hence try with multiple catch blocks is possible & recommended to use.

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

Not Recommended

```
try
{
  ≡
}
catch(AE e)
{
  perform these alternative arithmetic operations
}
catch(FileNotFoundException e)
{
  Use local file instead of remote file
}
catch(SQLException e)
{
  Use MySQL db instead of Oracle db
}
```

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

Highly Recommended

***

→ If try with multiple catch blocks present then the order of catch blocks is very important & it should be from child to parent. By mistake if we are taking from parent to child then we will get CE saying, exception xxx has already been caught.

Ex:

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

✗

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

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

✓

→ If we are trying to take multiple catch blocks for same Exception then we will get CE.

Ex:

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

✗

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

9. finally block:-

→ It is not recommended to maintain clean up code inside try block becoz there is no guarantee for the execution of every statement inside try block always.

→ It is never recommended to maintain clean up code inside catch block becoz if there is no Exception then catch block won't be executed.

→ We required some place to maintain clean up code which should be executed always irrespective of whether Exception raised or not, whether handled or not handled such type of best place is nothing but finally block.

→ Hence the main purpose of finally block is to maintain clean up code.

Ex:

```
try
{
  Risky code
}
catch(X e)
{
  Handling code
}
finally
{
  Clean up code
}
```

→ The speciality of finally block is it will be executed always irrespective of whether Exception raised or not and handled or not handled.

Ex:
```
class Test
{
  p s v m(-)
  {
    try
    {
     S.o.p("try");
    }
    catch(Exception e)
    {
     S.o.p("catch");
    }
    finally
    {
     S.o.p("finally");
    }
  }
}
```
o/p : try
finally

```
class Test
{
  p s v m(-)
  {
    try
    {
     S.o.p("try");
     S.o.p(10/0);
    }
    catch(Exception e)
    {
     S.o.p("catch");
    }
    finally
    {
     S.o.p("finally");
    }
  }
}
```
o/p : try
catch
finally

```
class Test
{
  p s v m(-)
  {
    try
    {
     S.o.p("try");
     S.o.p(10/0);
    }
    catch(NPE e)
    {
     S.o.p("catch");
    }
    finally
    {
     S.o.p("finally");
    }
  }
}
```
o/p : try
finally
RE: AE: / by zero

finally Vs return statement :—

→ Eventhough return statement present inside try and catch blocks first finally will be executed and then return statement will be considered. i.e., finally block dominates return statement.

Ex:
```
class Test
{
  p s v m(-)
  {
    try
    {
      S.o.p("try");
      return;
    }
    catch(Exception e)
    {
      S.o.p("catch");
      return;
    }
    finally
    {
      S.o.p("finally");
    }
  }
}
```
o/p : try
finally

→ If return statement inside try, catch & finally blocks then finally block return statement will be considered.

Ex:
```
class Test
{
  P S V m(-)
  {
    S.O.P(m1());
  }
  P S int m1()
  {
    try
    {
      return 777;
    }
    catch(Exception e)
    {
      return 888;
    }
    finally
    {
      return 999;
    }
  }
}
```
o/p : 999

finally Vs System.exit(0) :—

→ There is only one situation where finally block won't be executed i.e., whenever we are using System.exit(0).

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

→ Hence System.exit(0) dominates finally block.

Ex:
```
class Test
{
  P S V m(-)
  {
    try
    {
      S.O.P("try");
      System.exit(0);
    }
    catch(Exception e)
    {
      S.O.P("catch");
    }
    finally
    {
      S.O.P("finally");
    }
  }
}
```

System.exit(0);

→ The argument represents status code.

→ Instead of zero we can take any valid int value.

→ 0 means Normal Termination.

non-zero means Abnormal Termination.

→ Whether zero or non-zero there is no difference in the impact and the program will be terminated.

→ Internally JVM will use this status code

***

9. Difference blw final, finally and finalize() :-

final :-

→ final is the modifier applicable for classes, methods & variables.

→ If we declare a class as final then we can't create child class.

→ If we declare a method as final then we can't override that method in the child class.

→ If we declare a variable as final then we can't change its value because it will become constant.

finally :-

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

→ The speciality of finally block is it will be executed always irrespective of whether Exception raised or not raised and handled or not handled.

finalize() :-

→ It is a method always called by Garbage Collector just before destroying an object to perform clean up activities.

→ Once finalize() method completes automatically Garbage Collector destroys that object.

Note:- When compared with finalize() method finally block is recommended to maintain clean up code becoz we can't expect exact behaviour of Garbage Collector.

11. Control Flow in try-catch-finally :-

```
try
{
 stmt1;
 stmt2;
 stmt3;
}
catch(X e)
{
  stmt4;
}
finally
{
  stmt5;
}
 stmt6;
```

Case(i): If there is no Exception.

1, 2, 3, 5, 6, NT

Case(ii): If an Exception raised at stmt2 & corresponding catch block matched.

1, 4, 5, 6, NT

Case(iii): If an Exception raised at stmt2 & corresponding catch block not matched.

1, 5, AT

Case(iv): If an Exception raised at stmt4 then it is always AT, but before that finally block will be executed.

Case(v): If an Exception raised at stmt5 or stmt6 then it always AT.

12. Control Flow in nested try-catch-finally:-

```
try
{
  stmt1;
  stmt2;
  stmt3;
  try
  {
    stmt4;
    stmt5;
  }
  stmt6;
  catch(X e)
  {
    stmt7;
  }
  finally
  {
    stmt8;
  }
  stmt9;
}
catch(X e)
{
  stmt10;
}
finally
{
  stmt11;
}
stmt12;
```

Case(i): If there is no Exception. then [1, 2, 3, 4, 5, 6, 8, 9, 11, 12, NT]

Case(ii): If an Exception raised at stmt2 & corresponding catch block matched.

[1, 10, 11, 12, NT]

Case(iii): If an Exception raised at stmt2 & corresponding catch block not matched.

[1, 11, AT]

Case(iv): If an Exception raised at stmt5 & corresponding inner catch block matched.

[1, 2, 3, 4, 7, 8, 9, 11, 12, NT]

Case(v): If an Exception raised at stmt5 & corresponding inner catch block not matched, but out catch block matched.

[1, 2, 3, 4, 8, 10, 11, 12, NT]

Case(vi): If an Exception raised at stmt5 & both inner & outer catch blocks are not matched.

1,2,3,4,8,11,AT

Case(vii): If an Exception raised at stmt7 & corresponding catch block matched.

1,2,3,•,•,•,8,10,11,12,NT

Case(viii): If an Exception raised at stmt7 & corresponding catch block not matched.

1,2,3,•,•,•,8,11,AT

Case(ix): If an Exception raised at stmt6 & corresponding catch block matched.

1,2,3,•,•,•,•,10,11,12,NT

Case(x): If an Exception raised at stmt6 & corresponding catch block not matched.

1,2,3,•,•,•,•,11,AT

Case(xi): If an Exception raised at stmt9 & corresponding catch block matched.

1,2,3,•,•,•,•,8,10,11,12,NT

Case(xii): If an Exception raised at stmt9 & corresponding catch block not matched.

1,2,3,•,•,•,•,8,11,AT

Case(xiii): If an Exception raised at stmt10. then it is always AT but before that finally block will be executed.

Case(xiv): If an Exception raised at stmt11 or stmt12 then it is always AT.

Ex:
```
class Test
{
  p s v m(-)
  {
    try
    {
      S.o.p(10/0);
    }
    catch(AE e)
    {
      S.o.p(10/0);
    }
    finally
    {
      String s=null;
      S.o.p(s.length());
    }
  }
}
```

① RE: AE :/ by zero
② RE: NPE
③ RE: AE & NPE
④ CE

Note:- Default Exception Handler can handle only one Exception at a time which is the most recently raised Exception.

Various possible combinations of try-catch-finally :-

→ We can take try-catch-finally inside try, catch and finally blocks i.e., nesting of try-catch-finally is possible.

→ Whenever we are taking try compulsory we have to write either catch or finally i.e., try without catch or finally is invalid.

→ Whenever we are writing catch compulsory try block should be required i.e., catch without try is always invalid.

→ Whenever we are writing finally block compulsory we should write try i.e., finally without try is always invalid.

→ In try-catch-finally order is important.

→ For try, catch & finally blocks curely braces are mandatory

→ Once we entered into the try block without executing finally block we can't go out.

→ If we are not entering into the try block then corresponding finally block won't be executed.

```
try
{
}
catch(X e)
{
}
```
✓

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

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

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

```
try
{
}
```
CE: try with out catch or finally

```
catch(X e)
{
}
```
CE: catch with out try

```
finally
{
}
```
CE: finally with out try

```
try
{
}
finally
{
}
```
✓

```
try
{
}
catch(X e)
{
}
S.o.p("Hello");
catch(X e)
{
}
```
CE: try with out catch or finally
CE: catch with out try

```
try
{
}
catch(X e)
{
}
S.o.p("Hello");
catch(Y e)
{
}
```
CE: catch with out try

```
try
{
}
catch(X e)
{
}
S.o.p("Hello");
finally
{
}
```
CE: finally with out try

```
try
{
}
finally
{
}
catch(X e)
{
}
```
CE: catch with out try

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

```
try
{
}
catch(X e)
{
}
finally
{
}
finally
{
}
```
CE: finally with out try

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

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

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

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

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

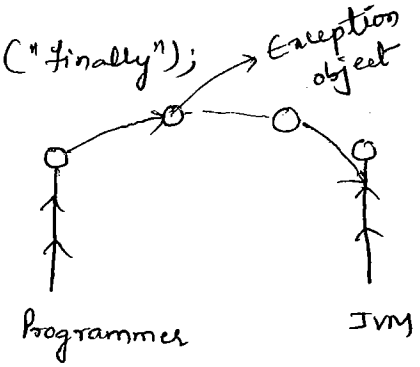
```
try
  S.o.p("Hello");
catch(X e)
{
}
```
✗

```
try
{
}
catch(X e)
  S.o.p("Hello");
```
✗

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

13. throw keyword:—

→ Sometimes we can create Exception object explicitly and we can handover that object to the JVM manually, for this we have to use throw keyword.



Ex: `throw new AE("I by zero");`

- To handover our created Exception object to the JVM manually
- Creation of Exception object explicitly

→ The result of following 2 programs exactly same.

Ex:
```
class Test
{
  p s v m(-)
  {
    S.o.p(10/0);
  }
}
```

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

In this case, main(-) method is responsible to create Exception object and handover to the JVM implicitly.

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

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

In this case, programmer is responsible to create Exception object explicitly and handover to the JVM.

→ Most of the times, we can use throw keyword for customized exceptions (our own exceptions) but not for predefined Exceptions.

Ex:
```
withdraw(double amount)
{
  if(amount > balance)
  {
    throw new InSufficientFundsException();
  }
  else
    Process the request.
}
```

Case(i): `throw e;`

If 'e' refers null then we will get NPE.

Ex:
```
class Test
{
  static AE e=new AE();
  p s v m(-)
  {
    throw e;
  }
}
```
RE: AE

```
class Test
{
  static AE e;
  p s v m(-)
  {
    throw e;
  }
}
```
RE: NPE

Case(ii): After throw statement we are not allowed to write any stmt directly, o.w. we will get CE saying, Unreachable statement.

Ex:
```
class Test
{
  p s v m(-)
  {
    S.o.p(10/0);
    S.o.p("Hello");
  }
}
```
RE: AE: / by zero
at Test.main()

```
class Test
{
  p s v m(-)
  {
    throw new AE("/ by zero");
    S.o.p("Hello");
  }
}
```
CE: Unreachable statement

Case(iii): We can use throw keyword only for Throwable types, o.w. we will get CE saying, incompatible types.

Ex:
```
class Test
{
  p s v m(-)
  {
    throw new Test();
  }
}
```
CE: incompatible types
found: Test
required: j.l.Throwable

```
class Test extends RuntimeException
{
  p s v m(-)
  {
    throw new Test();
  }
}
```
RE: Exception in thread "main": Test
at Test.main()

14. throws keyword :-

→ In our programs, if there is any chance of raising checked Exception compulsory we should handle that checked Exception, o.w. we will get CE saying,

Unreported exception xxx; must be caught or declared to be thrown.

Ex①:
```
class Test
{
  P S V m(-)
  {
    PrintWriter pw=new PW("abc.txt");
    pw.println("Hello");
  }
}
```
CE: Unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown

Ex②:
```
class Test
{
  P S V m(-)
  {
    Thread.sleep(5000);
  }
}
```
CE: Unreported exception java.lang.InterruptedException;
must be caught or declared to be thrown

→ We can handle this CE by using the following 2 ways.

1) by using try-catch
2) by using throws keyword.

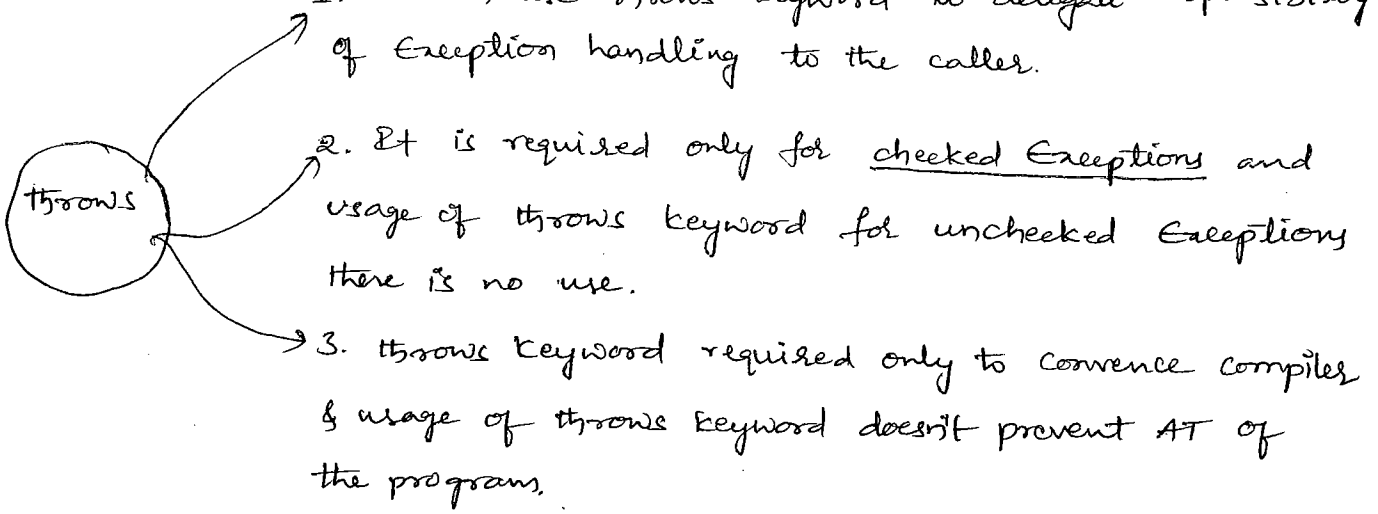
1. By using try-catch:-

Ex:
```
class Test
{
  P S V m(-)
  {
    try
    {
      Thread.sleep(5000);
    }
    catch(InterruptedException e)
    {
    }
  }
}
```
(code compiles fine).

2. By using throws keyword:-

→ We can use throws keyword to delegate responsibility of Exception Handling to the caller (It may be method or JVM) then caller is responsible to handle that checked Exception.

Ex:
```
class Test
{
  p s v m(-) throws IE
  {
    Thread.sleep(5000);
  }
}
```
(Code compiles fine)



Ex:
```
class Test
{
  p s v m(-) throws IE
  {
    doStuff();
  }
  p s v doStuff() throws IE
  {
    doMoreStuff();
  }
  p s v doMoreStuff() throws IE
  {
    Thread.sleep(5000);
  }
}
```
CE: Unreported exception j.l.IE; must be caught or declared to be thrown

(Code compiles fine)

*** → If we remove atleast One throws keyword in the above program then we will get CE i.e., all throws statements must be required.

Note: It is recommended to use try-catch over throws statement.

Case(i): We can use throws keyword for methods & constructors, but not for classes.

Ex:
```
class Test throws Exception        → CE
{
  Test() throws Exception          ✓
  {
  }
  public void m1() throws Exception
  {                                ✓
  }
}
```

Case(ii): We can use throws keyword only for Throwable types, o.w. we will get CE saying, incompatible types.

Ex:
```
class Test
{
  p s v main(-) throws Test
  {
  }
}
```
CE: incompatible types
found: Test
required: j.l.Throwable

```
class Test extends Exception
{
  p s v m(-) throws Test
  {
  }
}
```
✓

Case(iii):

Ex:
```
class Test
{
  p s v m(-)
  {
    throw new Exception();
  }
}
```
↓ checked

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

```
class Test
{
  p s v m(-)
  {
    throw new Error();
  }
}
```
↓ unchecked.

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

Case(iv):

→ In try block, if there is no chance of raising an exception then we can't write catch block for that exception, o.w. we will get CE saying,

exception XXX is never thrown in body of corresponding try statement

→ But this rule is applicable only for fully checked exceptions.

Ex:
```
class Test
{
  p s v m(-)
  {
    try
    {
      S.o.p("Hello");
    }
    catch(AE e)
    {
    }
  }
}
```
O/P: Hello

unchecked

```
class Test
{
  p s v m(-)
  {
    try
    {
      S.o.p("Hello");
    }
    catch(Exception e)
    {
    }
  }
}
```
O/P: Hello

partially checked.

```
class Test
{
  p s v m(-)
  {
    try
    {
      S.o.p("Hello");
    }
    catch(IOE e)
    {
    }
  }
}
```
CE: exception java.io.IOE is never thrown in body of corresponding try statement

fully checked

```
class Test
{
  p s v m(-)
  {
    try
    {
      S.o.p("Hello");
    }
    catch(InterruptedException e)
    {
    }
  }
}
```
CE: exception java.l.IE is never thrown in body of corresponding try statement

fully checked.

```
class Test
{
  p s v m(-)
  {
    try
    {
      S.o.p("Hello");
    }
    catch(Error e)
    {
    }
  }
}
```
O/P: Hello

unchecked.

15. Exception Handling keywords Summary :—

1. try → To maintain Risky code
2. catch → To maintain Exception handling code
3. finally → To maintain clean up code.
4. throw → To handover our created Exception object to the JVM.
5. throws → To delegate responsibility of Exception handling to the caller.

16. Various possible Compile time errors in Exception handling:—

1) exception xxx has 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) Unreachable statement
5) incompatible types
   found : Test
   required : j.l.Throwable.
6) try without catch or finally
7) catch without try.
8) finally without try.

*** 17. Customized or User defined Exceptions :—

→ Sometimes to meet programming requirements we have to define our own Exceptions such type of Exceptions are called Customized (or) User defined Exceptions.

Eg: TooYoungException, TooOldException, InSufficientFundsException etc.

Ex:
```
class TooYoungException extends RuntimeException
{
  TooYoungException(String s)
  {
    super(s);
  }
}
class TooOldException extends RuntimeException
{
   TooOldException(String s)
   {
     super(s);
   }
}
class CustExceptionDemo
{
  p s v main(String[] args)
  {
    int age=Integer.parseInt(args[0]);
    if(age>60)
    {
      throw new TooYoungException("plz wait some more time
                                   definitely U will get best match");
    }
    else if(age<18)
    {
      throw new TooOldException("Ur age already crossed marriage
                                 age-- no chance of getting marriage");
    }
    else
    {
      S.o.p("U will get match details soon by email!!!");
    }
  }
}
```

Note ①: throw keyword is best use for customized Exceptions, but not for predefined Exceptions.

② It is highly recommended to define customized Exceptions as unchecked.

i.e, our Exception class should extends RuntimeException but not Exception class.

18. Top-10 Exceptions:—

→ Based on the person who is raising Exception, all Exceptions are divided into the following 2 types.

1) JVM Exceptions
2) Programmatic Exceptions.

1) JVM Exceptions:—

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

Ex: ArithmeticException, NPE, AIOOBE, etc.

2) Programmatic Exceptions:—

→ The Exceptions which are raised explicitly either by programmer or API Developer to indicate that something goes wrong are called Programmatic Exceptions.

Ex: TooYoungException, TooOldException, IllegalArgumentException etc.

1) ArrayIndexOutOfBoundsException:—

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

Ex:
```
int[] a=new int[10];
S.o.p(a[0]); ✓
S.o.p(a[15]); → RE: AIOOBE
S.o.p(a[-15]); → RE: AIOOBE
```

2) NullPointerException :—

→ It is the child class of RuntimeException & hence it is unchecked.

→ Raised automatically by JVM whenever we are toying to perform any method call on null reference.

Ex:- String s=null;
S.o.p(s.length());→(RE:NPE)

3) ClassCastException :—

→ It is the child class of RuntimeException & hence it is unchecked.

→ Raised automatically by JVM whenever we are trying to typecast parent object to the child type.

Ex: String s=new String("durga");
Object o=(Object)s; ✓
Object o=new String("durga");
String s=(String)o; ✓

Object o=new Object();
String s=(String)o;→(RE:CCE)

4) NoClassDefFoundError :—

→ It is the child class of Error & hence it is unchecked.

→ Raised automatically by JVM whenever JVM unable to find required .class file.

Ex: java Test

If Test.class file is not available then we will get RE saying, NoClassDefFoundError : Test.

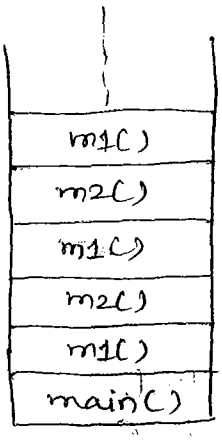
5) StackOverFlowError :—

→ It is the child class of Error & hence it is unchecked.

→ Raised automatically by JVM whenever we are toying to perform recursive method call.

Ex:
```
class Test
{
  p s v m1()
  {
    m2();
  }
  p s v m2()
  {
    m1();
  }
  p s v main()
  {
    m1();
  }
}
```

RE: StackOverFlowError



Runtime Stack

***
6) ExceptionInInitializerError :—

→ It is the child class of Error & hence it is unchecked.

→ Raised automatically by JVM whenever an Exception occurs while executing static variable assignments & static blocks.

Ex:
```
class Test
{
  static int i=10/0;
}
```

RE: Exception in thread "main": j.l.ExceptionInInitializerError caused by: j.l.AE: / by zero

```
class Test
{
  static
  {
    String s=null;
    S.o.p(s.length());
  }
}
```

RE: ExceptionInInitializerError caused by: j.l.NPE

7) IllegalArgumentException :—

→ It is the child class of RuntimeException & hence it is unchecked.

→ Raised explicitly either by programmer or API developer to indicate that a method has been invoked with illegal argument.

→ The valid range of Thread priorities is 1 to 10. If we are trying to set with any other value then we will get RE saying, IllegalArgumentException.

Ex: Thread t=new Thread();
t.setPriority(10); ✓
t.setPriority(100); → RE: IAE

8) NumberFormatException :—

→ It is the direct child class of IllegalArgumentException, which is the child class of RuntimeException & hence it is unchecked.

→ Raised explicitly either by programmer or API developer to indicate we are trying to convert String to number, but the String is not properly formatted.

Ex:- int i=Integer.parseInt("10"); ✓
int i=Integer.parseInt("ten"); → RE: NFE: "ten"

RE
↑
IAE
↑
NFE

9) IllegalStateException :—

→ It is the child class of RuntimeException & hence it is unchecked.

→ Raised explicitly either by programmer or API developer to indicate that a method has been invoked at wrong time.

Ex ①:

→ After starting a thread we are not allowed to restart the same thread again, o.w. we will get RE saying, IllegalThreadStateException.

```
Thread t=new Thread();
t.start();
  |
  |
t.start(); → RE: IllegalThreadStateException
```

Ex ②:

→ Once Session expires we are not allowed to call any method on that

Session object. If we are trying to call any method then we will get RE saying, IllegalStateException.

```
HttpSession session=req.getSession();
S.o.p(session.getId());
session.invalidate();
S.o.p(session.getId()); → RE: ISE
```

10) AssertionError :—

→ It is the child class of Error & hence it is unchecked.

→ Raised explicitly by the programmer or API developer to indicate that assert statement fails.

Ex: `assert(x>10);`

if x is not > 10 then we will get RE saying, AssertionError.

Exception / Error	Raised by
1) AIOOBE 2) NPE 3) CCE 4) NoClassDefFoundError 5) StackOverFlowError 6) ExceptionInInitializerError	Raised automatically by Jvm and these are Jvm Exceptions.
7) IAE 8) NFE 9) ISE 10) AE	Raised explicitly by programmer or API developer and hence these are Programmatic Exceptions.

Exception Propagation:—

→ Inside a method if an Exception raised & if we are not handle that Exception then the Exception object will be propargated to caller method then caller method is responsible to handle that Exception.

19) 1.7 version enhancements w.r.t Exception Handling:—

→ As the part of 1.7 version the following two concepts introduced in Exception Handling.

1) Multi-catch block
2) try with resources.

1) Multi-catch block :—

→ Eventhough multiple Exceptions having same handling code we have to write a separate catch block for every exception type in 1.6 version.

Ex:
```
try
{
  ==
}
catch(AE e)
{
  e.printStackTrace();
}
catch(IE e)
{
  e.printStackTrace();
}
catch(NPE e)
{
  S.O.p(e.getMessage());
}
catch(IOE e)
{
  S.O.p(e.getMessage());
}
```

→ The problems in this approach are

1) length of the code will be increased.
2) Readability of the code will be reduced.

→ To resolve these problems SUN people introduced multi-catch block in 1.7 version.

→ According to this we can write a single catch block which can handle multiple different type of Exceptions simultaneously. Such type of catch block is called Multi-catch block.

Ex:-
```
try
{
  ≡
}
catch(AE|IE e)
{
  e.printStackTrace();
}
catch(NPE|IOE e)
{
  s.o.p(e.getMessage());
}
```

→ In Multi-catch block there should not be any relation b/w Exception type (either child-parent or parent to child or same type), o.w. we will get CE.

Ex:
```
catch(Exception|AE e)
{
  e.printStackTrace();
}
```
→ CE ←
```
catch(AE|Exception e)
{
  e.printStackTrace();
}
```

2. try with resources:—

→ Until 1.6 version, whatever the resources we opened at the part of try block should be closed in finally block.

Ex:
```
BufferedReader br=null;
try
{
  br=new BR(new FR("input.txt"));
  // Use br based on our requirement
}
```

```
catch(IOE e)
{
  // Handling code
}
finally
{
  if(br!=null)
  {
    br.close();
  }
}
```

→ The problems in this approach are

1) We should compulsory close the resources in finally block and hence complexity of the programming will be increased.
2) We should compulsory write finally block which increases the length of the code so that readability will be reduced.

→ To overcome these problems SUN people introduced try with resources in 1.7 version.

→ The main advantage of try with resources, the resources which are opened as the part of try block will be closed automatically once the control reaches end of try block either normally or abnormally.

→ Hence we are not required to write finally block explicitly, which reduces complexity and length of the code.

Eg:-
```
try(BR br=new BR(new FR("input.txt")))
{
  Use br based on our requirement br will be
  closed automatically once control reaches end of
  try block either normally or abnormally.
}
catch(IOE e)
{
  Handling code.
}
```
(BR br=new BR(new FR("input.txt"))) → resource

Conclusions:-

1) we can declare any no. of resources, but these resources should be separated with ; (semicolon).

Ex: try(R1; R2; R3)
{
=
}

2) The resource reference variables are implicitly final. Hence within the try block we can't perform reassignment for that reference variable.

Ex: try(BR br=new BR(new FR("abc.txt")))
{
br=new BR(new FR("input.txt"));
}

3) The resources should be AutoClosable.

→ A resource is said to be AutoClosable iff the corresponding class implements j.l.AutoClosable interface.

→ AutoClosable interface introduced in 1.7 version & it contains only one method i.e., close() method.

4) Until 1.6 version, try should be followed by either catch or finally but from 1.7 version onwards we can take only try with resources with out catch or finally blocks.

Ex: try(R)
{
=
}

→ The main advantage of try with resources is we are not required to close resources explicitly & hence we are not required to write finally block.

→ Hence finally block will become dummy.

→ Until 1.6 version finally block is hero, but 1.7 version onwards finally block will become zero.