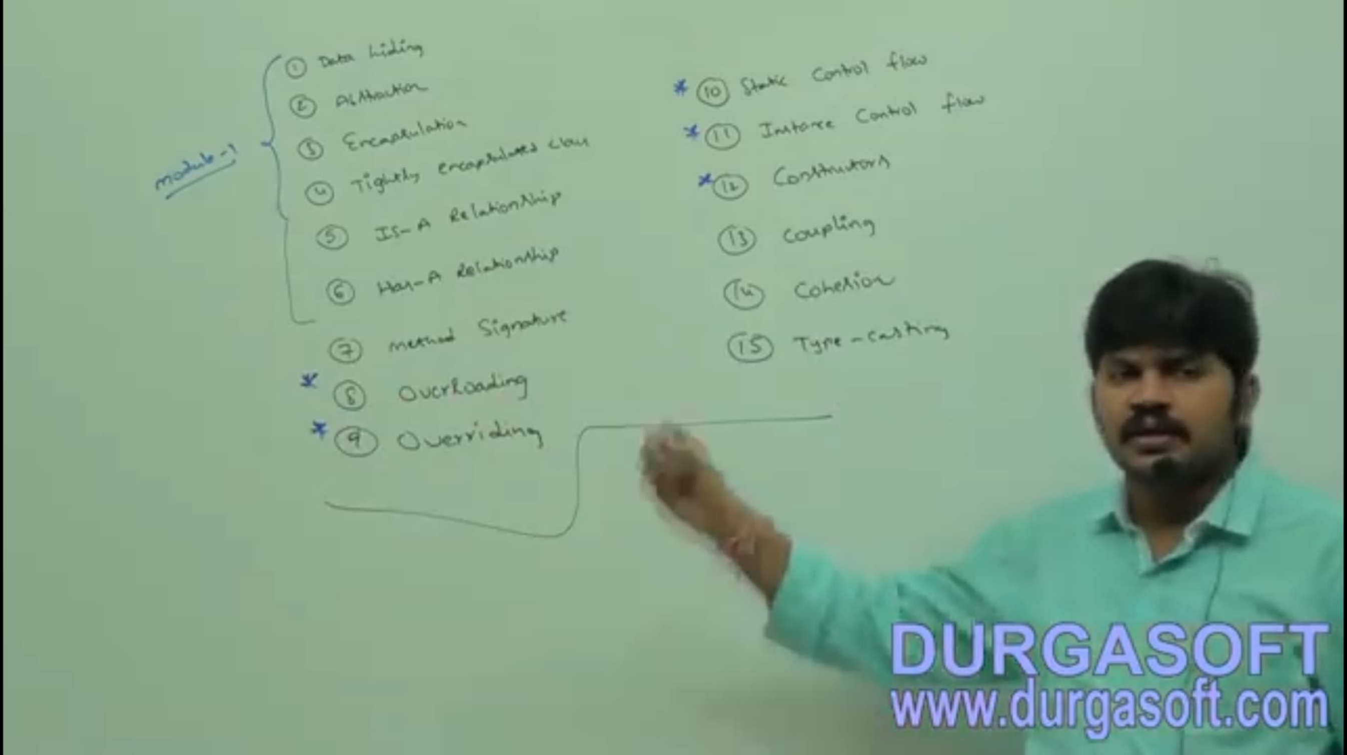
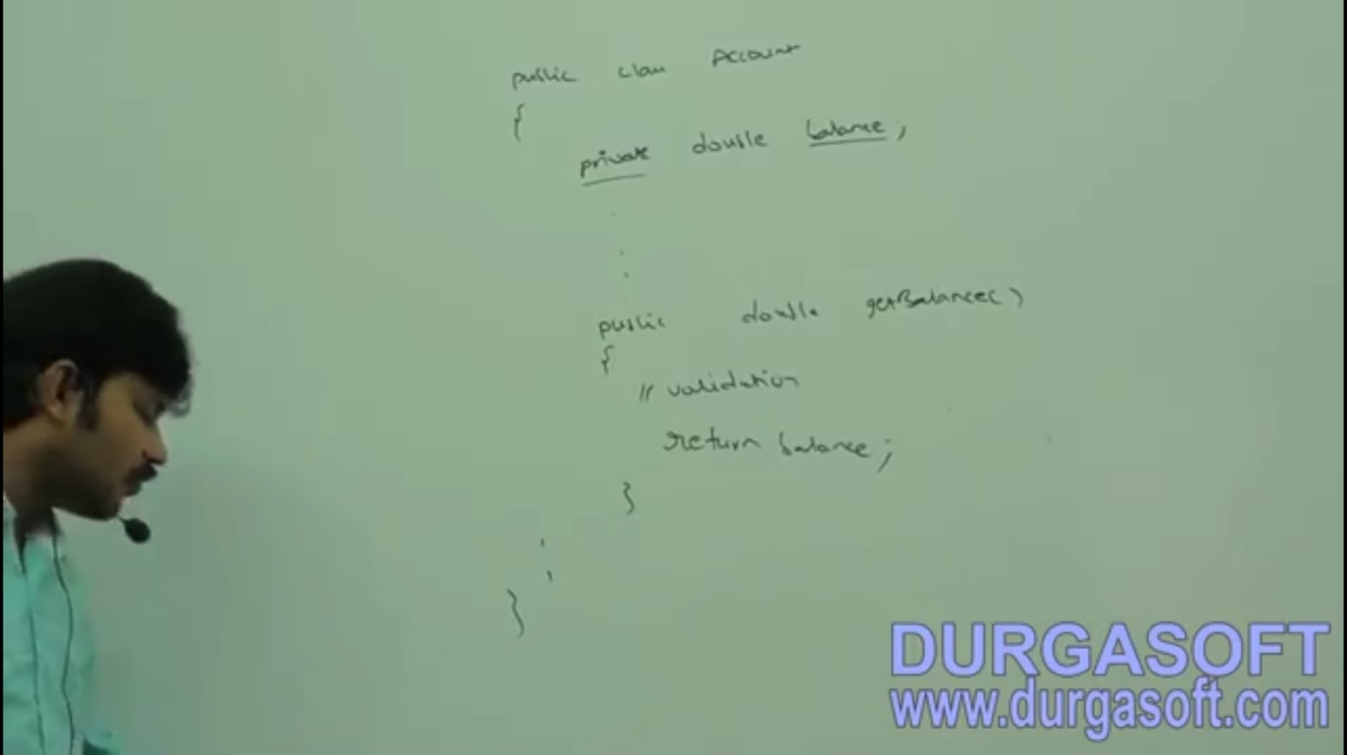
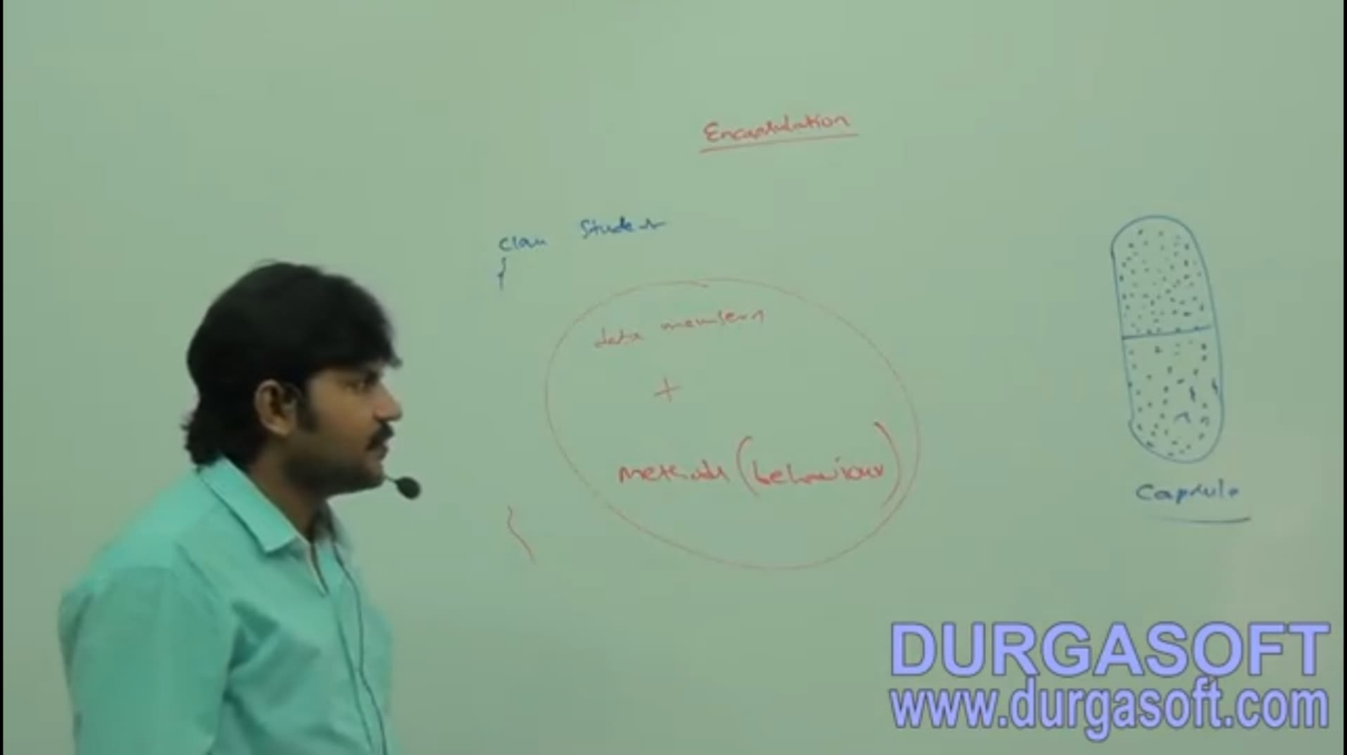


module-1

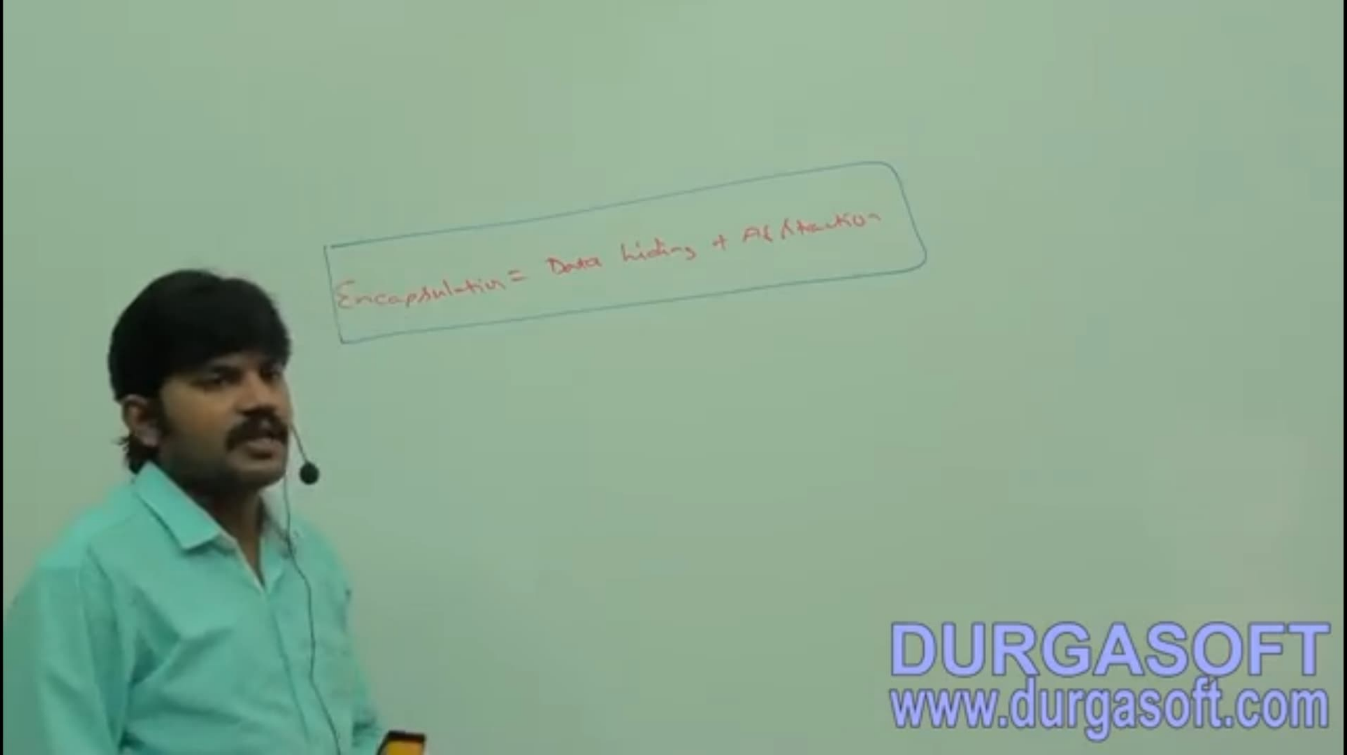
1. Data hiding
2. Abstraction
3. Encapsulation
4. Tightly encapsulated class
5. IS-A Relationship
6. Has-A Relationship
7. method Signature
8. * OverLoading
9. * Overriding
10. * Static Control flow
11. * Instance Control flow
12. * Constructors
13. Coupling
14. Cohesion
15. Type-casting



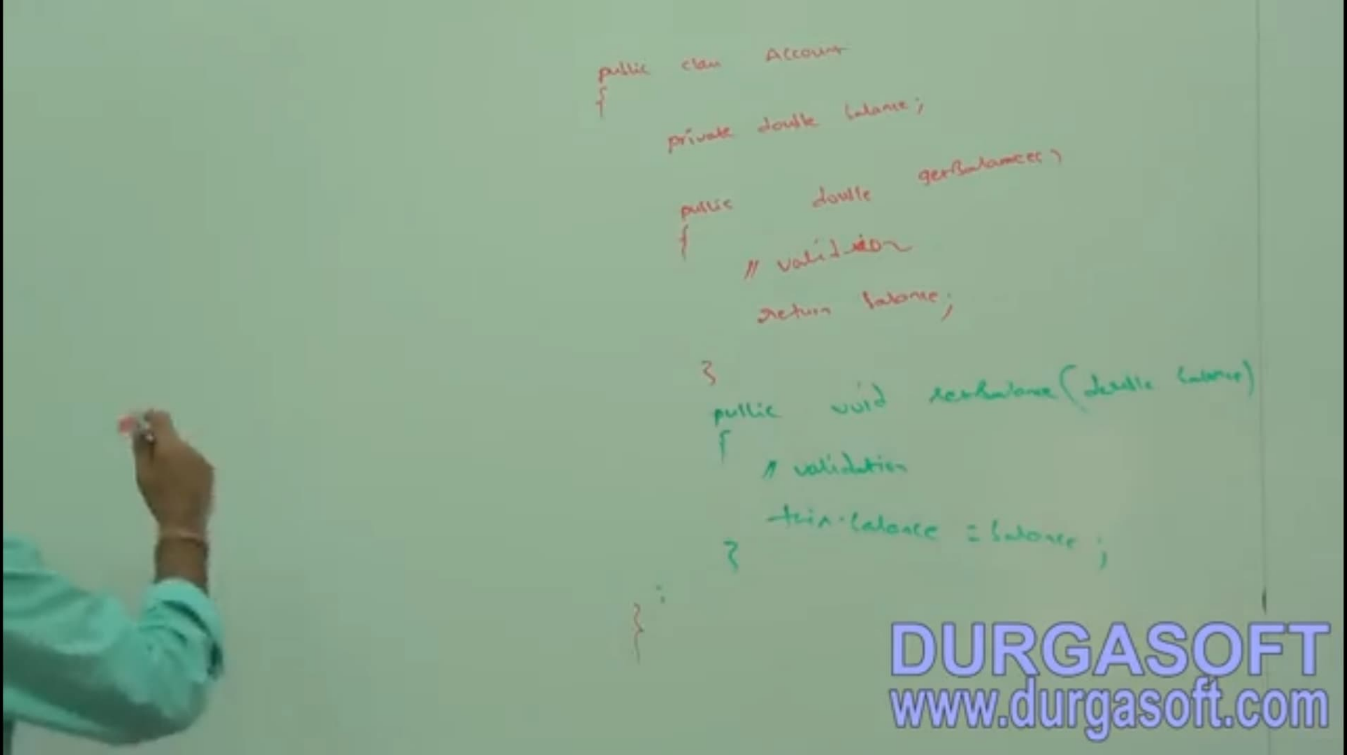
```
public class Account
{
    private double balance;
      .
      .
      .
      public double getBalance()
      {
        // validation
          return balance;
        }
   .
   .
}
```



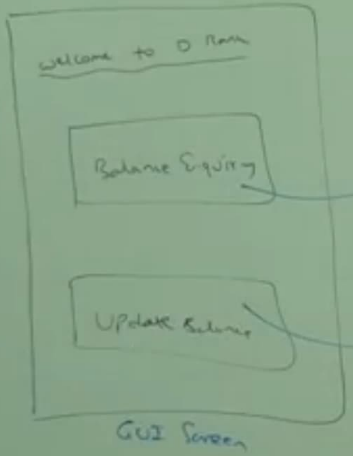
Encapsulation
class Student
{
data members
+
methods (behaviour)
Capsule



Encapsulation = Data hiding + Abstraction



```
public class Account
{
    private double balance;

    public double getBalance()
    {
        // validation
        return balance;

    }
    public void setBalance(double balance)
    {
        // validation
        this.balance = balance;
    }
  :
}
```



```
public class Account
{
    private double balance;

    public double getBalance()
    {
        // validation
        return balance;
    }
    public void setBalance(double balance)
    {
        // validation
        this.balance = balance;
    }
}
```

✓

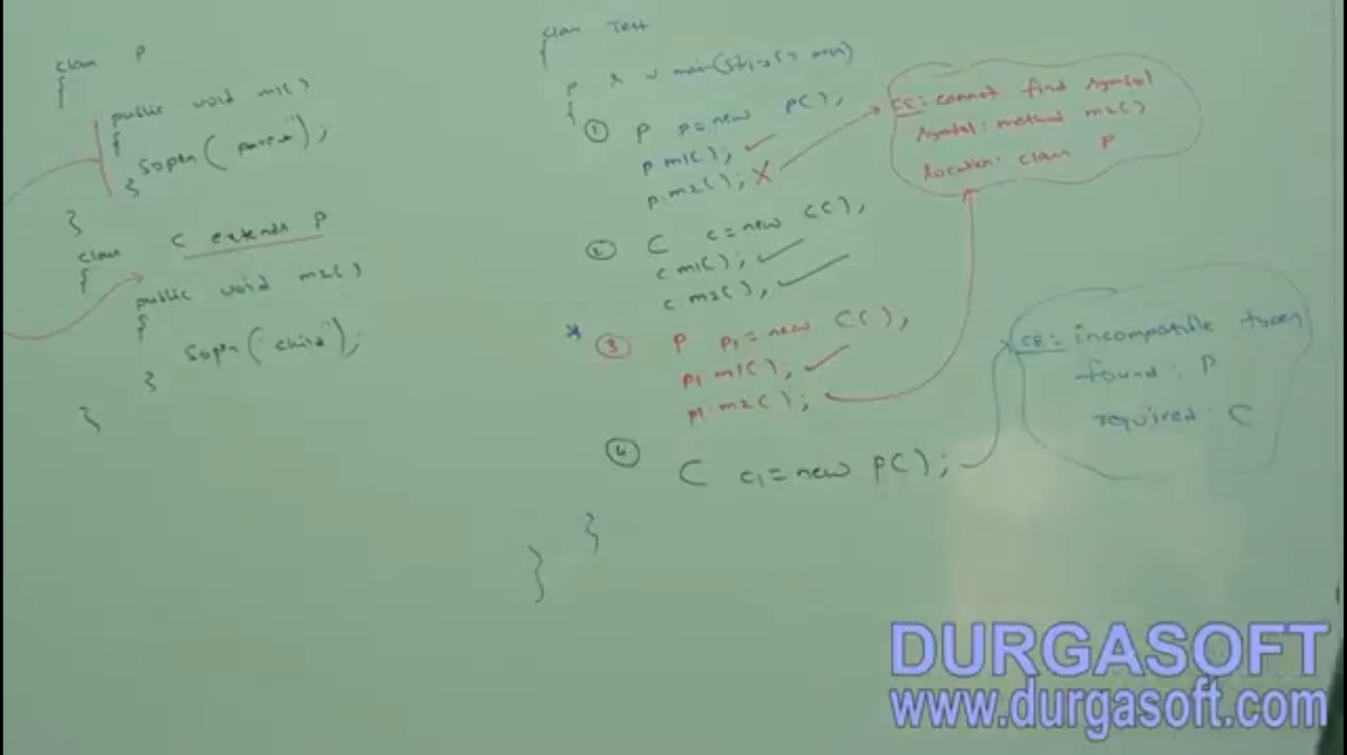
```
class A
{
    private int x=10;
}
```

✗

```
class B extends A
{
    int y=20;

}
```

✓

```
class C extends A
{
    private int z=30;

}
```

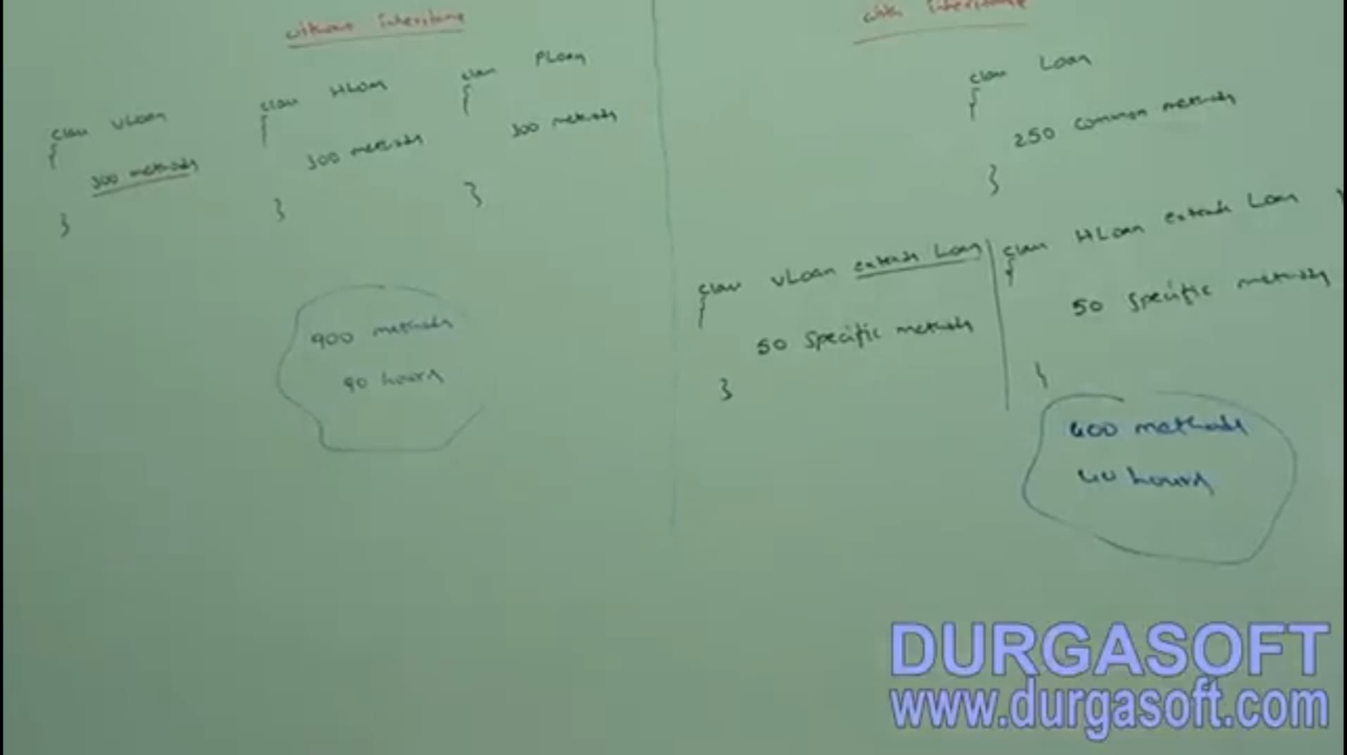


```
class P
{
    public void m1()
    {
        Sopln("parent");
    }
}
class C extends P
{
    public void m2()
    {
        Sopln("child");
    }
}
```

```
class Test
{
  p s v main(String[] args)
  {
    ① P p=new P();
       p.m1(); ✓
       p.m2(); ✗

    ② C c=new C();
       c.m1(); ✓
       c.m2(); ✓

  * ③ P p1=new C();
       p1.m1(); ✓
       p1.m2();

    ④ C c1=new P();
  }
}
```

CE: cannot find symbol
symbol: method m2()
location: class P

CE: incompatible types
found: P
required: C



without inheritance

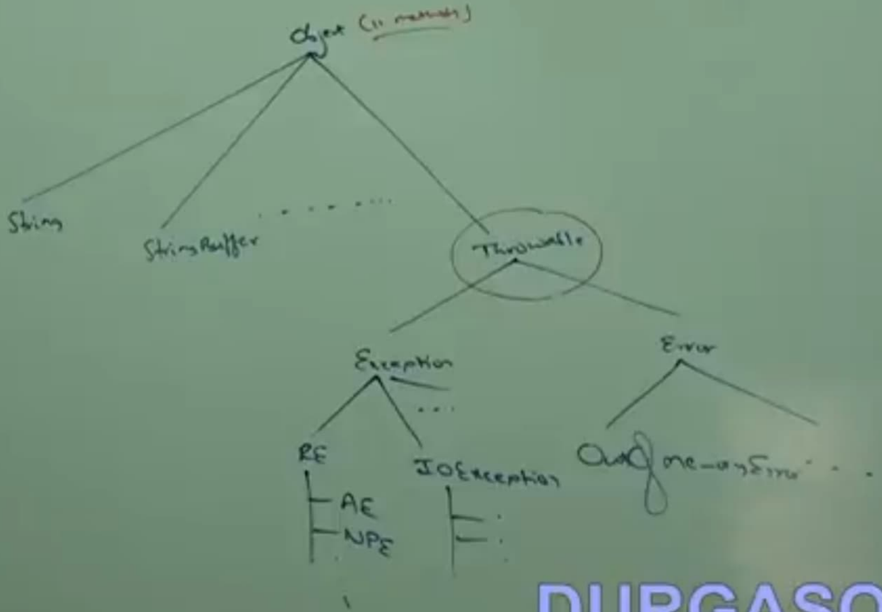
```
class VLoan
{
    300 methods
}

class HLoan
{
    300 methods
}

class PLoan
{
    300 methods
}
```

900 methods
90 hours

with inheritance

```
class Loan
{
    250 common methods
}

class VLoan extends Loan
{
    50 Specific methods
}

class HLoan extends Loan
{
    50 Specific methods
}
```

600 methods
60 hours


Object (11 methods)
String
StringBuffer
Throwable
Exception
Error
RE
AE
NPE
IOException
OutOfMemoryError

multiple inheritance

```
class A extends B, C
{

}
```

$\times$

$\boxed{CE}$

```
class A extends B
{

}
```

$$\begin{tikzcd}
B & Object \\
& A \arrow[ul] \arrow[u]
\end{tikzcd}$$

multiple inheritance

$\times$

$$\begin{tikzcd}
Object \\
B \arrow[u] \\
A \arrow[u]
\end{tikzcd}$$

✓

multilevel inheritance

```
class A
{
}
```

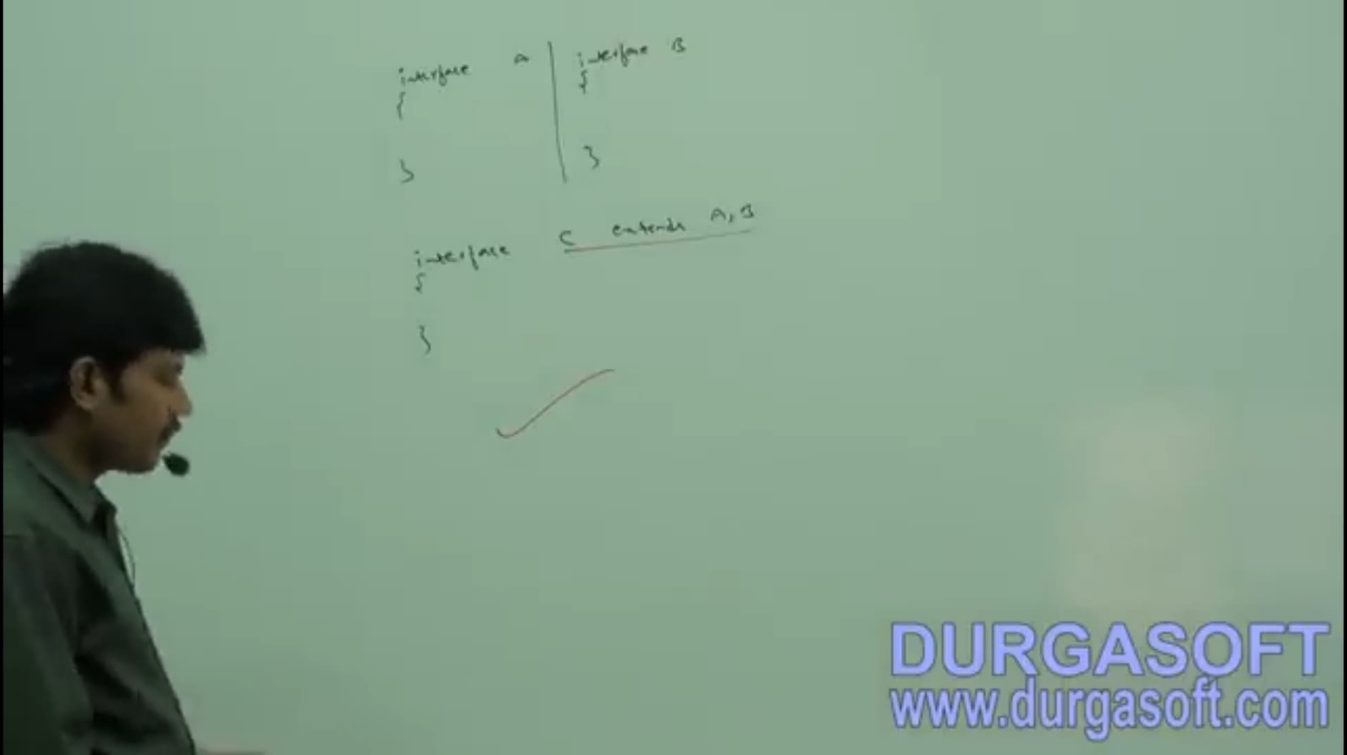
$$\begin{tikzcd}
Object \\
A \arrow[u]
\end{tikzcd}$$

✓

$$\begin{tikzcd}
P_1 \arrow[r] & \underline{m1()} & & P_2 \arrow[r] & \underline{m1()} \\
 & & C \arrow[ull] \arrow[urr] & &
\end{tikzcd}$$

$\textcircled{c}$.m1();

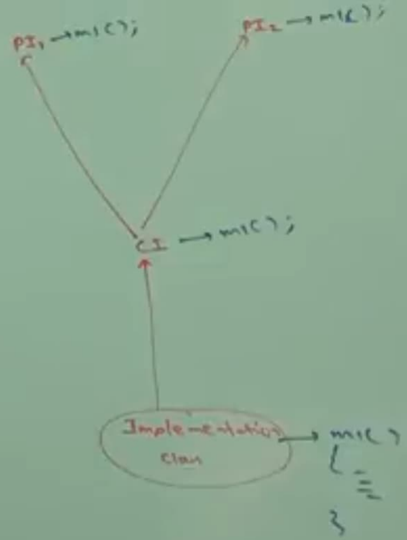
$\boxed{\text{Ambiguity problem}}$



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

interface C extends A,B
{

}
```

✓

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

interface C extends A, B
{

}
```



Cyclic Inheritance:

```
class A extends A
{

}
```

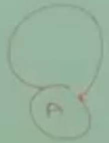
$$\begin{tikzcd}
A \arrow[loop above]
\end{tikzcd}$$

```
class A extends B
{

}
class B extends A
{

}
```

$$\begin{tikzcd}
B \arrow[d, bend left] \\
A \arrow[u, bend left]
\end{tikzcd}$$

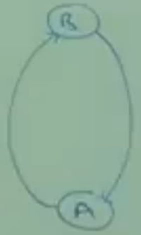
cyclic inheritance:

```
class A extends A
{

}
```

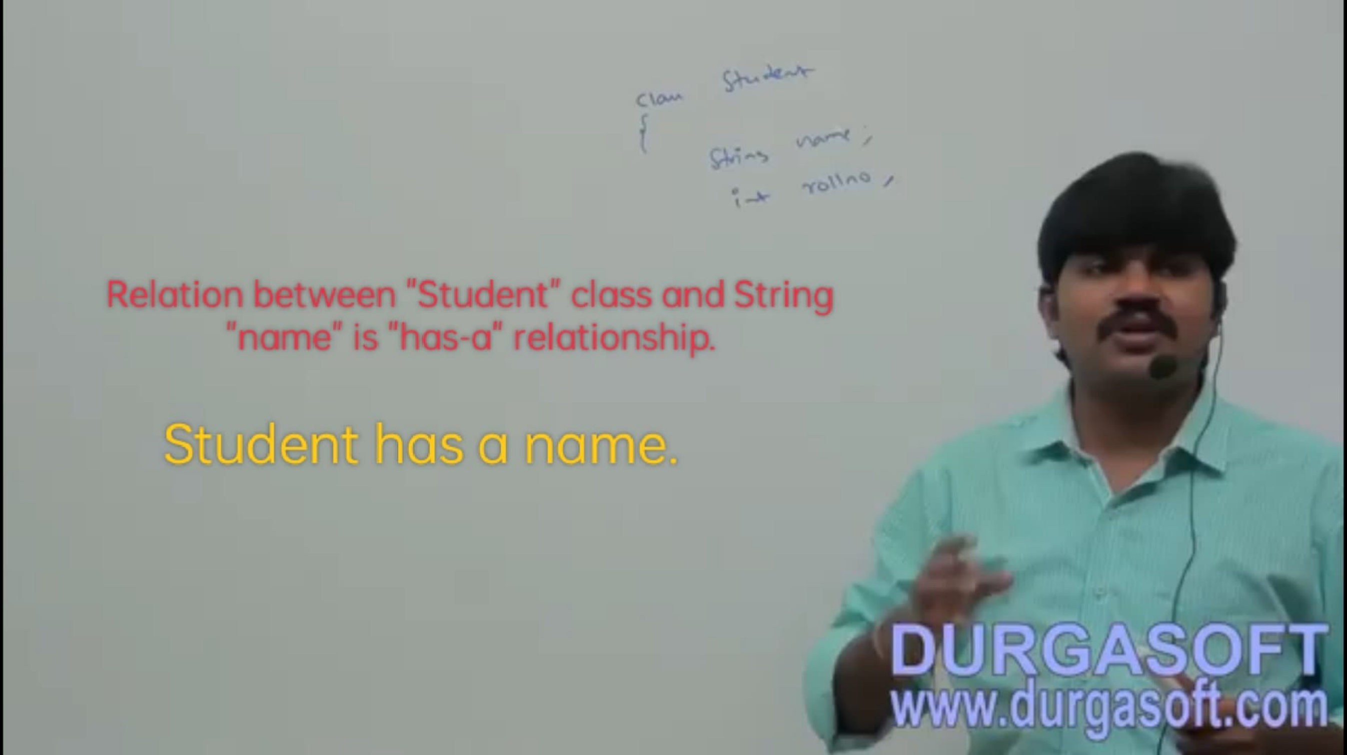


CE: cyclic inheritance involving A

```
class A extends B
{

}
class B extends A
{

}
```



Has-a Relationship



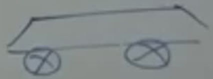
```
class Student
{
    String name;
    int rollno;
```

Relation between "Student" class and String "name" is "has-a" relationship.

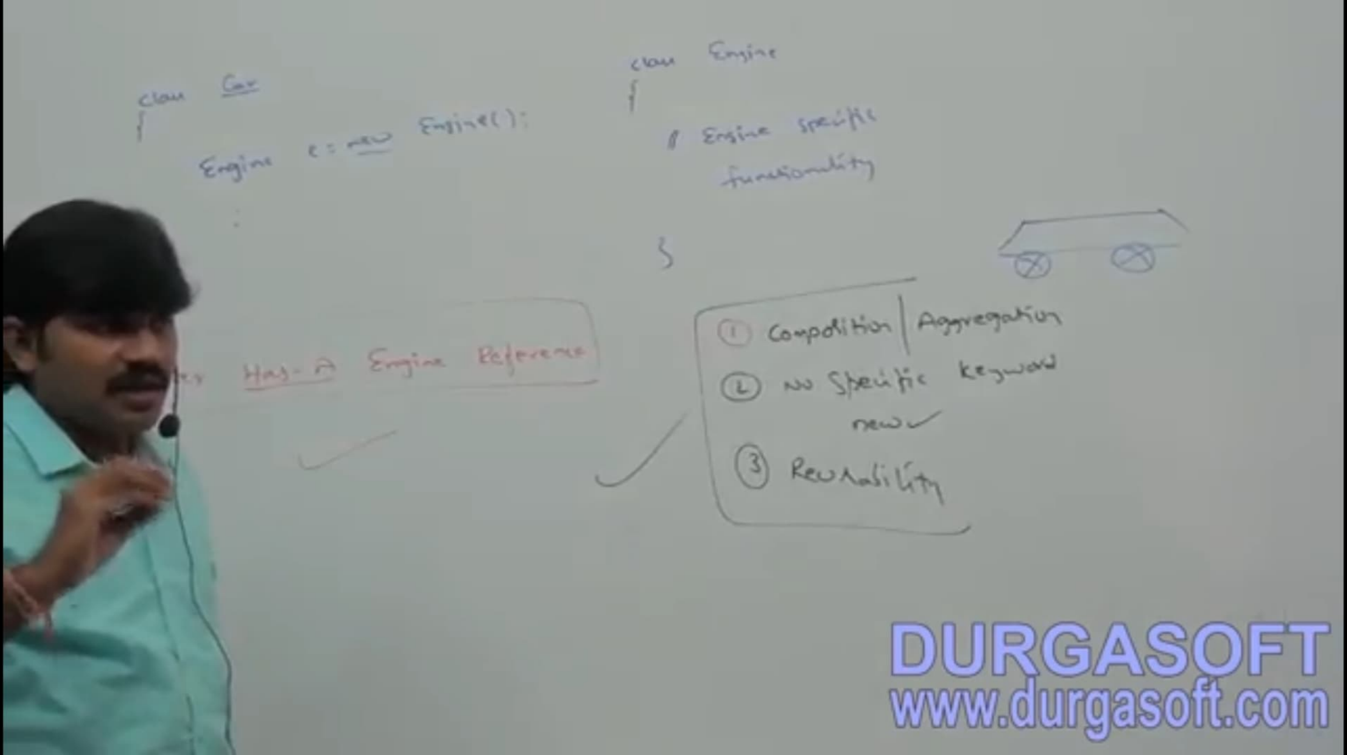
Student has a name.

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

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



Car Has-A Engine Reference ✓



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

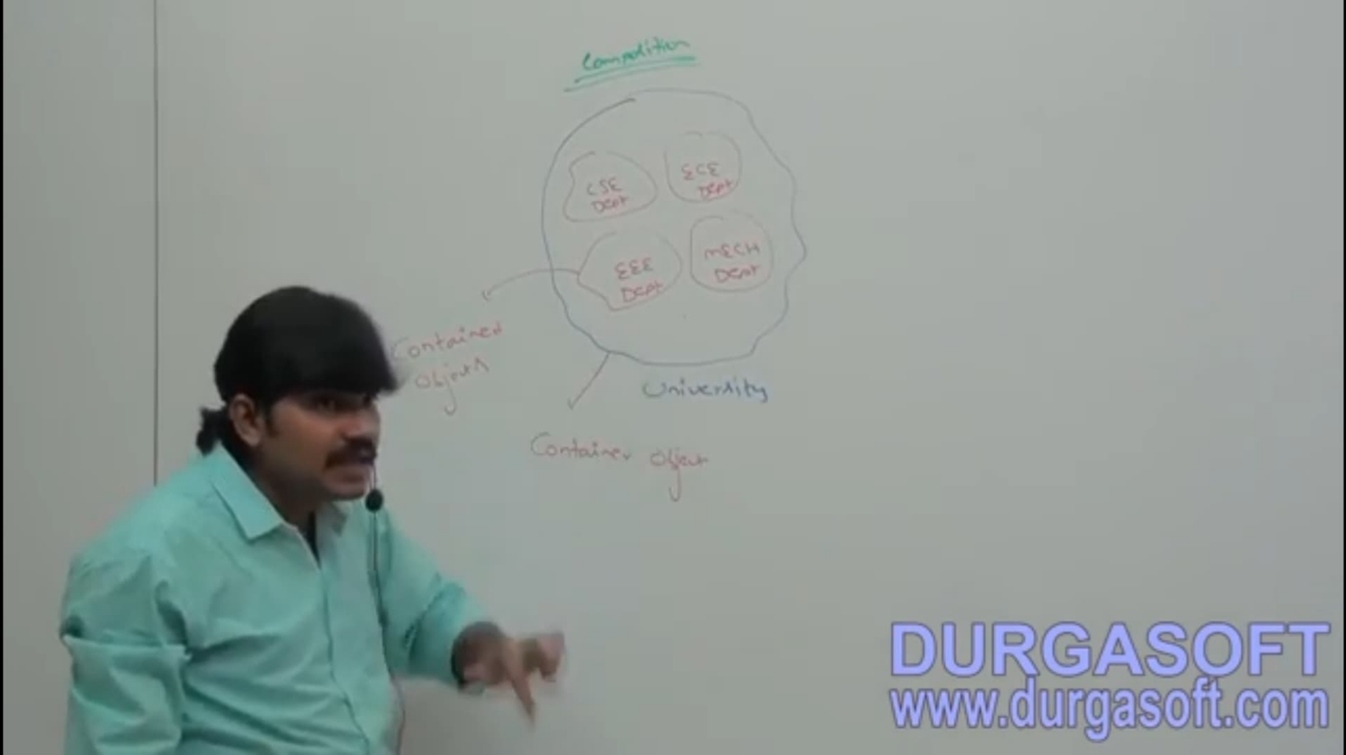
```
class Engine
{
    // Engine specific
       functionality
}
```

Has-A Engine Reference

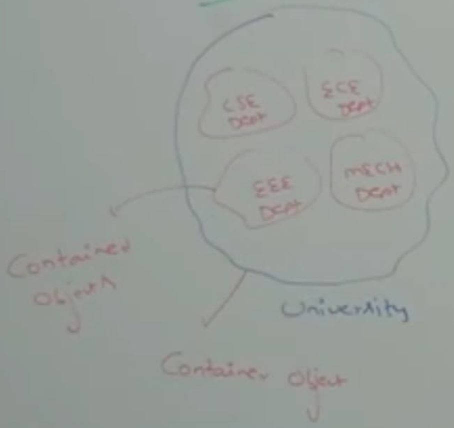
① Composition | Aggregation

② no specific keyword
new ✓

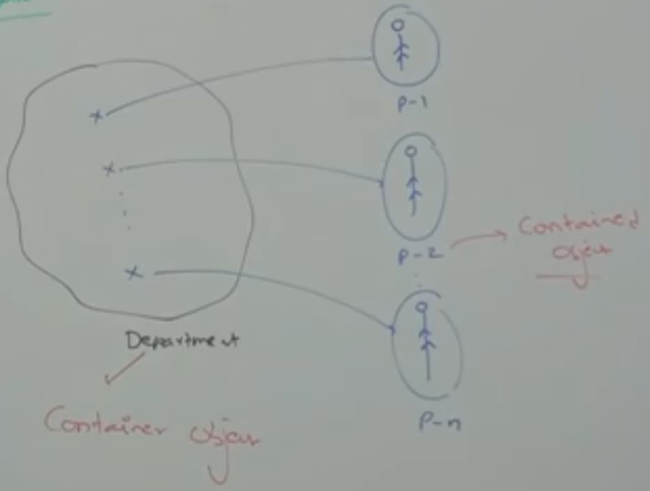
③ Reusability

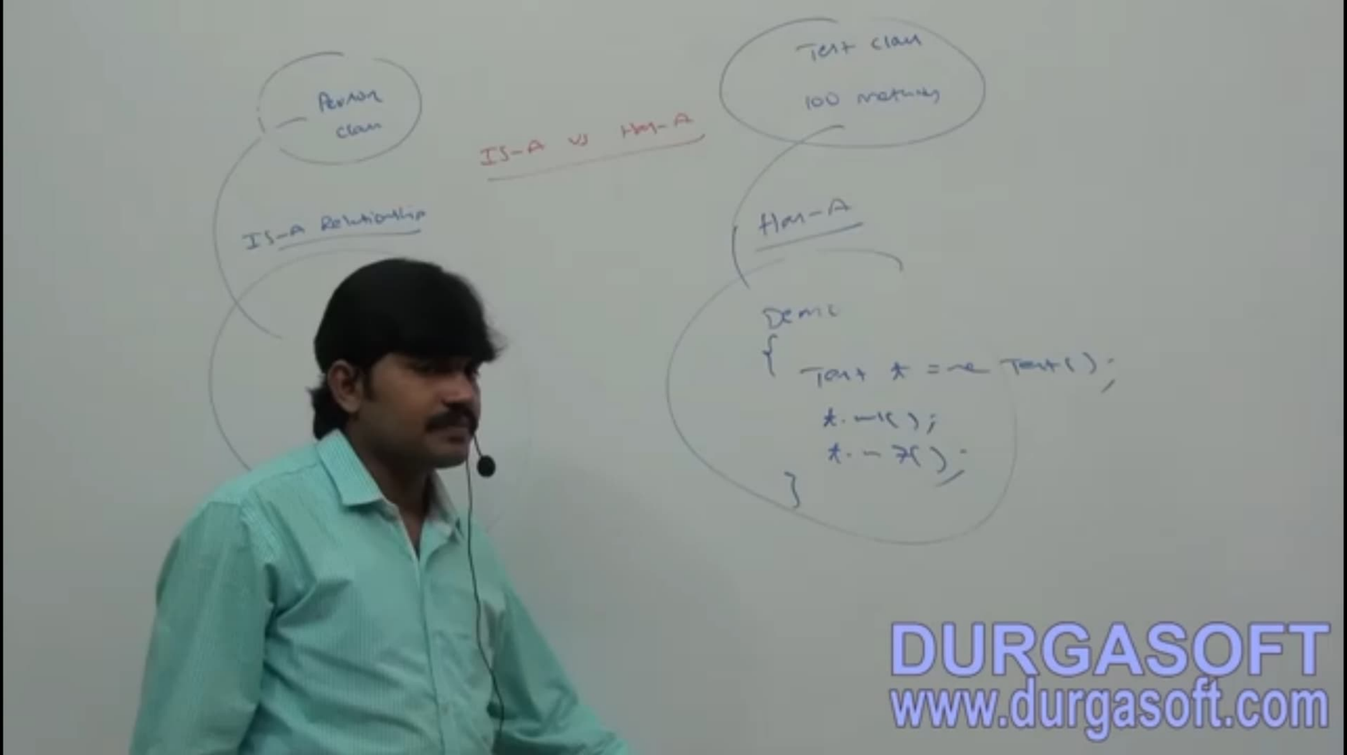

Composition
CSE Dept
ECE Dept
EEE Dept
MECH Dept
Contained Object
University
Container Object

Composition

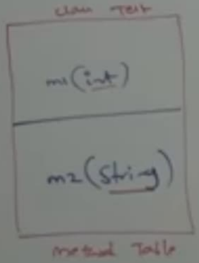


Aggregation





```
Demo
{
    Test t = new Test();
    t.m1();
    t.m7();
}
```



```
class Test
{
    public void m1(int i)
    {
    }
    public void m2(String s)
    {

    }
}
```

```
Test t = new Test();
t.m1(10); ✓
t.m2("durga"); ✓
t.m3(10.5); → CE: cannot find symbol
              symbol: method m3(double)
              location: class Test
```

```
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

```
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);
```

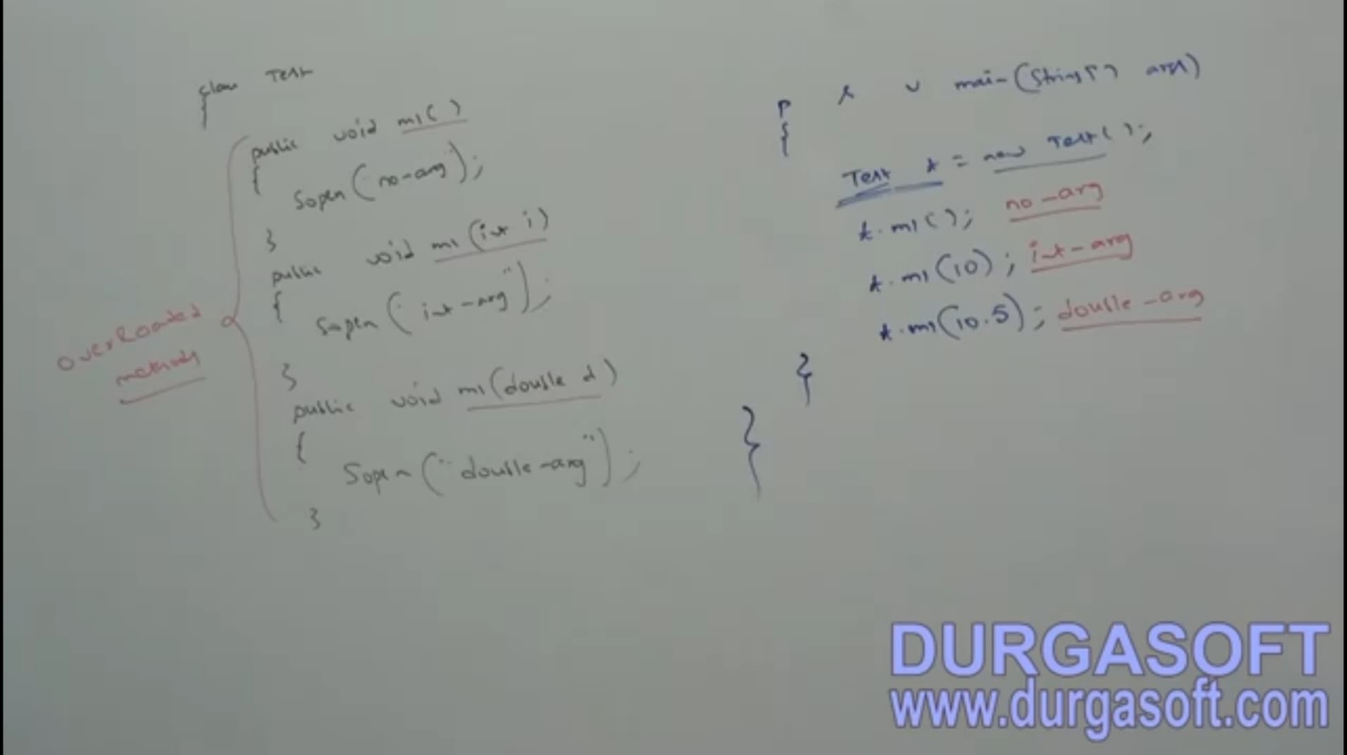
CE: m1(int) is already defined in Test

C

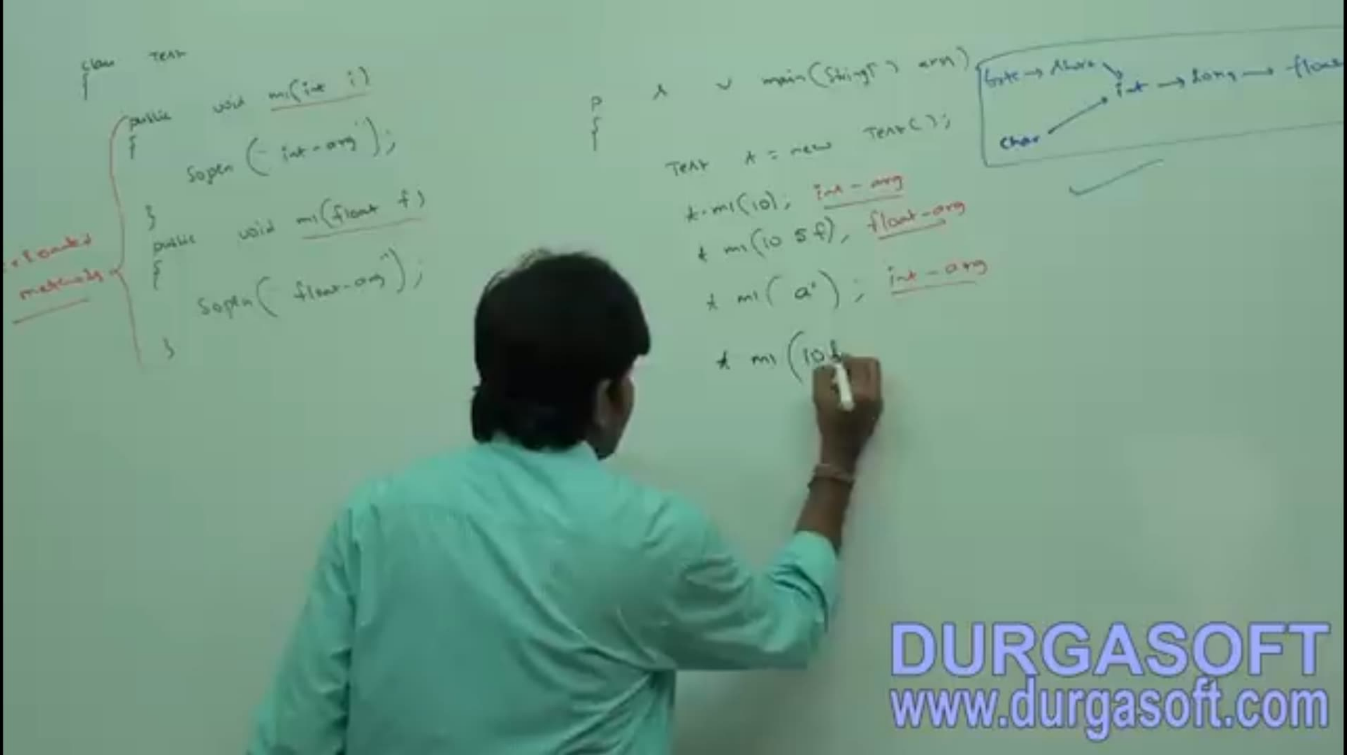
```
abs(int i) ⇒ abs(10);
labs(long l) ⇒ labs(10l);
fabs(float f) ⇒ fabs(10.5f);
```

Java

```
abs(int i)     }
abs(long l)    } overloaded methods
abs(float f)   }
```



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



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

overloaded methods

```
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
```

$$\begin{tikzcd}
byte \arrow[r] & short \arrow[r] & int \arrow[r] & long \arrow[r] & float \\
char \arrow[urr] & & & &
\end{tikzcd}$$

```
Test

ublic 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 find symbol
                  symbol: method m1(double)
                  location: class Test
}
}
```

$$\begin{tikzcd}
byte \arrow[r] & short \arrow[r] & int \arrow[r] & long \arrow[r] & float \arrow[r] & double \\
char \arrow[urr] & & & & &
\end{tikzcd}$$

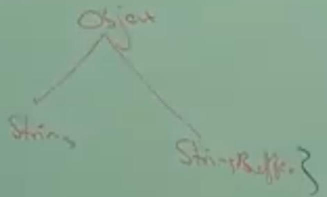
```
class Test
{
    public void m1(String s)
    {
        Sopln("String version");
    }
    public void m1(Object o)
    {
        Sopln("Object version");
    }

    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
    }
}
```

overloaded methods

$$\begin{tikzcd} \text{Object} \\ \text{String} \arrow[u] \end{tikzcd}$$

overloaded methods

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

    p s v main(String[] args)
    {
        Test t = new Test();
        t.m1("durga");                      String version
        t.m1(new StringBuffer("durga"));    StringBuffer version
        t.m1(null);   CE: reference to m1() is ambiguous
    }
}
```



```
class Test
{
    public void m1(int i, float f)
    {
        Sopln("int - float version");
    }
    public void m1(float f, int i)
    {
        Sopln("float - int version");
    }
    p s v main(String[] args)
    {
        Test t = new Test();
        t.m1(10,10.5f);   → int - float version
        t.m1(10.5f,10);   → float - int version
        t.m1(10,10);      → CE: reference to m1() is ambiguous
        t.m1(10.5f,10.5f); → CE: cannot find symbol
                              symbol: method m1(float,f
                              location: class Test
    }
}
```

overloaded methods

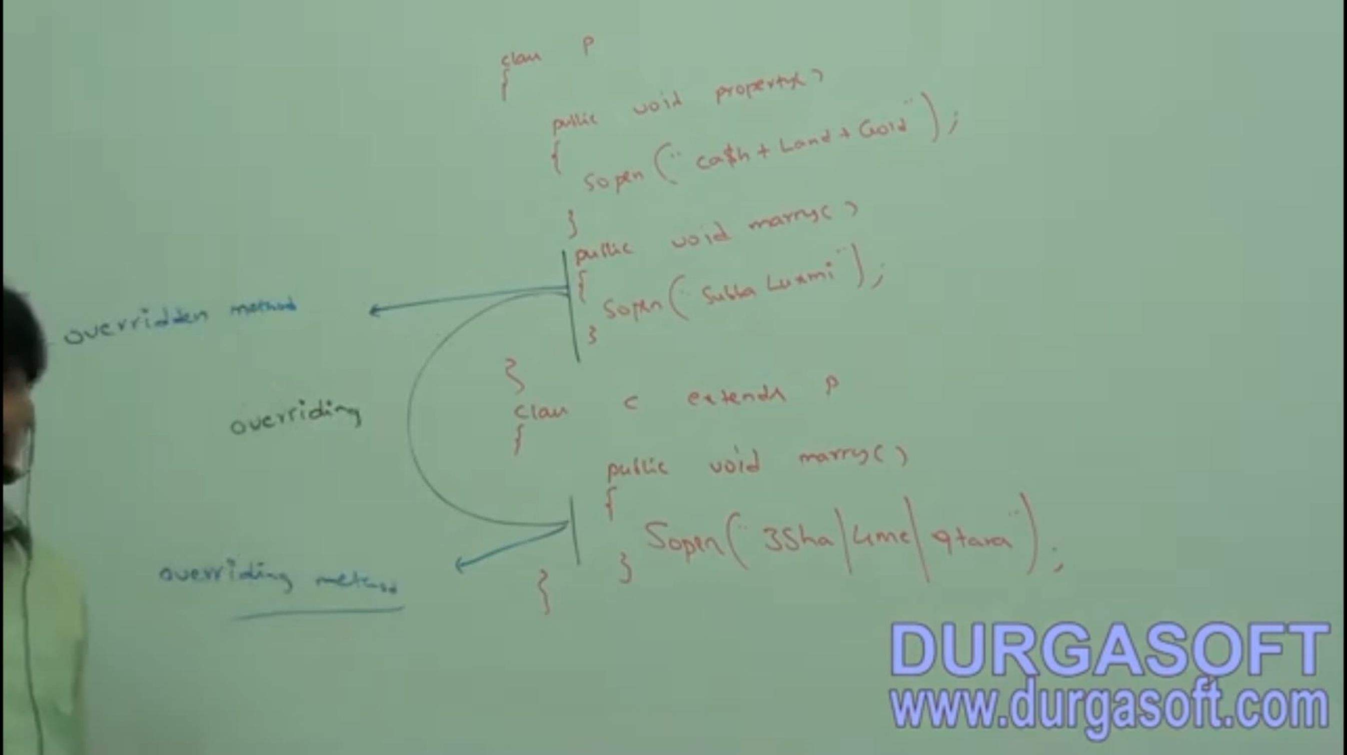
```
class Test
{
                    public void m1(int x)
                    {
                        Sopln("General method");
                    }
overloaded          public void m1(int... x)
method              {
                        Sopln("var-arg method");
                    }

    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
    }
}
```

```
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
    }
}
```

overloaded method

```
class Animal
{
}
class Monkey extends Animal
{
}
class Test
{
                  public void m1(Animal a)
                  {
                      Sopln("Animal version");
overloaded        }
methods           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
  }
}
```

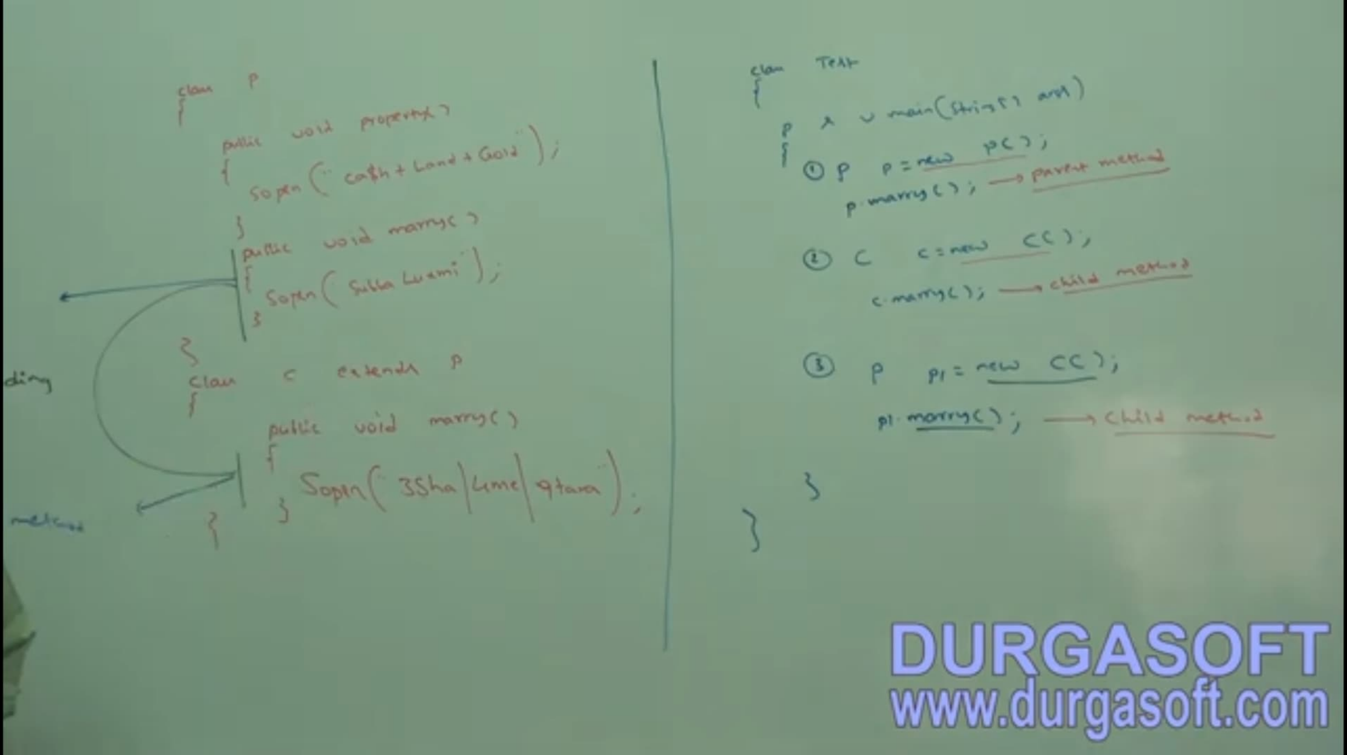


```
class P
{
   public void property()
   {
      Sopln(" ca$h + Land + Gold");
   }
   public void marry()
   {
      Sopln(" Subba Laxmi");
   }
}
class C extends P
{
   public void marry()
   {
      Sopln(" 3Sha|4me|9tara");
   }
}
```

overridden method

overriding

overriding method

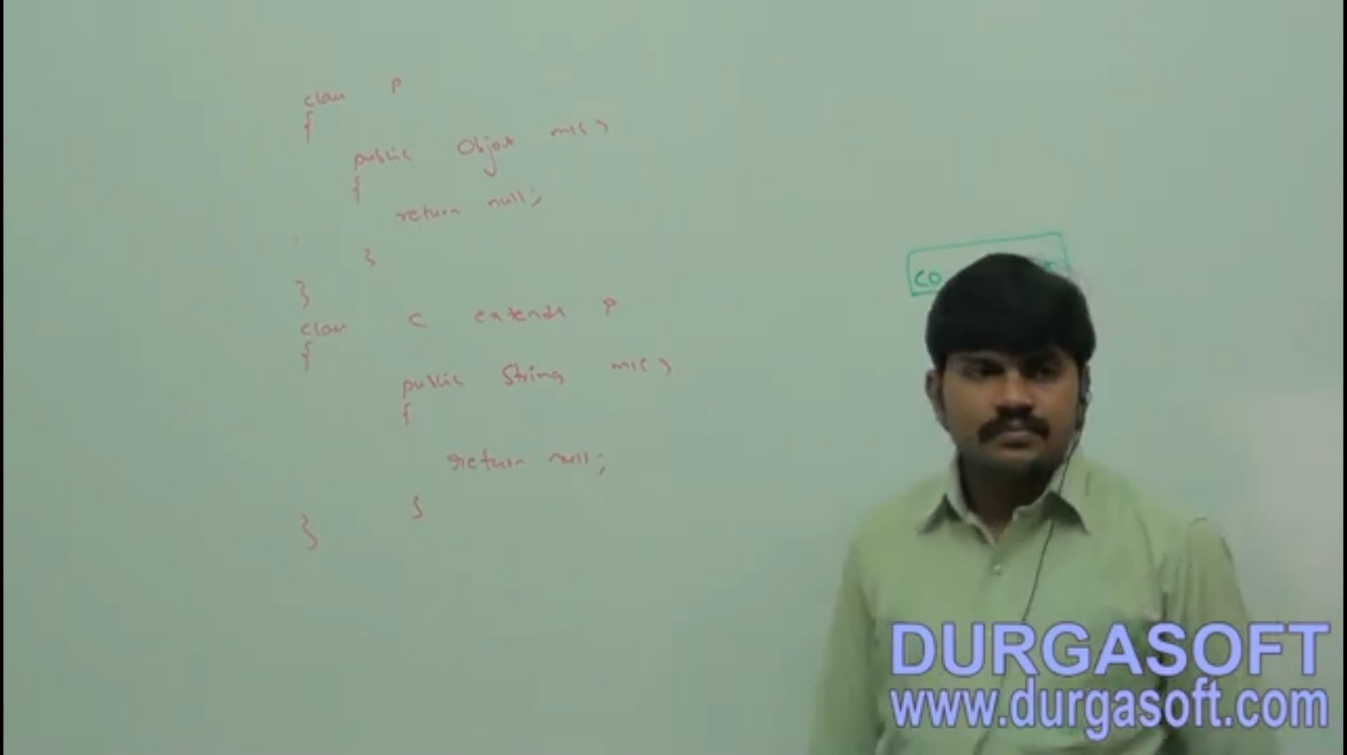


ding

method

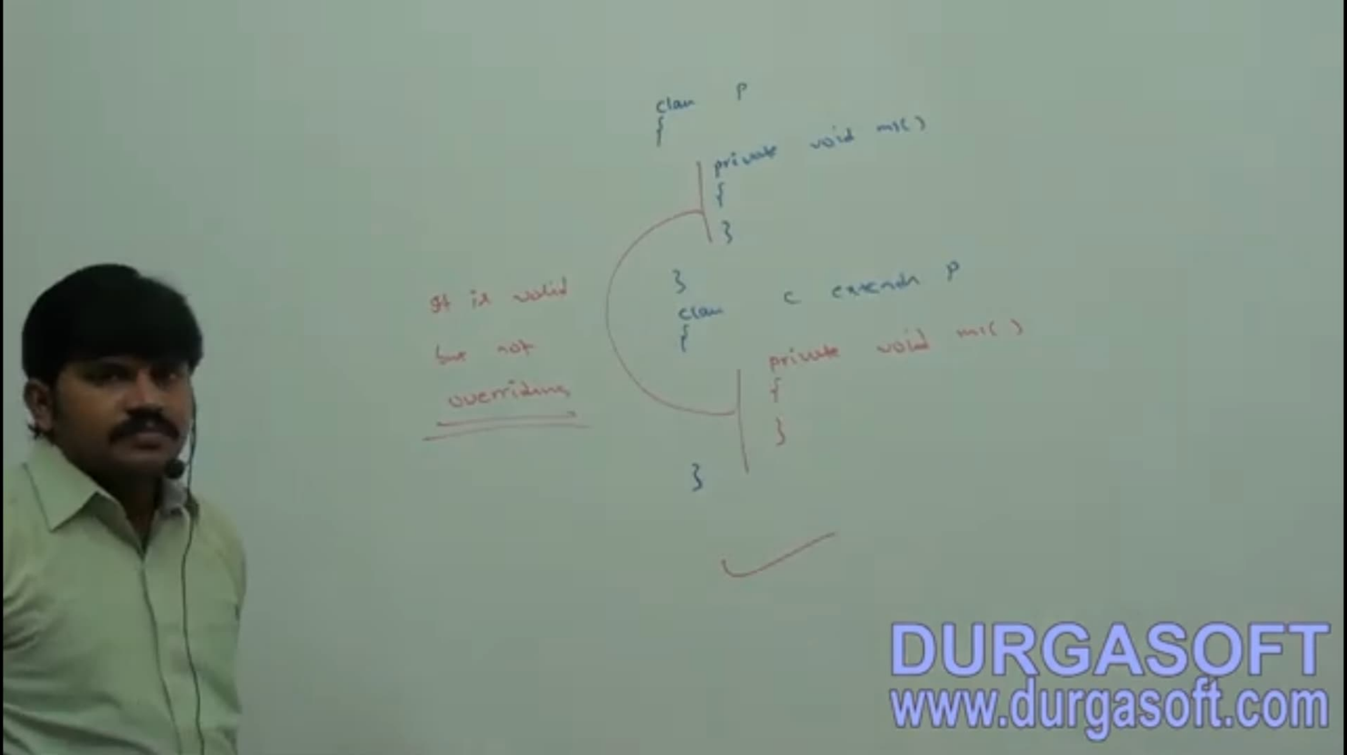
```
class P
{
    public void property()
    {
        Sopln("cash+Land+Gold");
    }
    public void marry()
    {
        Sopln("Subba Laxmi");
    }
}
class C extends P
{
    public void marry()
    {
        Sopln("3Sha|4me|9tara");
    }
}
```

```
class Test
{
    p s v main(String[] args)
    {
        ① P p = new P();
           p.marry(); → parent method

        ② C c = new C();
           c.marry(); → child method

        ③ P p1 = new C();
           p1.marry(); → child method
    }
}
```



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

parent class method Return type:	Object	Number	String	double
Child class method Return type:	Object | String | StringBuffer ...	Number | Integer |	Object	int
	✓	✓	✗	✗



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

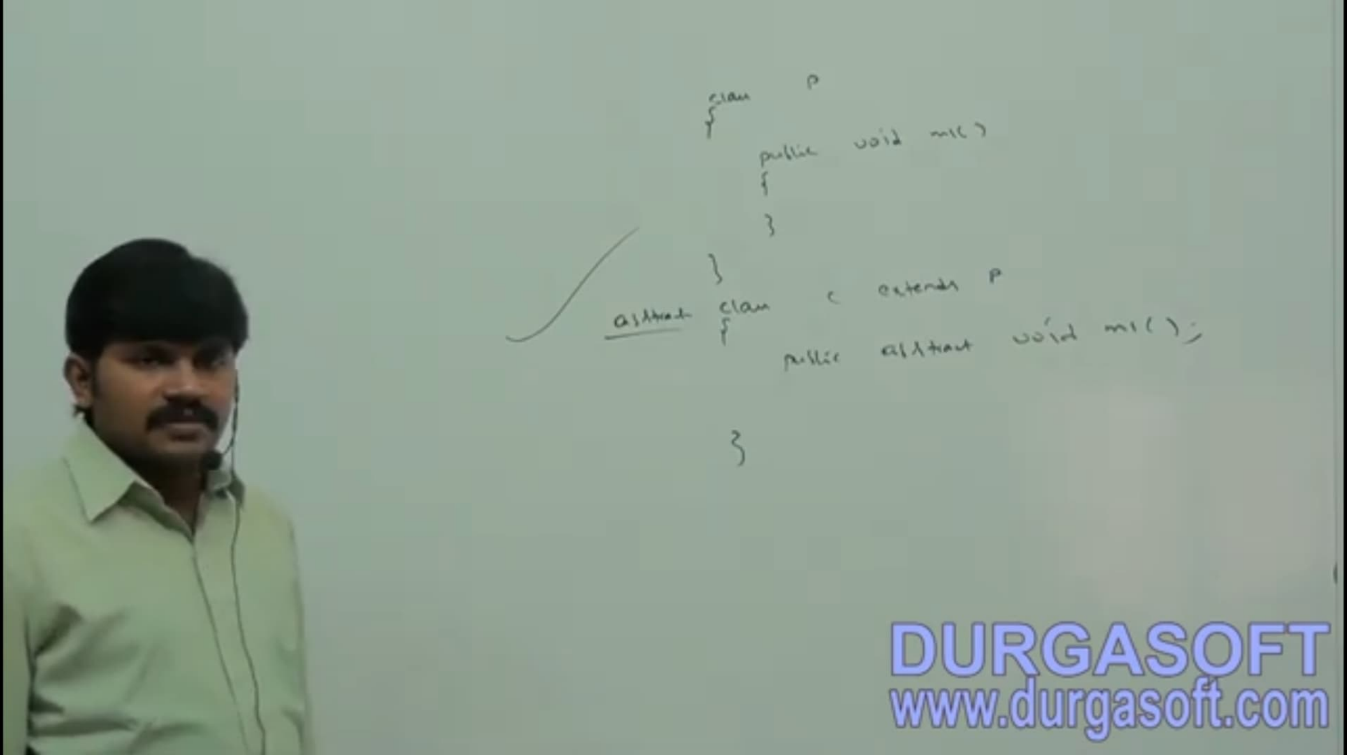
It is valid
but not
overriding

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

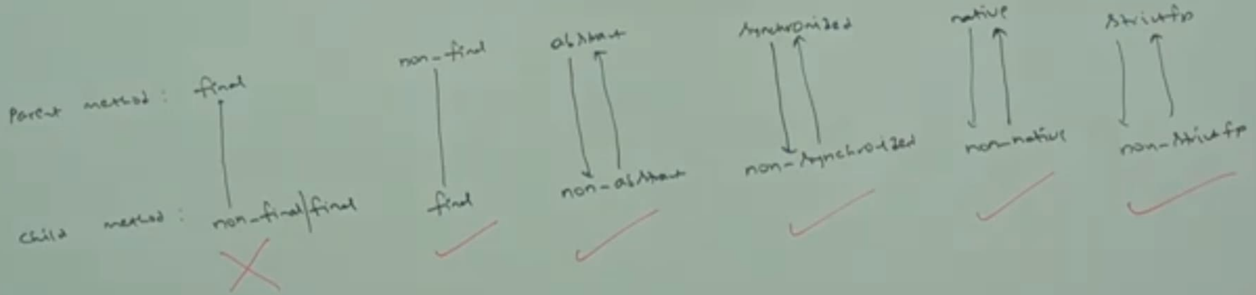
CE: m1() in C cannot override m1() in P;
overridden method is final

```
abstract class P
{
    public abstract void m1();

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

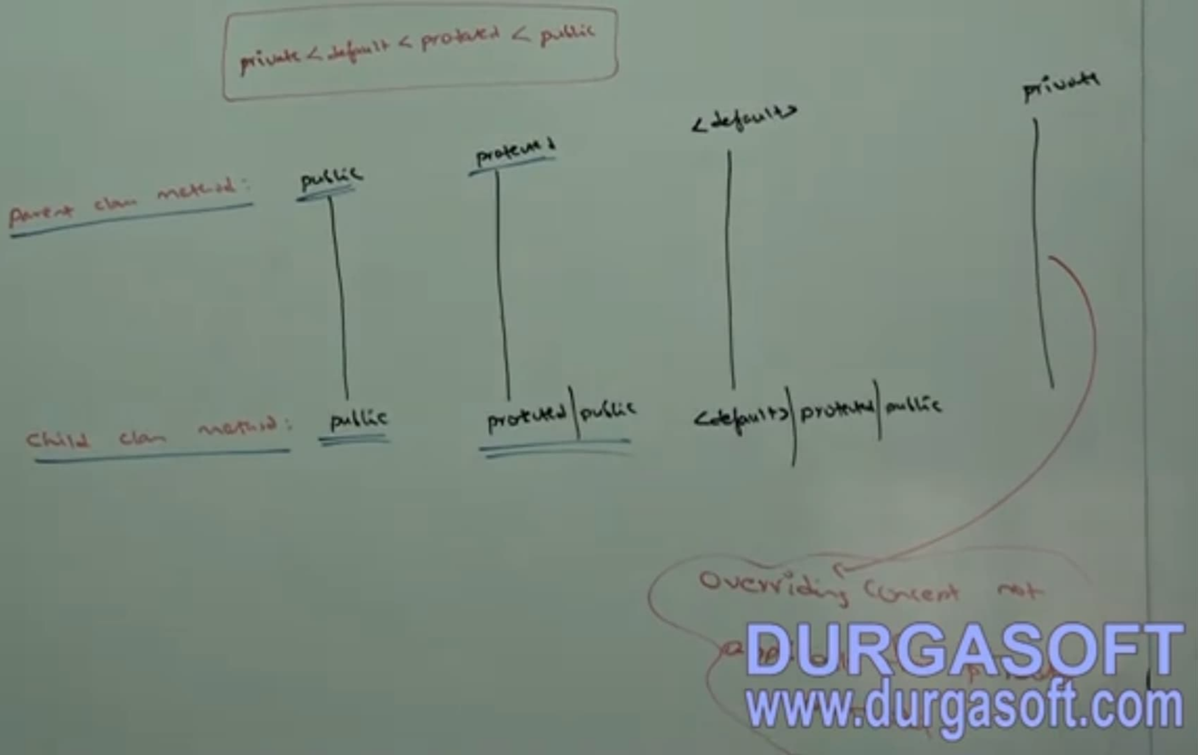
✓

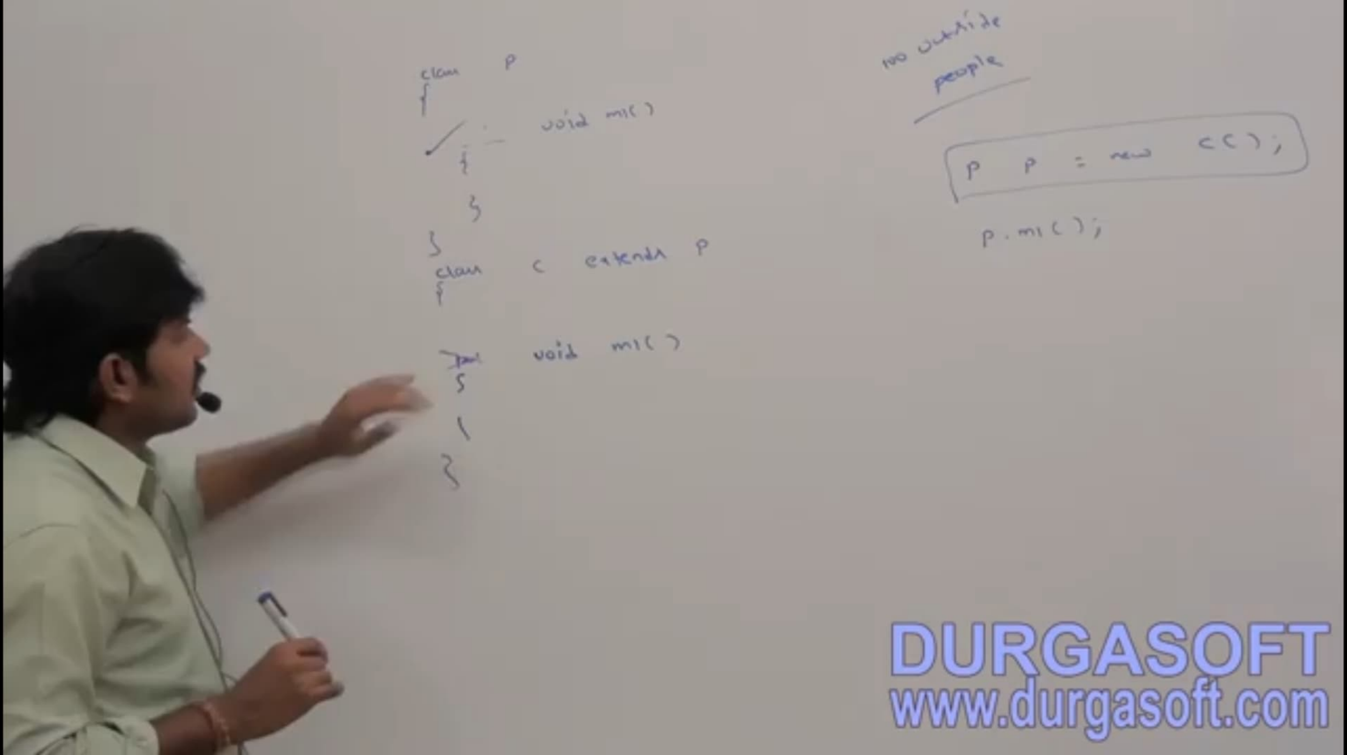


```
class P
{
    public void m1()
    {

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


}
```


Parent method :
Child method :
final
non-final|final
✗
non-final
final
✓
abstract
non-abstract
✓
synchronized
non-synchronized
✓
native
non-native
✓
strictfp
non-strictfp
✓

$\boxed{\text{private} < \text{default} < \text{protected} < \text{public}}$


private < default < protected < public
Parent class method :
public
protected
<default>
private
Child class method :
public
protected | public
<default> | protected | public
Overriding concept not
applicable for private



```
class P
{
    void m1()
    {
    }
}
class C extends P
{
    ~~pu~~ void m1()
    {
    }
}
```

no outside people

```
P p = new C();

p.m1();
```

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, Inte

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

✓ 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

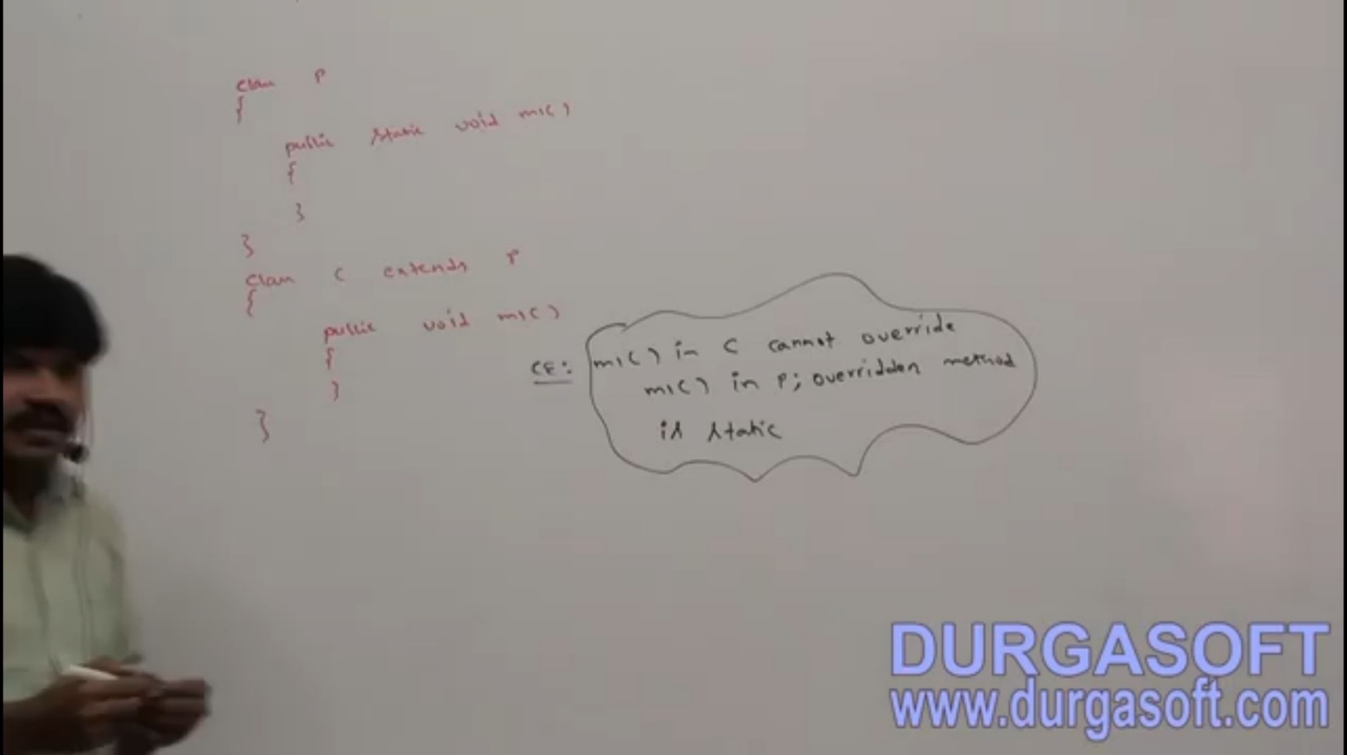
✗ ⑥ p: public void m1() throws IOExcep
c: public void m1() throws EO

✓ ⑦ p: public void m1() throws IOEx
c: public void m1() throws A

```
class P
{
    public void m1() throws IOException
    {
    }
}
class C extends P
{
        public void m1()   XCE
        {
        }
}
```

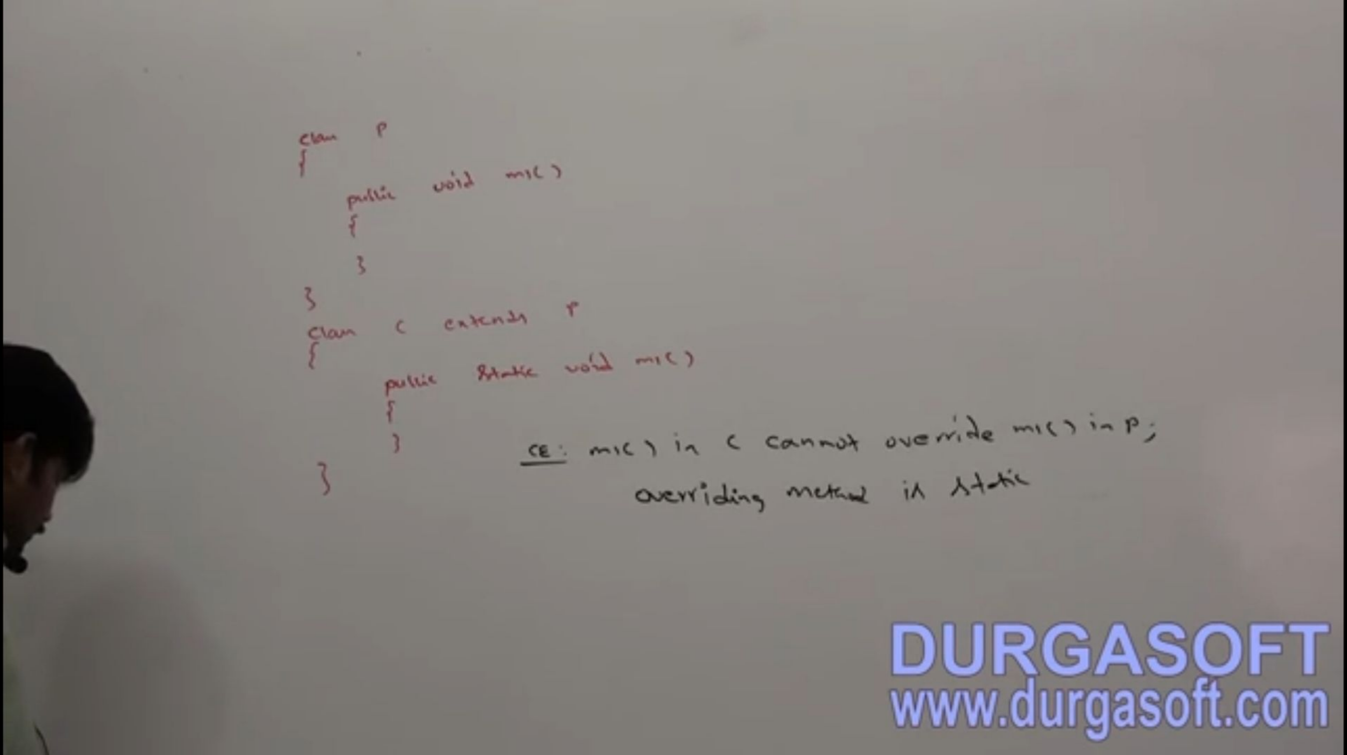
1000's of
outside
people

```
P p = new C();
try
{
p.m1();
}
catch(IOException e)
{
}
```



```
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

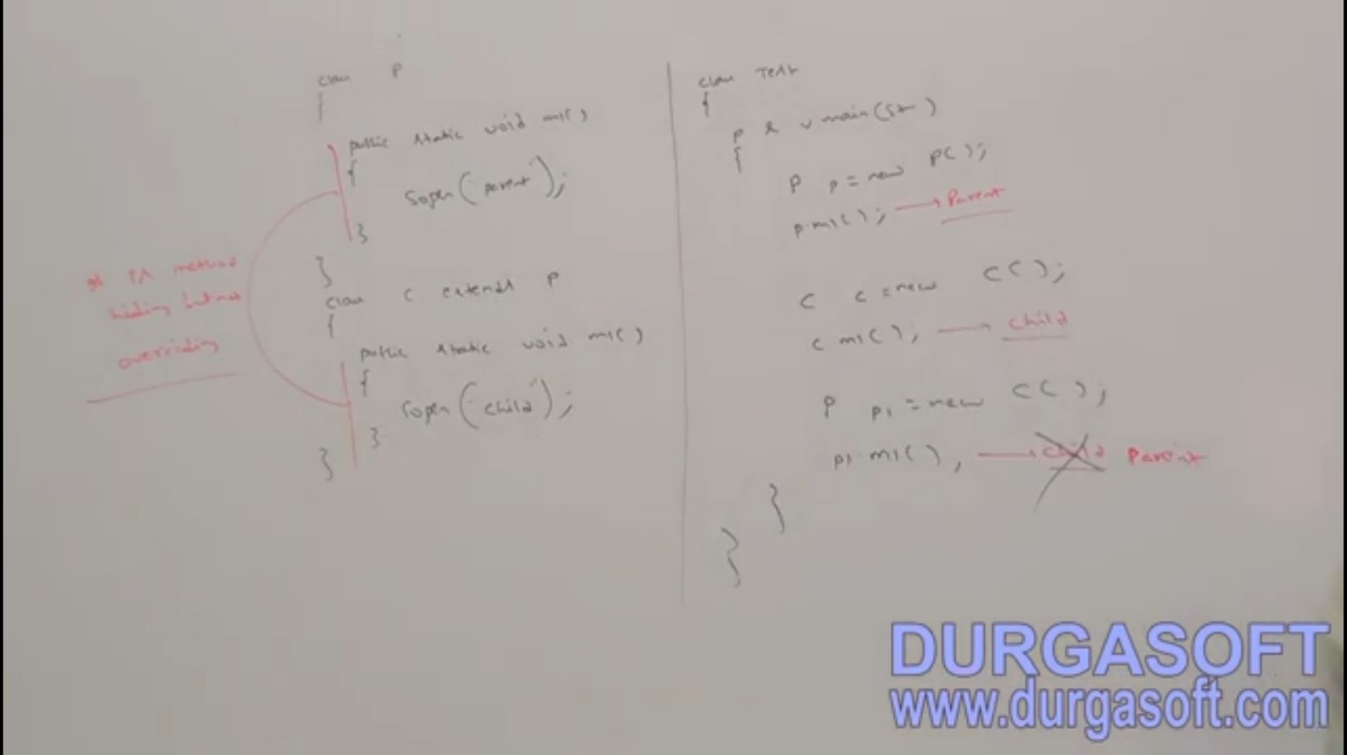


```
class P
{
    public void m1()
    {

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

    }
}
```

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

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

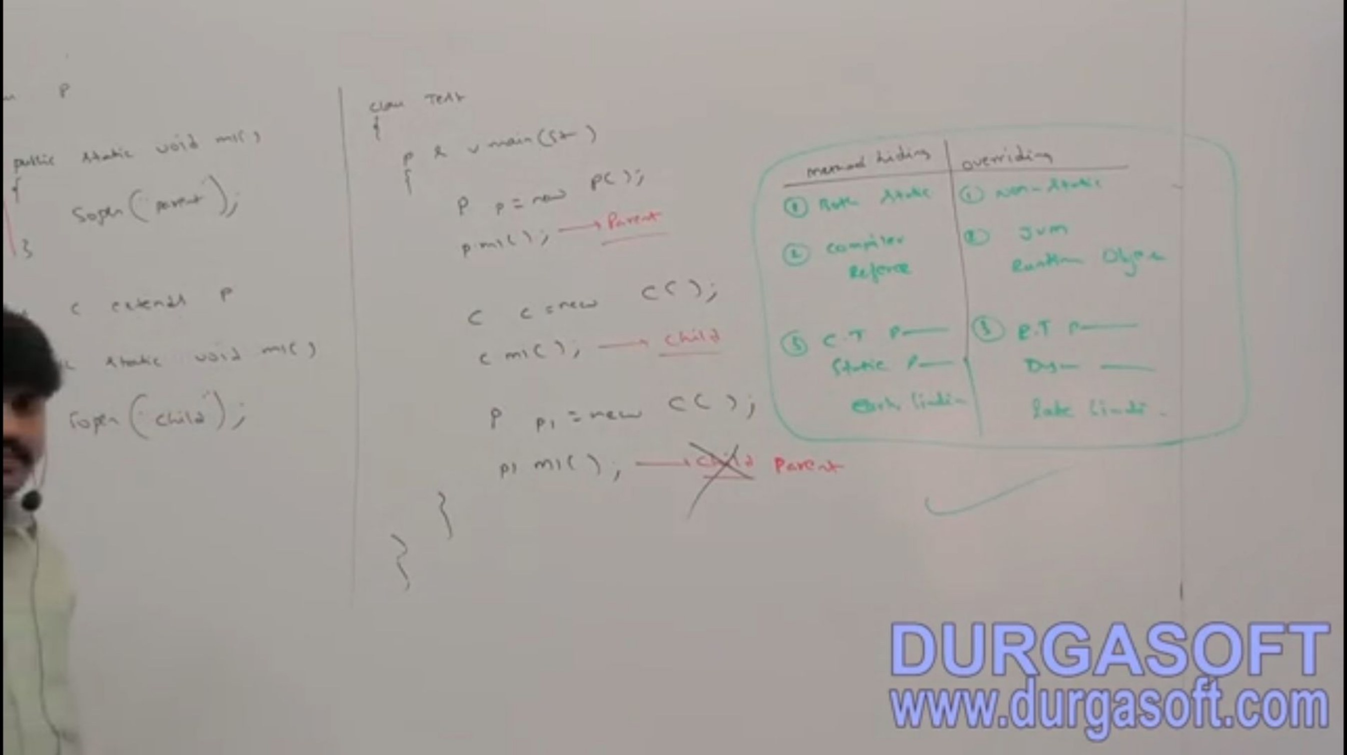
It is method hiding but not overriding ✓



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

it is method hiding but not overriding

```
class Test
{
    p s v main(Str )
    {
        P p = new P();
        p.m1(); → Parent

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

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



```
P

public static void m1()
{
    Sopln("parent");
}

c extends P

static void m1()

    Sopln("child");
```

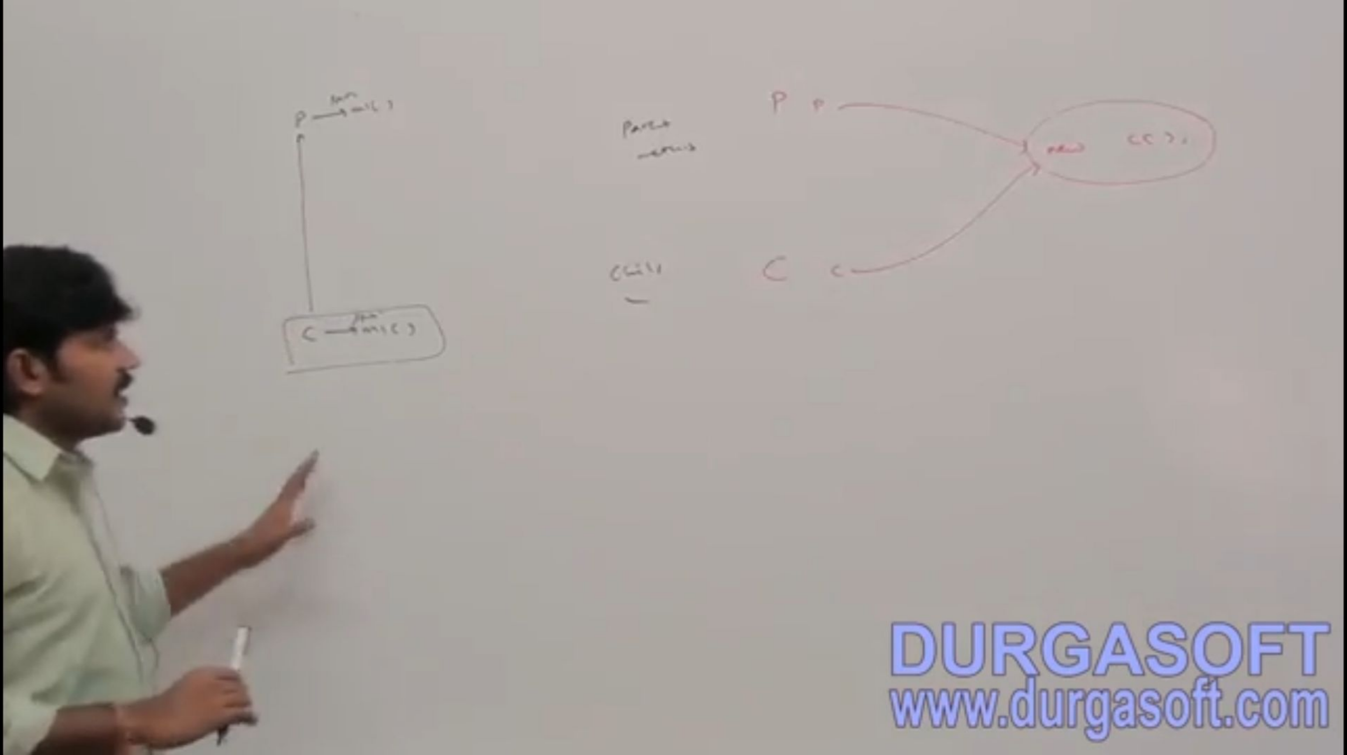
```
class Test
{
    p s v main(St )
    {
        P p = new P();
        p.m1(); ⟶ Parent

        C c = new C();
        c.m1(); ⟶ child

        P p1 = new C();
        p1.m1(); ⟶ ~~child~~ Parent
    }
}
```

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.

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

it is method hiding but not overriding

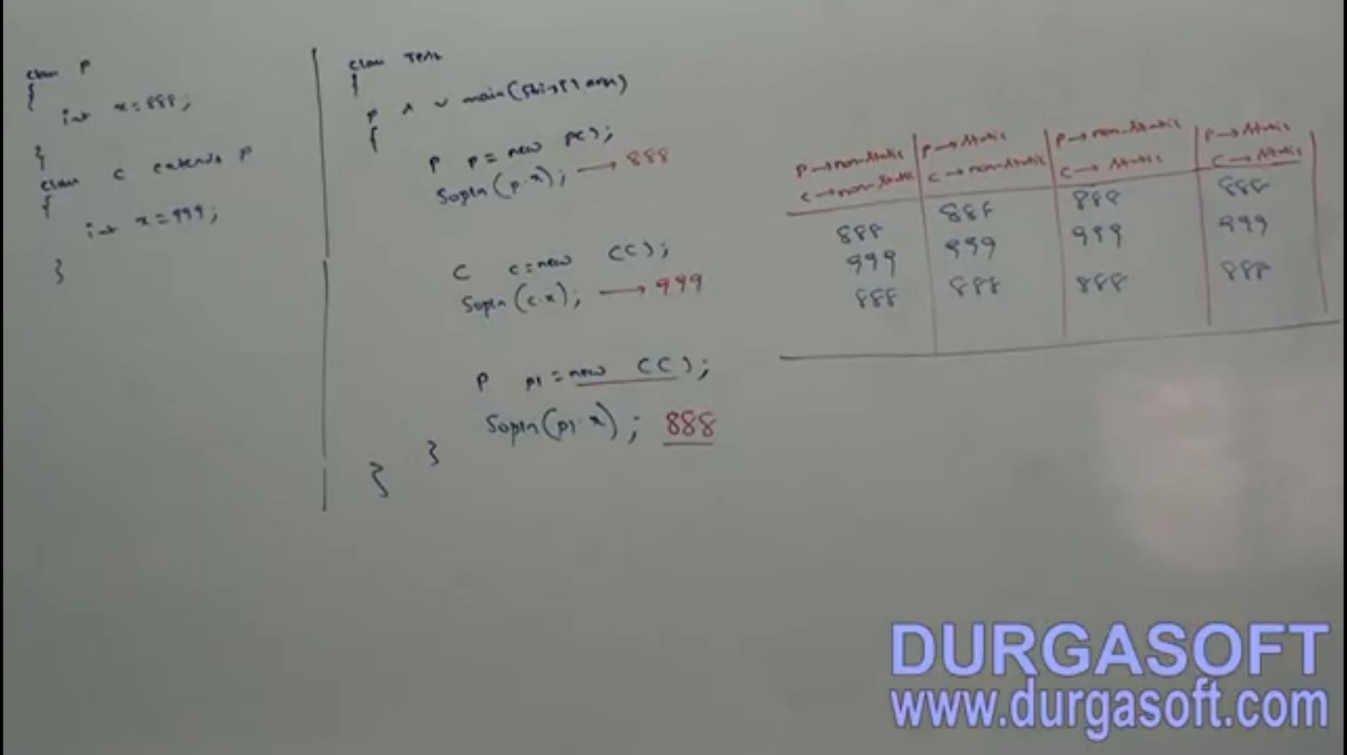
```
class Test
{
    p s v main(St )
    {
        P p = new P();
        p.m1(); → Parent

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

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


P → m1()
C → m1()
Parent
Child
P p
C c
new C();

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

it overloading but not overriding

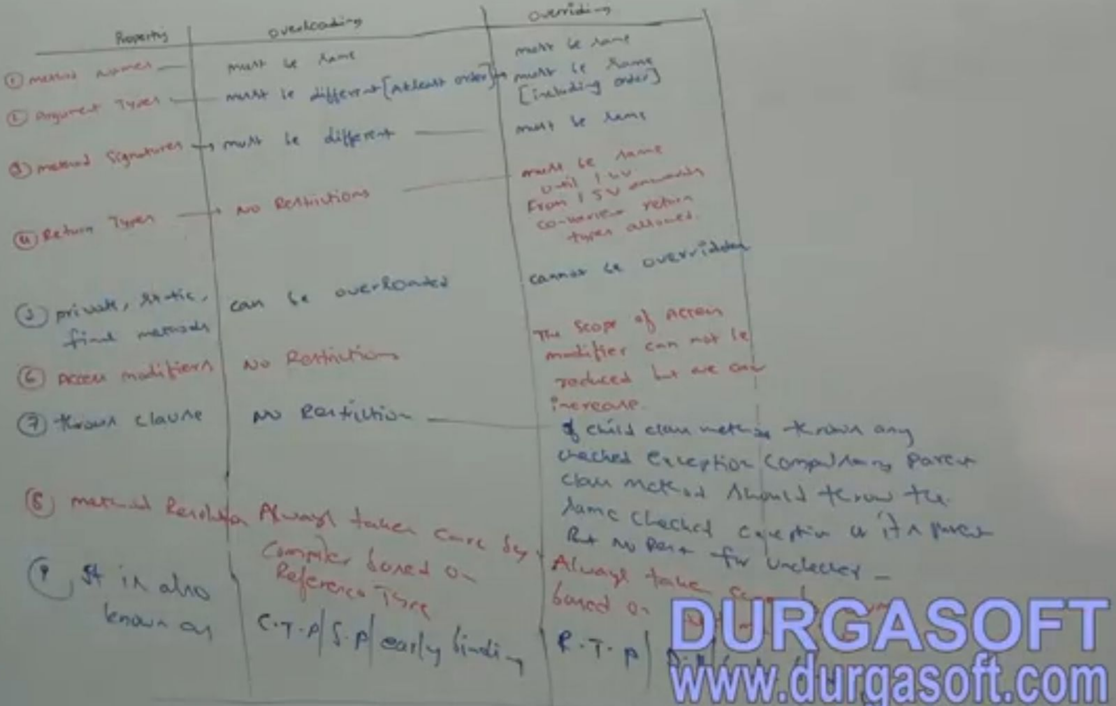
```
class Test
{
    p s v main(String[] args)
    {
        P p = new P();
        p.m1(10); → Parent

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

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



```
class P
{
    int x=888;

}
class C extends P
{
    int x=999;

}
```

```
class Test
{
  p s v main(String[] args)
  {
      P p=new P();
      Sopln(p.x); ⟶ 888

      C c=new C();
      Sopln(c.x); ⟶ 999

      P p1=new C();
      Sopln(p1.x); 888
  }
}
```

P → non-static C → non-static	P → static C → non-static	P → non-static C → static	P → static C → static
888	888	888	888
999	999	999	999
888	888	888	888



Property	overloading	overriding
① method names	must be same	must be same
② Argument Types	must be different [Atleast order]	must be same [including order]
③ method signatures	must be different	must be same
④ Return Types	No Restrictions	must be same until 1.4v From 1.5v onwards co-varient return types allowed.
⑤ private, static, final methods	can be overloaded	cannot be overridden
⑥ Access modifiers	No Restrictions	The scope of access modifier 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 its parent but no Rest for unchecked
⑧ method Resolution	Always taken care by compiler based on Reference Type	Always take care by [illegible] based on [illegible]
⑨ It is also known as	C.T.P / S.P / early binding	R.T.P / [illegible]

$\boxed{\text{public void m1(int x) throws IOException}}$

overriding ① public void m1(int i)

overloading ② public static int m1(long l)

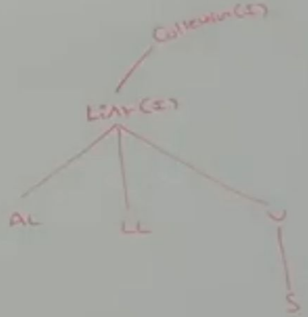
overriding ~~③~~ public static void m1(int i)

overriding ~~④~~ public void m1(int i) throws Exception

~~⑤~~ public static abstract void m1(double d);

CE: illegal combination of modifiers

```
           ┌ new AL();
           │ new LL();
List l = ──┤ new Stack();
           └ new Vector();
```

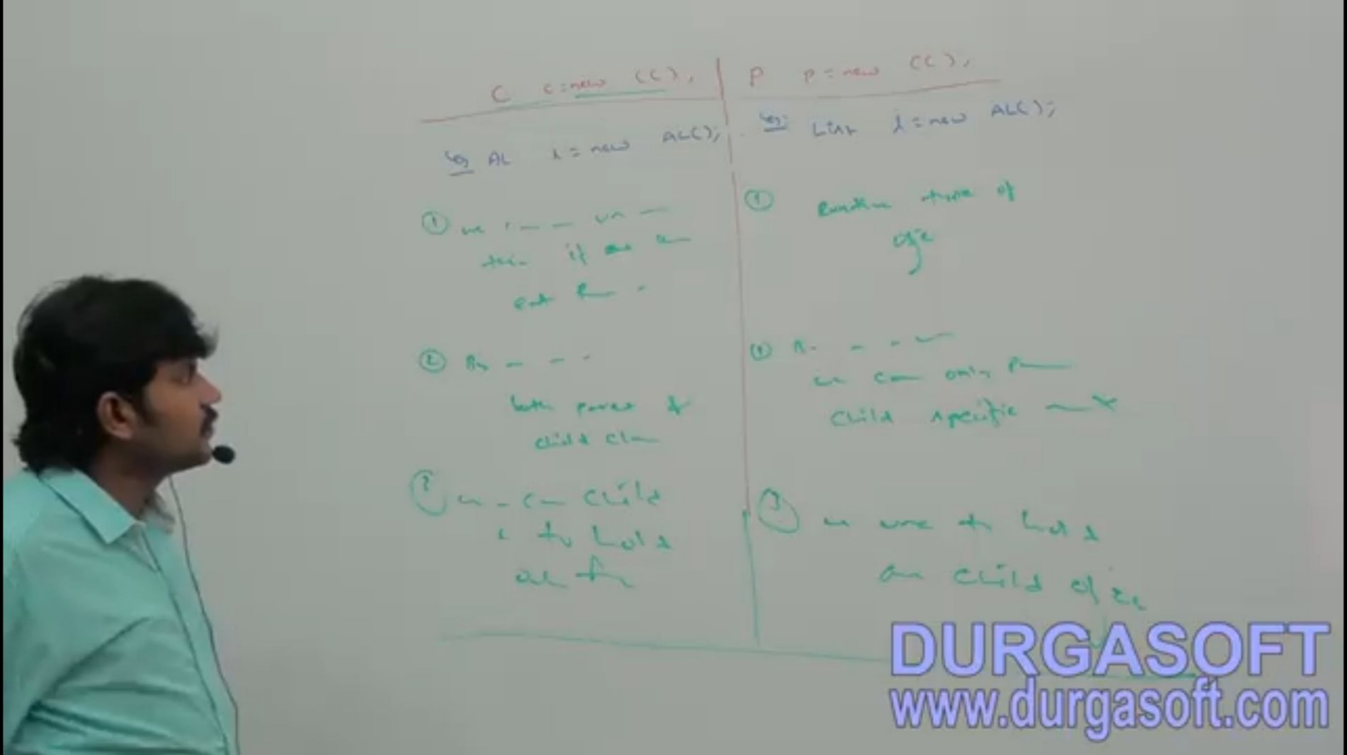


$$\begin{tikzcd}
P \arrow[r] & m1() \\
C \arrow[u] \arrow[r] & m2()
\end{tikzcd}$$

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

```
p.m1(); ✓
p.m2(); ✗ → CE: cannot find symbol
               symbol : method m2()
               location : class P
```

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



C c=new C(); | P p=new C();

eg: AL l=new AL(); | eg: List l=new AL();

① we [illegible] this if [illegible] [illegible]

② By [illegible] both parent & child cl[illegible]

③ we [illegible] child [illegible] to hold [illegible]

① Runtime [illegible] of obj

② B[illegible] we can only [illegible] child specific [illegible] ×

③ we use [illegible] to hold [illegible] child obj[illegible]

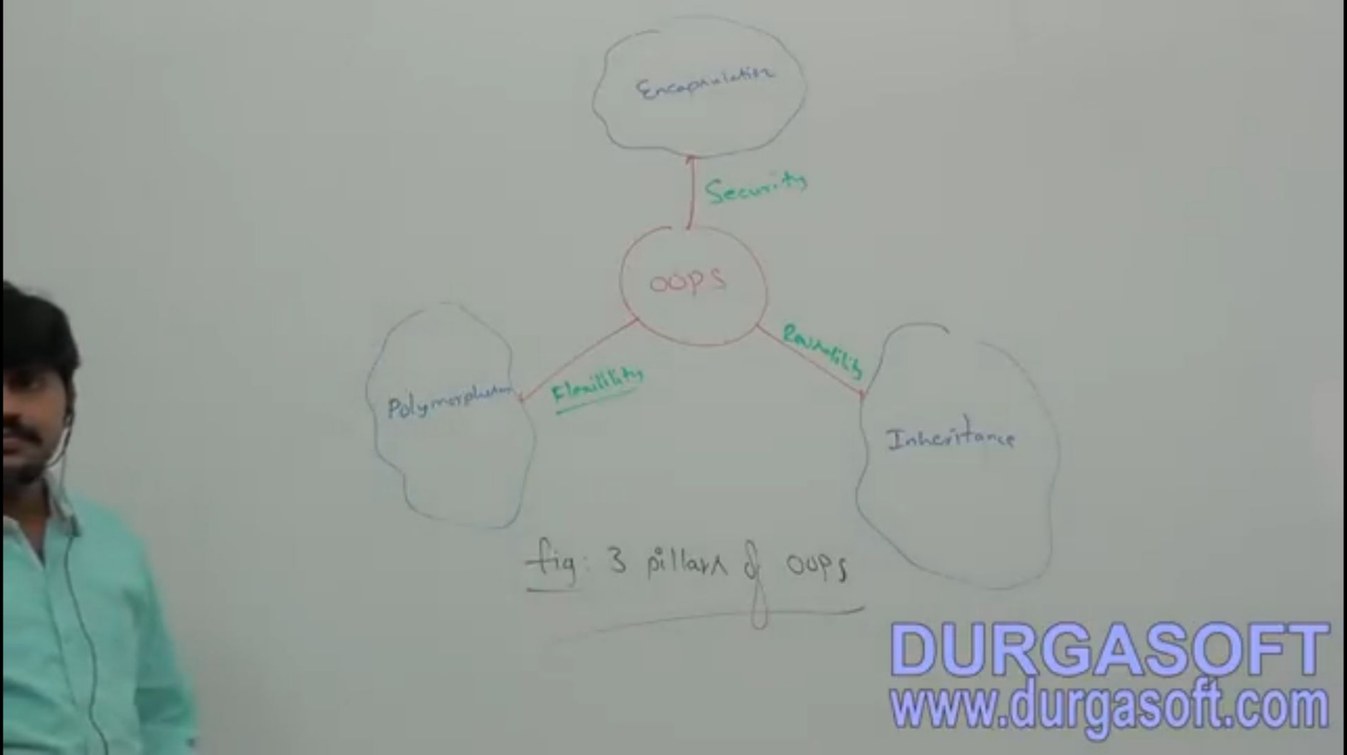
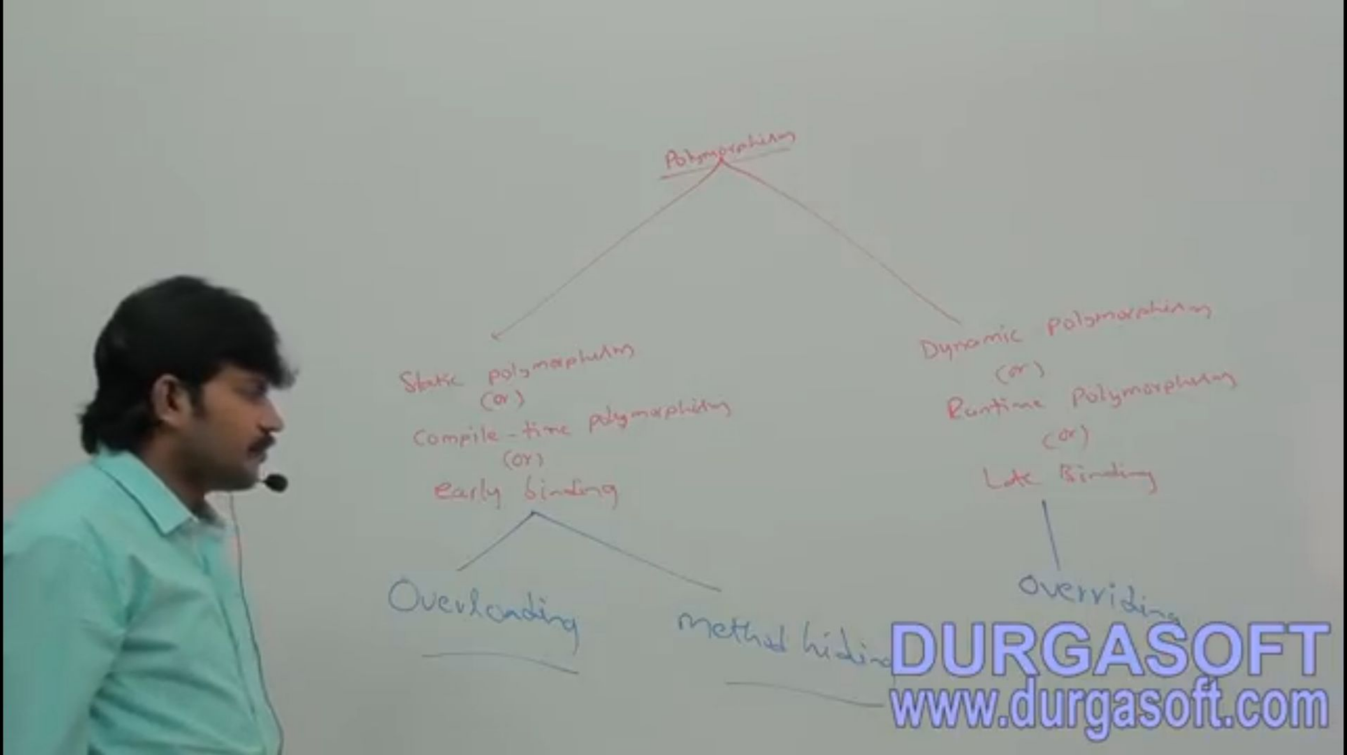
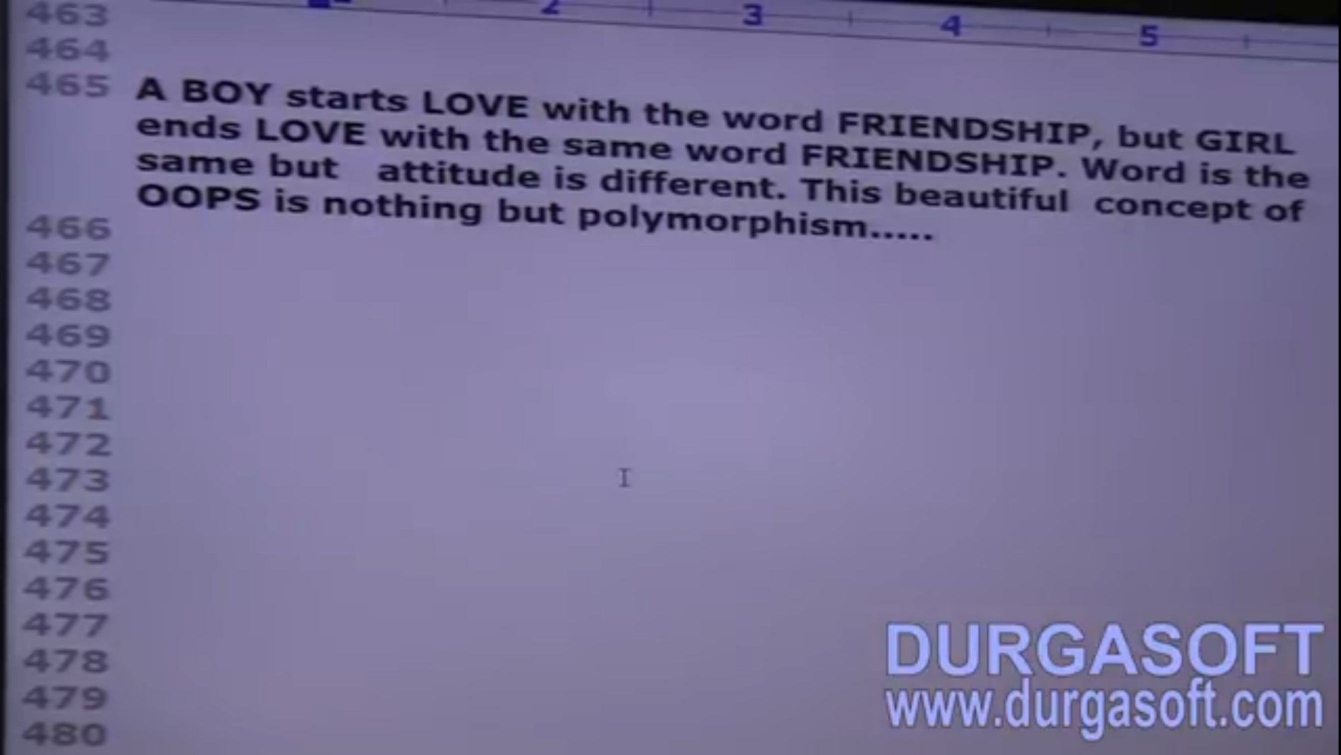


fig: 3 pillars of OOPs


Polymorphism
Static polymorphism
(or)
Compile-time polymorphism
(or)
early binding
Overloading
method hiding
Dynamic polymorphism
(or)
Runtime polymorphism
(or)
Late Binding
Overriding
DURGASOFT
www.durgasoft.com



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.....