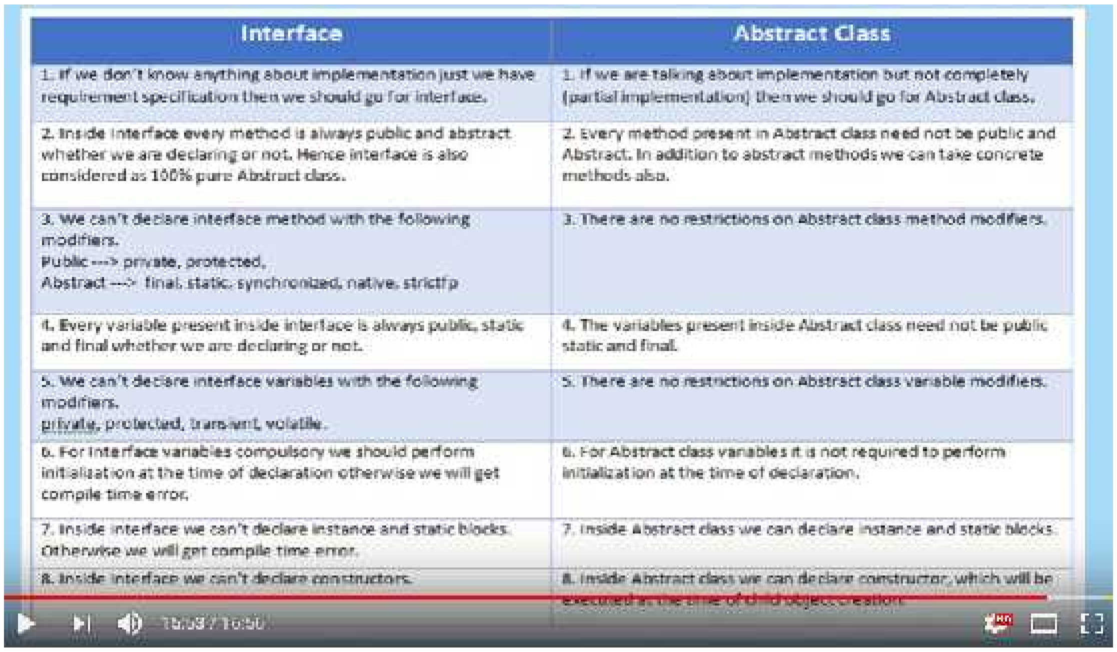
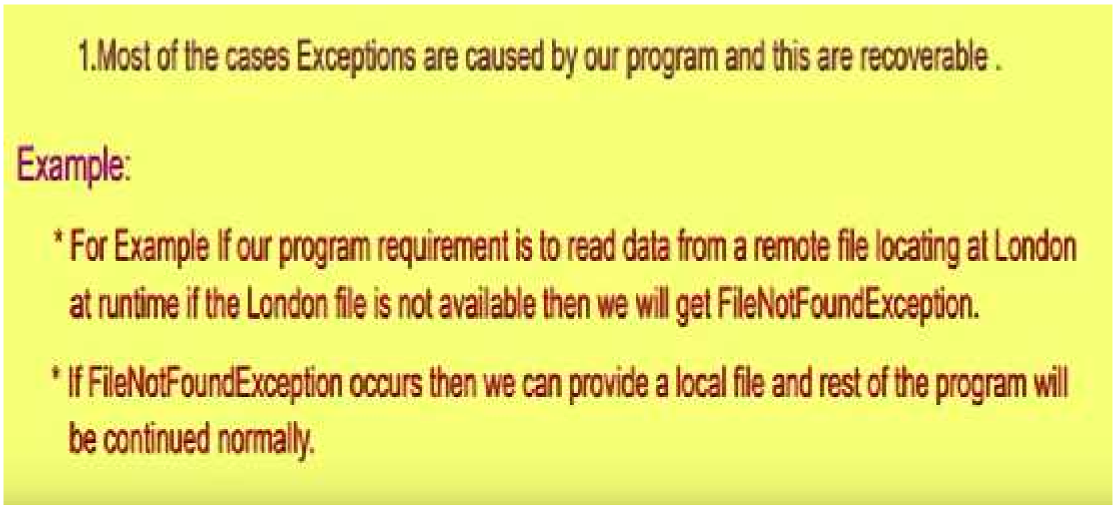


Interface	Abstract Class
1. If we don't know anything about implementation just we have requirement specification then we should go for interface.	1. If we are talking about implementation but not completely (partial implementation) then we should go for Abstract class.
2. Inside interface every method is always public and abstract whether we are declaring or not. Hence interface is also considered as 100% pure Abstract class.	2. Every method present in Abstract class need not be public and Abstract. In addition to abstract methods we can take concrete methods also.
3. We can't declare interface method with the following modifiers. Public ---> private, protected, Abstract ---> final, static, synchronized, native, strictfp	3. There are no restrictions on Abstract class method modifiers.
4. Every variable present inside interface is always public, static and final whether we are declaring or not.	4. The variables present inside Abstract class need not be public static and final.
5. We can't declare interface variables with the following modifiers. private, protected, transient, volatile.	5. There are no restrictions on Abstract class variable modifiers.
6. For interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.	6. For Abstract class variables it is not required to perform initialization at the time of declaration.
7. Inside interface we can't declare instance and static blocks. Otherwise we will get compile time error.	7. Inside Abstract class we can declare instance and static blocks.
8. Inside interface we can't declare constructors.	8. Inside Abstract class we can declare constructor, which will be executed at the time of child object creation.

Throwable acts as root for java exception hierarchy

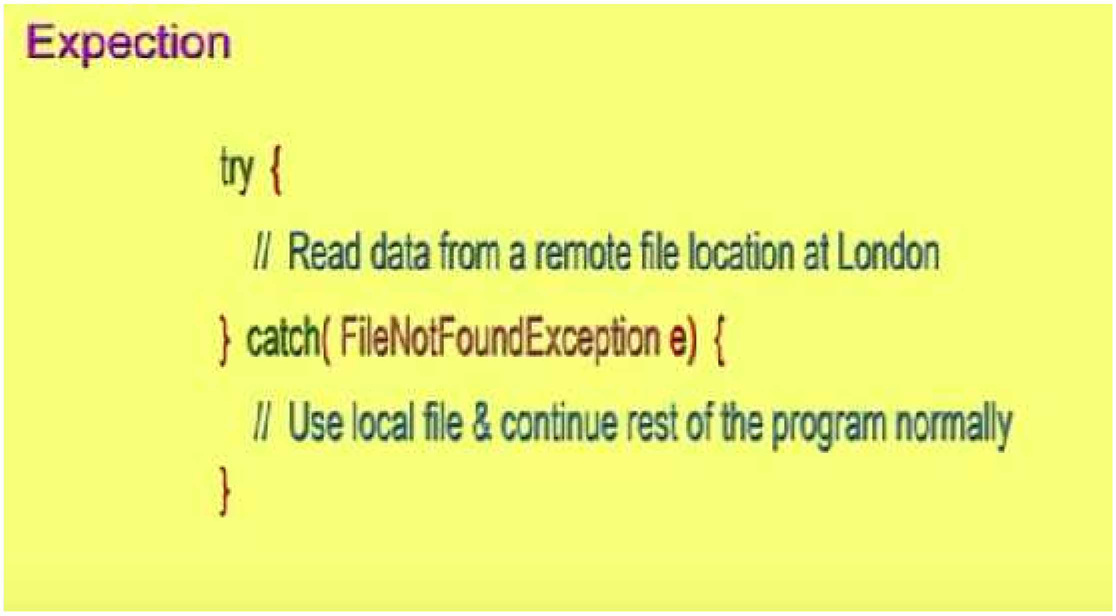


1.Most of the cases Exceptions are caused by our program and this are recoverable .

Example:

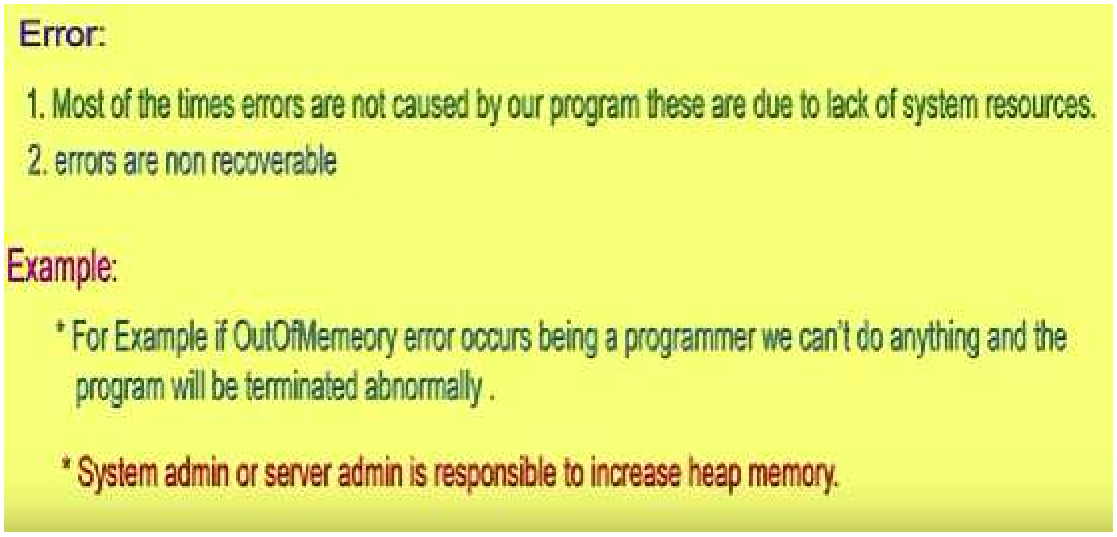
* For Example If our program requirement is to read data from a remote file locating at London at runtime if the London file is not available then we will get FileNotFoundException.

* If FileNotFoundException occurs then we can provide a local file and rest of the program will be continued normally.



Expection

```
try {
   // Read data from a remote file location at London
} catch( FileNotFoundException e) {
   // Use local file & continue rest of the program normally
}
```



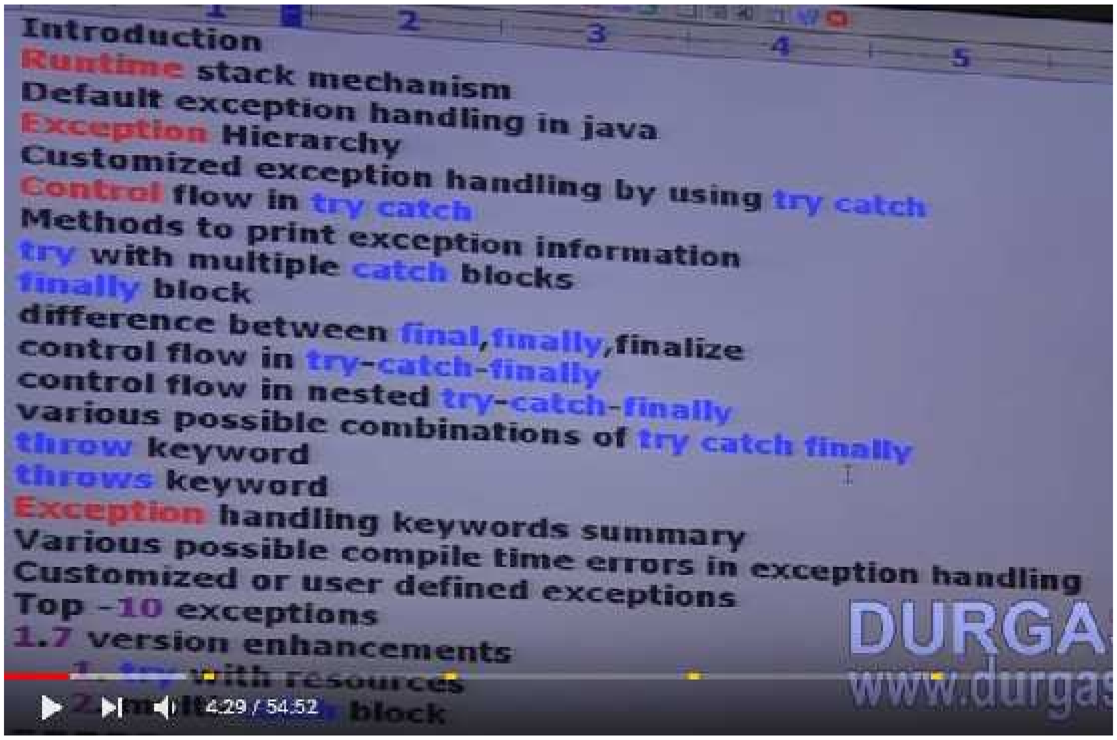
Error:

1. Most of the times errors are not caused by our program these are due to lack of system resources.
2. errors are non recoverable

Example:

* For Example if OutOfMemeory error occurs being a programmer we can't do anything and the program will be terminated abnormally .

* System admin or server admin is responsible to increase heap memory.



Introduction
Runtime stack mechanism
Default exception handling in java
Exception Hierarchy
Customized exception handling by using try catch
Control flow in try catch
Methods to print exception information
try with multiple catch blocks
finally block
difference between final,finally,finalize
control flow in try-catch-finally
control flow in nested try-catch-finally
various possible combinations of try catch finally
throw keyword
throws keyword
Exception handling keywords summary
Various possible compile time errors in exception handling
Customized or user defined exceptions
Top -10 exceptions
1.7 version enhancements
1. try with resources
2. multi [illegible] block

Introduction:

An unexpected unwanted event that distrubes normal flow of the program is called exception.

e.g. TyrePunchered Exception. SleepingException, FileNotFound Exception etc.

It is highly recomended to handle exceptions and the main objective of exception handling is graceful terminarion of the program.

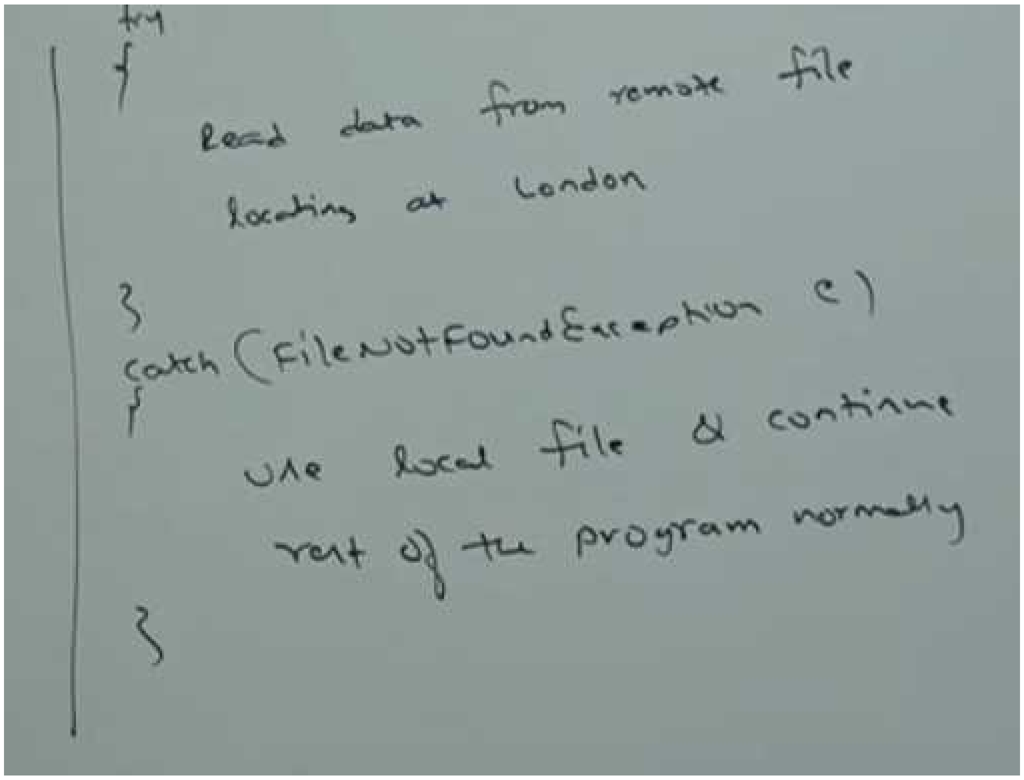
Exception handling does not mean repairing an exception, we have to provide alternative way to continue rest of the program normaly, is the concept of exception handling.

e.g.

Our programming requirement is to read data from remote file locating at London. At runtime if London file is not available our programe should not be terminated abnormally.

We have to provice some local file to continue rest of the program normally.

This way of defining alternative is nothing but exception handling.



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

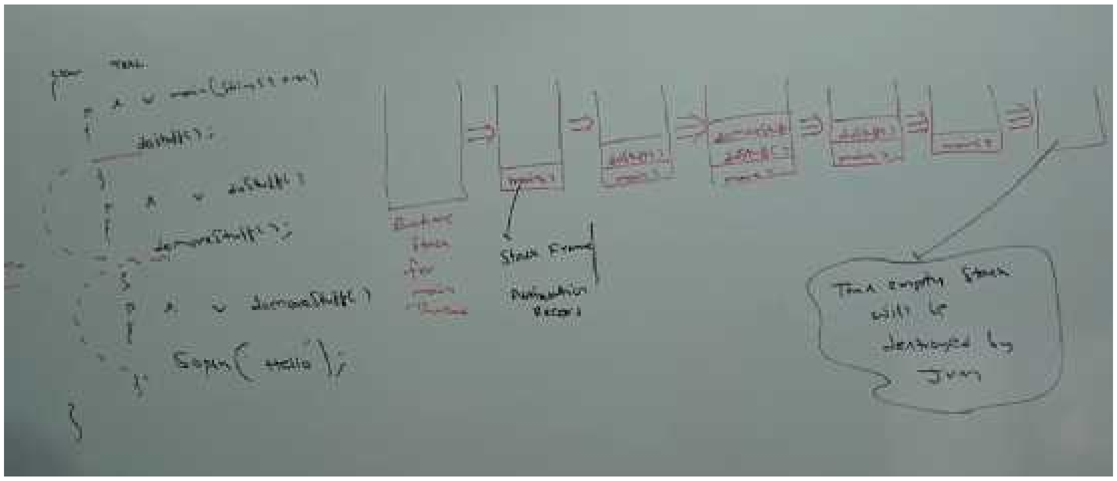
Runtime Stack Mechanisem:

For every thread JVM will create a runtime stack. Each and every method call performed ny that thread will be stored in the corrosponding stack.

Each entry in the stack is called stack frame or activation record.

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

After completing all method calls the stack will become empty and the empty stack will be destroyed by jvm just before terminating the thread.



Default Exception Handling in Java:

1. Inside a method if any exception occurs the method in which it is raised is responsible to create exception object by including the following information,

    i. Name of exception

    ii.Description of exception

    iii.Location at which exception occures (Stack trace)

2. After creating xception object method handovers that object to the JVM.

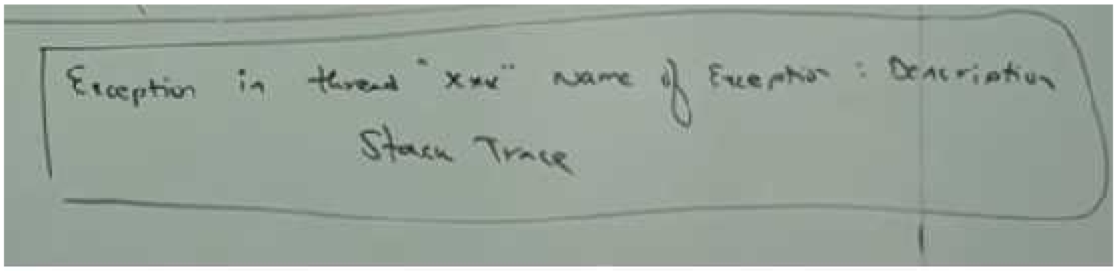
3. JVM will check whether the method contains any exception handling code or not.If the mthod does not contain any exception handling code then JVM terminates that method abnormally and the removes the corrosponding entry from the stack.

4. Then JVM identifies caller method and checks whether caller method contains any handling code or not . If the caller method doesnot contain any handling code then JVM terminates that caller method also abnormally and removes the corrosponding entry from the stack.

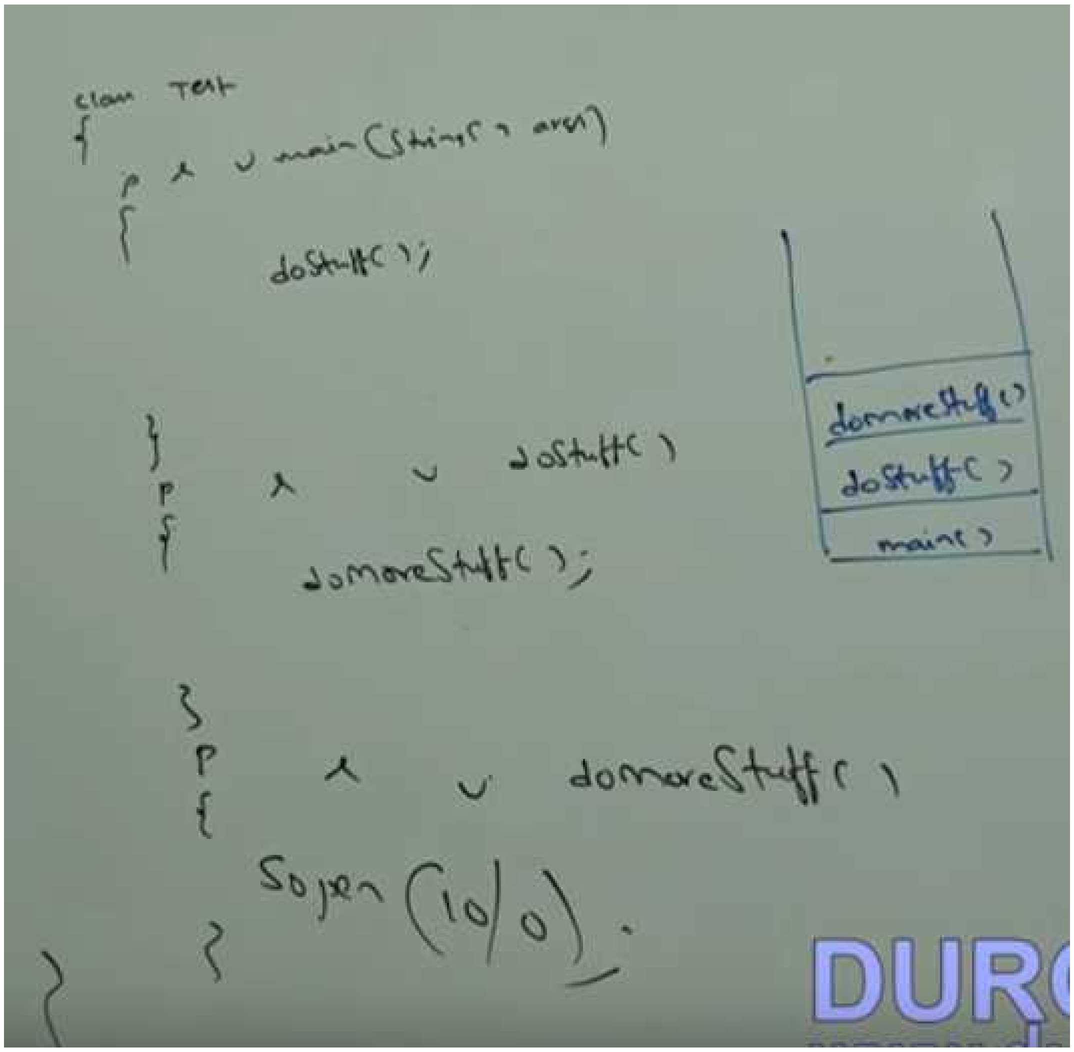
5. This process will be continued until main method and if the main method also does not contain handling code then JVM terminates main method also abnormally and removed corrospondig entry from the stack.

6. Then JVM handovers responsibility of exception handling to the default exception handler, which is the part of JVM.

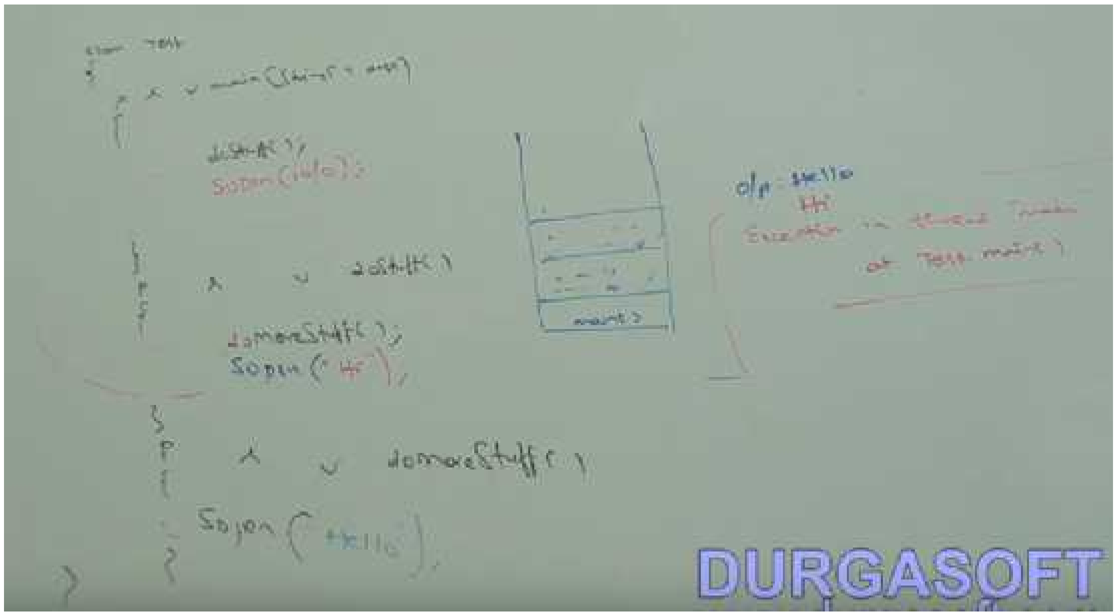
7. Default exception handler prints exception info in the following format and terminates program abnormally.



Exception in thread "xxx" name of Exception : Description
Stack Trace



```
class Test
{
   p s v main(String[] args)
   {
         doStuff();
   }
     p   s   v   doStuff()
     {
         doMoreStuff();
     }
      p   s   v   doMoreStuff()
      {
        Sopln(10/0).
      }
}
```



Note:

In a program if at least one method terminates abnormally then the program termination is abnormal termination.

If all methods terminates normally then only programe termination is normal termination.

Exception herichiey:

Throwable class acts as root for java exception hierchiecy.

Thowable class defines two child classes.

1. Exception

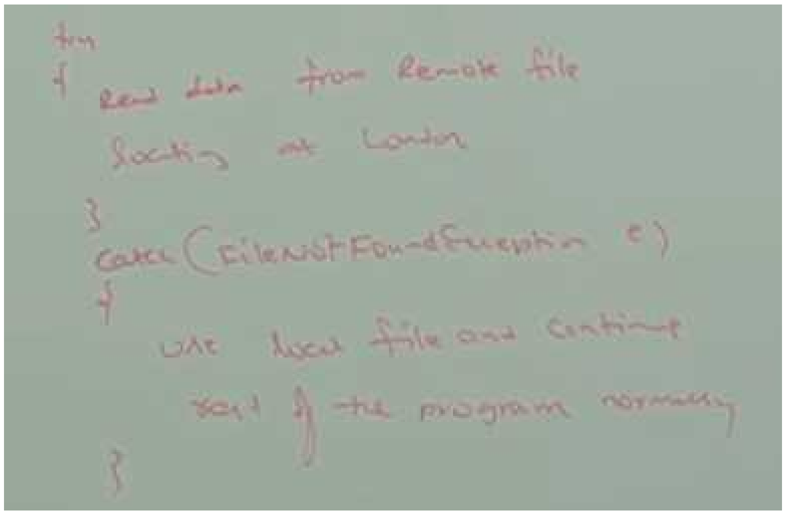
2. Error

Exception:

Most of the times exceptions are caused by our program and these are recoverable.

e.g. If our programming requirement is to read data from remote file locating at London at runtime if remote file is not avilable then we will get runtime exception saying FileNotFound Exception.

If FileNotFoundException occurs we can provide local file and continue rest of the program normaly.



```
try
{
  Read data from Remote file
   locating at London
 }
 catch(FileNotFoundException e)
 {
    use local file and continue
      rest of the program normally
  }
```

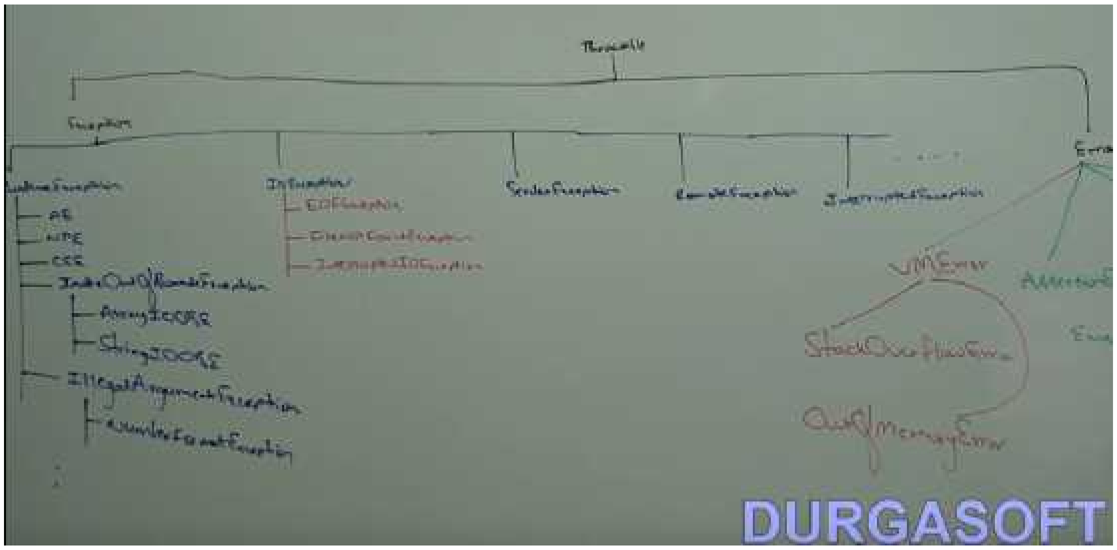
Error:

Most of the times errors are not caused by our progrmes and these are due to lack of system resources.

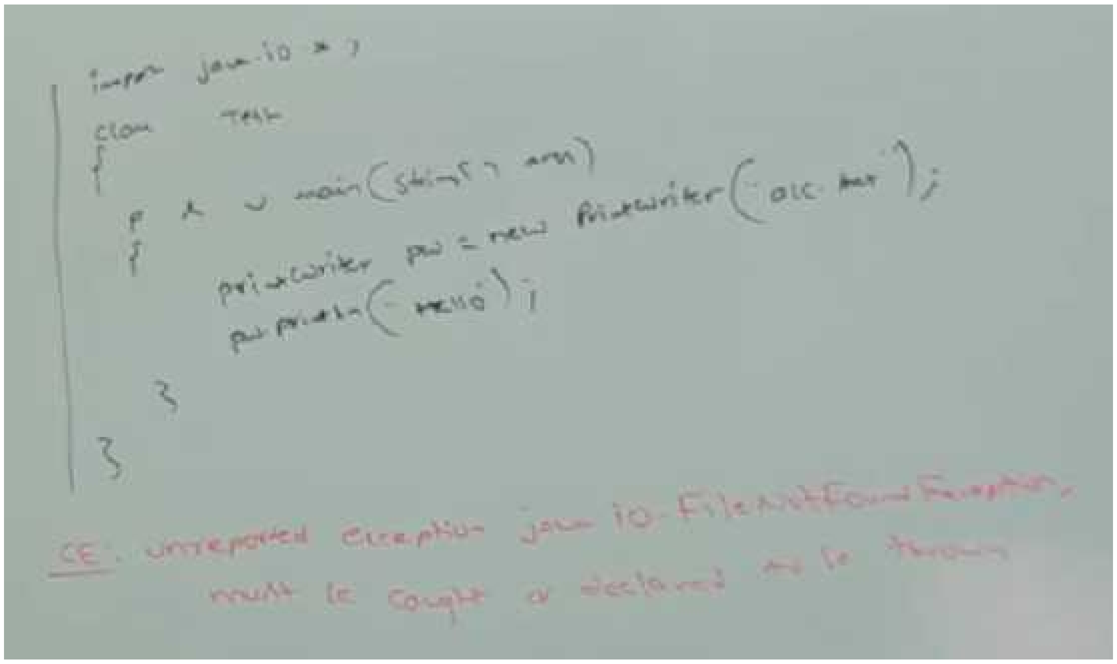
Errors are non recoverable.

e.g. If OutOfMemoryError occors being a programmer we can't do anything and the program will be terminated abnormally.

System Admin ar Server Admin is responsible to increase heap memory.



Checked vs Unchecked Exceptions:



```
import java.io.*;
class Test
{
    p s v main(String[] args)
    {
        PrintWriter pw = new PrintWriter("abc.txt");
        pw.println("hello");
    }
}
```

CE: unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown

The exceptions which are checked by compiler for smooth excecution of the program, are called checked exceptions.

e.g. HallTicketMissingException, PenNotWorkingException, FileNotFoundException, SQLException etc.

In our program if there is a chance of raising checked exception then compulsory we should handle that checked exception(either by try cache or by throws ketword), otherwise we will get compiletime error.

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

e.g. ArithmaticException, NullpointerException, BombBlastException etc.

Note:

Whethr it is checked or unchecked every exception occurs at runtime only. There is no chance of occuring any exception at compile time.

Note:

RuntimeException and it's child classes, error and it's child classes are unchecked. Except these remaining are checked.

Fully Checked vs Partially Checked:

A checked exception is said to be fully checked iff all it's child classes also checked.

e.g. IOException, InterruptedException

A checked exception is said to be partially checked iff some of it's child classes are unchecked.

e.g. Exception, Throwable

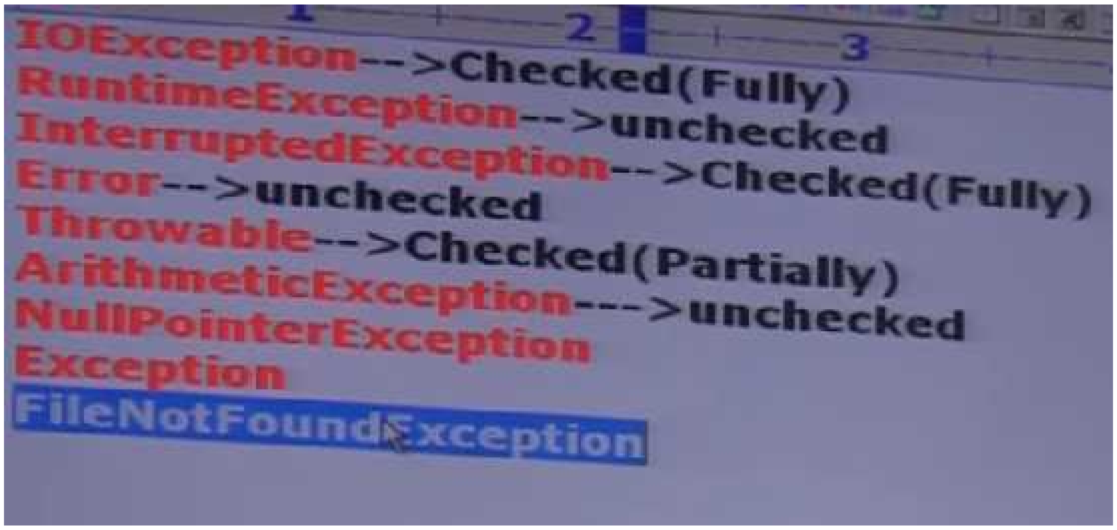
Note:

The only possible partialy checked exceptions in java are

1. Exception

2. Throwable

Q. Describe the behaviour of following exceptions ?

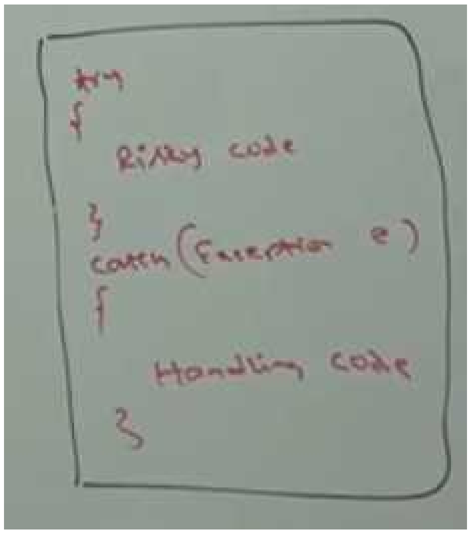


IOException-->Checked(Fully)
RuntimeException-->unchecked
InterruptedException-->Checked(Fully)
Error-->unchecked
Throwable-->Checked(Partially)
ArithmeticException--->unchecked
NullPointerException
Exception
FileNotFoundException

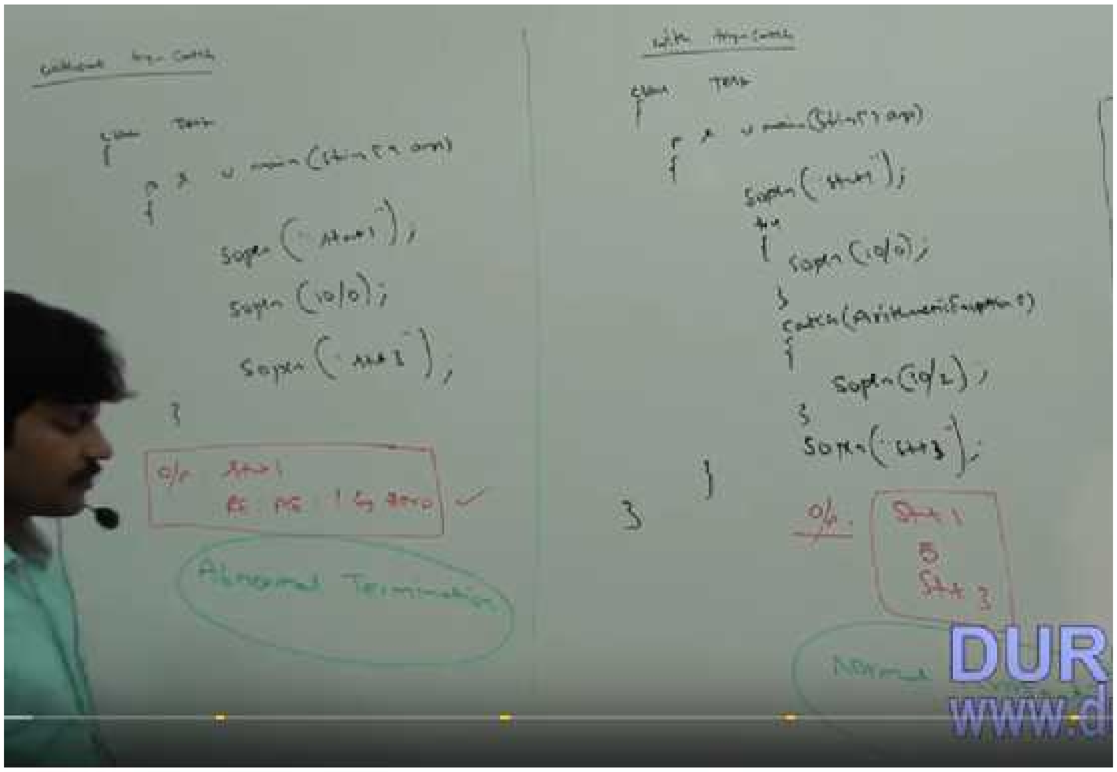
Customized Exception Handling by using try catch:

It is highely recomended to handle exceptions.

The code which may raise an exception is called riskey code and we have to define that code inside try block and corrosponding 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(String[] args)
  {
      Sopln("stmt1");
      Sopln(10/0);
      Sopln("stmt3");
  }
```

```
o/p  stmt1
     RE: AE: / by zero
```

Abnormal Termination

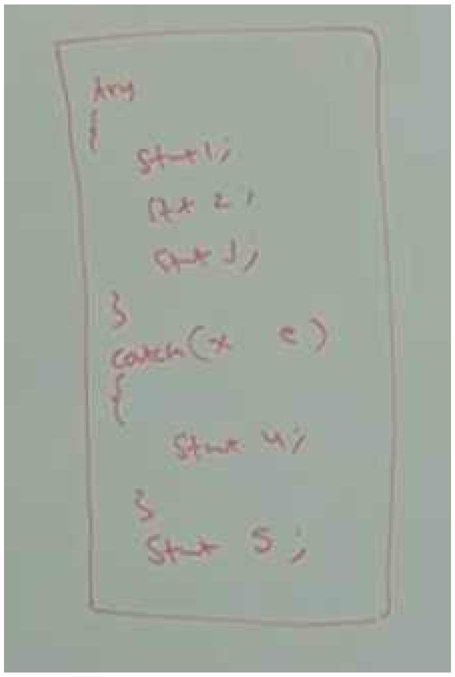
with try-catch

```
class Test
{
  p s v main(String[] args)
  {
      Sopln("stmt1");
      try
      {
        Sopln(10/0);
      }
      catch(ArithmeticException e)
      {
         Sopln(10/2);
      }
      Sopln("stmt3");
  }
}
```

```
o/p  stmt1
     5
     stmt 3
```

Normal

Control flow in try catch:



```
try
{
   Stmt1;
   Stmt 2;
   Stmt 3;
}
catch(X   e)
{
     Stmt 4;
}
Stmt 5;
```

Case 1: If there is no exception

If there is exception raised in statement 4 or 5, then there is abnormal termination.

Note:

Within try block if anywhere exception raised then rest of the try block won't be executed even though we handled that exception. Hence within the try block we have to take only riskey code and length of try block should be as less as possible.

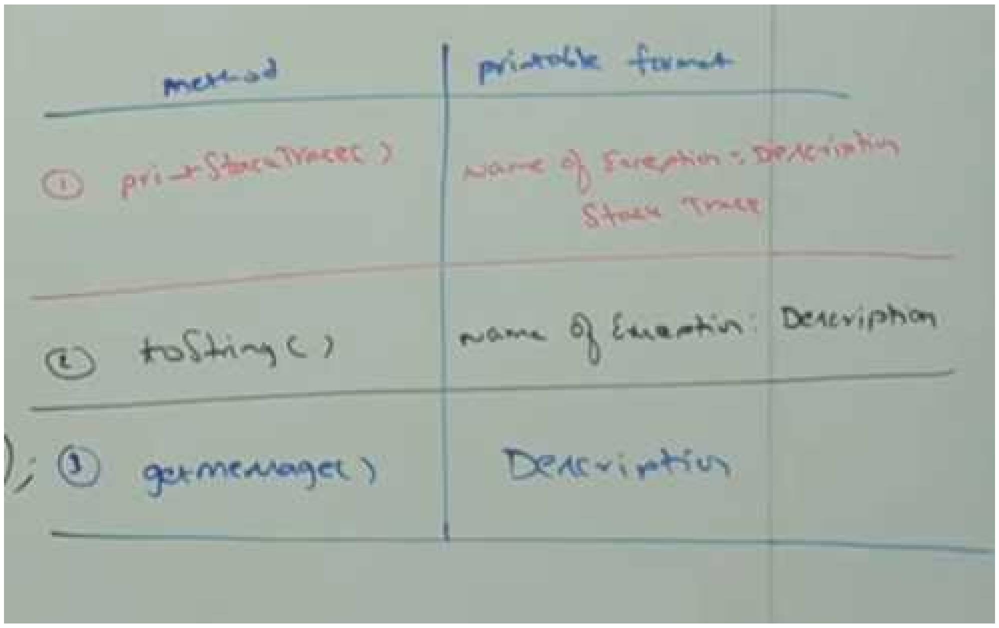
Note 2:

In addition to try block there may be a chance of raising an exception inside catch and finally blocks also.

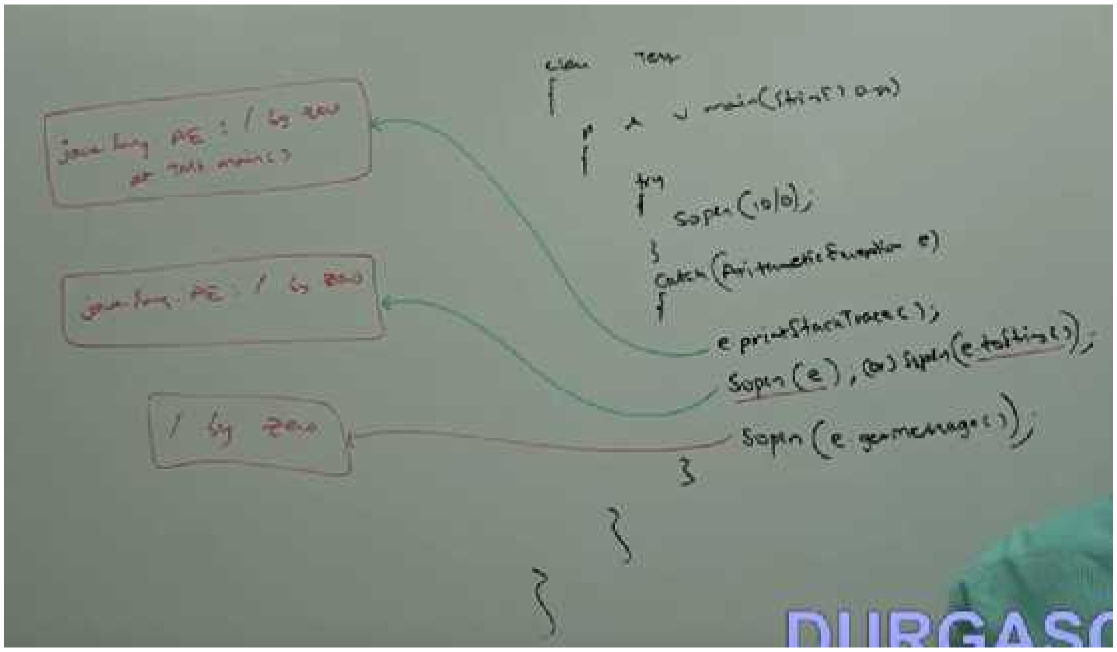
If any statement which is not part of try block and raises an exception then it is always abnormal termination.

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



```
class Test
{
  p s v main(String[] args)
  {
    try
    {
      Sopln(10/0);
    }
    catch(ArithmeticException e)
    {
      e.printStackTrace();
      Sopln(e); (or) Sopln(e.toString());
      Sopln(e.getMessage());
    }
  }
}
```

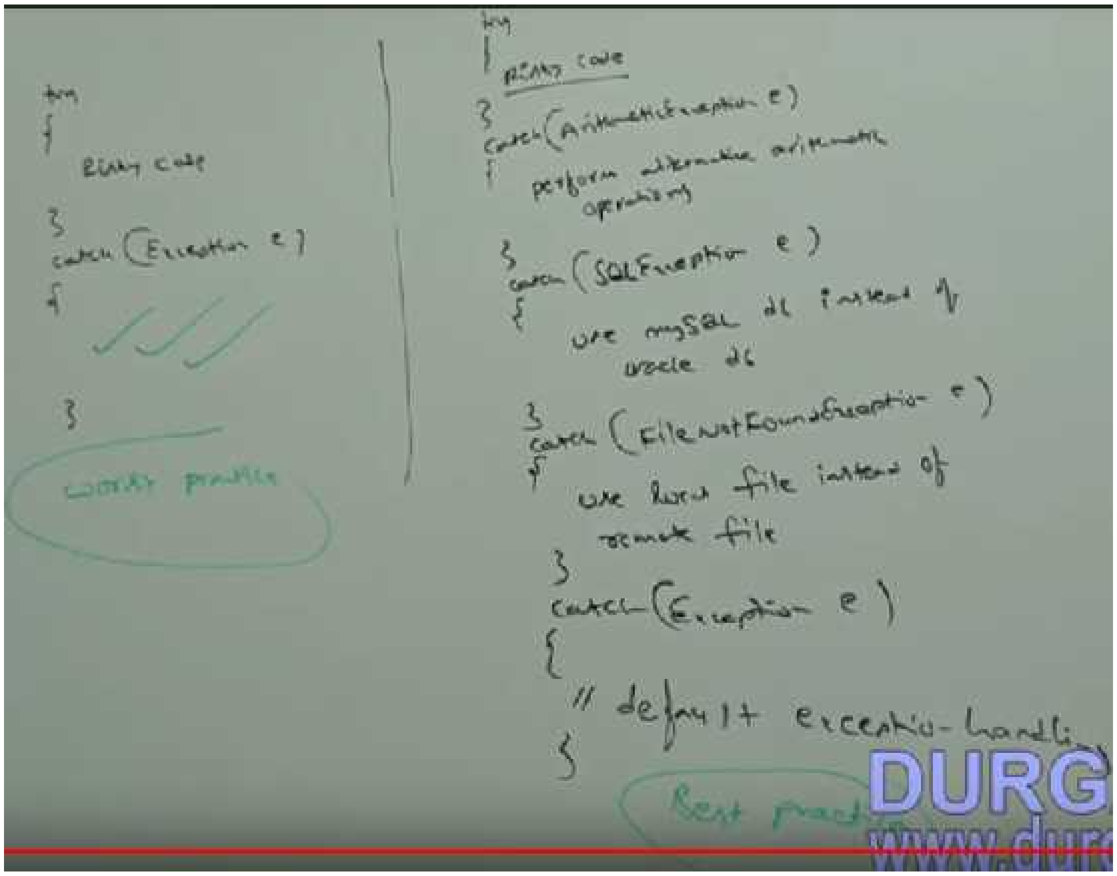
```
java.lang.AE : / by zero
  at Test.main()
```

```
java.lang.AE : / by zero
```

```
/ by zero
```

try with multiple catch blocks:

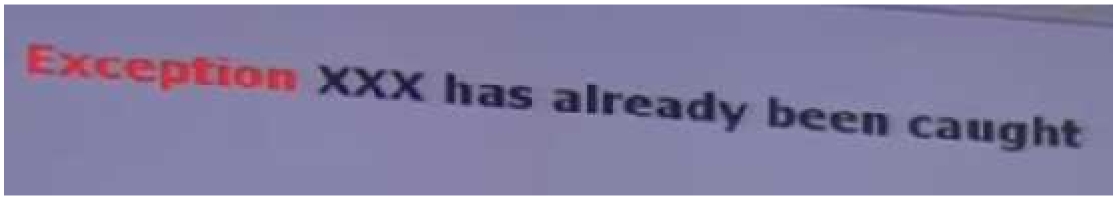
The way of handling an exception is varied from exception to exception hence for every exception type it is highely recomended to take separate catch block i.e. try with multiple catch block is always possible and recomended to use.



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

```
try
{
    Risky code
}
catch(ArithmeticException e)
{
    perform alternative arithmetic
        operations
}
catch (SQLException e)
{
    use mySQL db instead of
        oracle db
}
catch (FileNotFoundException e)
{
    use local file instead of
        remote file
}
catch(Exception e)
{
    // default exception handli
}
```

If try with multiple catch blocks present then the order of catch blocks is very important. We have to take child first and then parent otherwise we will get compile time error saying



Exception XXX has already been caught

final, filally, finalize:

final is a modifier applicable for classes, methods and variables.

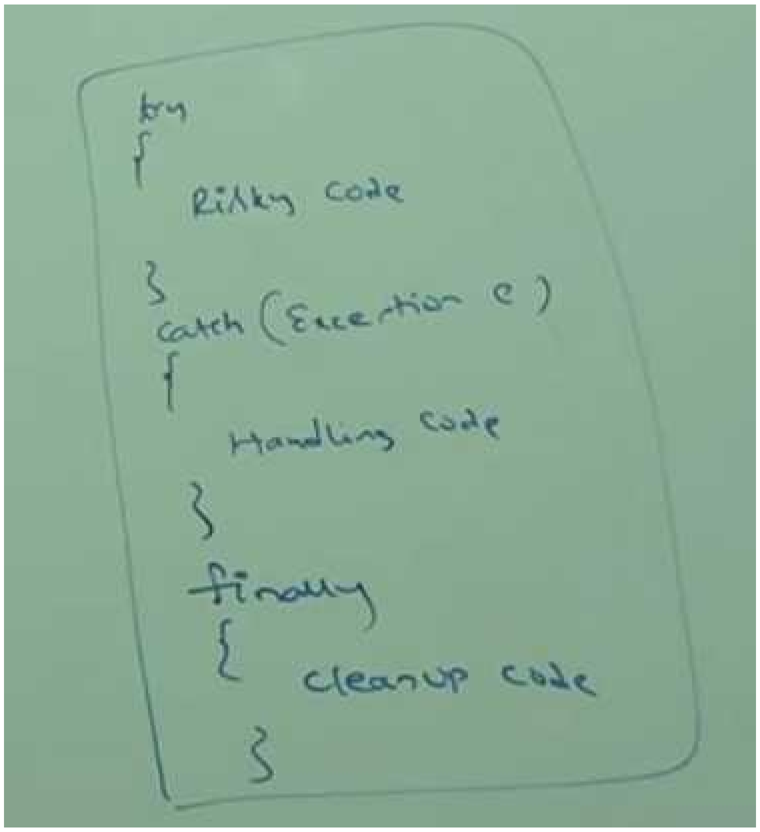
If a class declared as final then we can't extend that class. i.e. we can't create child class for that class i.e. inheritance is not possible for final classes.

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

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

finally:

finally is a block always associates with try catch to maintain cleanup code.



```
try
{
   Risky code

}
catch (Exception e )
{
    Handling code
 }
 finally
  {   cleanup code
   }
```

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

finalize():

finalize(), is a method always invoked by GC, just befor destroying an object to perform clean up activities.

Once finalize(), method completes immediately GC destroyes that object.

Note:

finally block is responsible to perform cleanup activities realted to try block. i.e. whatever resources we opened at the part of try block will be closed inside finally block.

whereas finalize method is responsible to perform clean up activities related to Object. i.e. whatever resources associated with object will be deallocated before destroying an 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 compiletime error i.e. try without catch or finally is invalid.

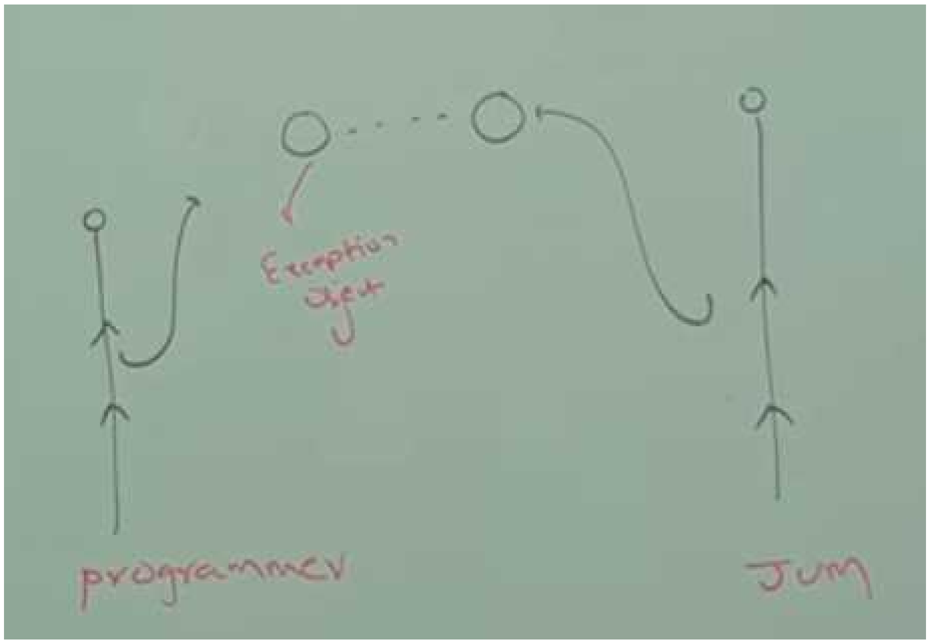
Whenever we are writting catch block compulsory try block must be required. i.e. catch without try is invalid.

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

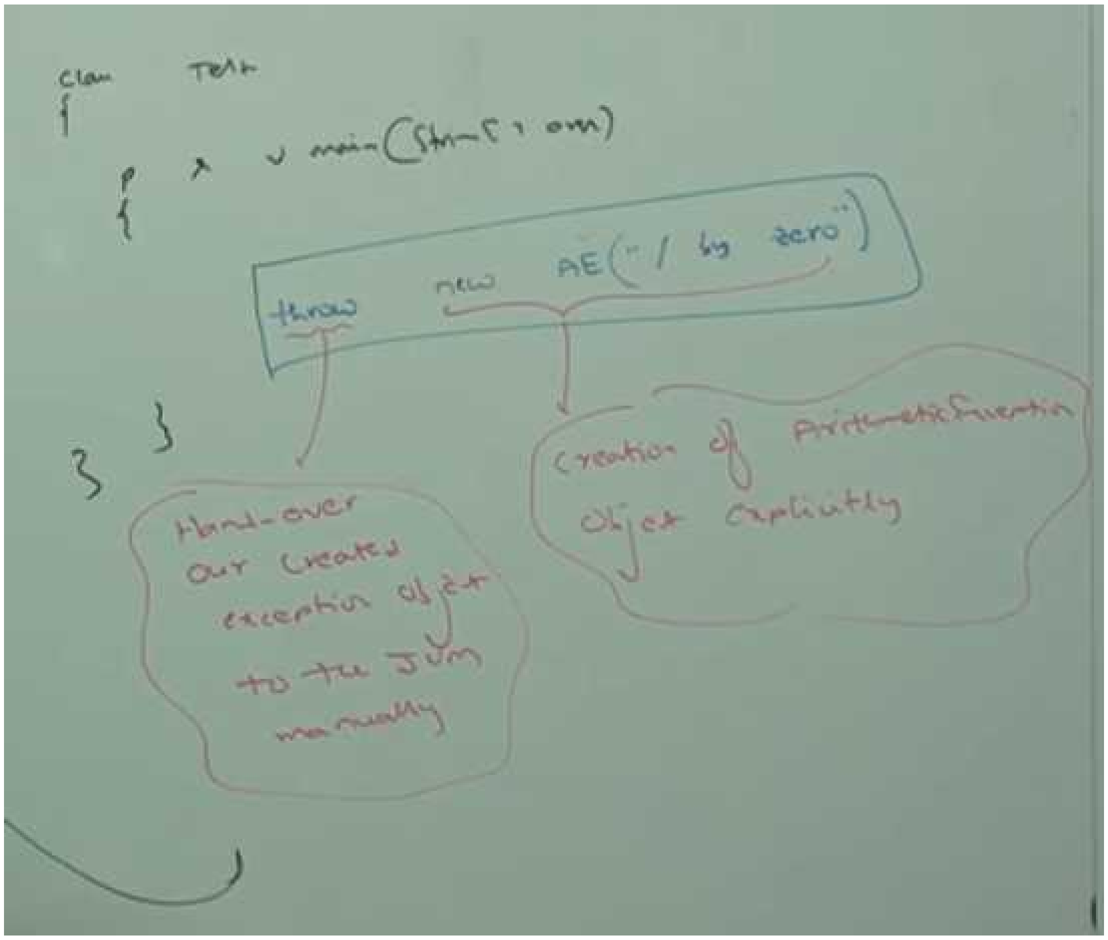
Inside try catch and finally blocks we can delcare try catch and finally blocks i.e. nesting of try catch finally alwas possible.

For try catch and finally blocks curley braces are mandatory.

throw keyword:

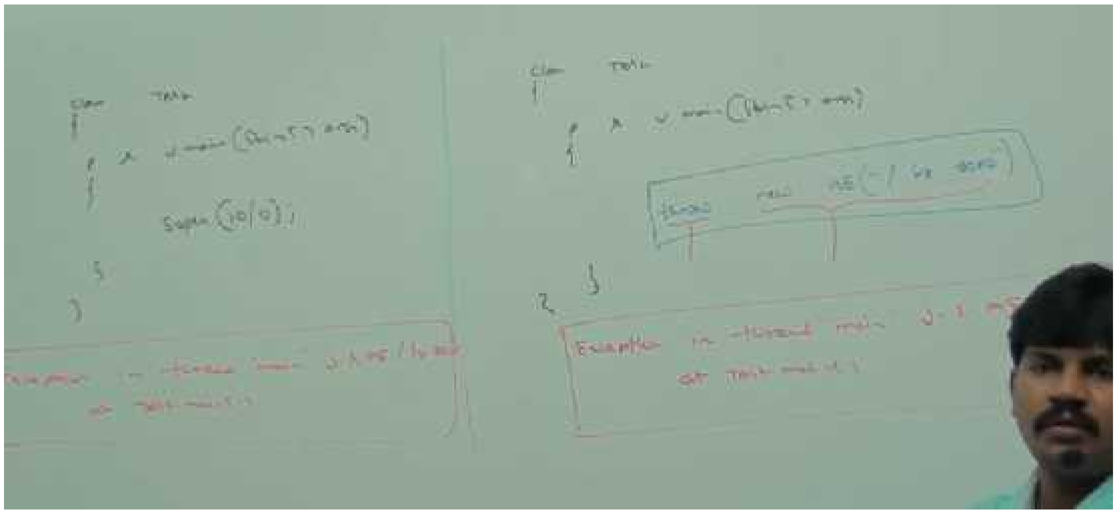


Sometimes we can create exception object explicitely and hand over to the JVM manually.For this we have to use throw keyword.


class Test
{
p s v main(String[] args)
{
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 the JVM manually.

Hence the reult of following two programs are exactly same

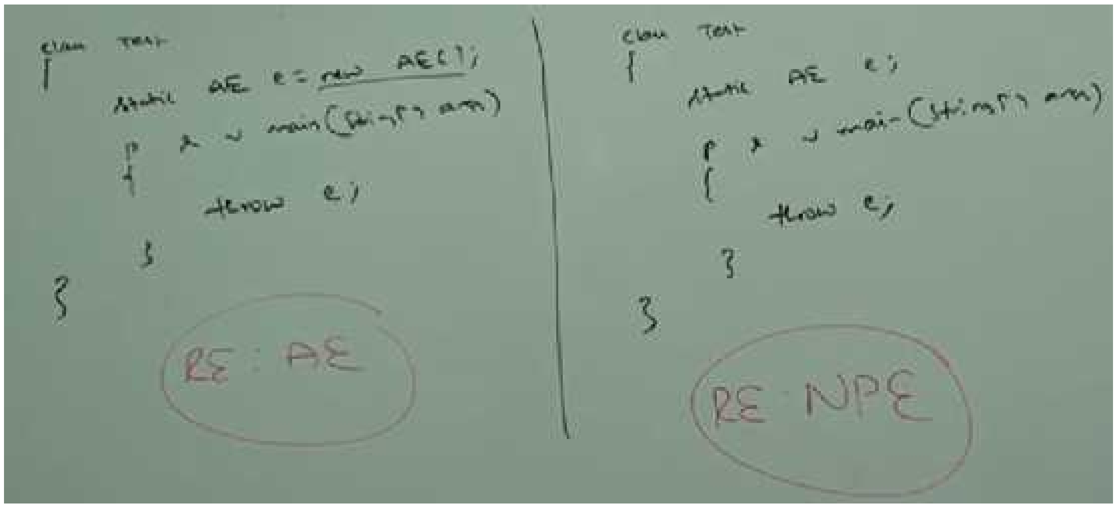

class Test
{
p s v main(String[] args)
{
Sopln(10/0);
}
}
Exception in thread main j.l.AE: / by zero
at Test.main()
class Test
{
p s v main(String[] args)
{
throw new AE("/ by zero")
}
}
Exception in thread main j.l.AE: / by zero
at Test.main()

In 1st case main(), method is responsible to create exception object and handover to the JVM, but in 2nd case programmer is creating exception object explicitely and handover to the JVM manually.

Note:

Best use of throw keyword is for user defined exceptions or customized exceptions.

Case 1:



```
class Test
{
    static AE e = new AE();
    p s v main(String[] args)
    {
        throw e;
    }
}
```

RE : AE

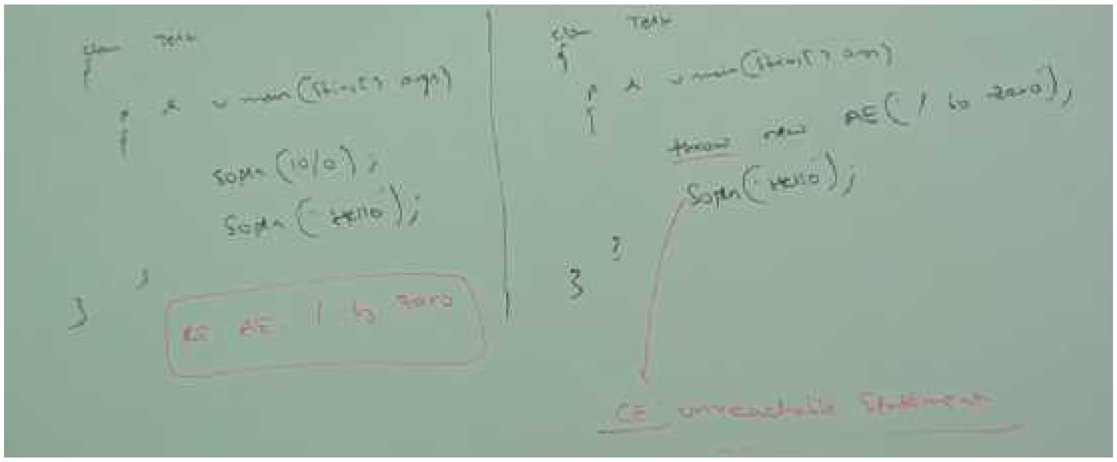
```
class Test
{
    static AE e;
    p s v main(String[] args)
    {
        throw e;
    }
}
```

RE : NPE

if e refers null then we will get nullPointer Exception.

Case 2:

After throw statement we are not allowed to write any statement directly. otherwise we will get compile time error saying unreachable statement.



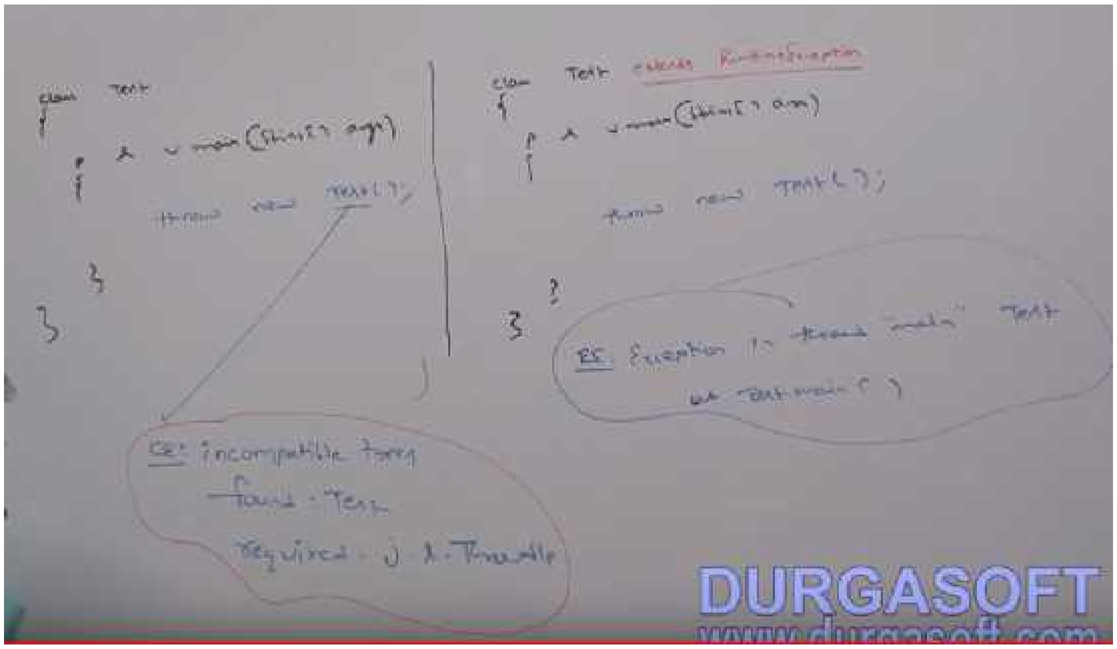
```
class Test
{
    p s v main(String[] args)
    {
        sopln(10/0);
        sopln("hello");
    }
}
```

RE AE / by zero

```
class Test
{
    p s v main(String[] args)
    {
        throw new AE("/ by zero");
        sopln("hello");
    }
}
```

CE unreachable statement

Case 3:



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

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

```
class Test extends RuntimeException
{
  p s v main(String[] args)
  {
    throw new Test();
  }
}
```

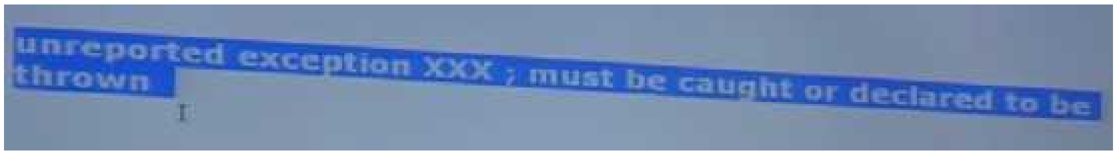
RE: Exception in thread "main" Test
at Test.main()

DURGASOFT
www.durgasoft.com

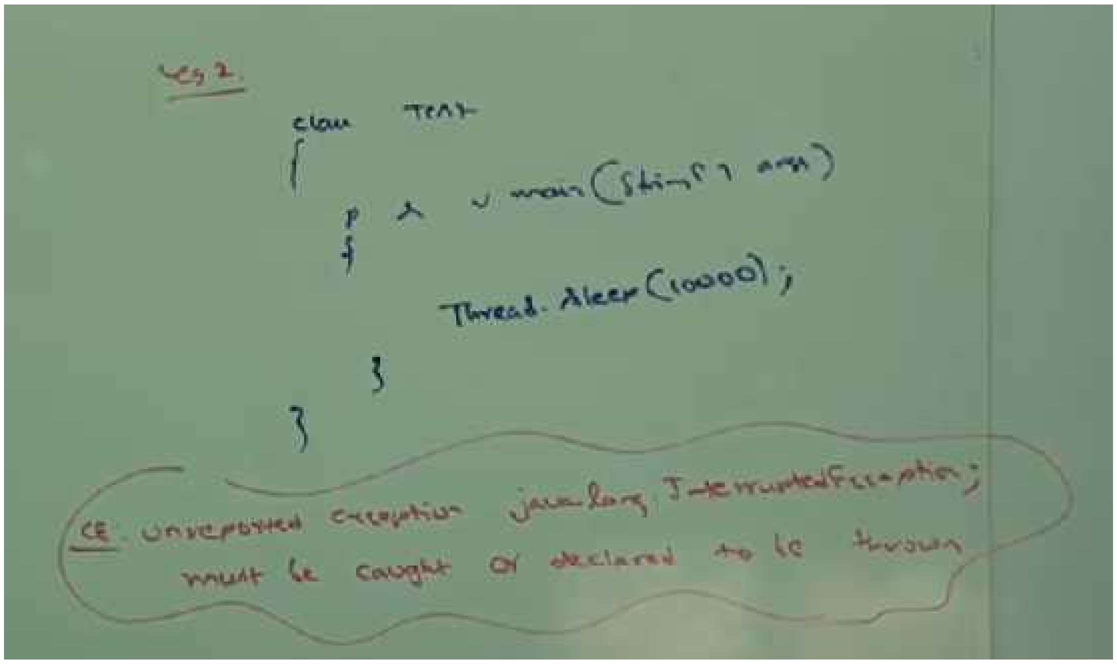
We can use throw keyword only for thowable types. If we are trying to use for normal java objects we will get compile time error saying

throws keyword:

In our program if there is a possibility of raising checked exception then compulsory we should handle that checked exception otherwise we will get compiletime error saying



```
unreported exception XXX ; must be caught or declared to be
thrown
```



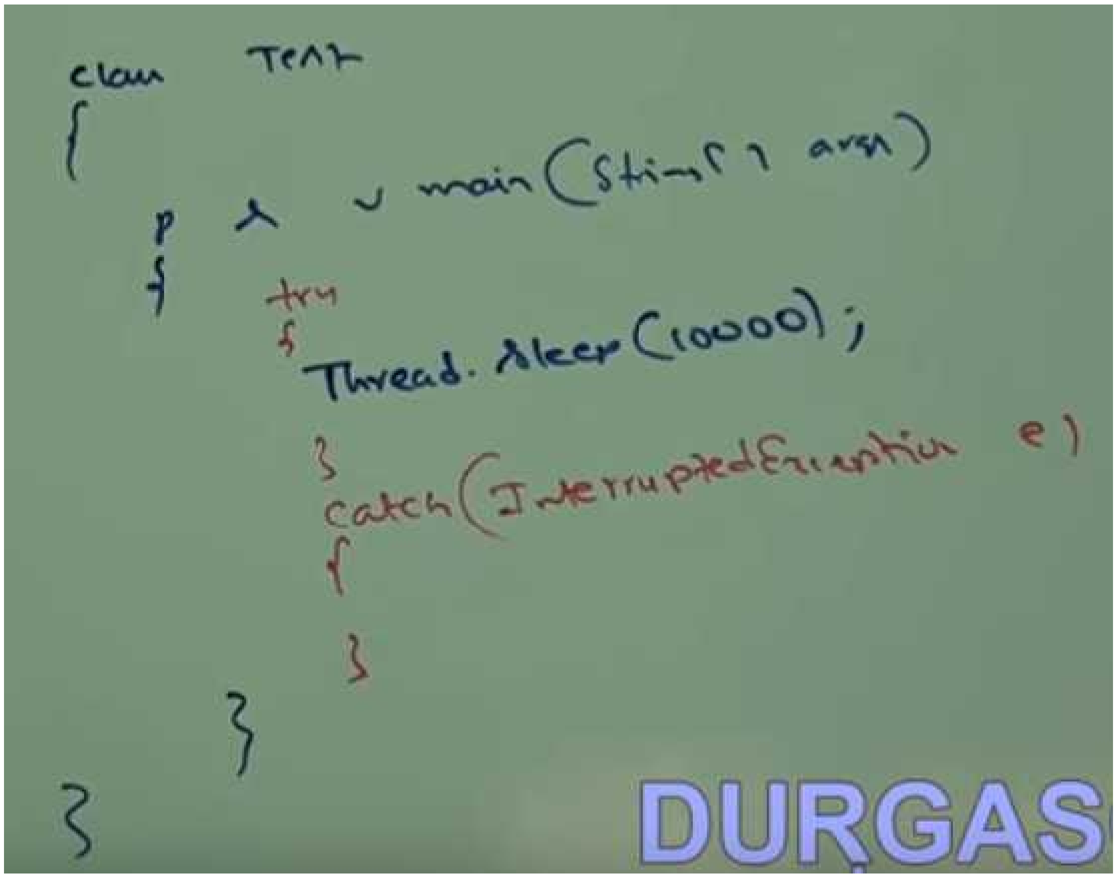
Eg 2.

```
class Test
{
    p s v main(String[] args)
    {
        Thread.sleep(10000);
    }
}
```

CE: unreported exception java.lang.InterruptedException; must be caught or declared to be thrown

We can handle this compiletime error by using the following two ways,

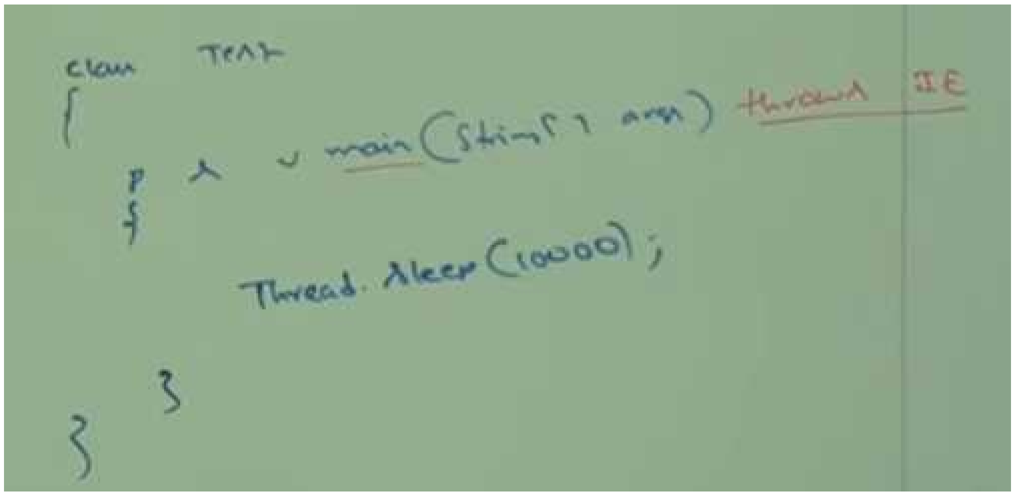
1. By using try catch



```
class   Test
{
    p  s  v  main(String[] args)
    {
        try
        {
          Thread.sleep(10000);
        }
        catch(InterruptedException  e)
        {

        }
    }
}
```

We ca use throws keyword to delegate responsibility of excepion handling to the caller, then caller(it may be another method or JVM).

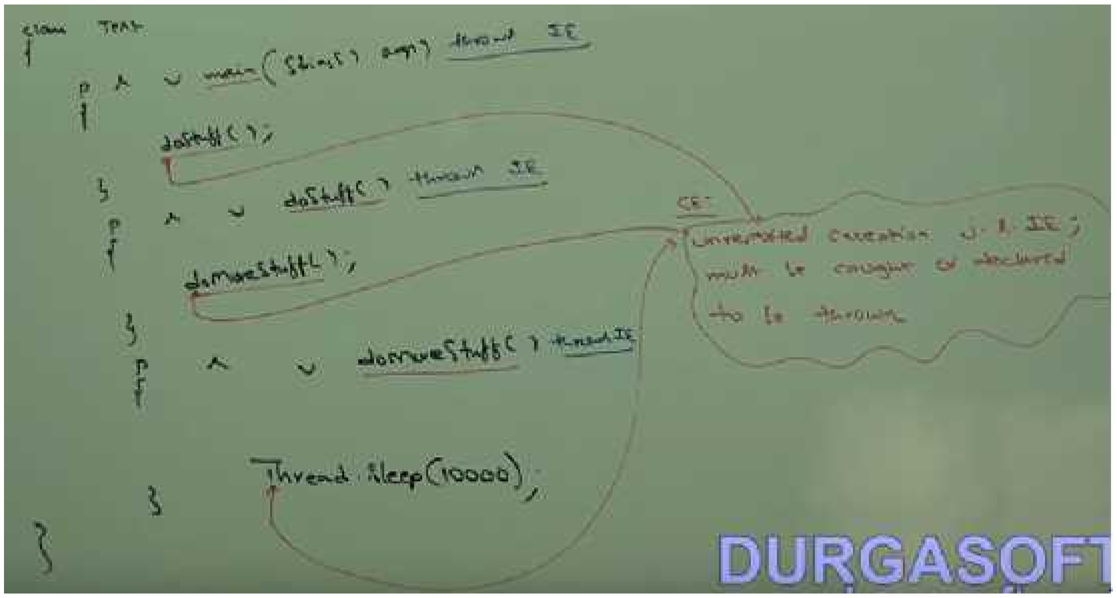
Then caller method is responsible to handle that exception.



```
class   Test
{
    p  s  v  main(String[] args)  throws  IE
    {
        Thread.sleep(10000);

    }
}
```

Throws keyword requires only for checked exceptions and usages of throws keyword for unchecked exceptions there is no use or impact.

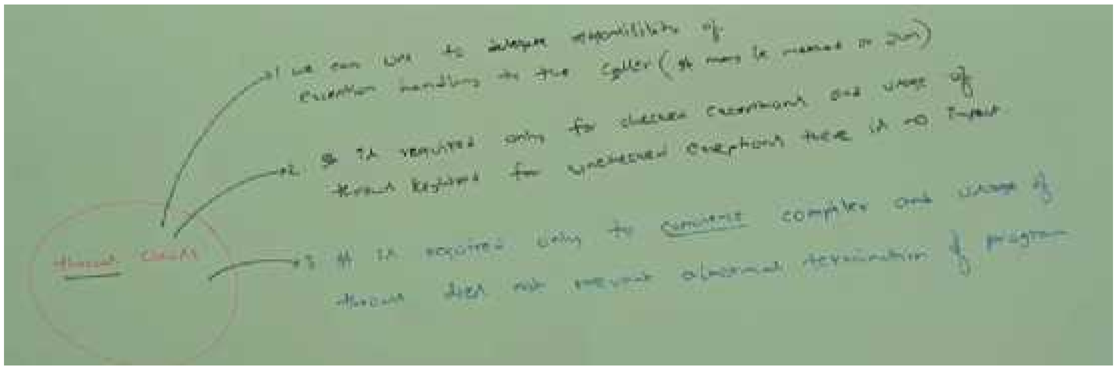
throws keyword required only to convince compiler and usages of throws keyword does not prevent abnormal termination of the program.



```
class Test
{
    p s v main(String[] args) throws IE
    {
        doStuff();
    }
    p s v doStuff() throws IE
    {
        doMoreStuff();
    }
    p s v doMoreStuff() throws IE
    {
        Thread.sleep(10000);
    }
}
```

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

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



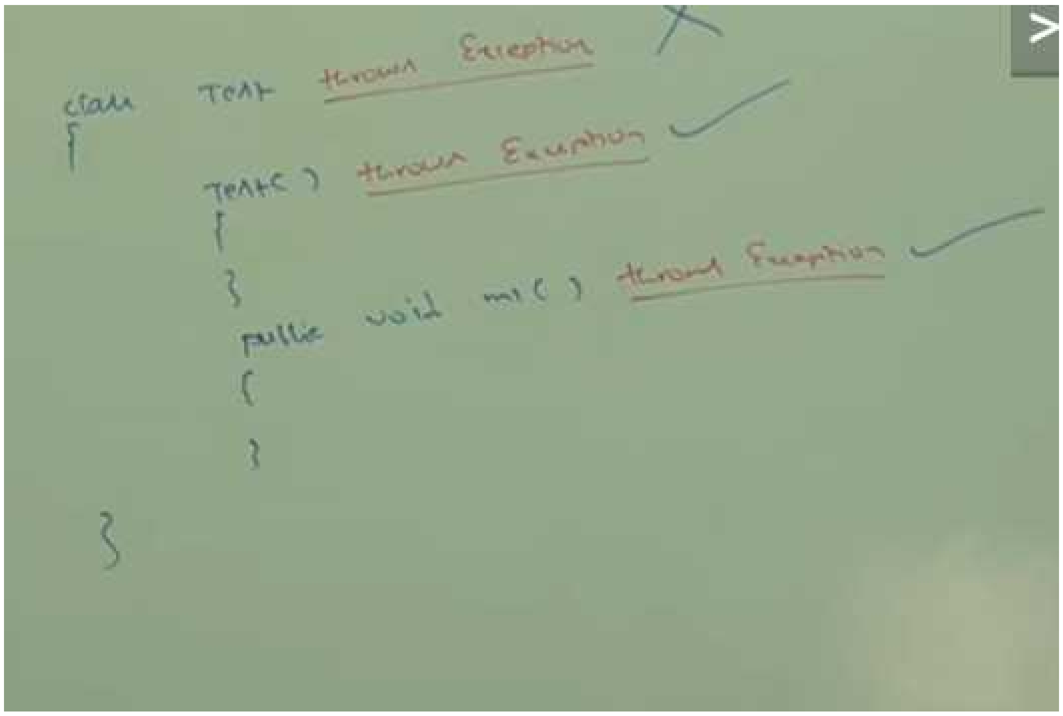
throws clause

1. we can use to delegate responsibility of exception handling to the caller (it may be method or JVM)
2. it is required only for checked exceptions and usage of throws keyword for unchecked exceptions there is no impact
3. it is required only to convince compiler and usage of throws does not prevent abnormal termination of program

It is recomended to use try catch over throws keyword.

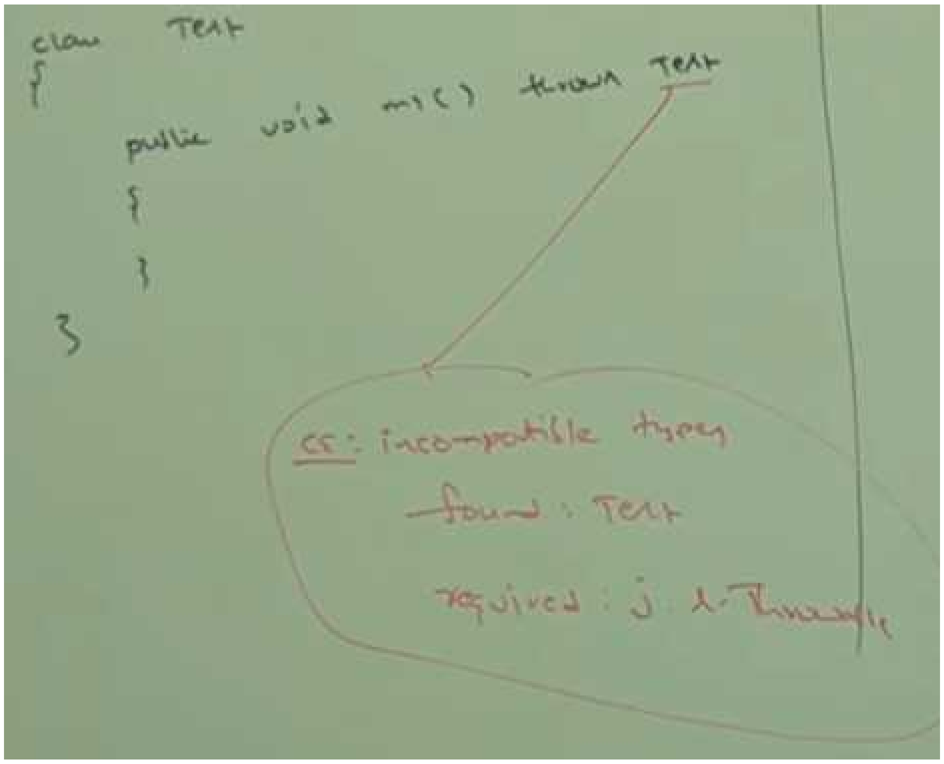
CASE 1

We can use throws keywod for methods and constructors but not for classes.



```
class Test throws Exception
{
    Test() throws Exception
    {
    }
    public void m1() throws Exception
    {
    }
}
```

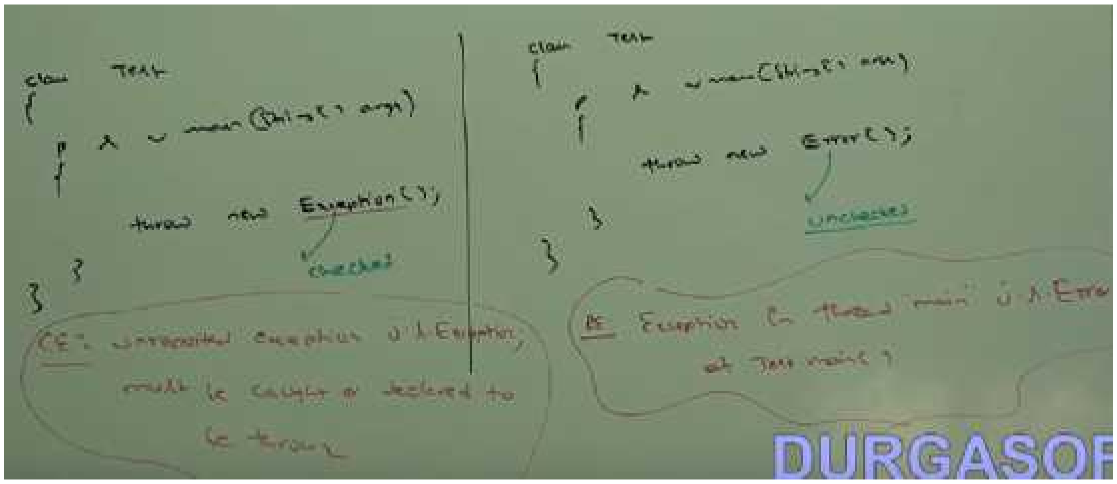
We can use throws keyword only for throwable types. If we are trying use for normal java classes then we will get compiletime error saying



```
class Test
{
    public void m1() throws Test
    {
    }
}
```

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

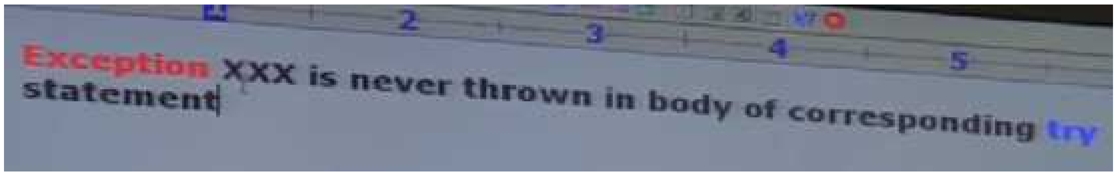
Case 3;



```
class Test
{
    p s v main(String[] args)
    {
        throw new Exception();
                  checked
    }
}
CE: unreported exception java.lang.Exception;
    must be caught or declared to
    be thrown
```

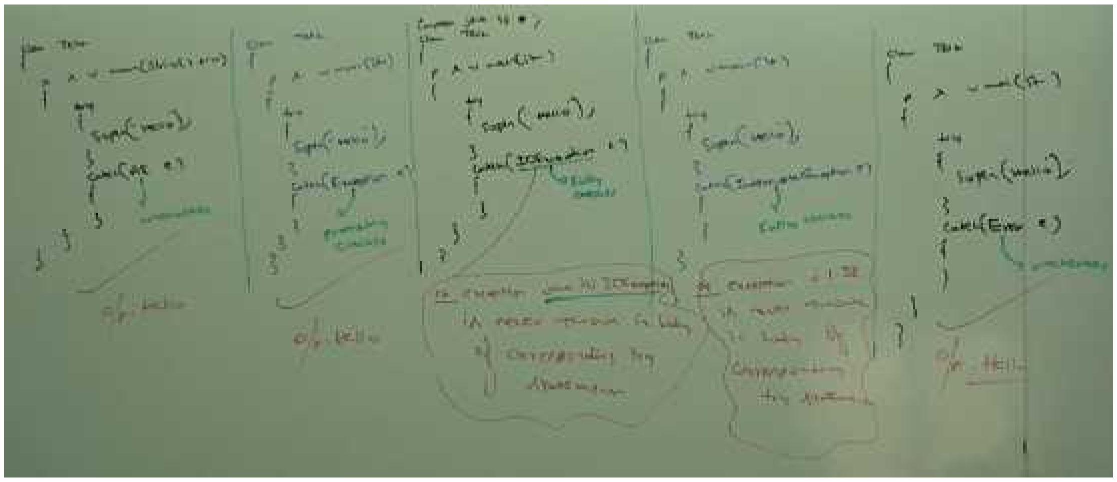
```
class Test
{
    p s v main(String[] args)
    {
        throw new Error();
                  unchecked
    }
}
RE: Exception in thread "main" java.lang.Error
    at Test.main()
```

Witin in the try block if there is no chance of raising an exception then we can't write catch block for that exception otherwise we will get compiletime error saying



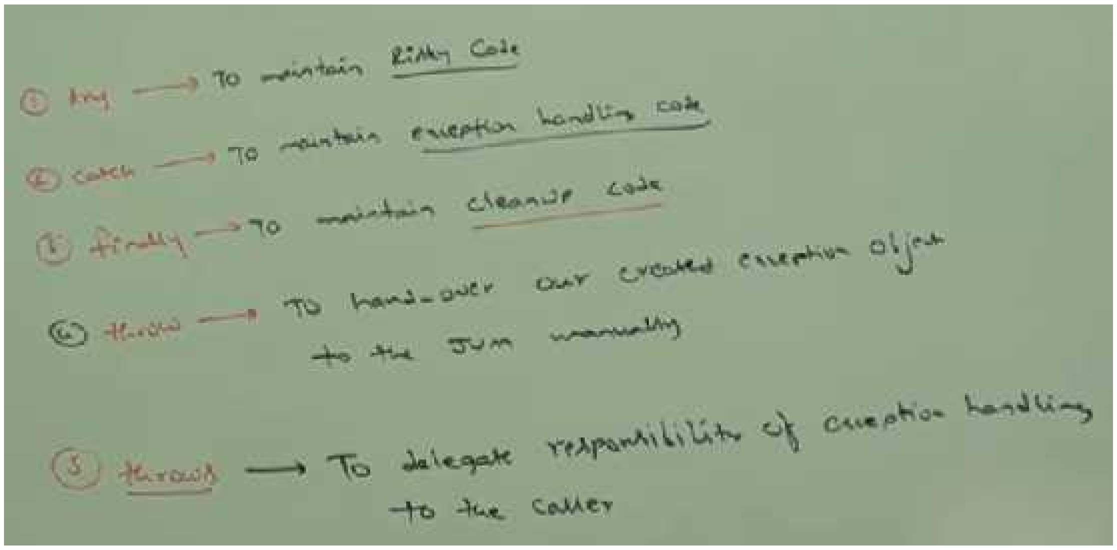
Exception XXX is never thrown in body of corresponding try statement

This rule only applicable for fullychecked exceptions.



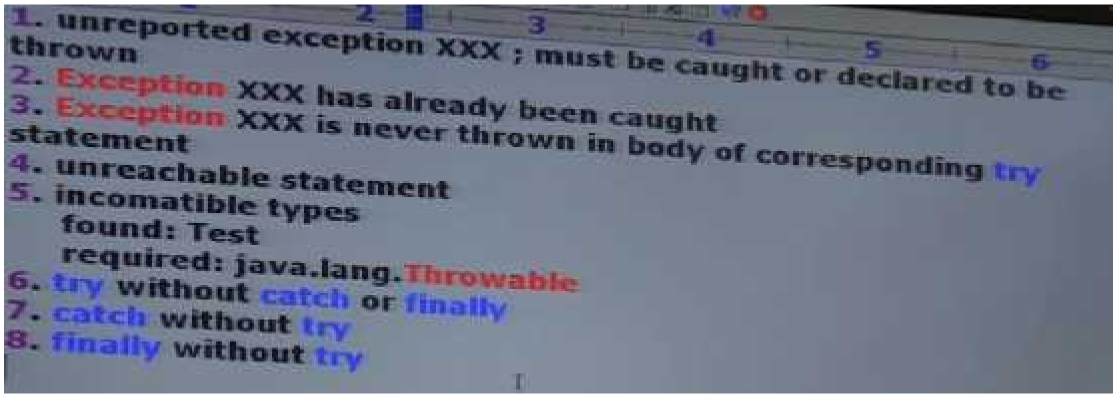
class Test
unchecked
partially checked
fully checked
CE: exception java.io.IOException is never thrown in body of corresponding try statement
CE: exception is never thrown in body of corresponding try statement
unchecked
o/p: Hello
o/p: Hello
o/p: Hello

Exception handling keywords Summary:



1. try → To maintain Risky Code
2. catch → To maintain exception handling code
3. finally → To maintain cleanup code
4. throw → To hand-over our created exception object to the JVM manually
5. throws → To delegate responsibility of exception handling to the caller

Various possible compiletime 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 statement
4. unreachable statement
5. incomatible types
   found: Test
   required: java.lang.Throwable
6. try without catch or finally
7. catch without try
8. finally without try

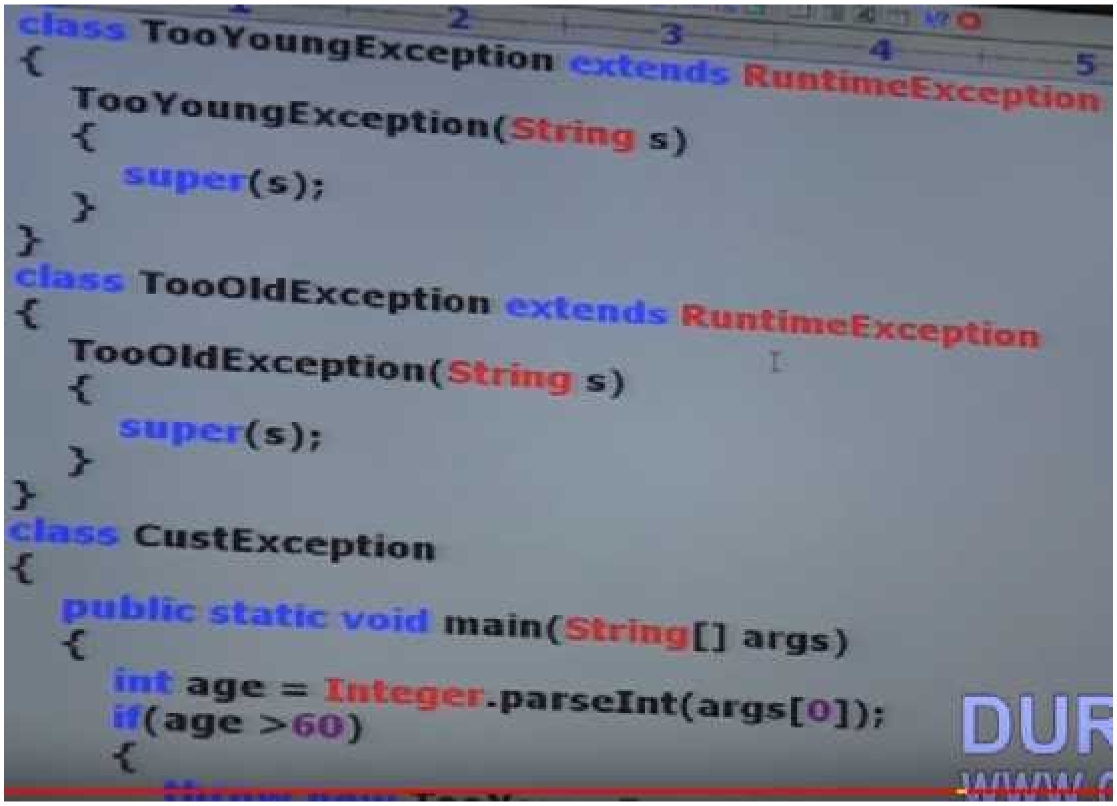
Customized or userdefined exceptions:

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

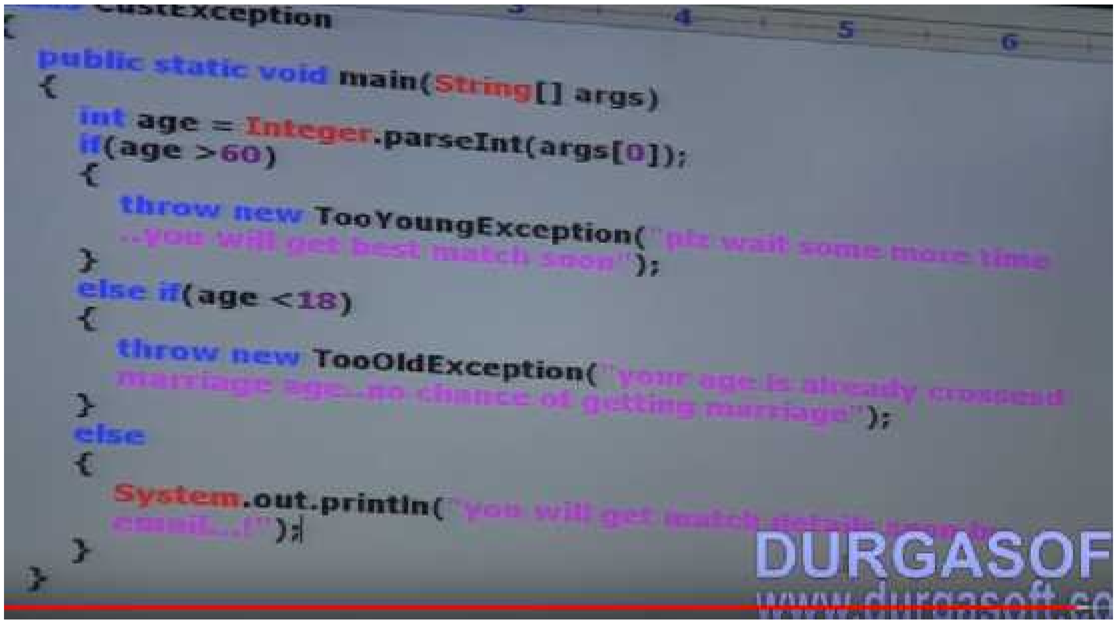
TooYoungException

TooOldException

InSufficientFundsException etc..



```java
class TooYoungException extends RuntimeException
{
   TooYoungException(String s)
   {
     super(s);
   }
}
class TooOldException extends RuntimeException
{
   TooOldException(String s)
   {
     super(s);
   }
}
class CustException
{
   public static void main(String[] args)
   {
     int age = Integer.parseInt(args[0]);
     if(age >60)
     {
```



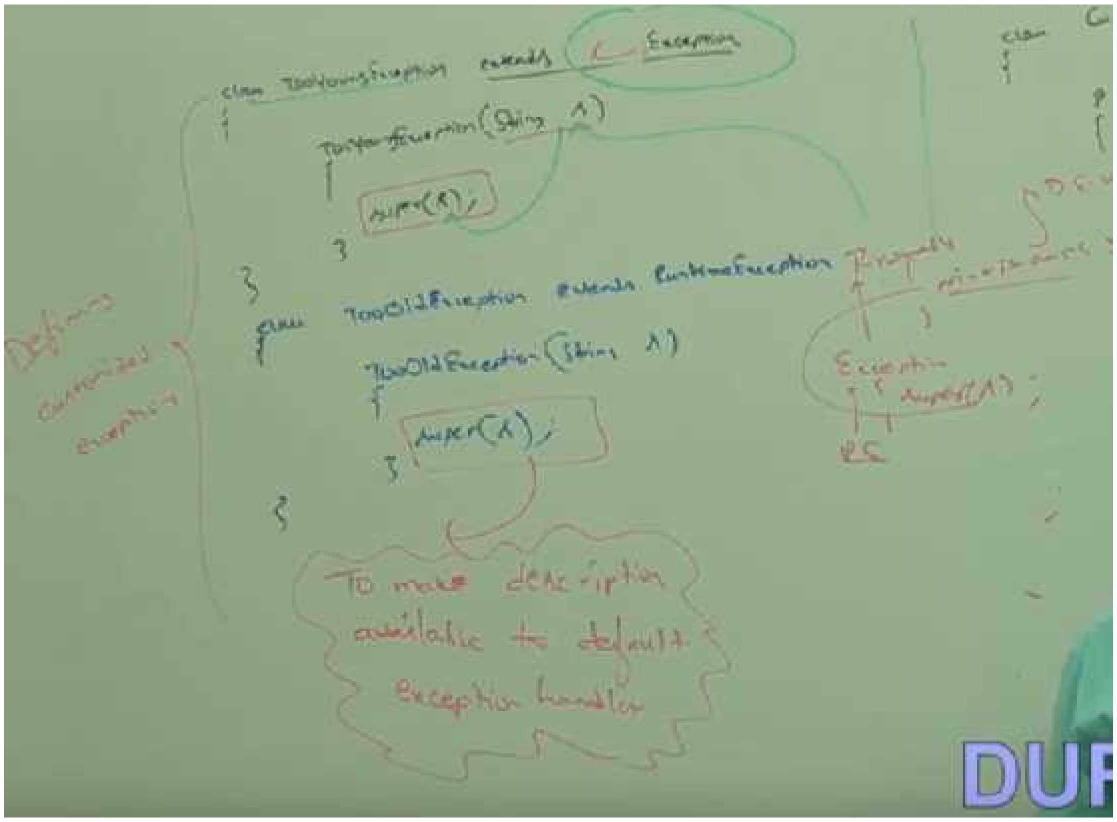
```java
  public static void main(String[] args)
  {
     int age = Integer.parseInt(args[0]);
     if(age >60)
     {
       throw new TooYoungException("plz wait some more time
       ..you will get best match soon");
     }
     else if(age <18)
     {
       throw new TooOldException("your age is already crossed
       marriage age..no chance of getting marriage");
     }
     else
     {
       System.out.println("you will get match details soon by
       email...!");
     }
  }
```

Note:

throw keyword is best suitable for userdefined or customized exceptions but not for pre defined exceptions.

It is highely recomended to define customized exceptions as unchecked i.e. we have to extends

RuntimeExcption but not Exception.



```
class TooYoungException extends Exception
{
    TooYoungException(String s)
    {
        super(s);
    }
}
class TooOldException extends RuntimeException
{
    TooOldException(String s)
    {
        super(s);
    }
}
```

Defining customized exceptions
To make description available to default exception handler
Exception
RE
DUR

Top-10 Exceptions in java:

Based on the persion who is raising an exception all exceptions are divided into 2 categories,

1. JVM Exceptions

2. Programmatic Exceptions

JVM Exceptions:

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

e.g. Arithmatic Exceptions, NullPointer Exception etc..

Programmatic Exceptions:

The exceptions which are raised explicitly either by programmer or by API developer to indicate that something goes wrong are called Programmatic exceptions.

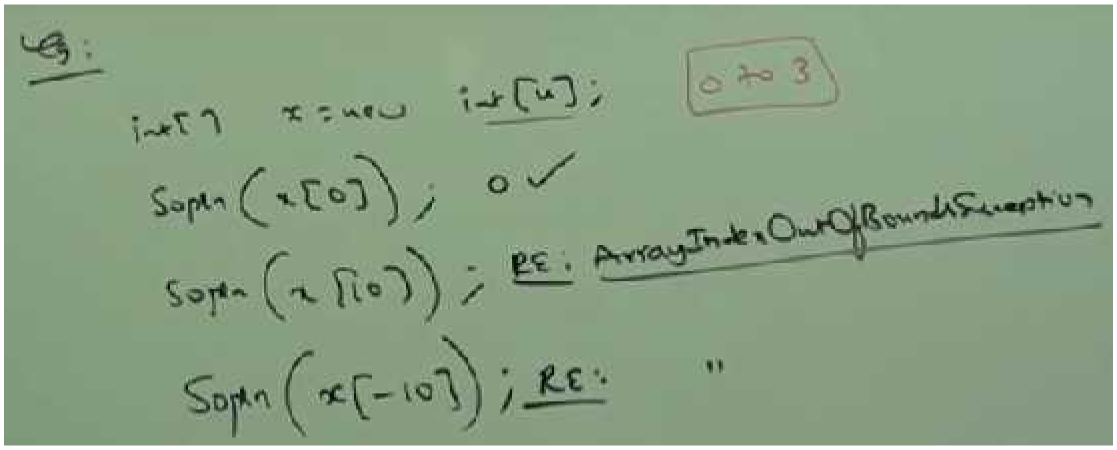
e.g. TooOldException, IlligalArgumentException etc..

ArrayInexOutOfBoundException:

It is the child class of RuntimeException and hence it is unchecked.

Raised automatically by JVM whenever we are trying to access array element with out of range index.

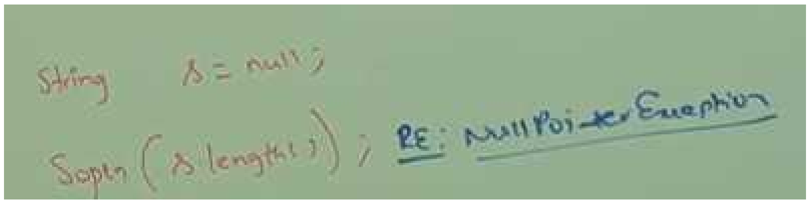
e.g.



```
eg:
int[] x = new int[4];              0 to 3
Sopln(x[0]); 0 ✓
Sopln(x[10]); RE: ArrayIndexOutOfBoundsException
Sopln(x[-10]); RE:     "
```

NullPointer Exception:

It is the child class of RuntimeException and hence it is unchecked. Raised automatically by JVM whenever we are trying to perform any operation on null.

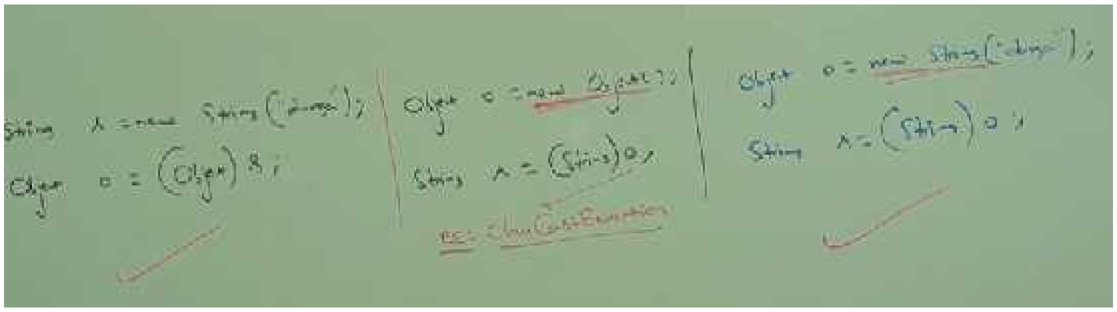


```
String s = null;
Sopln(s.length()); RE: NullPointerException
```

ClassCastException:

It is the child class of RuntimeException and hence it is unchecked.

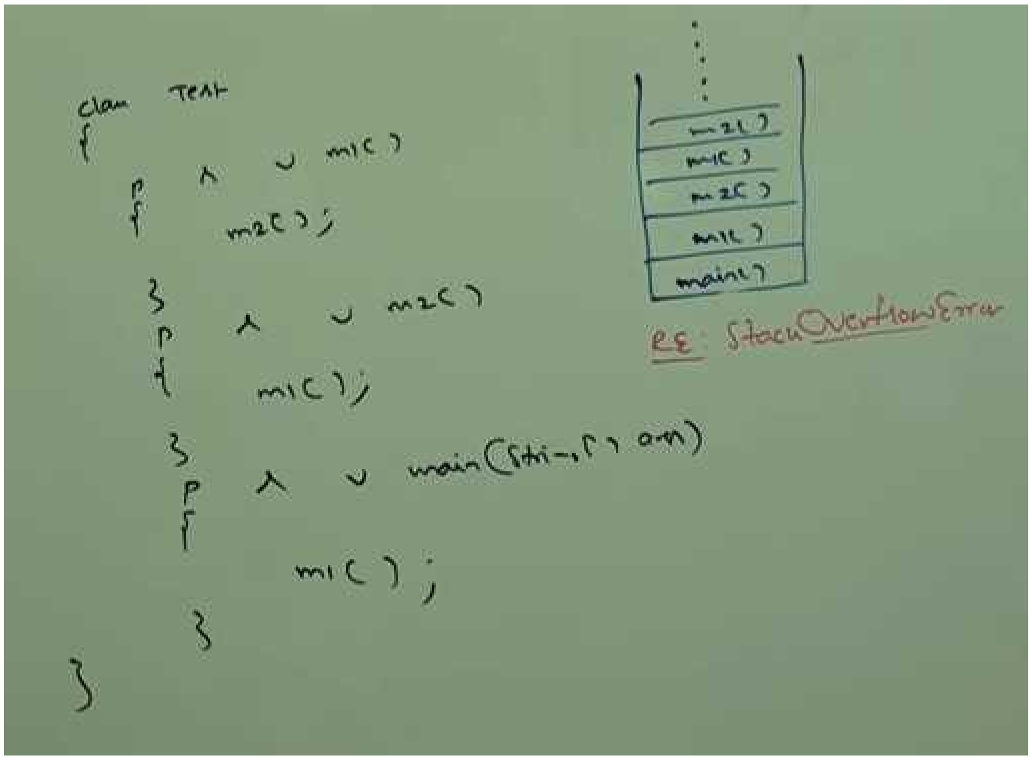
Raised automatically by JVM whenever we are trying to typecast parent object to child type.



```
String s = new String("[illegible]");
Object o = (Object) s;
✓

Object o = new Object();
String s = (String) o;
RE: ClassCastException

Object o = new String("[illegible]");
String s = (String) o;
✓
```

StackOverFlowError:

It is the child class of error and hence it is unchecked. Raised automatically by JVM whenever we are trying to perform recursive method call.



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

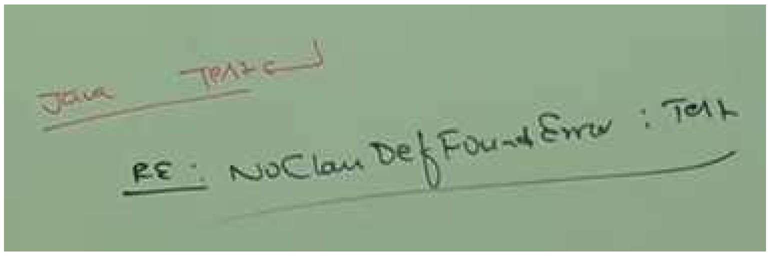
NoClassDefFoundError:

It is the child class of error and hence it is unchecked.

Raised automaticaly by JVM, whenever JVM unaable to find required .class file.

e.g.

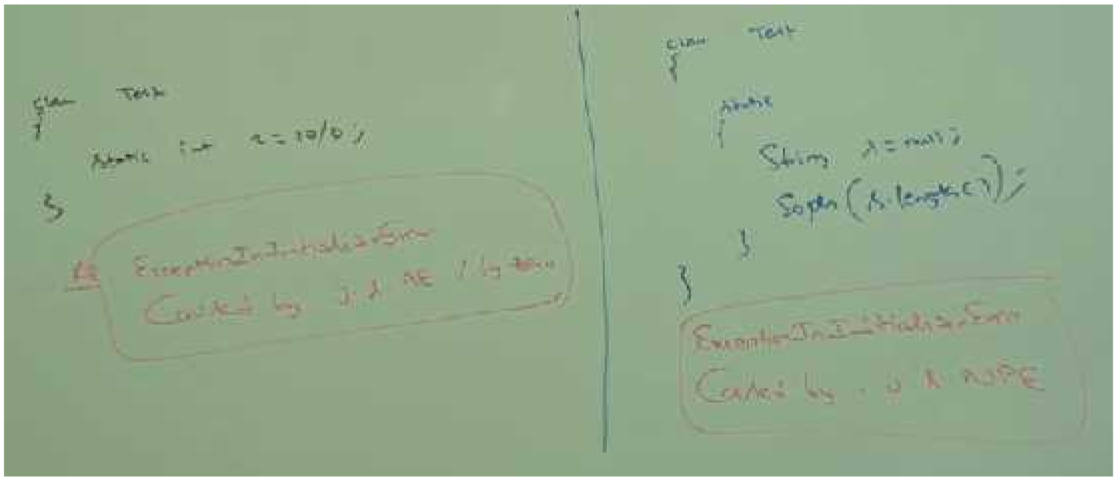
if Test.class file is not available then we will get runtime exception saying NoClassDefFoundError:Test.



```
java Test ↵
RE: NoClassDefFoundError : Test
```

ExceptionInInitializerError:

It is the child class of error and hence it is unchecked. raised automatically by JVM if any exception occurs while executing staic variable assignments and static blocks.



```
class Test
{
    static int x = 10/0;
}
RE: ExceptionInInitializerError
    Caused by: j.l.AE: / by zero
```

```
class Test
{
    static
    {
        String s = null;
        Sopln(s.length());
    }
}
ExceptionInInitializerError
Caused by: j.l.NPE
```

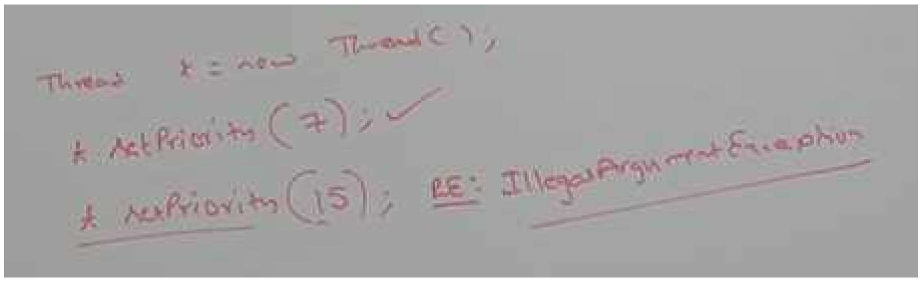
IlligalArgumentException:

It is the child class of RuntimeException and hence it is unchecked.

Raised explicitely either by programmer or by API developer to indicate that a method has been invoked with illigalargument.

e.g.

The valid range of thread prorities is 1 to 10. If we are trying to set the priority with any other value then we will get RuntimeException saying IlligalArgumentException.

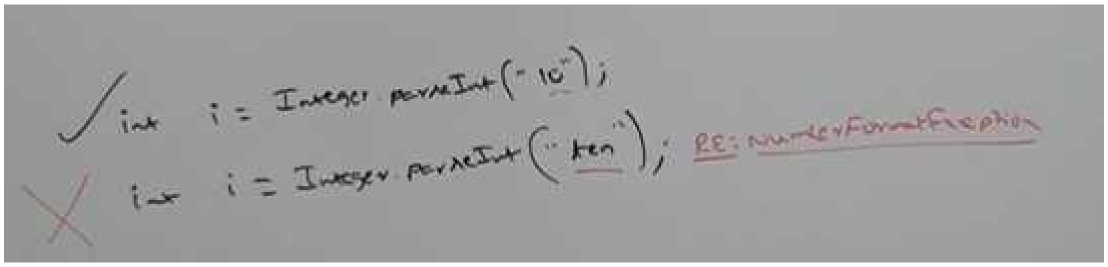


```
Thread t = new Thread();
t.setPriority(7); ✓
t.setPriority(15); RE: IllegalArgumentException
```

NumberFormatException:

It is the direct child class of IlligalArgumentException which is the child class of RuntimeException and hence it is unchecked.

raised explicitely either by programmer or by API developer to indicate that we are trying to convert string to number and the string is not properly formatted.



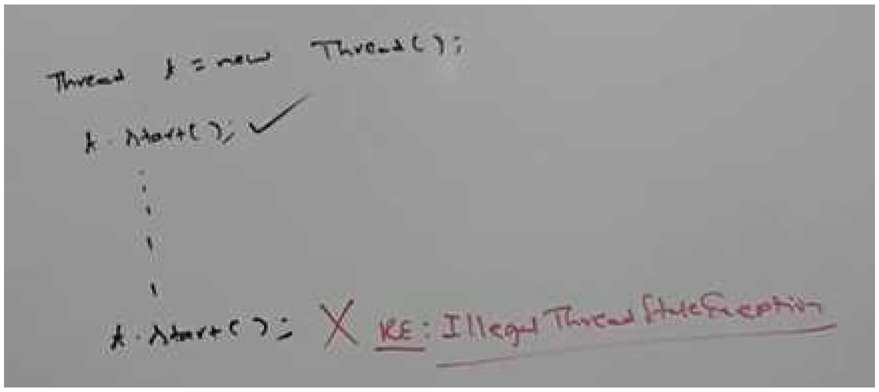
```
✓ int i = Integer.parseInt("10");
✗ int i = Integer.parseInt("ten"); RE: NumberFormatException
```

IlligalStateException:

It is the child class of RuntimeException and hence i is unchecked.

raised explicitely either by programmer or by API develper to indicate that a method has been invoked at wrong time.

After starting of a thread we are not allowed to restart the same thread once again otherwise we will get runtime exception saying IlligalThreadStateException.

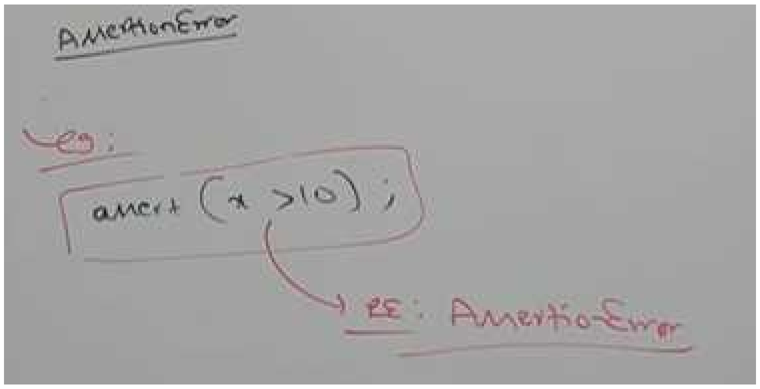


```
Thread t = new Thread();
  t.start(); ✓
     :
     :
  t.start(); ✗ RE: IllegalThreadStateException
```

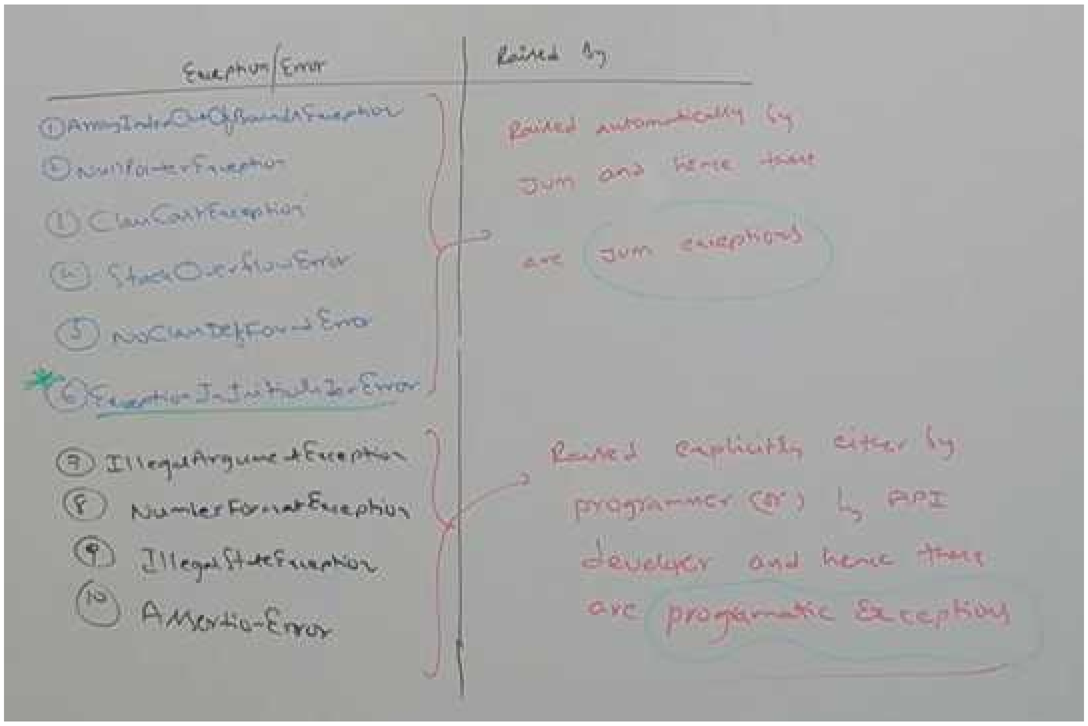
AssertionError:

It is the child class of error and hence it is unchecked.

Raised explicitely by the programmer or by API developer to indicate that Assert statement fails.



```
AssertionError

eg:
 assert (x >10);
        RE: AssertionError
```



Exception/Error	Raised by
① ArrayIndexOutOfBoundsException	Raised automatically by JVM and hence these are JVM exceptions
② NullPointerException	
③ ClassCastException	
④ StackOverflowError	
⑤ NoClassDefFoundError	
⑥ ExceptionInInitializerError	
⑦ IllegalArgumentException	Raised explicitly either by programmer (or) by API developer and hence these are programatic Exceptions
⑧ NumberFormatException	
⑨ IllegalStateException	
⑩ AssertionError	

1.7 version enhancements wrt Exception Handling:

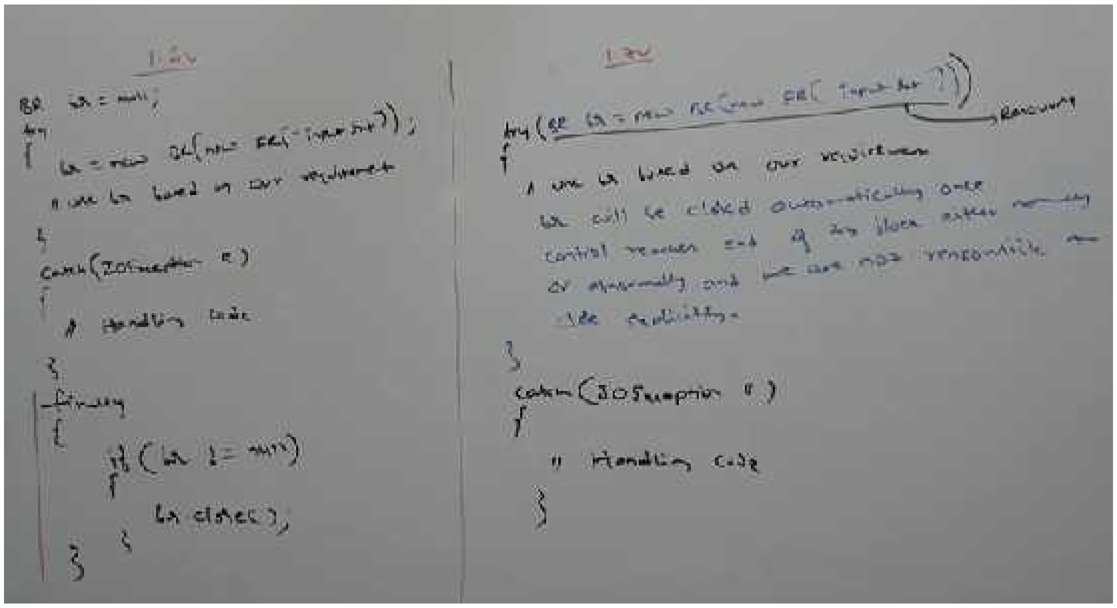
As the part of 1.7 in Exception handling the following two concepts introduced:

1. try with resources

2. milti catch block

try with resources:

Until 1.6 version it is highely recomended to write finally block to close resources which are opened as the part of try block.

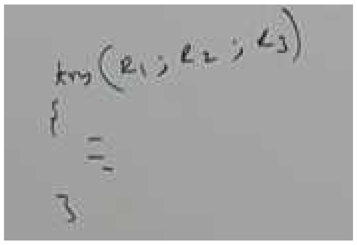


```
1.6v

BR br = null;
try
{
    br = new BR(new FR("input.txt"));
    // use br based on our requirement
}
catch(IOException e)
{
    // Handling code
}
finally
{
    if(br != null)
    {
        br.close();
    }
}
```

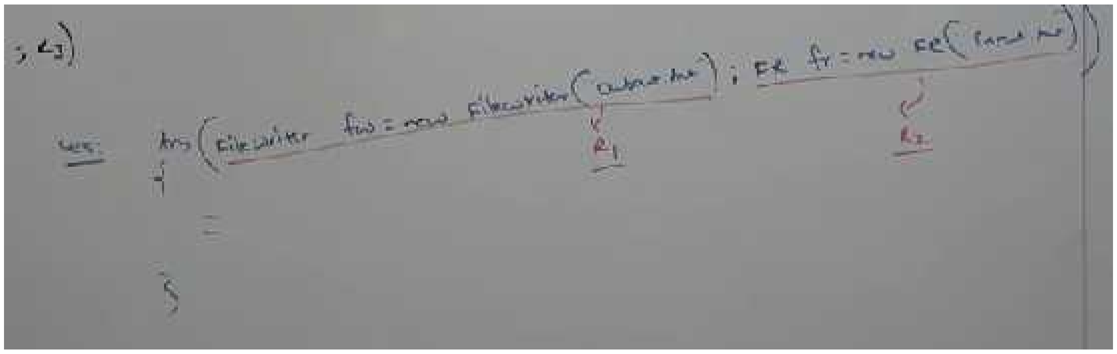
```
1.7v

try(BR br = new BR(new FR("input.txt")))   Resource
{
    // use br based on our requirement
    br will be closed automatically once
    control reaches end of try block either normally
    or abnormally and we are not responsible to
    close explicitly.
}
catch(IOException e)
{
    // Handling code
}
```

Conclusions:

We can declare multiple resources but these resources should be separated with ";"



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



```
; R3)

try(FileWriter fw = new FileWriter("output.txt"); FR fr = new FR("input.txt"))
{                                    R1                                  R2
    =
}
```

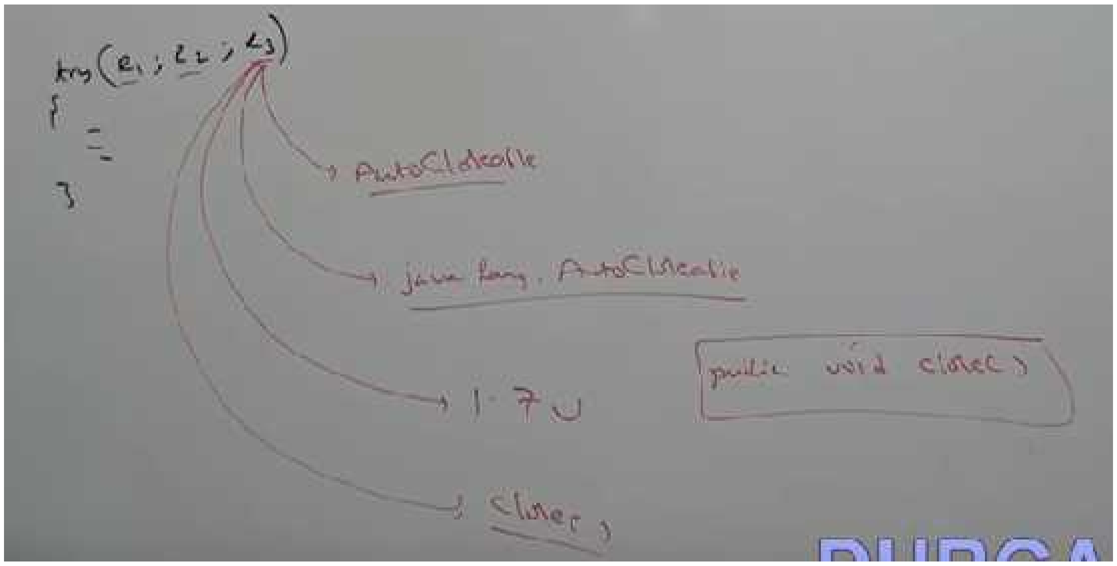
All resources should be AutoClosable resources.

A resource is said to be auto closable iff corrosponding class implemets java.lang.autoclosable interface.

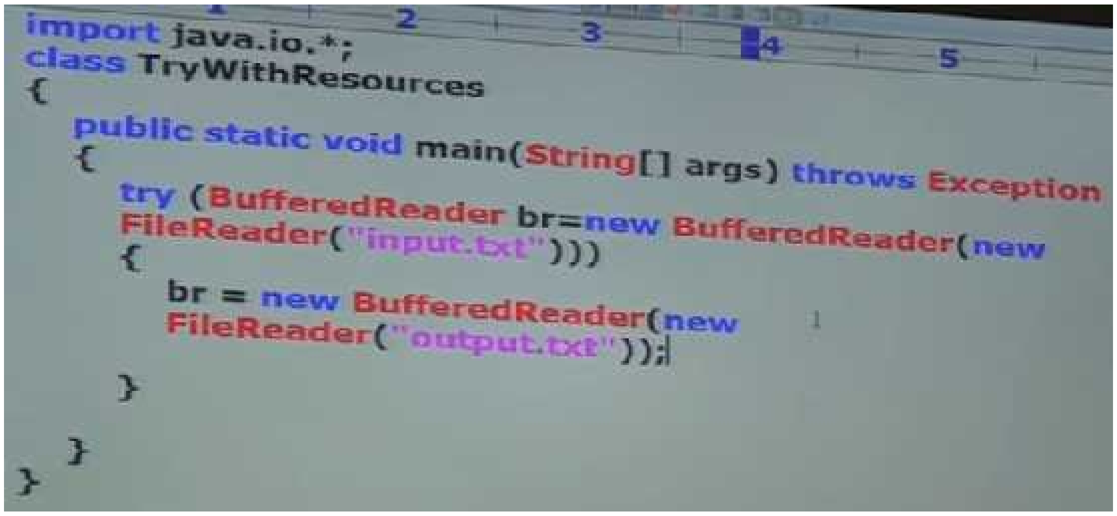
All IO related resources, database related resources and network related resources are already implemented autoclosable Interface.

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

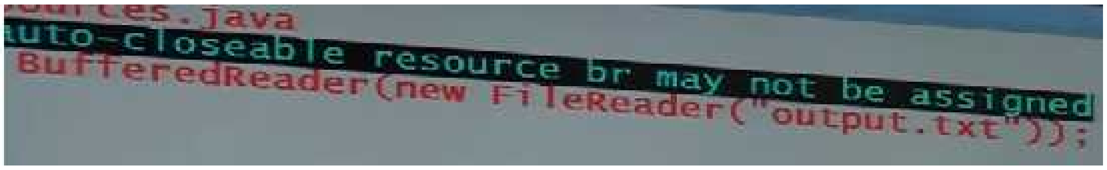
Autoclosable interface came in 1.7 version and it contains only one method close().



All resource reference variables are implicitely final and hence within the try block we can't perform reassignment otherwise we will get compiletime error.



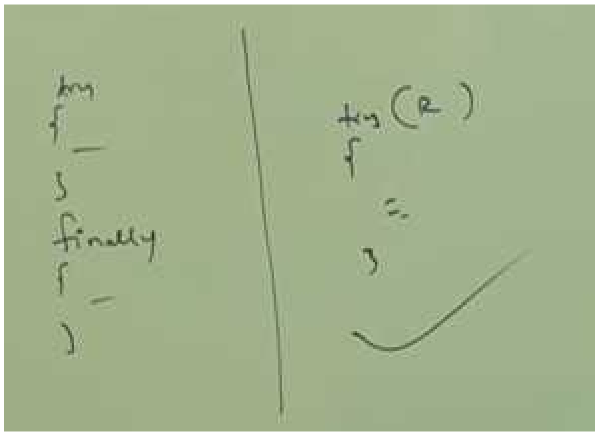
```
import java.io.*;
class TryWithResources
{
    public static void main(String[] args) throws Exception
    {
        try (BufferedReader br=new BufferedReader(new
        FileReader("input.txt")))
        {
            br = new BufferedReader(new
            FileReader("output.txt"));

        }

    }
}
```



```
ources.java
uto-closeable resource br may not be assigned
 BufferedReader(new FileReader("output.txt"));
```

Until 1.6 version try should be associated with either catch or finally. But from 1.7 version onwards we can take only try with resource without catch or finally.

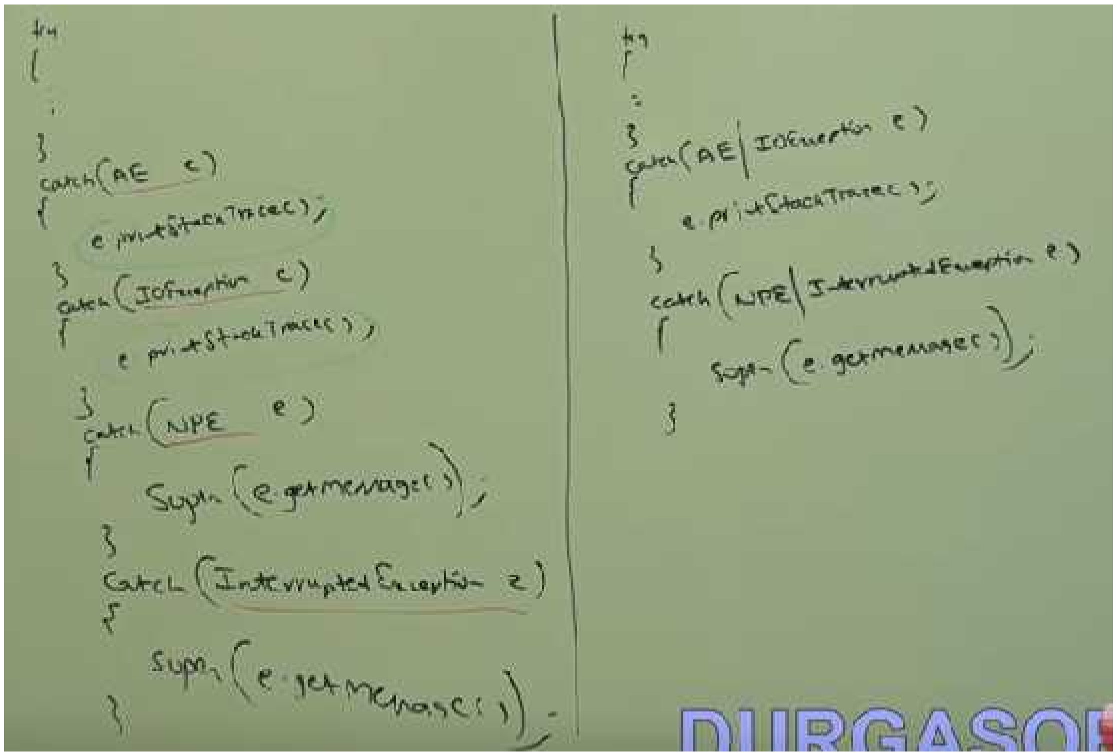
The scenario is I want abnormal termination only but close the opened resources.



```
try                 try(R)
{                   {
 _                   :
}                   }
finally
{
 _
}
```

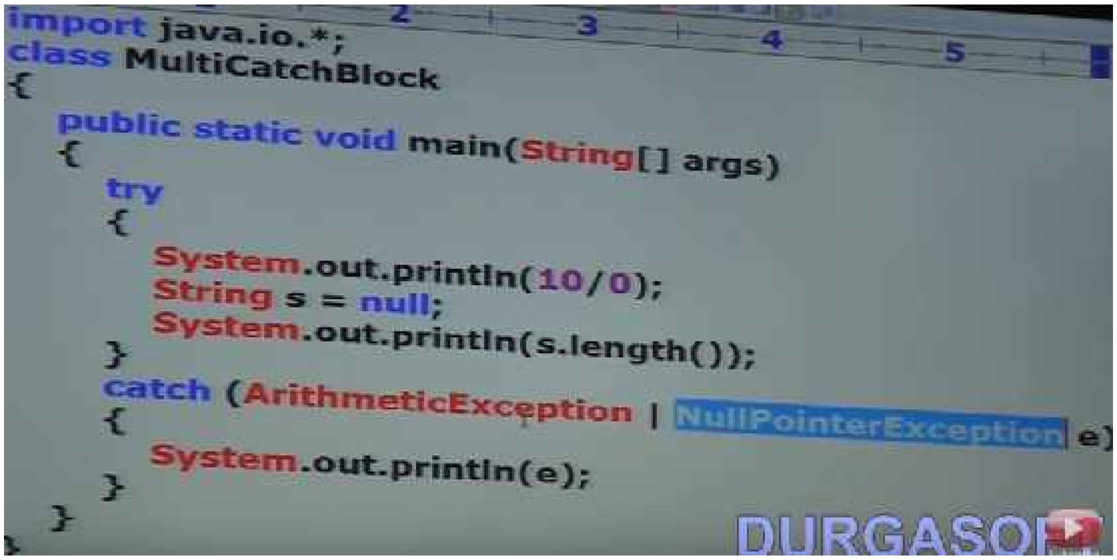
Multi catch block:

The main advantadge of this approach length of the code will be reduced and redability will be improved.



```
try
{
 :
}
catch(AE e)
{
  e.printStackTrace();
}
catch(IOException e)
{
  e.printStackTrace();
}
catch(NPE e)
{
  Sopln(e.getMessage());
}
catch(InterruptedException e)
{
  Sopln(e.getMessage());
}
```

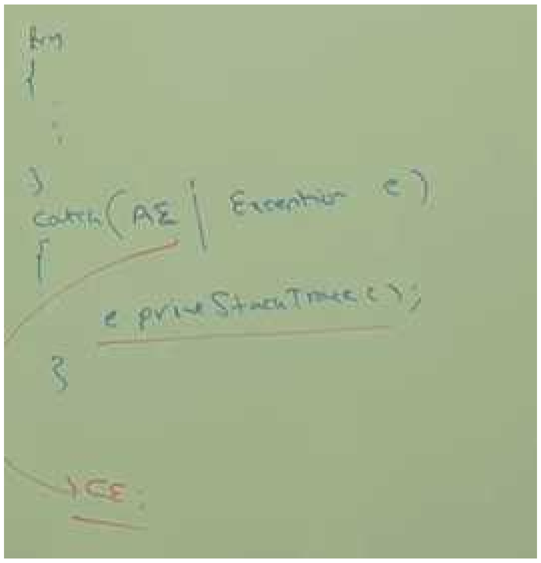
```
try
{
 :
}
catch(AE|IOException e)
{
  e.printStackTrace();
}
catch(NPE|InterruptedException e)
{
  Sopln(e.getMessage());
}
```



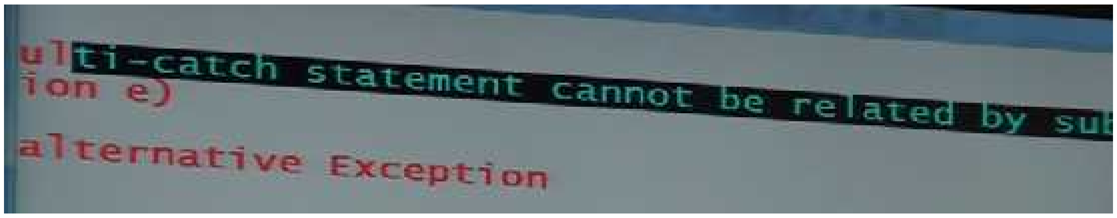
```
import java.io.*;
class MultiCatchBlock
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println(10/0);
            String s = null;
            System.out.println(s.length());
        }
        catch (ArithmeticException | NullPointerException e)
        {
            System.out.println(e);
        }
    }
}
```

In multicatch block there should not be any realtion between exception types(

either child to parent or parent to chils or same type) otherwise we will get completime error.



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

CE
```



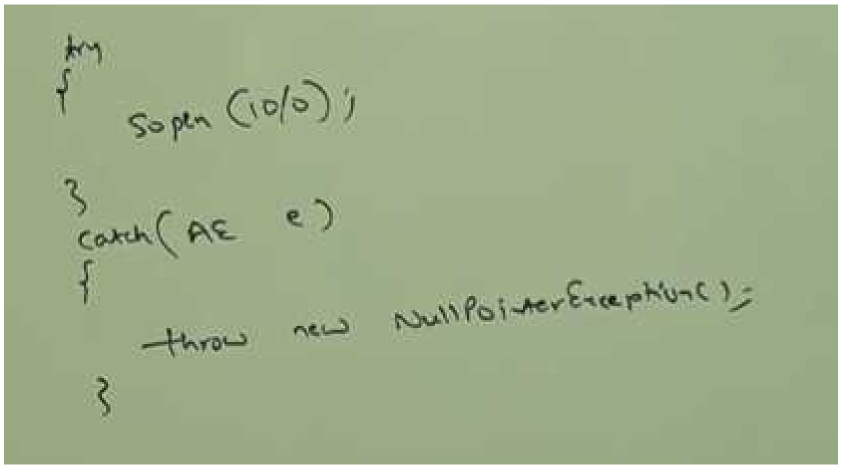
```
ulti-catch statement cannot be related by sub
ion e)
alternative Exception
```

Exception Propagation:

Inside a method if an exception raised and if we are not handling that exception then exception object will be propagated to caller then caller method is responsible to handle exception. This process is called exception propagation.

Rethrowing Exception:

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



```
try
{
    Sopln (10/0);
}
catch( AE   e )
{
    throw   new   NullPointerException();
}
```

==

OOPs:

1. Data hiding
2. Abstration
3. Encapsulation
4. Tightly Encapsulated class
5. IS-A Relationship
6. Has-A relationship
7. Method Signature
8. Overloading

9. Overriding
10. Static control flow.
11. Instance Control flow
12. Constructers
13. Coupling
14. Cohesion
15. Type casting

Data hiding:

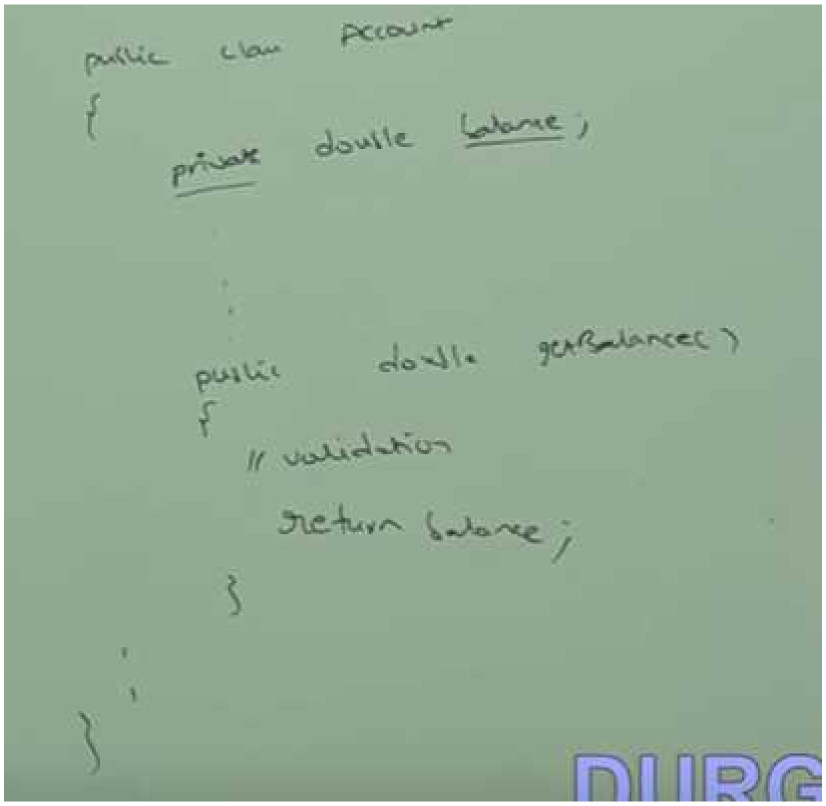
Outside person can't access our internal data directly or our internal data should not go out directly. This OOP feature is nothing but data hiding.

After validation or authentication outside persion can access our internal data.

e.g. After providing proper username and password we can able to access our gmail inbox informatiom.

e.g. Even though we are valid customer of the bank we can only able to access our account information and we can't access other's account information.

By declaring data member(variable) as 'private' we can achieve data hiding.



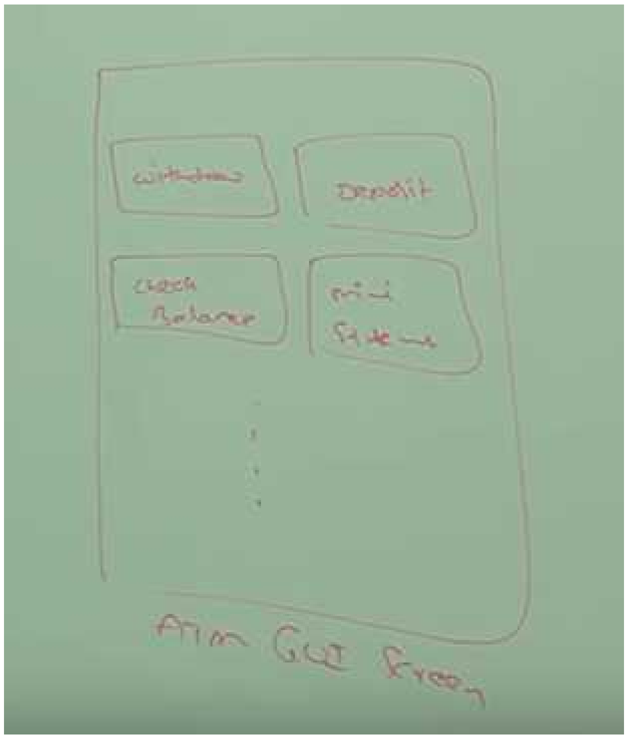
```
public class Account
{
    private double balance;
        :
        :
    public double getBalance()
    {
      // validation
        return balance;
    }
  :
  :
}
```

The main advantadge of data hiding is security.

It is highly recommanded to declare data member(variable) as 'private'.

Abstraction:

Hiding internal implementation and just highlight the set of services what we are offering is the concept of Abstraction.



The main advantadges of abstraction are,

1. Security

2. Enhancement

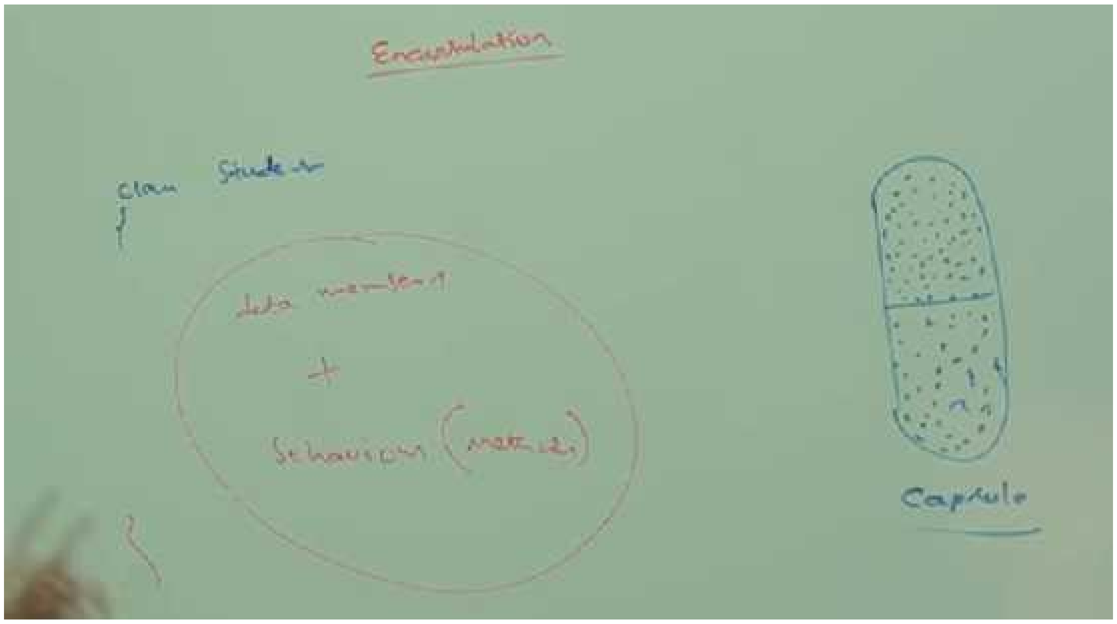
3. Improves easyness

4. Maitainability

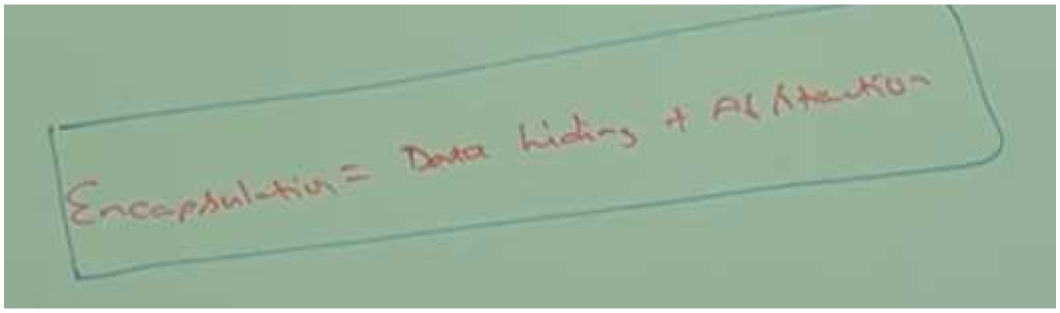
By using interfaces and abstract classes we can implement abstraction.

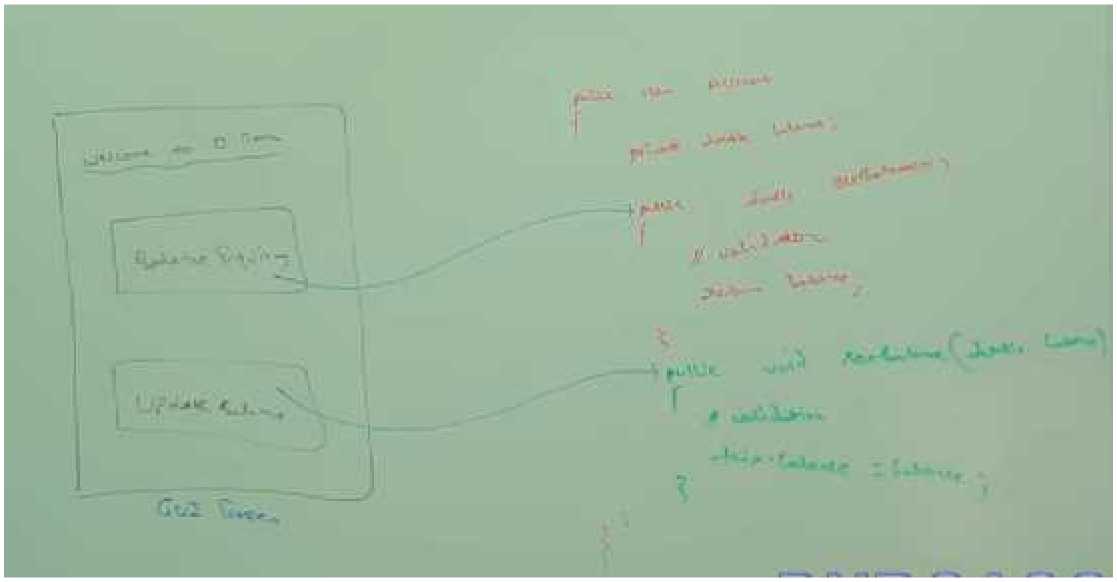
Encapsulation:

The process of binding data and corrosponding methods into a single unit is nothing but encapsulation.



If any component follows data hiding and abstraction such type of component is said to be encapsulated component.





The main advantadges of Encapsulation are we can achieve security

Enhancement will become easy

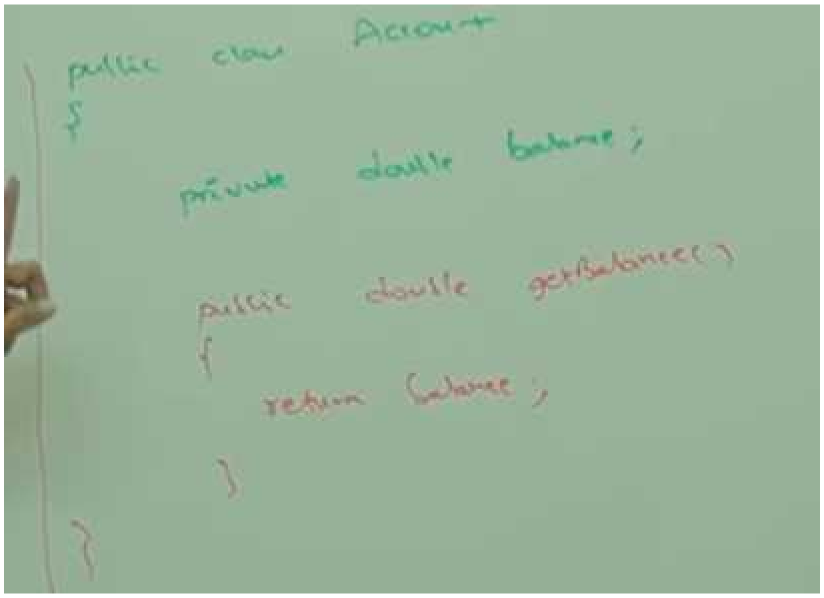
It improves maintainability of the application.

**The main advantadge of encapsulation is we can achieve security but the main disadavantadge of encapsulation is it increases length of the code and slows down execution.

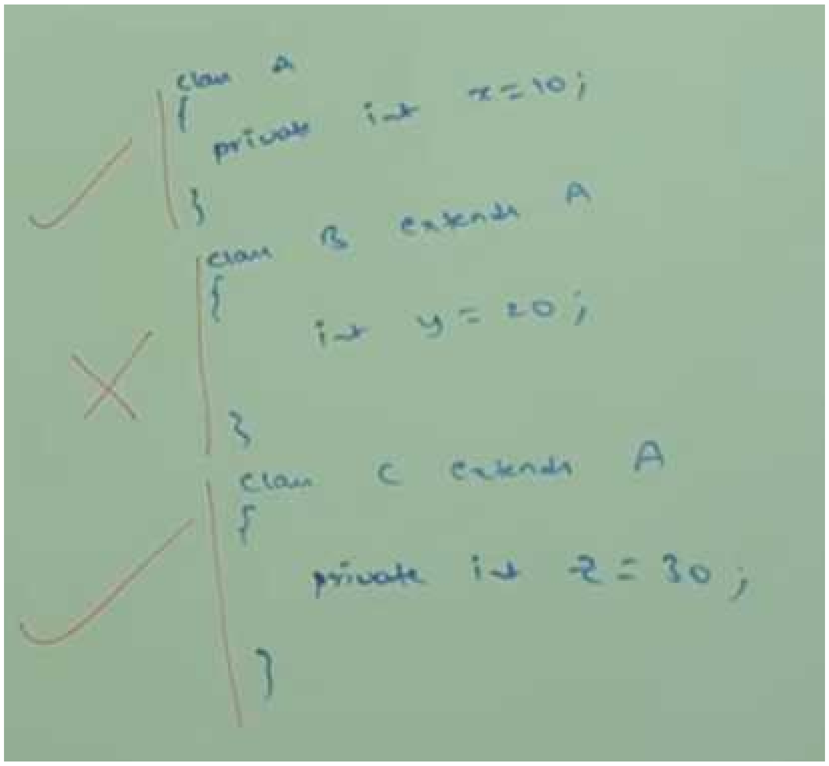
Tightly Encapsulated Class:

A class is said to be tightly encapsulated if and only if each and every variable declared as private.

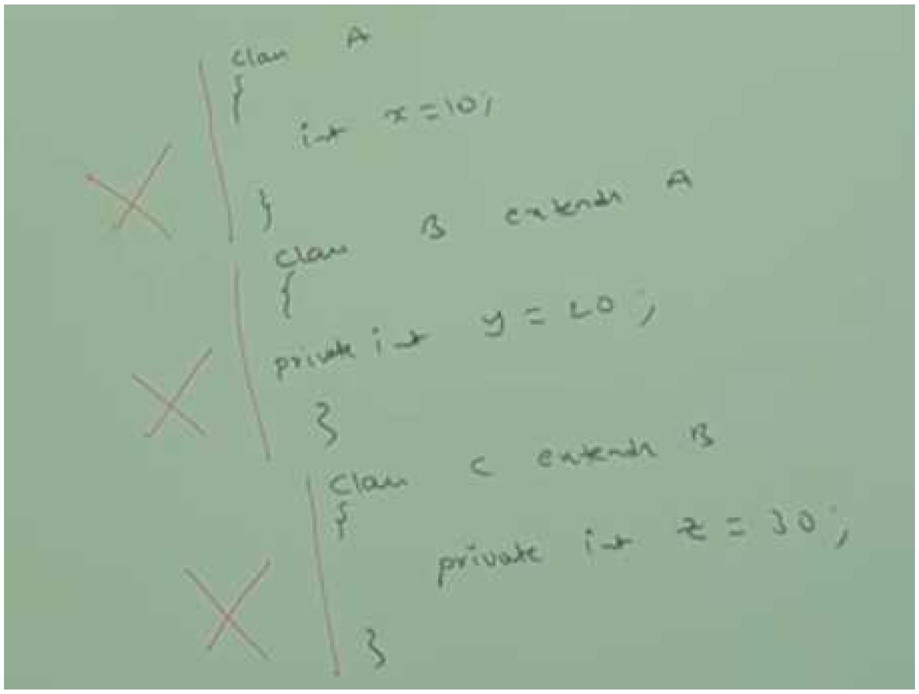
Whether class contains corrosponding getter and setter methods are not and whether these methods are declared as public or not these things we are not required to check.



```
public class Account
{
    private double balance;

    public double getBalance()
    {
        return balance;
    }
}
```



```
class A
{
  private int x=10;
}
class B extends A
{
  int y=20;
}
class C extends A
{
  private int z=30;
}
```



```
class A
{
  int x=10;
}
class B extends A
{
private int y=20;
}
class C extends B
{
  private int z=30;
}
```

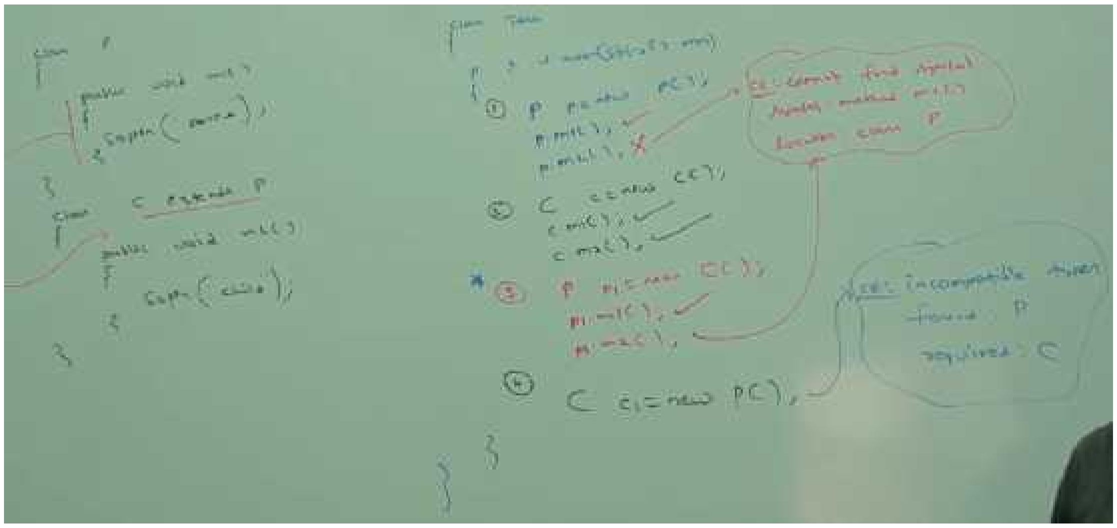
If the parent class is not tightly encapsulated then no child class is tightly encapsulated.

IS-A Relationship:

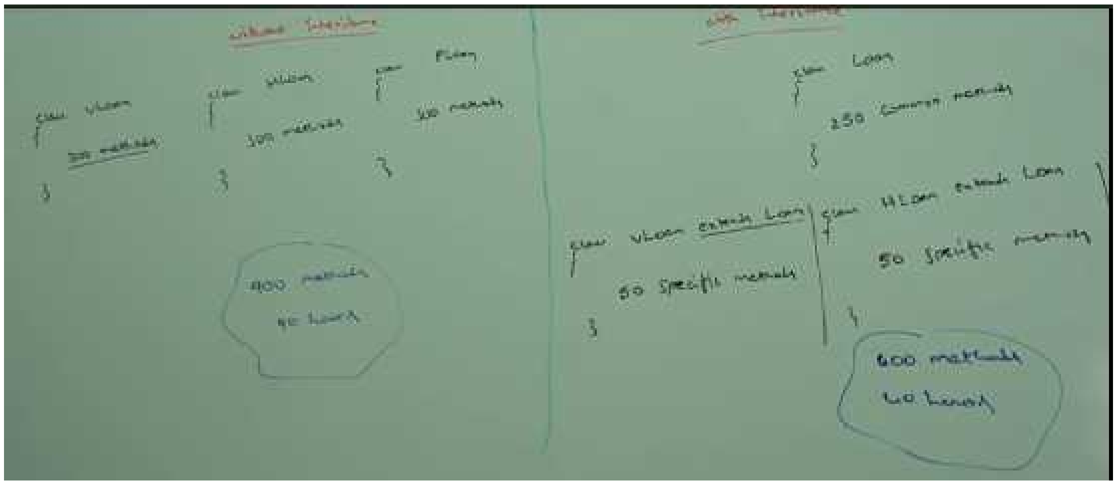
It is also known as inheritance

The main advantadge of IS-A relationship is code reusability.

By using extends keyword we can implement IS-A relationship.



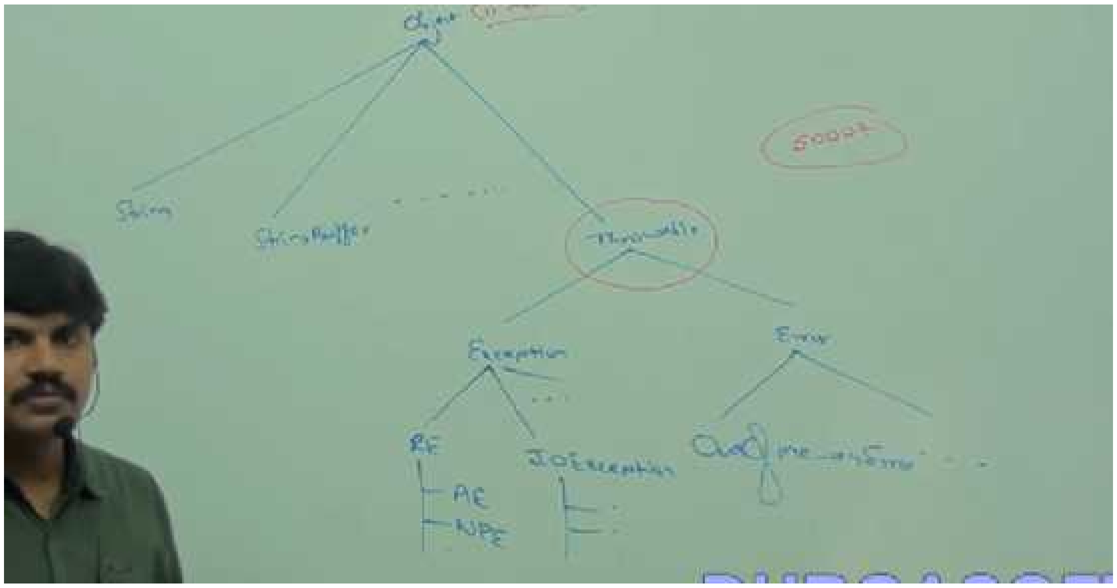
```
class P
{
    public void m1()
    {
        Sopln("parent");
    }
}
class C extends P
{
    public void m2()
    {
        Sopln("child");
    }
}

class Test
{
    p s v main(String[] args)
    {
①      P p=new P();
        p.m1(); ✓
        p.m2(); ✗        CE: cannot find symbol
                         symbol: method m2()
                         location: class P
②      C c=new C();
        c.m1(); ✓
        c.m2(); ✓
③      P p1=new C();
        p1.m1(); ✓
        p1.m2();
④      C c1=new P();    CE: incompatible types
                         found: P
                         required: C
    }
}
```

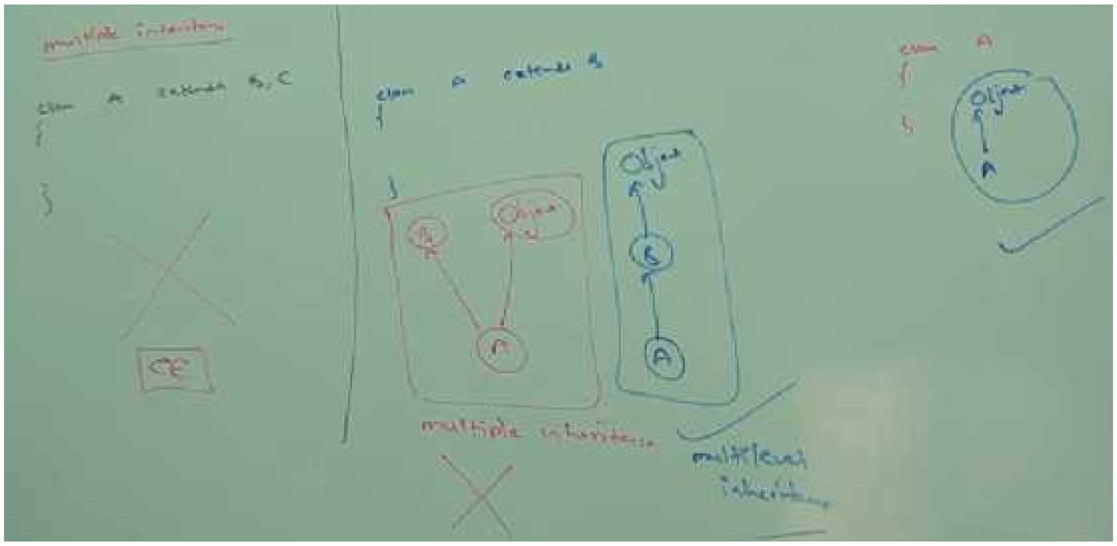


```
without inheritance

class VLoan          class HLoan          class PLoan
{                    {                    {
  300 methods          300 methods          300 methods
}                    }                    }

900 methods
90 hours

with inheritance

class Loan
{
  250 common methods
}

class VLoan extends Loan      class HLoan extends Loan
{                             {
  50 specific methods           50 specific methods
}                             }

600 methods
60 hours
```

Total Java api is impemented based on inheritance concept.

the most common methods which are applicable for any java object are defined in Object class and hence every class in java is the child class of object either directly or indirectly, so that Object class methods by default available to every java class without rewriting.



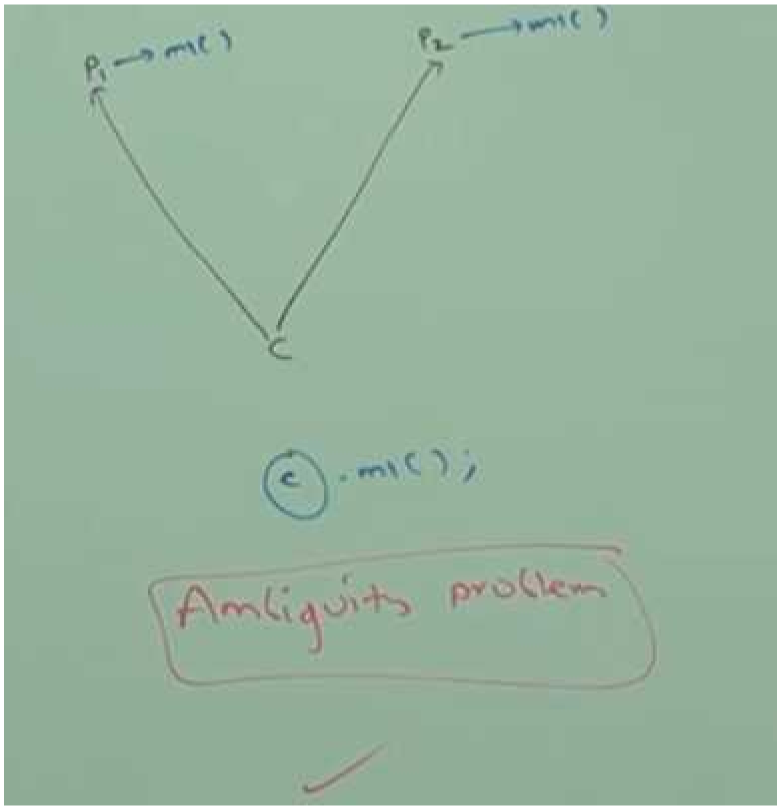


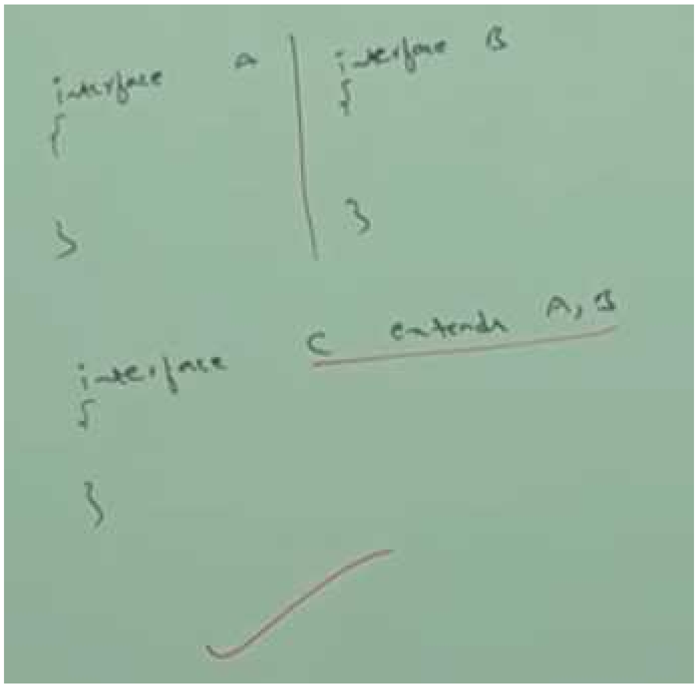
note:

1. If our class does not extend any other class then only our class is direct child class of object.

2. If our class extends any other class then our class is indirect child class of object.

Q. Why Java won't provide support for multiple inheritance ?

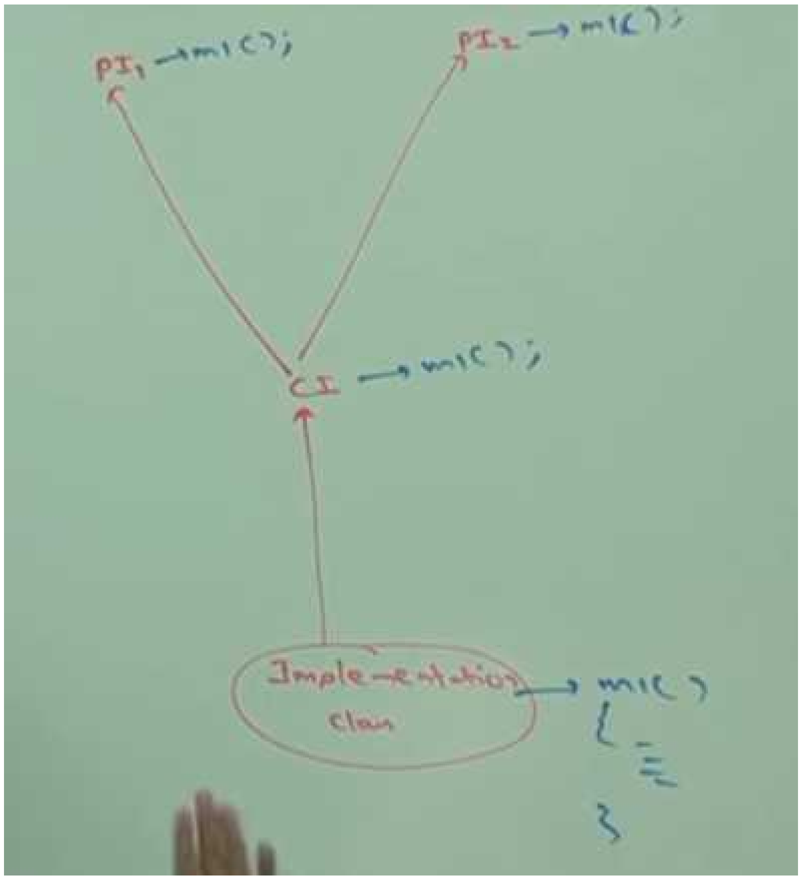




```
interface A      | interface B
{                | {
                 |
}                | }

interface C extends A, B
{

}
```

Q. Why amiguity problem won't be there in interfaces ?

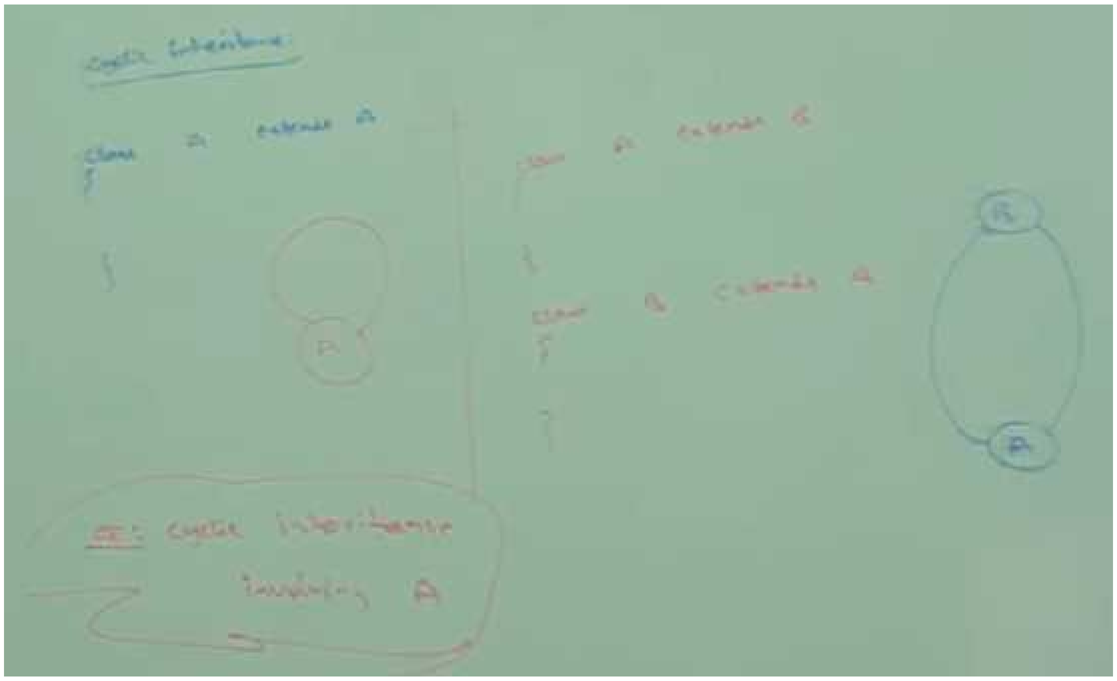
Ans:



Even though multiple method declarations are available but implementation in unique and hence there is no chance of ambiguity problem in interfaces.

Note:

Strictly speaking through interfaces we won't get any inheritance.



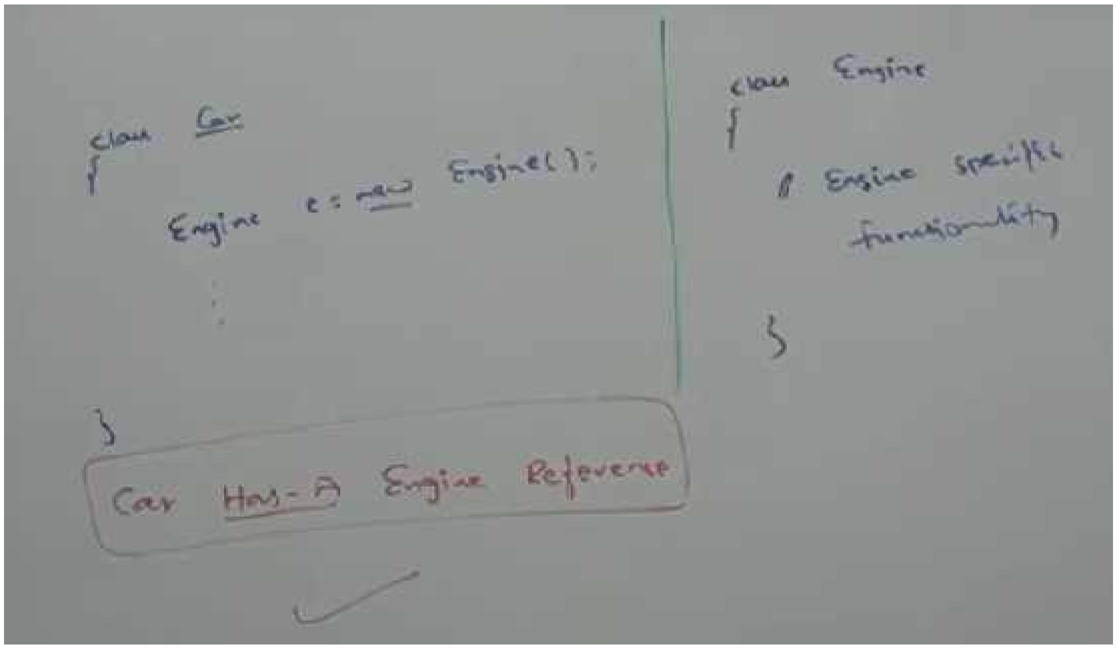
HAS-A Relationship:

HAS-A relationship is also known as composition/Aggregartion.

There is no specific keyword to implement has-a relation but most of the times we are depending on 'new' keyword.

The main advantadge of HAS-A relationship is reusability of the code.

e.g.



```
class Car
{
    Engine e = new Engine();
        :
        :
}
```

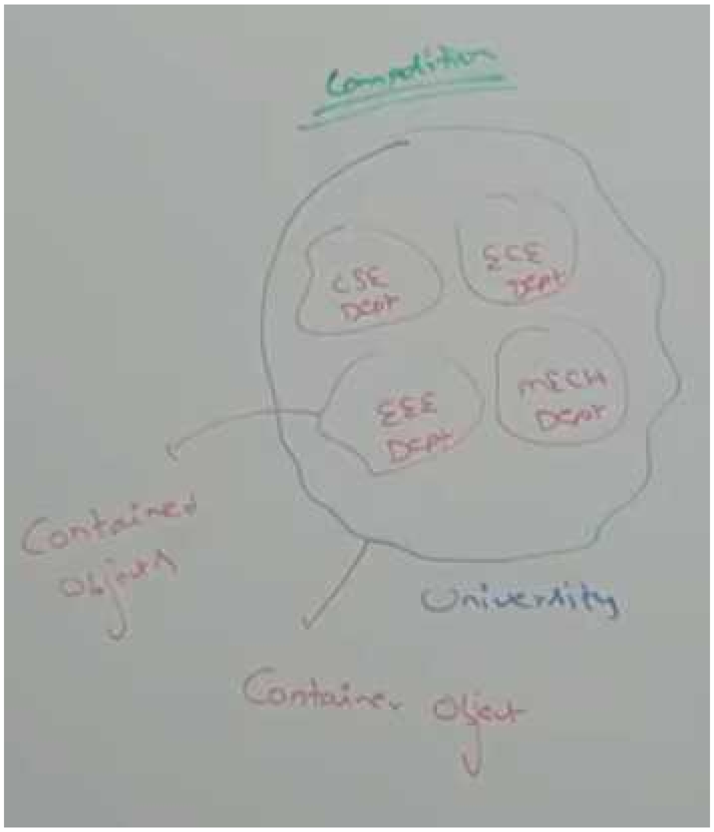
Car Has-A Engine Reference

```
class Engine
{
    // Engine specific
       functionality
}
```

Difference between composition and aggragation ?

Without existing container object if there is no chance of existing contained objects then container and contained objects are strongly associated and this strong association is nothing but composition.

e.g. University consists of several departments, without existing university there is no chance of existing departments hence University and department are strongly associated and this strong association is nothing but composition.

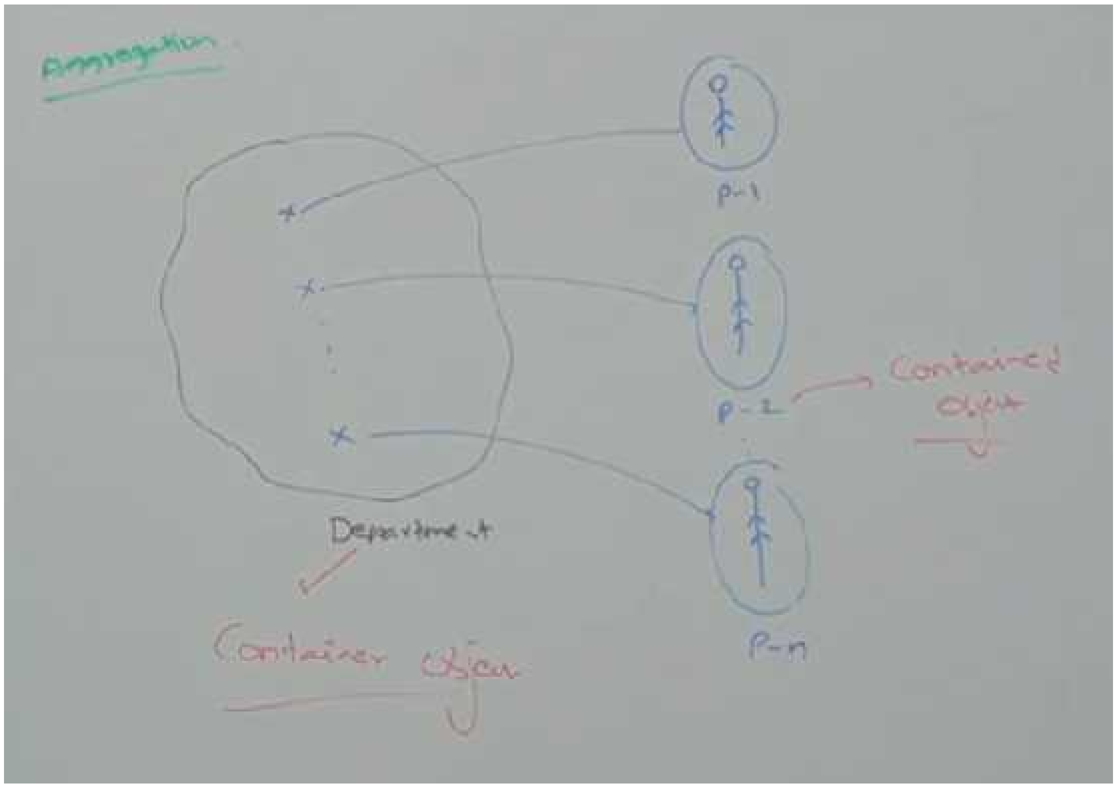


Aggregation:

Without existing container object if there is a chance of existing contained object then container and conatined objects are weakly associated and this weak association is nothing but aggregation.

e.g.

Department consists of several professors without existing department there may be a chance of existing professor objects hence department and professer objects are weakly associated and this weak association is nothing but aggregation.



In composition objects are strongly associated, where as in aggregation objects are weakly associated.

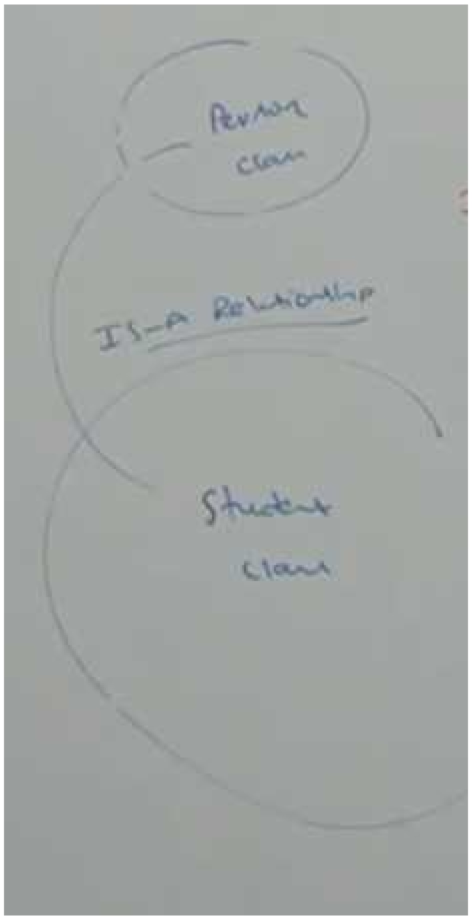
Note:

In composition container object holds directly contained objects, where as in aggregation container object holds just references of contained objects.

IS-A vs HAS-A:

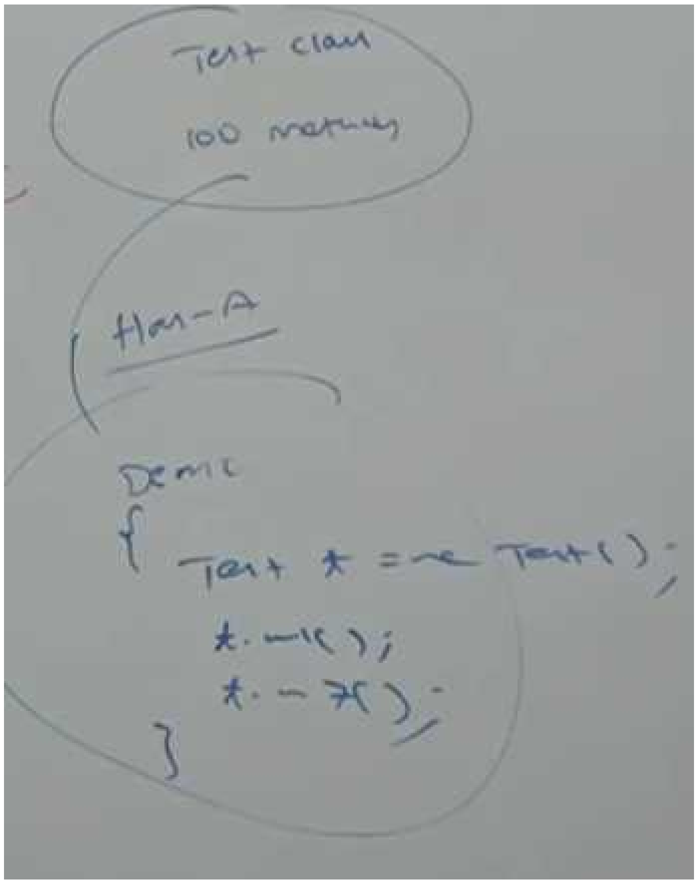
If we want total functionality of a class automatically then we should go for IS-A relationship.

e.g.



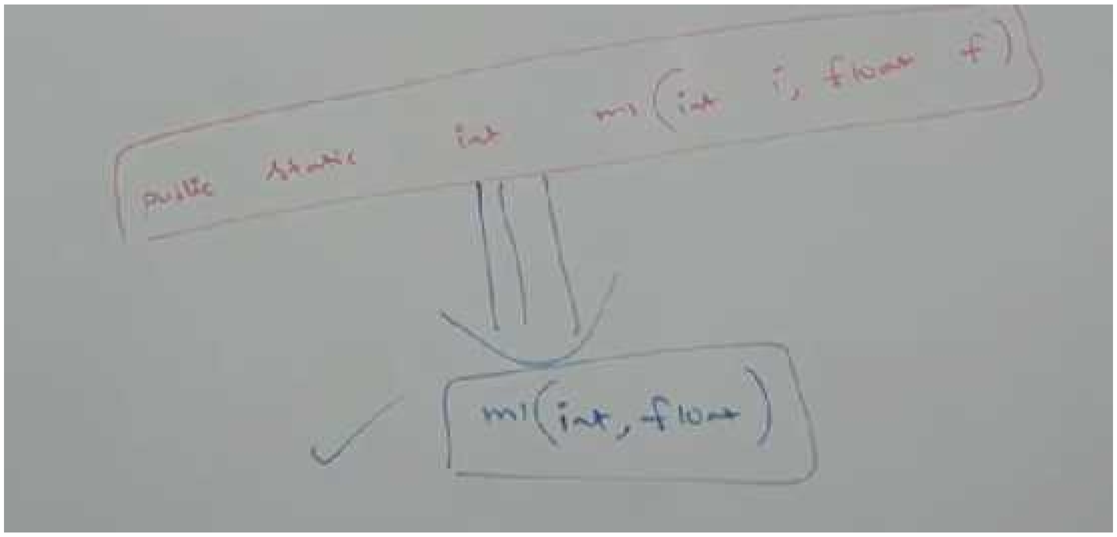
If we want part of the functionality then we should go for HAS-A relationship.

e.g.

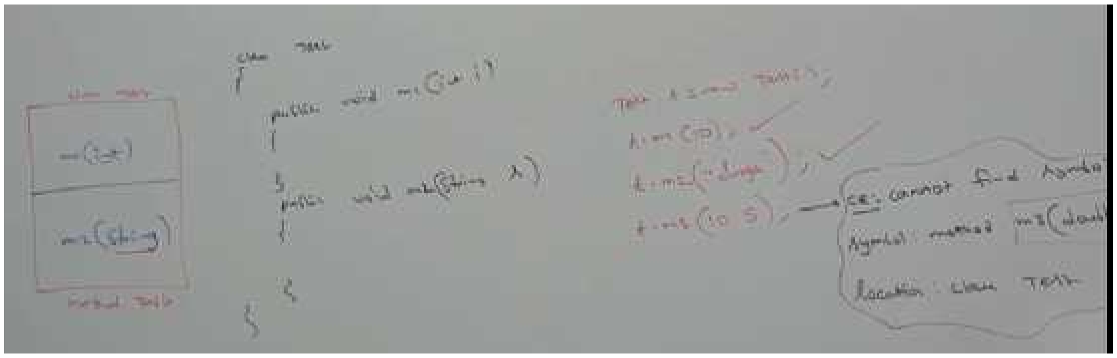


Method Signature:

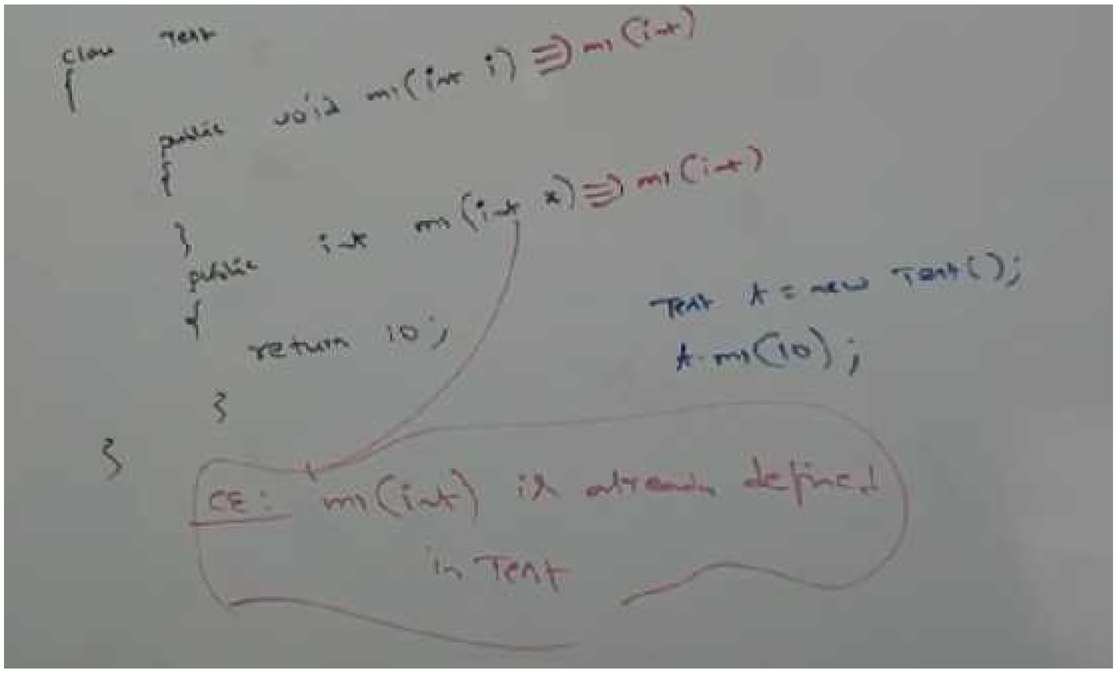
In Java method signature consists of method names followed by argument types



Compiler will use method signature to resolve method calls



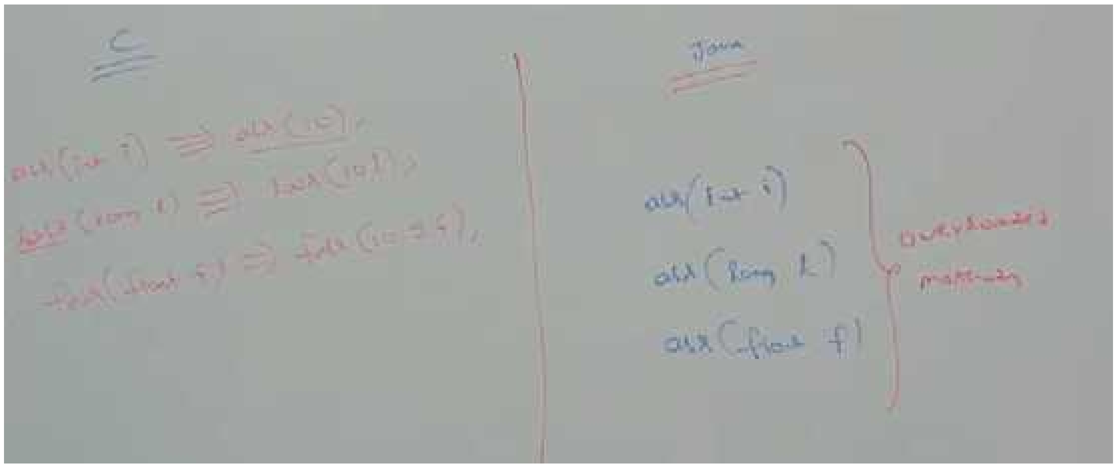
Within a class two methods with the same signature not allowed.



```
class Test
{
    public void m1(int i) => m1(int)
    {
    }
    public int m1(int x) => m1(int)
    {
        return 10;
    }
}
                    Test t = new Test();
                    t.m1(10);

CE: m1(int) is already defined
        in Test
```

Overloading:

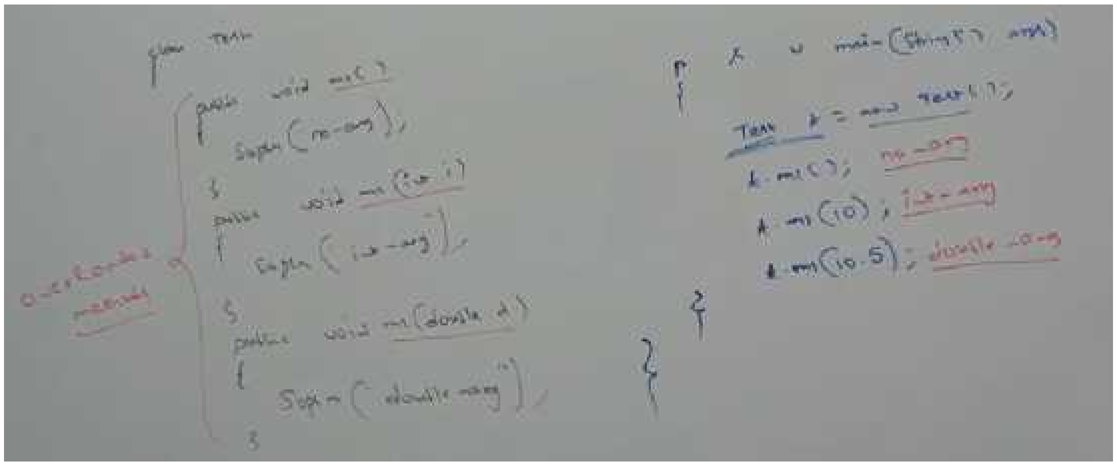
Two methods are said to be overloaded if and only if both methods having same name but different argument types.



```
C                                   Java
abs(int i) => abs(10);              abs(int i)
labs(long l) => labs(10l);          abs(long l)    } overloaded methods
fabs(float f) => fabs(10.5f);       abs(float f)
```

same name and different argument types such type of methods are called overloaded methods.

Having overloading concept in java reduces complexity of programming.



```
class Test
{
    public void m1()
    {
        Sopln("no-arg");
    }
    public void m1(int i)
    {
        Sopln("int-arg");
    }
    public void m1(double d)
    {
        Sopln("double-arg");
    }

    p s v main(String[] args)
    {
        Test t = new Test();
        t.m1();      no-arg
        t.m1(10);    int-arg
        t.m1(10.5);  double-arg
    }
}
```

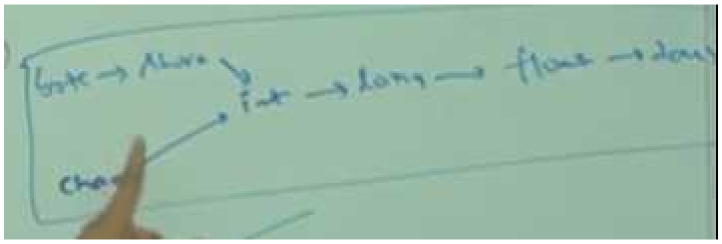
In Overloading method resolution always takes care by compiler based on reference type. hence overloading is also considered as compiletime polymorphisem or static polymorphisem or early binding.

Case 1:

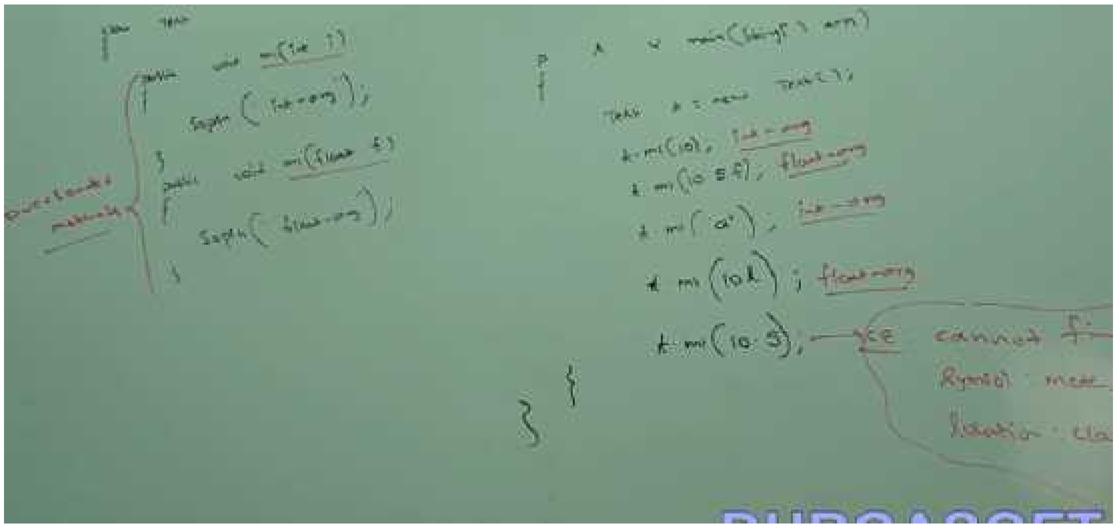
automatic promotion in overloading:

While resolving overloaded methods if exact matched methods is not avilable we won't get any compile time error immediately, ist it will promote argument to the next level and check whether matched method is available or not. If matched method is available it will be considered and if the matched method is not available then compiler promotes argumrnt once again to the next level, this process will be continued until all possible promosions still if the matched method is not availale then we will get compile time error.

The following are all possible promotions in overloading,



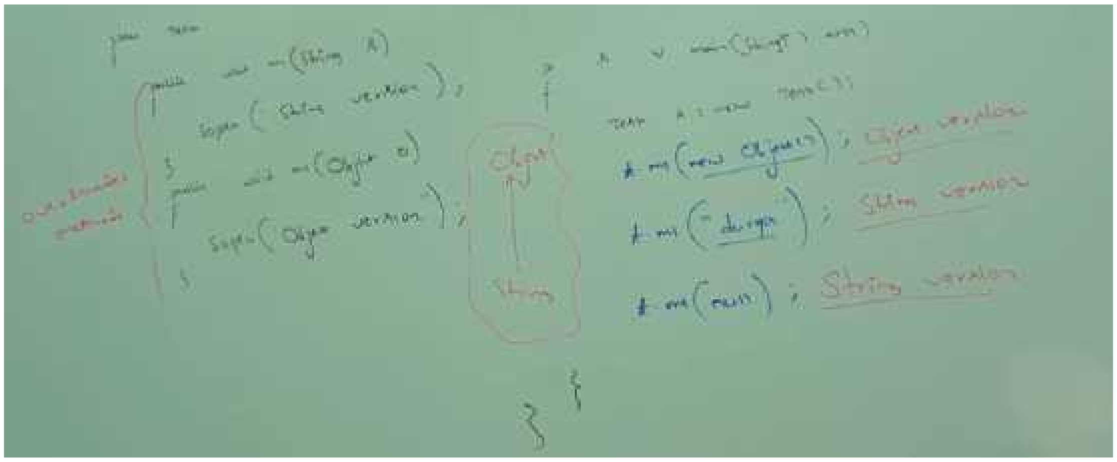
This process is called automatic promotion in overloading.


overloaded methods

```
class Test
{
    public void m1(int i)
    {
        Sopln("int-arg");
    }
    public void m1(float f)
    {
        Sopln("float-arg");
    }
```

```
p s v main(String[] args)
{
    Test t = new Test();
    t.m1(10);     int-arg
    t.m1(10.5f);  float-arg
    t.m1('a');    int-arg
    t.m1(10l);    float-arg
    t.m1(10.5);  — CE cannot f
                   symbol : met
                   location : cla
}
}
```

Case 2:

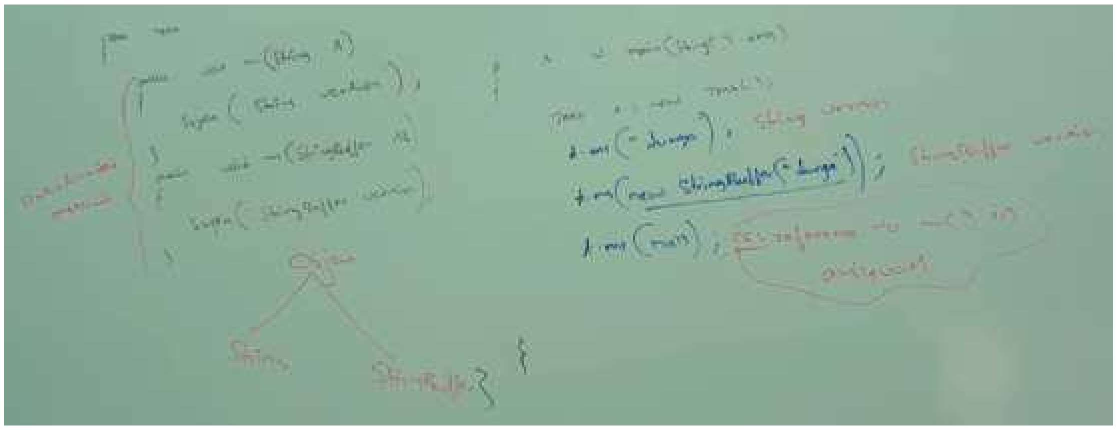

overloaded methods
Object
String

```
class Test
{
    public void m1(String s)
    {
        Sopln("String version");
    }
    public void m1(Object o)
    {
        Sopln("Object version");
    }
```

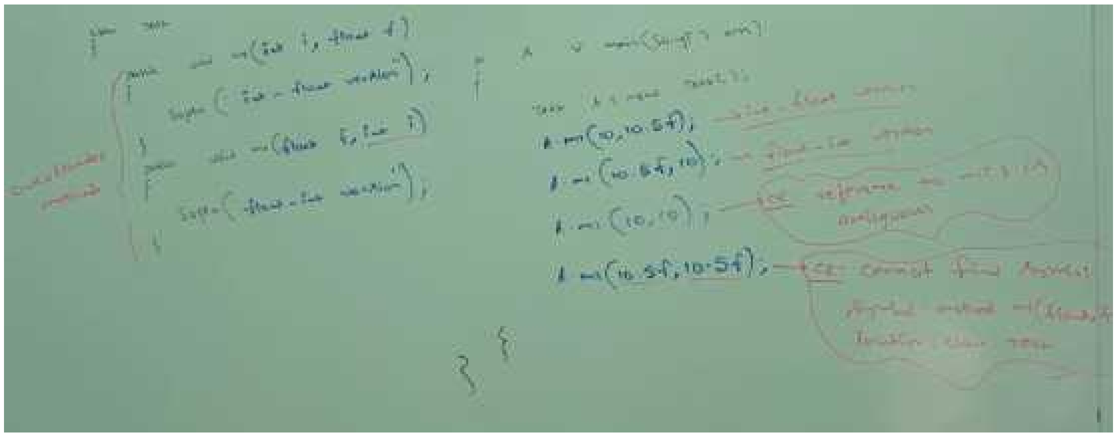
```
p s v main(String[] args)
{
    Test t = new Test();
    t.m1(new Object());  Object version
    t.m1("durga");       String version
    t.m1(null);          String version
}
}
```

While resolving overloaded methods compiler will gives the precedence for child type argumet than compared with parent type argument.

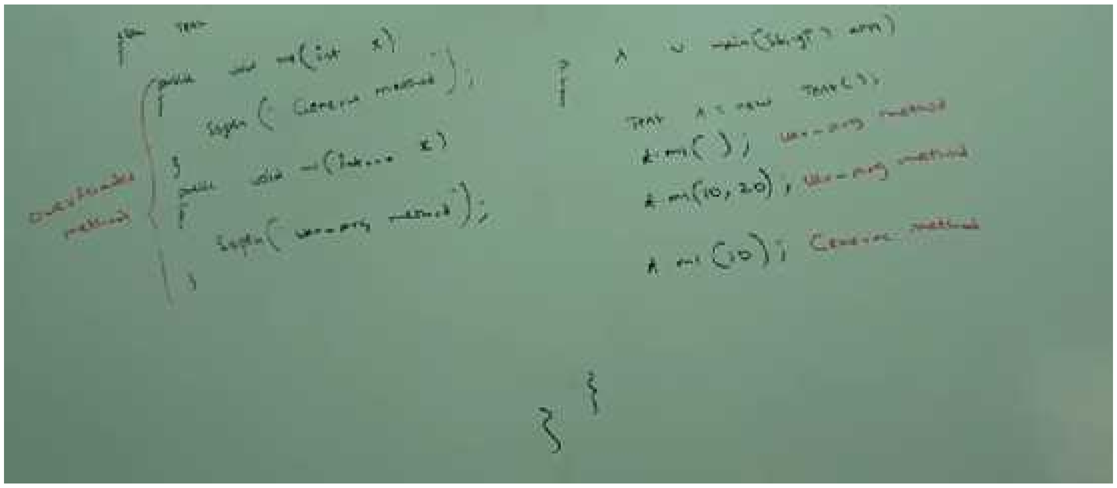
case:3



case:4



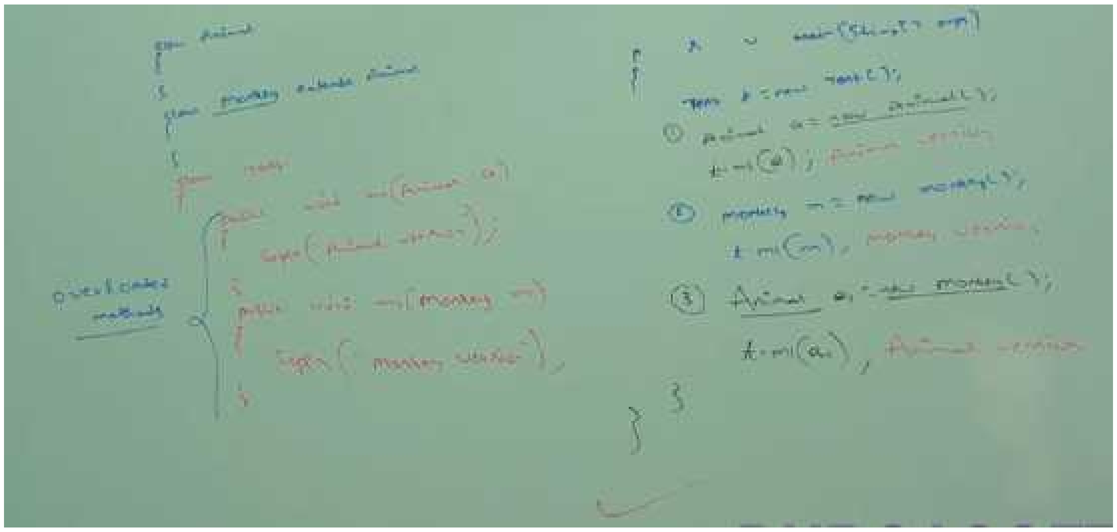
case 5:



```
class Test
{
    public void m1(int x)
    {
        Sopln("General method");
    }
    public void m1(int... x)
    {
        Sopln("var-arg method");
    }
    p s v main(String[] args)
    {
        Test t = new Test();
        t.m1();        var-arg method
        t.m1(10, 20);  var-arg method
        t.m1(10);      General method
    }
}
```

In general var-arg method will get least prority i.e. if no other method matched then only var-arg method will get the chance. it is exactly same as default case in case switch.

case: 6



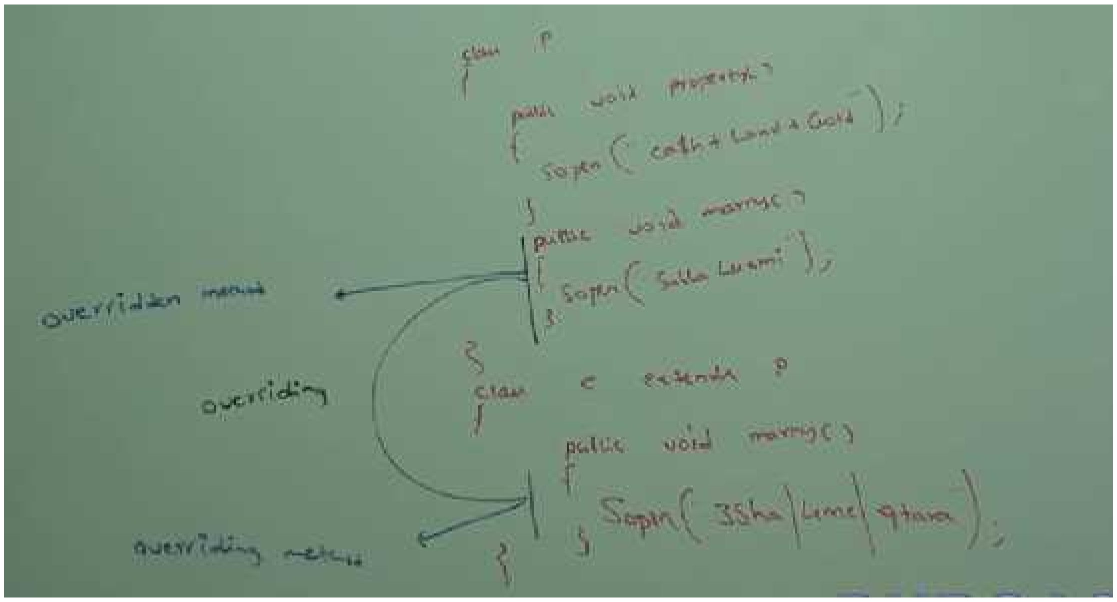
```
class Animal
{
}
class Monkey extends Animal
{
}
class Test
{
    public void m1(Animal a)
    {
        Sopln("Animal version");
    }
    public void m1(Monkey m)
    {
        Sopln("Monkey version");
    }
    p s v main(String[] args)
    {
        Test t = new Test();
        ① Animal a = new Animal();
           t.m1(a);   Animal version
        ② Monkey m = new Monkey();
           t.m1(m);   Monkey version
        ③ Animal a1 = new Monkey();
           t.m1(a1);  Animal version
    }
}
```

Method resolution always takes care by reference type. In overloading runtime object won't play any role.

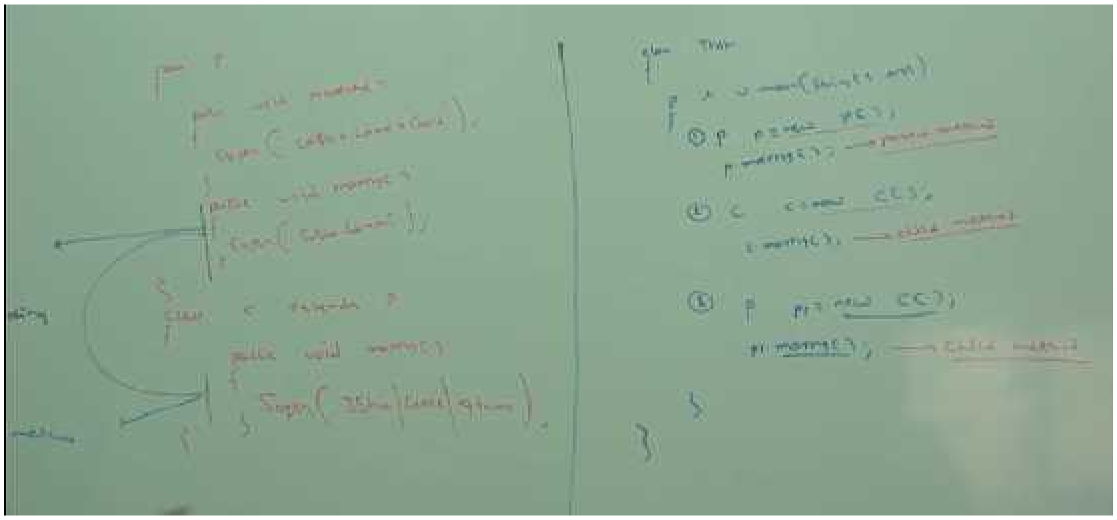
Overriding:

Whatever methods parent has by default available to the child through inheritance. If child class not satisfied with parent class implementation then child is allowed to redefine that method based on it's requirement the process is called overriding.

The parent class method which is overridden is called overridden method and the child class method which is overriding is called overriding method.



```
class P
{
    public void property()
    {
        Sopln("cash+Land+Gold");
    }
    public void marry()
    {
        Sopln("Subba Laxmi");
    }
}
class C extends P
{
    public void marry()
    {
        Sopln(3sha|4me|9tara);
    }
}
```

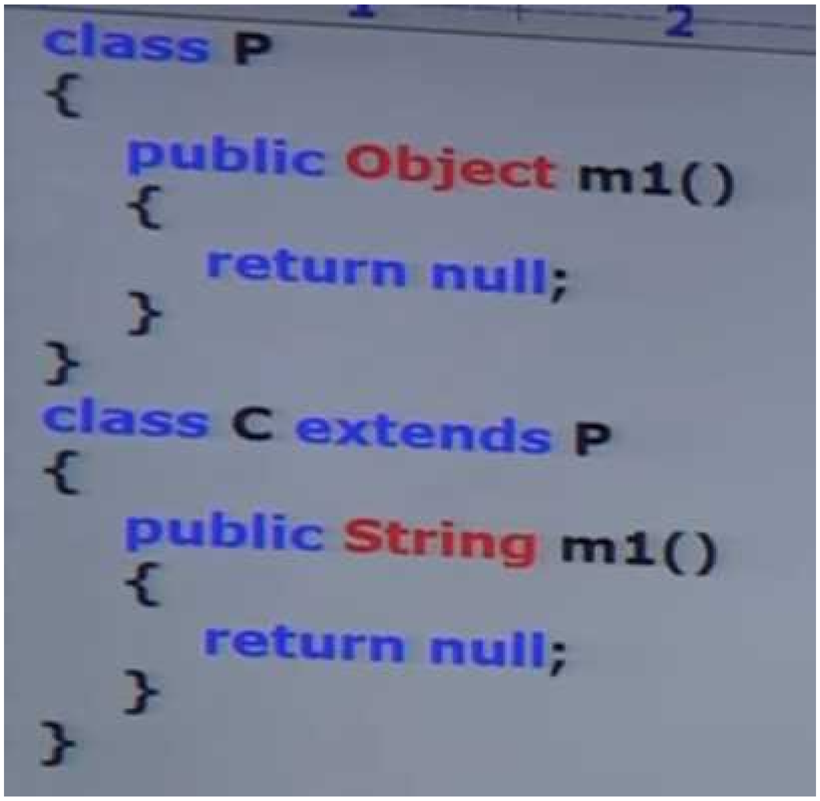


**In overriding method resolution always takes care by JVM based on runtime object and hence overriding is also considered as runtime polymorphisem, dynamic polymorphisem and late binding.

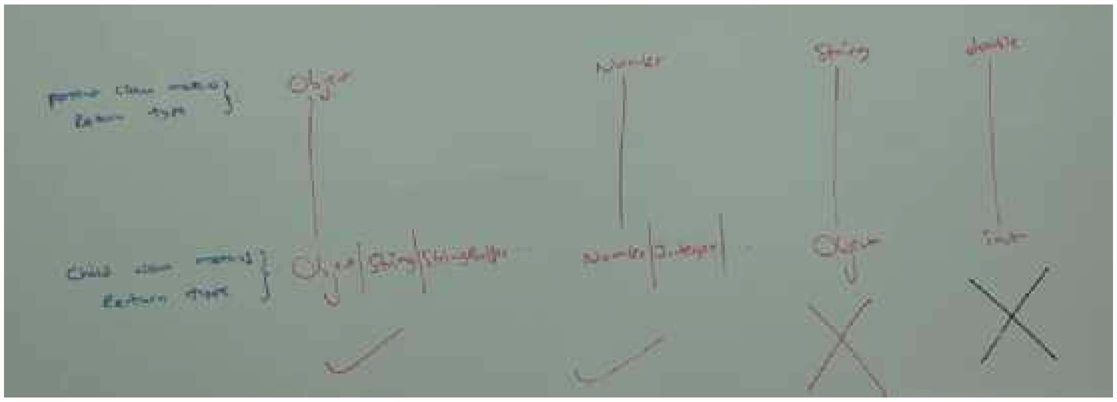
Rules for Overriding:

In overriding method names and argument types must be matched i.e. method signatures must be same.

In overriding return types must be same but this rule is applicable until 1.4 version only. from 1.5 version onwards we can take co-variant return types. According to this child class method return type need not be same as parent method return type, it's child type also allowed.

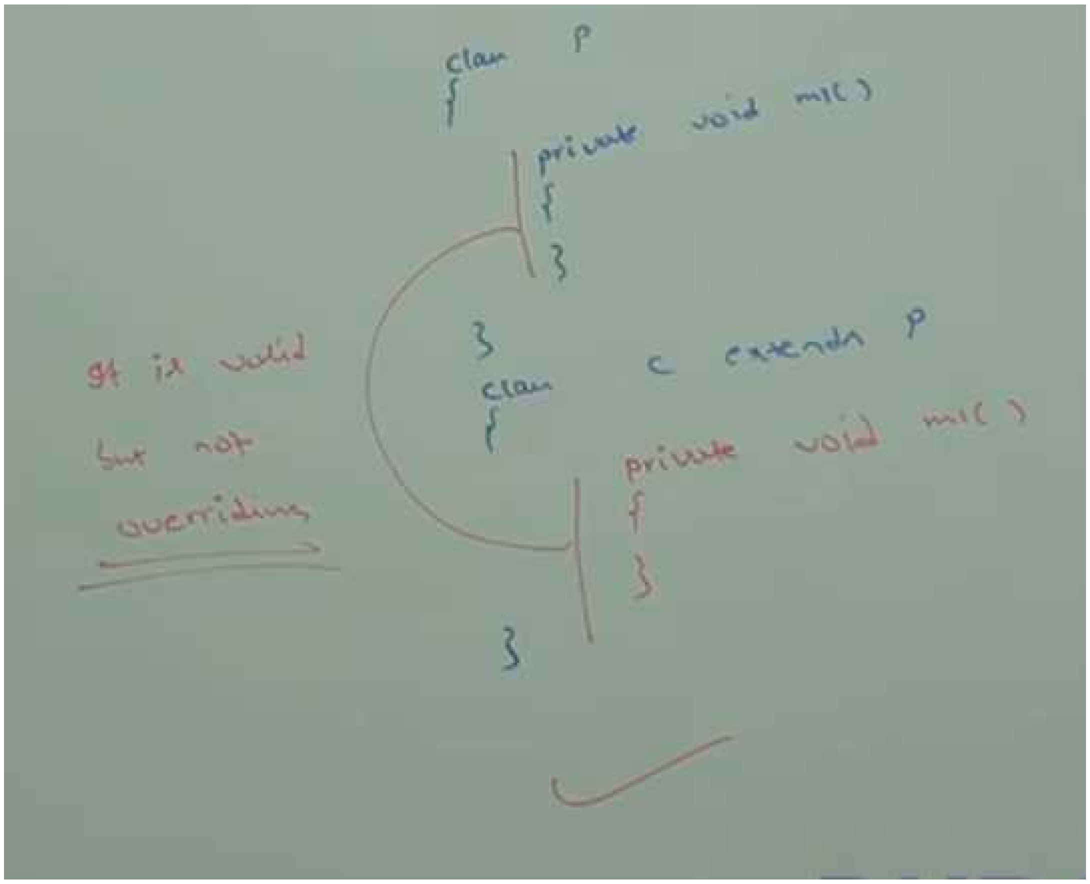


```
class P
{
    public Object m1()
    {
        return null;
    }
}
class C extends P
{
    public String m1()
    {
        return null;
    }
}
```


parent class method's Return type
Object
Number
String
double
child class method's Return type
Object | String | StringBuffer
Number | Integer
Object
int

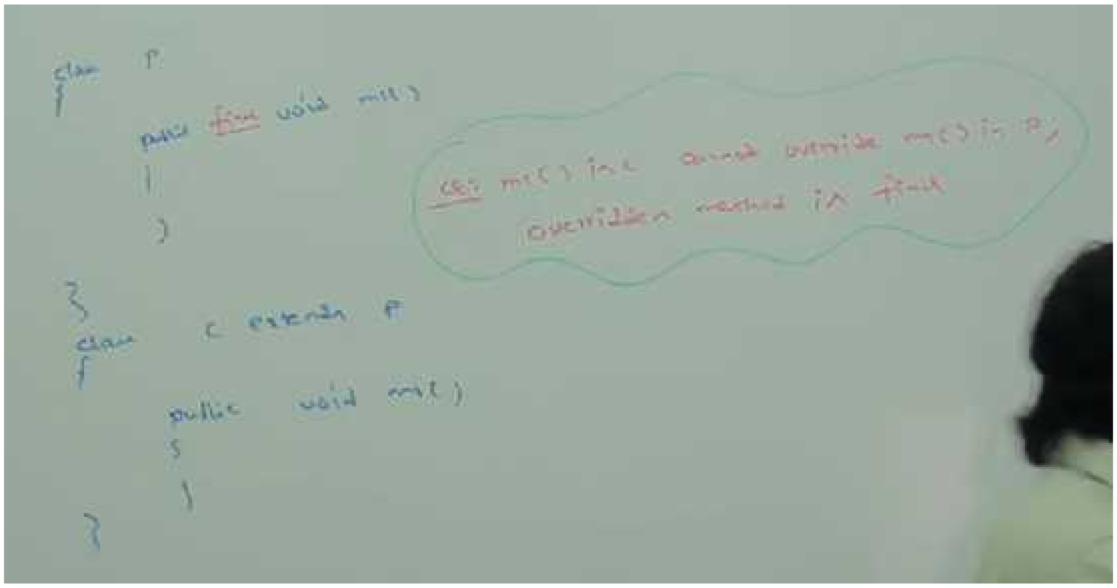
Parent class private methods not available to the child and overriding concept not applicable for private methods.

Based on our requirement we can define exactly same private method in child class. It is valid but not overriding.



```
class P
{
    private void m1()
    {
    }
}
class C extends P
{
    private void m1()
    {
    }
}
```

We can't override parent class final methods in child classes. If we are trying to do so we will get compile time error.

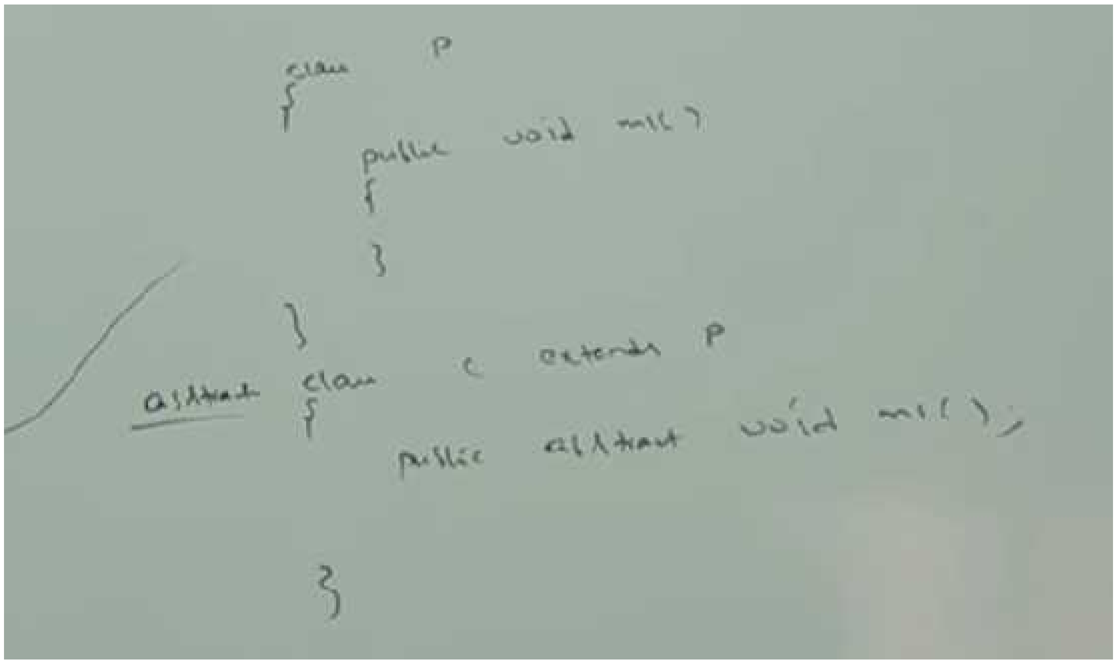


```
class P
{
    public final void m1()
    {
    }
}
class C extends P
{
    public void m1()
    {
    }
}
```

CE: m1() in C cannot override m1() in P; overridden method is final

Parent class abstract methods we should override in child class to provide implementation.

We can override non abstract method as abstract

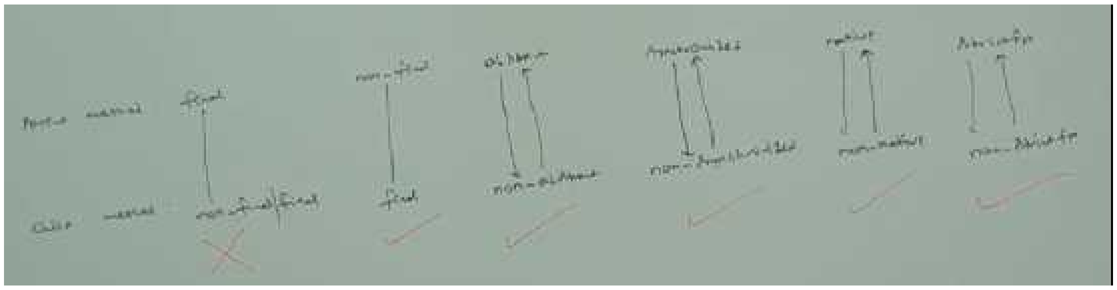


```
class P
{
    public void m1()
    {
    }
}
abstract class C extends P
{
    public abstract void m1();
}
```

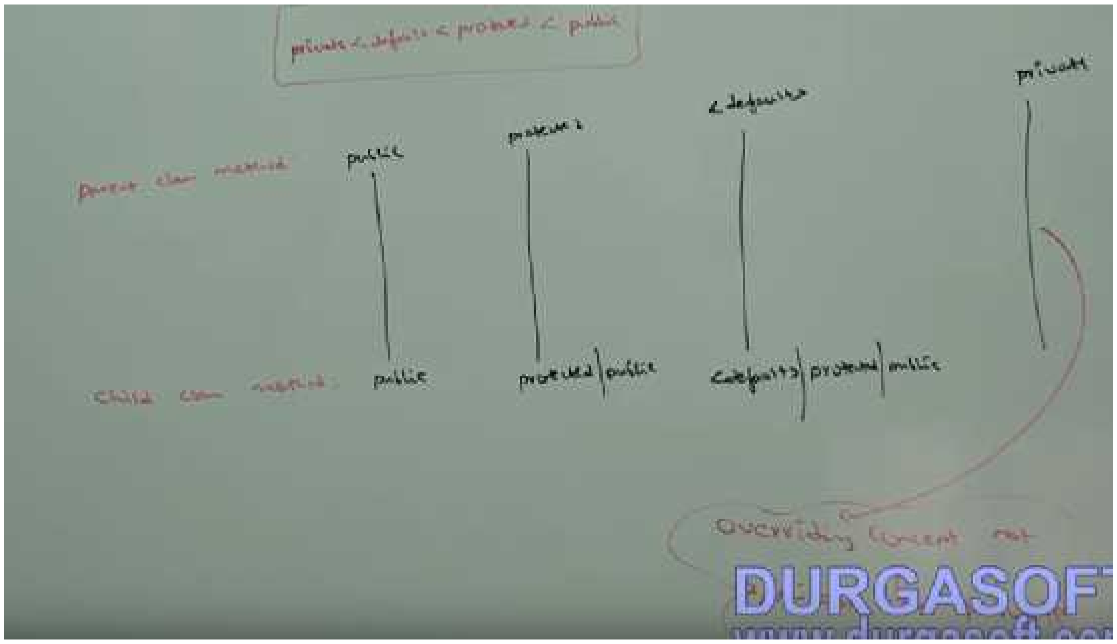
The main advantadge of this approach is we can stop the availablity of parent method implemetation to the next level child classes.

The following modifiers won't keep any restriction

syncronized, native, strictfp



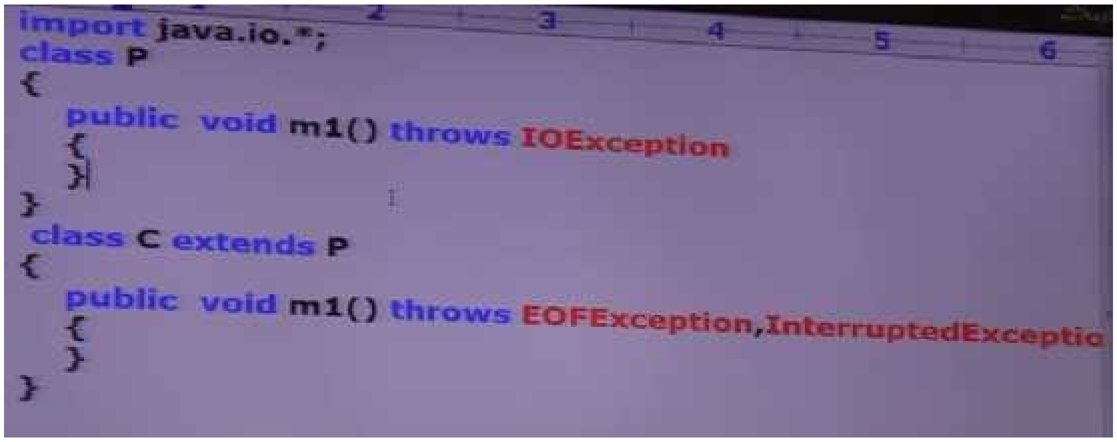
While overriding we can't reduce scope of access modifier, but we can increase the scope.



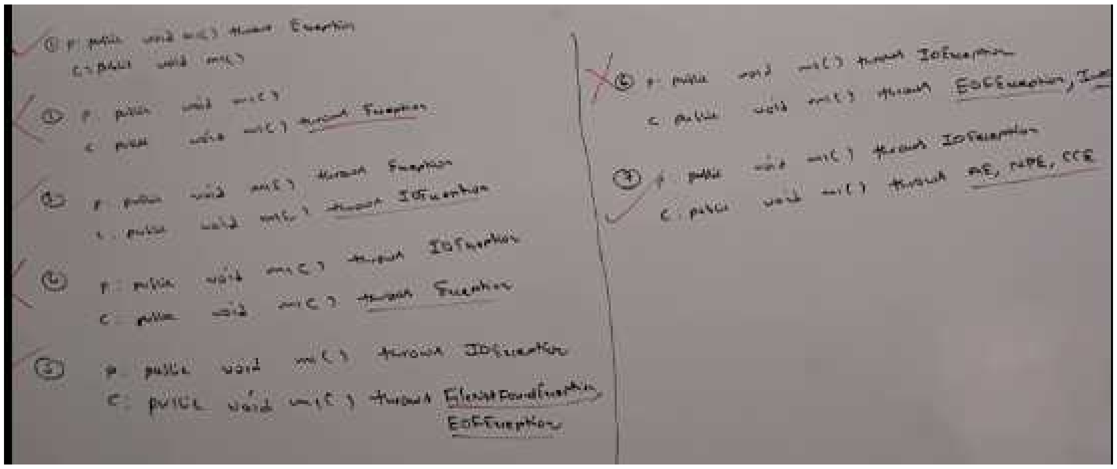
If child class method throws any checked exception compulsory parent class method should through the same checked exception or it's parent otherwise we will get compile time error.

But there are no restrictions for unchecked exceptions.

Invalid case:



```
import java.io.*;
class P
{
   public void m1() throws IOException
   {
   }
}
 class C extends P
{
   public void m1() throws EOFException,InterruptedExceptio
   {
   }
}
```



✓ ① P: public void m1() throws Exception
C: public void m1()

✗ ② P: public void m1()
C: public void m1() throws Exception

✓ ③ P: public void m1() throws Exception
C: public void m1() throws IOException

✗ ④ P: public void m1() throws IOException
C: public void m1() throws Exception

✓ ⑤ P: public void m1() throws IOException
C: public void m1() throws FileNotFoundException, EOFException

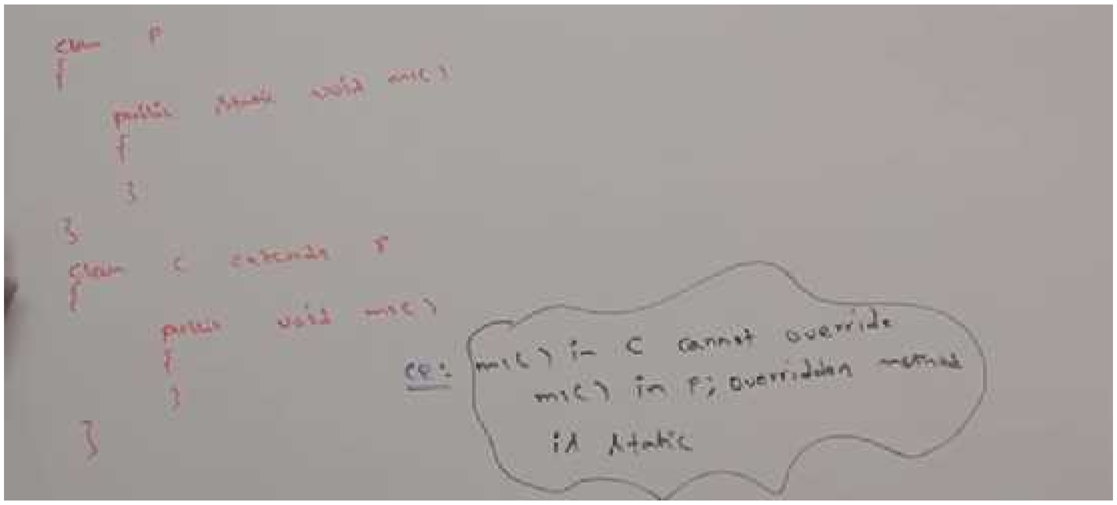
✗ ⑥ P: public void m1() throws IOException
C: public void m1() throws EOFException, I

✓ ⑦ P: public void m1() throws IOException
C: public void m1() throws AE, NPE, CCE

Overriding w.r.t static methods:

Case1:

We can't override a static method as non static otherwise we will get compiletime error.

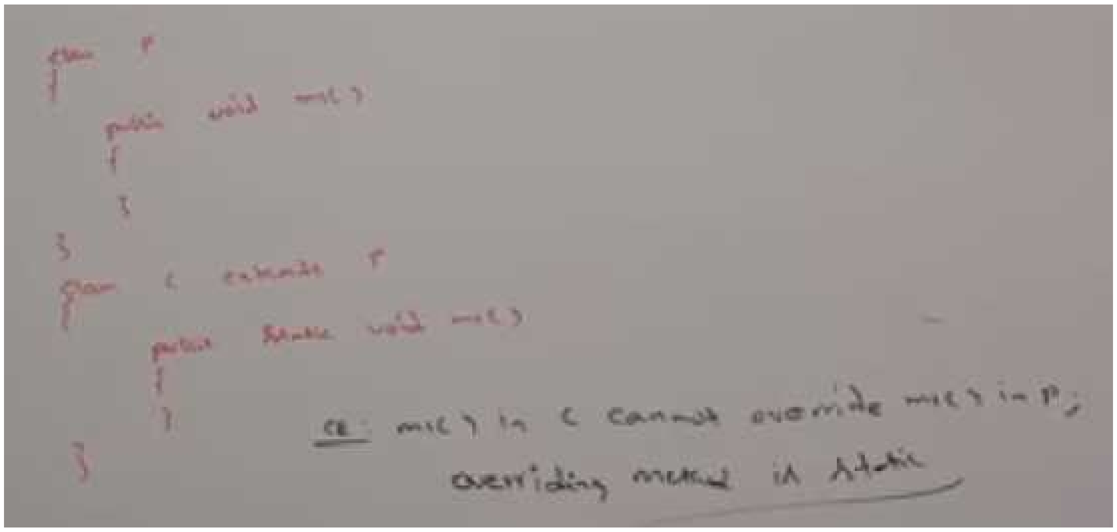


```
class P
{
    public static void m1()
    {
    }
}
class C extends P
{
    public void m1()
    {
    }
}
```

CE: m1() in C cannot override m1() in P; overridden method is static

case 2:

similarly we can't override a non static method as static.

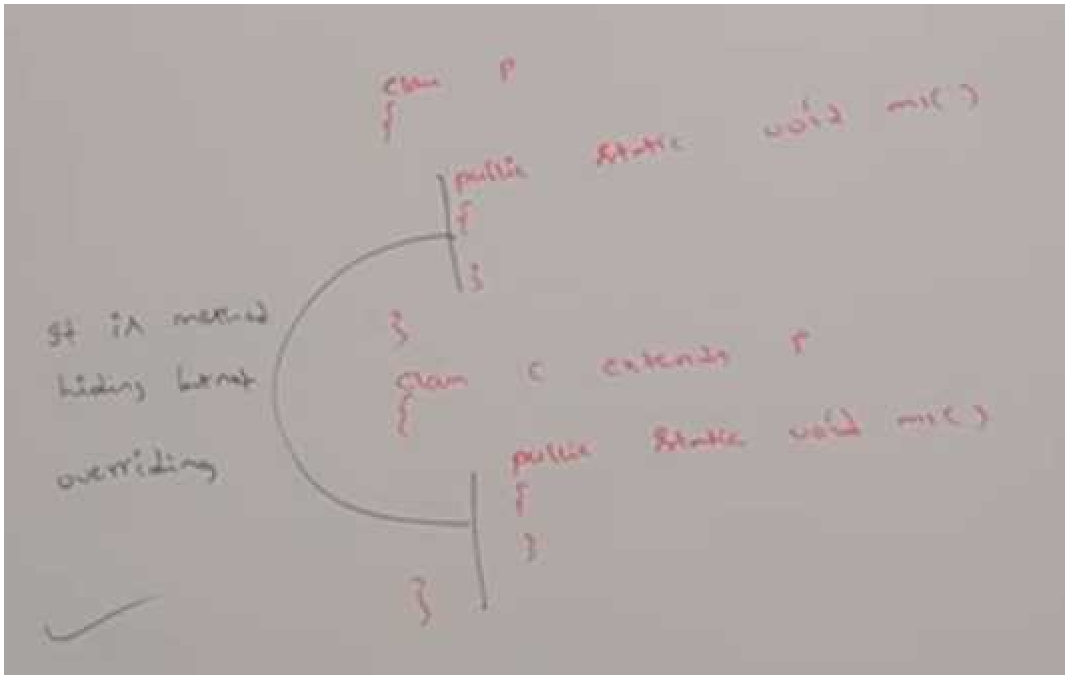


```
class P
{
    public void m1()
    {
    }
}
class C extends P
{
    public static void m1()
    {
    }
}
```

CE: m1() in C cannot override m1() in P; overriding method is static

Case 3:

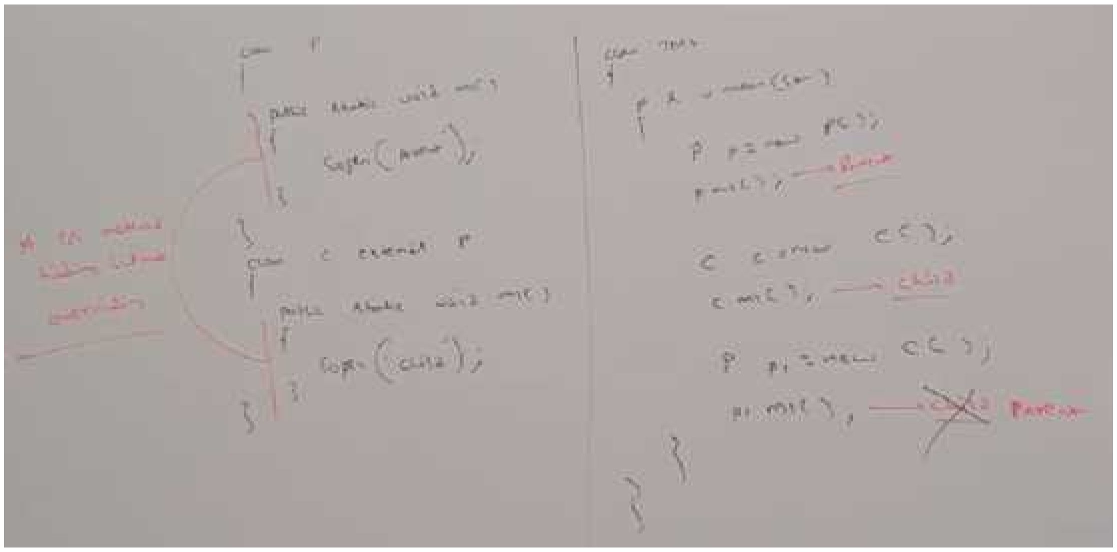
If both parent and child class methods are static then we won't get any compile time error, it seems overriding concept applicable for static methods but it is not overriding and it is method hiding.



```
class P
{
    public static void m1()
    {
    }
}
class C extends P
{
    public static void m1()
    {
    }
}
```

it is method hiding but not overriding

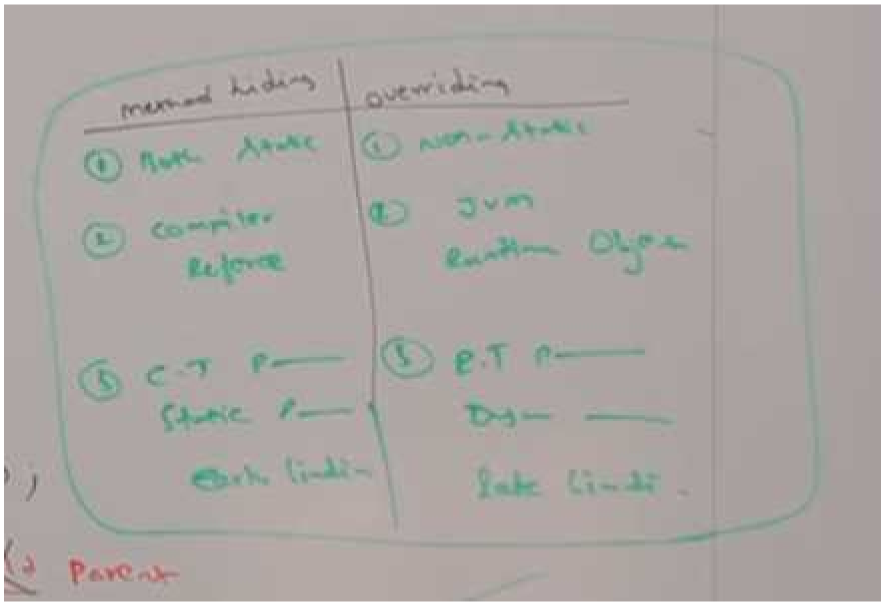
Method hiding:



```
class P
{
    public static void m1()
    {
        Sopln("Parent");
    }
}
class C extends P
{
    public static void m1()
    {
        Sopln("Child");
    }
}

class Test
{
    p s v main(S[] a)
    {
        P p = new P();
        p.m1(); → Parent

        C c = new C();
        c.m1(); → Child

        P p1 = new C();
        p1.m1(); → ~~Child~~ Parent
    }
}
```

it is method hiding but not overriding

All rules of method hiding exactly same as overriding except the following differences,



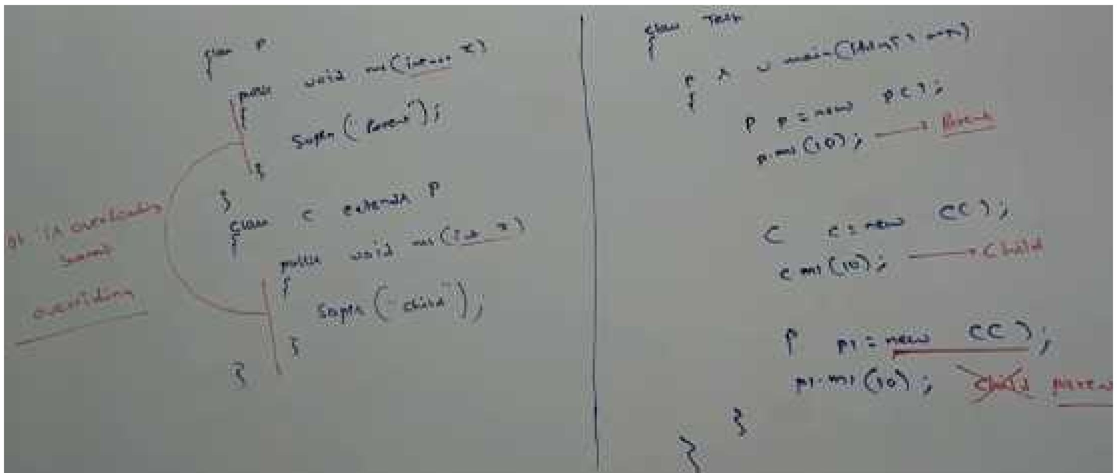
method hiding	overriding
① both static	① non-static
② compiler Reference	② JVM Runtime Object
③ C.T P—— Static P—— Early binding	③ R.T P—— Dyn—— —— late binding

Parent

Overriding w.r.t var-arg methods:

We can override var-arg method with another var-arg method only.

If we are trying to override with normal method then it will become overloading but not overriding.



```
class P
{
    public void m1(int... x)
    {
        Sopln("Parent");
    }
}
class C extends P
{
    public void m1(int x)
    {
        Sopln("child");
    }
}
```

it is overloading but not overriding

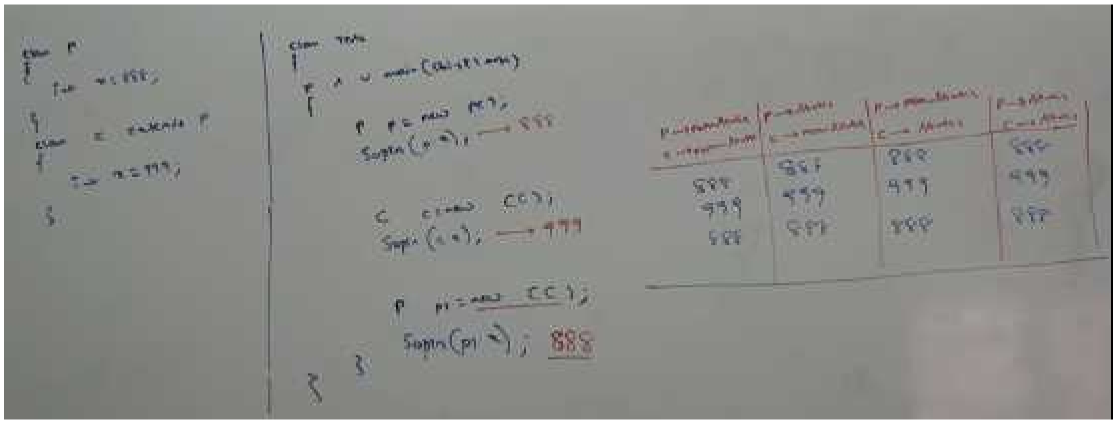
```
class Test
{
    p s v main(String[] args)
    {
        P p = new P();
        p.m1(10);  ——→ Parent

        C c = new C();
        c.m1(10);  ——→ Child

        P p1 = new C();
        p1.m1(10);  ~~Child~~ Parent
    }
}
```

If we replace child method with var-arg method then it will become overriding.

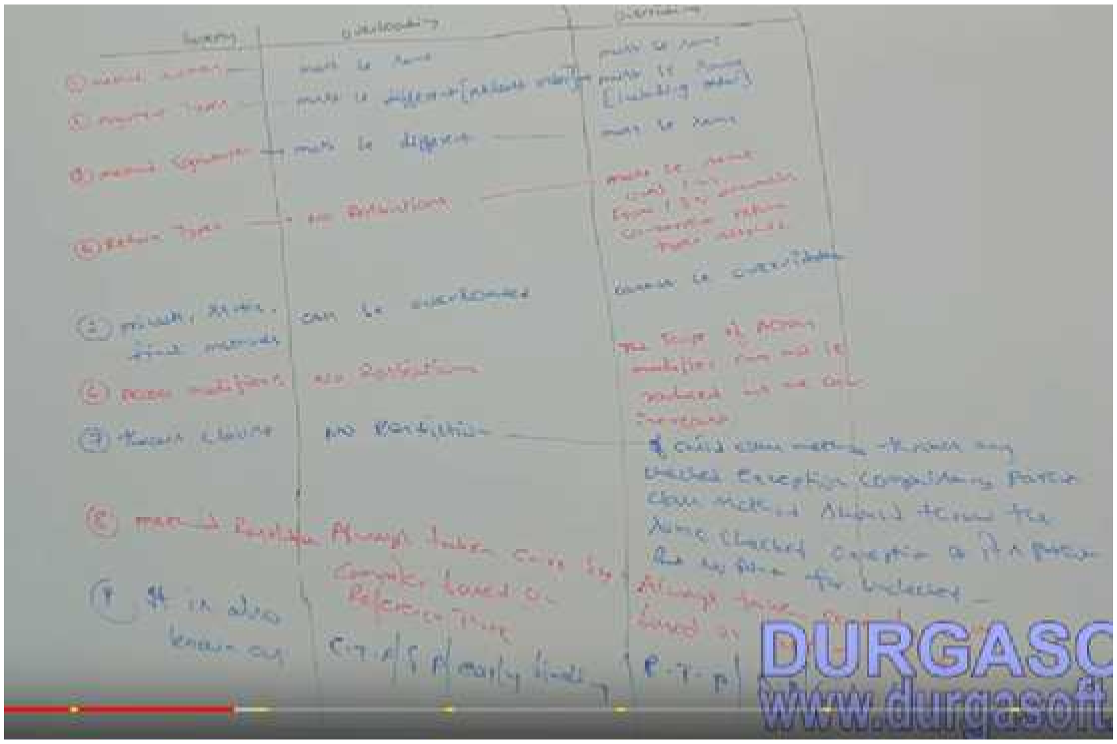
In this case the output is parent, child, child

Overriding w.r.t Variables:

Variable resolution always takes care by compiler based on reference type irrespective of whether the variable is static or non static. (Overriding concept applicable only for methods but not for variables.)

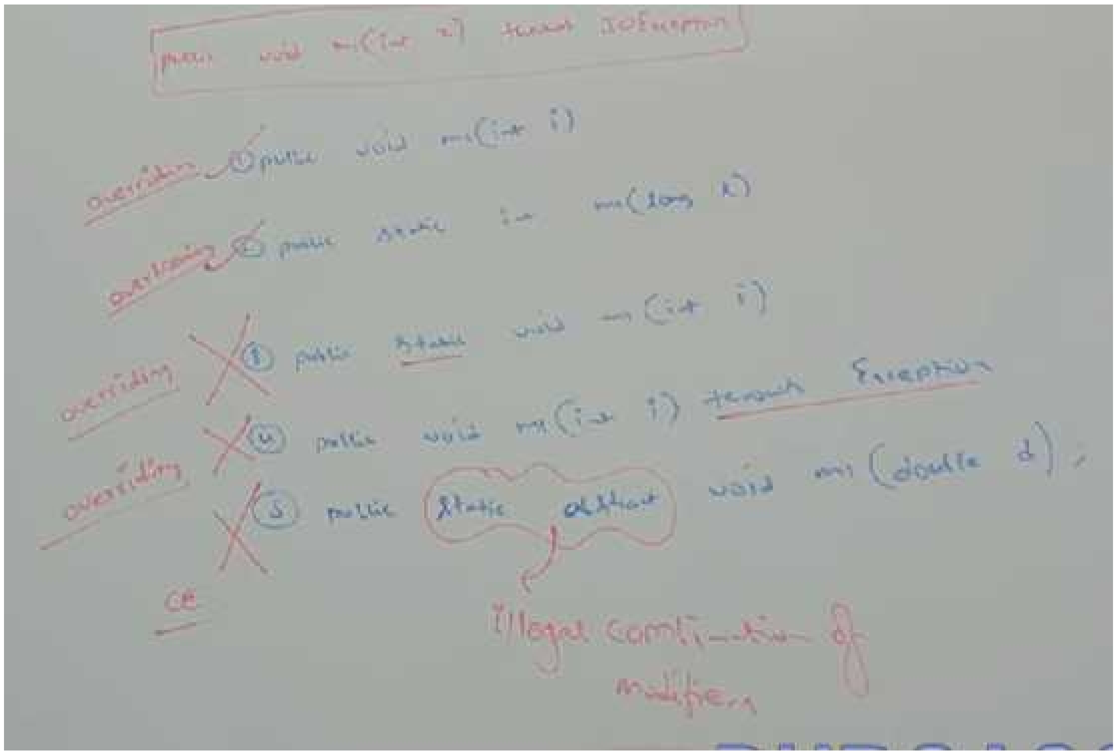


Overloading vs Overriding:



Property	Overloading	Overriding
① method names	must be same	must be same
② argument types	must be different [at least order]	must be same [including order]
③ method signature	must be different	must be same
④ Return types	no Restrictions	must be same until 1.4v from 1.5v onwards co-variant return types allowed
⑤ private, static, final methods	can be overloaded	cannot be overridden
⑥ access modifiers	no Restrictions	the scope of access modifiers can not be reduced but we can increase
⑦ throws clause	no Restriction	if child class method throws any checked exception compulsory parent class method should throw the same checked exception or it's parent but no restrictions for unchecked
⑧ method Resolution	Always taken care by compiler based on Reference Type	Always take [illegible] based on [illegible]
⑨ It is also known as	C.T.P / S.P / early binding	R.T.P [illegible]

Red color method in parent class,



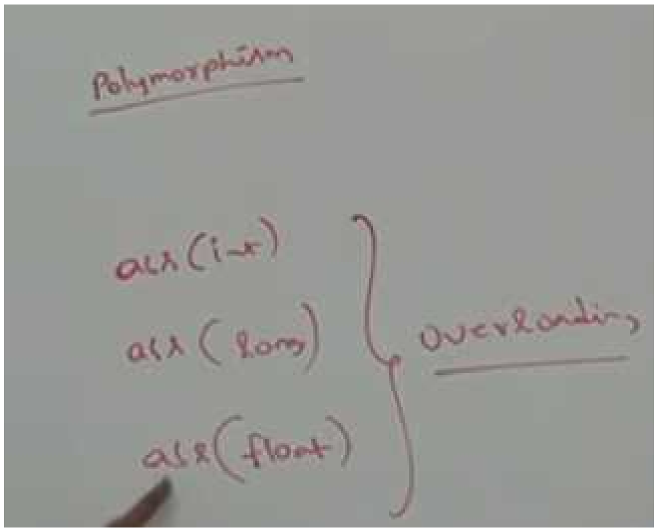
```
public void m1(int x) throws IOException

overriding ✓ ① public void m1(int i)
overloading ✓ ② public static int m1(long x)
overriding ✗ ③ public static void m1(int i)
overriding ✗ ④ public void m1(int i) throws Exception
           ✗ ⑤ public static abstract void m1(double d);
CE                         Illegal combination of modifiers
```

Ploymorphisem:

One name but multiple forms is the concept of poliymorphisem.

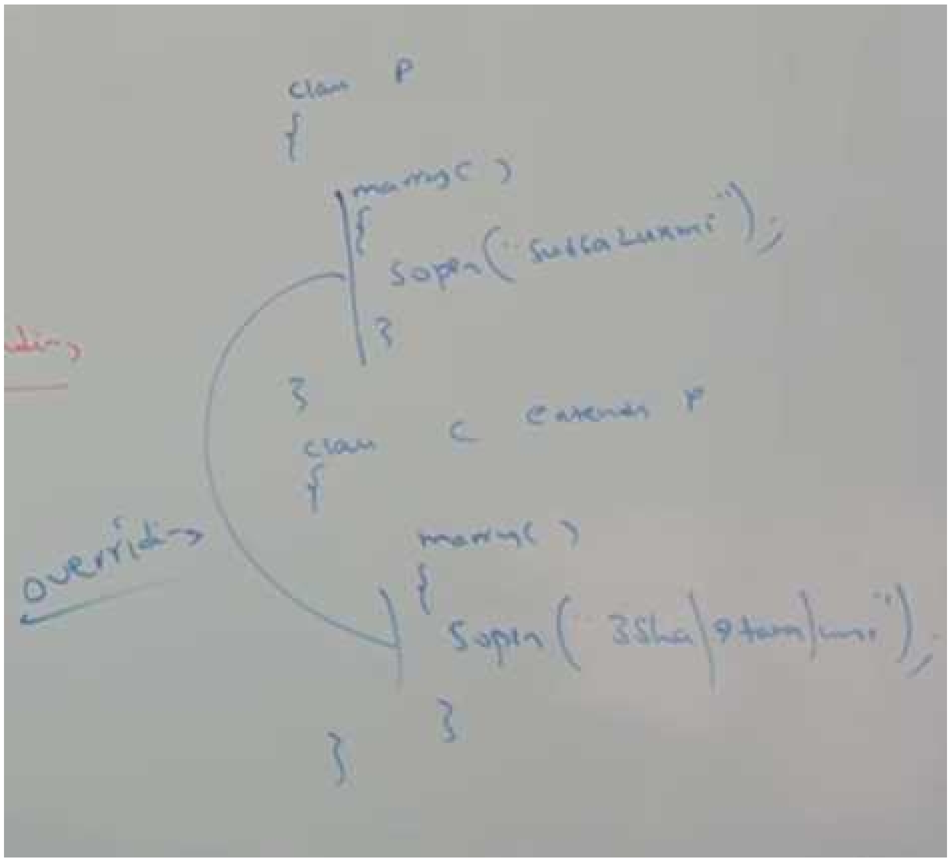
e.g. 1

method name is the same but we can apply for different types of arguments(Overloading)



e.g.2

Method signature is same but in parent class one type of implementation and in the child class another type of implementation(overriding).

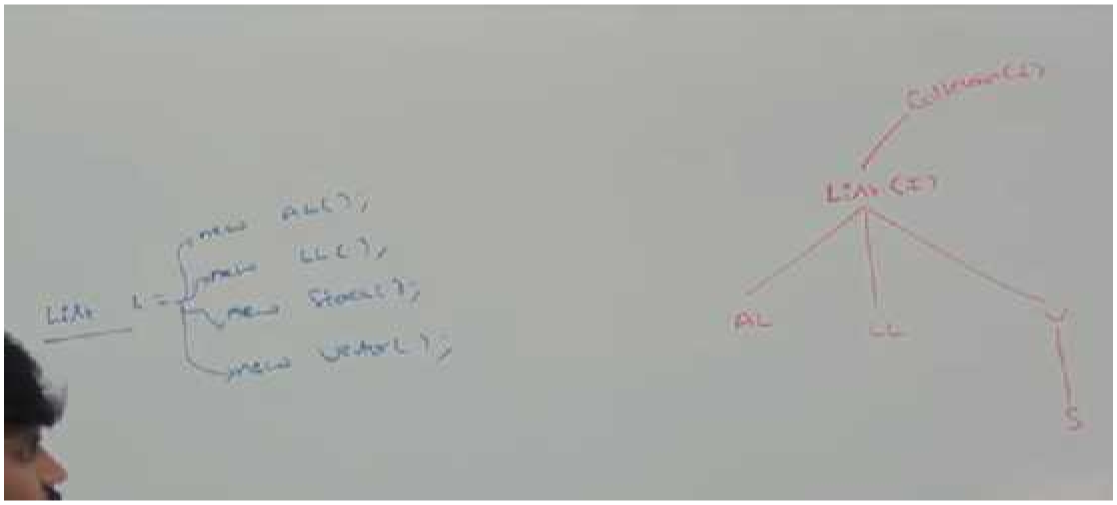

overriding

Handwritten code on the whiteboard:

```
class P
{
    marry()
    {
        sopn("Subbalaxmi");
    }
}
class C extends P
{
    marry()
    {
        sopn("3Sha|9tam|umi");
    }
}
```

e.g.3

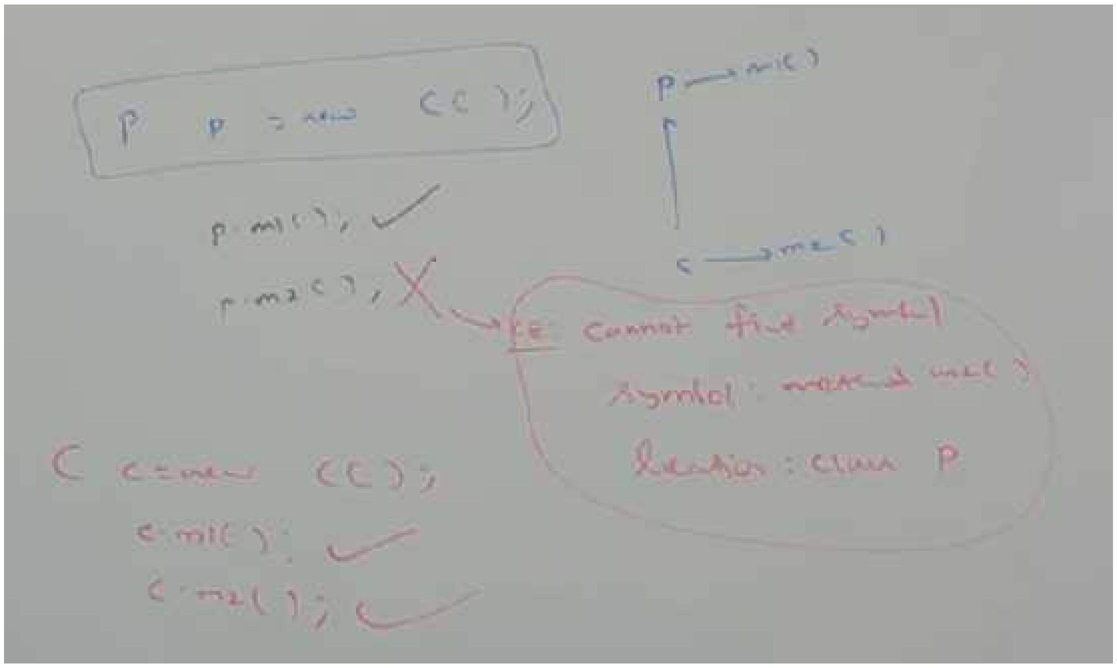
Usages of parent reference to hold child object is the concept of polymorphisem.



Handwritten code on the whiteboard:

```
List l = new AL();
         new LL();
         new Stack();
         new Vector();
```

Parent class reference can be used to hold child object but by using that refernce we can call only the methods available in parent class and we can't call child specific methods.



```
P p = new C();
p.m1(); ✓
p.m2(); X → CE: cannot find symbol
            symbol: method m2()
            location: class P

C c = new C();
c.m1(); ✓
c.m2(); ✓
```

But by using child refernce we can call both parent and child class methods.

Q. When we should go for parent reference to hold child object ?

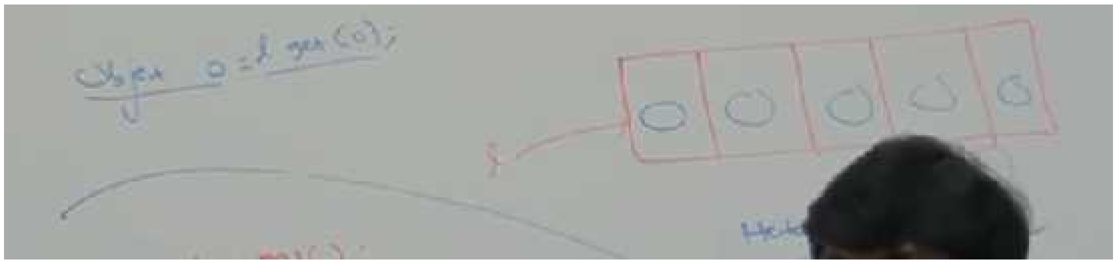
Ans:

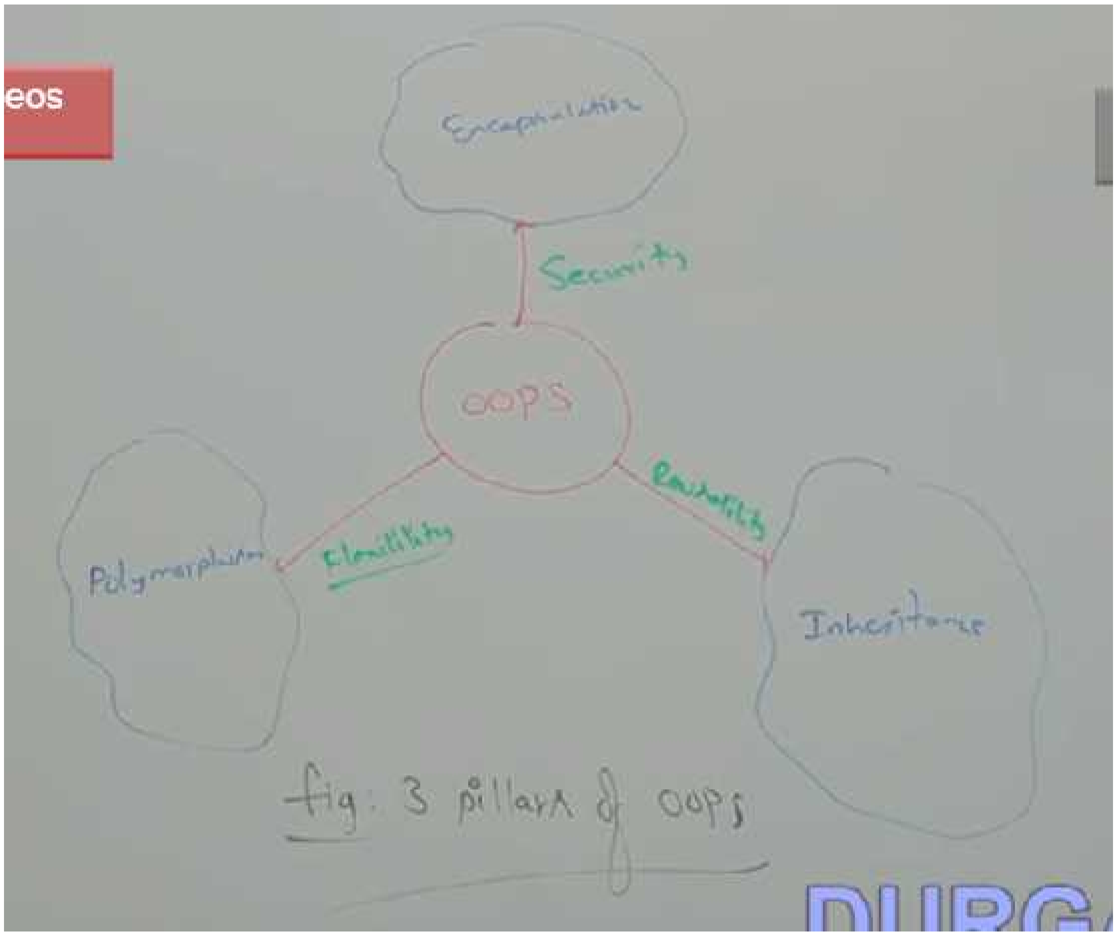
If we don't know exact runtime type of object then we should go for parent reference.

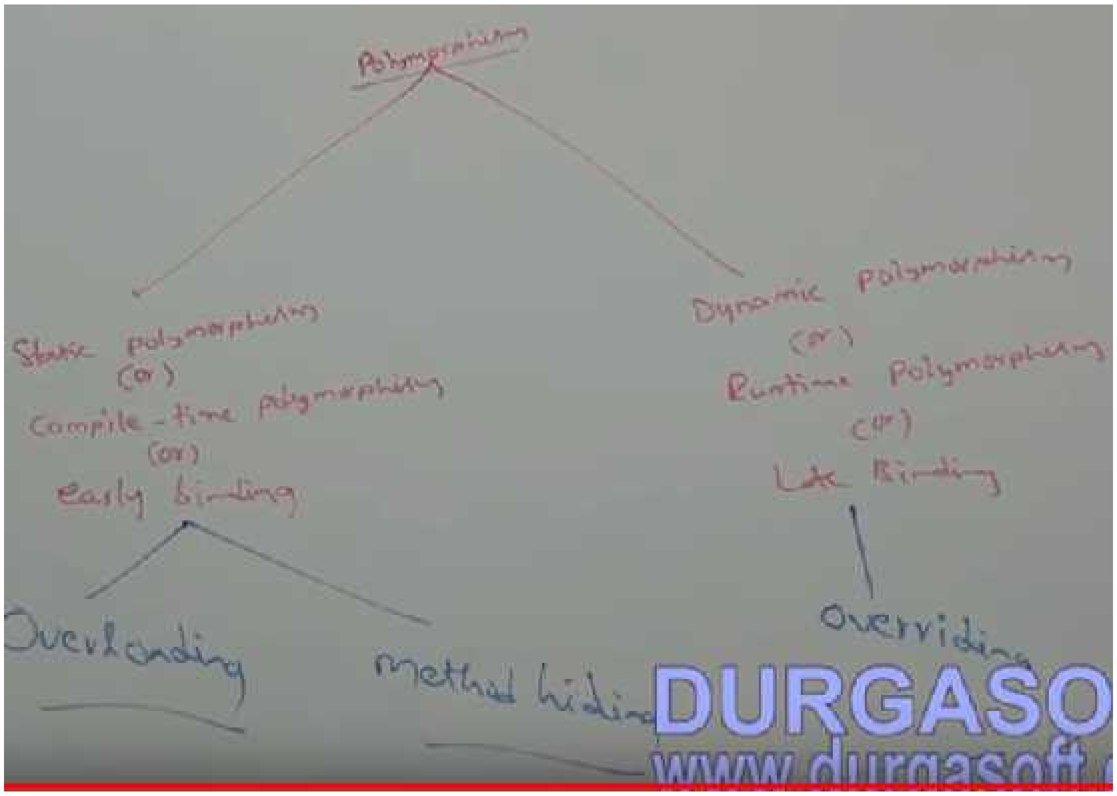
e.g.

The first element present in the arrayList can be any type. It may be Student object, String Object or Customer

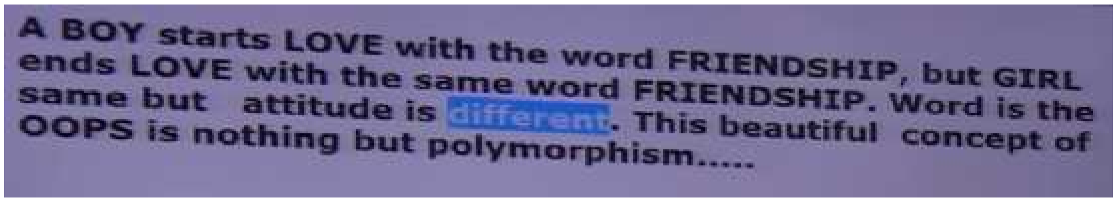
Object, hence the return type of get method is object, which can hold any object.







Beautiful definition of Polymorphisem:



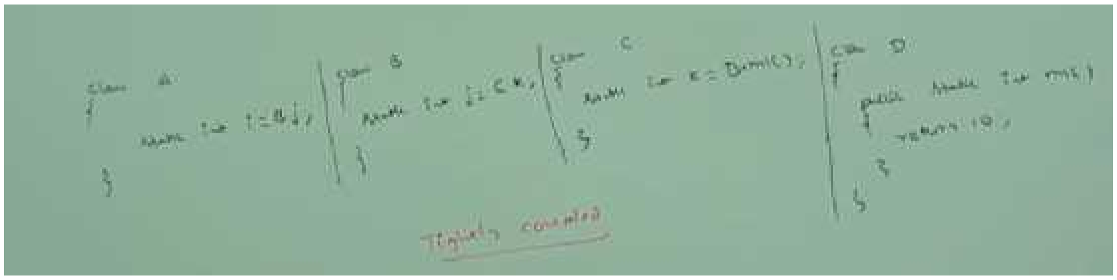
A BOY starts LOVE with the word FRIENDSHIP, but GIRL ends LOVE with the same word FRIENDSHIP. Word is the same but attitude is different. This beautiful concept of OOPS is nothing but polymorphism.....

Coupling:

The degree of dependency between the components is called coupling.

If dependency is more then it is considered as tightly coupling and if dependency is less then it is considered as loosely coupling.

e.g.



The above components are said to be tightly coupled with each other bcz dependency between the components is more.

Tightly coupling is not a good programming practice bcz it has several serious disadvantadges.

1. Without effecting remaining components we can't modify any component and hence enhancement will become difficult.

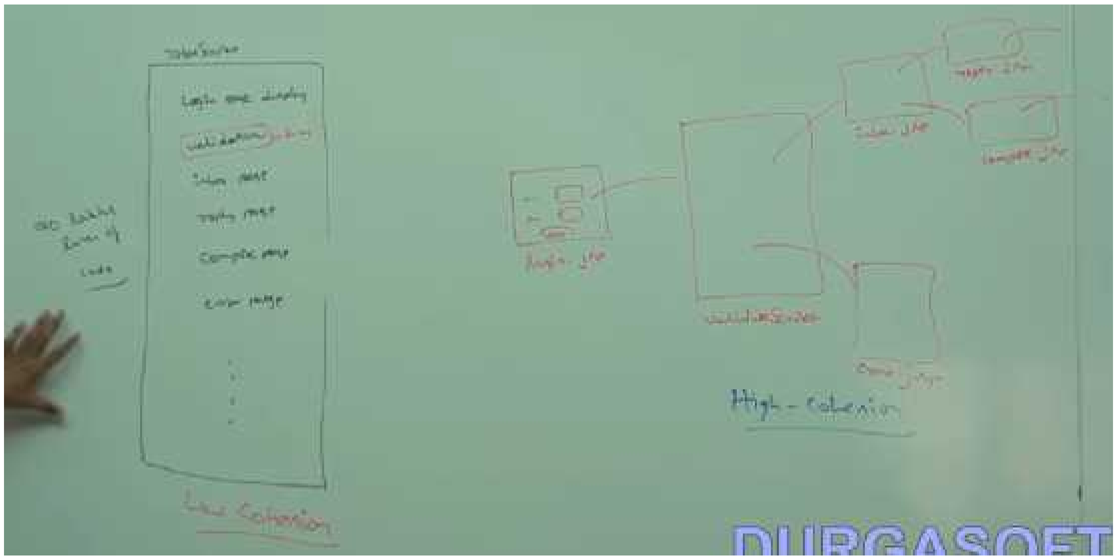
2. It suppress reusability.

3. It reduces maintainability of the application.

Hence we have to maintain dependency between the components as less as possible i.e. loosely coupling is a good programming practice.

Cohesion:

For every component a clear well defined functionality is defined, then that component is said to be follow high cohesion.



High cohesion is always a good practice bcz it has several advantadges

1. Without affecting remaining components we can modify any component, hence enhancement will become easy.

2. It promots reusability of the code. whereever validation is required we can reuse the same validation servlet without rewriting.

3. It improves maintainability of the application.

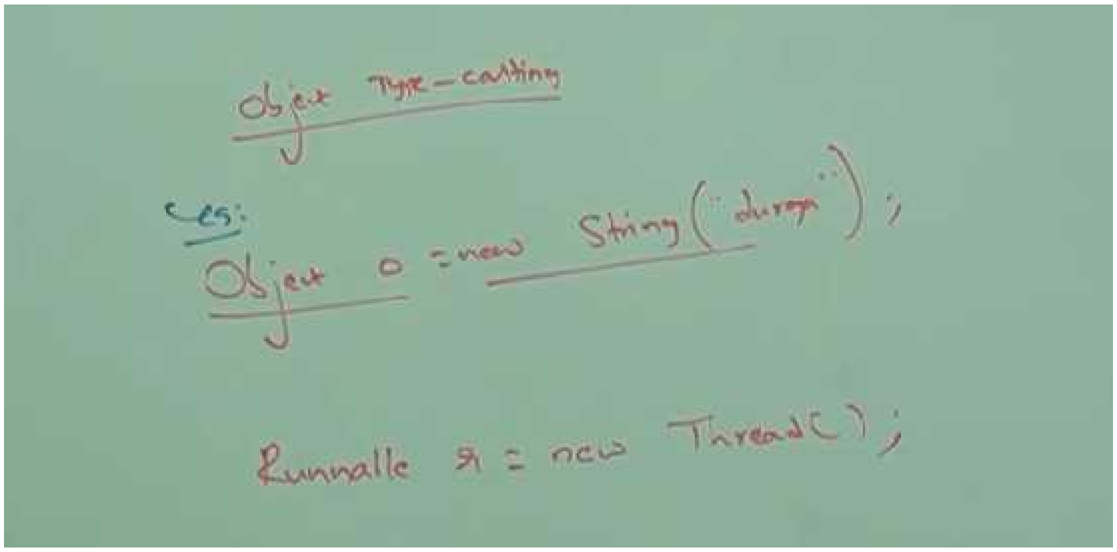
Note:

Loosely coupling and high cohesion are good programming practices.

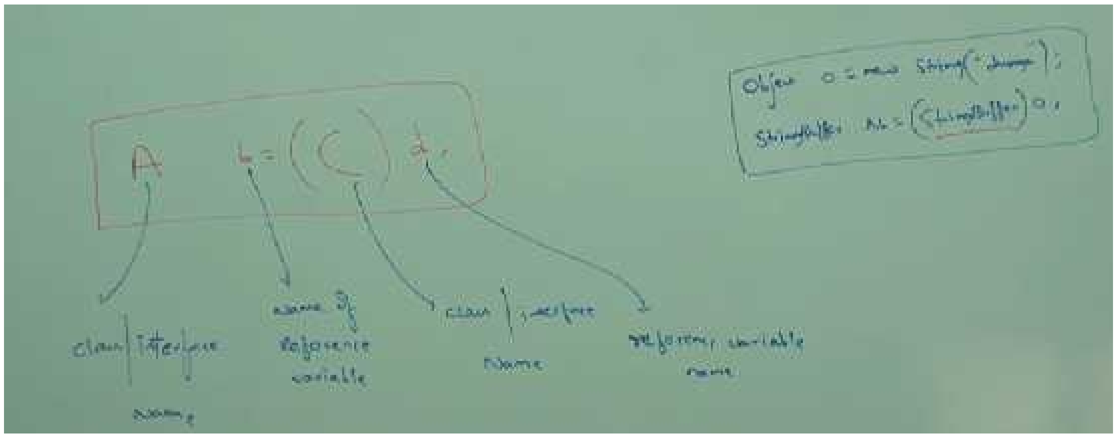
Object type casting:

We can use parent reference to hold child object.

e.g.


Object Type-casting
es:

```
Object o = new String("durga");

Runnable r = new Thread();
```



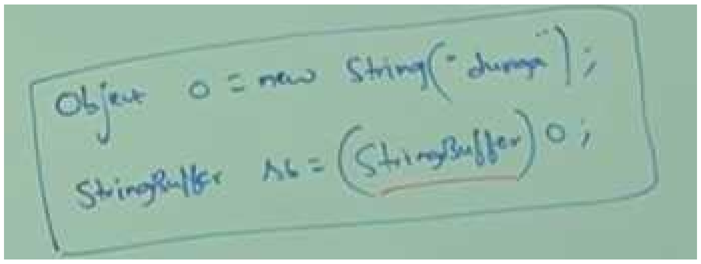
```
A b = (C) d;
```

```
Object o = new String("durga");
StringBuffer sb = (StringBuffer) o;
```

mantra 1 (Compiletime checking 1):

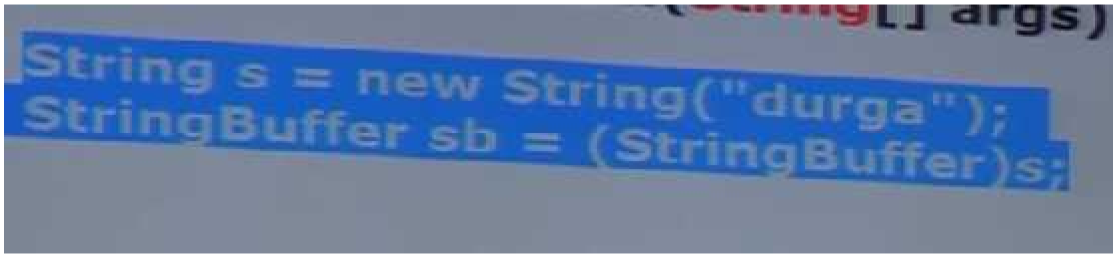
The type of 'd' and 'C' must have some relation, either child to parent or parent to child or same type, otherwise we will get compiletime error saying inconvertible type found 'd' type required 'C'.

e.g. 1



```
Object o = new String("durga");
StringBuffer sb = (StringBuffer) o;
```

e.g. 2



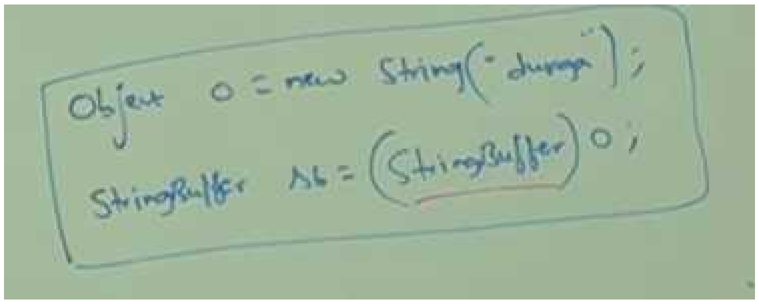
```
String s = new String("durga");
StringBuffer sb = (StringBuffer)s;
```

CE error: Incompactibe types

Mantra 2(Compiletime checking 2):

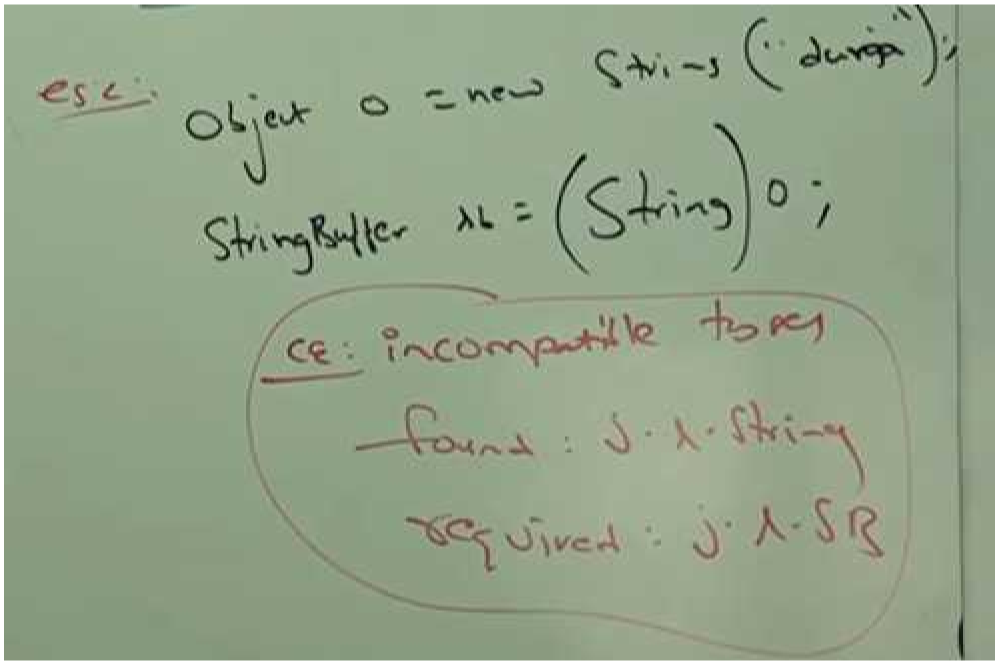
'C' must be either same or derived type of 'A' otherwise we will get compiletime error saying incompactibe types found 'C' required 'A'.

e.g. 1



```
Object o = new String("durga");
StringBuffer sb = (StringBuffer) o;
```

e.g. 2


esc:

```
Object o = new String("durga");
StringBuffer sb = (String) o;
```

ce: incompatible types

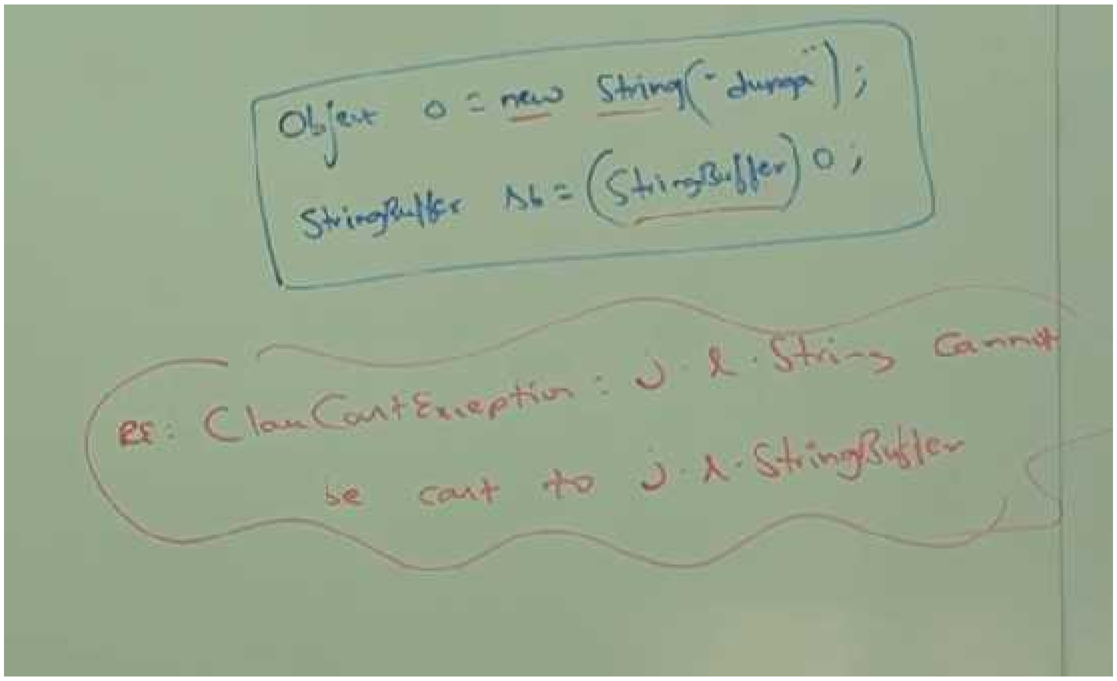
found : j.l.String

required : j.l.SB

Mantra 3(Runtime checking):

Runtime object type of 'd' must be either same or derived type of 'C', otherwise we will get runtime exceptions sayiing ClassCastException.

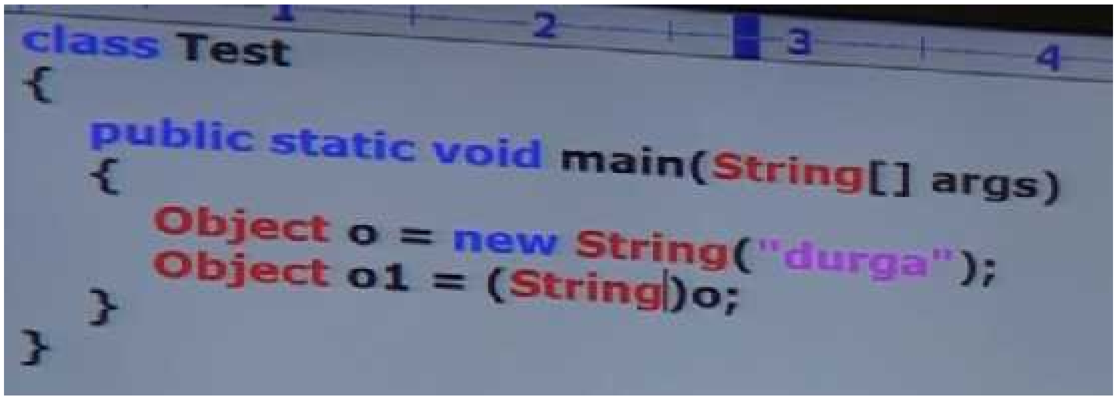
e.g 1:



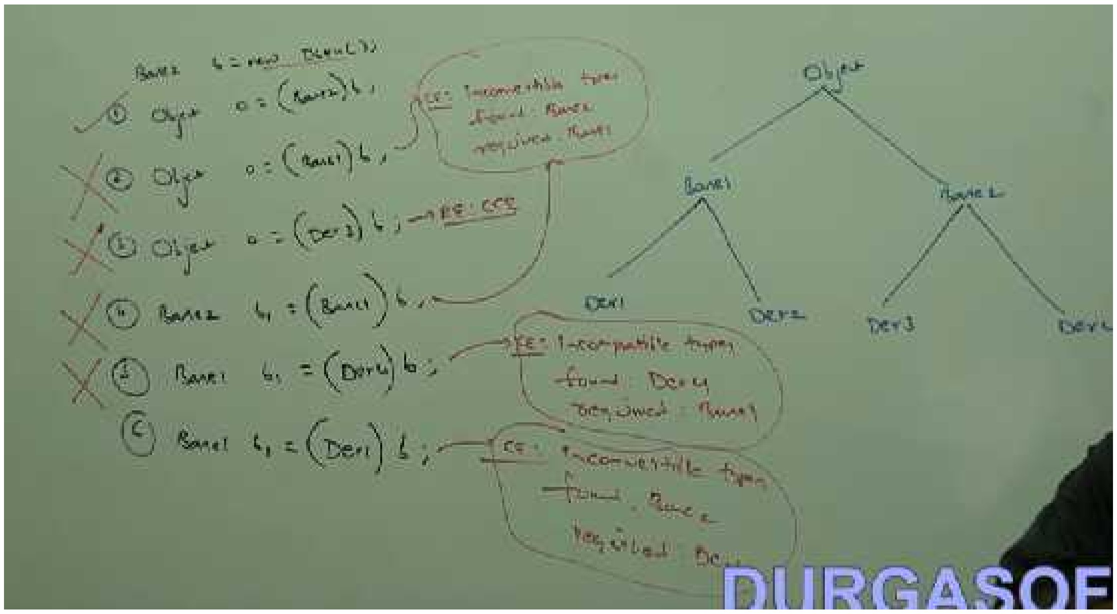
```
Object o = new String("durga");
StringBuffer sb = (StringBuffer) o;
```

RE: ClassCastException : j.l.String cannot be cast to j.l.StringBuffer

e.g 2:

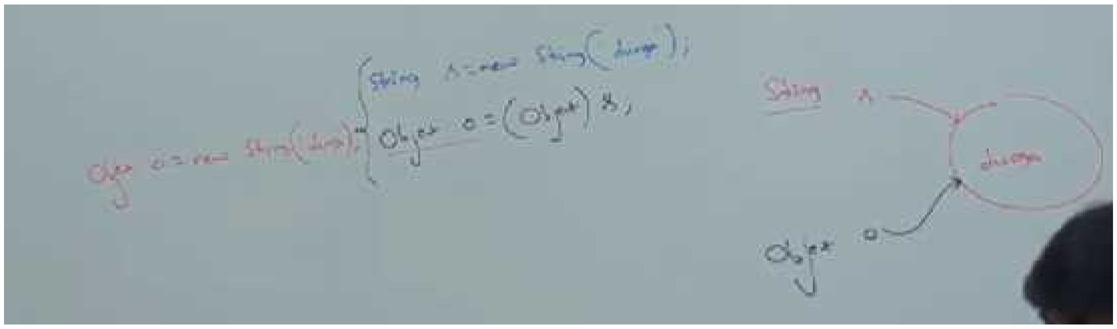


```
class Test
{
   public static void main(String[] args)
   {
      Object o = new String("durga");
      Object o1 = (String)o;
   }
}
```



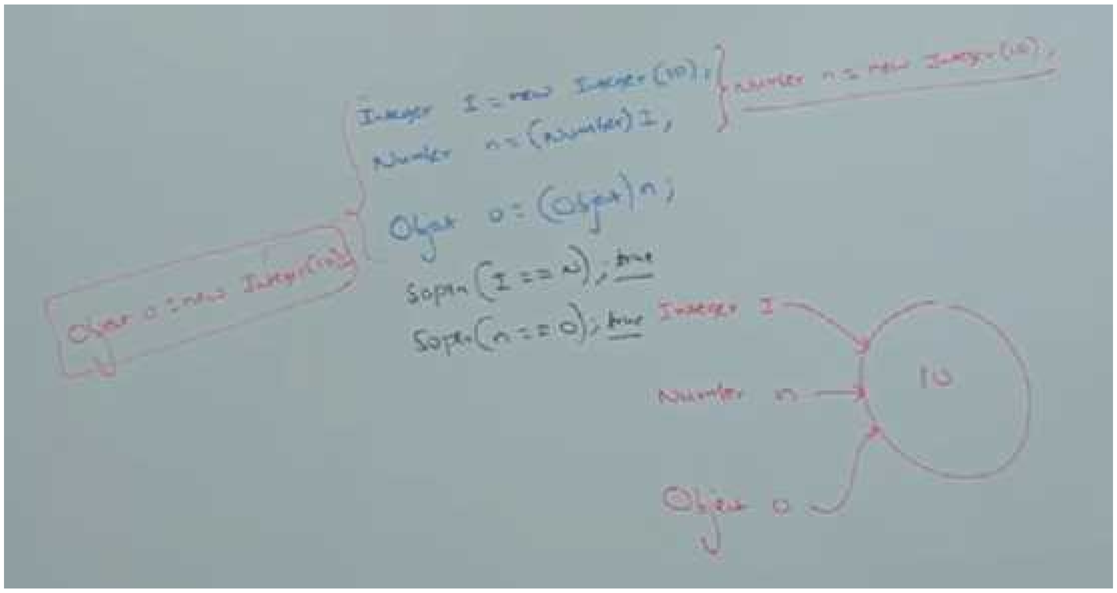
```
Base2 b = new Base2();
① Object o = (Base2) b;
② Object o = (Base1) b;  CE: incompatible types
                             found: Base2
                             required: Base1
③ Object o = (Der3) b;  → RE: CCE
④ Base2 b1 = (Base1) b;
⑤ Base1 b1 = (Der4) b;  → CE: incompatible types
                             found: Der4
                             required: Base1
⑥ Base1 b1 = (Der1) b;  → CE: incompatible types
                             found: Base2
                             required: De
```

Strictly speaking through typecasting we are not creating any new object. For the existing object we are providing another type of reference variable i.e. we are performing typecasting but not object casting.



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

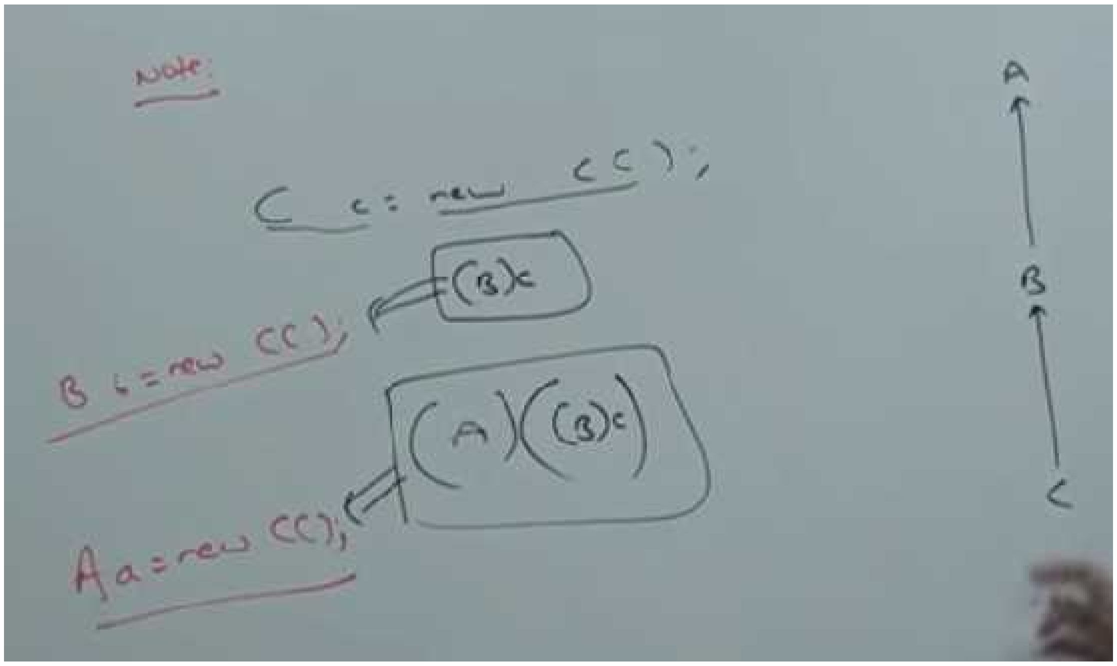
e.g.2



```
Integer I = new Integer(10);
Number n = (Number) I;
Object o = (Object) n;
Sopln(I == n); true
Sopln(n == o); true
```

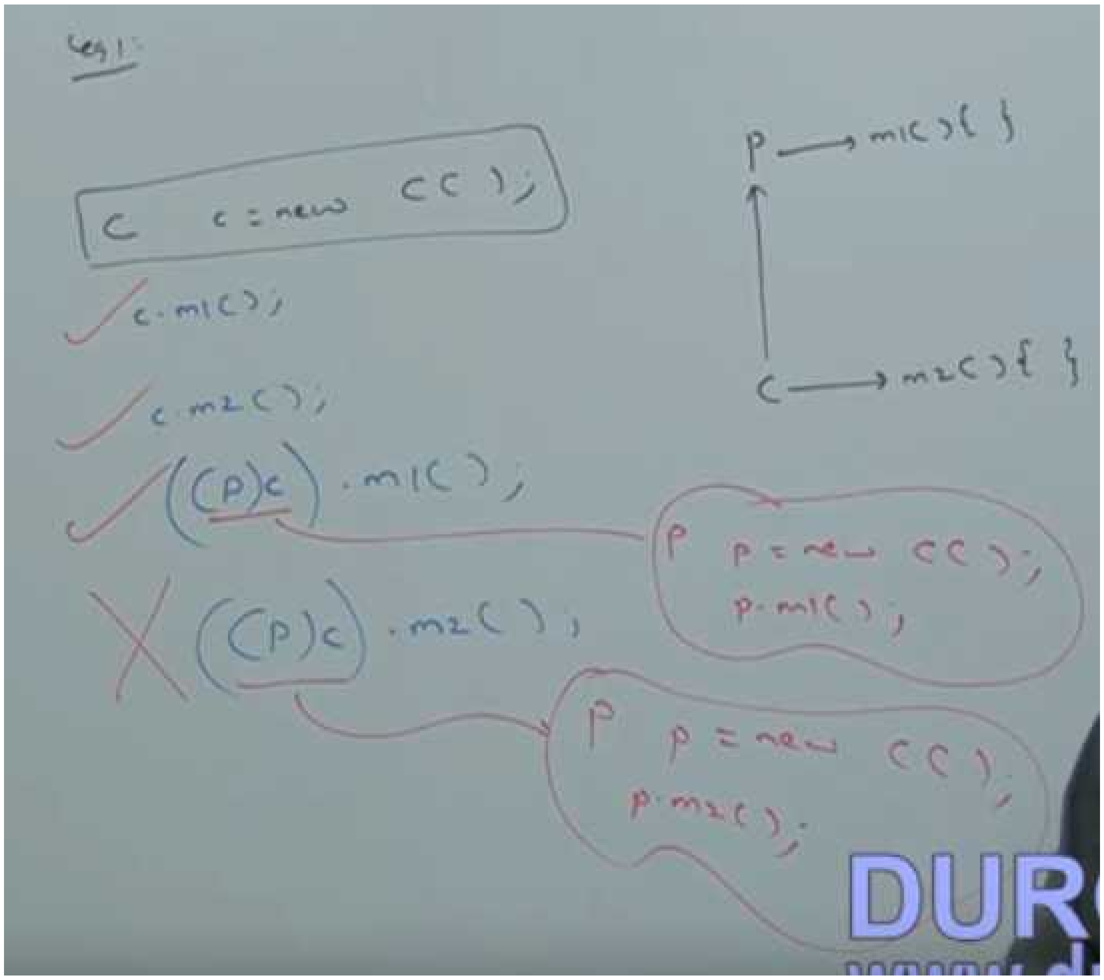
```
Number n = new Integer(10);
```

```
Object o = new Integer(10);
```



Note:

```
C c = new C();
B b = new C();
(B)c
A a = new C();
(A)((B)c)
```



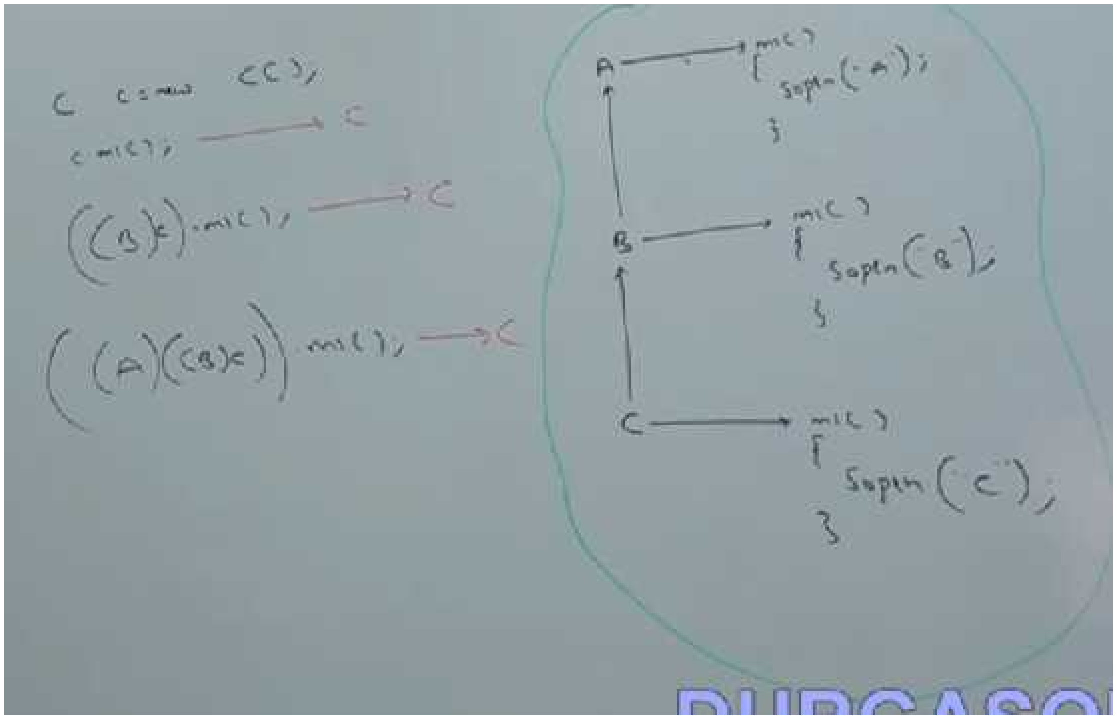
Eg 1:

```
C c = new C();
✓ c.m1();
✓ c.m2();
✓ ((P)c).m1();
✗ ((P)c).m2();
```

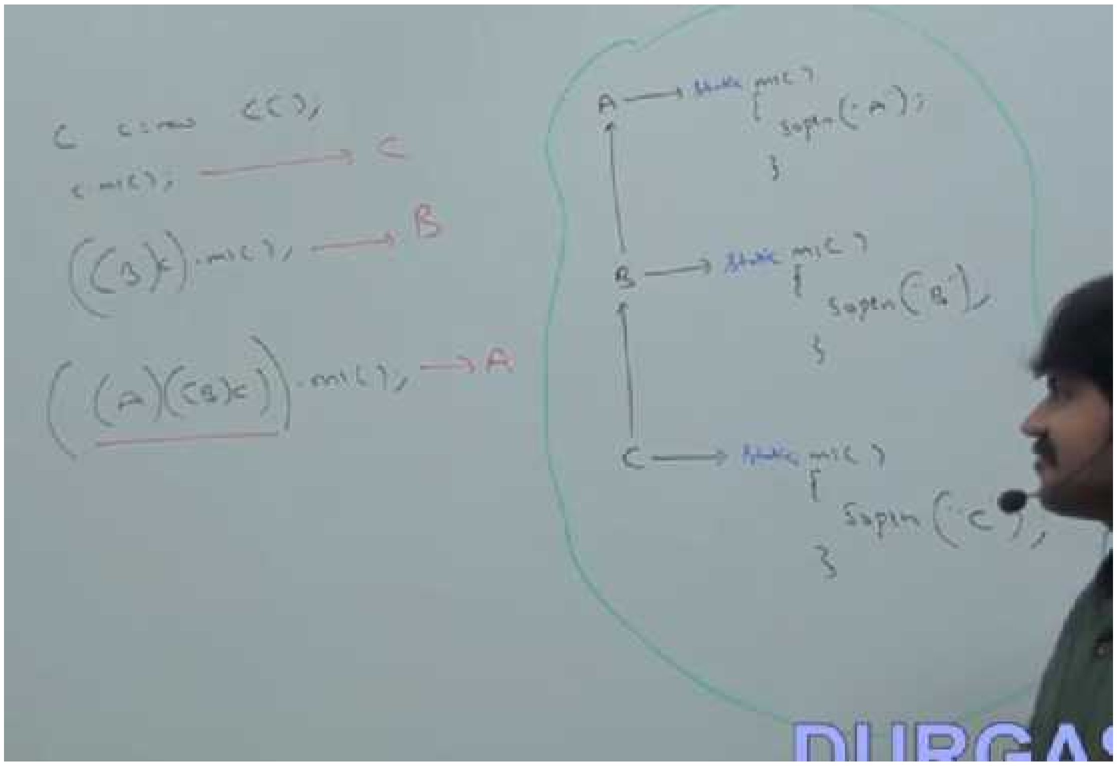
```
P p = new C();
p.m1();
```

```
P p = new C();
p.m2();
```

$$\begin{tikzcd} P \arrow[r] & m1()\{\ \} \\ C \arrow[u] \arrow[r] & m2()\{\ \} \end{tikzcd}$$



It is overriding and method resolution is always based on runtime object.



It is method hiding and method resolution is always based on reference type.

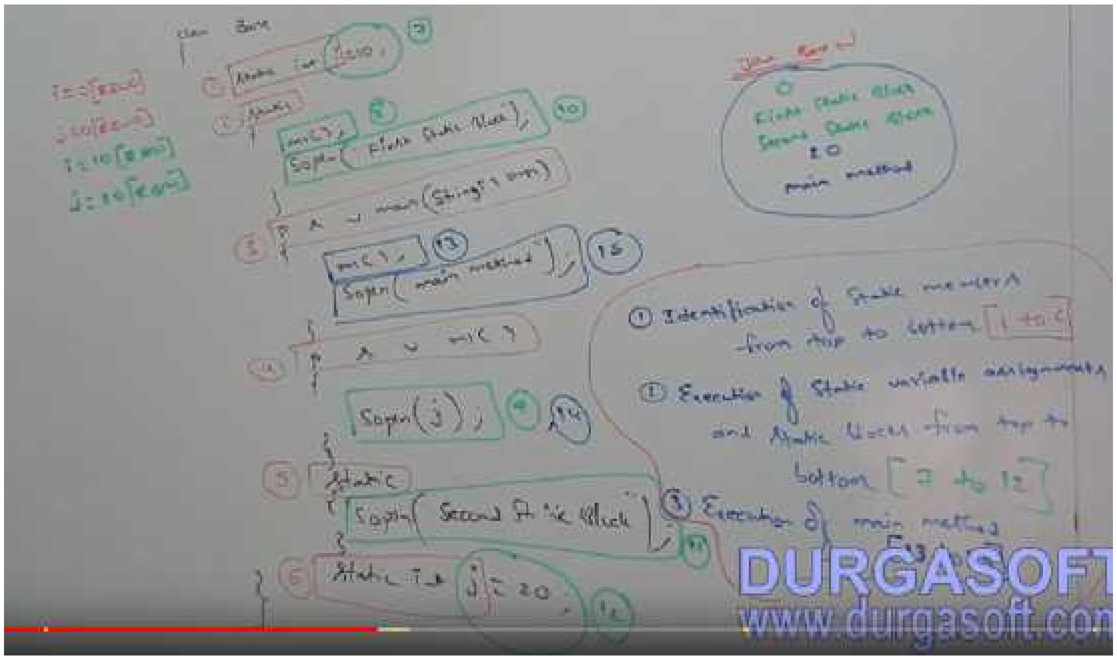
Static Control Flow:

Whenever we are executing a java class, the following sequence of steps will be executed as the part of static control flow,

1. Identification of static members from top to bottom.

2. Execution of static variable assignments and static blocks from top to bottom

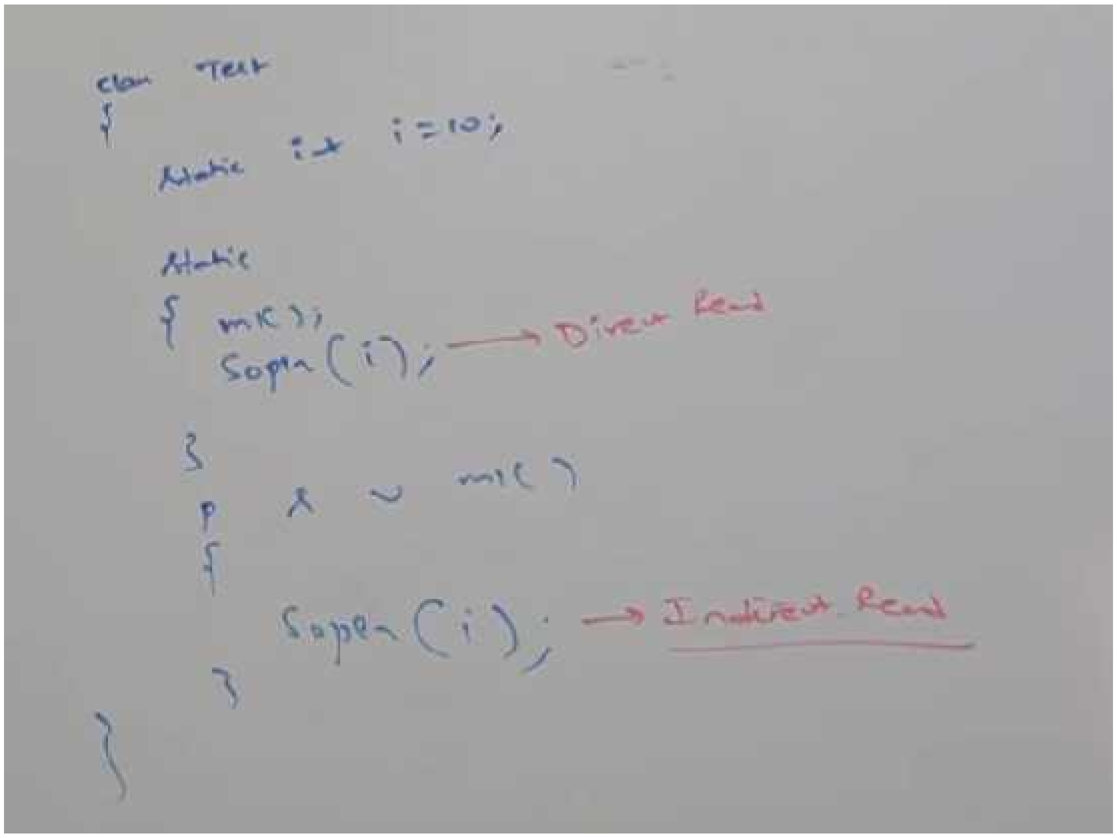
3. Execution of main method



Read indirectly and Write only:

Inside static block if we are trying to read a variable that read operation is called direct read.

If we are calling a method and within that method if we are trying to read a variable that read operation is called indirect read.

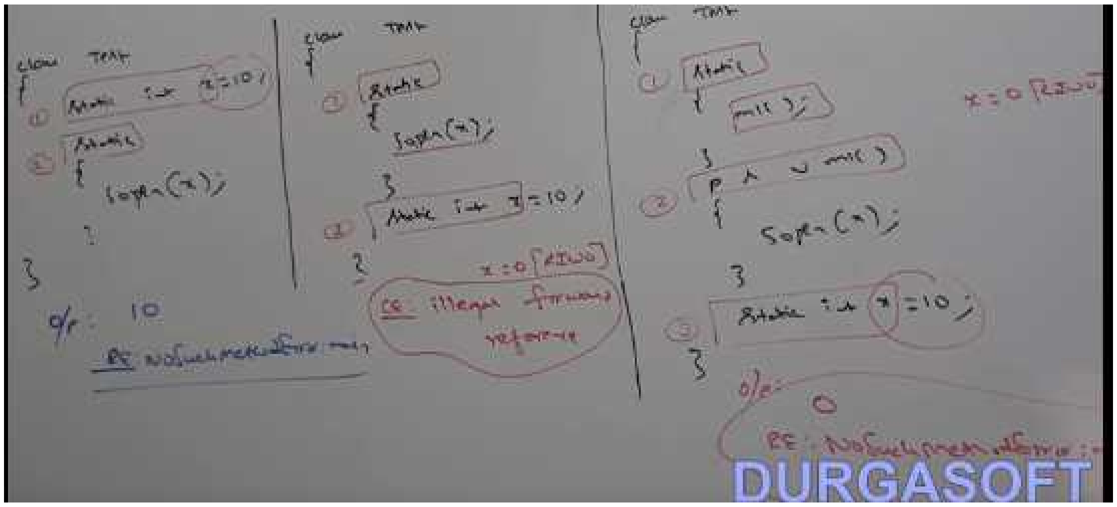


```
class Test
{
   static int i=10;

   static
   { m1();
     Sopln(i); ——> Direct Read

   }
    p s v m1()
    {
       Sopln(i); ——> Indirect Read
    }
}
```

If a variable is just identified by the JVM and original value not yet assigned then the variable is said to be in read indirectly and write only state.(RIWO)

If a variable is in RIWO state then we can't perform direct read but we can perform indirect read.

If we are trying to read directly then we will get compiletime error saying illigalForwardReference.

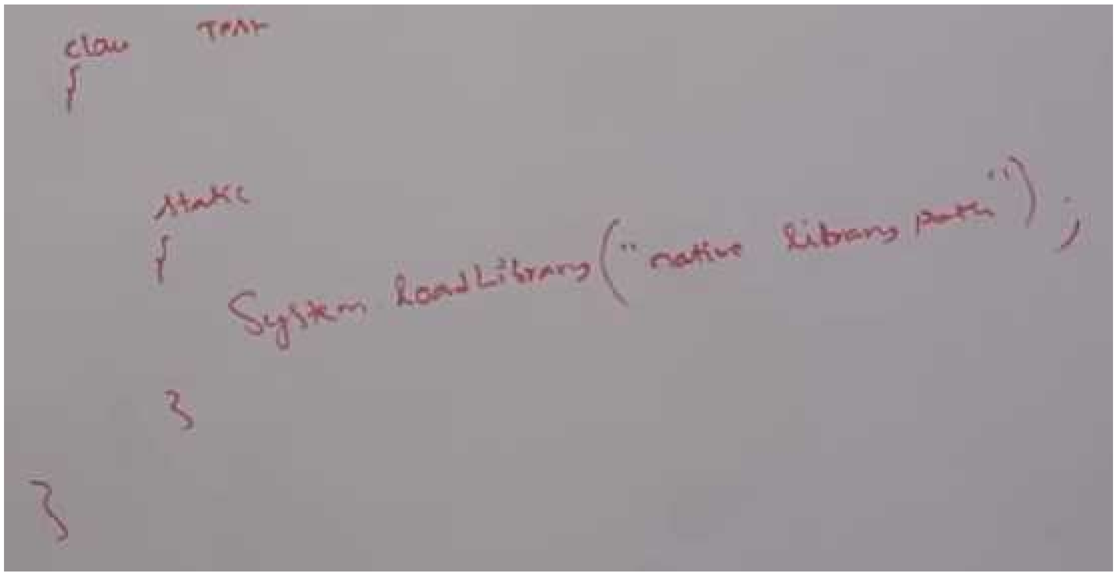


```
class Test
{
① static int x=10;
② static
  {
    sopln(x);
  }
}
o/p: 10
RE: NoSuchMethodError: main

class Test
{
① static
  {
    sopln(x);
  }
② static int x=10;
}
x=0 [RIWO]
CE: illegal forward reference

class Test
{
① static
  {
    m1();
  }
② p s v m1()
  {
    sopln(x);
  }
③ static int x=10;
}
x=0 [RIWO]
o/p: 0
RE: NoSuchMethodError: main
```

Static block:

static blocks will be executed at the time of class loading hence at the time of class loading if we want to perform any activity we have to define that inside static block.

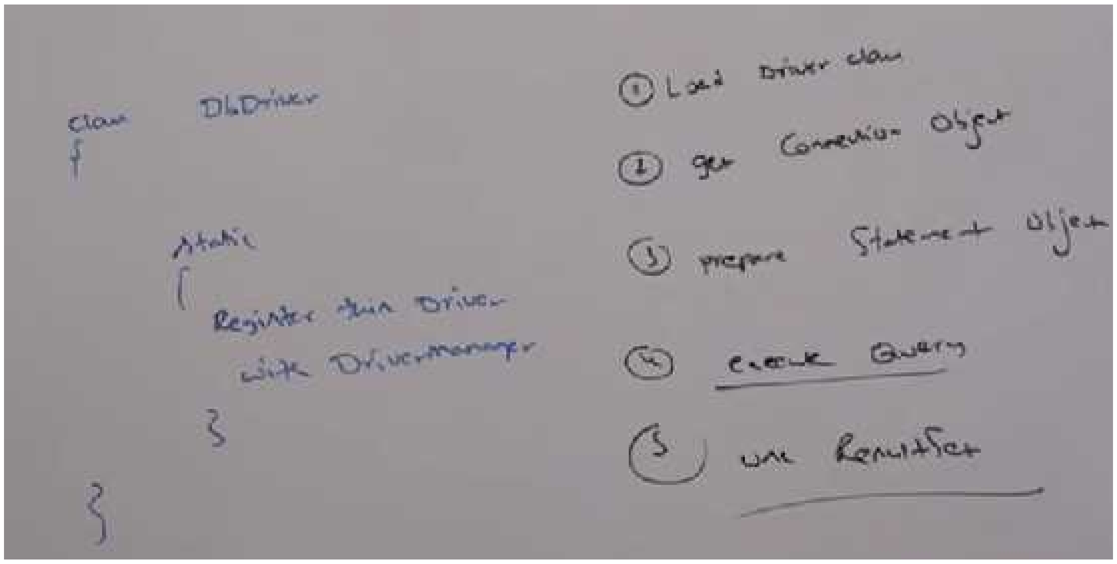
At the time of java class loading the corrosponding native libraries should be loaded, hence we have to define this activity inside static block.



```
class Test
{
    static
    {
        System.loadLibrary("native library path");
    }
}
```

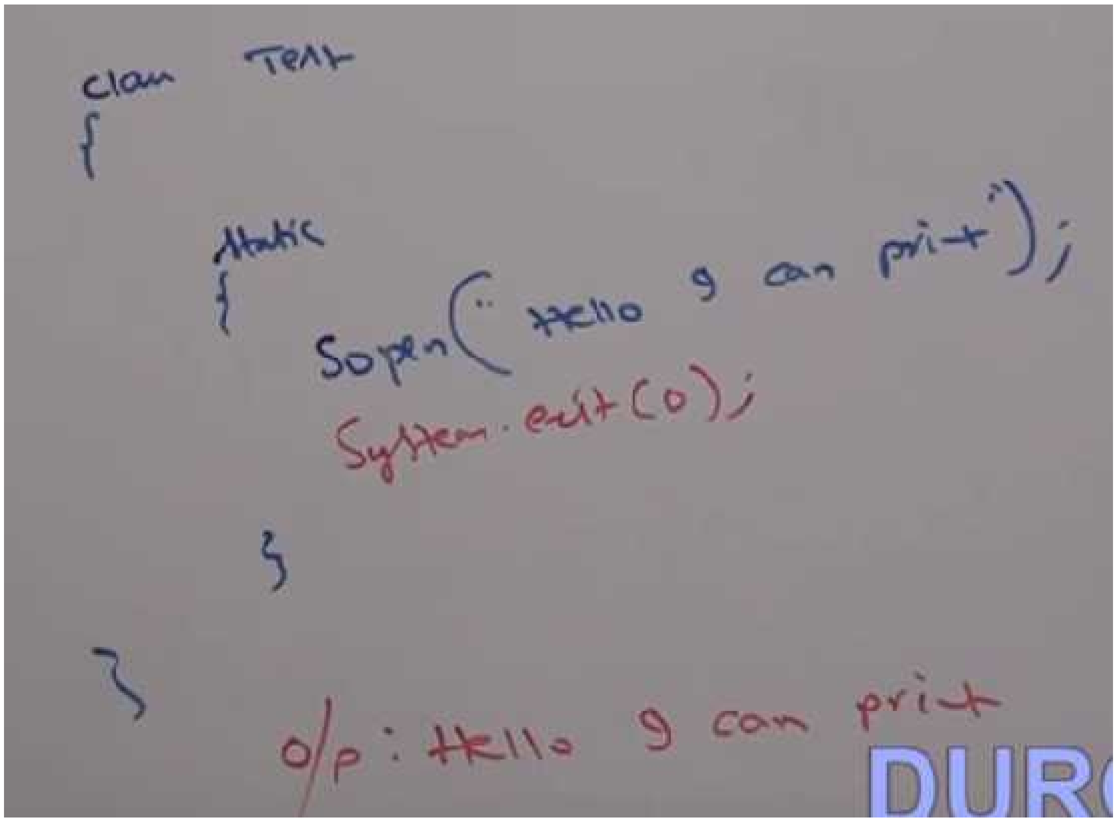
e.g.2

After loading every database driver class we have to register driver class with driver manager but inside database driver class there is a static block to perform this activity and we are not responsible to register explicitely.



Note:

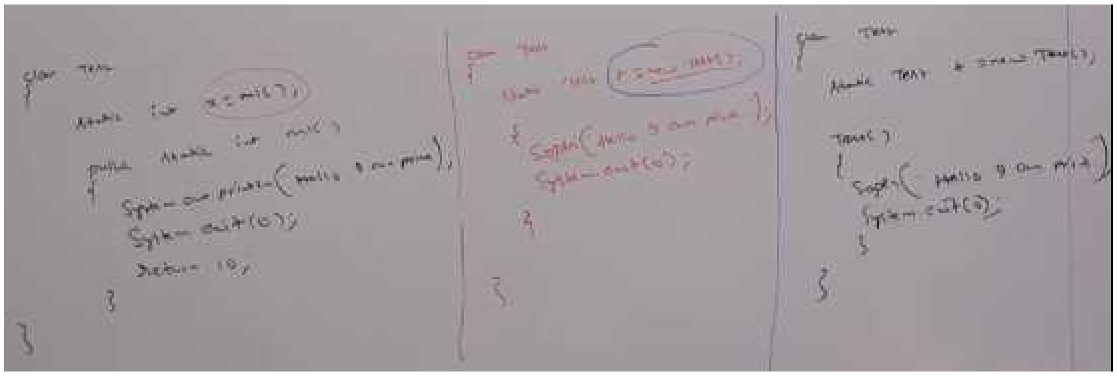
With in a class we can declare any number of static blocks but all these static blocks will be executed from top to bottom.



```
class Test
{
    static
    {
        Sopln("Hello I can print");
        System.exit(0);
    }
}
o/p: Hello I can print
```

Q. Without writing static block and main method is it possible to print some satements to the console ?

Ans: Yes, there are multiple ways,



```
class Test
{
    static int x = m1();
    public static int m1()
    {
        System.out.println("Hello I can print");
        System.exit(0);
        return 10;
    }
}
```

```
class Test
{
    static Test t = new Test();
    {
        Sopln("Hello I can print");
        System.exit(0);
    }
}
```

```
class Test
{
    static Test t = new Test();
    Test()
    {
        Sopln("Hello I can print");
        System.exit(0);
    }
}
```

Note:

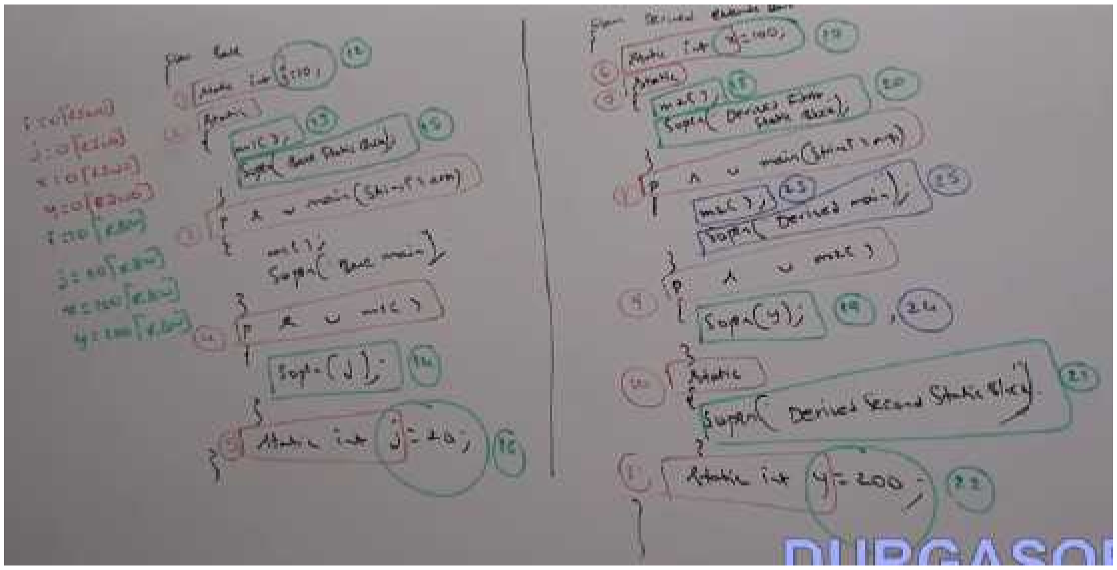
Form 1.7 version onwards main(), method is mandatory to start a program execution. Hence from 1.7 version onwards without writing main(), method it is impossible to print some statements to the console.

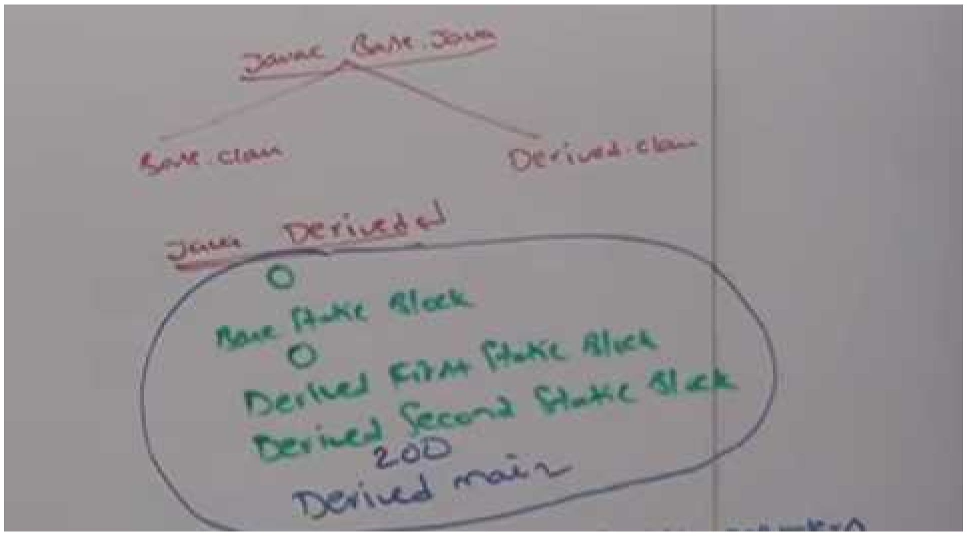
Static Control Flow in parent to child relationship:

1. Identification of static members from parent to child[1 to 11]

2. Execution of static variable assignments and static blocks from parent to child.[12 to 22]

3. Execution of only child class main method.[23 to 25]





Instance Control Flow:

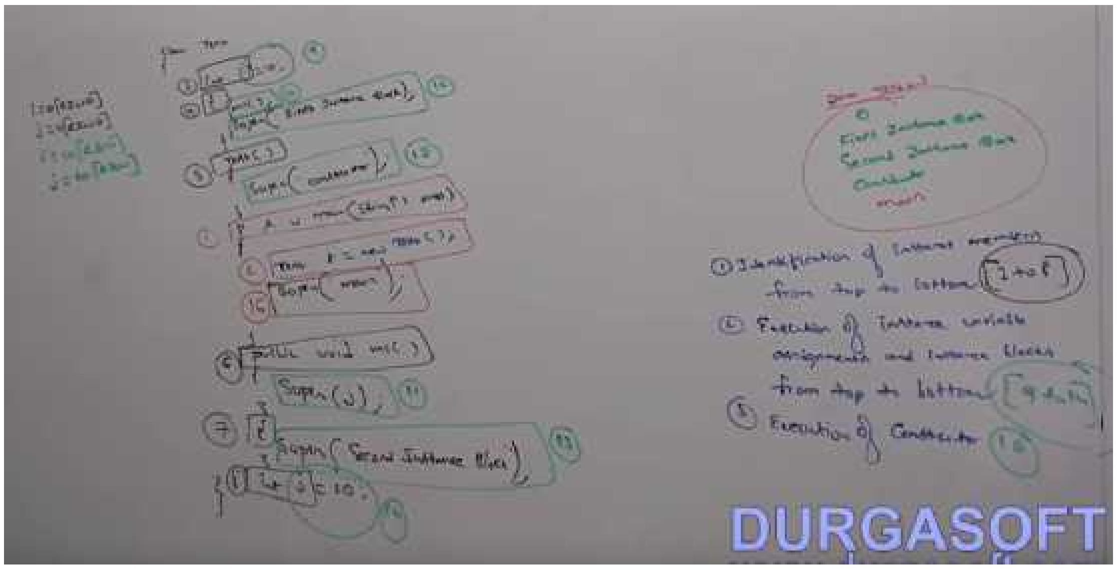
Whenever we are executing a java class 1st static control flow will be executed.

In the static control flow if we are creating an object the following sequence of events will be executed, as a part of instance control flow,

1. Indentification of instance members from top to bottom

2. Execution of instance variable assignments and instance blocks from top to bottom.

3. Execution of constructer.



Note:

Static control flow is onetime activity, which will be performed at the time class loading.

But instance control flow is not one time activity and it will be performed for every object creation.

Object creation is the most costly operation if there is no specific requiremt then it is not recommanded to create object.

Instace control Flow in parent to child relationship:

Whenever we are creating child class object the following sequence of events will be performed automatically as the part of instance control flow,

1. Identification of instance members from parent to child.[4 to 14]

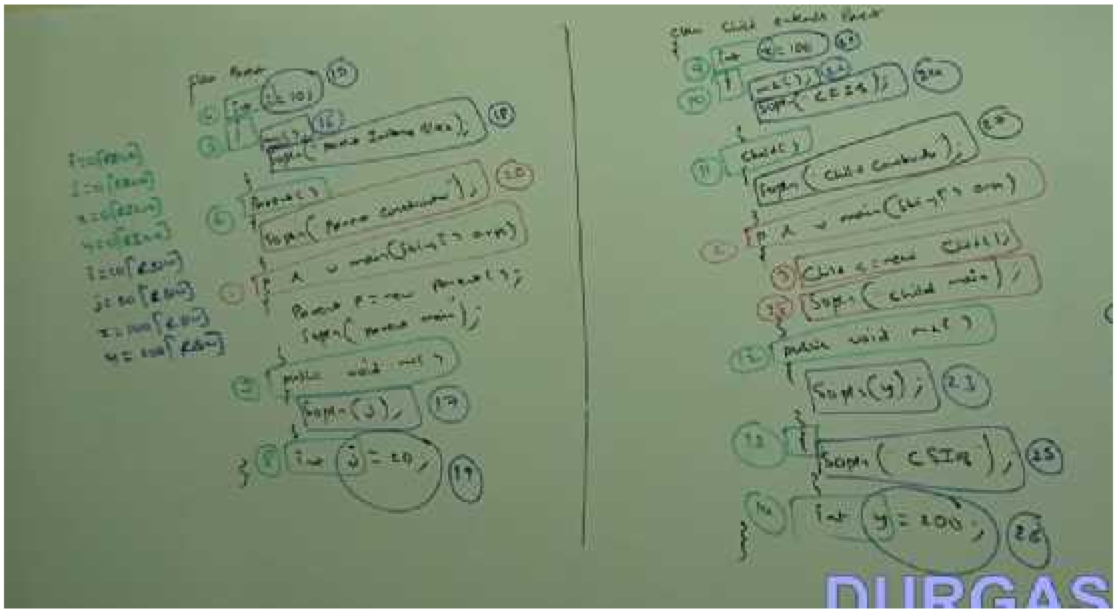
2. Execution of instance variable assignments and instance

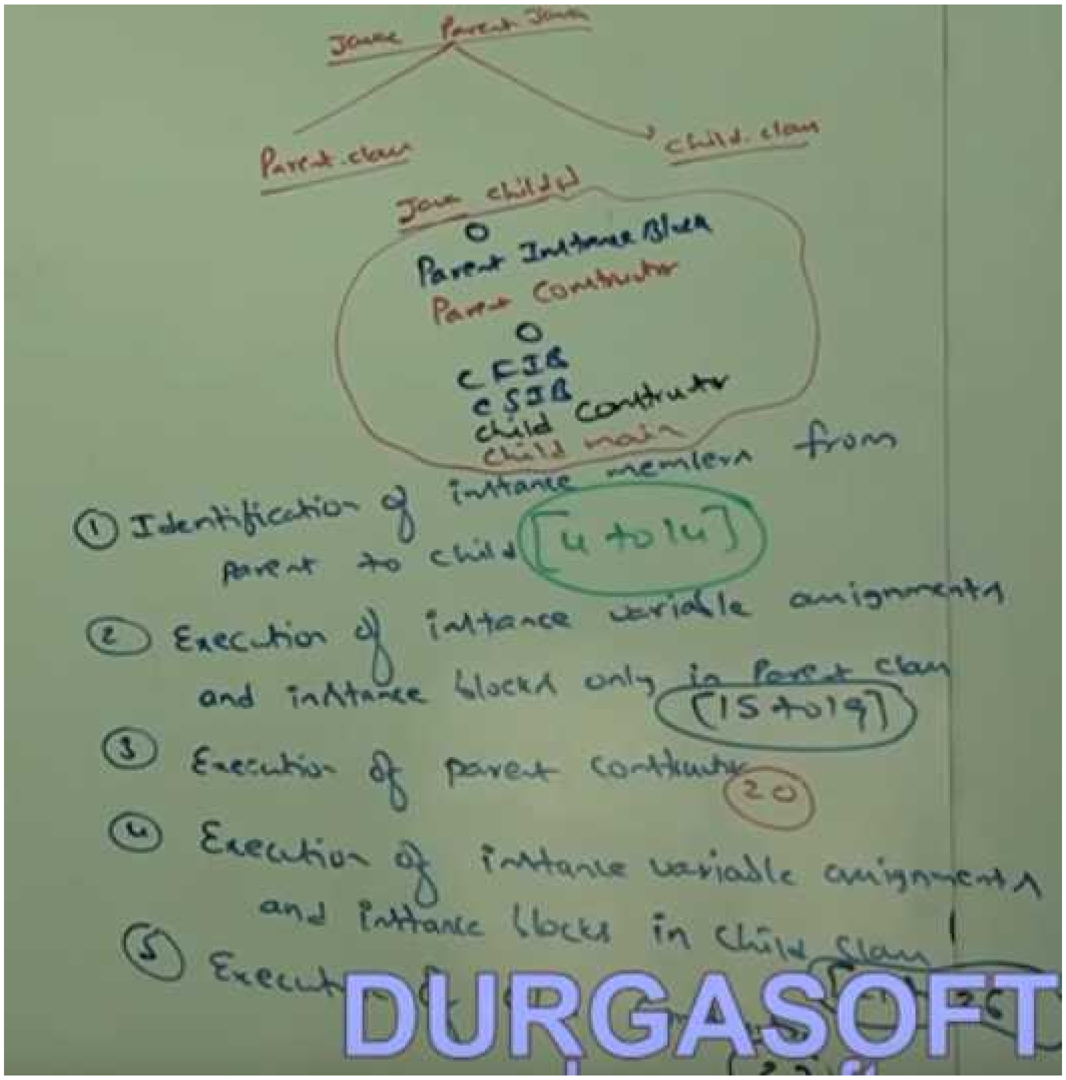
blocks only in parent class.[15 to 19]

3. Execution of parent constructor.[20]

4. Execution of instance variable assignments and instance blocks in child class.[21 to 26]

5. Execution of child constructor.[27]





Java Parent Java

Parent.class

Child.class

Java child

0
Parent Instance Block
Parent Constructor
0
CFIB
CSIB
Child Constructor
Child main

① Identification of instance members from parent to child [4 to 14]

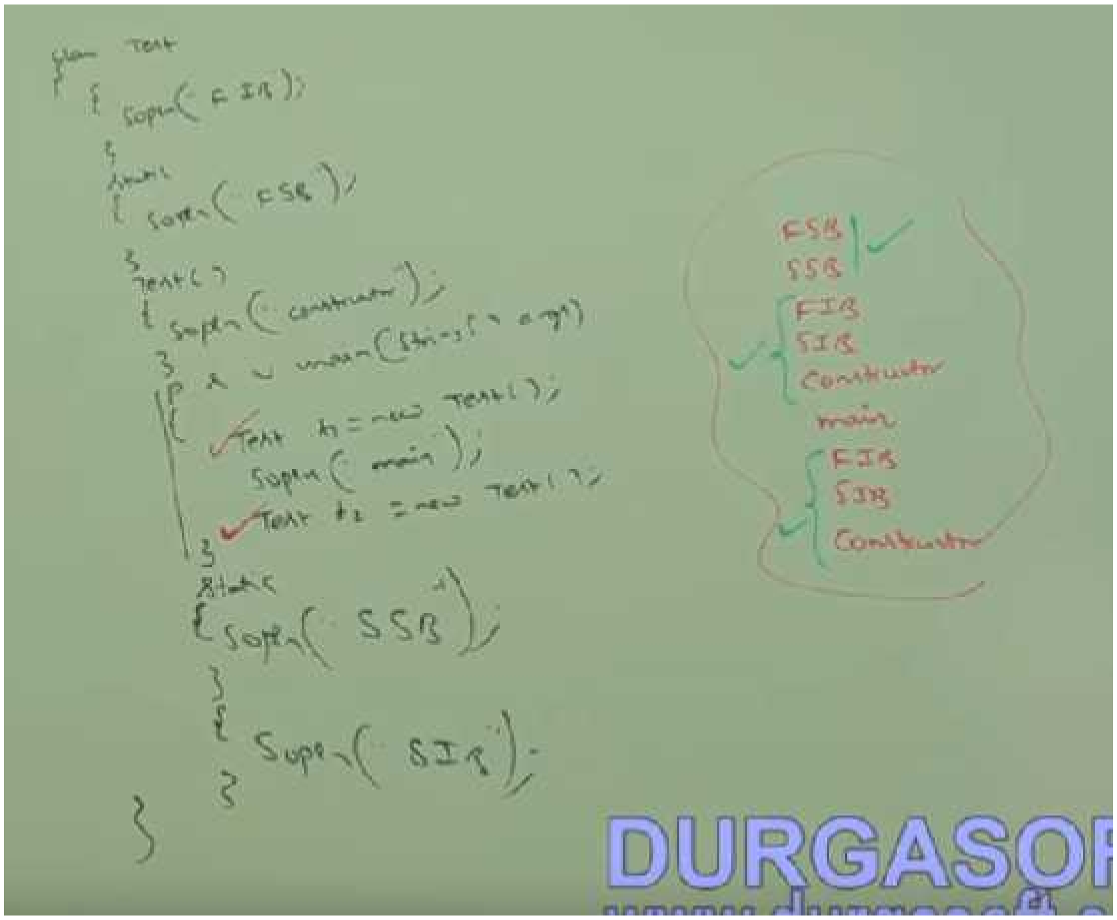
② Execution of instance variable assignments and instance blocks only in Parent class [15 to 19]

③ Execution of parent constructor (20)

④ Execution of instance variable assignments and instance blocks in child class

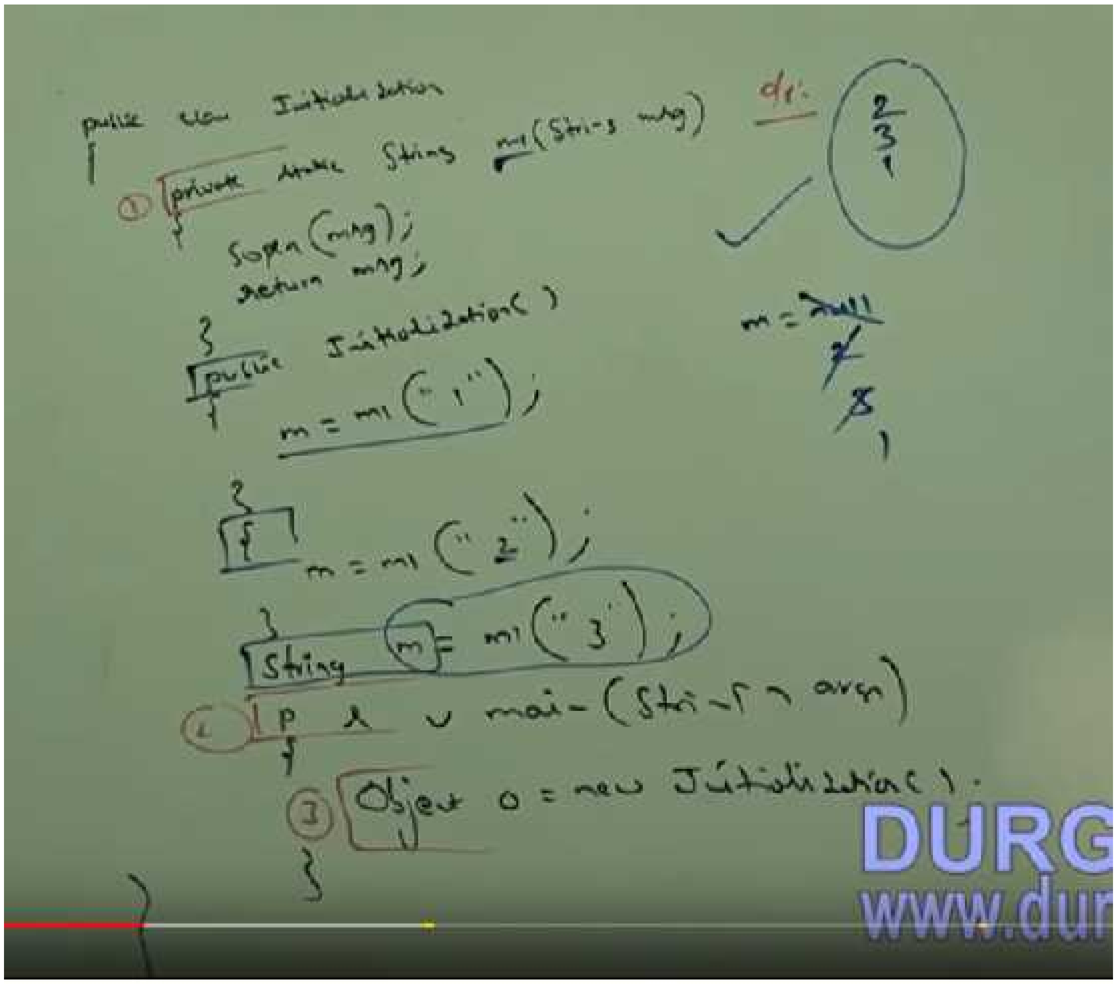
⑤ Execu

DURGASOFT



```
class Test
{
  {
    Sopln("FIB");
  }
  static
  {
    Sopln("FSB");
  }
  Test()
  {
    Sopln("constructor");
  }
  p s v main(String[] args)
  {
    ✓Test t1 = new Test();
     Sopln("main");
    ✓Test t2 = new Test();
  }
  static
  {
    Sopln("SSB");
  }
  {
    Sopln("SIB");
  }
}
```

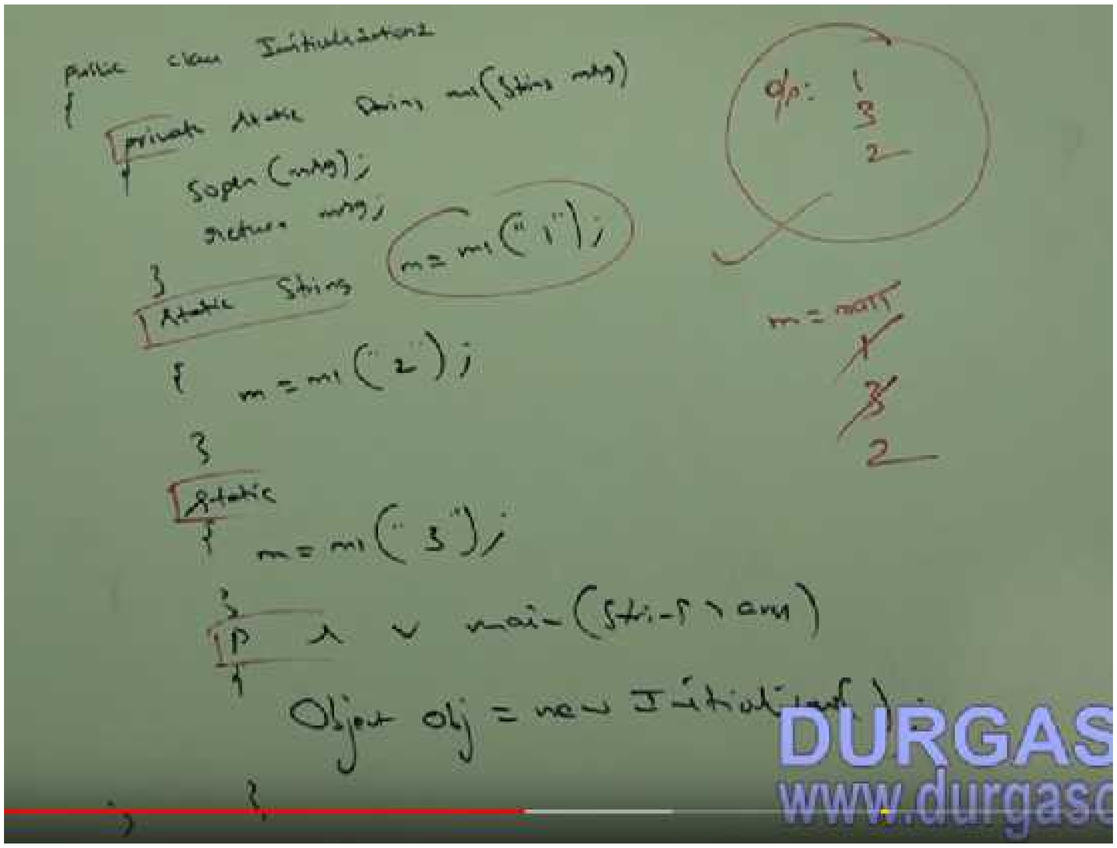
```
FSB  ✓
SSB
FIB
SIB  ✓
Constructor
main
FIB
SIB  ✓
Constructor
```



```
public class Initialization
{
 ① private static String m1(String msg)
   {
      Sopln(msg);
      return msg;
   }
   public Initialization()
   {
      m = m1("1");
   }
   {
      m = m1("2");
   }
   String m = m1("3");
 ② p s v main(String[] args)
   {
    ③ Object o = new Initialization();
   }
}
```

o/p:
2
3
1

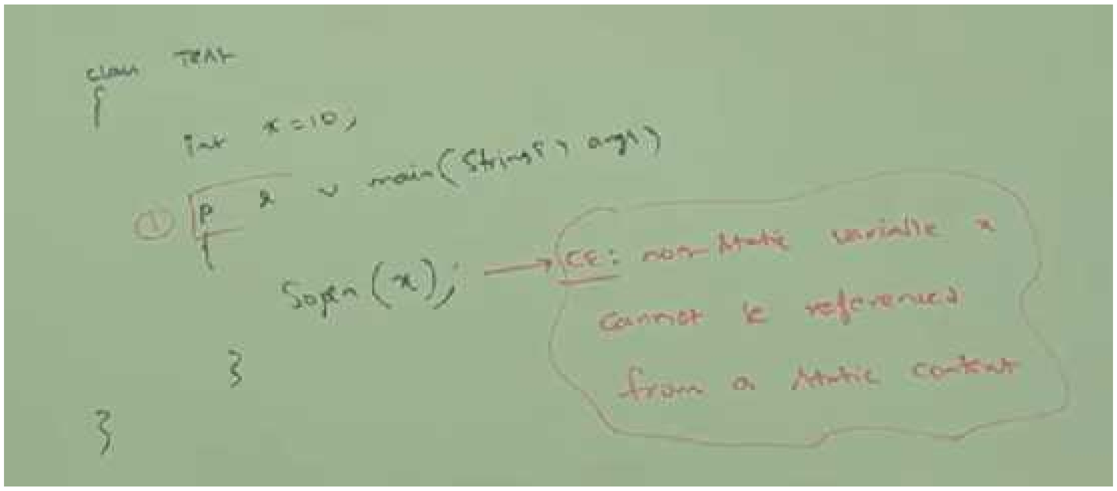
m = ~~null~~
~~2~~
~~3~~
1



```
public class Initialization2
{
  private static String m1(String msg)
  {
    Sopln(msg);
    return msg;
  }
  static String          m = m1("1");
  {
    m = m1("2");
  }
  static
  {
    m = m1("3");
  }
  p s v main(String[] args)
  {
    Object obj = new Initialization2();
  }
}
```

Note:

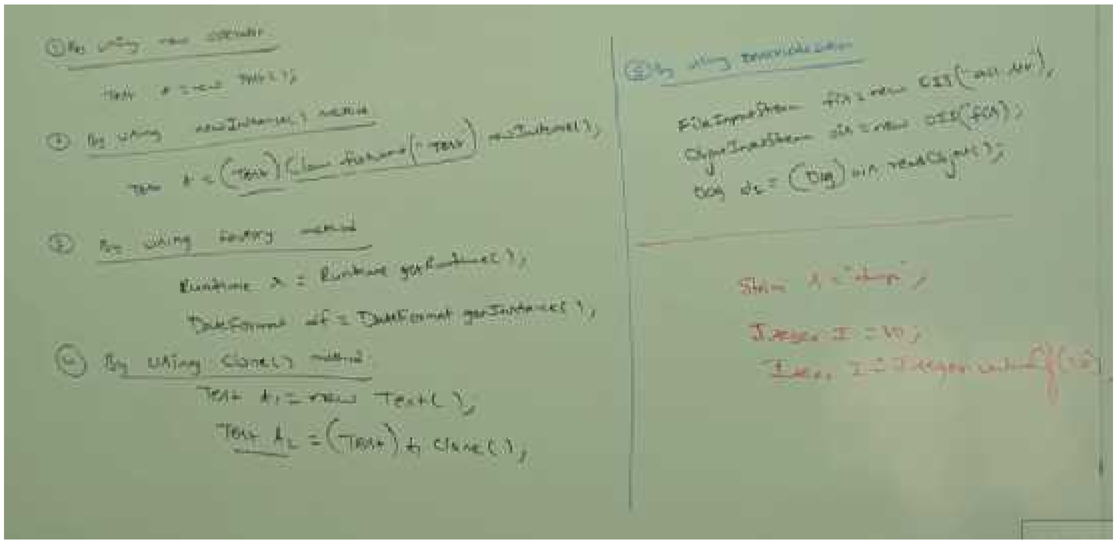
From static area we can't access instance members directly bcz while executing static area JVM may not identify instance members.



```
class Test
{
    int x=10;
① p s v main(String[] args)
    {
        Sopln(x); → CE: non-static variable x
                        cannot be referenced
                        from a static context
    }
}
```

Q. In how many ways we can create an object in java ?

or, In how any ways we can get object in java ?



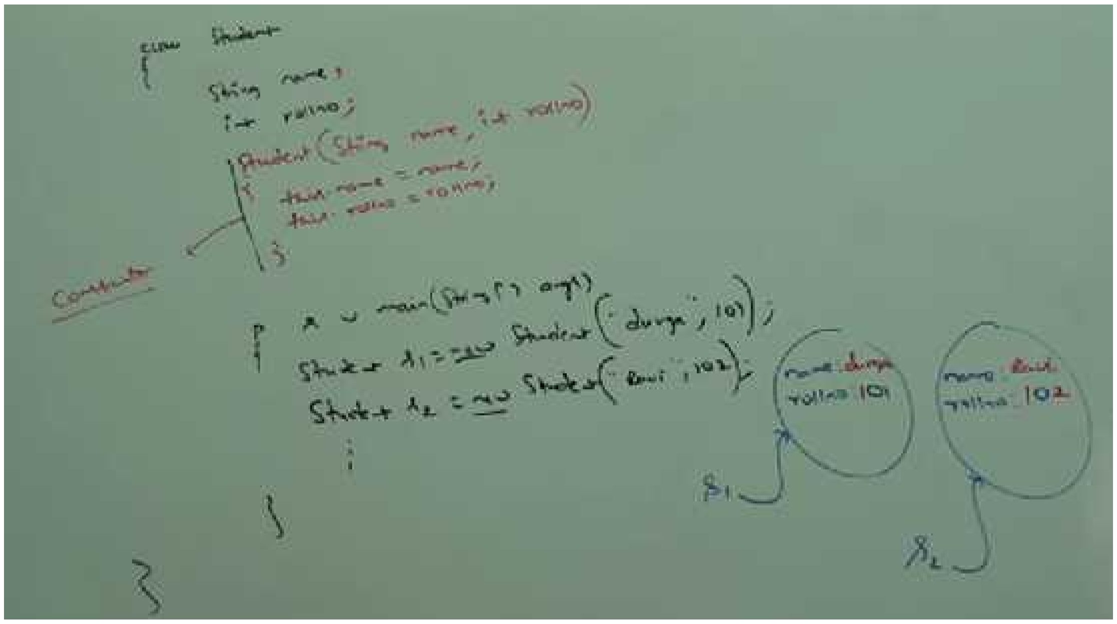
```
① By using new operator
     Test t = new Test();
② By using newInstance() method
     Test t = (Test) Class.forName("Test").newInstance();
③ By using factory method
     Runtime r = Runtime.getRuntime();
     DateFormat df = DateFormat.getInstance();
④ By using clone() method
     Test t1 = new Test();
     Test t2 = (Test) t1.clone();

⑤ By using deserialization
     FileInputStream fis = new FIS("abc.ser");
     ObjectInputStream ois = new OIS(fis);
     Dog d2 = (Dog) ois.readObject();

     String s = "durga";
     Integer I = 10;
     Integer I = Integer.valueOf(10);
```

Constructors:

Once we creat an object, compusory we should perform initialization, then only the object is in a position to respond properly.

Whenever we are creating an object some piece of the code will be executed automatically to perform initialization of the object. This piece of the code is nothing but constructor.

Hence the main purpose of the constructor is to perform initialization of an Object.



```
class Student
{
    String name;
    int rollno;
    Student(String name, int rollno)
    {
        this.name = name;
        this.rollno = rollno;
    }
    p s v main(String[] args)
    {
        Student s1 = new Student("durga", 101);
        Student s2 = new Student("Ravi", 102);
        ;
    }
}
```

Note:

The main purpose of the constructor is to perform initialization of an object but not to create object.

Difference between Constructor and Instance block:

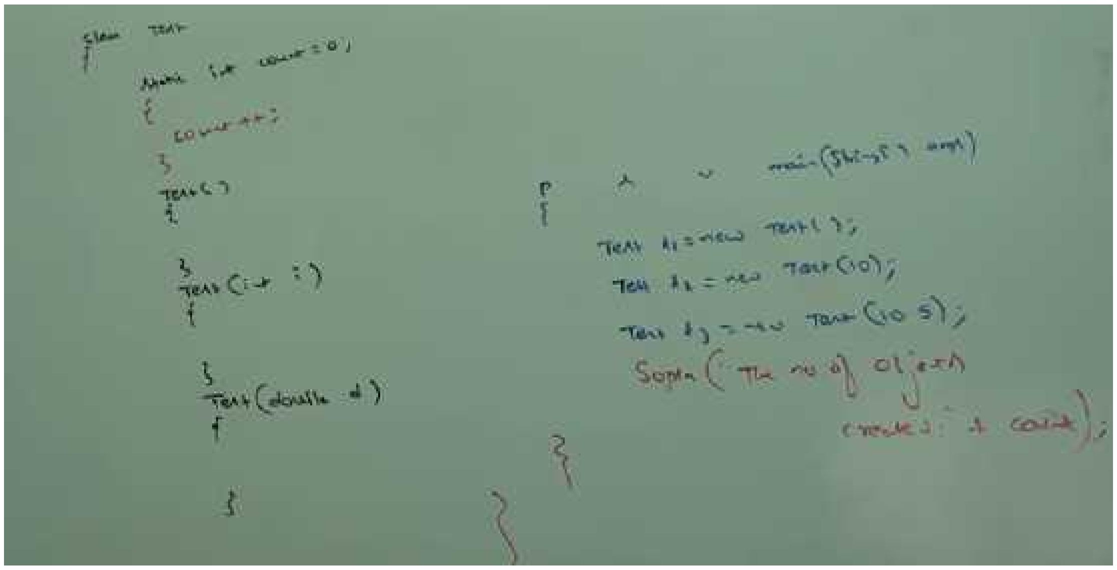
The main purpose of constructor is to perform initialization of an object.

But other than initialization if we want to perform at any activity for every object creation then we should go for instance block (like updatting one entry in the DB, for every object creation or incrementing count value for every object creation etc).

Both constructer and instance block have their own different purposes and replaceing one concept with another concept may not work always.

Both constructor and instance block will be executed for every object creation but instance block first followed by constructor.

Demo program to print number of objects created for a class:



```
class Test
{
    static int count=0;
    {
        count++;
    }
    Test()
    {
    }
    Test(int i)
    {
    }
    Test(double d)
    {
    }
    p s v main(String[] args)
    {
        Test t1=new Test();
        Test t2=new Test(10);
        Test t3=new Test(10.5);
        Sopln("The no of objects created:"+count);
    }
}
```

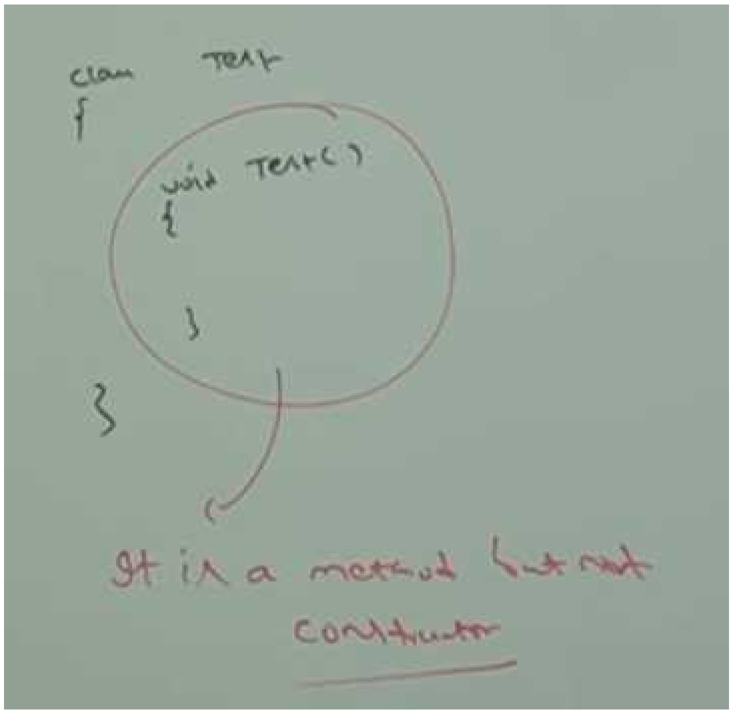
Rules of writing constructors:

1. Name of the class and name of the constructor must be matched.

2. Return type concept not applicable for constructor even void also.

3. By mistake if we are trying to declare return type for the

constructor then we won't get any compiletime error because compiler treats it as a method.



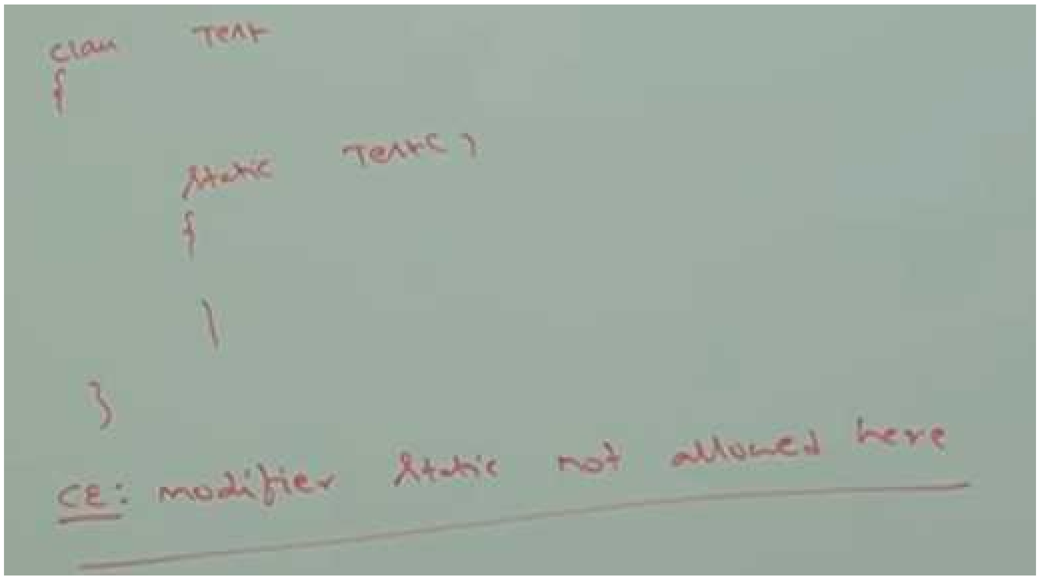
```
class Test
{
    void Test()
    {

    }
}
```

It is a method but not constructor

Hence it is legal (but stupid) to have a method whose name is exactly same as class name.

The only applicable modifier for constructors are pubic, private, protected and default.

If we are trying to use any other modifier we will get compiletime error.



```
class Test
{
    static Test()
    {
    }
}
```

CE: modifier static not allowed here

Default constructor:

Compiler is responsible to generate default constructer(but not JVM).

If we are not writing any constructer then only compiler will generate default constructer i.e. if we are writing at least one constructer then compiler won't generate default constructer.

Hence every class in java can contain constructer, it may be default constructer generated by compiler or customized constructer expilcitly provided by programmer but both simultaneously.

Prototype of default constructer:

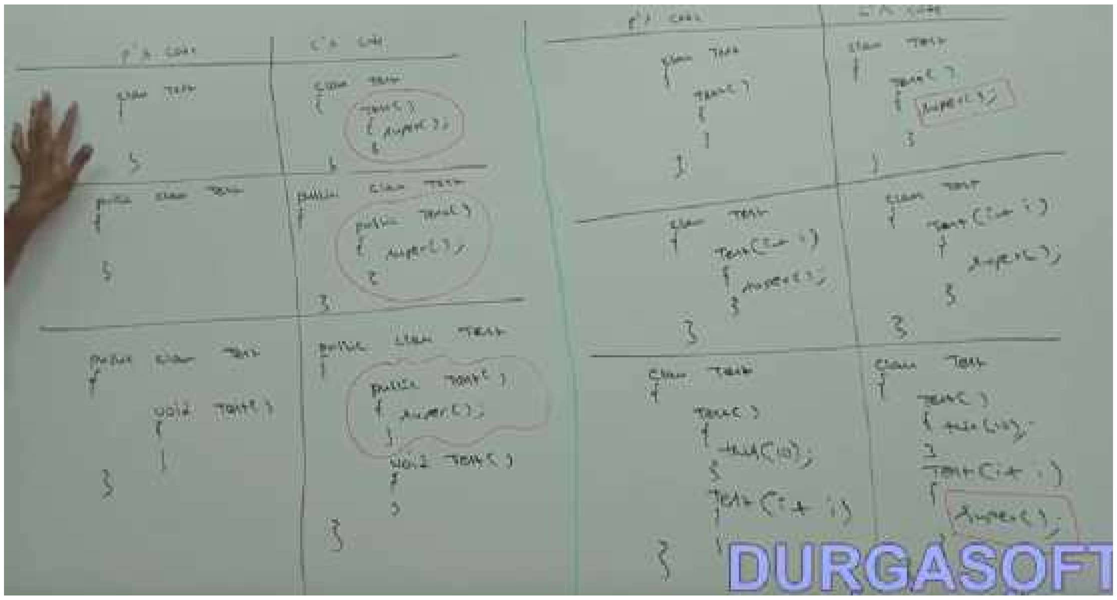
1. It is always no-arg constructer.

2. The access modifier of default constructer is exactly same as access modifier of class.(this rule is applicable

only for public and default)

3. It contains only one line super();

It is a no-arg call to super class constructer.



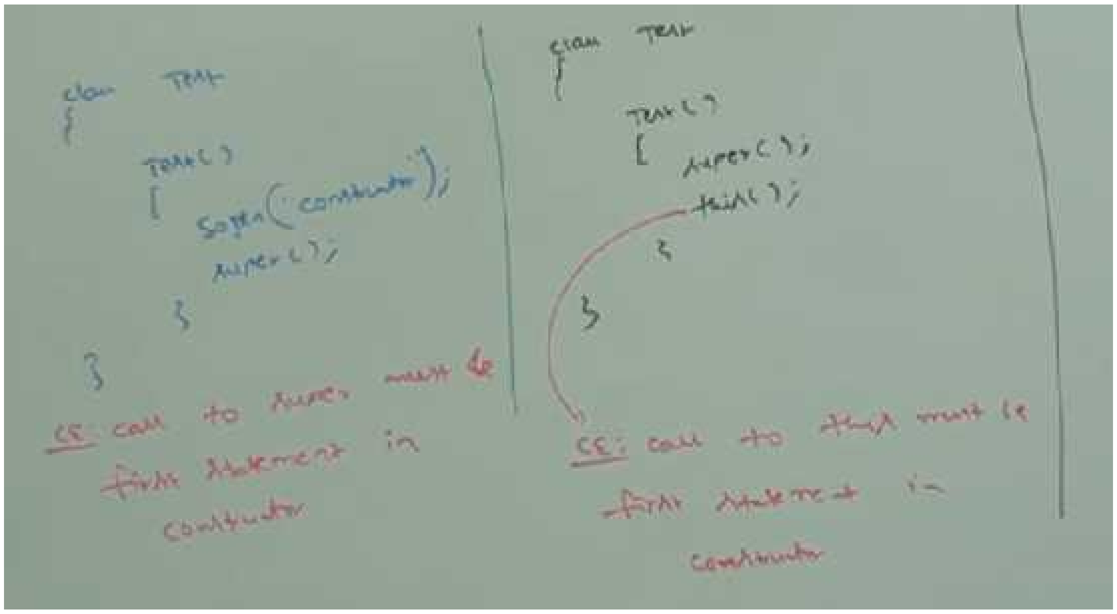
the first line inside every constructer should be either supre() or this() and we are not writing anything compiler will always place super().

Case 1:

We can take super() or this(), only in 1st line of constructer. If we are trying to take anywhere else, we will get compiletime error.

Case 2:

Within the constructer we can take either super() or this(), but not both simultaneously.



```
class Test
{
    Test()
    {
        Sopln("constructor");
        super();
    }
}
```

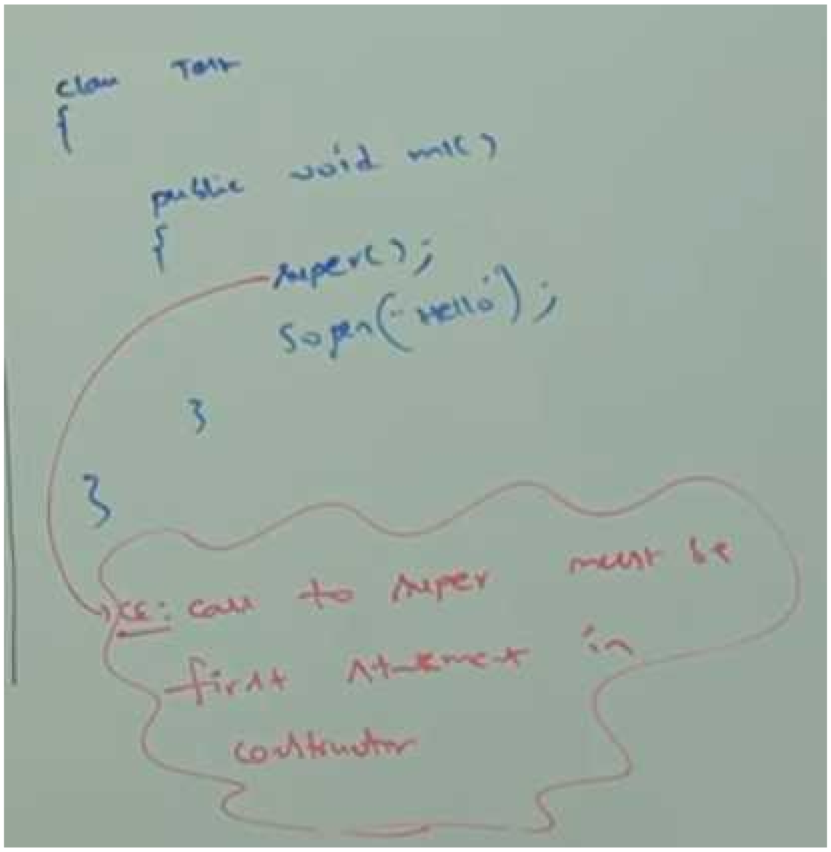
CE: call to super must be first statement in constructor

```
class Test
{
    Test()
    {   super();
        this();
    }
}
```

CE: call to this must be first statement in constructor

Case 3:

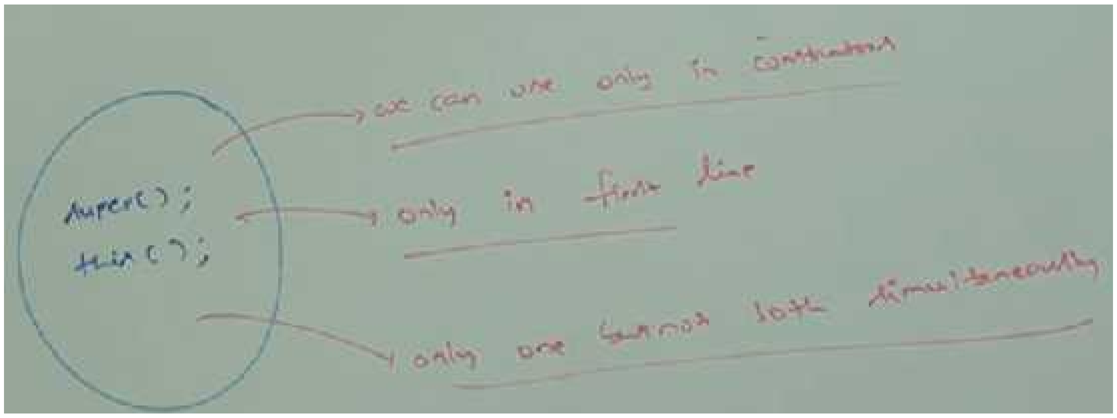
We can user super() or this(), only inside constructer. If we are trying to use outside of constructer we will get compiletime error.



```
class Test
{
    public void m1()
    {
        super();
        Sopln("Hello");
    }
}
```

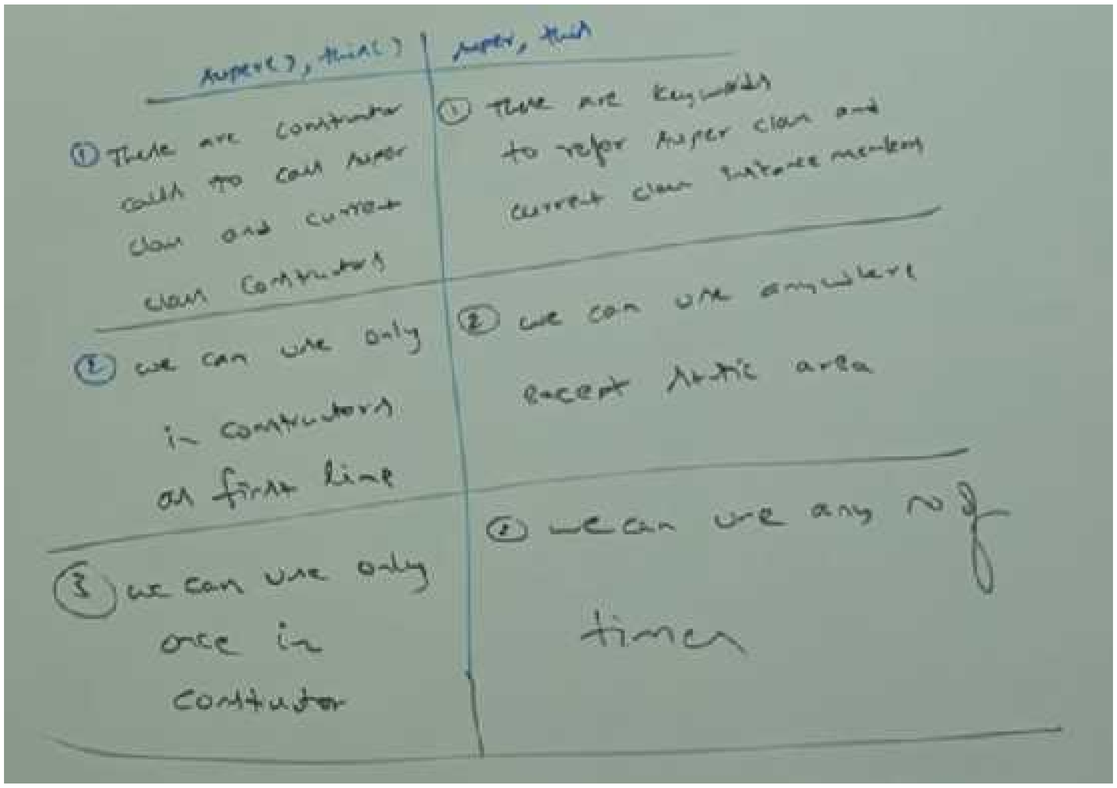
CE: call to super must be first statement in constructor

i.e. we can call a constructer directly from another constructer only.



```
super();
this();
```

we can use only in constructors
only in first line
only one but not both simultaneously

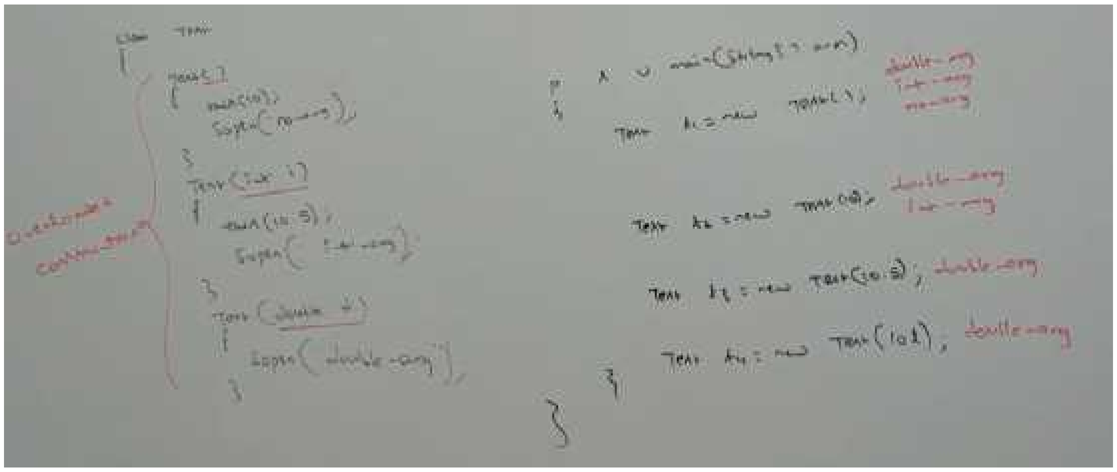


super(), this()	super, this
① These are constructor calls to call super class and current class constructors	① These are keywords to refer super class and current class instance members
② we can use only in constructors as first line	② we can use anywhere except static area
③ we can use only once in constructor	③ we can use any no of times

Overloaded Constructers:

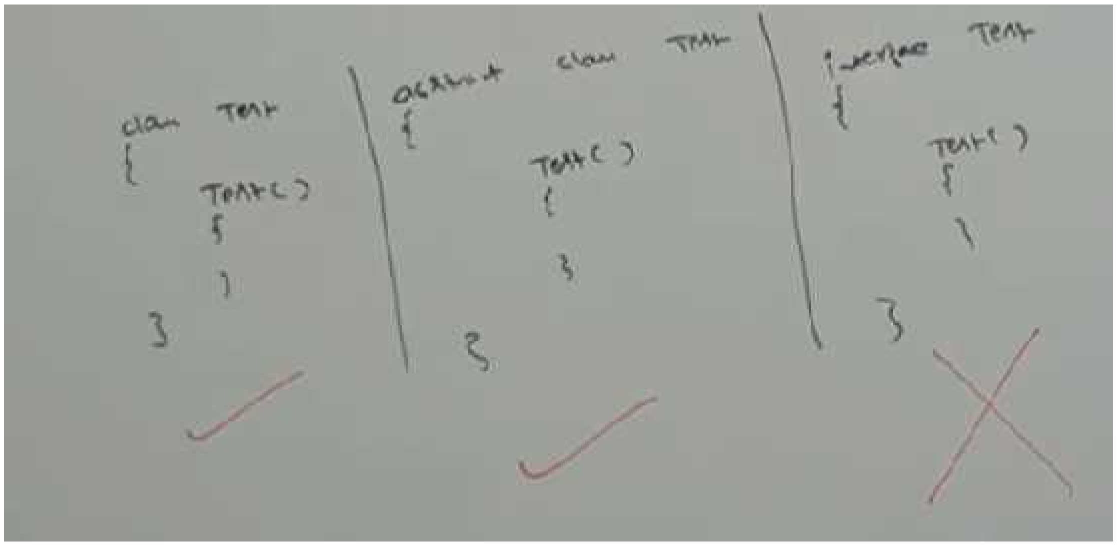
Within class we can declare multiple constructers and all these constructers having same name but diferent type of arguments, hence all these constructers are considered as overloaded constructers.

Hence ovrloading concept applicable for constructers.



For constructors inheritance and overriding concepts are not applicable but overloading concept is applicable.

Every class in java including abstarct class can contain constructer but interface can't contain constructer.

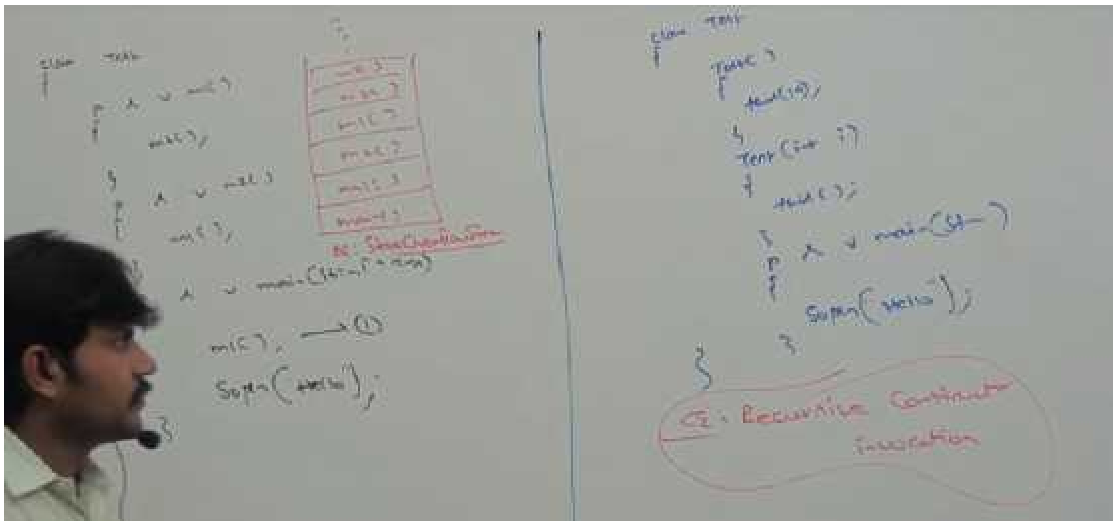


```
class Test        abstract class Test        interface Test
{                 {                          {
    Test()            Test()                     Test()
    {                 {                          {
    }                 }                          }
}                 }                          }
```

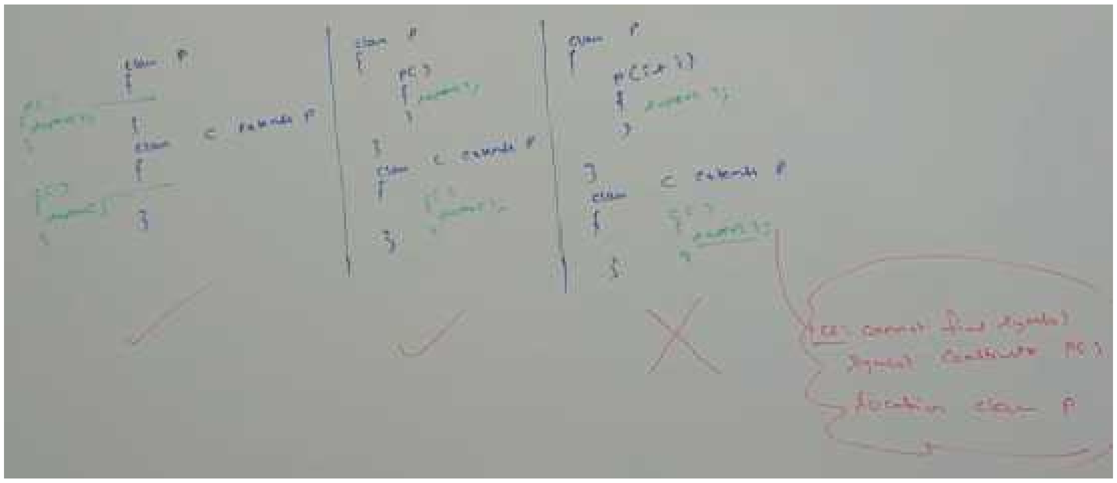
Case 1:

Recursive method call is a runtime exception saying "Stack overflow error"

But In our program if there is a chance of recursive constructor invocation then the code won't compile and we will get compile time error.



Case 2:

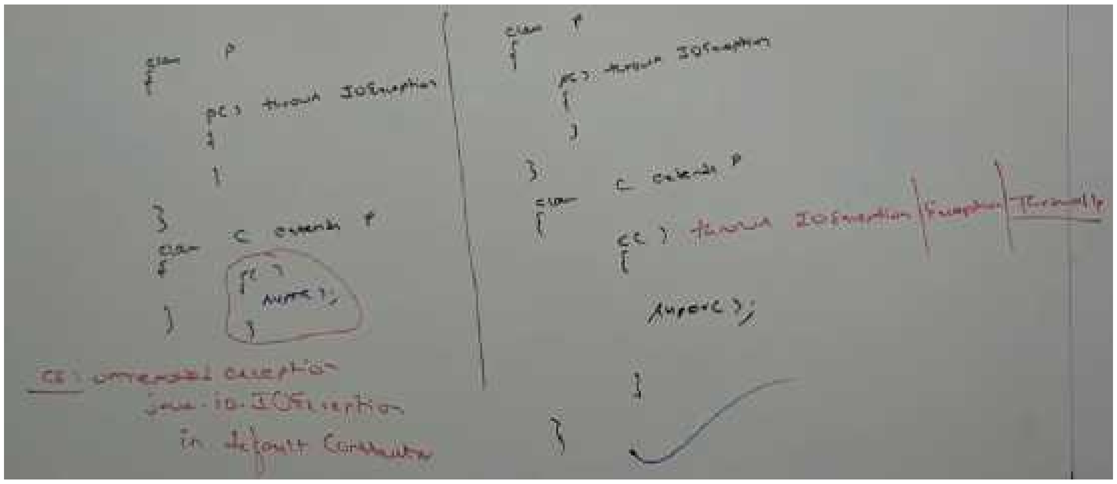


Note:

If parent class contains any argument constructor then while writing child classes we have to take special care w.r.t constructors.

**Whenever we are writing any argument constructor it is highely recomended to write no-arg constructor also.

Case 3:



```
class P
{
    P() throws IOException
    {
    }
}
class C extends P
{
    C()
    {
        super();
    }
}
CE: unreported exception
    java.io.IOException
      in default constructor
```

```
class P
{
    P() throws IOException
    {
    }
}
class C extends P
{
    C() throws IOException|Exception|Throwable
    {
        super();
    }
}
```

Note:

If parent class constructor throws any checked exception compulsory child class constructor should through the same checked exception or it's parent otherwise the code won't compile.

Q. Which of the following is valid ?

1. The main purpose of constructor is to create an object.(invalid)

2. The main purpose of constructor is to perform initialization of an object(valid).

3. The name of the constructor need not be same as class name(Invalid)

4. Return type concept applicable for constructors but only

void(Invalid)

5. We can apply any modifier for constructor.(Invalid)

6. default constructor generated by JVM(false).

7. Compiler is responsible to generate default constructor(True).

8. Compiler will always generates default constructor.(False)

9. If we are not writing no-arg constructor then compiler will generate default constructor(False)

10. Default constructor is always no-arg constructor(True).

11. The first line inside every constructor shouls be either super() or this(). If we are not writing anything then compiler will generate super(). (True)

12. For constructor only overloading conept is applicable.(True)

13. Inheritance concept is not applicable(True)

Singleton Classes:

For any java class if we are allowed to create only one object such type of class is called singleton class.

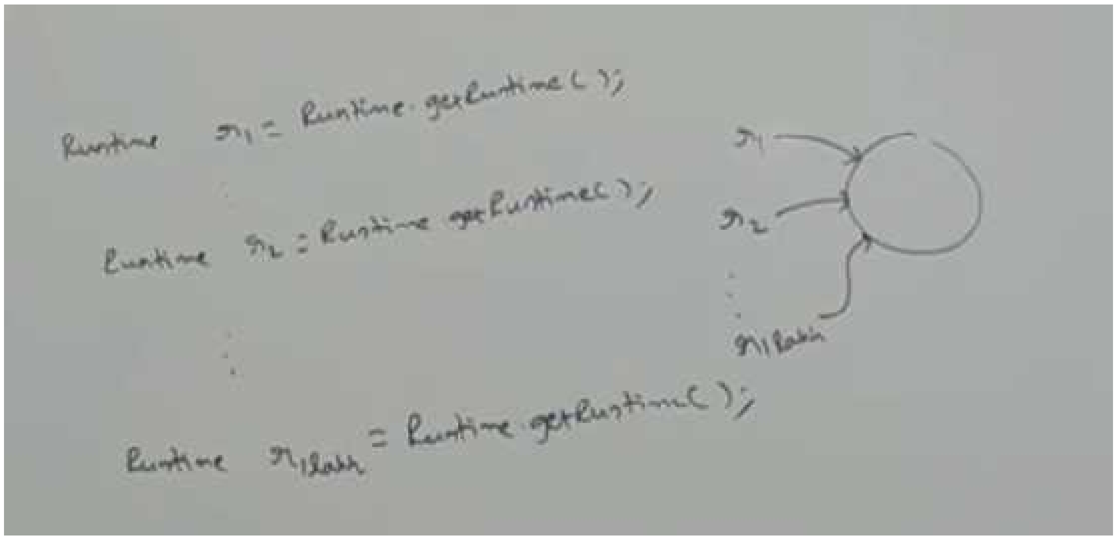
e.g. Runtime, BusunessDeligate, ServiceLocator etc.

Advantadge:

If several people have same requirement then it is not recomanded to create a separate object for every requirement.

We have to create only one object and we can reuse same object for every similar requirement so that performance and memory utilizations will be improved.

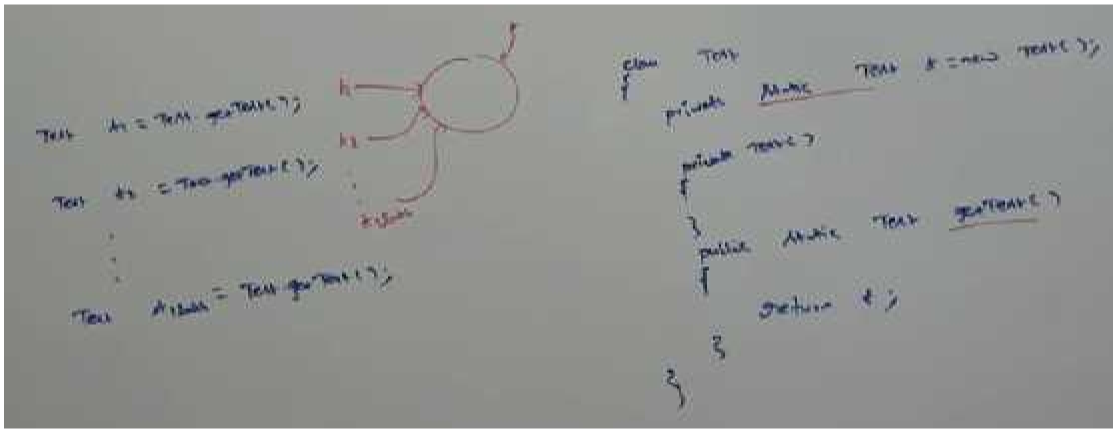
This is the cental idea of singleton classes.


Runtime r1 = Runtime.getRuntime();
Runtime r2 = Runtime.getRuntime();
Runtime r1lakh = Runtime.getRuntime();
r1
r2
r1lakh

How to create our own singleton classes:

We can create our own singleton classes for this we have to use private constructor and private static variable and public factory method.

Approach 1:



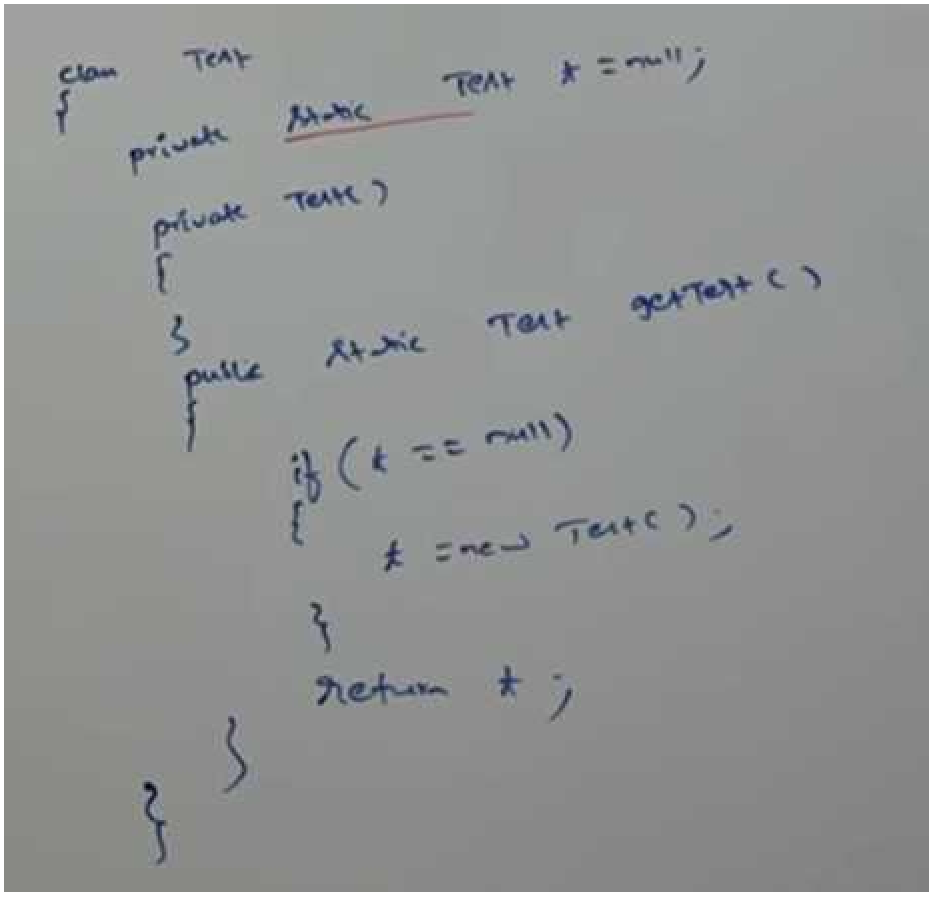
```
Test t1 = Test.getTest();
Test t2 = Test.getTest();
.
.
.
Test t1lakh = Test.getTest();
```

```
class Test
{
    private static Test t = new Test();
    private Test()
    {
    }
    public static Test getTest()
    {
        return t;
    }
}
```

Note:

Runtime class is internally implemented by using this appraoach.

Approach 2:



```
class Test
{
    private static Test t =null;
    private Test()
    {
    }
    public static Test getTest()
    {
        if(t == null)
        {
            t =new Test();
        }
        return t;
    }
}
```

At any point of time for Test class we can create only one object, hence test class is singleton class.

Q. Class is not final but we are not allowed to create child classes how it is possible ?

Ans:

By declaring every constructer as private we can restrict child class creation.