* Final keyword :

→ Final is a keyword, which is used to avoid overriding.
→ final variable value cannot be overridden.
→ final Method cannot be over overridden.
→ final class cannot be inherited so that we cannot override any of the final class method.

OR:

final class method's cannot be overdone, because final class cannot be inherited.

e.g: showing CTE when trying to override final method

e.g: showing CTE when trying to inherit from final class e.g - String class.

* Difference b/w method overridding and method overloading :

	Method overloading	Method overridding
→	We can overload both static and Non-static Method	We can override only non-static Method.
→	We can overload with or without inheritence	for overridding inheritence is mandatory.
→	for overloading Method Name should be same but with variations in the argument list	for overridding complete Method Signature should be same including arguments.
→	We can Overload Main() Method.	We cannot override Main() method.

→ We can overload Constructor → We cannot override Constructor.

→ We can achive Compile time polymorphism using Method Overloading → we can achive Run time polymorphism using Method overridding.

Q.1 Can we override static method ?

Ans: NO, because ~~static member/methods will not be inherited to Subclass and if there is no inheritance we can't achive overridding~~. Method overridding is only for non-static method

Q.2 Can we override Constructor ?

Ans Even though Constructors are non-static, Constructors will not be inherited to subclass and to achive overriding inheritance is mandatory.

Q.3 Can we override Main Method ?

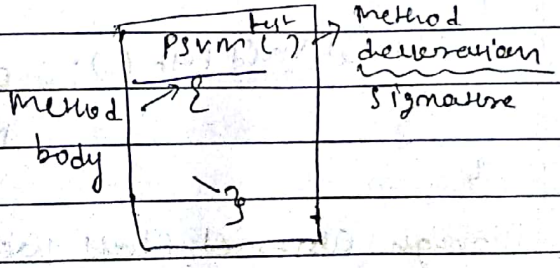
Ans: NO, because Main is a static method ~~and static members will not be inherited by the subclass and to achive overriding, inheritance is mandatory.~~ and Method overridding is only for non-static

⇒ Abstract Class :

Concreate Method :

→ The method which will have declerations and definations together is called Concereite method.

Concreate Class



→ The class in which we can write any Concrete method is called as Concrete class.

```
Public Class A
default {
        static void test()
        {
        }
}
```

* Abstract Method : The method which will be have just declarations, without any definitions is called as Abstract method or Incomplete method.

Public static void test(); only declaration.

* Abstract Class :

→ The class in which we have option to develop abstract method is calling as Abstract class.

→ Abstract Class should be declared with keyword abstract.

→ Abstract method should be declared with keyword abstract.

→ We cannot create object of Abstract Class or Abstract Class cannot be instantiated.

```
Abstract Class B
{
    Abstract void test();
}
```

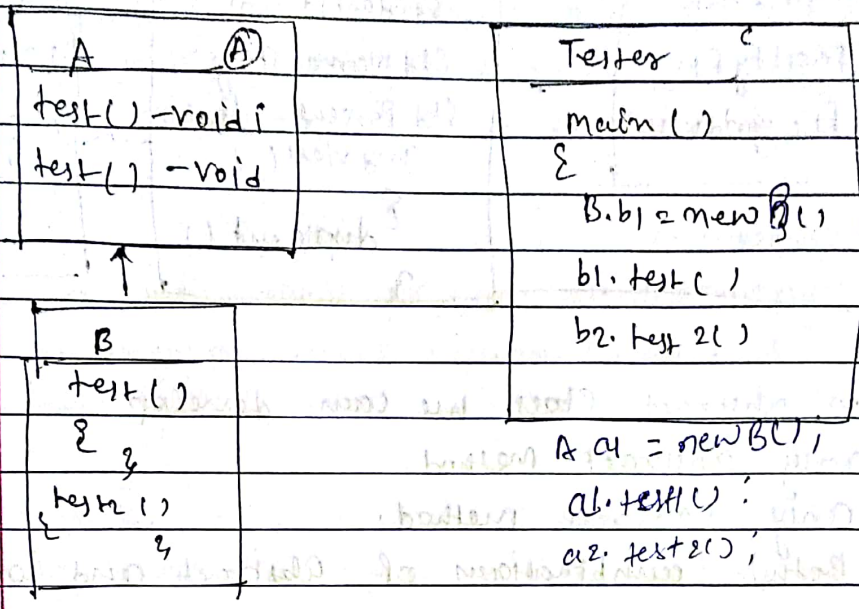
Abstract Class should have a subclass & subclass should override All abstract method.

→ Through Abstract Class we can achive standardization and Abstraction.

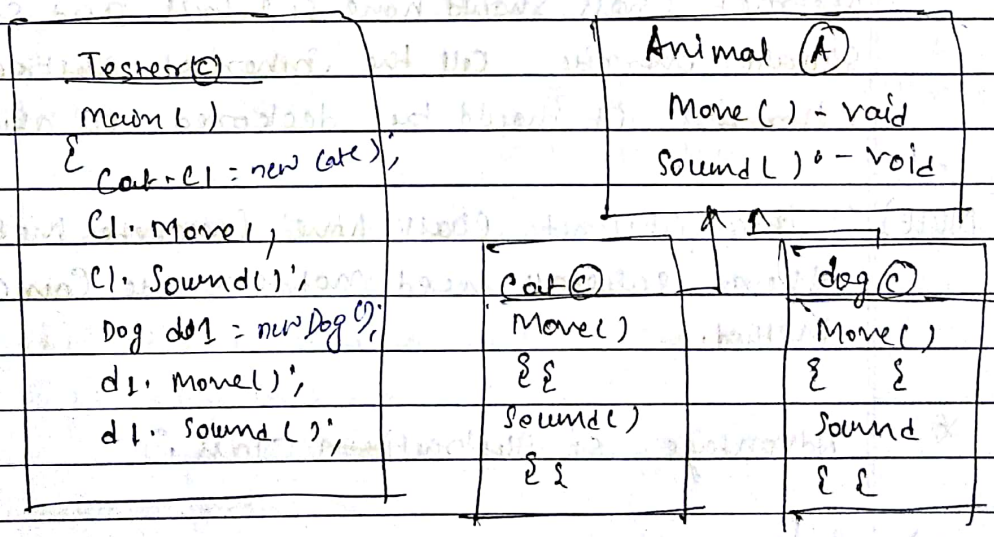
OR

We Can achieve Standersations or Abstractions (from 0 to 100%) using abstract class.

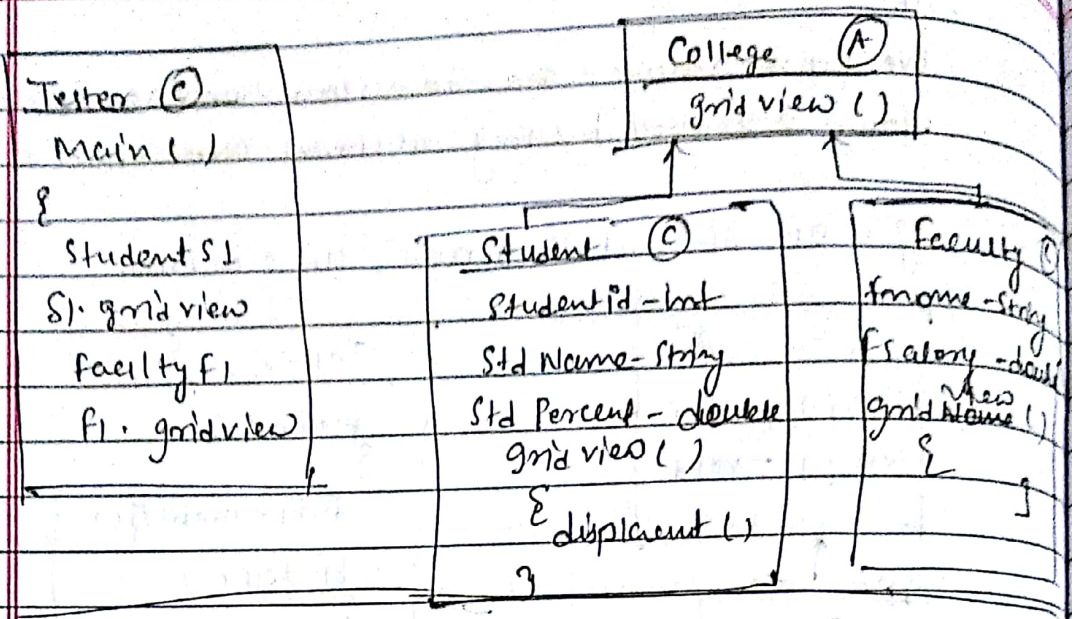
Eg: of Abstract class and its sub class



Steandersisations example:



Abstractions leads to standersisations and simplicity and standarisation leads to achieving polymorphism.



→ In abstract class we can develop
(1) only abstract method
(2) only Concrete method.
(3) (Both) Combination of abstract and Concrete Method.

→ Abstract class should have subclass and subclass should override all the inherited abstract method otherwise it should be declared as abstract.

[NOTE] : if a abstract class have Concrete method then subclass need not override Concrete method.

* Advantage of Abstraction class :

→ Abstract class is not 100% abstract.
→ By using abstract class we can achive upto 100% abstraction
→ Standarisation → Simplicity

Q:1 Can abstract class have Constructor?

Ans: Yes, abstract class will have Constructor to main taining Constructor Chaining.

Q2 Can we declare can abstract method as static?

Ans No, static method cannot be overridden but Abstract method should be overridden somewhere.

Q3 Can we declare abstract method as final?

Ans No, because final method cannot be overridden and abstract method should be overridden.

Q.4 Can we declare can abstract class as final?

Ans No, final class cannot be inherited and abstract class should be overridden.

#) JAVA TYPE

- Class
- Interface
- Enum (Enumeration)
- Annota-tions

① Enumeration: If you want to group the fix number of constants then we will be using enumerations.

e.g: Days in week

Keys of a keyword in keyboard.

General Syntax

enum A

Eg: enum Month { JAN, ..., DEC }

⇒ Through enumeration we can achive uniformity while developing.

(2) Annotations are basically used to give some information to Compiler, Developer and Run-Time environments

e.g: @override @Supress Warning.
@Test @BeforeMethod

* Interface

→ Interface is another Java type through which we can achive Multiple Inheritance in Java.

→ Through Interface we can achive Standarisation and 100% abstraction.

→ Interface is 100% abstract.

→ In Interface we cannot develop Concrete Method. We can develop only abstract Methods.

→ Interface Methods will be automatically abstract and Public.

→ We cannot create object of an Interface.

→ Interface should have a subclass and that subclass should follow two rules/Contracts

(I) the subclass should override all the inherited abstract Method

(II) If the subclass doesnot override the abstract Method then the subclass should be declared as abstract.

→ If a class is inheriting from an Interface then to inheritance we should use implements keyword

→ Interface does not have Constructor.

→ Interface does not inherit from Object class.

→ Interface can have variable. All variable inside interface will be public static final.

Tester
Main
{ sop (E.a)

E
a_{sp} - int

Interface variable cannot be just declared It has to be initialized as it is static (global) final.

Tester
Main ( )
{ B b_1 = new B ( )
b_1. test ( )
b_2. test ( )
{

A (I)
test1 ( ) - void
test ( ) - void

B	C
test2 ( ) - void	
test ( ) - void	

* Marker Interface :

→ Interface without any methods are considered as marker interface.

Purpose of Marker Interface :

→ Using marker interface, we can give some special information to Run-time environment.

→ Example of built in marker interface.
- Clonable
- Serializable

→ In interface all the variable will be public static & final.

```
class G implements Cloneable
{
  PSVM(String[] args)
  { G rv = new G();
    try { rv.clone();
          SOP("Cloning Archived"); }
    Catch(CloneNotSupportedException)
    {
      SOPln("Cloning Not Archived"); }
  }
}
```

→ If a class is inheriting from interface, we should use implements keyword.

→ If an interface is inheriting from another interface we should use extends keywords.

→ If a class is inheriting from another class, we should use extends keywords.

* Difference b/w Abstract class and interface.

	Abstract Class	Interface
①	Abstract Class is not 100% Abstract	① Interface is 100% pure Abstract.
②	Abstract Class will have Constructor	② Interface will not have Constructor.
③	Through Abstract Class Multiple inheritance is not possible.	③ Through interface, multiple inheritance is possible to some extend.
④	In Abstract Class, Abstract Method should be declared with keyword abstract	④ In interface, it is not Mandatory to use use keyword abstract as interface is 100% abstract.

		Now it is portant.
(5)	In Abstract Class, We can develop Static Concrete Method	(5) In interface we can never develop static method.
(6)	Abstract Class will inherit from object Class	(6) Interface will not inherit from object class.
(7)	In Abstract Class, Method will have acess level according to programmer.	(7) In interface acess level of any method will be public automatically.
(8)	In Abstract Class, variable will not be automatically, Public, Static and final it depends on programmer	(8) In interface All the vari-ables will be automatically public static final.
(9)	In abstract class we can just declare a variable and initialize later	(9) Through interface we can achive 100% abstraction. In interface we cannot just declare variable, we should initialize
(10)	In Abstract class we can achive 0 to 100% abstraction	(10) Through interface we can achive 100
(10)	In Abstract class we can achive 0 to 100% Abstraction.	(10) through interface we can achive 100% abstraction.
(11)	If a class inherit from Abstract Class, we should use extends keywords	(11) if a class inherit from Interface, we should use implements keywords.

Type Casting :

⇒ There are two types of Method invocations

① Call by value

: Calling a method by supplying primitive data is called as call by value method invocations

```
test(int) {
    test(40) ... }
```

② Call by reference

: Calling a method by supplying an object

```
test(A, a1)
{   }
        test(new A()), A rr = new A(), test(rr).
```

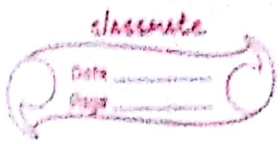
A method can recieve an object as i/p and it can operate at the content of supplied object.

If you want to design develop such type of method, then the method Argument should be class type or interface type or Java type or derived type

e.g: Demo;

```
Class B
{ int i, j;
static void swap(B b1)
{ int temp = b1.i;
  b1.i = b1.j;
  b1.j = temp; }
PSVM (String args)
{ Sopln ("Main starts");
  B.rr = new B();
  rr.i = 10; rr.j = 20
Sopln ("i: " + rr.i + "\nj: " + rr.j);
swap (rr);
Sopln ("==========");
```

```
Class B
{ static void test(B a1)
{ Sopln("from test(): "
      a1.i "); }
PSVM (String [] args)
{ A rr = new A();
  test (rr);
{ test (new A()); }
```



```
Sop(" i : " +rv.i + " in j : " +rv.j)
Sop("Main Ends");
}
}
```

Tester ①	D ②	E ③
Main { E C1 = new A() C1.test() C.test2()	i - int demo() - void { sop;	test(D d) - void static test (D d2) - void

Type Casting :

→ Casting means Conversion

→ Type Casting → Converting / Casting the data / objects from one type to another type.

There are two types of type casting

① Primitive Casting

② object Casting.

① Primitive Casting : Casting the primitive data from one primitive type to another primitive type is called as primitive casting.

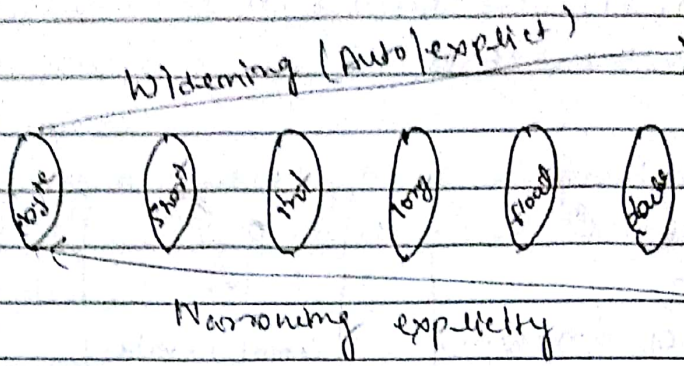
→ Homogenous Statement : Those statements in which there will be no type mismatch.

e.g - int a = 10.5 (CTE)

There are two types of primitive casting

① Widening

② Narrowing.



(1) Widening :
Converting the primitive data from lower type to anyone of the higher type is called as widening
eg: double d = (double) 10;

(a) Auto Widening / Implicit Widening :
Compiler performing the widening operation automatically at compile time is called as auto/implicit Widening eg: double d = 10;

(b) Explicit Widening :
programmer performing the the widening operation while developing the code is called as explicit widening.
eg: double d = (double) 10;

(2) Narrowing : Converting the primitive data from higher type to anyone of the lower type is called as Narrowing.

→ Narrowing is not automatic. Programmer should perform narrowing while developing the code also called as explicit widening
a - int a = (int) 10.5;

[NOTE]: only 6 primitive data types supports primitive casting

```
Main
{
    double re1 = 5/2;        | 2.0
    res1 = 5.0/2;            | 2.5
    res = 5/2.0;             |
    res = (double)5/2;       |
    res = (double)5/2;       |
    Sop(res.... res 5)       |
}
```

```
Main()
{
    int i = 10.9 Narrowing
    Sop(i)
    double d = 10 Widening
    Sop(d)
}
```

[NOTE]: Any operation having double and other 6 primitive data will result in double

```
Tester3
{
    Static void Percent(int, M1, M2, mm)
    {
        double percent;
        Sop percent;
    }
    Main
    Percent(95, 94, 95)
}
```

```
Test 4
{
    Static void test(double d)
    {
        Sop('d' = '' + d)
    }
    Main()
    {
        test()
    }
}
```

```
Class tester {
{
    Static void add(int a, double d)
    {
        Sop("The result is" + (a+d))
    }
    Static void add(double d, int a)
    {
        Sop("The result is" + (a+d));
    }
    PSVM(String[] args)
    {
        add(10.5, 10)
        add(10, 10.5)
        add(10, 10); // CTE - reference to add function's ambiguous
    }
}
```

```
float f = 10.5 f;
float d = (float)10.5;
```

Similarly for Integer format, default will be int 4, 10, 248, 999497318.

```
Main
{ Sopln (Integer, Max_value)
  Sop (999497); CTE Number too long
  Sopln (999497318)
{
  Class Tester {
{
  Static void test (int a)
{ Sopln ("from test (int a)"); }
  Static void test (byte b)
  {
   Sopln ("From test (byte b)"); }
   psvm (String[] args)
  {
    test (90);
    test (9);
    test (byte) 90);
  }
```

⇒ Widening can happen at the time of method invocation.

* Char to Int Conversion :

```
Class Testers 12
{
  PSVM ( String[] Args )
  {
    Char Ch1 = 'A';
    Sopln (Ch1);
// Char Ch2 = '90'; CTE
    Char Ch3 = 90; Sopln(Ch3)
    int a = 'z';
    Sopln (a);
    int b = '#';
    Sop.ln (b)
    Sopln ('A' + 'B')
  }
}
```

A

z

90 → A

35 → Z

→ 90

(35)

→ 35

131

** output $a^2 + b^2$ 253 ASCII

```
Method (double)
{{
  Method (10)
```

```
double method()
{
  return 10;
}
```

```
int read()
{
  return 'z';
}
```

OBJECT CASTING

- → Inheritance
- → downcasting
- ↓ UP casting

→ Casting the object of one class type to another class type or inherface type.

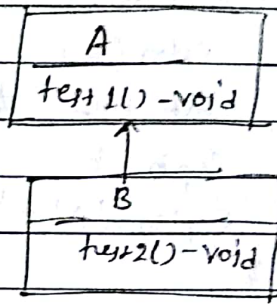
→ To perform object casting inheritance is mandatory

→ There are two types of object Casting
① Upcasting
② downcasting

→ Through object Casting we can achive
① Runtime polymorphism.
② We can design graphic functions which can recive any input.

Upcasting example:

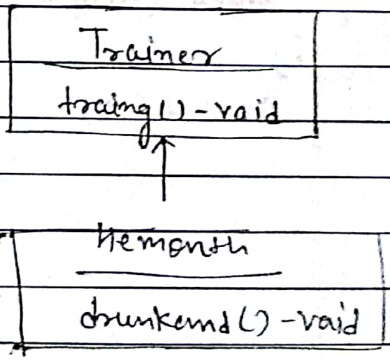
```
A a1 = A; new B()
a1.test();
a2.test2() //CTE
```



→ Subclass feature will be hidden in case of upcasting only superclass features can be accessed.

→ Upcasting is Automatic.

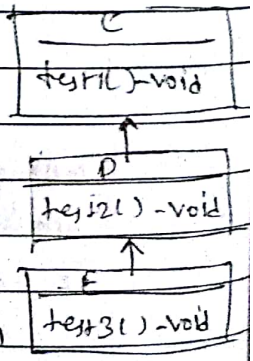
```
Trainer t1 = new Hemanth()
t1.training();
t1.download(); //CTE
```



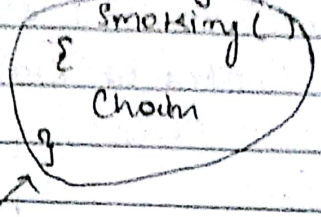
Auto-Casting Implied

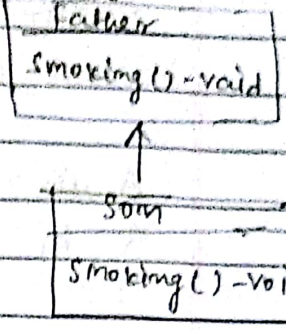
Upcasting in Multilevel inheritance

```
test1()                    test1()
test2()                    test2()
test3() →Hidden            test3()
d1 = new E()               C c1 = new E()
```



*** Method overridding in case of Upcasting

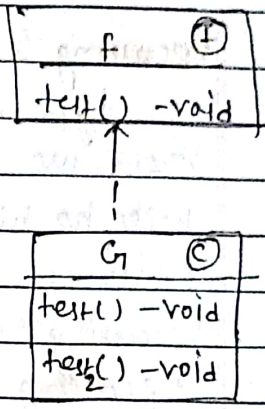




Father f1 = new Son

→ Only Latest implementation is available.

* Upcasting with Interface



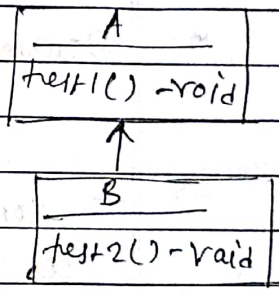
test()',
{
{

test2()
} }

→ hidden

f f1 = new G()

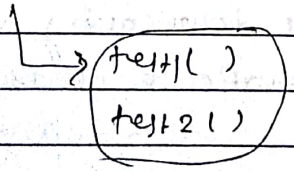
* Downcasting :



test1()
test2()
→ hidden

A a1 = new B() // Upcasting

B b1 = (B) a1 ; // explicit downcasting



→ both visib No

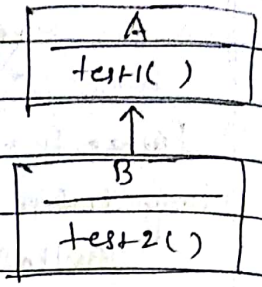
b1.test()

b1.test2()

→ downcasting is performed on upcasted ~~reference~~ reference variable, it is explicit Not implicit/Auto.

→ Direct downcasting is not possible
Runtime Error / Exception → ClassCast Exception
because objects are created at Runtime.

```
A a1 = new A();
B b1 = (B) a1; // CTE...
```



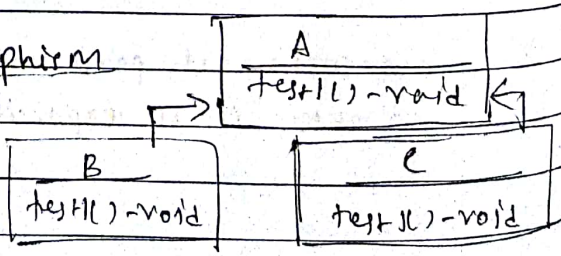
→ Upcasting : Converting subclass object into Superclass type is called as upcasting.
→ When we perform upcasting, subclass features will be hidden, we can access only super class features
→ Upcasting is automatic i.e Auto-Upcasting / Implicit Upcasting.
→ In case of upcasting we can access overriden Methods but we will get latest implementation.

* Downcasting : Converting Superclass type object into subclass type is called as downcasting

→ To access the hidden features due to upcasting, we will perform downCasting.
→ Direct downcasting is not possible.
→ Direct downcasting means downcasting without Upcasting
→ Downcasting is not automatic so, it is also called as explicit downcasting.

Polymorphism

```
A a1 = new A()
    test1()
{ B class
};
```



A a2 = new B()

test()
{
c class
}
A a3 = new (c)

```
PSVM (String [] Args)
{
My class .demo (new A());
My class . demo (new B());
My class .demo (new C());
}
```

```
My class
Static Demo (A a1) -void
{
a1 . test()
}
```

Usb Port (I)
read() -void
write() - void

Pendrive (1)
read(i) -void
{}
write() - void
{}

Mobile
read() - void
{}
write() - void
{}

MyClass (C)
Static My Method (UsbPort driver) -void
{ driver . read();
driver . Writer();

```
psvm (String [] args)
{ My class.
My method (new Pendrive);
My Class . My method
(new Mobile());
}
```

WebDriver driver = new FirefoxDriver();
→ WebDriver is an Interface
→ driver is a reference variable of the Interface WebDriver.
→ new is an operator.
→ FirefoxDriver() is Constructor of FirefoxDriver class
→ Auto upcasting is done in above line.