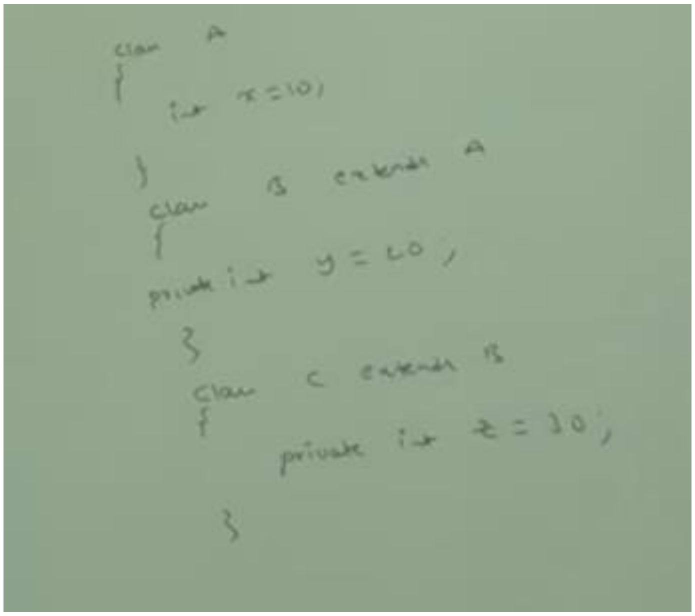
OOP

Object Oriented Programming

OCTOBER 21, 2016

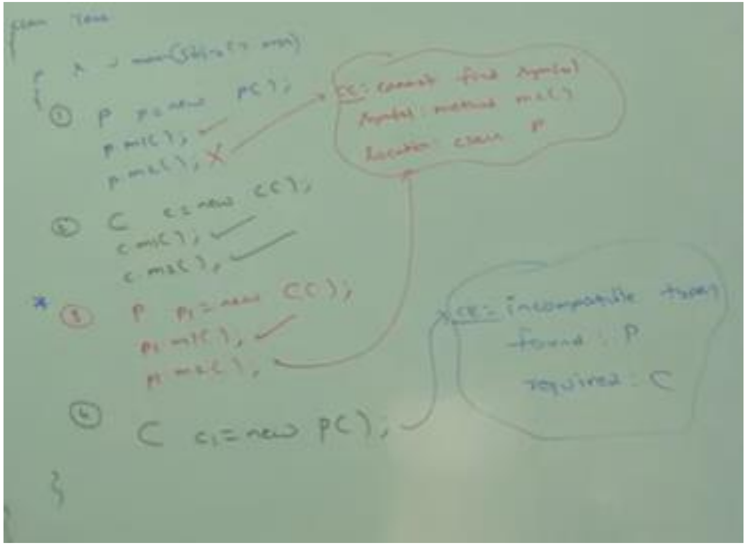
OOPS concepts

1. Data Hiding
    a. Hiding of data; outside users cannot access the internal data directly or internal data cannot be accessed outside.
    b. After successful authentication only data should be accessible.
    c. How to achieve data hiding programmatically?
        i. By declaring data variables as private we can achieve
    d. Advantages:
        i. Data Security
    e. Note: It is highly recommended to declare data members as private
2. Abstraction
    a. Hiding the internal implementation
    b. Advantages:
        i. Security, Since the internal implementation is not available to outside world
        ii. Easy enhancement
        iii. Easiness to use the system.
        iv. Maintainability
    c. How can we implement Abstraction?
        i. Interfaces: For full abstraction
        ii. Abstract class: Partial implementation
3. Encapsulation
    a. The process of binding data and corresponding methods into a single unit
    b. Every Java class is an example of Encapsulation
    c. Encapsulation = Data Hiding + Abstraction
    d. Advantages:
        i. Security, Since the internal implementation is not available to outside world
        ii. Easy enhancement
        iii. Easiness to use the system.
        iv. Maintainability
    e. Disadvantages
        i. Increases length of the code
        ii. Slowdowns the system performance
4. Tightly Encapsulated class
    a. Each and every member of the class is private such classes are Tightly Encapsulated classes
    b. It is irrespective of method access modifiers
    c. If the parent class is non-tightly encapsulated class then none of the child classes are Tightly Encapsulated



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

Note: All the above topics are mostly related to Security of data

5. IS-A Relationship
    a. It is also known as Inheritance
    b. How to implement IS-A relationship
        i. Using **extends** keyword
    c. Advantages:
        i. Reusability: Parent class members and methods are reused
    d. Disadvantages:
        i. 

```
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

    e. When to use Inheritance?
        i. To reduce redundancy of code
        ii. The most common methods which are applicable for any type of child, we have to define in Parent class
        iii. The specific methods which are applicable for a particular child, we have to define in child class

f. *Any type of java classes inherit the methods from Object class*

g. Total Java API is implemented based on Inheritance concept.

h. The most common methods which are applicable for any Java object are defined in Object class and hence every class in Java is a child class of Object either directly or indirectly so that Object class methods are default available to every Java class.

i. Due to this Object acts as root for all Java classes

j. Throwable class defines the most common methods which are required for every exception and error classes hence this class acts as root for Java Exception hierarchy.

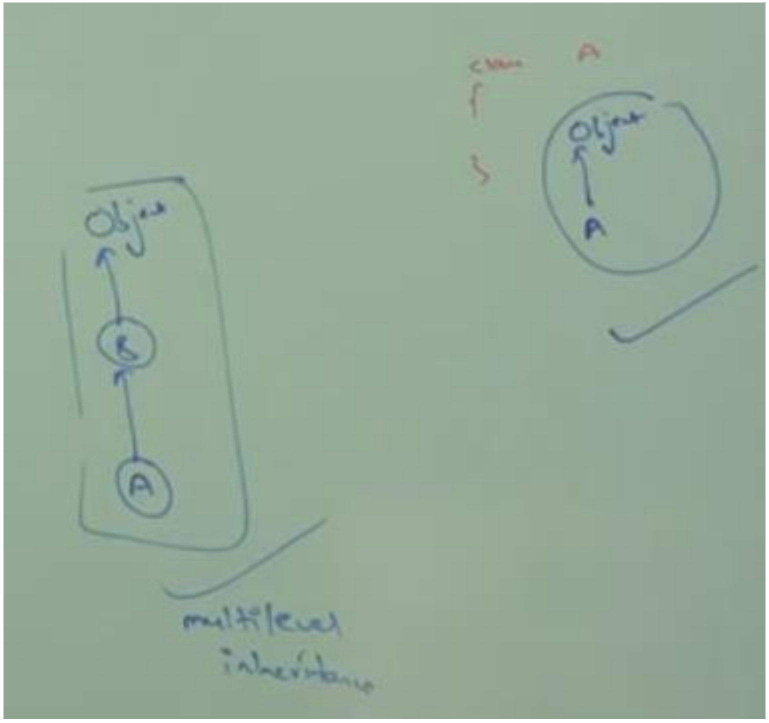
k. **Multiple Inheritance**

   i. A Java class can't extend more than one class at a time hence Java does not provide support for Multiple Inheritance w.r.t. classes.

l. Note:

   i. If our class does not extend any other class, then only our class is direct child class of Object.

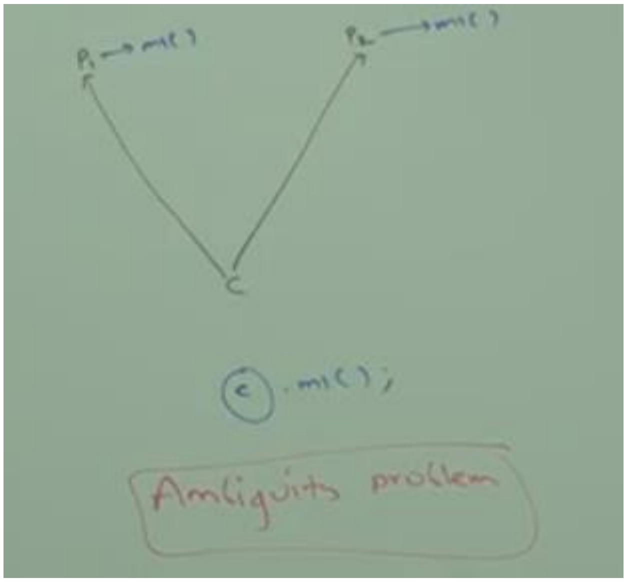
   ii. If our class extends any other class then our class is indirect child class of Object (Multi level Inheritance).



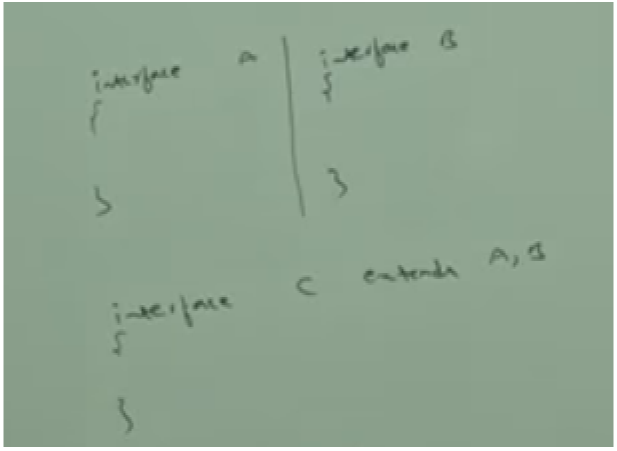
   iii. Either directly or indirectly Java does not provide for Inheritance w.r.t. classes

m. Why Java does not support Multiple Inheritance?

   i. There may be a chance of Ambiguity problem hence Java does not support Multiple Inheritance.

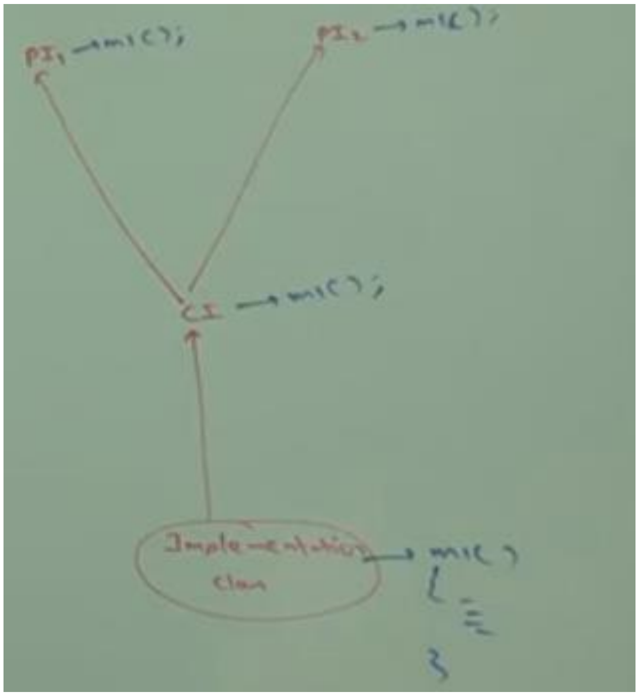


ii. Java provides Multiple Inheritance w.r.t. Interfaces



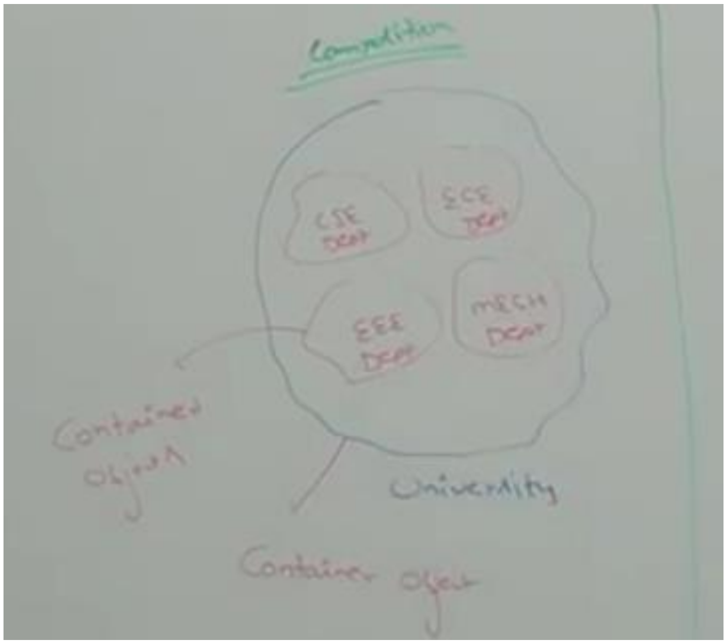
iii. Why Ambiguity problem does not happen in Interfaces?

1. ParentInterface1 contains m1() and ParentInterface2 also contains m1().
2. ChildInterface extends both PI1 and PI2
3. Implementation class had only one m1()
4. Even though Multiple method declarions are available, implementation is unique and hence there is no chance of Ambiguity problem in Interfaces.
5. Strictly, through Interface we wont get any Inheritance.



   iv. Cyclic Inheritance is not allowed in Java
      1. "class A extends class A"
      2. "class A extends B" and "class B extends A"

6. HAS-A Relationship
   a. Most generally used relationship is HAS-A relationship
   b. It is also known as Composition/Aggregation
   c. No specific keyword to implement HAS-A relationship. Most of the times we use **new keyword** to implement HAS-A relationship.
   d. Advantages:
      i. Write once and use any times: Code Reusability
   e. Difference between Composition and Aggregation
      i. Composition: Strong association between Objects
         1. Without existing Container Object, if there is no chance of existing Contained Objects then Container and Contained Objects are **Strongly** associated, this association is Composition
            a. University consists of several departments. Without existing University, there is no chance of existing departments, hence department and University Objects are strongly associated.



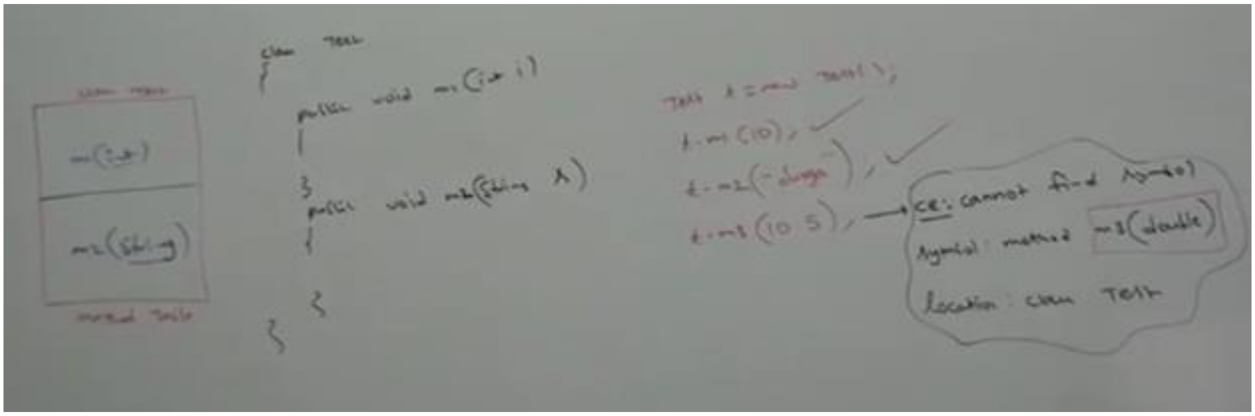
ii. Aggregation: Weak association between Objects
   1. Without existing Container Object if there is a chance of existing Contained Object then Container and Contained Objects are **weakly** associated, this association is Aggregation.
      a. Department consists of several professors, without existing departments, there may be a chance of existing professor Objects, hence department and Professor Objects are weakly associated.

iii. Note:
   1. In Composition objects are strongly associated whereas in Aggregation objects are weakly associated.
   2. In Composition container Object holds directly contained objects whereas in Aggregation Container Object holds just the reference of Contained Objects

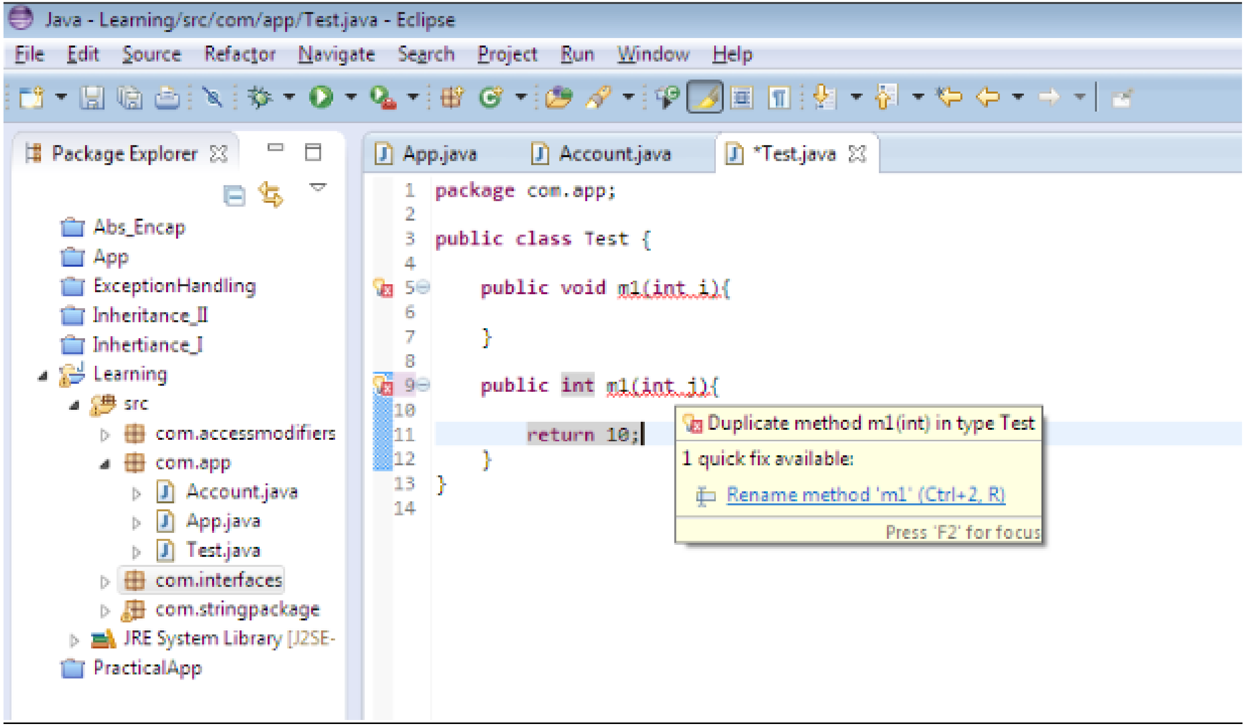
When to use IS-A and HAS-A relationship?

IS-A	HAS-A
If we want to implement total functionality of a class we use IS-A relationship.	If we want partial functionality of a class we use HAS-A relationship.

7. Method Signature
   a. In Java method signature consists of method names and argument types
      i. public static int m1(int i, float f)
      ii. Return type is not part of Method signature
   b. Compiler uses method signature to resolve method calls



We cannot use same method signature for two or more methods



```
package com.app;

public class Test {

    public void m1(int i){

    }

    public int m1(int i){

        return 10;
    }
}
```

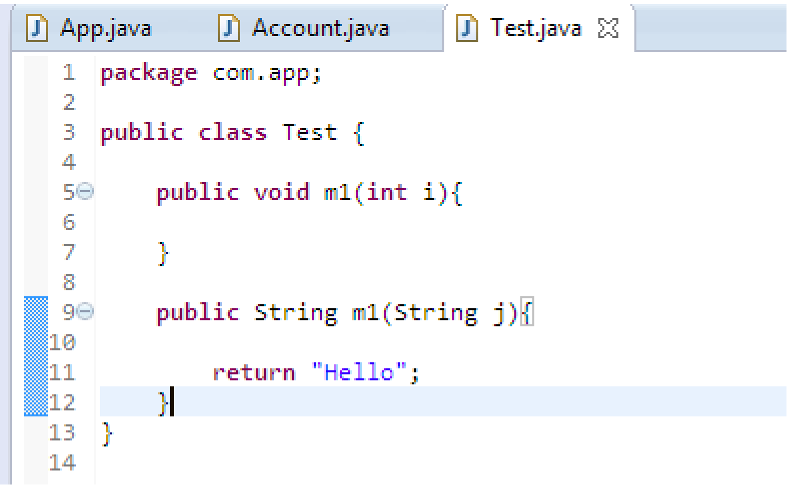
Duplicate method m1(int) in type Test

1 quick fix available:

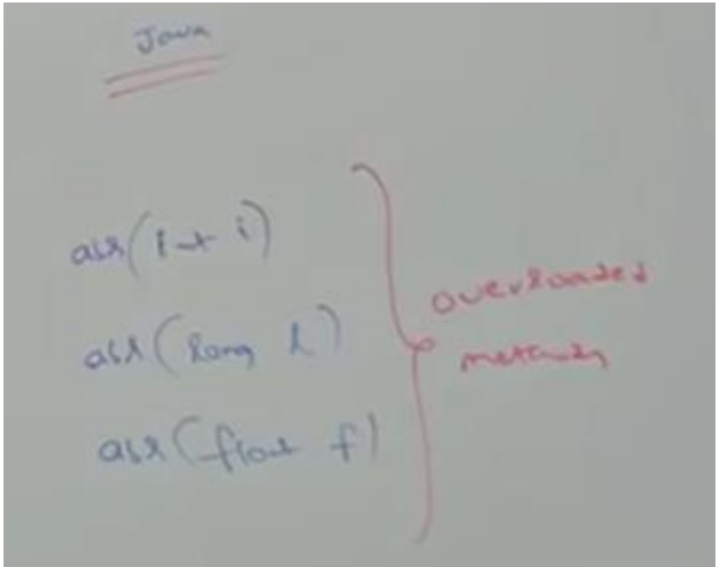
Rename method 'm1' (Ctrl+2, R)

Press 'F2' for focus

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



```
package com.app;

public class Test {

    public void m1(int i){

    }

    public String m1(String j){

        return "Hello";
    }
}

```



b. Like abs(int), abs(float), abs(long)

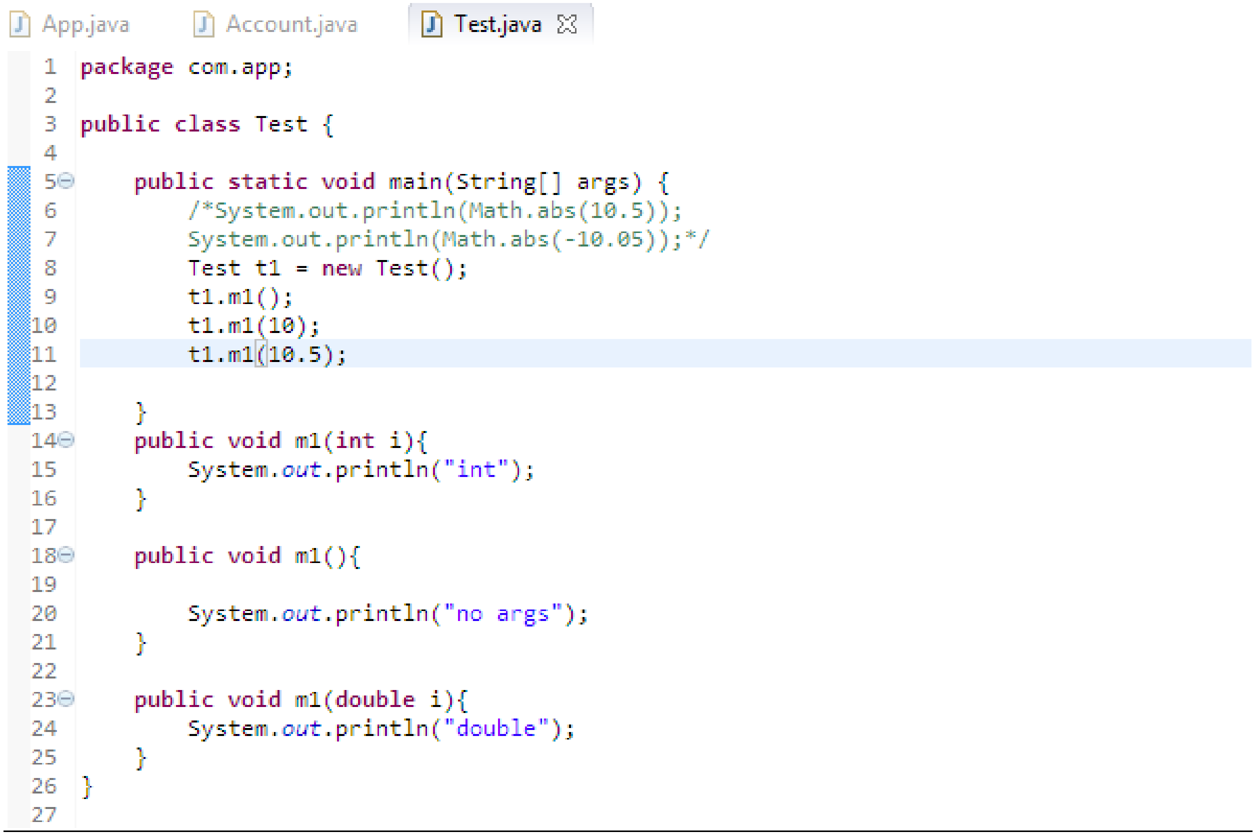
c. In Java we can declare multiple methods with same name but different argument types

d. Advantages:

   i. Reduces the complexity of the programming

   ii. Reduces the source lines of code

e. In overloading method resolution is taken care by compiler based on **reference type**. Overloading is also known as Compile-time Polymorphism/Static Polymorphism /Early Binding.

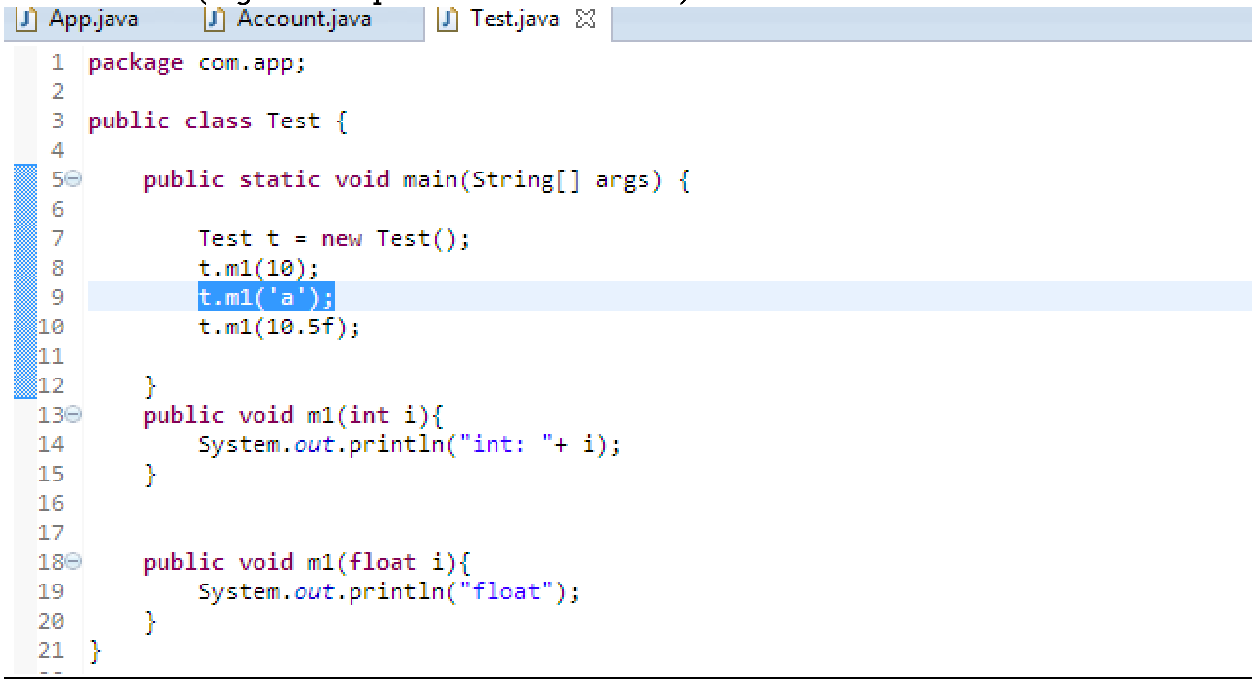


```java
package com.app;

public class Test {

    public static void main(String[] args) {
        /*System.out.println(Math.abs(10.5));
        System.out.println(Math.abs(-10.05));*/
        Test t1 = new Test();
        t1.m1();
        t1.m1(10);
        t1.m1(10.5);

    }
    public void m1(int i){
        System.out.println("int");
    }

    public void m1(){

        System.out.println("no args");
    }

    public void m1(double i){
        System.out.println("double");
    }
}

```

f. **Scenario 1: Automatic Promotion**:

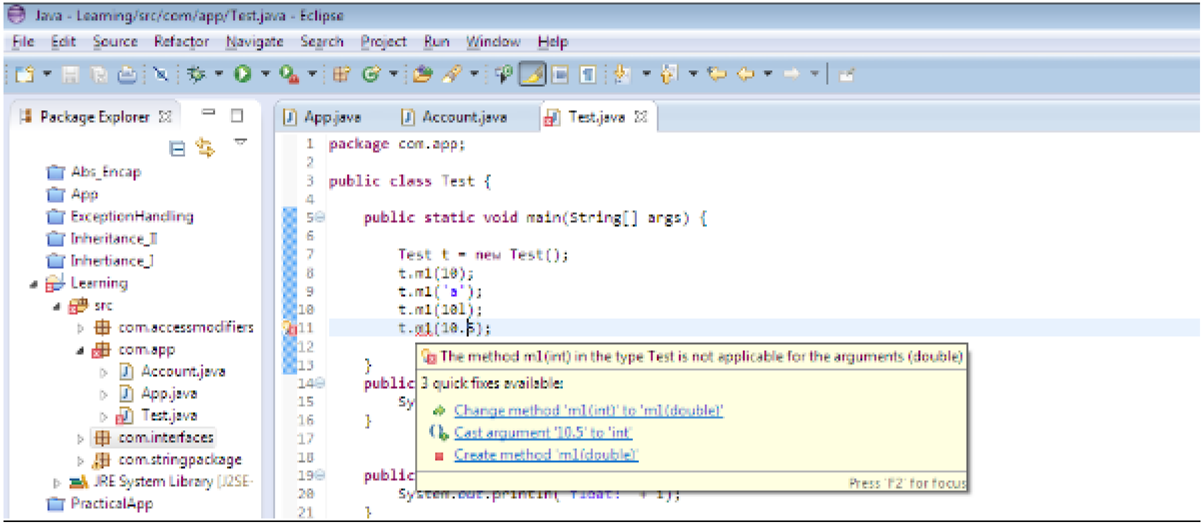
i. In overloading method resolution compiler always checks for exact match method.

ii. If exact match method is not found then automatic promotion (argument is promoted to next level) is occurred.



```java
package com.app;

public class Test {

    public static void main(String[] args) {

        Test t = new Test();
        t.m1(10);
        t.m1('a');
        t.m1(10.5f);

    }
    public void m1(int i){
        System.out.println("int: "+ i);
    }


    public void m1(float i){
        System.out.println("float");
    }
}
```

iii. This happens until all the methods are completed, after then it will throw compiler error



```java
package com.app;

public class Test {

    public static void main(String[] args) {

        Test t = new Test();
        t.m1(10);
        t.m1('a');
        t.m1(101);
        t.m1(10.5);

    }
    public
        Sy
    }


    public
        System.out.println("float: " + i);
    }
```

The method m1(int) in the type Test is not applicable for the arguments (double)

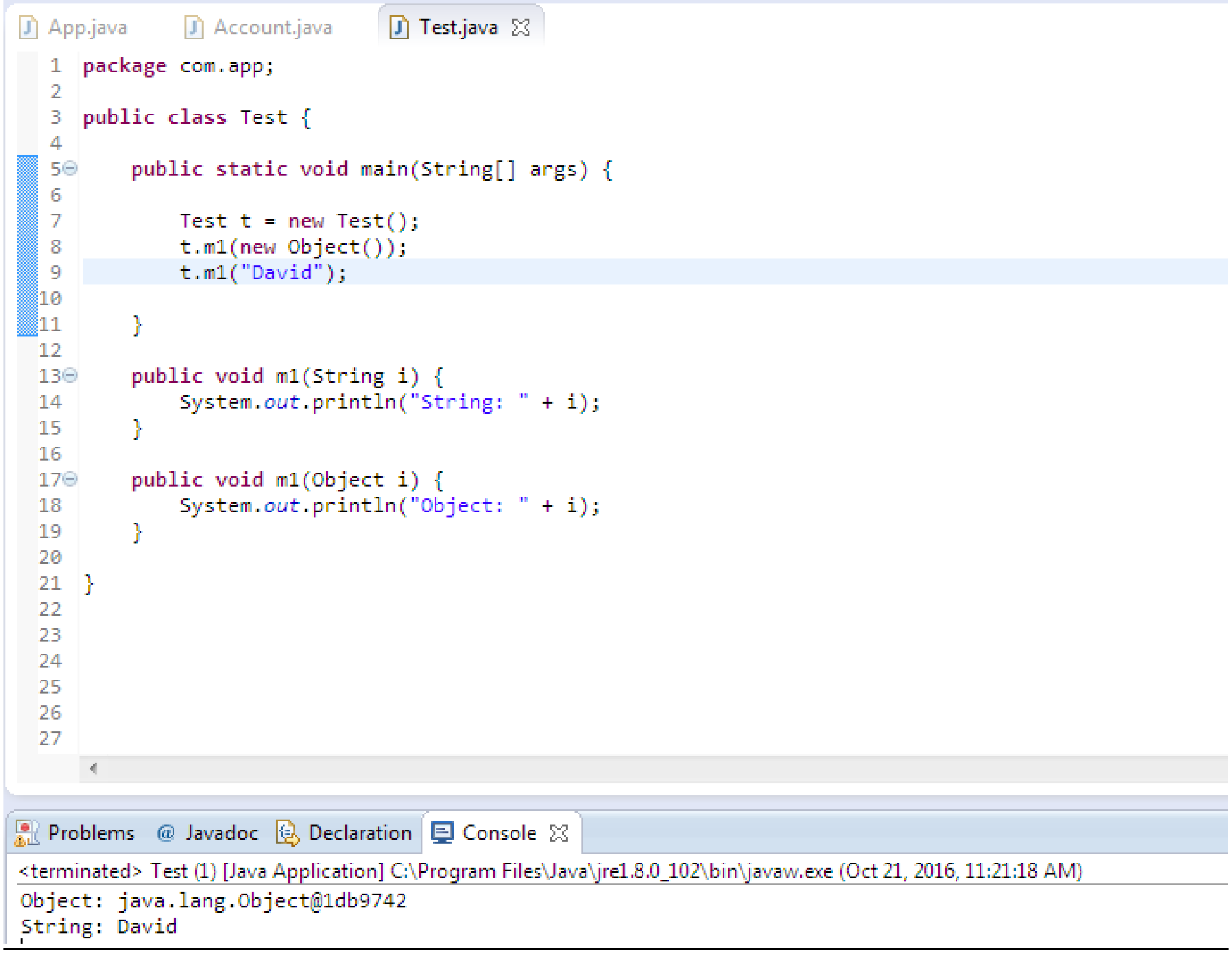
3 quick fixes available:

- Change method 'm1(int)' to 'm1(double)'
- Cast argument '10.5' to 'int'
- Create method 'm1(double)'

Press 'F2' for focus

g. **Scenario 2:** While resolving overloaded methods

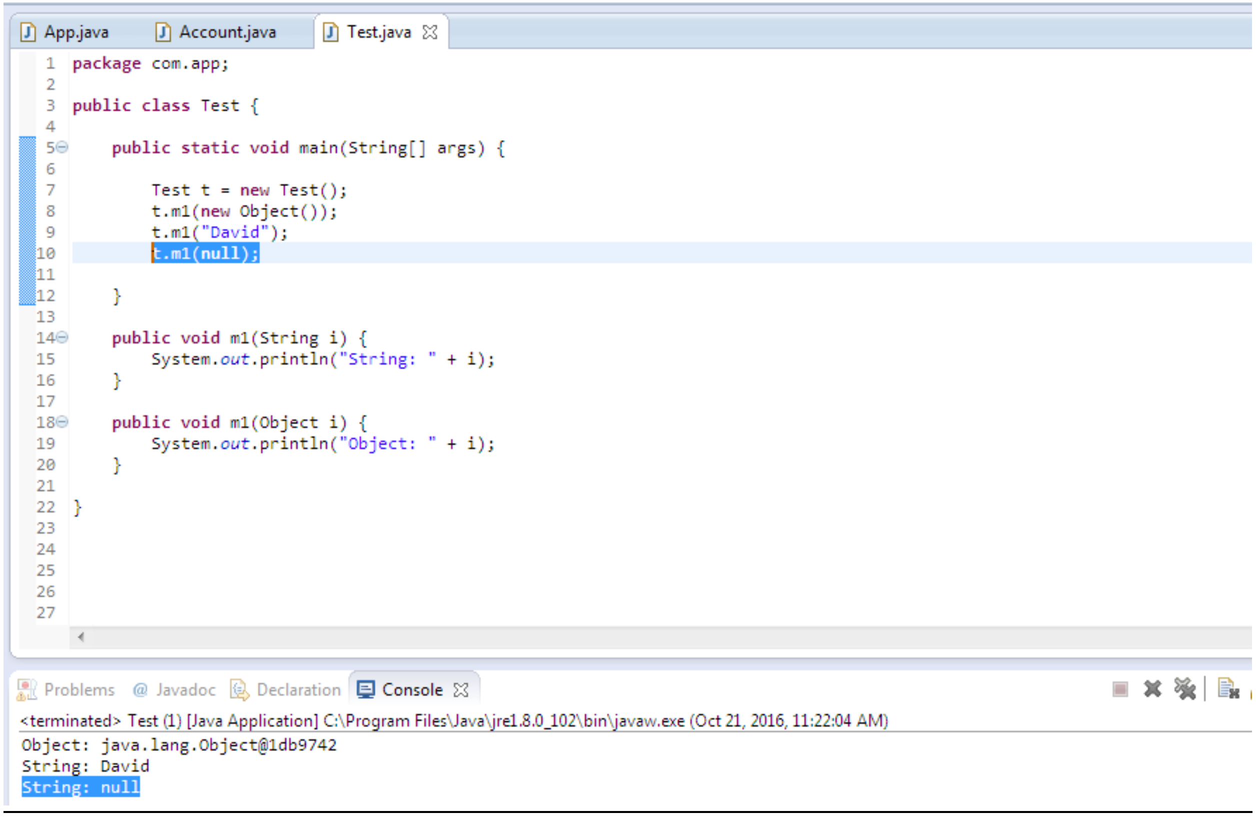
i. In the below scenario, the exact match method will be executed.



```java
package com.app;

public class Test {

    public static void main(String[] args) {

        Test t = new Test();
        t.m1(new Object());
        t.m1("David");

    }

    public void m1(String i) {
        System.out.println("String: " + i);
    }

    public void m1(Object i) {
        System.out.println("Object: " + i);
    }

}
```

```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Oct 21, 2016, 11:21:18 AM)
Object: java.lang.Object@1db9742
String: David
```

ii. In the below scenario, both Object and String arg methods are valid for argument type "null". Compiler will always give child type argument when compared with parent type argument.

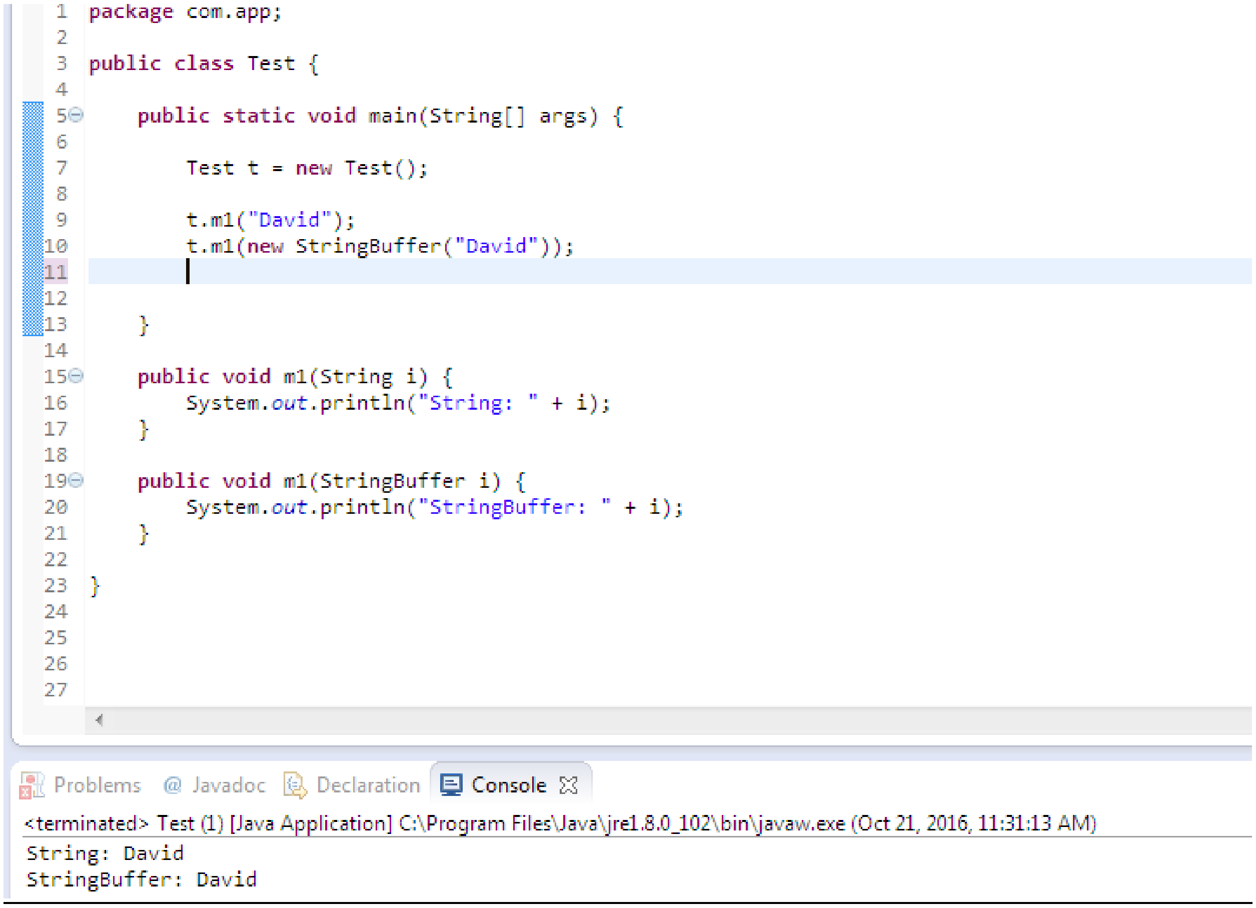


```java
package com.app;

public class Test {

    public static void main(String[] args) {

        Test t = new Test();
        t.m1(new Object());
        t.m1("David");
        t.m1(null);

    }

    public void m1(String i) {
        System.out.println("String: " + i);
    }

    public void m1(Object i) {
        System.out.println("Object: " + i);
    }

}
```

```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Oct 21, 2016, 11:22:04 AM)
Object: java.lang.Object@1db9742
String: David
String: null
```

h. **Scenario 3:**

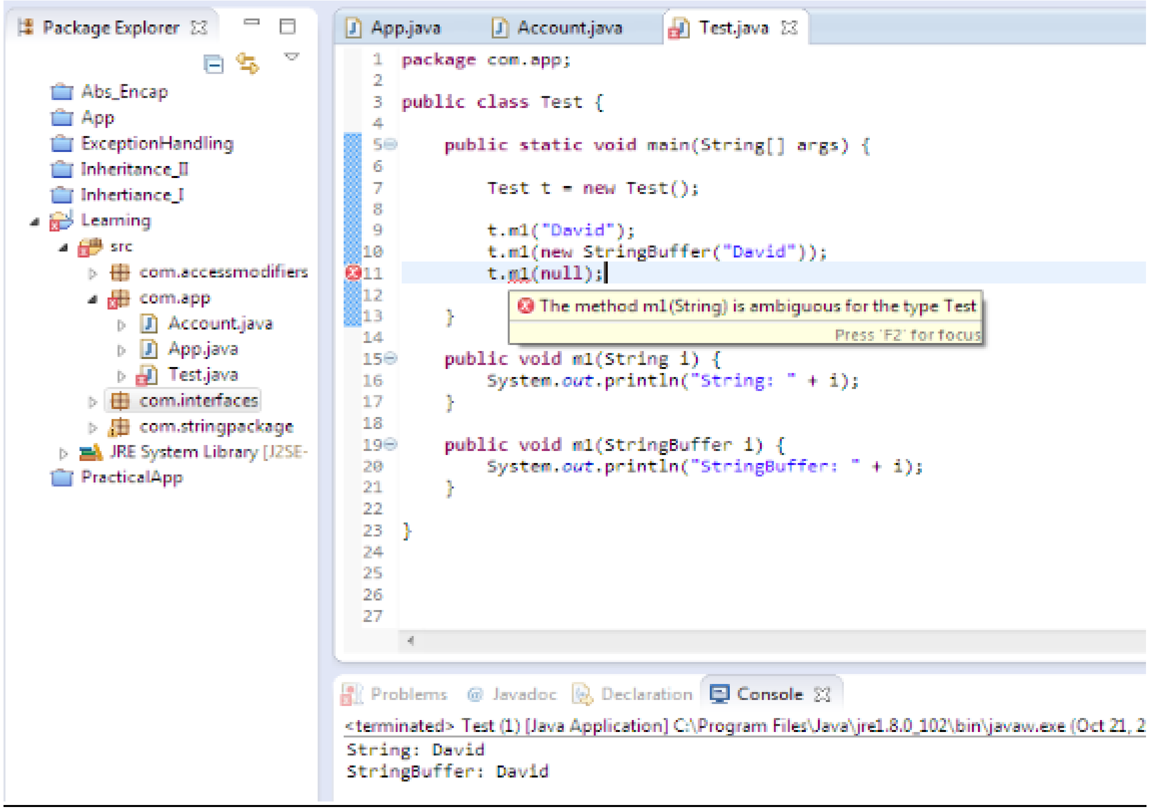
i. While method resolution always exact match method is executed



```java
package com.app;

public class Test {

    public static void main(String[] args) {

        Test t = new Test();

        t.m1("David");
        t.m1(new StringBuffer("David"));


    }

    public void m1(String i) {
        System.out.println("String: " + i);
    }

    public void m1(StringBuffer i) {
        System.out.println("StringBuffer: " + i);
    }

}
```

```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Oct 21, 2016, 11:31:13 AM)
String: David
StringBuffer: David
```

ii. In the below example, both String and StringBuffer are at same level, hence method ambiguity is occurred resulting in compile-time error



```java
package com.app;

public class Test {

    public static void main(String[] args) {

        Test t = new Test();

        t.m1("David");
        t.m1(new StringBuffer("David"));
        t.m1(null);

    }

    public void m1(String i) {
        System.out.println("String: " + i);
    }

    public void m1(StringBuffer i) {
        System.out.println("StringBuffer: " + i);
    }

}
```

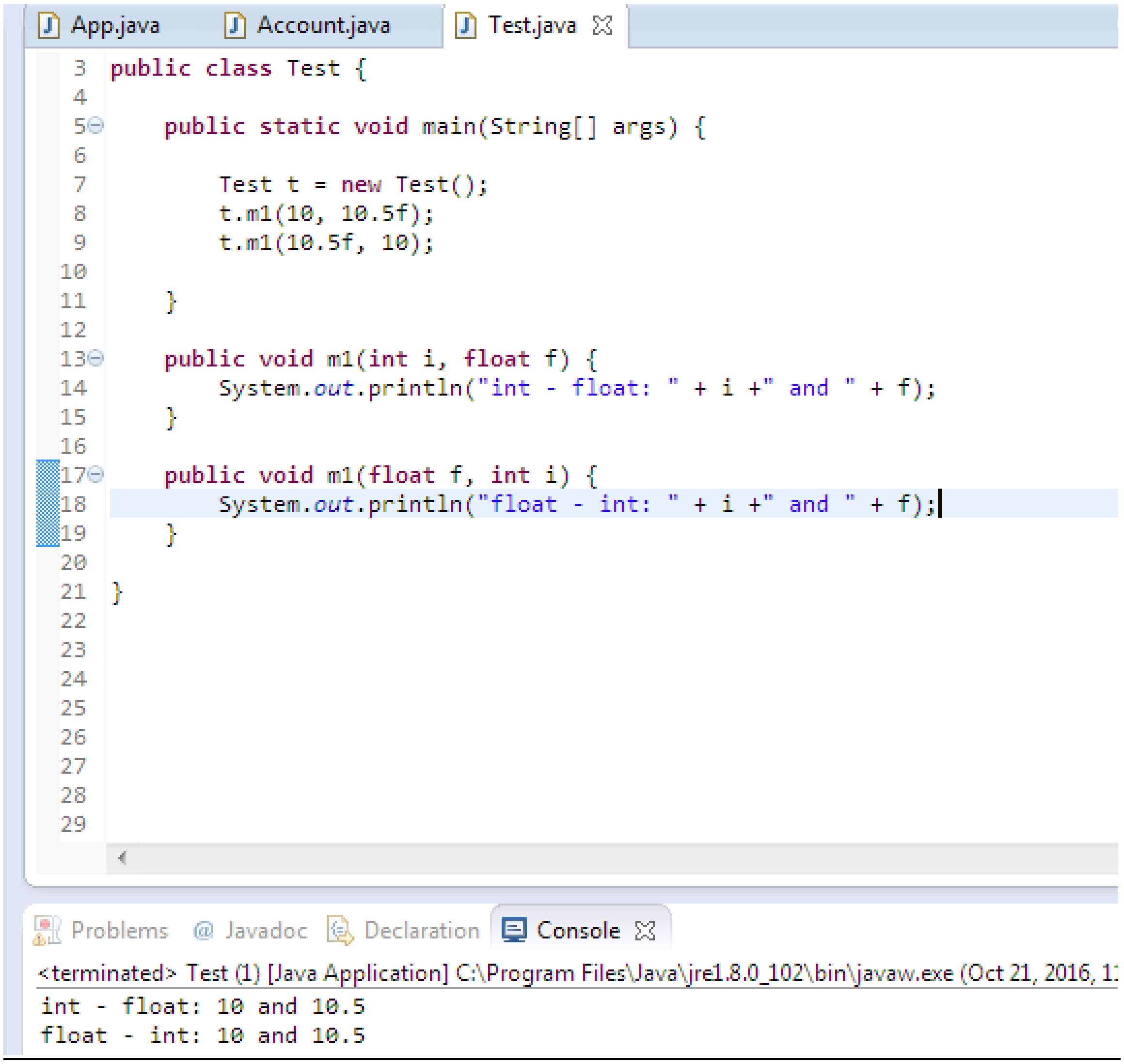
The method m1(String) is ambiguous for the type Test

Press 'F2' for focus

```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Oct 21, 2
String: David
StringBuffer: David
```

i. **Scenario 4:**

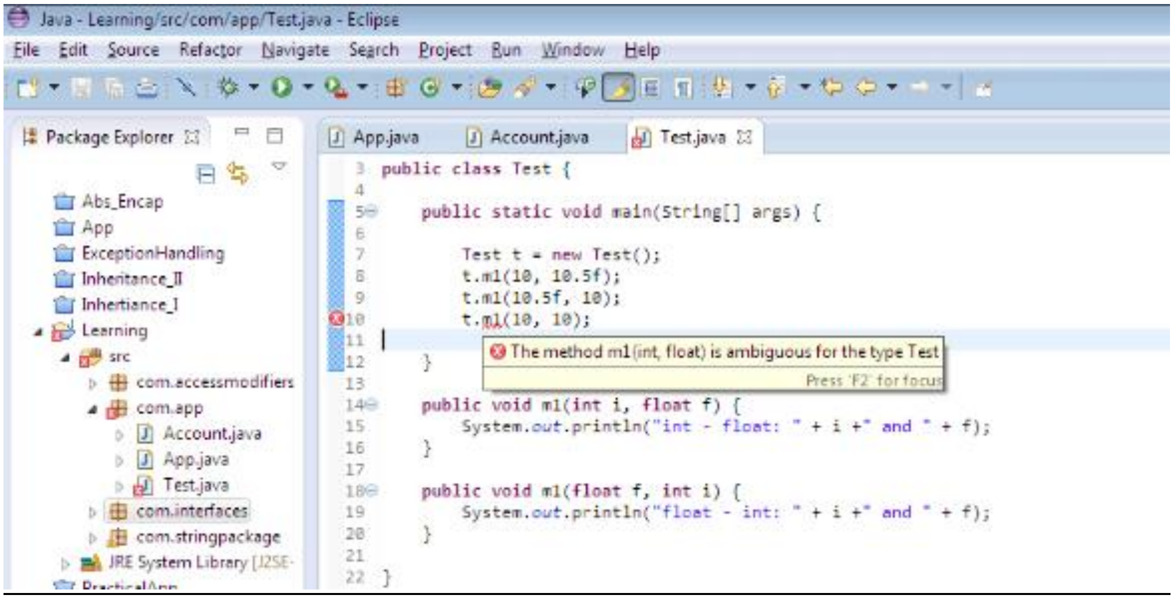
   i. While method resolution always exact match method is executed



```java
public class Test {

    public static void main(String[] args) {

        Test t = new Test();
        t.m1(10, 10.5f);
        t.m1(10.5f, 10);

    }

    public void m1(int i, float f) {
        System.out.println("int - float: " + i +" and " + f);
    }

    public void m1(float f, int i) {
        System.out.println("float - int: " + i +" and " + f);
    }

}
```

```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Oct 21, 2016, 1
int - float: 10 and 10.5
float - int: 10 and 10.5
```

ii. Ambiguity occurs since int value is a match method for both methods

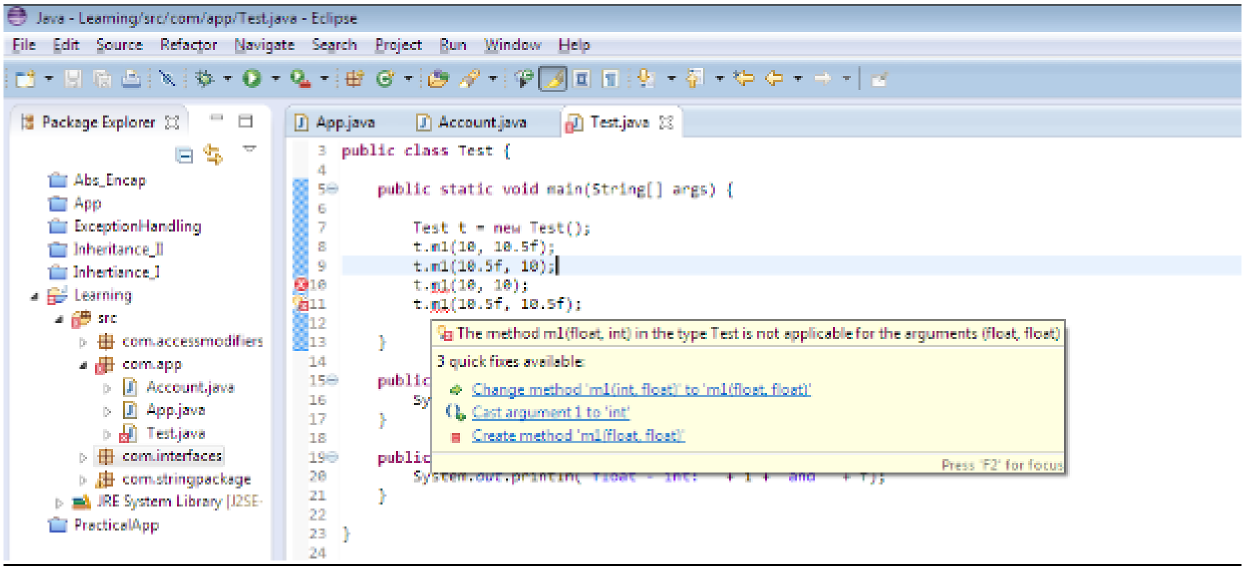


```
public class Test {

    public static void main(String[] args) {

        Test t = new Test();
        t.m1(10, 10.5f);
        t.m1(10.5f, 10);
        t.m1(10, 10);

    }

    public void m1(int i, float f) {
        System.out.println("int - float: " + i +" and " + f);
    }

    public void m1(float f, int i) {
        System.out.println("float - int: " + i +" and " + f);
    }
}
```

The method m1(int, float) is ambiguous for the type Test

Press 'F2' for focus

iii. Compile-time error occurs as method type (float,float) is not available.



```
public class Test {

    public static void main(String[] args) {

        Test t = new Test();
        t.m1(10, 10.5f);
        t.m1(10.5f, 10);
        t.m1(10, 10);
        t.m1(10.5f, 10.5f);

    }

    public
        Sy

    }

    public
        System.out.println("float - int: " + i +" and " + f);
    }
}
```

The method m1(float, int) in the type Test is not applicable for the arguments (float, float)

3 quick fixes available:

- Change method 'm1(int, float)' to 'm1(float, float)'
- Cast argument 1 to 'int'
- Create method 'm1(float, float)'

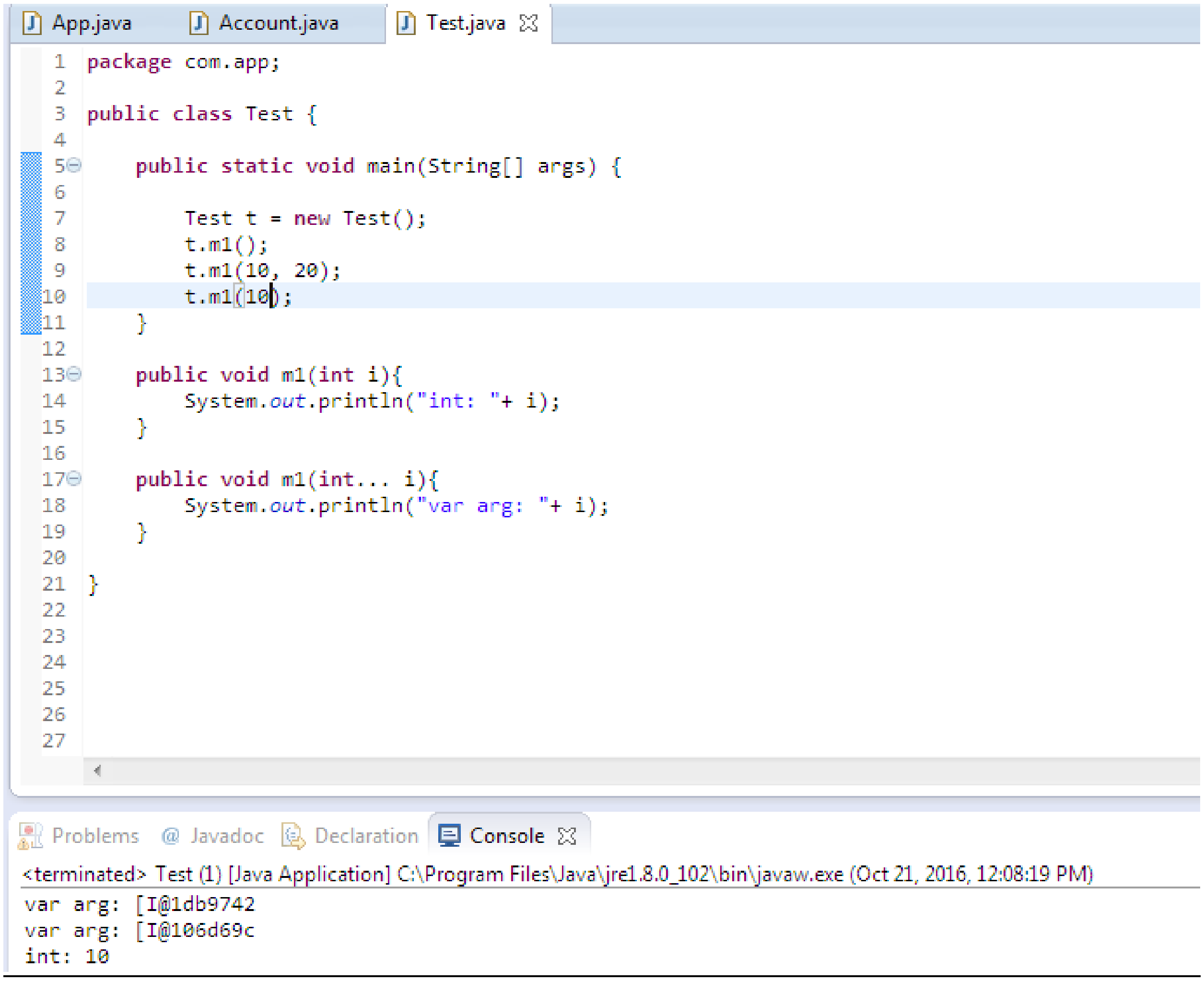
Press 'F2' for focus

j. **Scenario 5:**

i. While method resolution always exact match method is executed. In the below scenario, int method is execute since it is the old version method

ii. Var-arg method (int… x) will get the least priority. If no other method is matched then only var-arg method is executed.

iii. It is similar to **default** case inside **switch**



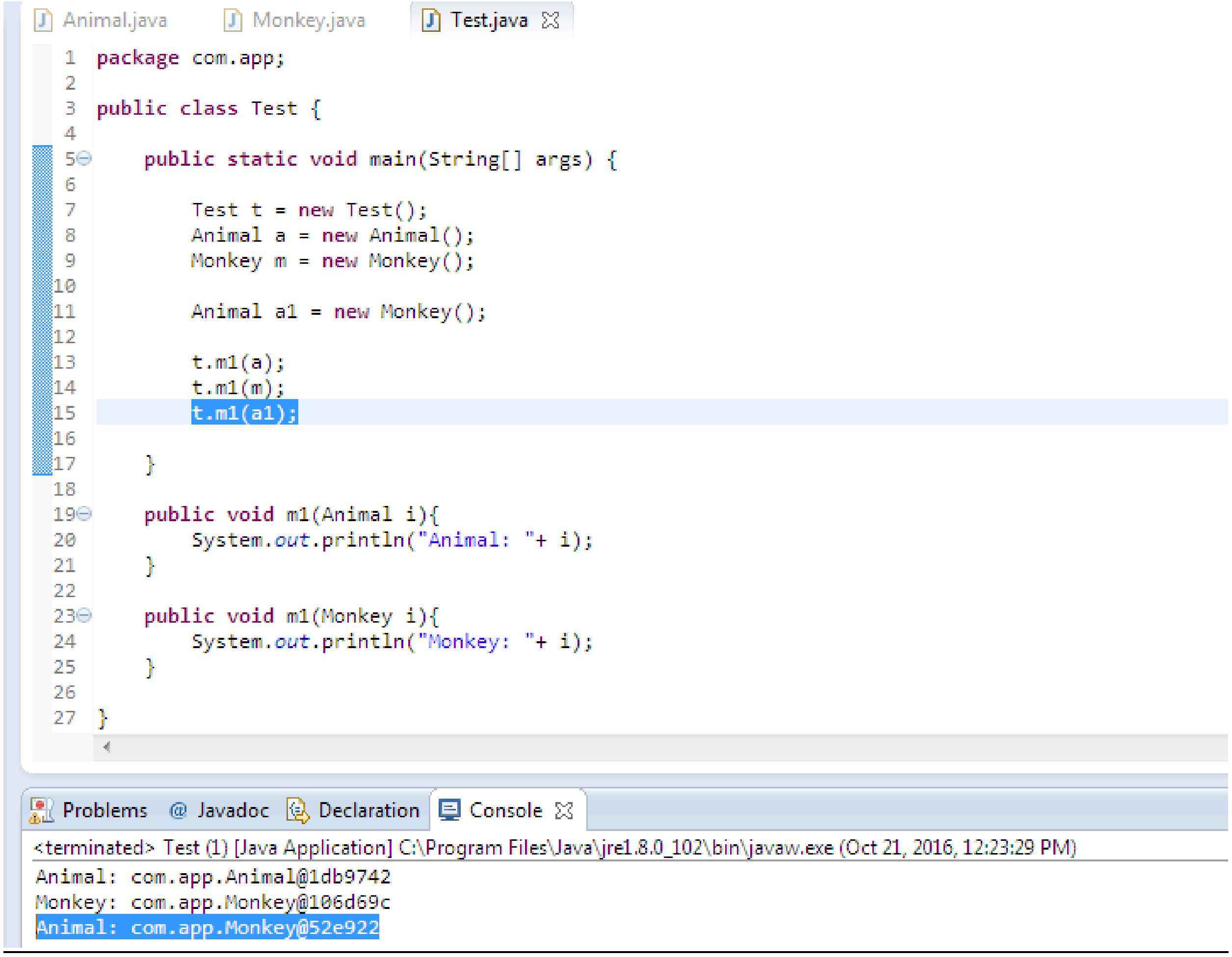
```java
package com.app;

public class Test {

    public static void main(String[] args) {

        Test t = new Test();
        t.m1();
        t.m1(10, 20);
        t.m1(10);
    }

    public void m1(int i){
        System.out.println("int: "+ i);
    }

    public void m1(int... i){
        System.out.println("var arg: "+ i);
    }

}
```

```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Oct 21, 2016, 12:08:19 PM)
var arg: [I@1db9742
var arg: [I@106d69c
int: 10
```

k. **Scenario 6:**

   i. Animal class and Monkey class. Monkey class is child class of Animal class.

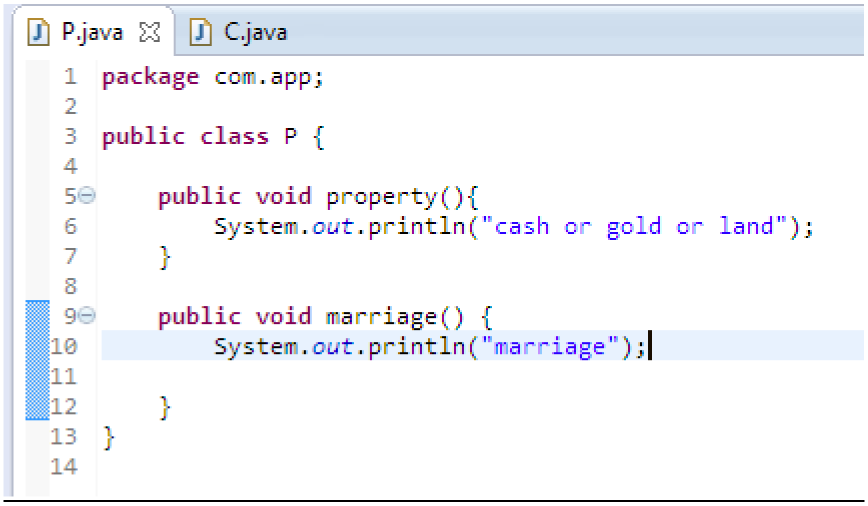
   ii. In overloading method resolution is taken lcare by compiler based on **reference type**, hence Animal method is executed.



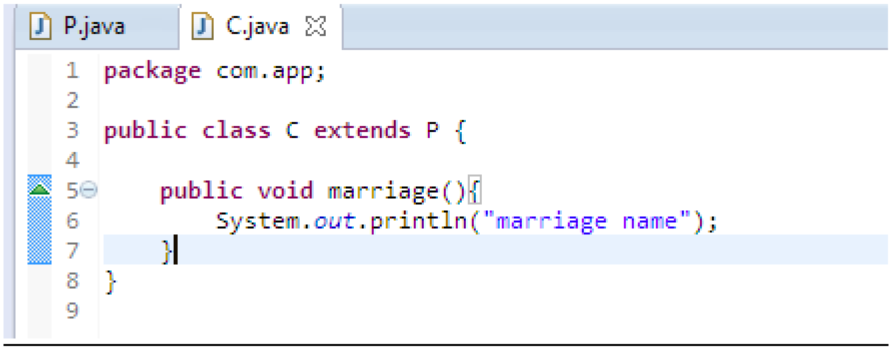
```java
package com.app;

public class Test {

    public static void main(String[] args) {

        Test t = new Test();
        Animal a = new Animal();
        Monkey m = new Monkey();

        Animal a1 = new Monkey();

        t.m1(a);
        t.m1(m);
        t.m1(a1);

    }

    public void m1(Animal i){
        System.out.println("Animal: "+ i);
    }

    public void m1(Monkey i){
        System.out.println("Monkey: "+ i);
    }

}
```

```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Oct 21, 2016, 12:23:29 PM)
Animal: com.app.Animal@1db9742
Monkey: com.app.Monkey@106d69c
Animal: com.app.Monkey@52e922
```

9. Overriding
   a. Parent class methods by default are available to child class through Inheritance.
   b. Child class can redefine the Parent class method in the Child class. This implementation is **overriding**.
   c. In overriding always the method signature should be same.
   d. Overridden method in Parent class and Overriding method in Child class

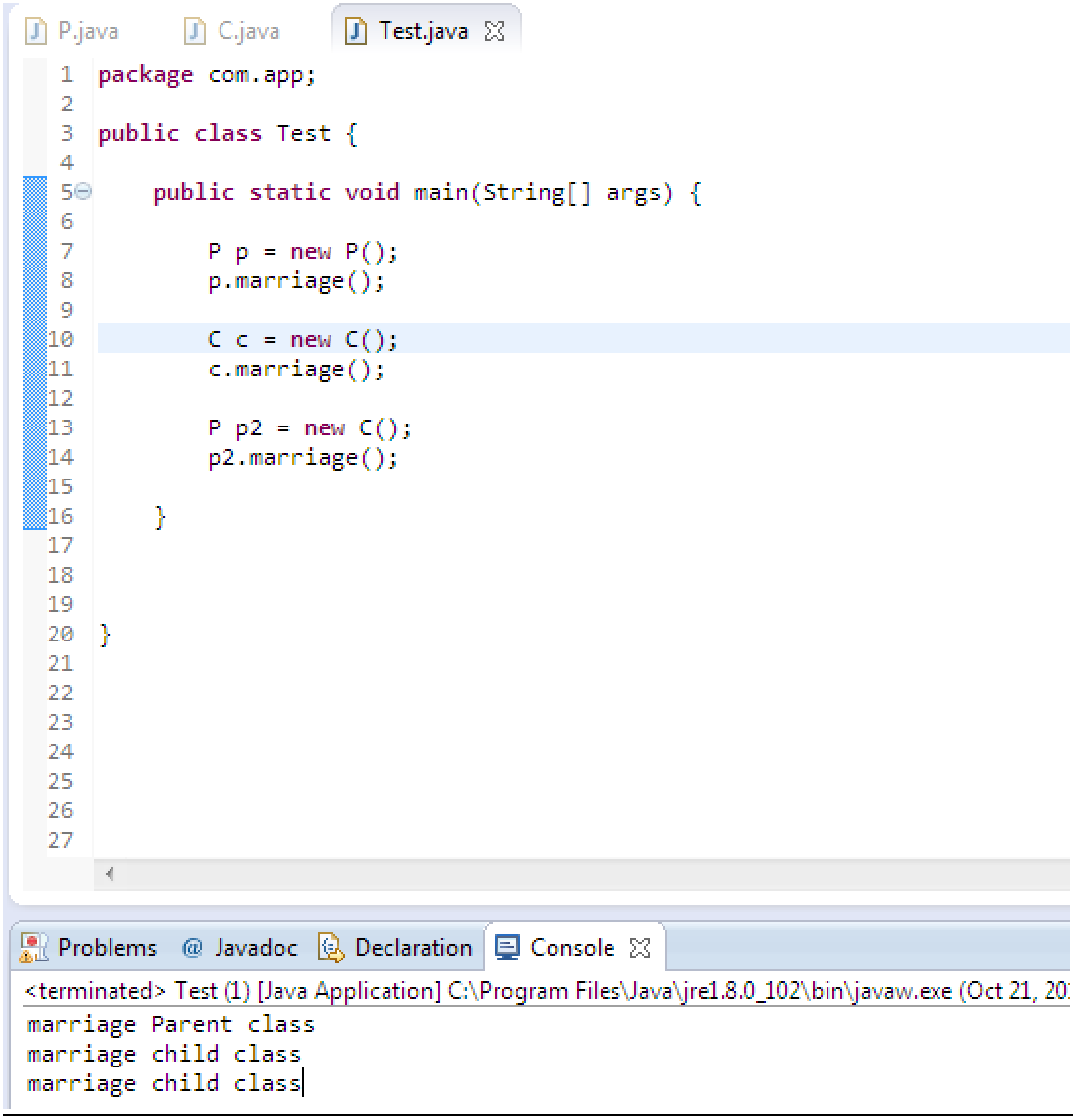


```java
package com.app;

public class P {

    public void property(){
        System.out.println("cash or gold or land");
    }

    public void marriage() {
        System.out.println("marriage");

    }
}

```



```java
package com.app;

public class C extends P {

    public void marriage(){
        System.out.println("marriage name");
    }
}

```

e. In method overriding method resolution is always executed by JVM based on runtime object.



```java
package com.app;

public class Test {

    public static void main(String[] args) {

        P p = new P();
        p.marriage();

        C c = new C();
        c.marriage();

        P p2 = new C();
        p2.marriage();

    }


}
```

```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Oct 21, 20
marriage Parent class
marriage child class
marriage child class
```

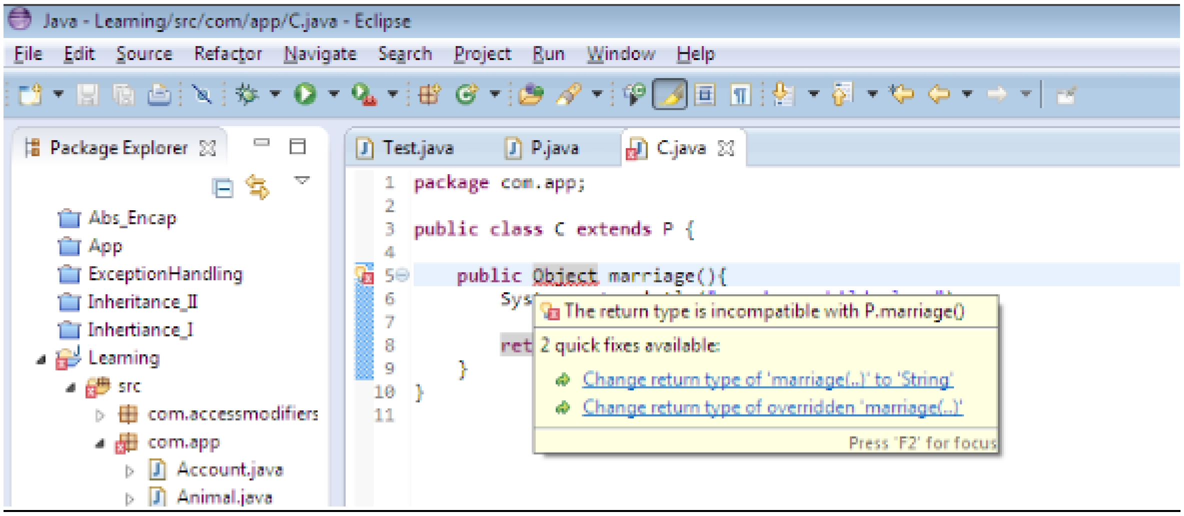
f. It is also known as Run-time polymorphism/Dynamic Binding/Late Binding.

g. Rules for Overriding

   i. **Return types**:

      1. Method names and argument types must be same i.e. method signatures must be same

      2. Return types should be same until 1.4 version. Since 1.5 version the Child class can have the return type as Parent class methods' return type or its child class. Such variables are called as co-variant variables.



```
package com.app;

public class C extends P {

    public Object marriage(){
        Sys
        
        ret
    }
}
```

The return type is incompatible with P.marriage()

2 quick fixes available:

- Change return type of 'marriage(..)' to 'String'
- Change return type of overridden 'marriage(..)'

Press 'F2' for focus

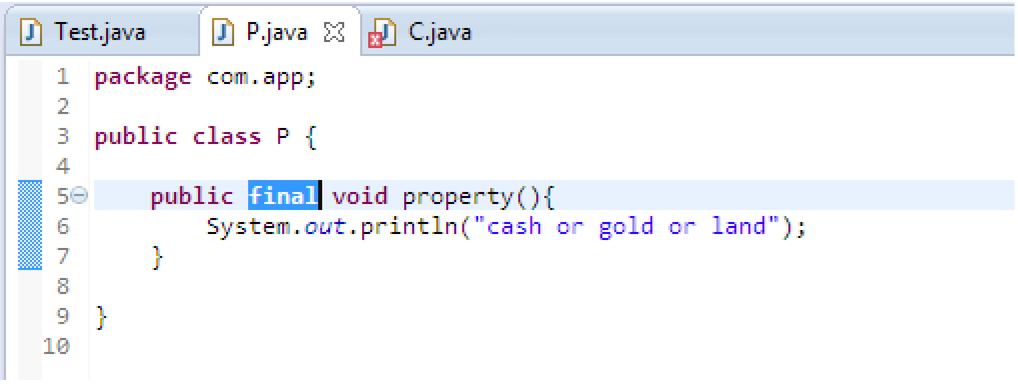
3. This co-variant return types are applicable only for Object type and not for Primitive types.

ii. **Access Modifiers**:

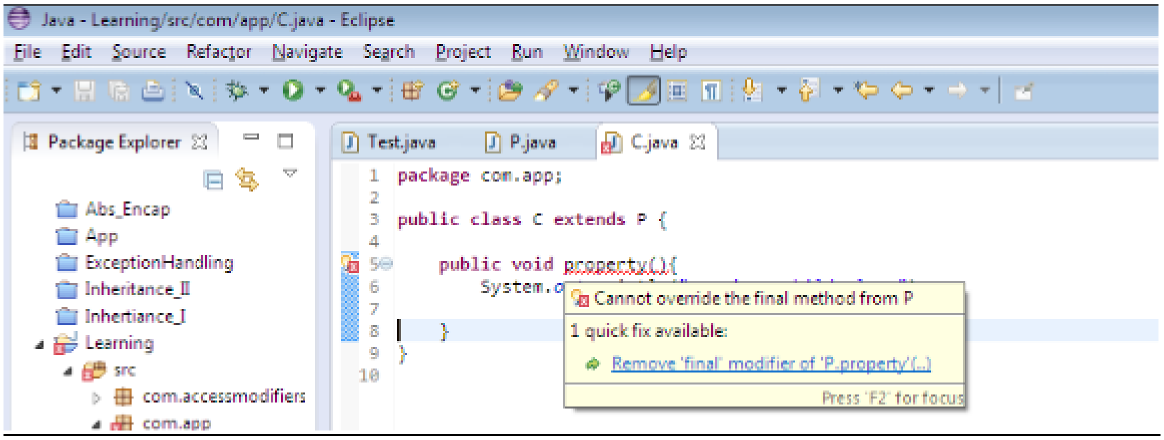
1. Parent class private methods not available to child and hence overriding concept does not applicable for private methods.
2. We can define exactly same private method in Child class it is valid, however it is not overriding.

iii. **Final methods**:

1. Any method declared as final cannot be overridden in the child class. If we try to override a compile time error is thrown.



```
package com.app;

public class P {

    public final void property(){
        System.out.println("cash or gold or land");
    }

}

```



```
package com.app;

public class C extends P {

    public void property(){
        System.o

    }
}

```

Cannot override the final method from P

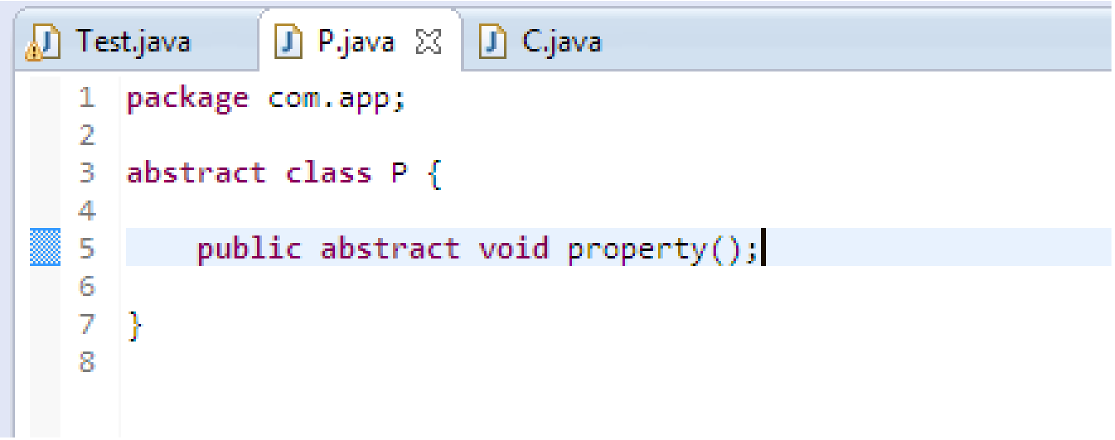
1 quick fix available:

Remove 'final' modifier of 'P.property'(..)

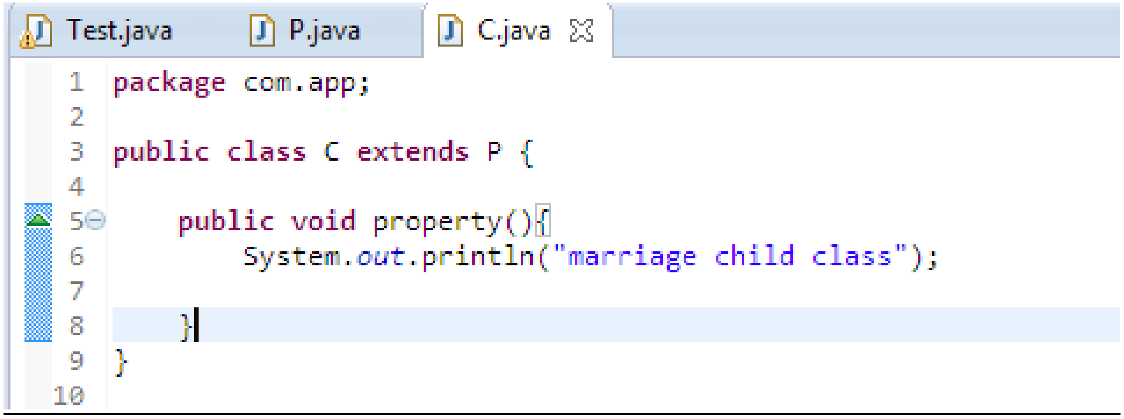
Press 'F2' for focus

iv. **Abstract Methods**:

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

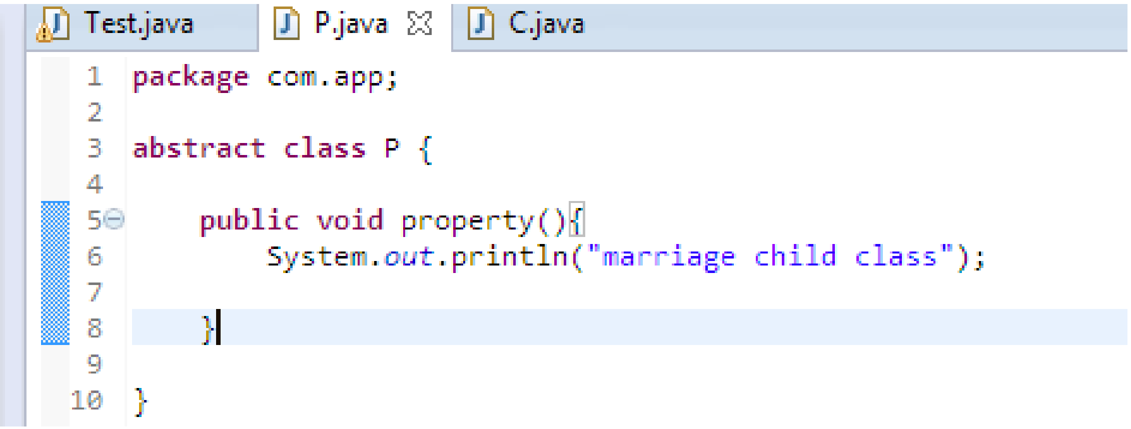


```
package com.app;

abstract class P {

    public abstract void property();

}

```

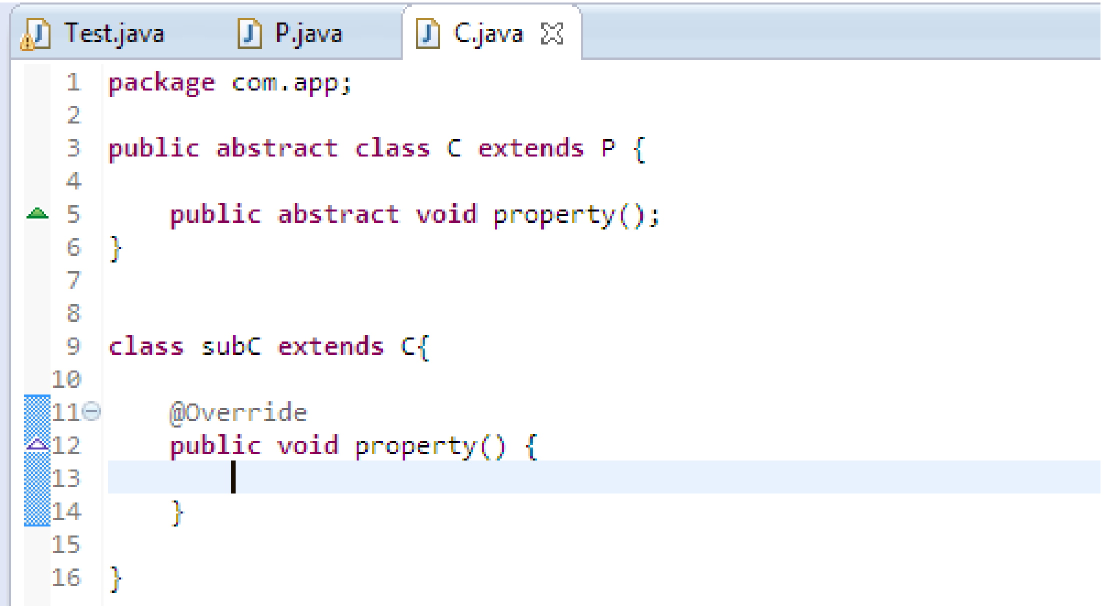


```
package com.app;

public class C extends P {

    public void property(){
        System.out.println("marriage child class");

    }
}

```

2. We can override non-abstract method as abstract. The main advantage of this approach is we can stop the availability of Parent method implementation to the next level child classes.

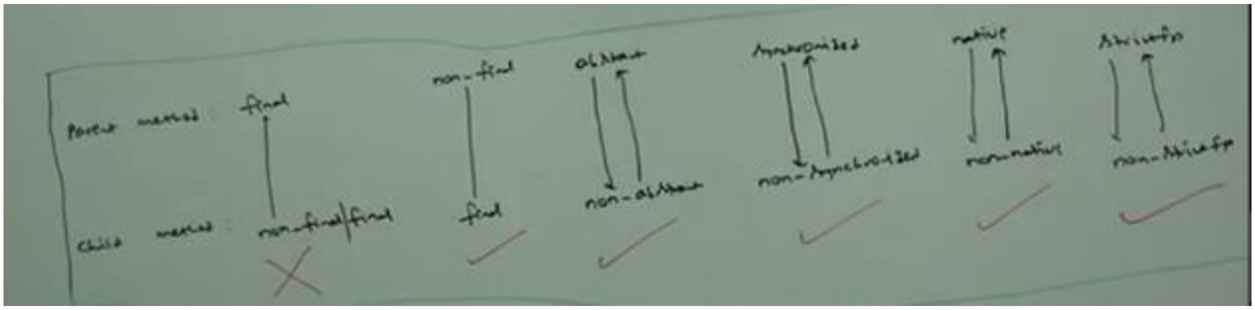


```java
package com.app;

abstract class P {

    public void property(){
        System.out.println("marriage child class");

    }

}
```



```java
package com.app;

public abstract class C extends P {

    public abstract void property();
}


class subC extends C{

    @Override
    public void property() {

    }

}
```

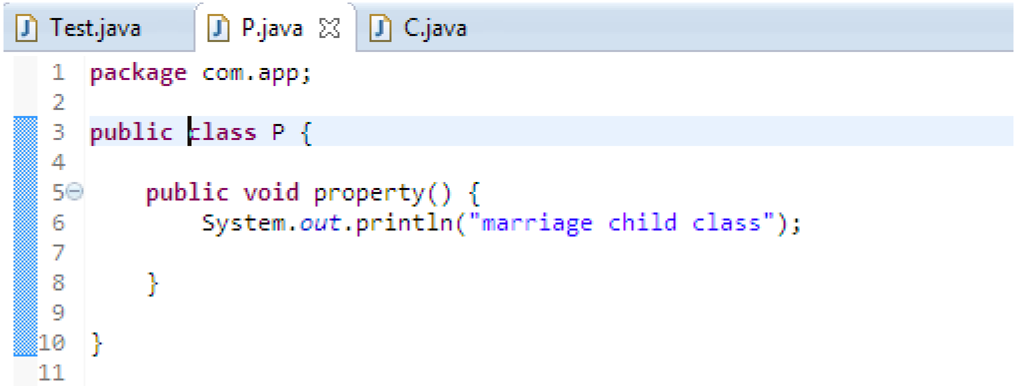
v. **Synchronized**: It does not affect the implementation of overriding.
vi. **Native**: It does not affect the implementation of overriding.
vii. **Strictfp**: It does not affect the implementation of overriding.



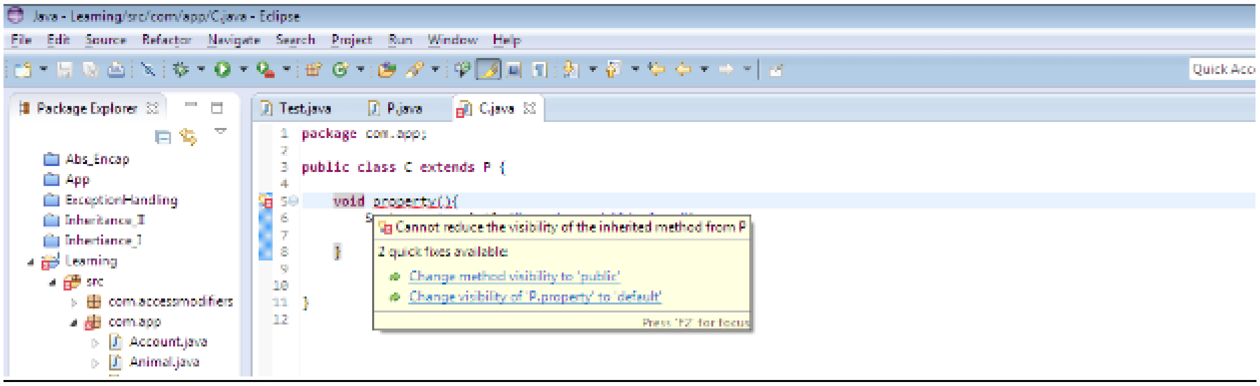
viii. **Access Privilege**:

1. While overriding we cannot weaken the scope of the modifiers, however we can increase the scope of access modifiers

a. Overriding should not weaken the scope since it will affect the implementation



```java
package com.app;

public class P {

    public void property() {
        System.out.println("marriage child class");

    }

}
```



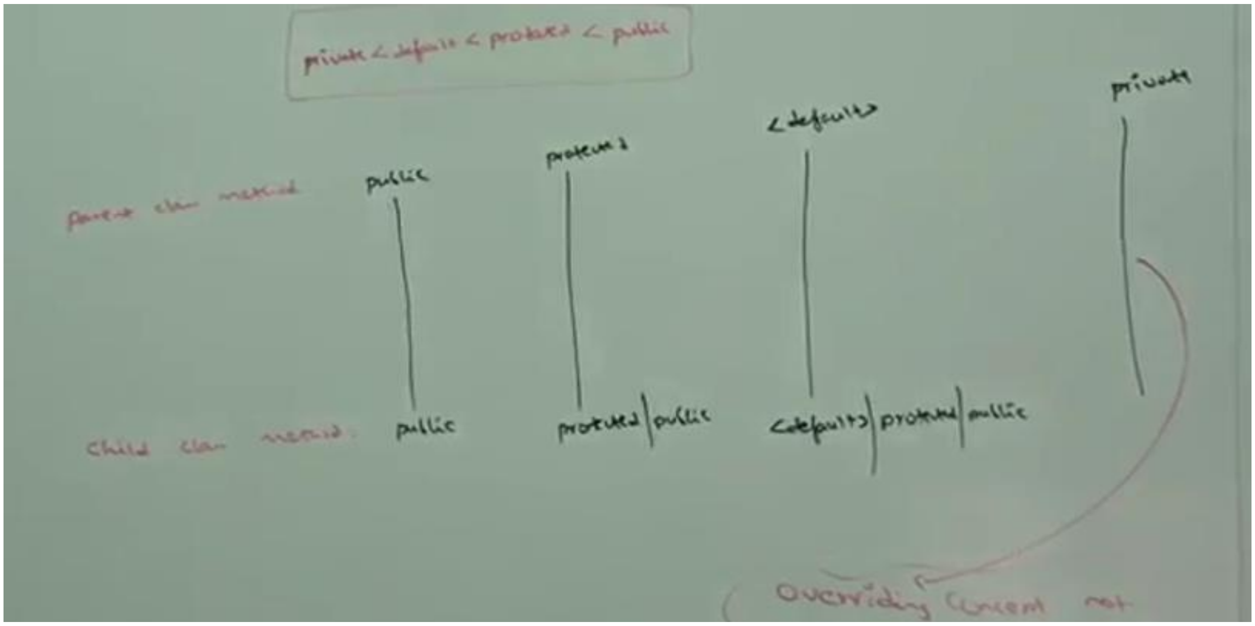
```java
package com.app;

public class C extends P {

    void property(){
        S

    }

}
```

Cannot reduce the visibility of the inherited method from P

2 quick fixes available

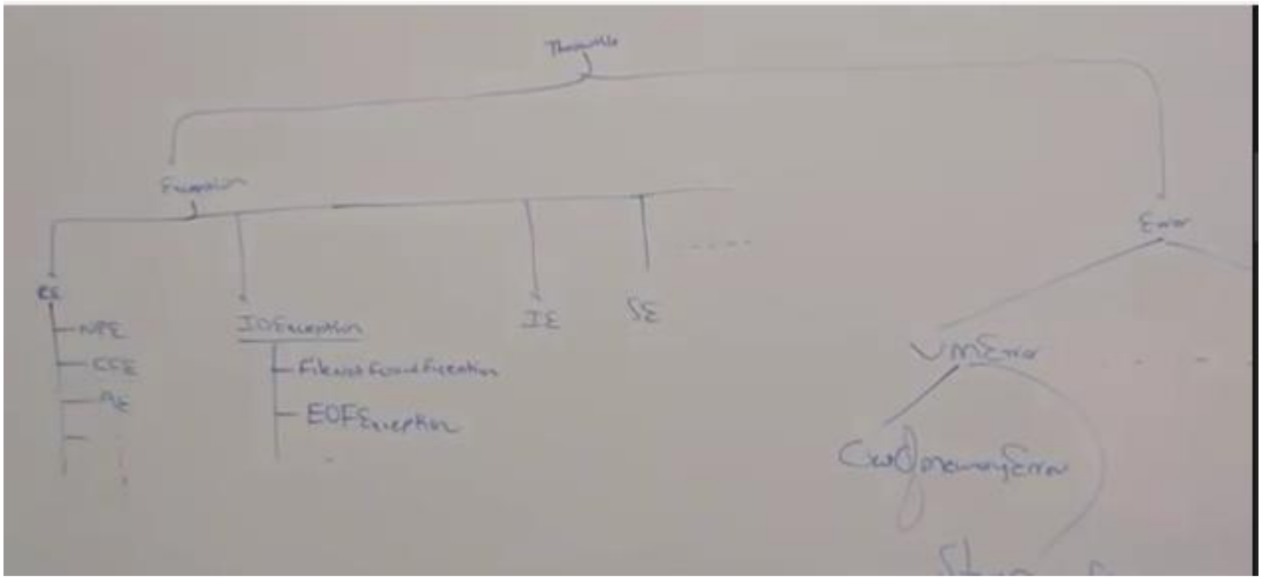
- Change method visibility to 'public'
- Change visibility of 'P.property' to 'default'

Press 'F2' for focus



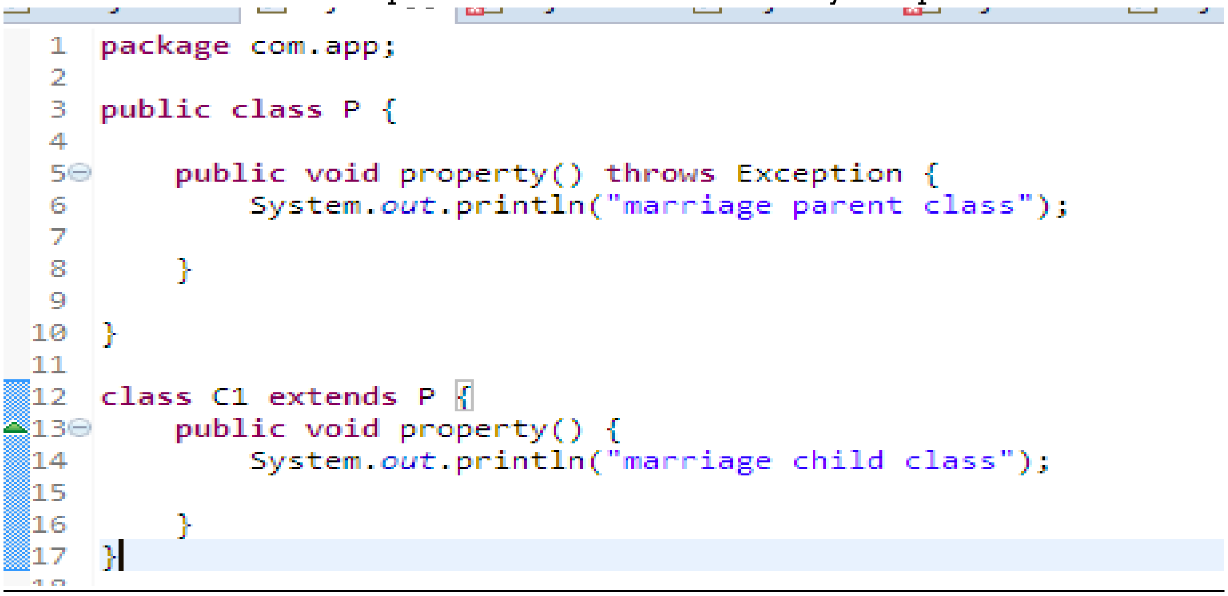
ix. **Exception**:

1. Run-time exceptions and Error are un-checked exception, all other are checked exception



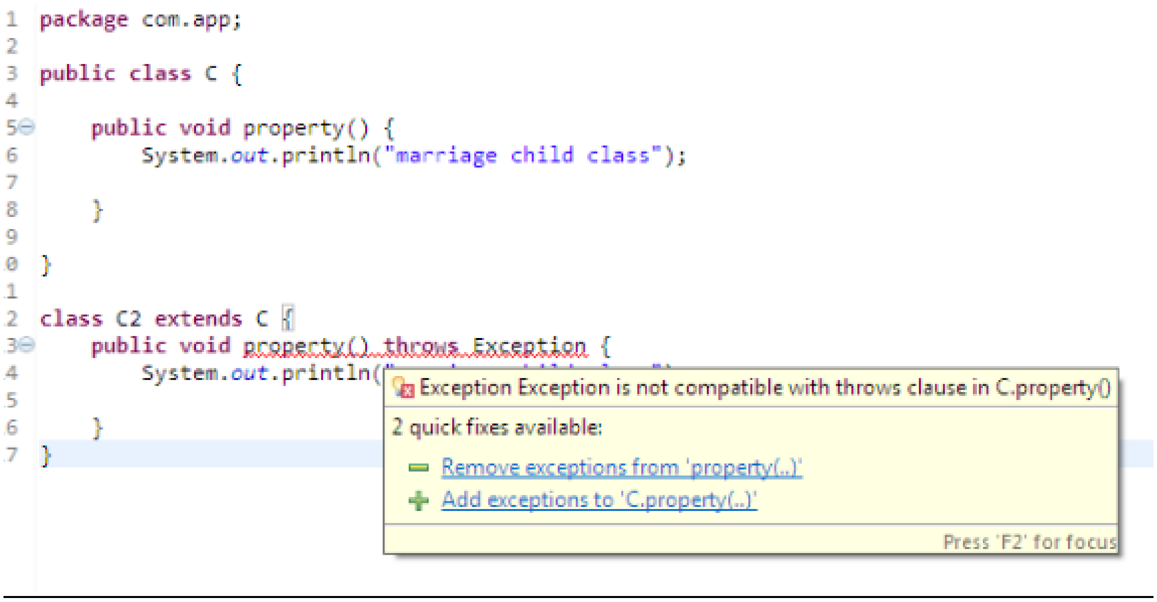
2. Scenario 1: If child class method throws any checked exception then the parent class should throw the same exception or its parent class exception.

   a. Child class method does not throw any exception, hence parent class method can have any Exception.



```
package com.app;

public class P {

    public void property() throws Exception {
        System.out.println("marriage parent class");

    }

}

class C1 extends P {
    public void property() {
        System.out.println("marriage child class");

    }
}
```

   b. Child class throws Exception, parent class method must throw Exception or its parent class Exception, otherwise compile error occurs.



```
package com.app;

public class C {

    public void property() {
        System.out.println("marriage child class");

    }

}

class C2 extends C {
    public void property() throws Exception {
        System.out.println("

    }
}
```

Exception Exception is not compatible with throws clause in C.property()

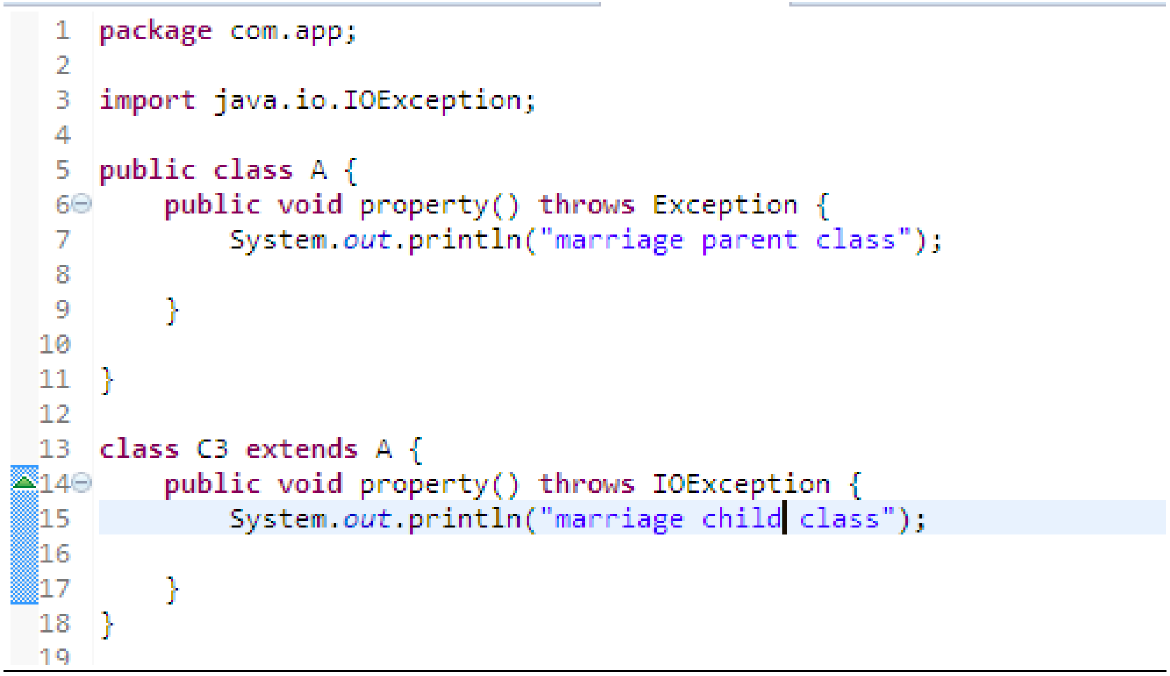
2 quick fixes available:

Remove exceptions from 'property(..)'

Add exceptions to 'C.property(..)'

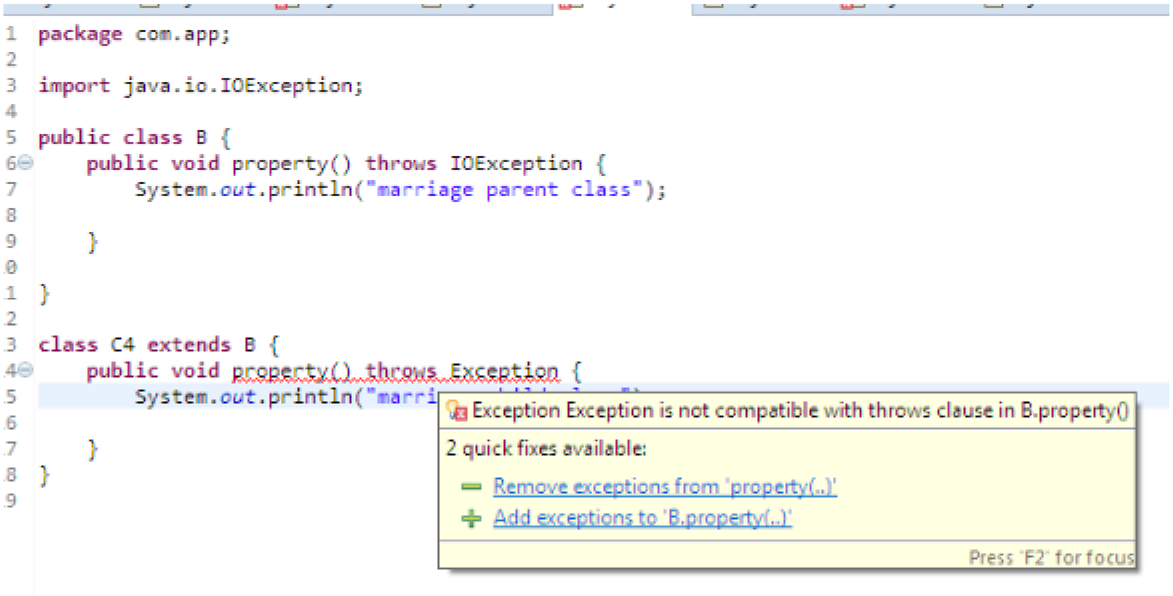
Press 'F2' for focus

c. Parent class method exception is parent class of child class method exception.



```
package com.app;

import java.io.IOException;

public class A {
    public void property() throws Exception {
        System.out.println("marriage parent class");

    }

}

class C3 extends A {
    public void property() throws IOException {
        System.out.println("marriage child class");

    }
}
```

d. Child class throws Exception, parent class method must throw Exception or its parent class Exception, otherwise compile error occurs.



```
package com.app;

import java.io.IOException;

public class B {
    public void property() throws IOException {
        System.out.println("marriage parent class");

    }

}

class C4 extends B {
    public void property() throws Exception {
        System.out.println("marri

    }
}
```

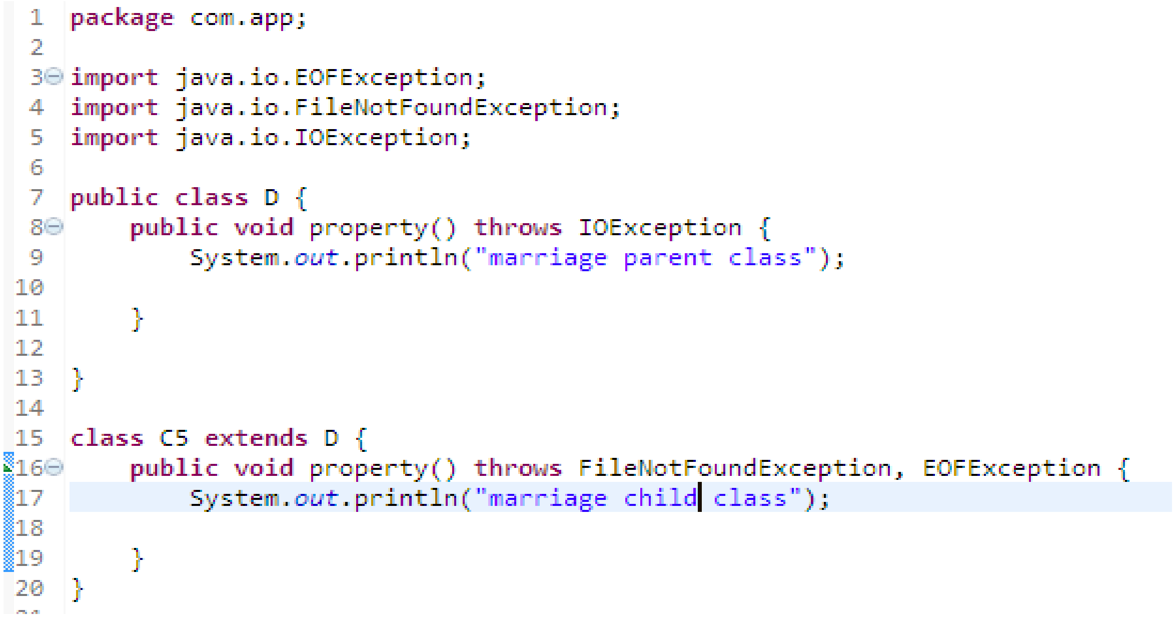
Exception Exception is not compatible with throws clause in B.property()

2 quick fixes available:

- Remove exceptions from 'property(..)'
- Add exceptions to 'B.property(..)'

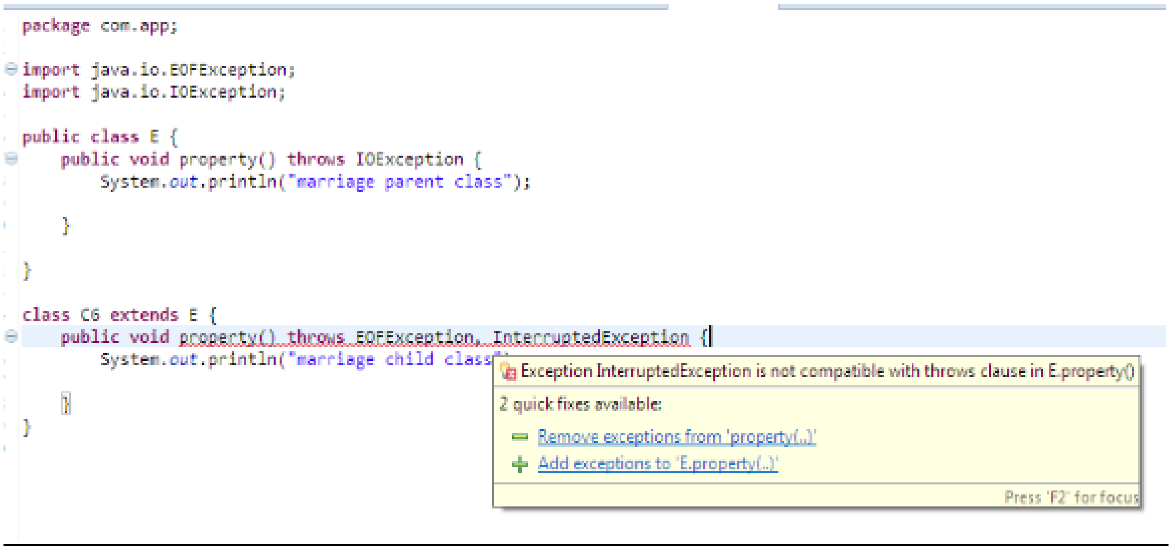
Press 'F2' for focus

e. Parent class method exception is parent class of child class method exception.



```
package com.app;

import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class D {
    public void property() throws IOException {
        System.out.println("marriage parent class");

    }

}

class C5 extends D {
    public void property() throws FileNotFoundException, EOFException {
        System.out.println("marriage child class");

    }
}
```

f. Child class throws Exception, parent class method must throw Exception or its parent class Exception, otherwise compile error occurs. As Interrupted Exception or its parent class Exception is not thrown in parent class method.



```
package com.app;

import java.io.EOFException;
import java.io.IOException;

public class E {
    public void property() throws IOException {
        System.out.println("marriage parent class");

    }

}

class C6 extends E {
    public void property() throws EOFException, InterruptedException {
        System.out.println("marriage child class"

    }
}
```

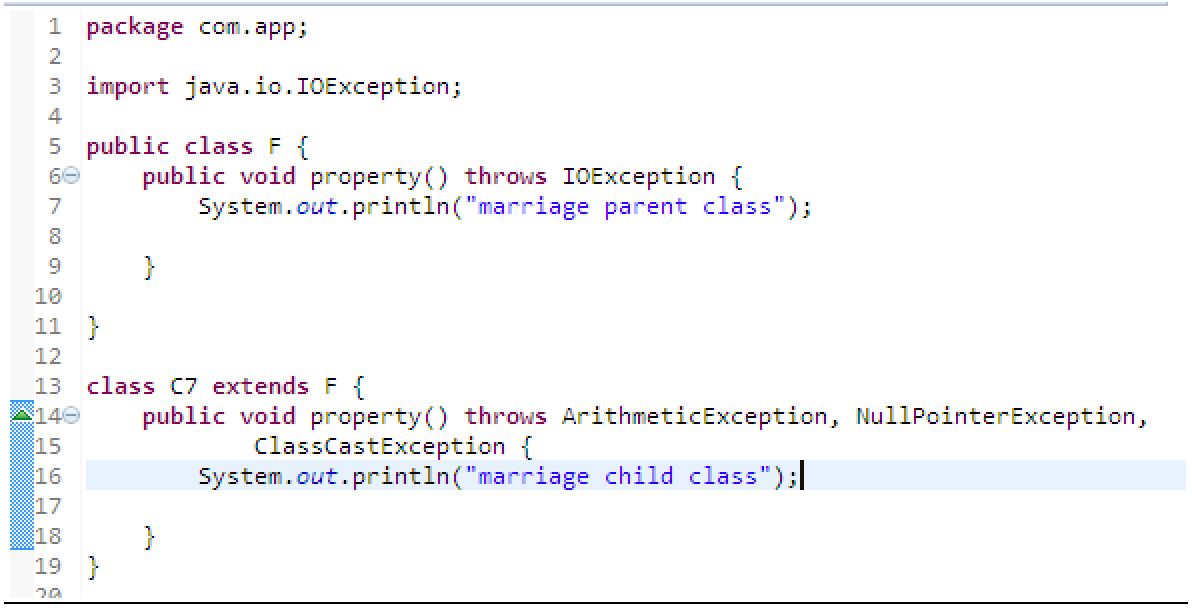
Exception InterruptedException is not compatible with throws clause in E.property()

2 quick fixes available:

- Remove exceptions from 'property(..)'
- Add exceptions to 'E.property(..)'

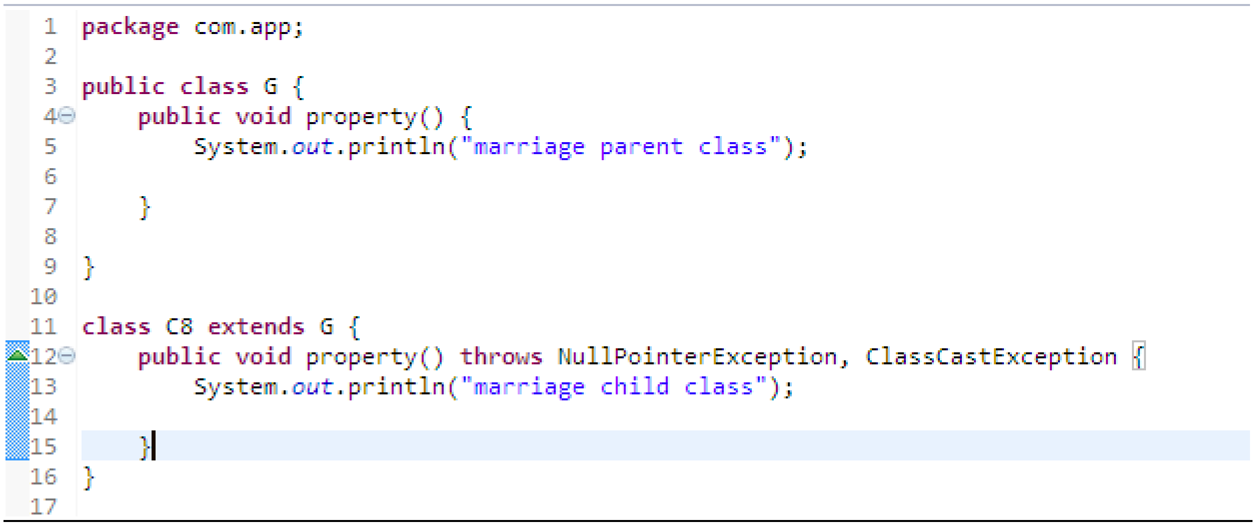
Press 'F2' for focus

g. Parent class method exception is parent class of child class method exception.



```
package com.app;

import java.io.IOException;

public class F {
    public void property() throws IOException {
        System.out.println("marriage parent class");

    }

}

class C7 extends F {
    public void property() throws ArithmeticException, NullPointerException,
            ClassCastException {
        System.out.println("marriage child class");

    }
}
```

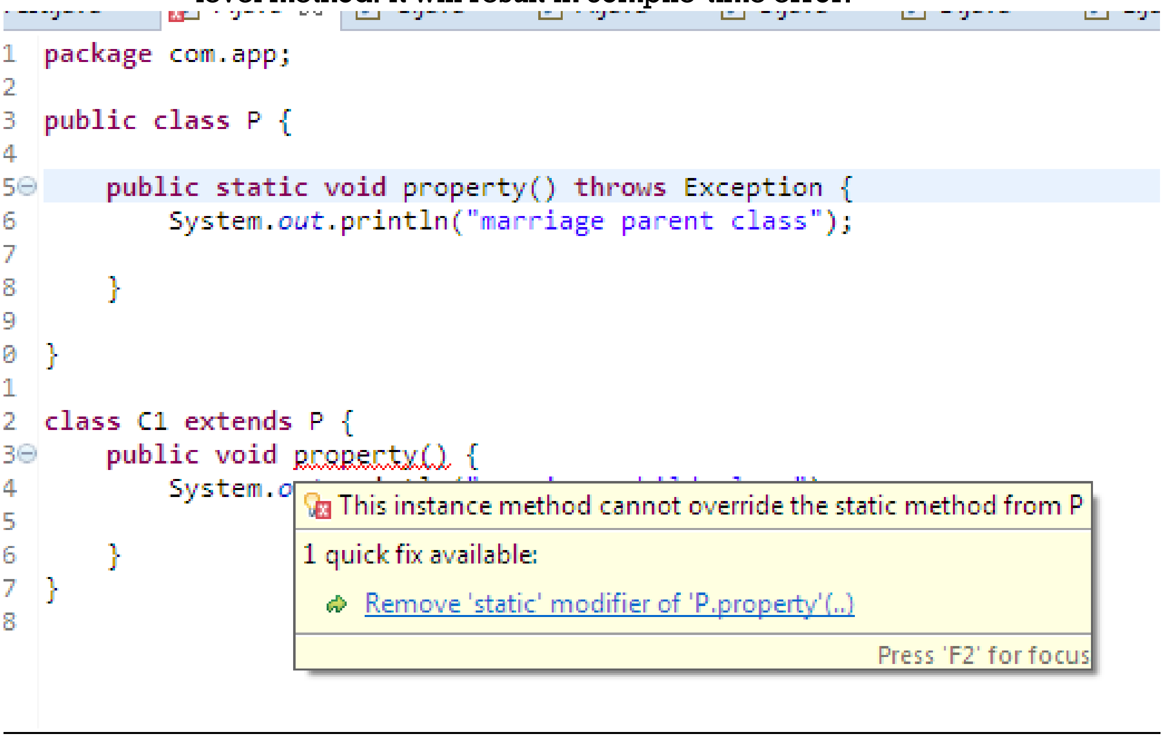
3. For un-checked exception there is no such rule



```
package com.app;

public class G {
    public void property() {
        System.out.println("marriage parent class");

    }

}

class C8 extends G {
    public void property() throws NullPointerException, ClassCastException {
        System.out.println("marriage child class");

    }
}

```

h. **Overriding w.r.t. static methods**:

i. We cannot override static method (Class level method) with Object level method. It will result in compile-time error.

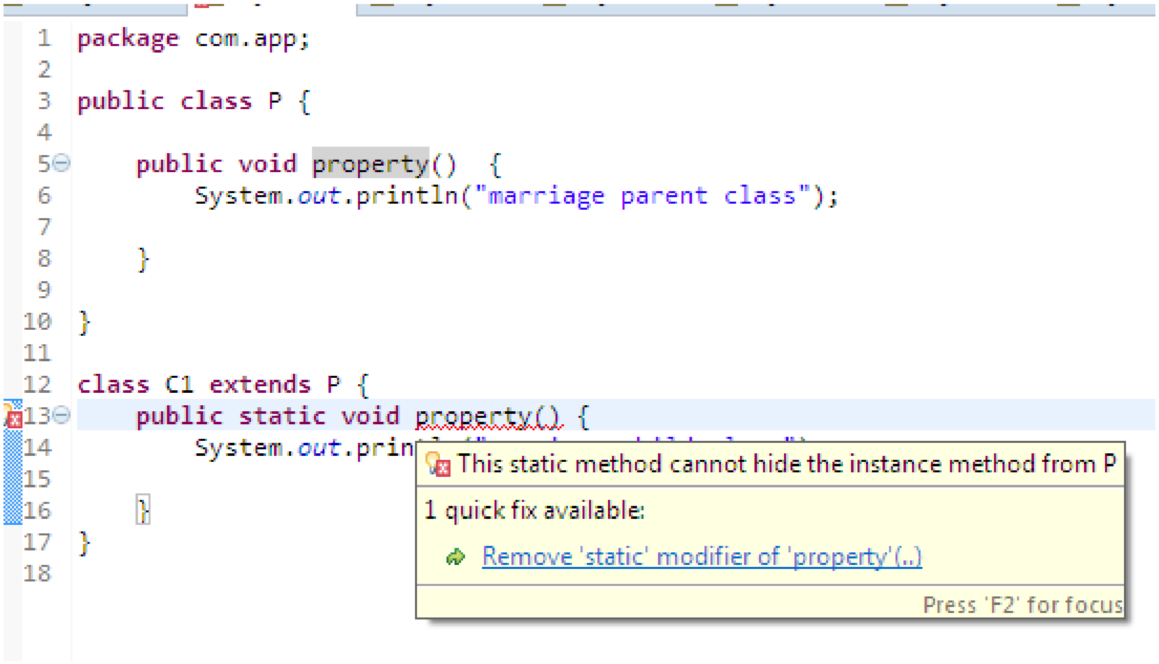


```
package com.app;

public class P {

    public static void property() throws Exception {
        System.out.println("marriage parent class");

    }

}

class C1 extends P {
    public void property() {
        System.o

    }
}

```

This instance method cannot override the static method from P

1 quick fix available:

ii. We cannot over Object level method with class level method. i.e. static method cannot override non-static method.



```java
package com.app;

public class P {

    public void property()  {
        System.out.println("marriage parent class");

    }

}

class C1 extends P {
    public static void property() {
        System.out.prin

    }
}

```

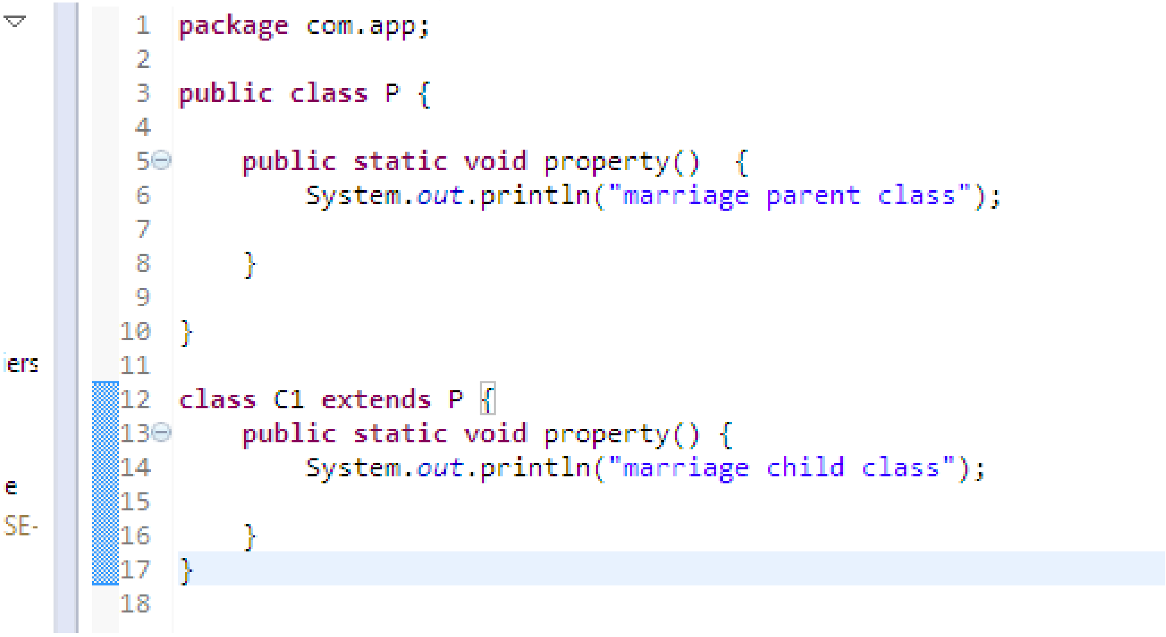
This static method cannot hide the instance method from P

1 quick fix available:

Remove 'static' modifier of 'property'(..)

Press 'F2' for focus

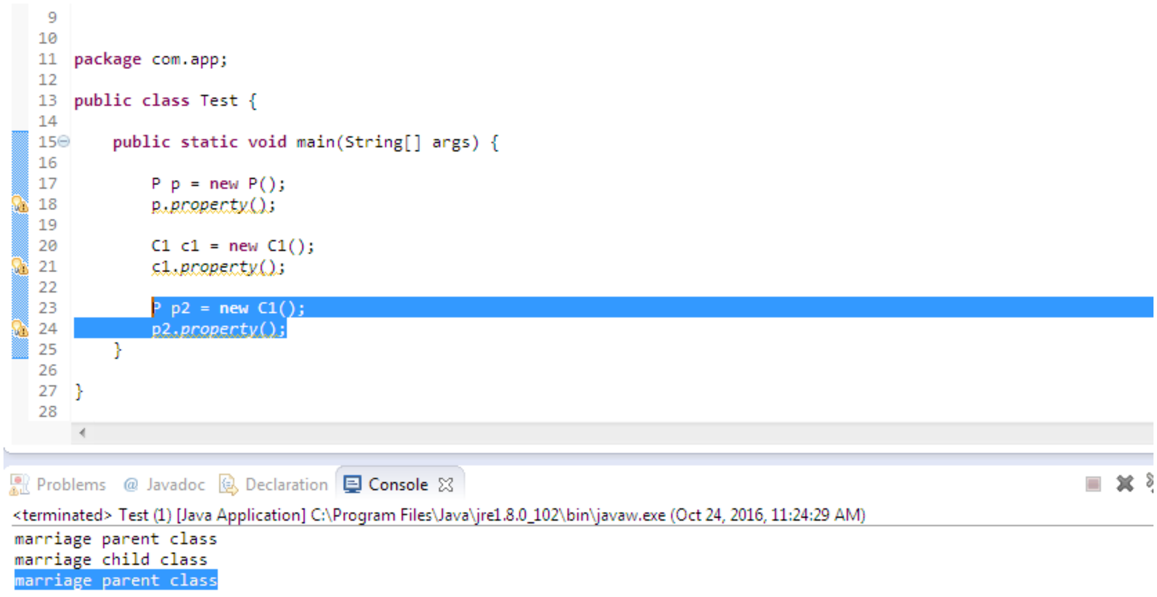
iii. If both parent and child class methods are static, it is called as Method hiding. It does not throw any compile-time error. It is not considered as method overriding.



```java
package com.app;

public class P {

    public static void property()  {
        System.out.println("marriage parent class");

    }

}

class C1 extends P {
    public static void property() {
        System.out.println("marriage child class");

    }
}

```

iv. ***Method Hiding***:

1. All the rules of method hiding are same as method overriding.
2. In method hiding, method resolution is compiler based on reference type.



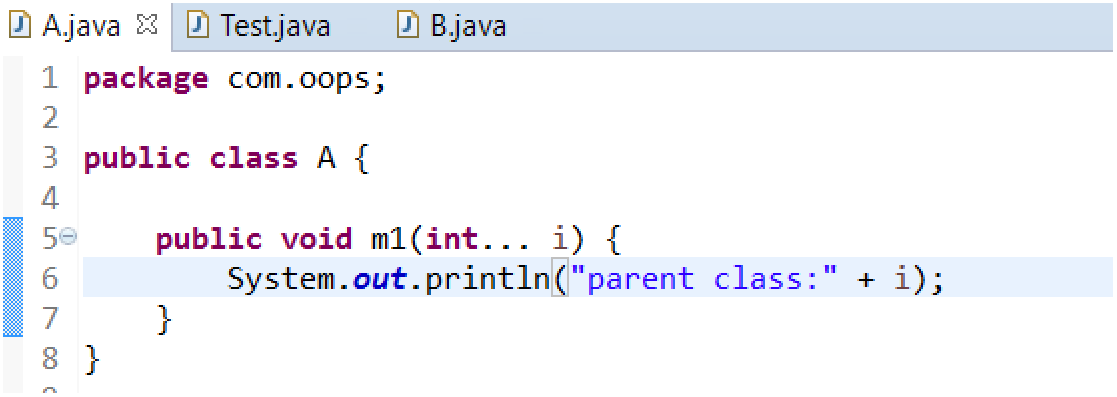
```
package com.app;

public class Test {

    public static void main(String[] args) {

        P p = new P();
        p.property();

        C1 c1 = new C1();
        c1.property();

        P p2 = new C1();
        p2.property();
    }

}
```

```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Oct 24, 2016, 11:24:29 AM)
marriage parent class
marriage child class
marriage parent class
```

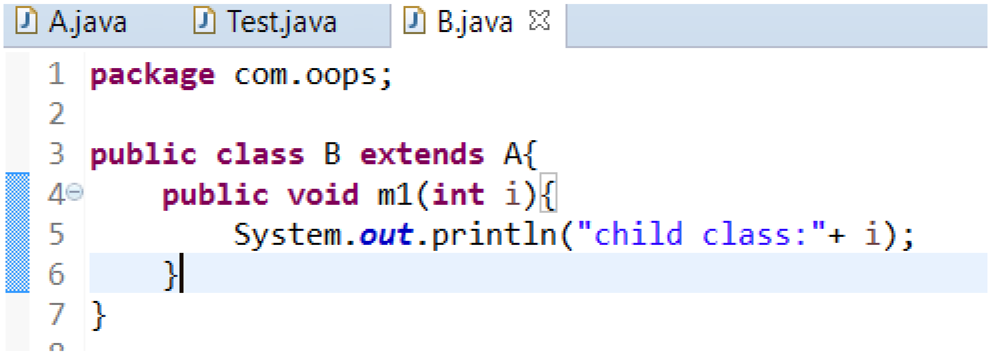
i. **Overriding w.r.t. var-arg methods**:

   i. Here in the below example, looks like overriding but it is overloading since a var-arg method should be overridden with var-arg, not the other argument.

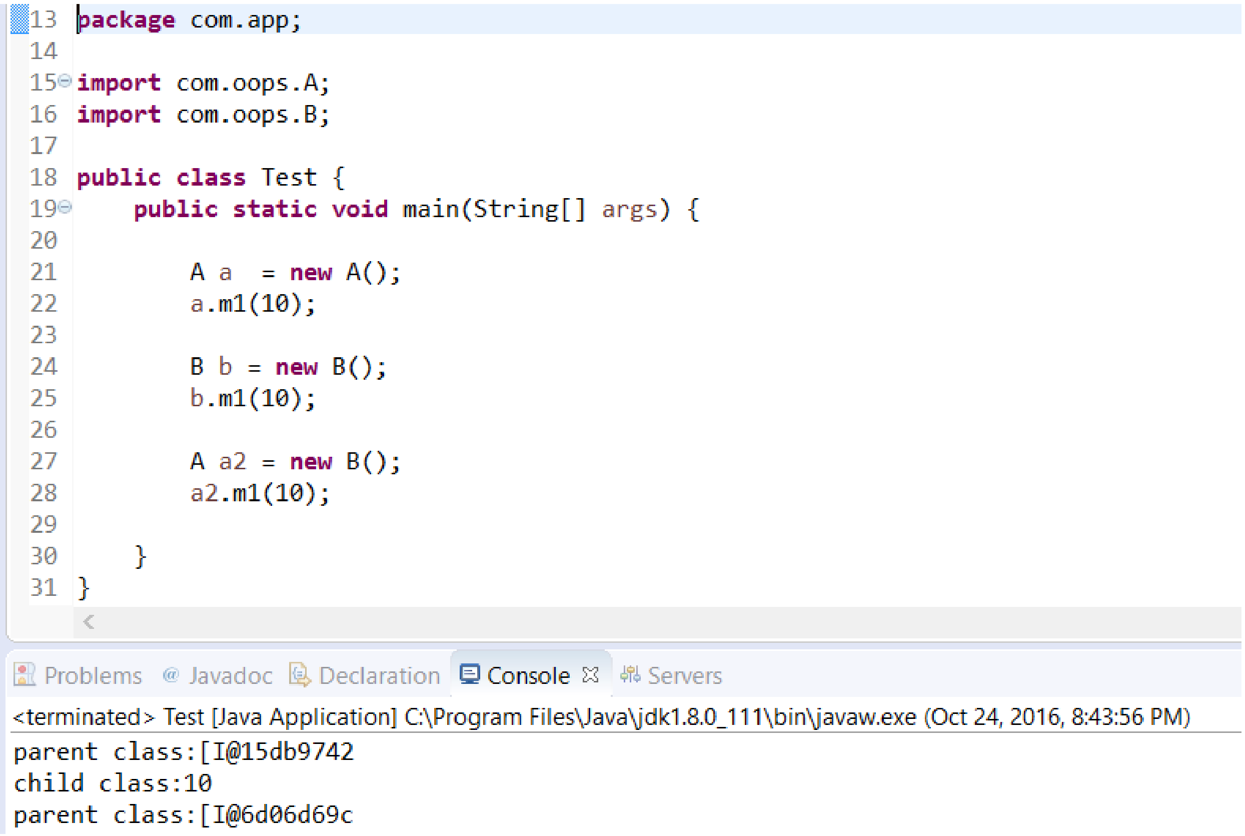
   ii. If we replace the child method with var-arg method then it will become method overriding.



```
package com.oops;

public class A {

    public void m1(int... i) {
        System.out.println("parent class:" + i);
    }
}
```



```
package com.oops;

public class B extends A{
    public void m1(int i){
        System.out.println("child class:"+ i);
    }
}
```

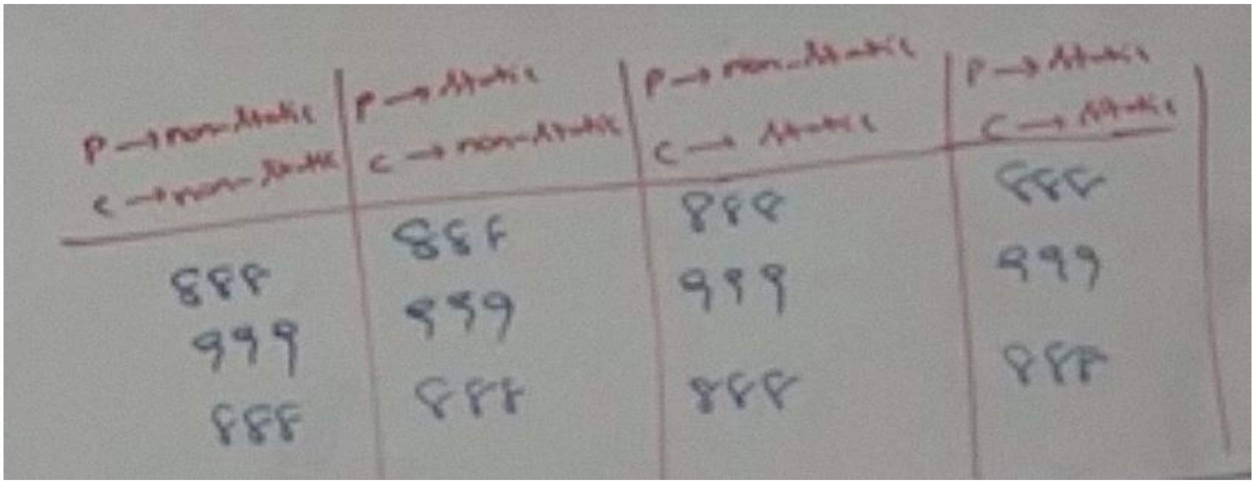


```java
package com.app;

import com.oops.A;
import com.oops.B;

public class Test {
    public static void main(String[] args) {

        A a  = new A();
        a.m1(10);

        B b = new B();
        b.m1(10);

        A a2 = new B();
        a2.m1(10);

    }
}
```

```
<terminated> Test [Java Application] C:\Program Files\Java\jdk1.8.0_111\bin\javaw.exe (Oct 24, 2016, 8:43:56 PM)
parent class:[I@15db9742
child class:10
parent class:[I@6d06d69c
```

j. Overriding is applicable only methods but not on variables. Variable resolution is always taken care by compiler based on reference type.

   i. It does not affect if it is static or non-static.



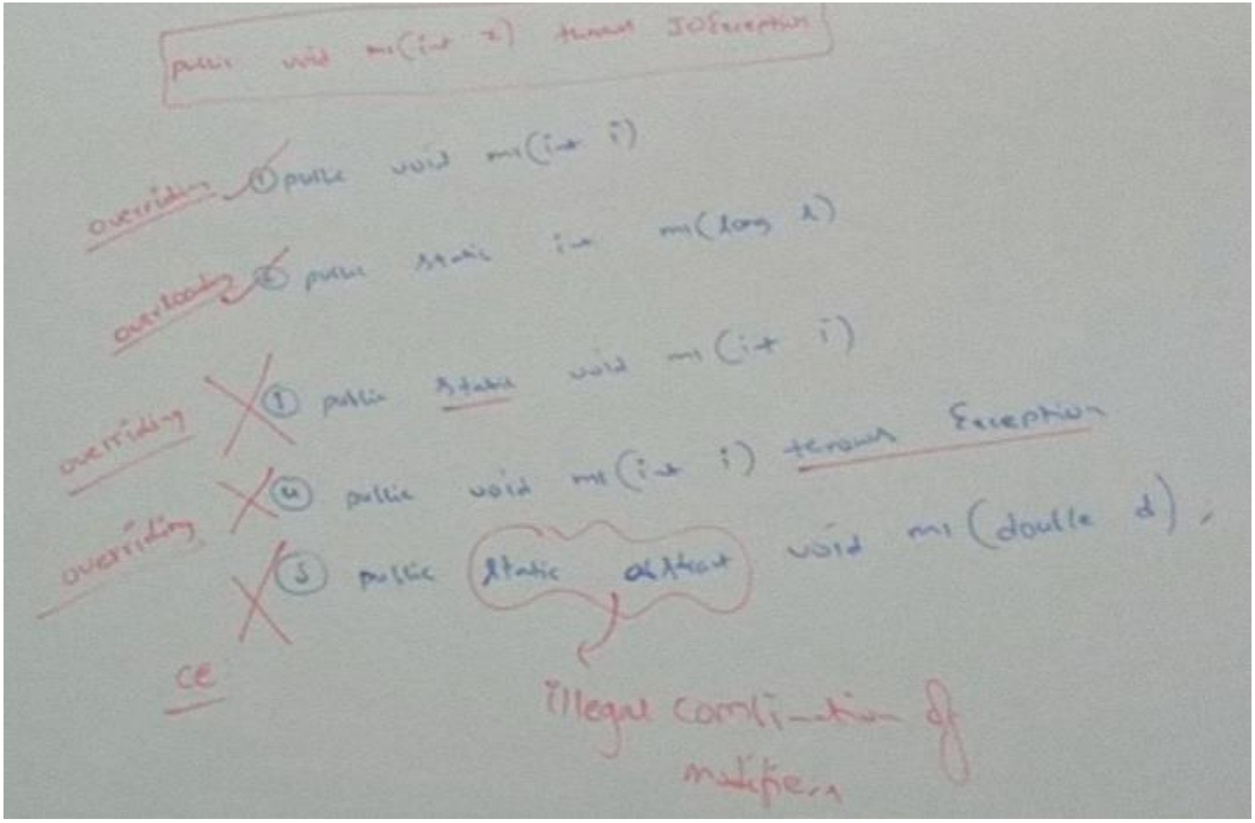
P → non-static C → non-static	P → static C → non-static	P → non-static C → static	P → static C → static
888	888	888	888
999	999	999	999
888	888	888	888

Property	Overloading	Overriding
Method Names	Same	Must be Same
Argument types	Different (at least order)	Must be Same including order
Method signatures	Different	Must be Same
Return Types	No rule	Must be Same till 1.4

		From 1.5 version co-variant return types are allowed
Private, static, final	Can be overloaded	Cannot override
Access Modifiers	No restrictions	The scope of access modifiers cannot be reduced.
Throws clause	No restrictions	If child class method throws any checked exception, parent class should throw the same checked exception or its parents. But no restrictions for Unchecked Exception.
Method resolution	Always takes care by compiler based in reference type	Always takes care by JVM based on runtime Object
Also known as	Compile-time polymorphism/Static Polymorphism/Early binding	Run-time Polymorphism/Dynamic binding/Late binding.

Note:

Consider the following 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 l)

overriding ✗ ③ public static void m1(int i)

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

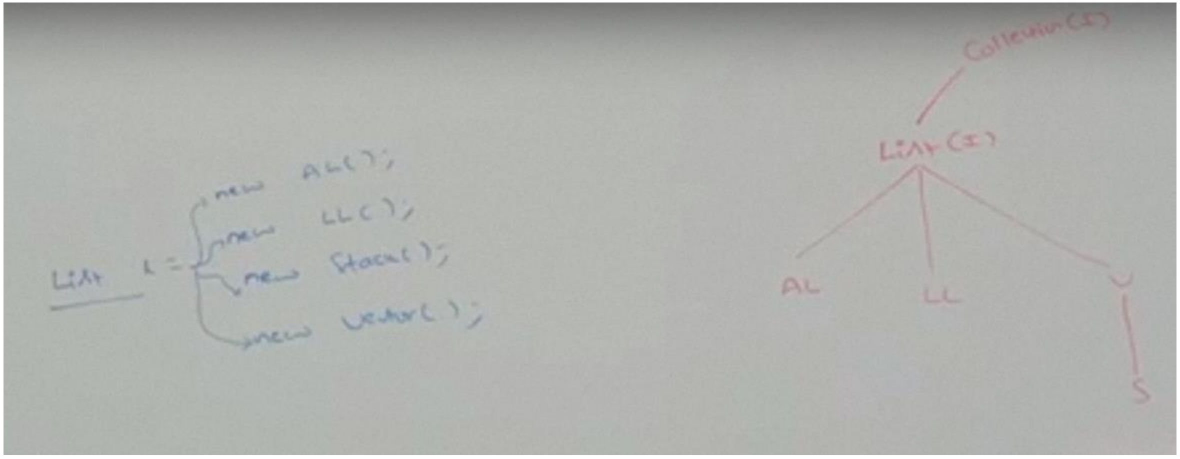
✗ ⑤ public static abstract void m1(double d);

CE

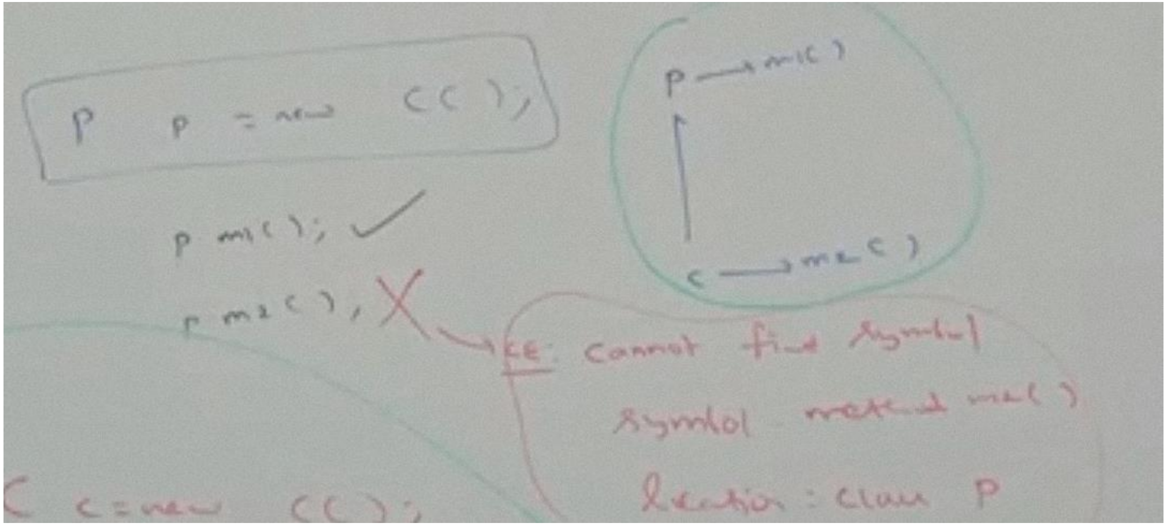
Illegal combination of modifiers

k. Polymorphism

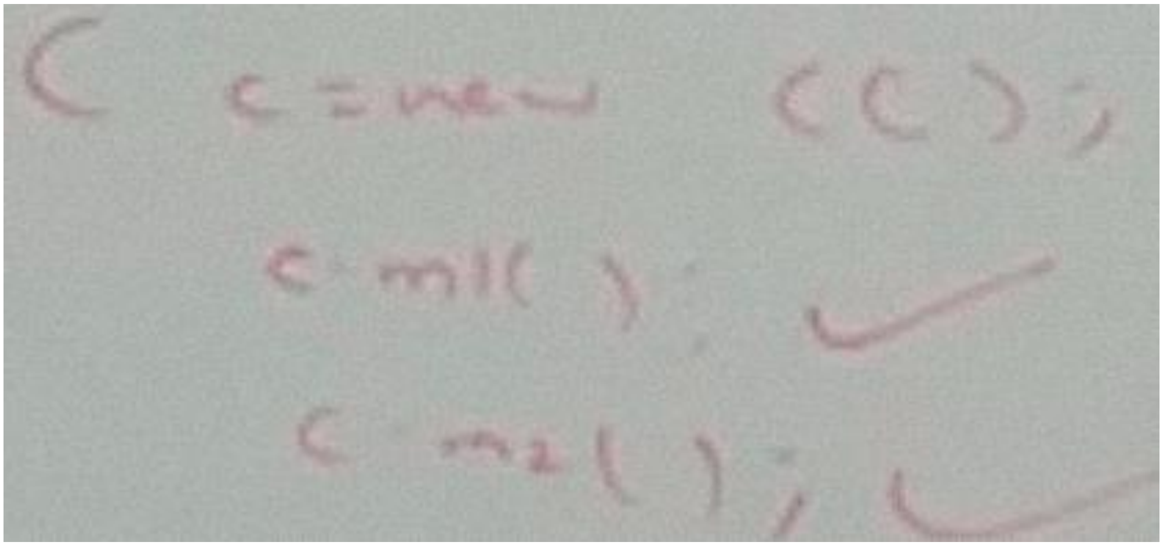
  i. One name but multiple forms is the concept of Polymorphism
  ii. Example 1: method overloading and method overriding.
  iii. Example 2: Usage of Child reference to hold the Parent object.



  iv. Parent class reference can be used to hold child object. But by using that reference we can call only the methods available in parent class. We cannot call child specific methods.



  v. But by using child object reference, we can call both parent class and child class methods.



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

vi. **When should we go for Parent reference to hold child object**?

1. If we are not sure of or if we do not know exact the run-time object then we can use parent reference.
2. The first element in the ArrayList can be of any type (Student Object/Customer Object/String Object or String Buffer Object), hence the return type of get method is Object which can hold any object.

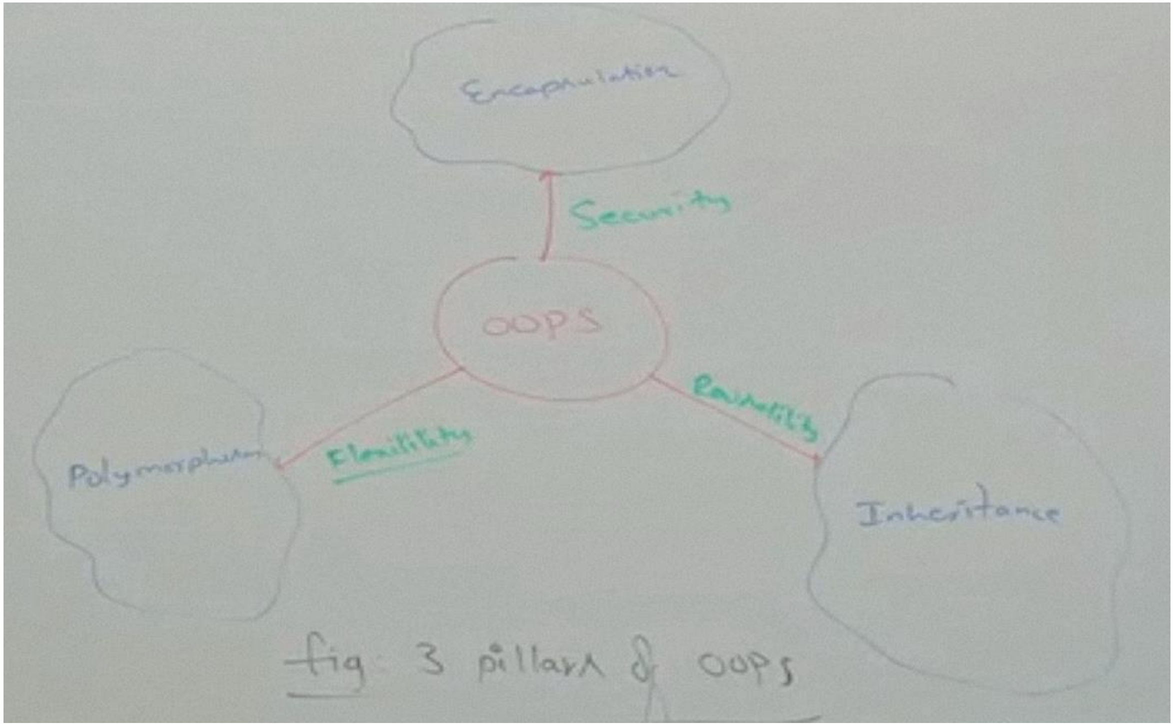
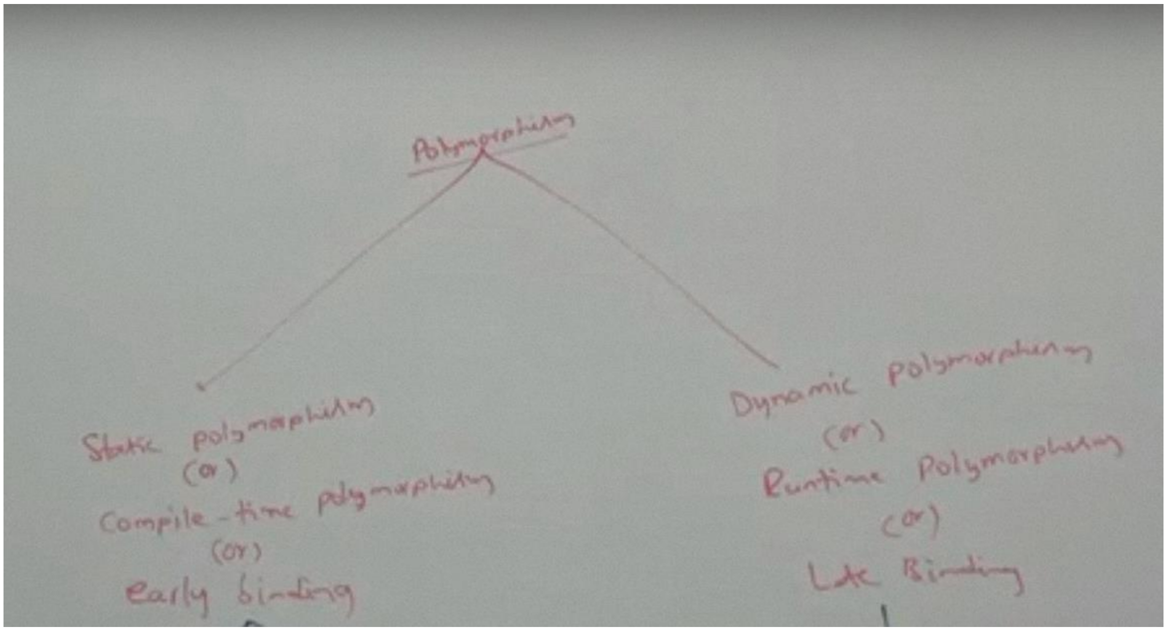
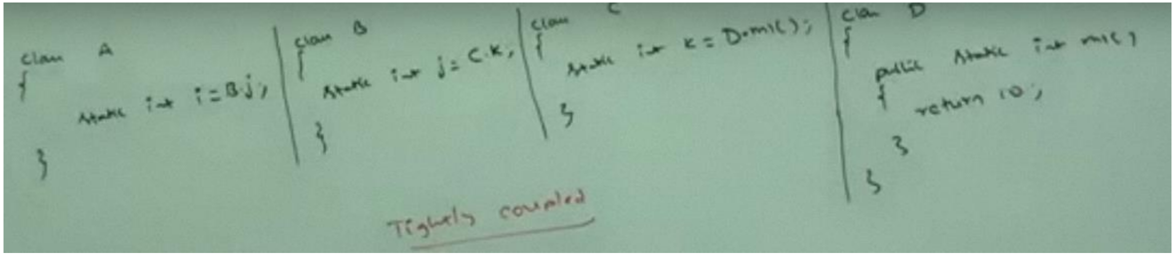


fig: 3 pillars of OOPS



10. Coupling
    a. The degree of dependency is called Coupling.
    b. If dependency is high, it is tightly coupled and if the dependency is less, it is loosely coupled.
    c. Tightly coupled:
        i. Enhancement will be difficulty.
        ii. Reusability will be suppressed.
        iii. Maintainability will be affected.



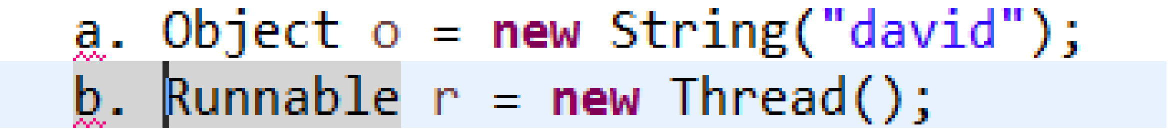
```
class A
{
    static int i=B.j;
}

class B
{
    static int j=C.k;
}

class C
{
    static int k=D.m1();
}

class D
{
    public static int m1()
    {
        return 10;
    }
}
```

Tightly coupled

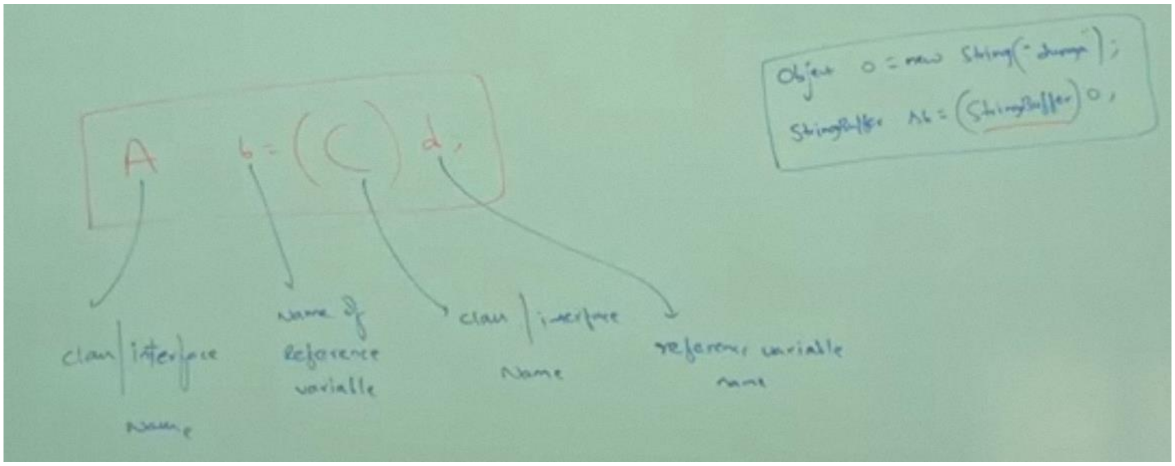
    d. Loosely coupled:
        i. Maintaining less dependency between the objects
11. Cohesion
    a. A clear well-defined functionality is defined for every component, such components are high Cohesion components.
    b. High cohesion is always good programming practice.
    c. Disadvantages of Low Cohesion:
        i. Enhancement will be difficulty.
        ii. Reusability will be suppressed.
        iii. Maintainability will be affected.
    d. Advantages of High cohesion

i. Without effecting remaining components we can modify any other component.
ii. Enhancement will be easy.
iii. Reusability.
iv. Maintainability.

12. Object type casting:
    a. We can use parent reference to hold child object.
    b. We can use interface reference to hold implemented class object.



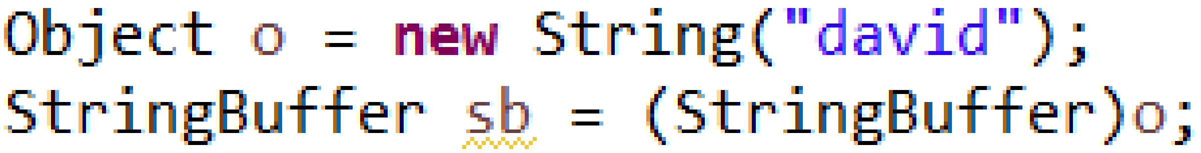
```
a. Object o = new String("david");
b. Runnable r = new Thread();
```



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

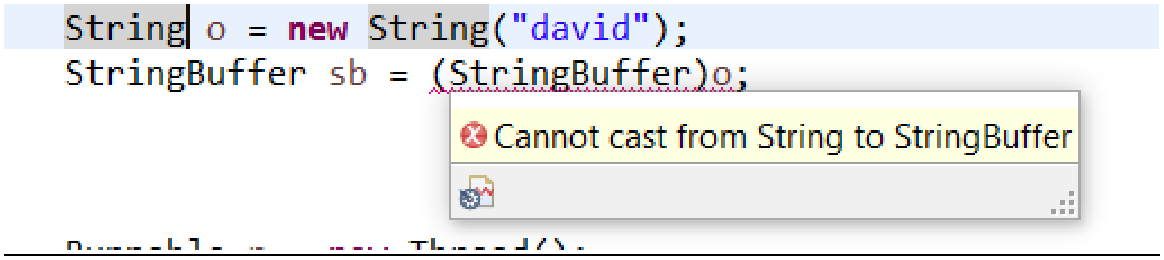
```
Object o = new String("durga");
StringBuffer sb = (StringBuffer)o;
```

c. Conditions to check for type casting:
    i. Compiler's rule:
        1. The type of "d" and "C" must have some relation either Child-Parent or Parent-Child or same type.



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

        2. Else it will result in compile time error.

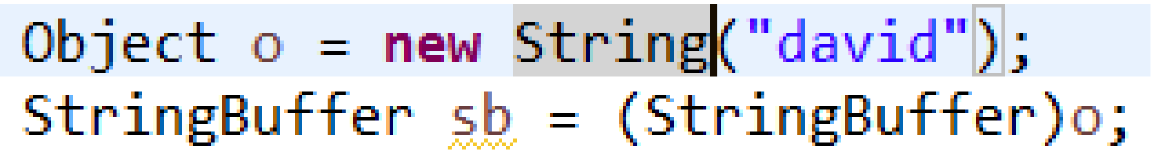


```
String o = new String("david");
StringBuffer sb = (StringBuffer)o;
```

Cannot cast from String to StringBuffer

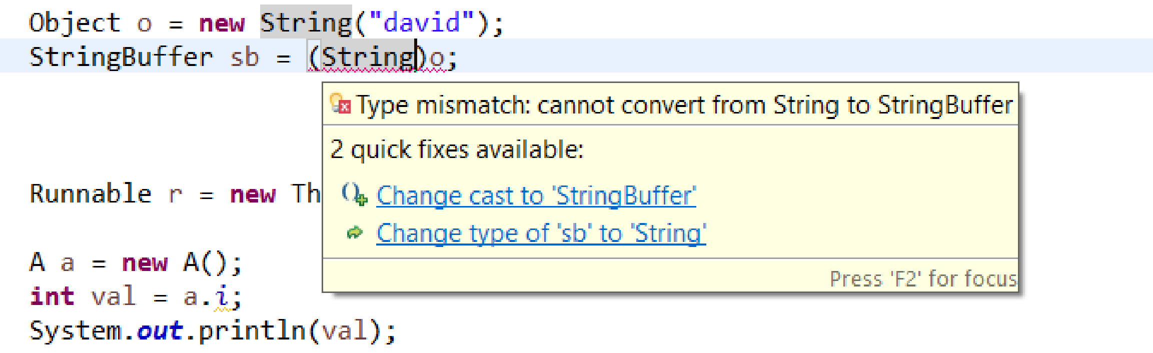
```
Runnable r = new Thread();
```

3. "C" must be either same type as "A" or derived type of "A"



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

4. Else it will result in Compile time error.



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

Runnable r = new Th

A a = new A();
int val = a.i;
System.out.println(val);
```

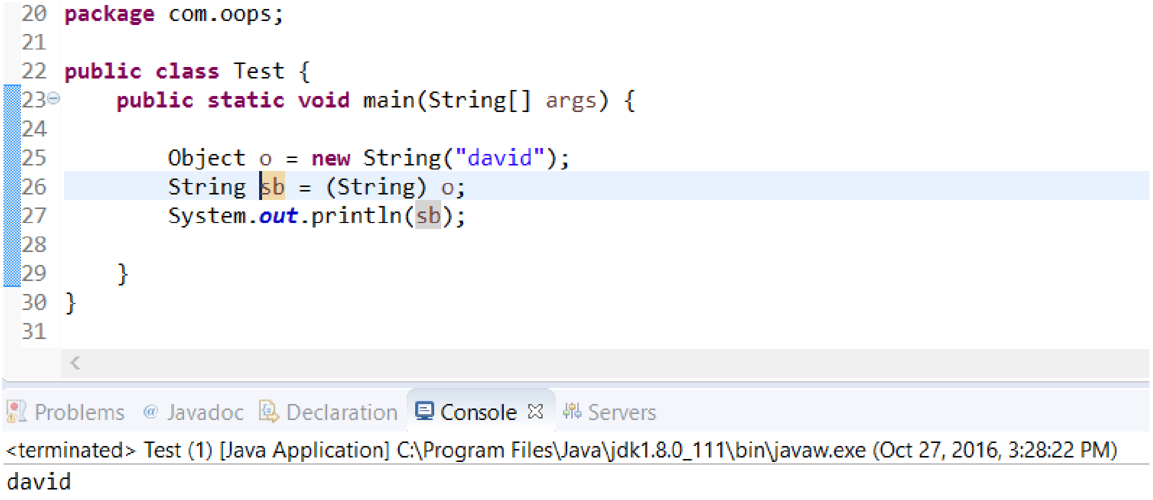
Type mismatch: cannot convert from String to StringBuffer

2 quick fixes available:

Change cast to 'StringBuffer'
Change type of 'sb' to 'String'
Press 'F2' for focus

ii. JVM's rule:

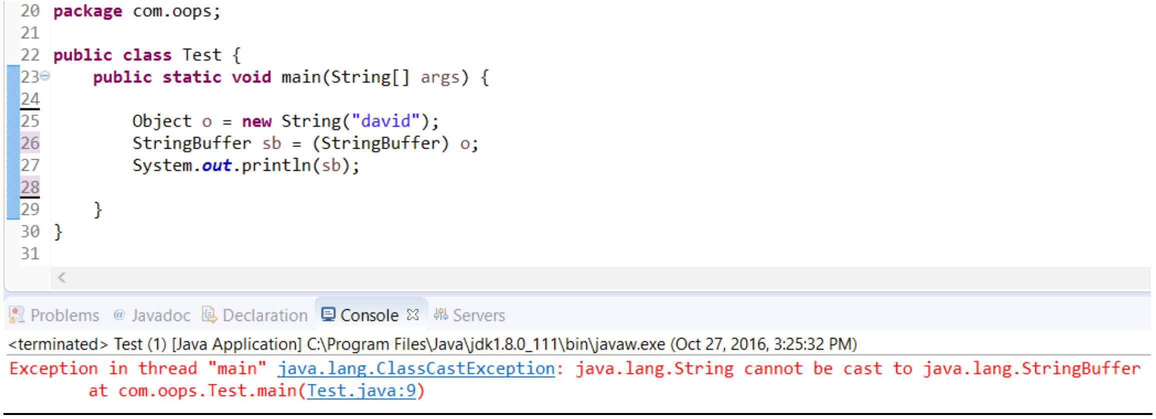
1. Run-time object of "d" must be either same or derived type of "C".



```java
package com.oops;

public class Test {
    public static void main(String[] args) {

        Object o = new String("david");
        String sb = (String) o;
        System.out.println(sb);

    }
}
```

```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jdk1.8.0_111\bin\javaw.exe (Oct 27, 2016, 3:28:22 PM)
david
```

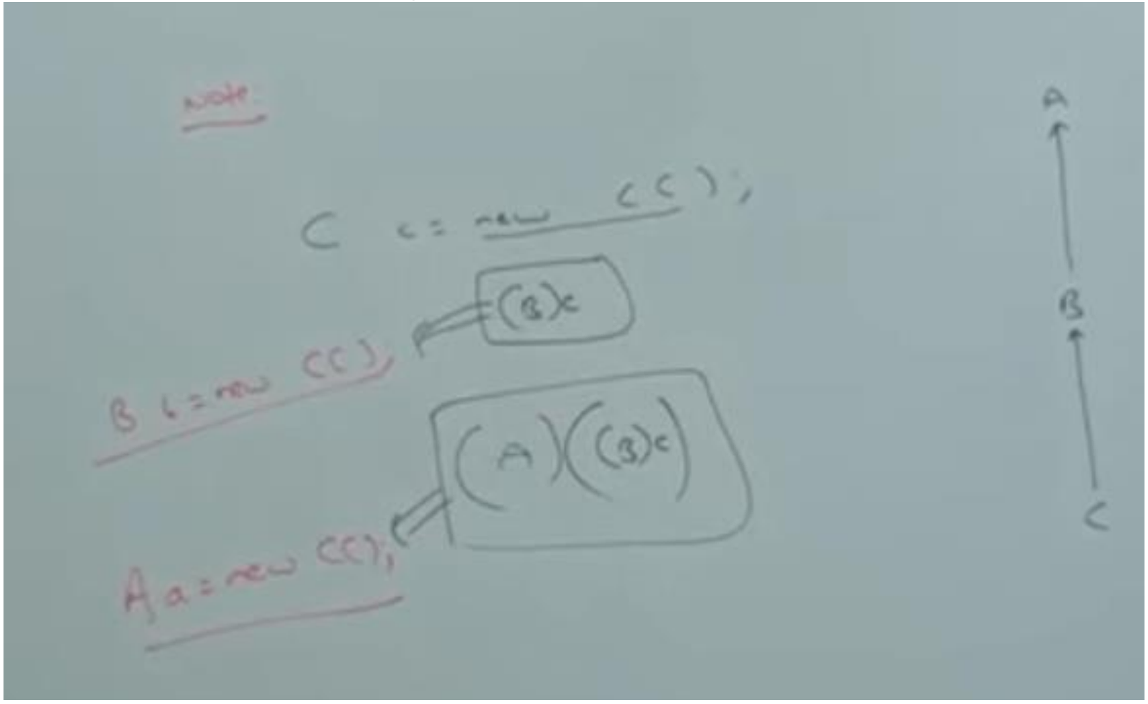
2. Else it will result run-time exception.



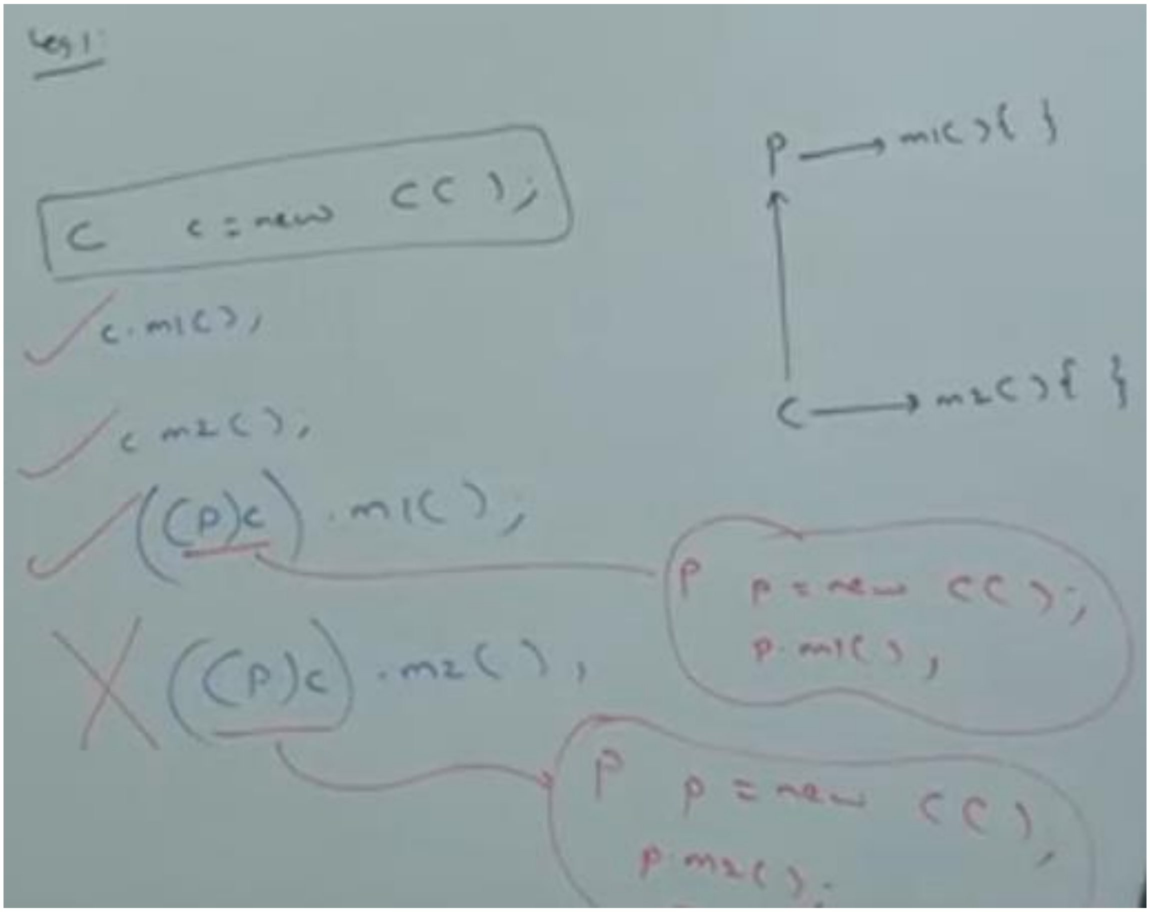
```java
package com.oops;

public class Test {
    public static void main(String[] args) {

        Object o = new String("david");
        StringBuffer sb = (StringBuffer) o;
        System.out.println(sb);

    }
}

```

```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jdk1.8.0_111\bin\javaw.exe (Oct 27, 2016, 3:25:32 PM)
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.StringBuffer
	at com.oops.Test.main(Test.java:9)
```

d. Through type casting we are not creating any new Object. For the existing object we are providing another type of reference variable.



e. In below example, parent reference can hold child object but can call only parent specific object.



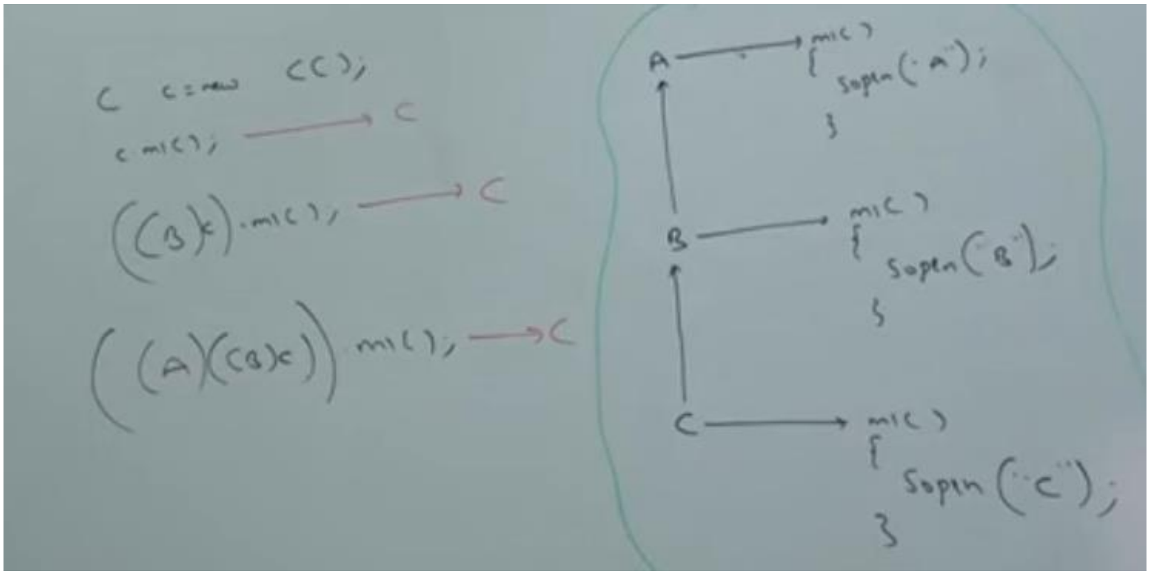
Eg1:

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

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

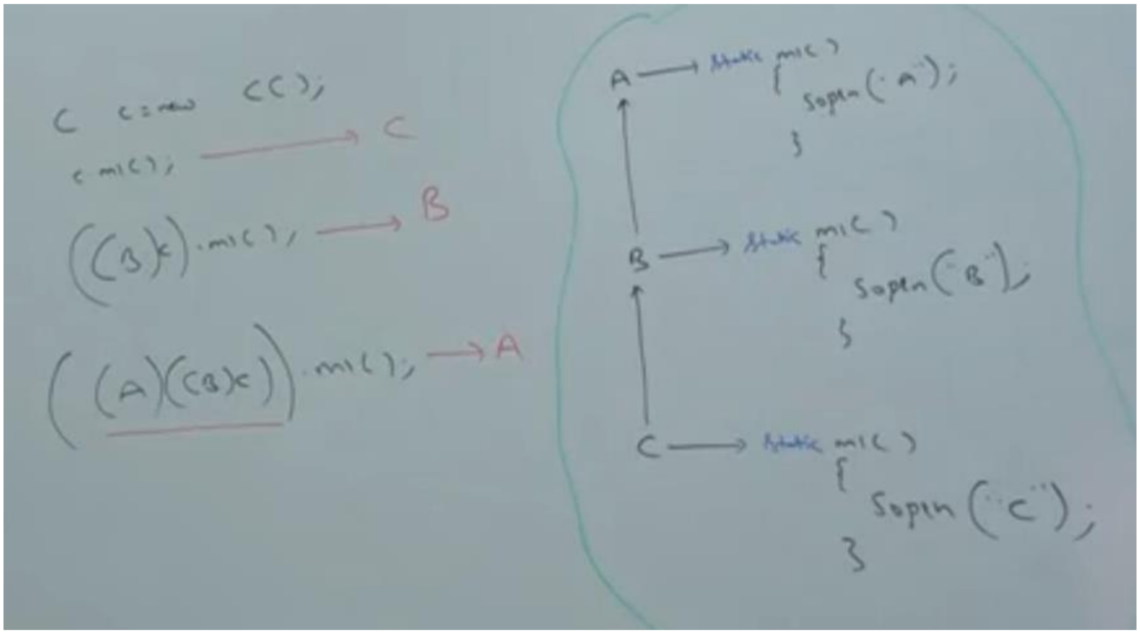
```
P p = new C();
p.m2();
```

f. The below example is overriding. And hence the method resolution is at run-time and prints only "C" all the time.



```
C c = new C();
c.m1(); → C
((B)c).m1(); → C
((A)((B)c)).m1(); → C
```

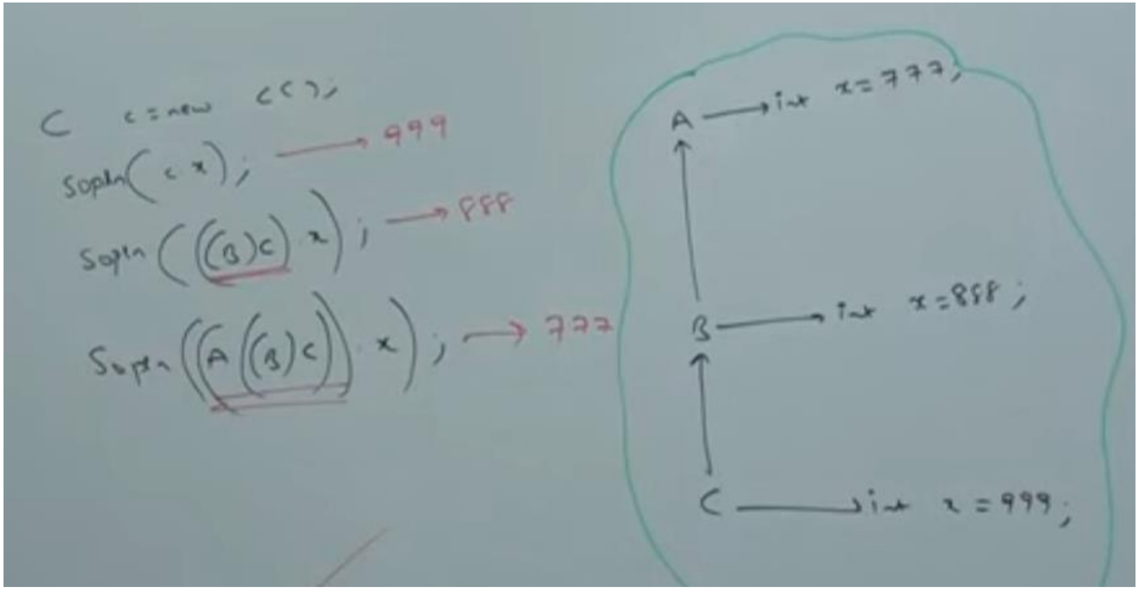
g. The below example is an example of Method hiding. Here the method resolution is based on reference type variable by compiler and hence it executes different class methods.



```
C c=new C();
c.m1(); ——→ C
((B)c).m1(); ——→ B
((A)((B)c)).m1(); ——→ A
```

```
A ——→ static m1()
      {
        Sopln("A");
      }
B ——→ static m1()
      {
        Sopln("B");
      }
C ——→ static m1()
      {
        Sopln("C");
      }
```

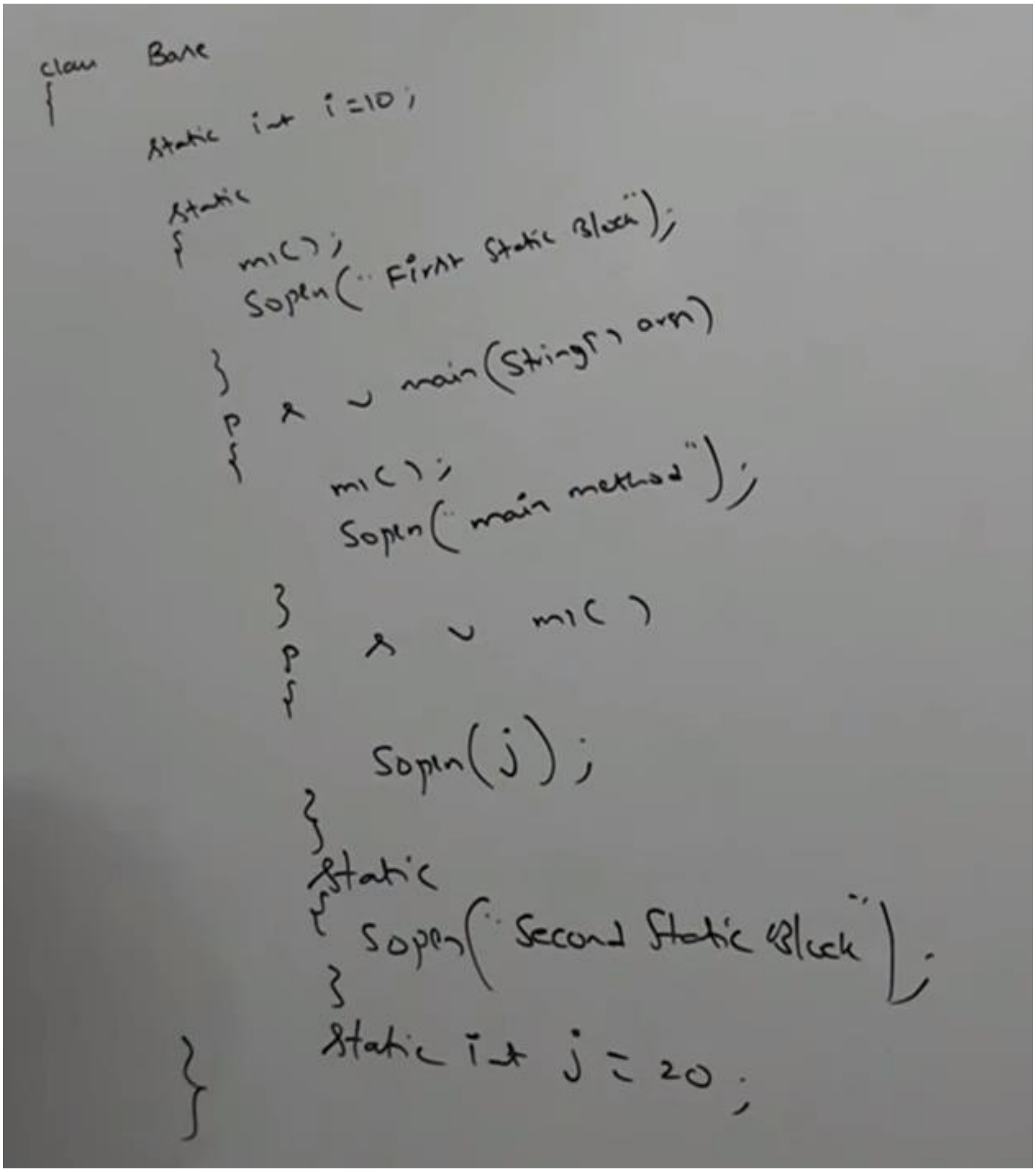
h. In the below example, variable resolution is always based on reference type and hence the values are printed as such.



```
C c=new C();
Sopln(c.x); ——→ 999
Sopln(((B)c).x); ——→ 888
Sopln(((A)((B)c)).x); ——→ 777
```

```
A ——→ int x=777;
B ——→ int x=888;
C ——→ int x=999;
```

13. Static Control flow:
    a. Identification of static members/variables from top to bottom.
    b. Execution of static variable assignments and static blocks from top to bottom.
    c. Execution of main method.

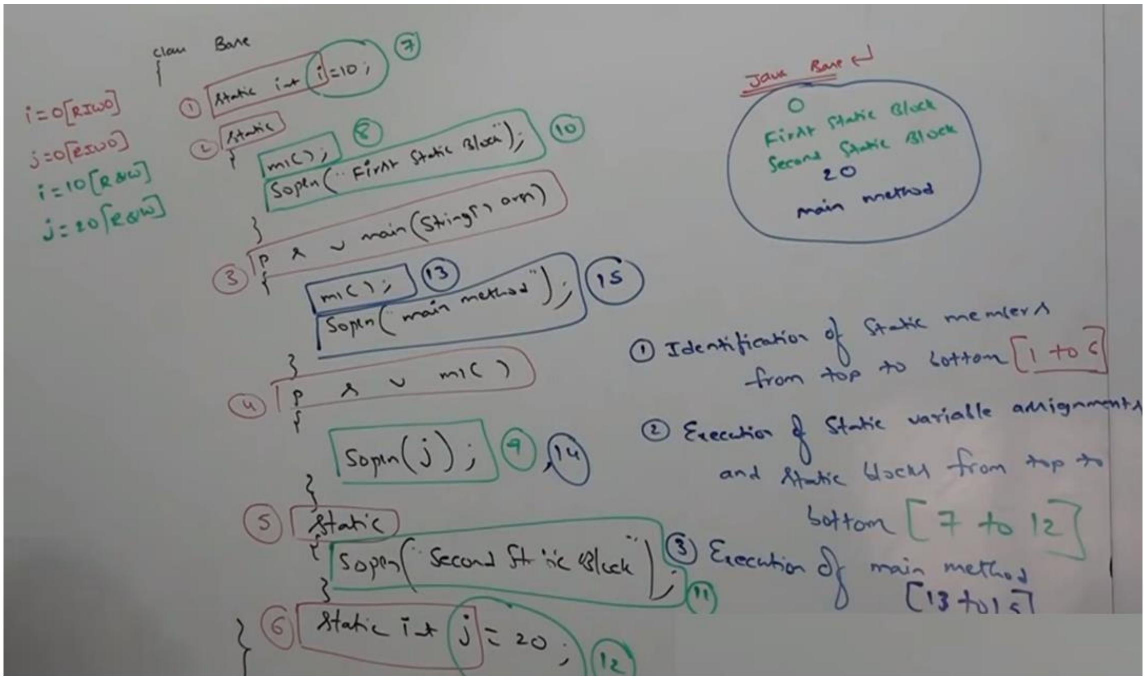


```
class Base
{
    static int i=10;
    static
    {
        m1();
        Sopln("First Static Block");
    }
    p s v main(String[] args)
    {
        m1();
        Sopln("main method");
    }
    p s v m1()
    {
        Sopln(j);
    }
    static
    {
        Sopln("Second Static Block");
    }
    static int j=20;
}
```

d. At the time of identification JVM will assign default value, hence "i and j" values to 0 (zero), this is called read indirectly write only.

e. Now the "i" value will be assigned as 10. This state is called read and write state.

f. The flow will be as follows.



```
class Base
{
  static int i=10;
  static
  {
    m1();
    Sopln("First Static Block");
  }
  p s v main(String[] arn)
  {
    m1();
    Sopln("main method");
  }
  p s v m1()
  {
    Sopln(j);
  }
  static
  {
    Sopln("Second Static Block");
  }
  static int j=20;
}
```

i=0[RIWO]
j=0[RIWO]
i=10[R&W]
j=20[R&W]

Java Base ↵

0
First Static Block
Second Static Block
20
main method

① Identification of Static members from top to bottom [1 to 6]

② Execution of Static variable assignments and Static blocks from top to bottom [7 to 12]

③ Execution of main method [13 to 15]

g. Note:

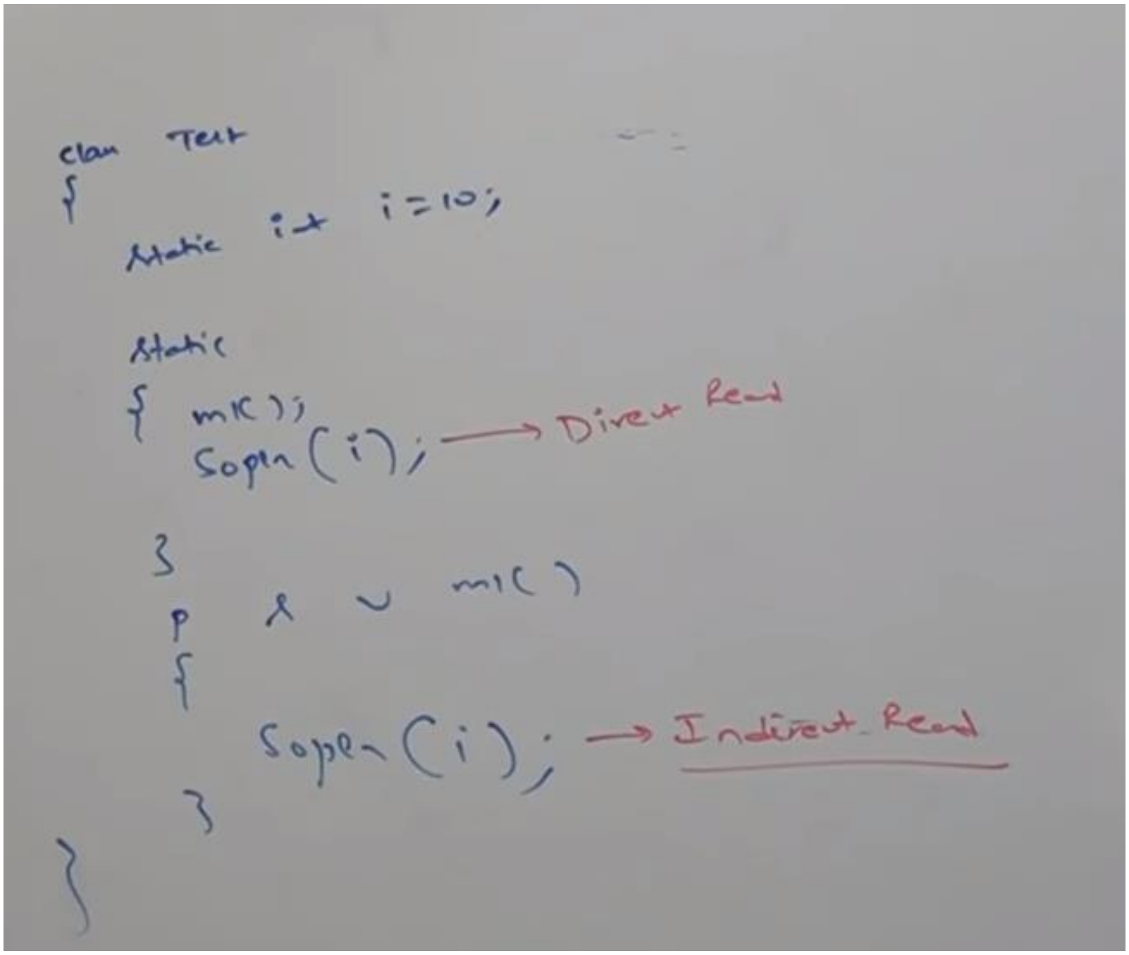
i. Inside static block if we are trying to read a variable, then that read operation is called Direct Read.

ii. 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.

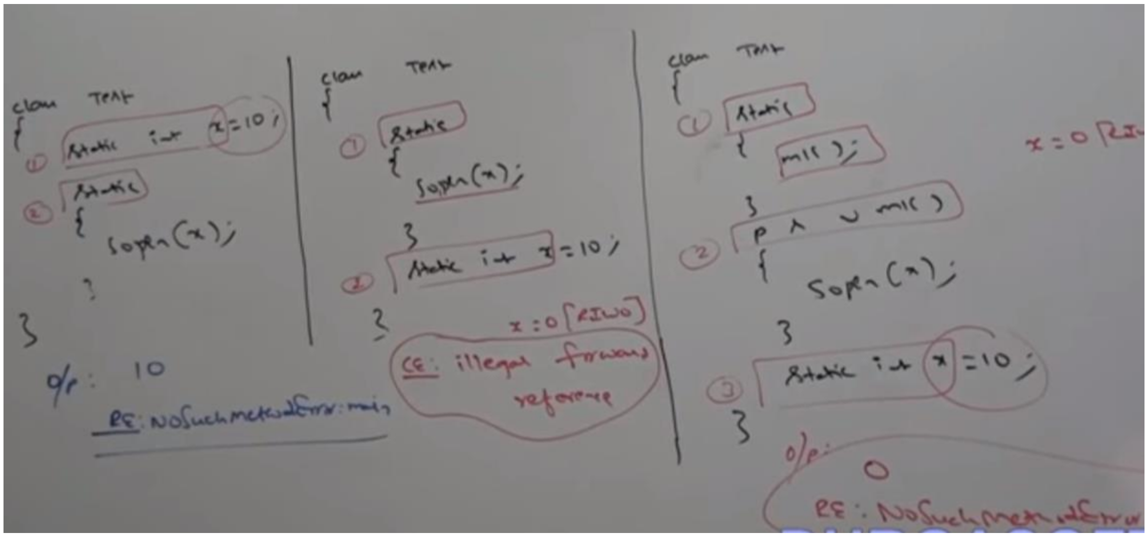
iii. If a variable is identified but original values is not assigned then the variable is said to be in Read Indirectly Write Only (RIWO).

iv. If a variable is in RIWO state then we cannot perform Direct Read but we can perform Indirect Read.

v. If we are trying to read directly then we will get compile time error saying “Illegal Forward Reference”.



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

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

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



```
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;
}
o/p:
     0
RE: NoSuchMethodError
```

x=0 [RI