* Exception Handling-

```
class Lab {
  public static void main(String[] arg){
   System.out.println("Main Started");
   String st = arg[0];
   int x = Integer.parseInt(st);
   int res = 10/x;
   System.out.println("Result :"+res);
   System.out.println("Main Completed");
  }
}
```

interpreter will convert source code to machine/byte code

java interpreter
java.exe

intimates about the problem

JVM

analyze the problem
lookup the reason for problem

predefined libraries
JVM will verify which is the matching class for the type of problem

once the matching class is found in predefined library then the next step will be performed by JVM. (AIOBE).

JVM will create the Object of matching class
new AIOBE();

ref

ref → AIOBE

Now JVM will verify Is there any statement to handle the plan. If there is no statement to handle then control should be transfered to caller of the method (i.e main).

Event Generation

JVM will throw error object to the application at which line the problem is occured.

event handling

JVM

Default Control Handler

This will handle the problem and after that it will terminate the JVM abnormally

#NOTE- There is no chance that the control has to return back to application.

Exception Handling

For achieving the normal termination of JVM.
(i.e using try-catch)

04/12/2017

* Exception Handling-

The process of Handling the errors explicitely by the developers to achieve normal termination of JVM by using exception handler is known as Exception Handling.

⇒ There are 2 types of problem in Java-

a) Error

b) Exceptions

a) Error-

This will occure due to lack of resources. Any type of error cannot be handled by developer explicitely.

eg- Heap memory error, stack memory error, method definition error

So, many predefined errors classes are provided for errors and these are all the subclasses of java.lang.Error.

b) Exception-

This will occur due to grammatical errors or problems. All type of exceptions can be handled explicitely by the developer

So many predefined classes (Exception classes) for all predefined Exception and these are the subclasses of java.lang.Exception.

* Hierarchy-

java.lang.Object
  java.lang.Throwable
    java.lang.Error
    java.lang.Exception

} There is no inheritance between these two classes but both the classes are sub class of Throwable class.

ex.

```
class Lab {
    public static void main (String[] args){
    System.out.println ("Main Started");
    String st = args [0];
    int x = Integer.parseInt (st);
    int res = 10/x;
    System.out.println ("Result is :"+ res);
    System.out.println ("Main Completed");
    }
}
```

⇒ Execution on different cases for runtime arguments-

Case 1 : java Lab ↵ → ArrayIndex Outof Bound Exception.

Case 2 : java Lab JLC ↵ → NumberFormatException.

Case 3 : java Lab 0 ↵ → Arithmetic Exception.

Case 4 : java Lab 3 ↵ → Normal Excution.

NOTE-

When JVM handles the error then it will lead to abnormal termination of the application.

So, to perform normal termination try-catch block is introduced.

* try-catch block-

These are used for normal terminal of JVM.

Syntax-

```
try {
   // problem creating statement;
} catch (Exception e){          // else → Object obj = h1.show();
   // error handling statement;
}
```

Can be used for collection but we are having problem in exception not the Object so exception is used.

(Exception e) → VVI

Which type of error will be thrown / exception will be thrown we don't know so thats why (exception e) is written.

NOTE-

try block cannot return without catch block.

ex-

```
class Lab {
  public static void main (String[] args){
  System.out.println ("Main Started");
  try {
     String st = args[0];                    // → block 1   A100B
     int x = Integer.parseInt (st);          // → block 2
     int res = 10/x;                         // → block 3
     System.out.println ("Result :"+res);
   } catch (Exception e){
       System.out.println ("Error is handled :"+ e );
     }
  System.out.println ("Main Completed");
  }
}
```

=> If problem occured for 1st block then control will be transfered to catch block and there is no chance to execute 2nd & 3rd block.

=> Only try block statement problem is there, that only will be handled by catch block.

=> Within catch block, if any error occured, it will not be handled by catch block.

=> If no problem in try block then catch block will not be executed.

* Rules for defining try and catch block-

1) Inside the try block, enclose the problem creating statement.
2) Inside the catch block, enclose the error handling statement.
3) Inside the try block whenever the statement has the problem then control will be transfered to handle the problem by executing the catch block then the other statement will be executed.
4) While declaring the catch block the reference variable of Exception / Throwable class is required because whatever the jvm has thrown the error object, it will be collected by the catch block of Exception reference variable since it is the super class of all exception.
5) Between try and catch block no any statements are allowed.
6) Always catch block has to be followed with the try block. Without try block we should not provide catch block.
7) try block should be declared with one or more catch block.
8) Inside the try block if problem is occured at the first statement or second statement then the control will be transfered to catch block to handle the exception and there is no chance to execute the other statements of try block
9) If there is no problem in try block then catch block will not be executed.

10) Between multiple catch block also no statements are allowed.

11) If the statements of try block has the problem then only catch block will handle the exception or else outside try block or inside the catch block statement if problem occures then JVM has to handle the problem.

12) While declaring multiple catch block the order should be sub type to super type.

13) If any problem occures in try block then for that problem if matching catch block is there then it will handle or else it will be handled by JVM.

05/12/2017

* Multiple Exceptions -

Syntax.

```
catch (E1 | E2 | E3 | E4 | E5 e){
    // Statement for error handling;
}
```

In above statement of catch block (Exception e) cannot be used in the same catch block for Multiple Exceptions. because of super type inheritance relation occurs. but can be used in another catch block.

Rule -

E1, E2, E3, E4, E5 should not contain any inheritance relationship.

=> We can provide custome error message for multiple exceptions by using single catch block.

ex-

```
class Lab {
    public static void main (String[] args){
    System.out.println("Main Started");
    try{
        String st = args[0];
        int x = Integer.parseInt(st);
        int res = 10/x;
        System.out.println("Result is :"+res);
    } catch (ArrayIndexOutOfBoundsException | NumberFormatException |
                ArithmeticException e){
        if (e instanceof ArrayIndexOutOfBoundsException){
            System.out.println("You are not giving i/p value");
        } else if (e instanceof NumberFormatException){
            System.out.println("You should provide numerical value");
        } else
            System.out.println("You are performing division by zero");
        }
    System.out.println("Main Completed");
    }
}
```

NOTE-

Catching of multiple exception with single catch block is provided to reduce the lengthy of source code and to overcome the duplicate piece of code.

* finally block-

It is a specified block which is declared with exception handler (try and catch blocks) only and it is used for mandatory execution of statements without any fail in any scenario.

⇒ Statements-

①
```
try {
} catch (Exception e){
} finally {
}
```

②
```
try {
} catch (Exception e) {
} catch (Exception e) {
} catch (Exception e) {
} finally {
}
```

③
```
try {
} finally {
}
```

finally block will not accept

1) if any exception in finally block
2) JVM termination with System.exit(0);

Q- What is the difference between final, finalize and finally

vvvImp

* final

=> final is a pre-defined keyword which is used for variable, methods and classes.

=> if we declare the variables of the class as the final then it will be considered as the constant so that we cannot modify the content of the variable

=> if we declare the methods of the class as the final then that method cannot be overridden in subclass.

=> if we declare the class as the final then we cannot inherit the property of that class (we can't define any subclass)

* finalize-

It is the predefined method of object class which is used in the Garbage Collector process i.e- before invoking GC to clean the memory, it is calling finalize() automatically for closing the connections with the resources on unused objects.

But Object class finalize() doesn't have any statement to close the connection with the resource. So we need to override the finalize() in each and every class to provide resources cleanup code.

* finally Block-

finally block must and should be used with the try-catch block only and it is used for executing any piece of code with or without any fail. It is also used for writing the try block without catch block by using finally block.

We can use finally block for closing the resources connection in an application by providing resources cleanup code explicitely.

06/12/2017

* Nested try-catch finally block-

The process of writting the try catch finally blocks inside the try catch and finally block to achieve normal termination of JVM always is known as nested try catch final Block.

There are some special scenario where we can use nested try catch finally block that are.

1) In try block we need to execute some statement without fail even through Exception occurs.

2) If any problem occurs in catch and finally blocks then by default it will handled by JVM and it will achieve abnormal termination of JVM. But in these cases also we need to handle the problem catch and finally blocks explicitely to achieve normal Termination.

* printStackTrace -

It is a method from throwable class which is used to get the details about the exception, these are types of exception, error messages of exception, at which line number the problem is occured, from where to where the control is transfering to handle the exception and if any cause is there for exception then it will display or else it dosn't display the cause whenever it is null.

=> printStackTrace() is tracing all the details about the type of problem where exactly the occurance is there.

=> It is an overloaded method which is used for printing the o/p on the console as it is or it will print the exception in an output device by using io API classes.

ex-

```
class Lab{
  public static void main (String[] args){
    System.out.println (java.io.PrintStream);
    System.out.println (java.io.PrintWriter);
  }
}
```

getMessage(); → displaying the error message for any exception.

getCause(); → displays if any cause for the exception.

ex-

```
class Lab{
  public static void main (String[] args){
    System.out.println ("Main() Started");
    try{
      new Hello().show();
    }catch(Exception e){
        System.out.println ("Exception :"+e);
        System.out.println ("Message :"+ e.getMessage());
        System.out.println ("Cause :"+ e.getCause());
        System.out.println ();
        e.printStackTrace();
      }
    System.out.println ("Main() Completed");
  }
}

class Hello{
  void show(){
      System.out.println ("show() begins");
      new A().m1();
      System.out.println ("show() ends");
  }
}
```

```
class A{
    void m1(){
    System.out.println("A -> m1() begins");
    new B().m2();
    System.out.println("A -> m1() ends");
    }
}

class B{
    void m2(){
     System.out.println("B -> m2() begins");
      new C().m3();
     System.out.println("B -> m2() ends");
     }
}

class C{
    void m3(){
      System.out.println("C -> m3() begins");
     int x=10/0;
     System.out.println("C -> m3() ends");
     }
}
```

=> Classification of Exceptions-

1) Checked Exceptions → compile time exception
2) Unchecked Exceptions → Runtime exception.
3) Builtin Exceptions
4) User defined Exceptions

ex-

```
class Lab {
    public static void main (String [] args) {
        System.out.println ("Main Started");
        System.out.println (10/0);      -> unchecked exception.
    }
}
```

For above, there will be no compilation error. The error will occur at the runtime.

1) Checked Exception-

The type of exception verified by the compiler at the compilation time is known as checked exception but every checked exception must and should be reported or we need to provide solution for the checked Exception in 2 ways:-

a) by using try-catch block
b) by propogating with throw keyword.

=> Reporting of checked exception is mandatory. In case, if you are not handling checked exception then compiler will force you to handle checked exceptions by giving some error.

=> Unreported exception CloneNotSupportedException, must be caught or declared to be thrown.

ex-

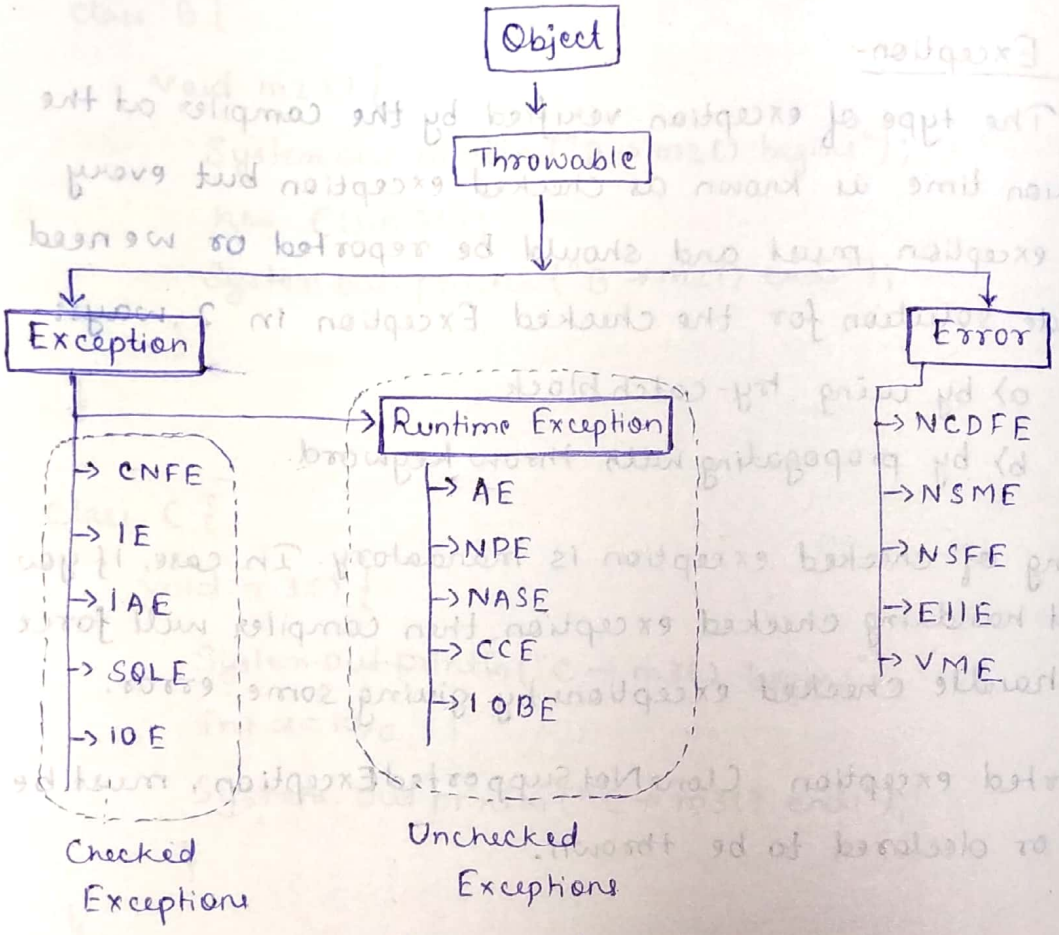
```
class Hello {          // implements Cloneable
    public static void main (String [] args) { // throws CloneNotSupportedException
        System.out.println (" Main Started");
        Hello h = new Hello();
        Object obj = h.clone();  ——→ Checked → compiler.
                                          ↓
        System.out.println (" Main completed");   error message: unreported exception CNSE; must be caught or thrown
    }
}
```

remove comments to get the o/p.

=> When the compiler forces to handle Exception then it is called as checked Exception.

=> Predefined exception class and its sub class except RuntimeException and its sub classes are known as checked Exception

=> hierarchy-



=> highly restrictive => less restrictive

2) Unchecked Exception-

The type of exception which are not verified by compiler at the compilation time and these exception occurs at runtime is known as Unchecked Exception.

=> Every unchecked Exception may or may not reported in 2 ways as exception as runtime may occurs or may not occurs.

a) by using try and catch block

b) by propogating with throw keyword.

=> Reporting of unchecked exceptions are optional. The predefined Runtime exception and its sub class is known as Unchecked exceptions.

=> To differenciate which is checked or which is unchecked among the multiple Exceptions we need to use following steps-

a) Verifying the hierarchy of exception with Runtime Exception class by using instanceof operator.

b) Verifying whether the Exception forcing by compiler to handle or not. If compiler is forcing to handle then checked or else it is unchecked.

c) In an empty try-catch block if we write any exception in catch block then it will give error means that it is a checked exception or else if it is not giving any error then it is an unchecked exception.

3) Builtin Exception-

They are the predefined exception which cannot be modified

4) User Defined Exception-

a) Declare the java class with custom Exception name.

b) Provide the hierarchy with predefined exception class and it is recomended to make the inheritace relationship with either exception class or Runtime Exception class

> Exception class → custom checked Exception

> Runtime Exception class → custom unchecked Exception.

c) Inside the custom Exception class if we want we can declare the variable and initialize using constructor.

d) If you want to provide any custom error message for user defined exception then we can give the implementation in two ways:-

(i) Overriding the getMessage() from Throwable class in custom exception class by returning string as a message.

(ii) Provide the constructor defination of custom exception class with String as a parameter and inside this constructor we need to call super class constructor with same String as parameter.

ex. for unchecked

```
class StudentNotFoundException extends RuntimeException{
    String sid;
    StudentNotFoundException(String sid){
        this.sid = sid;
    }
    public String getMessage(){
        return sid+"is invalid in the Database";
    }
}
```

ex. for checked-

```
class InvalidPinException extends Exception{
    public InvalidPinException(String msg){
        super(msg);
    }
}
```

```
class StudentNotFoundException extends RuntimeException{
    StudentNotFoundException(String sid){
        super(sid);
    }
}
```

07/12/2017

* throws Keyword-

It is a predefined keyword which is used to throw any type of Exceptions explicitely any where in the application

syntax-

> throw < throwable type Object >;
> throw new ClassNotFoundException();
> throw new ArithmeticException();

ex-

```
class Lab {
  public static void main (String[] arg) {
    System.out.println("Main Started");
    String nm = " ";
    try {
      StudentService serv = new StudentService();
      nm = serv.getNameBySid("JLC-099"); //-> (null) or (" ")
      System.out.println("Name :"+nm);
    } catch (Exception e){
        e.printStackTrace();
    }
      System.out.println("Main Completed");
  }
}

class EmptySidException extends RuntimeException{}

class StudentService{
    String getNameBySid (String sid){
      if (sid==null){
          throw new NullPointerException();
      }else if (sid.isEmpty()){
          throw new EmptySidException();
      }else if (sid.equals("JLC-099")){
        return "SRINIVAS";
      }else
          throw new StudentNotFoundException(sid);
    }
}
```

* throws Keyword-

It is a predefined keyword which is used for propaging (enlimating) about the Exception to the JVM but we are not handling and throws keyword is mainly for reducing the length of source code that everytime we need to report the Exception with try and catch block for multiple times so we can report about the exception to the JVM only once at the method level declaration.

=> throws keyword is mandatory for checked Exceptions and optional for unchecked Exceptions

=> throws keyword propogation about the Exception can't be handled explicitely by the developer and it must and should be handled by the JVM component and it will achieve abnormal termination.

=> throws keyword is used only at the method level of application and it can't be used anywhere in the application

Syntax-

```
throws  throwableTypeName;
```

ex-

```
public class Lab {
    public static void main (String[] args){
     System.out.println ("Main() Started");
     try {
        StudentService serv = new StudentService();
        serv.getNameBySid (null);
      } catch (Exception e){
        e.printStackTrace();
      }
       System.out.println ("Main() Ends");
    }
```

```
class StudentService {
    String getNameBySid(String sid){
      if (sid == null || sid.isEmpty() || !$id.equals("JLC-099"))
        throw new StudendNotFoundException(sid);
      else
        return "SRINIVAS";
      }
  }

class StudentNotFoundException extends RuntimeException {
      StudentNotFoundException(String sid) {
          super(sid);
      }
}
```

o/p → Main Started
Main Completed
Exception

ex-

```
public Lab {
  public static void main (String[] arg) {
   System.out.println("Main Started");
   try{
    StudentService sesw = new StudentService();
      sesw.getNameBySid(null);
   } catch (Exception e){
     e.printStackTrace();
    }
    System.out.println("Main Completed");
  }

class StudentService {
    String getNameBySid(String sid) throws StudentNotFoundException{
      if (sid == null || sid.isEmpty() || !sid.equals("JLC-099")){
          throw new StudentNotFoundException(sid);
      } else
        return "SRINIVAS";
    }
}
```

```
class StudentNotFoundException extends Exception {
    StudentNotFoundException (String sid){
        super (sid);
    }
}
```

ex-

```
public class Lab{
    public static void main (String[] args) {
        System.out.println("Main Started");
        StudentService serv = new StudentService();
        String ss = serv.getNameBySid (null);
        System.out.println (ss);
        System.out.println ("Main Completed");
    }
}

class StudentService {
        String getNameBySid(String sid) throws StudentNotFoundException
            return "SRINIVAS";
        }
}

class StudentNotFoundException extends Exception {
        StudentNotFoundException (String sid){
            super (sid);
        }
}
```

o/p

```
Main Started
SRINIVAS
Main Completed.
```

* Difference between throw and throws keyword- #VVVImp

throw	throws
1) It is a keyword which is used for throwing any exception explicitely as per the application requirement.	1) It is the keyword which is used for propagating about the Exception to the JVM at the method level.
2) throw keyword can be used o any where in the class definition.	2) throws keywords are restricted at the method level declaration.
3) It can be used for either checked or unchecked exception	3) It is used for both checked or unchecked but it is mandatory for checked and optional for unchecked Exception.
4) If we are throwing exception explicitely by using throw keyword then we can write with try and catch block and handle the exception to achieve normal termination of JVM.	4) If we are intémating about the exception with throws keyword at the method level then no statement is there to handle the exception to acheive normal termination of JVM.
5) While using throw keyword we need to specify throwable type object.	5) While using throws keyword we need to specify throwable class name.
6) If we are throwing exception explicitely by using throw keyword we can handle the exception by using try and catch block.	6) Propagating the exception with throws keyword it can't handle Exceptions. Always it will achieve Abnormal termination.

Q- Reporting of Exceptions can be done with try catch block and throws keyword then what are benifits and drawbacks?

Ans-

=> If we are reporting the exception with the throws keyword instead of try and catch block then we can reduce lengthiness of source code and multiple times we need to report with try catch block but we can report any exception by using throws keyword only once i.e at the method level.

=> If we are reporting the exception with throws keyword then try catch block there is a drawback, i.e. with throws keyword always we will achieve abnormal termination of JVM but with try catch block, catch block will handle the exception to achieve normal termination of JVM.

NOTE-

Most of the applications while reporting exceptions either checked or unchecked, we will use throws keyword and with try and catch block, there is complexity.

08/12/2017

* Revision of method overloading - overriding

1) When super class method is not specified with method level exception then sub class method can do the following:
   a) Sub class method may not throw any method level exception
   b) Sub class method can throw the unchecked exception.
   c) Sub class method can't throw the checked exception.

2) When super class method is specified with some method level checked exception then sub class method can do following:
   a) Sub class method can ignore that method level exception.
   b) Sub class method can throw the same exception.
   c) Sub class method can throw the exception which is sub class to super class method exception.
   d) Sub class method can throw any unchecked exception.
   e) Sub class method cannot throw the exception which is super class to super class method exception.
   f) Sub class method cannot throw the exception which is non-sub class of superclass method exception.

3) When super class method is specified with some method level unchecked exception then sub class method can do following:-
   a) Sub class method can ignore that method level exception.
   b) Sub class method can throw the same exception.
   c) Sub class method can't throw any checked exception.
   d) Sub class method can throw any otherunchecked Exception.

ex-

```
class Hello {
    void show(){}
    void display(){}
}
class Hai extends Hello{        //-> valid
    void show(){}
    void display() throws NullPointerException{}        -unchecked
}
```

ex-
```
Class Hello {
    void show() {}
}
Class Hai extends Hello {          //-> invalid
    void show () throws ClassNotFoundException {}     -> checked
}
```

ex-
```
import java.io.IOException;
Class Hello {
    void show() throws IOException {}          -> checked
}
Class Hai extends Hello {
    void show() {}          //-> valid
}
```

ex-
```
import java.io.IOException;
Class Hello {
    void show() throws IOException {}          -> checked
}
Class Hai extends Hello {
    void show() throws IOException {} //-> valid.  -> checked
}
```

ex-
```
import java.io.IOException;
Class Hello {
    void show() throws IOException {}          -> checked
}
Class Hai extends Hello {
    void show() throws NullPointerException {}  //-> valid.  -> unchecked
}
```

ex-

```
import java.io.*;
class Hello{
    void show() throws IOException{}          -> checked
}
class Hai extends Hello{
    void show() throws FileNotFoundException{} //-> valid.
}                                  └> checked
                                      sub class
```

ex-

```
import java.io.IOException;
class Hello{
  void show() throws IOException{}          -> checked
}
class Hai extends Hello{
    void show() throws Exception{} //-> not valid.
}                                  └> super type  so invalid
```

ex-

```
import java.io.*;
class Hello{
    void show() throws IOException{}          -> checked
}                                              (domain/hierarchy)
class Hai extends Hello{
    void show() throws ClassNotFoundException{}  //-> invalid
}                                  └> checked (~~exception hierarchy~~)
                                      (checked Exception has to be same)
                                      (different checked Exception are not valid)
```

ex-

```
class Hello{
  void show() throws ArrayIndexOutOfBoundsException{} ->unchecked
}
class Hai extends Hello{  //-> valid.
    void show(){}                          ->no exception
}
```

ex-

```
class Hello {
    void show () throws ArrayIndexOutOfBoundsException {}  -> unchecked
}
class Hai extends Hello {
    void show () throws NullPointerException {} //-> valid.
}
```

↳ unchecked unchecked = any unchecked = valid

ex-

```
class Hello {
    void show() throws ArrayIndexOutOfBoundsException {}  -> unchecked
}
class Hai extends Hello {  //-> valid
    Void show () throws RuntimeException {} //-> valid.
}
```

↳ unchecked

ex-

```
class Hello {
    void show () throws ArrayIndexOutOfBoundsException {}  -> unchecked
}
class Hai extends Hello {
    void show () throws Exception {}  //-> not valid.
}
```

↳ super type to super type

ex-

```
class Hello {
    Void show () throws ArrayIndexOutOfBoundsException {}-> unchecked
}
class Hai extends Hello {
    void show() throws ClassNotFoundException {} //-> not valid
}
```

↳ checked.

unchecked ≠ checked
not valid.

* try-with-resource statement-

before java 7.

ex.

```
try {

   }
catch (Exception e) {
   }
finally {
   st;
  }
```

=> JDBC connectivity-

```
Connection conn = null;
Statement  st   = null;
ResultSet  rs   = null;

try {
   conn = new Connection();
   st = new Statement();
   rs = new ResultSet();
} catch (Exception e){
   System.out.println ("Error Handling");
} finally{
   if (rs != null)
      rs.close;
   if(st!=null)
      st.close;
   if (conn!=null)
      Conn.close();
}
```

Explicitely resource clean up

Closing all of the connection explicitely before java 7

=> It is used for closing the connection with resource Objects automatically once the application is completed by JVM.

=> try-with-resource statement can be declared without catch block and finally block.

=> Any resources which we are using in the application is a java Object and that should be subtype of AutoCloseable interface.

Syntax-

```
try (Resource Declaration and Initialization){
    statement;
}
```

from java 7

```
try ( Connection conn = DM.getConnection ("");
      Statement st = conn.createStatement();
    )
    { .....
    } Catch( SQL.Exception e){
    -----
    }.
```

#NOTE.

Here we don't need to close the connection, automatically it will be closed.

* Before java 7 resource cleanup code should be provided explicitely in finally block.

ex-

```
class Lab {
    public static void main (String [] args) throws Exception {
        System.out.println ("Main Started");
        JLCResources res1 = null;
        JLC Resources res2 = null;
        try{
        res1 = new JLC Resources(101);
        res2 = new JLC Resources(102);
        System.out.println (" In try block we are using resource Object");
        } catch (Exception e){
            System.out.println (" Catch block ");
            e.printStackTrace ();
        } finally {
            System.out.println ("Finally Block");
            if (res1 != null)
                res1.close ();
            if (res2 != null)
                res2.close ();
            }
        System.out.println ("Main Completed");
    }

class JLCResources implements AutoCloseable{
        int rid;
        public JLCResources (int rid){
            this.rid = rid;
        }
public void close() throws Exception{
        System.out.println ("JLC Resources Object is closed:"+rid);
    }
}.
```

* After java 7 with "try-with resource statement we can close the resources Automatically.

```
public class Lab{
    public static void main (String[] args) throws Exception{
        System.out.println ("Main Started");
        try(
            JLCResources res1 = new JLCResources (101);
            JLC Resources res2 = new JLC Resources (102);
        ){System.out.println ("In try block we are using resource Object");
        }
        catch (Exception e) {
            System.out.println ("Catch Block");
            e.printStackTrace();
            }
        System.out.println ("Main Completed");
        }
    }

class JLCResources implements AutoCloseable{
    int rid;
    JLC Resources (int rid){
        this.rid = rid;
    }
    public void close() throws Exception {
        System.out.println ("JLC Resource Object is closed:":rid);
    }
}
```

* Invalid cases of try with Resource Statement-

1)
```
JLCResources res1 = null;
JLCResources res2 = null;
try {
    res1 = new JLCResources(101);
    res2 = new JLCResources(102);
}
```
} invalid

Resource declaration and initialization has to be provided in same line

2)
```
class JLCResources {
    try {
        JLCResources res1 = new JLCResources(101);
    }
```

Since resources is not subtype of Autocloseable interface then it will throw the error as follows:-

> the resource type JLCResources does not implement java.lang.Autocloseable.

3)
```
try (){}
```

if we are not using any resource connections then we should not specify try with resource statement.

* throwing exceptions with improvement-

throw new Exception();	catch (Exception){}	void m1() throws Exception
throw new Error();	catch (Error){}	void m1() throws Error
throw new Throwable();	catch (Throwable){}	void m1() throws Throwable
throw new Object();	catch (Object){}	void m1() throws Object
not OK	invalid	invalid
invalid		

* throwing Exceptions with improved type checking-

1) valid for all JDK-

```
class Hello{
    void show() throws CloneNotSupportedException{
        try{
            this.clone();
        }catch(CloneNotSupportedException e){
            throw e;
        }
    }
}
```

2) valid for all JDK-

```
class Hello{
    void show() throws Exception{
        try{
            this.clone();
        }catch(CloneNotSupportedException e){
            throw e;
        }
    }
}
```

3) valid for all JDK-

```
class Hello{
    void show() throws Exception{
        try{
            this.clone();
        }catch(Exception e){
            throw e;
        }
    }
}
```

4) Invalid before Java 7 but valid from Java 7.

```
class Hello{
    void show() throws CloneNotSupportedException{
        try{
            this.clone();
        }catch(Exception e){
            throw e;
        }
    }
}
```

5) Some errors can be handled using try catch block such as Outofmemory error.

ex.

```
public class Lab{
    public static void main(String[] args){
        Student sarr[] = new Student[500]; //-> outofmemory error
        try{
            for(int i=0; i<sarr.length; i++){
                sarr[i] = new Student();
                System.out.println((i+1)+" Object created");
            }
        }catch(Error e){                    //-> Object -> x   Throwable -> ✓
            System.out.println("\n** Error Occured :"+e);
        }
        System.out.println("\n After Handling");
        Student st= new Student();
    }
}

class Student{
    long arr[] = new long[215833];
}
```

error handling

again OutofMemory error will occur