OBJECT ORIENTED PROGRAMMING IN JAVA

OBJECT ORIENTED PROGRAMMING:

OOP is a programming paradigm which delas with the concept of object to build programs and software application. OOP such as programming that allows a mode of modularizing programs by forming separate memory area for data as well as function that is used as object for making copy of module as per the requirement. OOP refers to languages that uses objects in programming. The aim of Object Oriented Programming aims to implement real world problem solving.

ELEMENTS IN OOP:

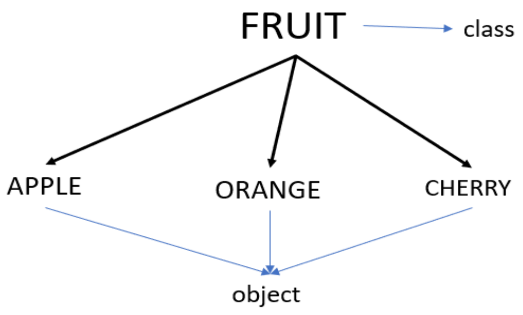
1. CLASS
2. OBJECT
3. METHOD
4. ABSTRACTION
5. INHERITANCE
6. ENCAPSULATION
7. BINDING
8. POLYMORPHISM
9. DELEGATION
10. MESSAGE PASSING

1. CLASS:

A class is the collection of similar object having common behavior and properties. A class is an user define datatype which is the collection of dissimilar of data and functions, method a class is a template or framework using which objects are created for a single class, there can be any number of objects. Class is defined as blueprint from an object. The description of a number of similar objects is also called a class. Classes are logical in nature. A class is also defined as a new data type, a user-define type which contains two things data members and methods.

2. OBJECT:

An object is the instance of a class. An object is a the basic run time entity in the object oriented system. The object may represent a person, place, a bank account or any items the program can handle. Objects of a class will have same attributes and behavior which are defined in that class. the only difference between objects would be the value of attributes, which may vary. Objects in real life as well as programming can be physical, conceptual or software. Objects have unique identity, state and behavior. Attributes define the data for an object. The response of an object when subjected to simulation is called its behavior. Behavior defines what can be done with the objects and may manipulate the attributes of an object. Behavior actually determines the way an object interacts with other objects.



3. METHOD:

It defines the ability of an object it is an entity through which can object interact with the external environment. If dog is an object then bark, eat, run are method of dog. It is similar to a function in any other programming language. None of the methods can be declared outside the class. all methods have a name that starts with lowercase character.

4. ABSTRACTION:

It is the process of representing essential features without including the unnecessary into a single unit. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. The objects that help to perform abstraction are encapsulated.

5. INHERITANCE:

It is the process of in which a new class created by inheriting of property and behavior of one or more existing classes. The existing class is known as base class/parent class. The newly created class is known as derived class/child class/sub class. inheritance is the way to adapt the characteristics of one class into another class. When a class inherits another class it has all the properties of the base class and it adds some new properties of its own. Inheritance aids in reusability. When we create a class, it can be distributed to other programmers which they can use in their programs is called reusability. A programmer can use a base class with or without modifying it. He can derive a child class from a parent class and then odd some additional features to his class.

TYPES OF INHERITANCE:

#. SINGLE INHERITANCE

#. MULTIPLE INHERITANCE

#. MULTILEVEL INHERITANCE

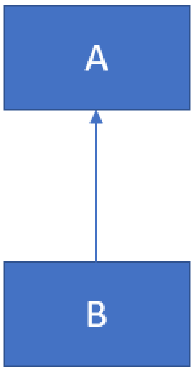
#. HIERARCHICAL INHERITANCE

#. HYBRID INHERITANCE

#. MULTIPATH INHERITANCE

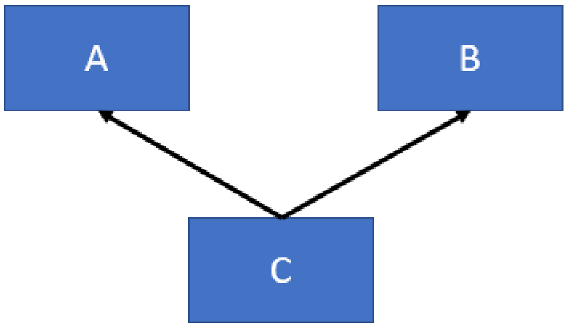
#. SINGLE INHERITANCE:

When new class is created by inheriting the properties and behavior of only one existing class then it is known as single inheritance mechanism.



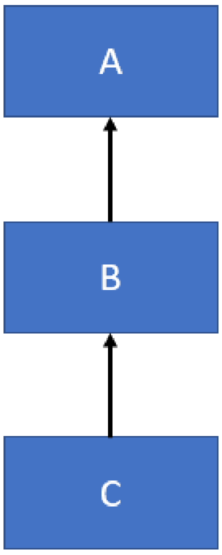
#. MULTIPLE INHERITANCE:

When new class is created by inheriting the property and behavior of two or more existing classes then it known as multiple inheritance mechanism is not supported java.



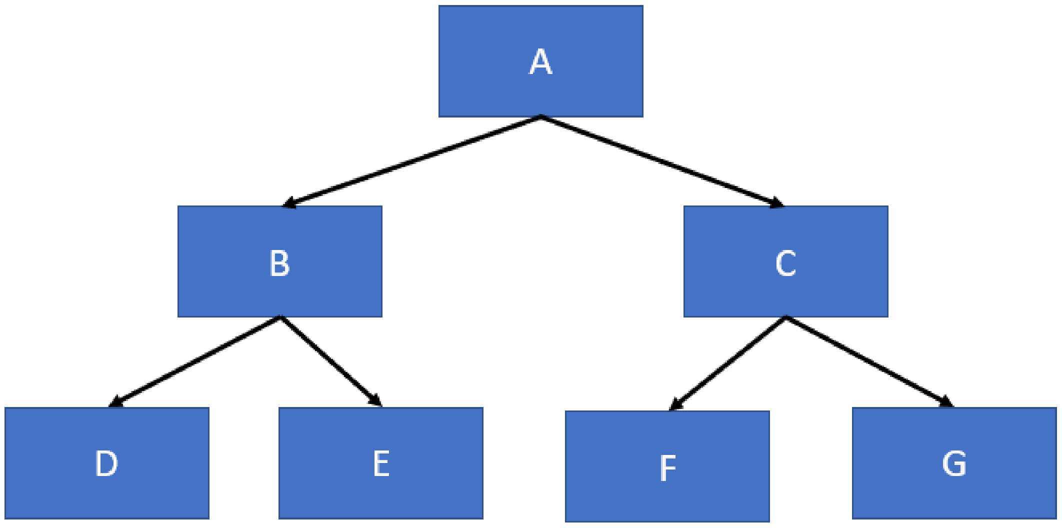
#. MULTILEVEL INHERITANCE:

When new class is created or derived from another derived class then it is multilevel inheritance.



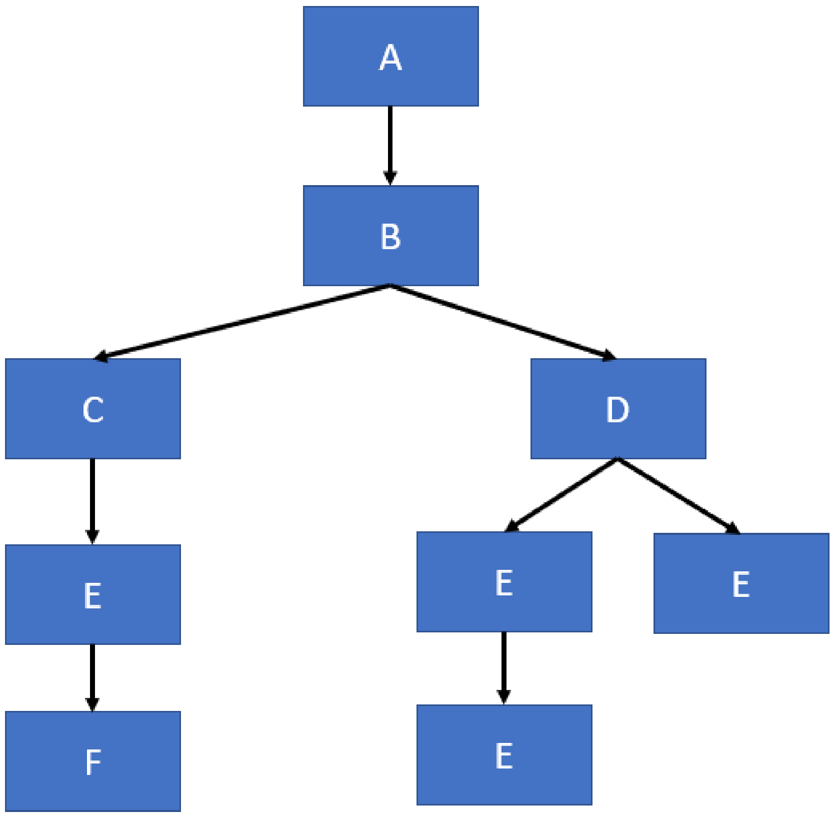
#. HIERARCHICAL INHERITANCE:

When classes are created or derived like tree structure then it is known as hierarchical inheritance mechanism.



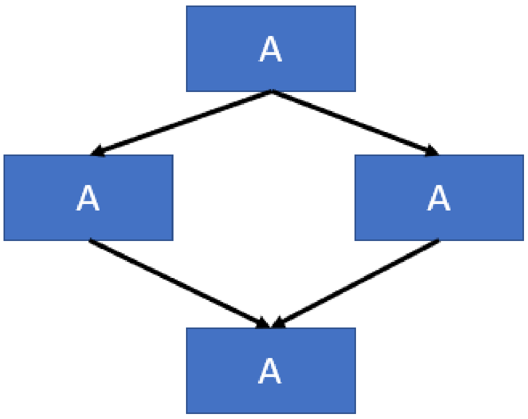
#. HYBRID INHERITANCE:

When classes are created by applying more than one inheritance mechanism then it is known as the hybrid inheritance.



#. MULTIPATH INHERITANCE:

When new class is created from those classes which are having the same parent class then such inheritance mechanism is known as multipath inheritance.



6. ENCAPSULATION:

It is the process of combining the data member and member function into a single unit. The process of binding the data procedures into objects to hide them from the outside world is called as encapsulation. Encapsulation is also called as data hiding. It provide to restrict anyone from directly altering the data. The data is hidden from the outside world and as a result it is protected. The details that are not useful for other objects should be hidden from them is called as Encapsulation.

7. DYNAMIC BINDING:

It is the process of connecting one program to another that is to be executed in replay to a call. When type of the object is determined at compiled time(by the compiler), it is known as static binding. If there is any private, final or static method in a class, there is static binding. When type of the object is determined at compile time is known as static binding. When type of the object is determined at run-time, it is known as dynamic binding.

The binding which can be resolved at compile time by the compiler is known as static or early binding. The binding of all the static, private, and final methods is done at compile-time. In Dynamic binding compiler doesn't decide the method to be called. Overriding is a perfect example of dynamic binding. In overriding both parent and child classes have the same method.

Static Binding	Dynamic Binding
It takes place at compile time for which is referred to as early binding	It takes place at runtime so do it is referred to as late binding.
It uses overloading more precisely operator overloading method	It uses overriding methods.
It takes place using normal functions	It takes place using virtual functions
Real objects never use static binding	Real objects use dynamic binding.

8. POLYMORPHISM:

The word polymorphism comes form word ploy(many) and morphos(forms). Polymorphism is the process and mechanism in which the same entity behaves differently in different situation. The (+) operator in java is however an exception as it can be used for addition of two integers as well as concatenation of two Strings or an integer with a String.

TYPES OF POLYMORPHISM:

#. STATIC POLYMORPHISM /COMPILE-TIME/EARLY BINDING:

When polymorphism is achieved during compile-time then it known as static polymorphism. Example: function overloading, operator overloading, constructor overloading.

#. DYNAMIC POLYMORPHISM/RUN-TIME/LATE BINDING:

When polymorphism is achieved during run-time & execution time then it known as dynamic polymorphism. Example: function overriding.

9. DELEGATION:

When object of one class will act as the data member[variable] in another class then such composition of object is delegation. Delegation is simply passing a duty off to someone/something else. Delegation can be an alternative to inheritance. Delegation means that you use an object of another class as an instance variable, and forward messages to the instance. Delegation can be viewed as a relationship between objects where one object forwards certain method calls to another object, called its delegate.

10. MESSAGE PASSING:

The communication between the object is referred as message passing. Message passing in Java is like sending an object i.e. message from one thread to another thread. It is used when threads do not have shared memory and are unable to share monitors or semaphores or any other shared variables to communicate.

O.O.P “CLASS, OBJECT & METHOD”

Software blueprint for objects are called classes. A class is a blueprint or prototype that defines the variables and methods common to all objects of a certain kind. In other words a class can be thought of a user-defined data type and an object as a variable of that data type that can contain data and methods, i.e., functions working on that data.

Class is a custom data type. class as nothing but template of object or a plan of object. Using OOP we can depict real life scenario in our programing using objects. Object has properties and functionalities.

Objects has states and behavior. ***State***: it is the property or variables declared inside a class. ***Behavior***: it is the function or methods created inside a class.

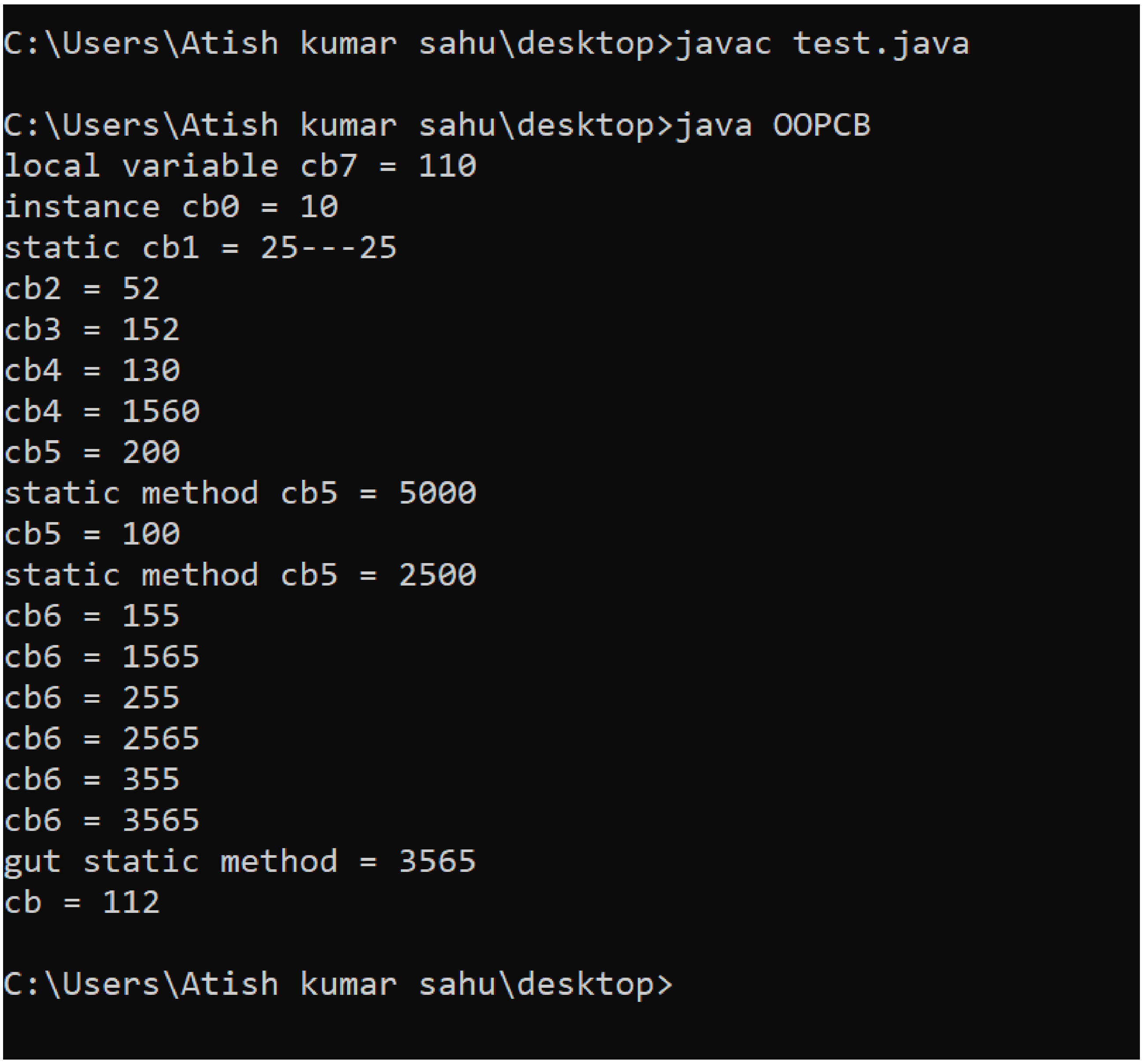
Classes and object can depict the real life objects. An object is a software bundle that encapsulate variables and methods operating on those variables.

Method is very common word used in OOP. It is similar to function. None of the methods can be declared outside the class all methods have a name that starts with lowercase character. Methods are used to make code reusable and simplify. There are two types of method ***Instance method***: for this we have to create object. ***Static/class method***: for this we don’t need object creation.

```
class OOPCB
{

int cb0 = 10; //instance variable
static int cb1 = 25; //static variable
int cb = 112;
void get(int cb2) //instance method & formal parameter cb2 = local variable
{
int cb3 = cb2 + 100; //cb3 = local variable
System.out.println("cb2 = "+cb2);
System.out.println("cb3 = "+cb3);
}

int got(int cb4)   //return type instance method & formal parameter cb4 = local variable
{
System.out.println("cb4 = "+cb4);
cb4 = cb4 * 12;
System.out.println("cb4 = "+cb4);
return cb4;
}
```

```java
public static void git(int cb5) //static method & formal parameter
{
System.out.println("cb5 = "+cb5);
System.out.println("static method cb5 = "+cb5 * 25);
}

public static int gut(int cb6) //return type static method and formal parameter
{
System.out.println("cb6 = "+cb6);
cb6 = cb6 * 10 + (20-5);
System.out.println("cb6 = "+cb6);
return cb6;
}
```

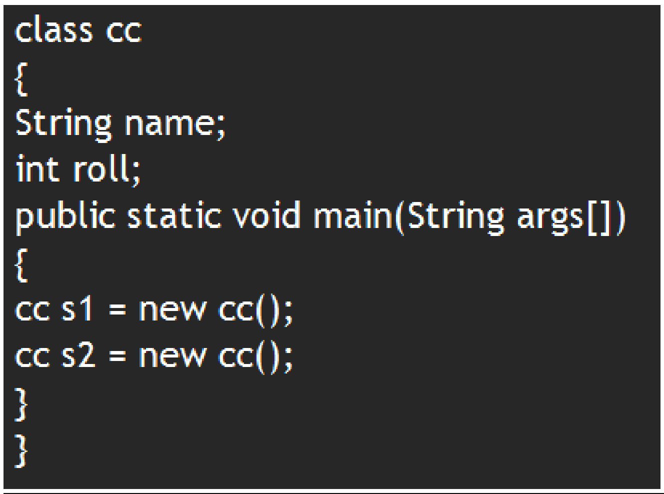
```java
public static void main(String args[])
{
int cb7 = 110; // local variable
System.out.println("local variable cb7 = "+cb7);

OOPCB oo = new OOPCB();
System.out.println("instance cb0 = "+oo.cb0);
System.out.println("static cb1 = "+oo.cb1+"---"+OOPCB.cb1);

OOPCB oo1 = new OOPCB();
oo1.get(52); // actual parameter
oo1. got(130); // actual parameter

OOPCB oo2 = new OOPCB();
oo2.git(200); //actual parameter
OOPCB.git(100); //actual parameter

oo2.gut(155); //actual parameter
OOPCB.gut(255); //actual parameter
System.out.println("gut static method = "+gut(355));

OOPCB oo3;
oo3 = new OOPCB();
System.out.println("cb = "+oo3.cb);
}
}
```



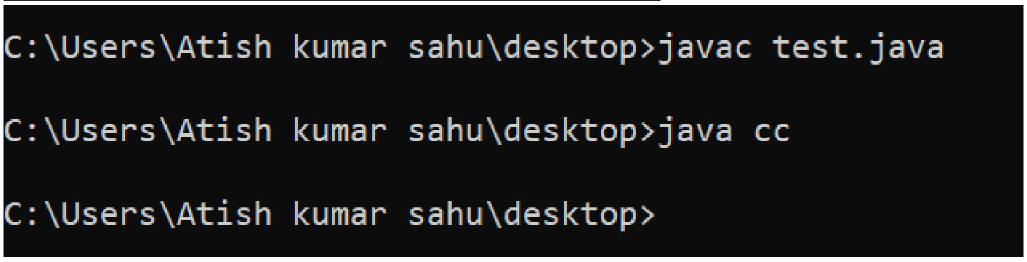
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java OOPCB
local variable cb7 = 110
instance cb0 = 10
static cb1 = 25---25
cb2 = 52
cb3 = 152
cb4 = 130
cb4 = 1560
cb5 = 200
static method cb5 = 5000
cb5 = 100
static method cb5 = 2500
cb6 = 155
cb6 = 1565
cb6 = 255
cb6 = 2565
cb6 = 355
cb6 = 3565
gut static method = 3565
cb = 112

C:\Users\Atish kumar sahu\desktop>
```

CONSTRUCTOR:

CONSTRUCTOR:



```
class cc
{
String name;
int roll;
public static void main(String args[])
{
cc s1 = new cc();
cc s2 = new cc();
}
}
```

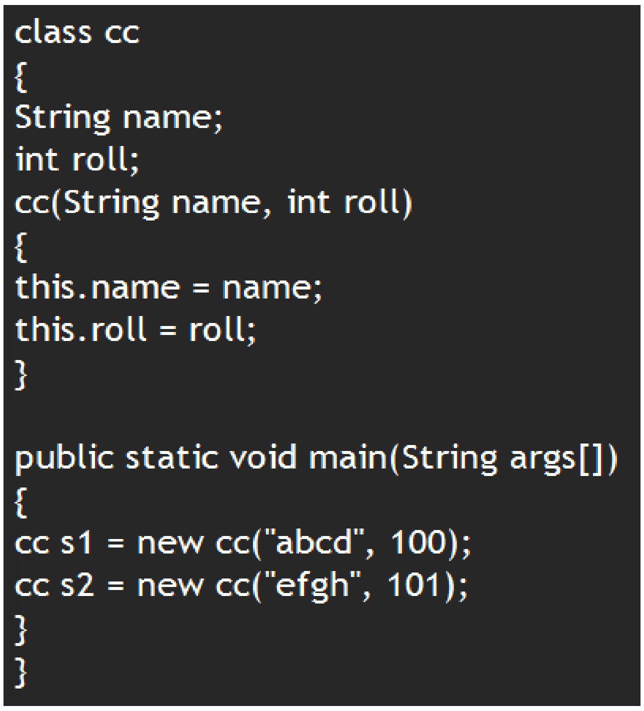


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java cc

C:\Users\Atish kumar sahu\desktop>
```

Once we creates an object compulsory we should perform initialization then only the object is an position to respond properly.

Whenever we are creating an object some piece of the code will be executed automatically to perform initialization of the object this piece of the code is nothing but constructor. Hence the main purpose of the constructor is to perform initialization of an object. To create an object we use "*new*" operator but for initialization "*constructor*" is there.

THE MAIN PURPOSE OF CONSTRUCTOR IS TO PERFORM INITIALIZATION OF AN OBJECT BUT NOT TO CREATE OBJECT.

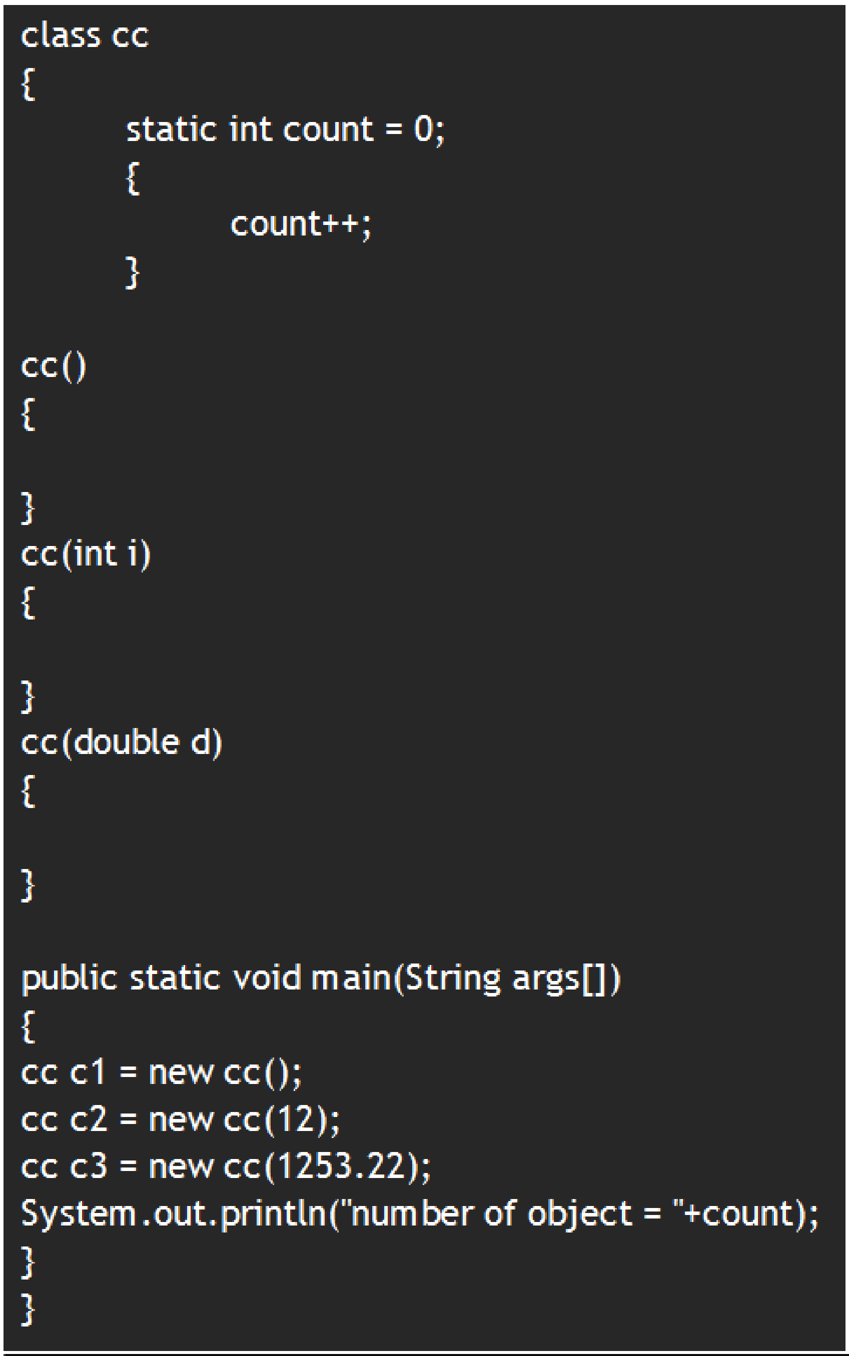


```
class cc
{
String name;
int roll;
cc(String name, int roll)
{
this.name = name;
this.roll = roll;
}

public static void main(String args[])
{
cc s1 = new cc("abcd", 100);
cc s2 = new cc("efgh", 101);
}
}
```

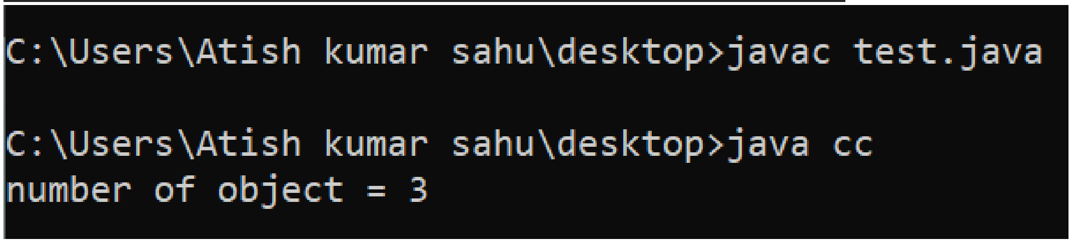
DIFFERENCE BETWEEN CONSTRUCTOR & INSTANCE BLOCK:

The main purpose of constructor is to perform initialization of an object. But other than initialization if we want to perform any activity for every object creation then we should go for instance block(like updating one entry in database for every object creation, incrementing count value for every object creation etc.). both constructor and instance block have their own different purposes and replacing one concept with another concept may not work always. Both constructor and instance block will be executed for every object creation but instance block first followed by constructor next.

Q. DEMO PROGRAM TO PRINT NUMBER OF OBJECTS CREATED FOR A CLASS? ******



```java
class cc
{
        static int count = 0;
        {
                count++;
        }

cc()
{

}
cc(int i)
{

}
cc(double d)
{

}

public static void main(String args[])
{
cc c1 = new cc();
cc c2 = new cc(12);
cc c3 = new cc(1253.22);
System.out.println("number of object = "+count);
}
}
```



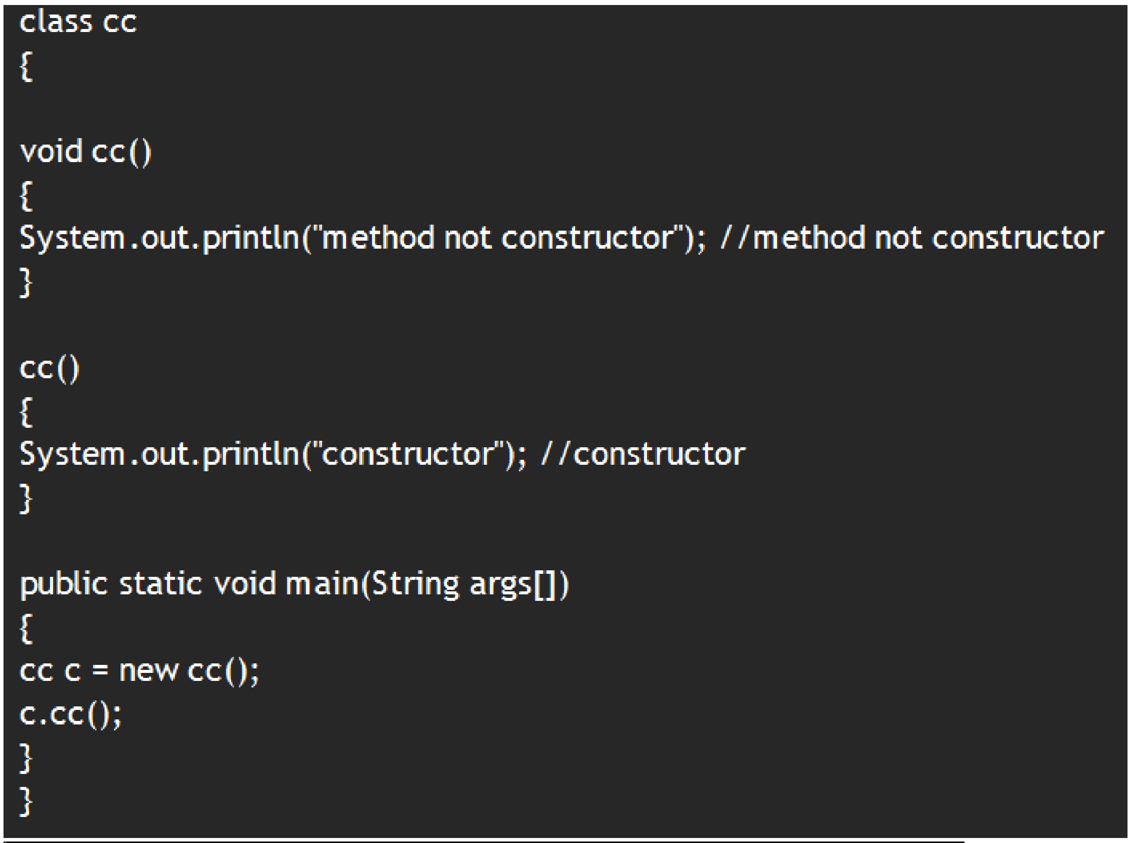
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java cc
number of object = 3
```

RULES OF WRITING CONSTRUCTORS:

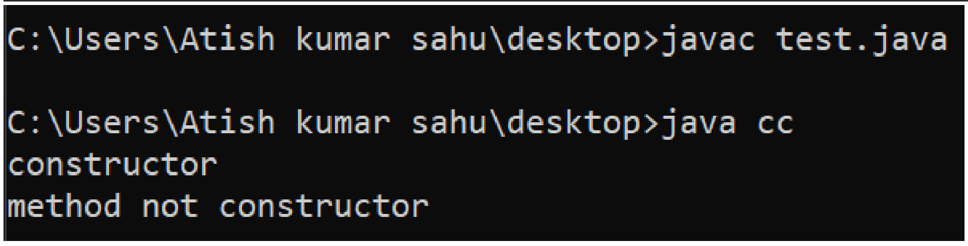
1. name of the class and name of the constructor must be matched.

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

3. by mistake if we are trying to declare return type for the constructor then we won't get any compile time error because compiler treats it as a method.



```
class cc
{

void cc()
{
System.out.println("method not constructor"); //method not constructor
}

cc()
{
System.out.println("constructor"); //constructor
}

public static void main(String args[])
{
cc c = new cc();
c.cc();
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java cc
constructor
method not constructor
```

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

4. the only applicable modifiers for constructors are public, private, protected, default. If we are trying to use anyother modifier we will get compile time error saying "modifier ___ not allowed here."

5. compiler is responsible to generate default constructor but not JVM. If we are not writing any constructor then only compiler will generate default constructor. That is if we are writing atleast one constructor then compiler won't generate default constructor. Hence every class in java can contain constructor it may be default constructor generated by compiler or customised constructor explicitly provided by programmer but not both simultaniously.

PROTOTYPE OF DEFAULT CONSTRUCTOR:

It is always no-arg constructor. The access modifier of default constructor is exactly same as access modifier of class (this rule is only applicable for public and default). it contains only one line “super();”. It is a no argument call to super class constructor. Every no argument constructor is not default constructor but default constructor is always no argument constructor because default constructor provided by compiler.

PROGRAMMER’S CODE

```
class test
{
}
```

COMPILER’S CODE

```
class test
{
        test()
        {
        super();
        }
}
```

PROGRAMMER’S CODE

```
public clas test
{
}
```

COMPILER’S CODE

```
public class test
{
        public test()
        {
                super();
        }
}
```

PROGRAMMER’S CODE

```
public class test
{
        void test()
        {

        }
}
```

COMPILER’S CODE

```
public class test
{
        public test()
        { //default costructor
        super();
        }
        void test() { }
}
```

```
class test
{
        test()
        {

        }
}
```

```
class test
{
        test()
        {
                super();
        }
}
```

```
class test
{
        test(int i)
        {
                super();
        }
}
```

```
class test
{
        test(int i)
        {
                super();
        }
}
```

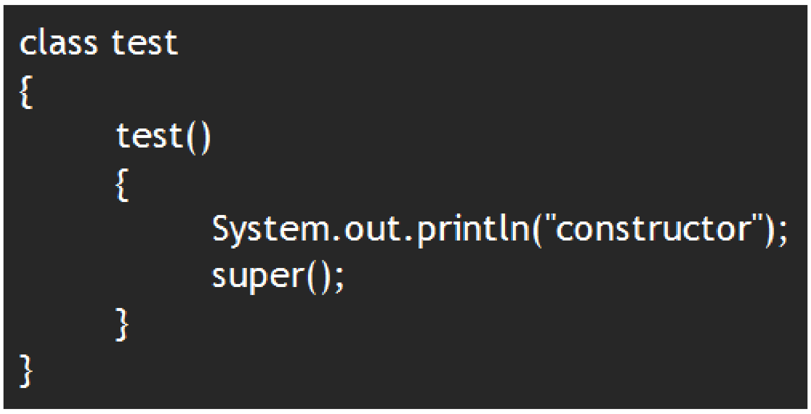
```
class test
{
        test()
        {
                this(10);
        }
        test(int i)
        {
        }
}
```

```
class test
{
        test()
        {
                this(10);
        }
        test(int i)
        {
                super();
        }
}
```

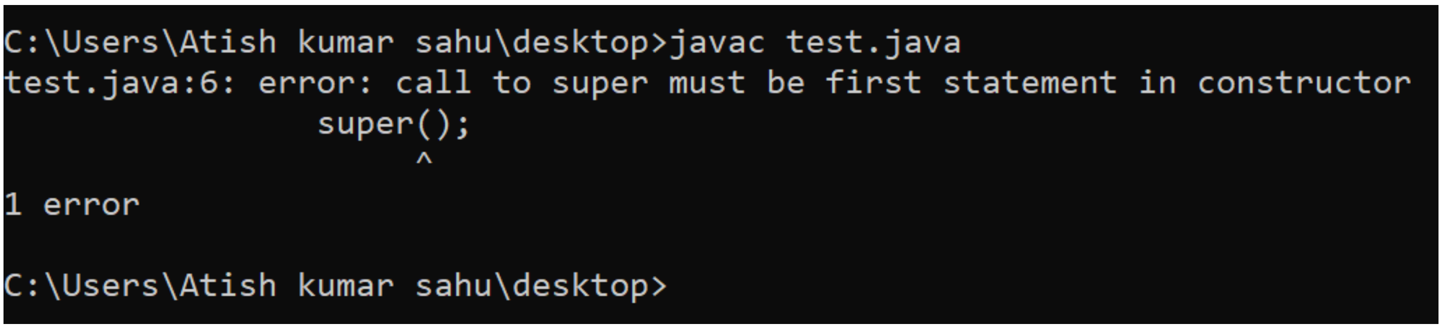
The first line inside every constructor should be either “super()” or “this()” and if we are not writing anything then compiler will always place “super();”

CASE 01:

We can’t take super() or this() only in first line of constructor. If we are trying to take anywhere else we will get compile time error. Saying “*call to super must be first statement in constructor*”



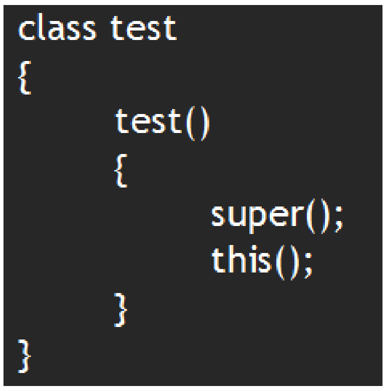
```
class test
{
	test()
	{
		System.out.println("constructor");
		super();
	}
}
```



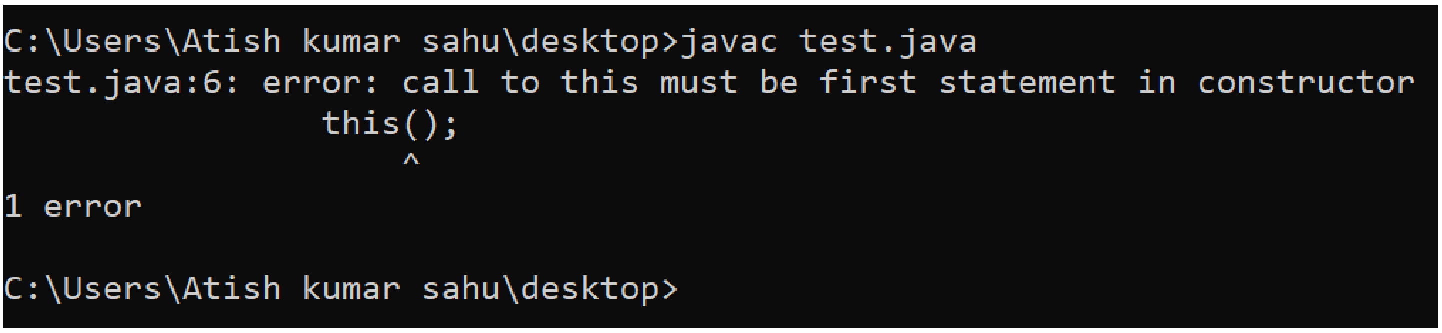
```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:6: error: call to super must be first statement in constructor
                super();
                     ^
1 error

C:\Users\Atish kumar sahu\desktop>
```

CASE 02:

With in the constructor we can take either super() or this() but not both simultaniously.



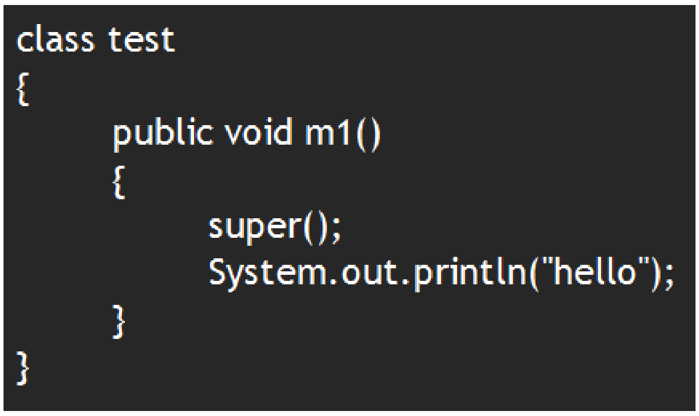
```
class test
{
	test()
	{
		super();
		this();
	}
}
```



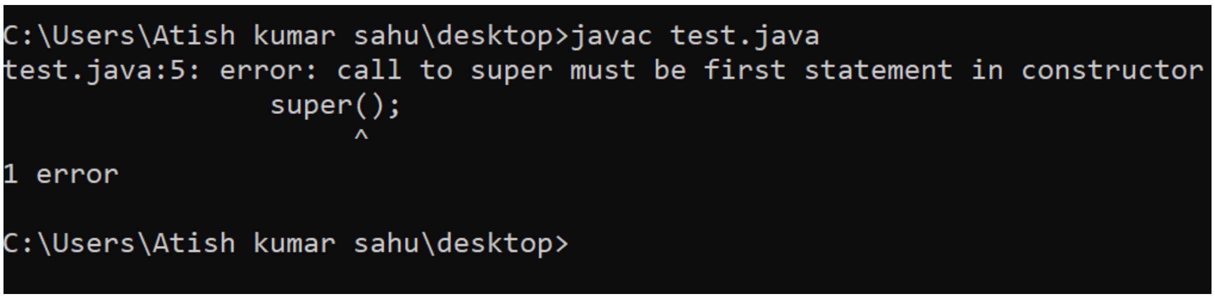
```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:6: error: call to this must be first statement in constructor
                this();
                    ^
1 error

C:\Users\Atish kumar sahu\desktop>
```

CASE 03:

We can use super() or this() only inside a constructor if we are trying to use outside of constructor we will get compile time error. That is we can call a constructor directly from another constructor only.



```
class test
{
	public void m1()
	{
		super();
		System.out.println("hello");
	}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:5: error: call to super must be first statement in constructor
                super();
                     ^
1 error

C:\Users\Atish kumar sahu\desktop>
```

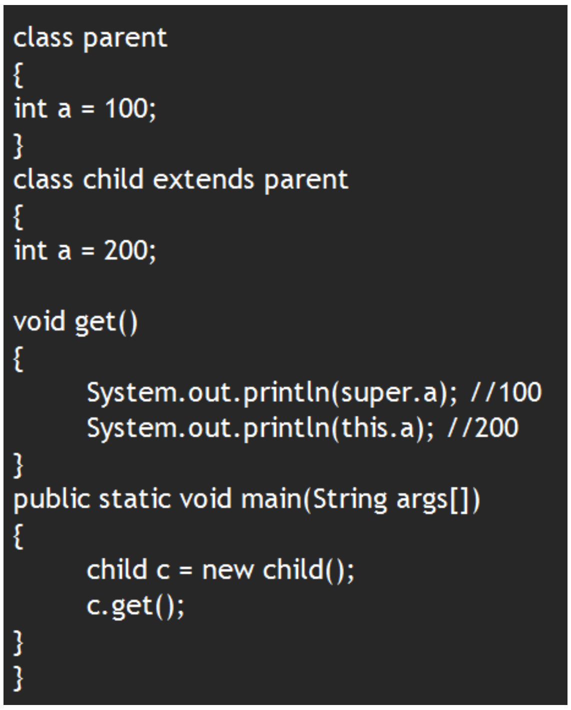
SUPER(); & THIS();

We can use only in constructor. Only in first statement. Only one but not both simultaneously.

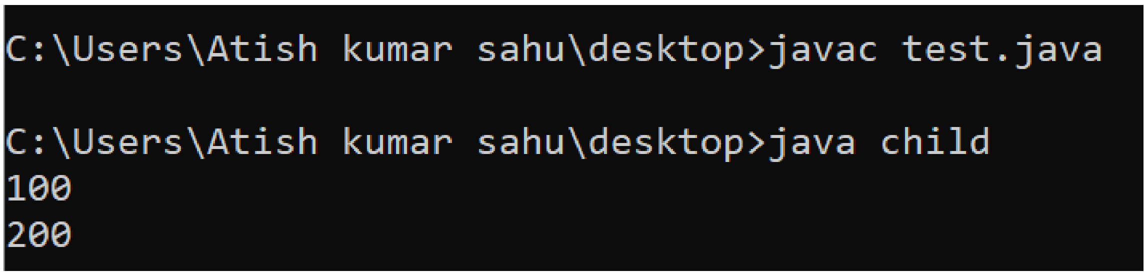
SUPER(), THIS() VS SUPER, THIS:

Super() & this() are constructor calls to call super class and current class constructor.

Super & this are keywords to refer super class and current class instance members.



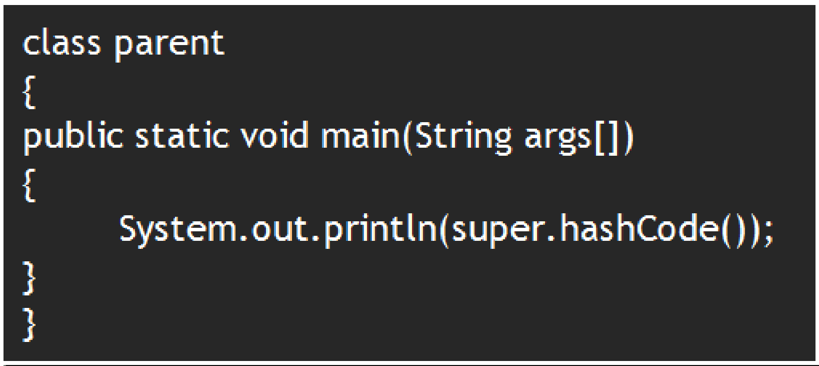
```
class parent
{
int a = 100;
}
class child extends parent
{
int a = 200;

void get()
{
	System.out.println(super.a); //100
	System.out.println(this.a); //200
}
public static void main(String args[])
{
	child c = new child();
	c.get();
}
}
```



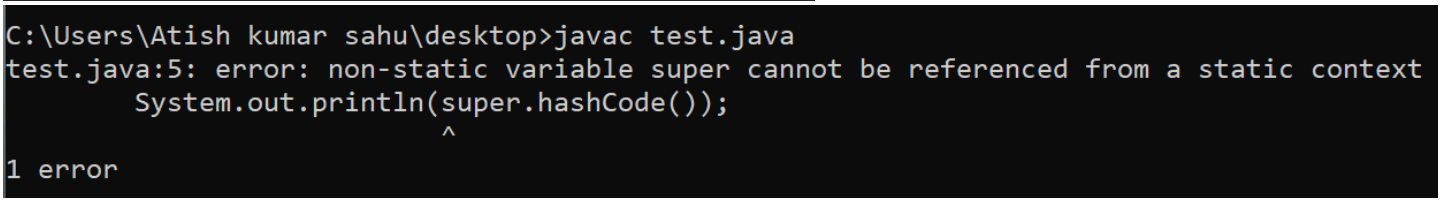
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java child
100
200
```

Super() and this() can use only in constructor as first line.

Super and this we can use anywhere except static area.



```
class parent
{
public static void main(String args[])
{
        System.out.println(super.hashCode());
}
}
```

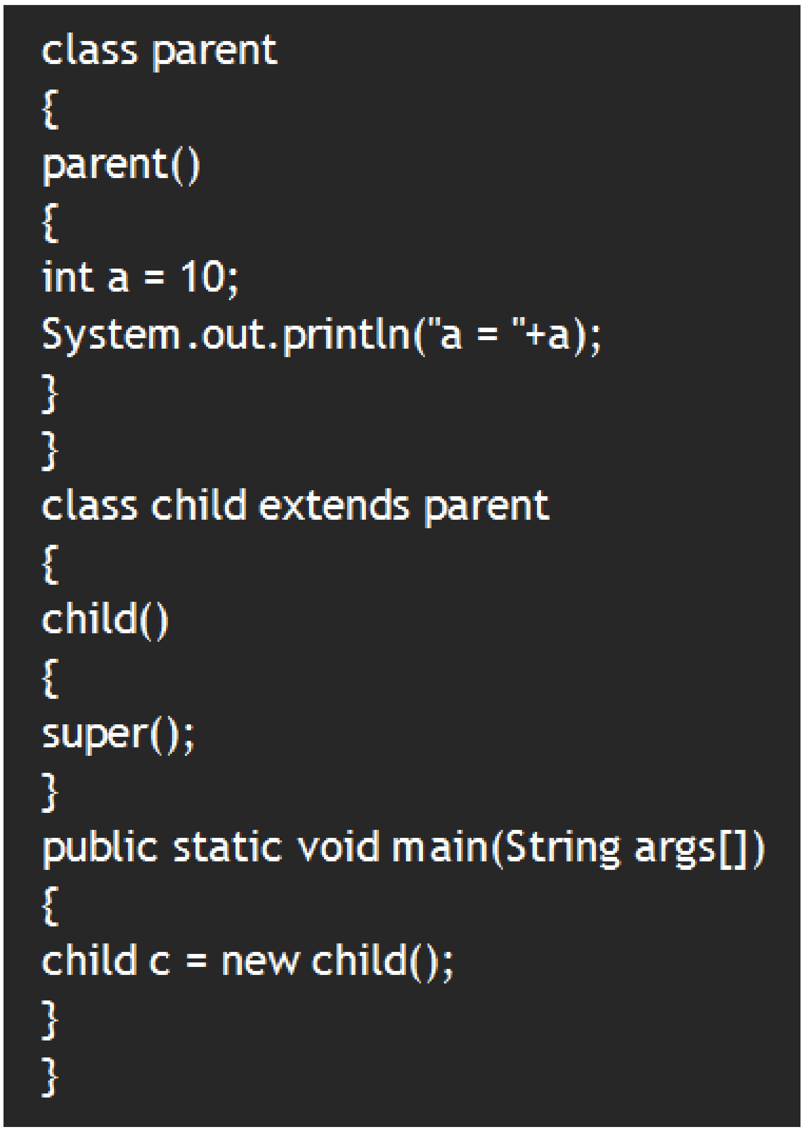


```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:5: error: non-static variable super cannot be referenced from a static context
        System.out.println(super.hashCode());
                           ^
1 error
```

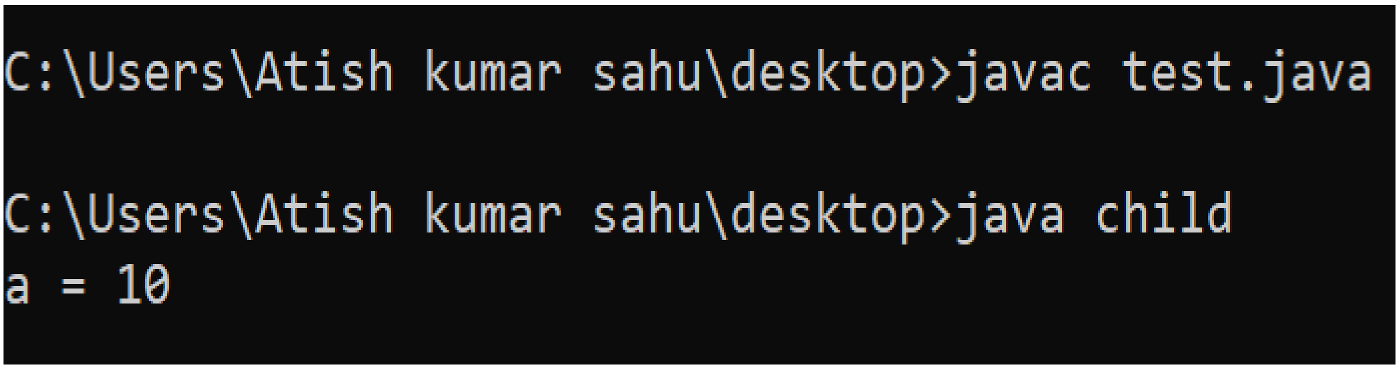
Super() and this() can use only once in constructor.

Super and this can use any number of times.

SUPER():

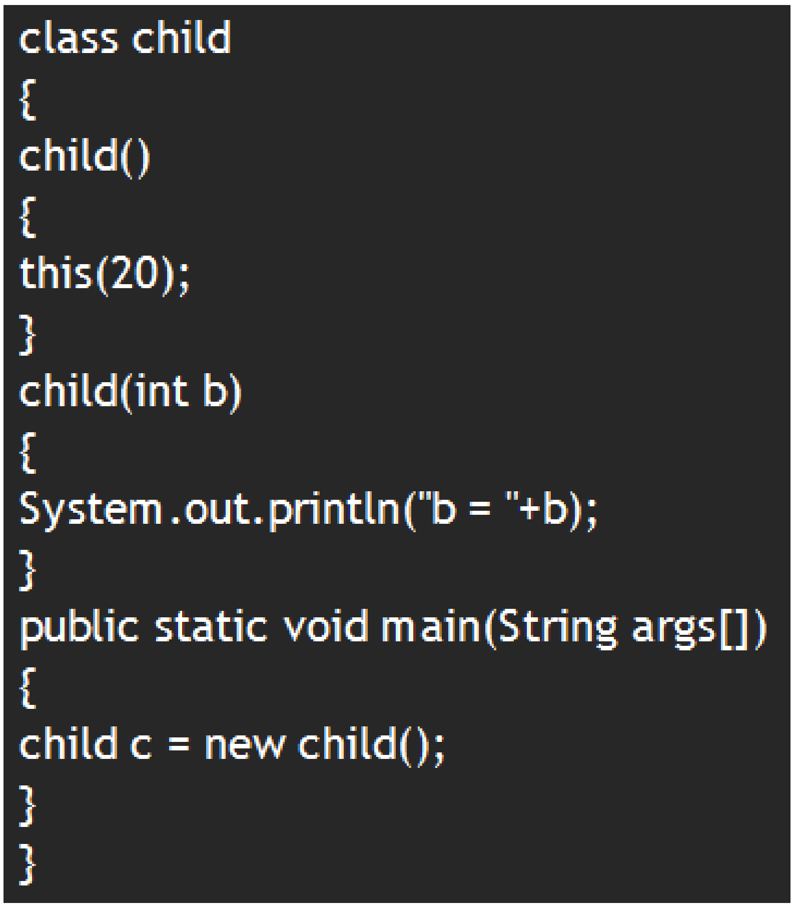


```
class parent
{
parent()
{
int a = 10;
System.out.println("a = "+a);
}
}
class child extends parent
{
child()
{
super();
}
public static void main(String args[])
{
child c = new child();
}
}
```

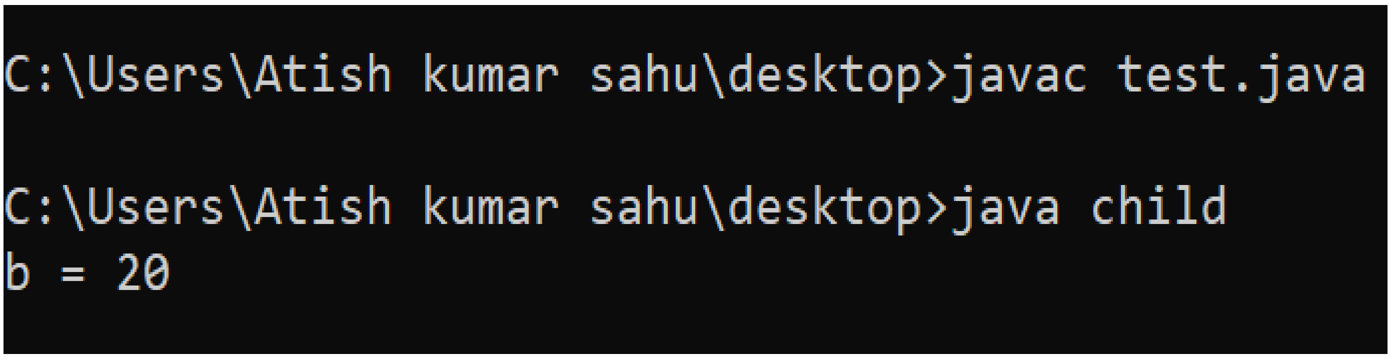


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java child
a = 10
```

THIS():



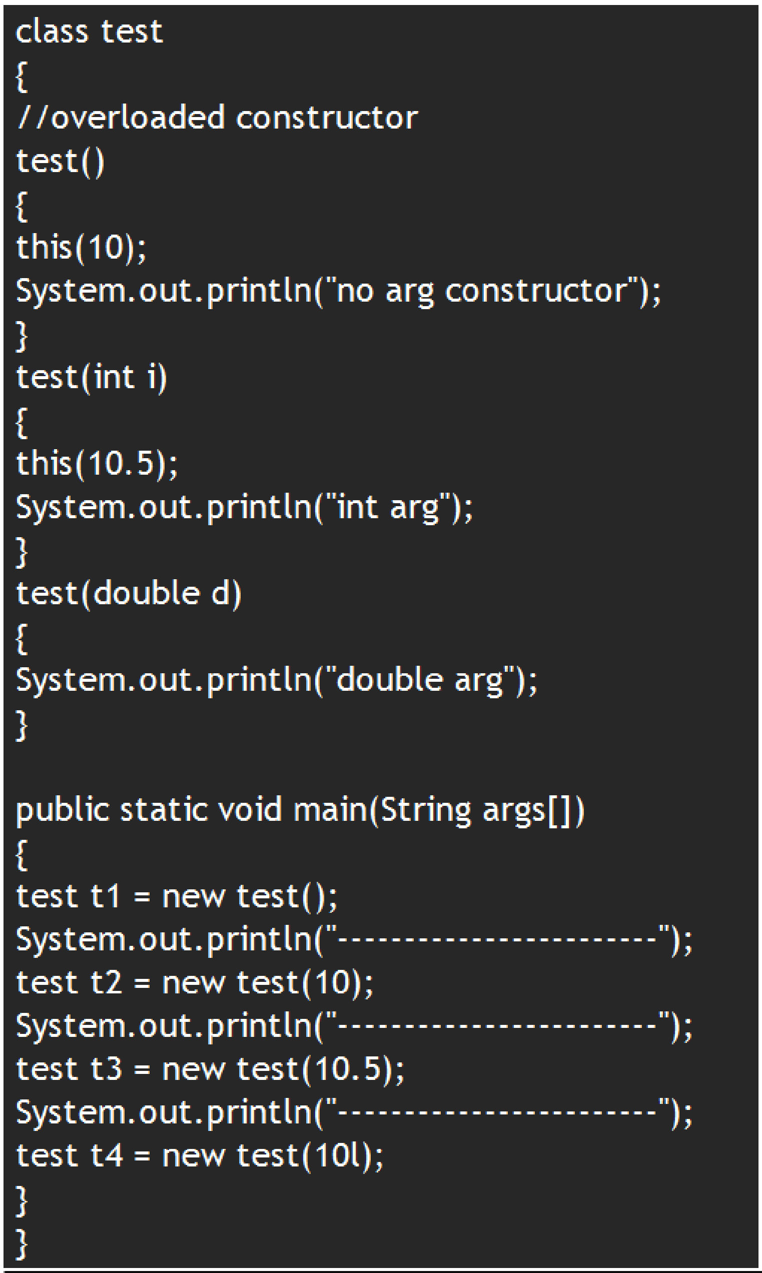
```
class child
{
child()
{
this(20);
}
child(int b)
{
System.out.println("b = "+b);
}
public static void main(String args[])
{
child c = new child();
}
}
```



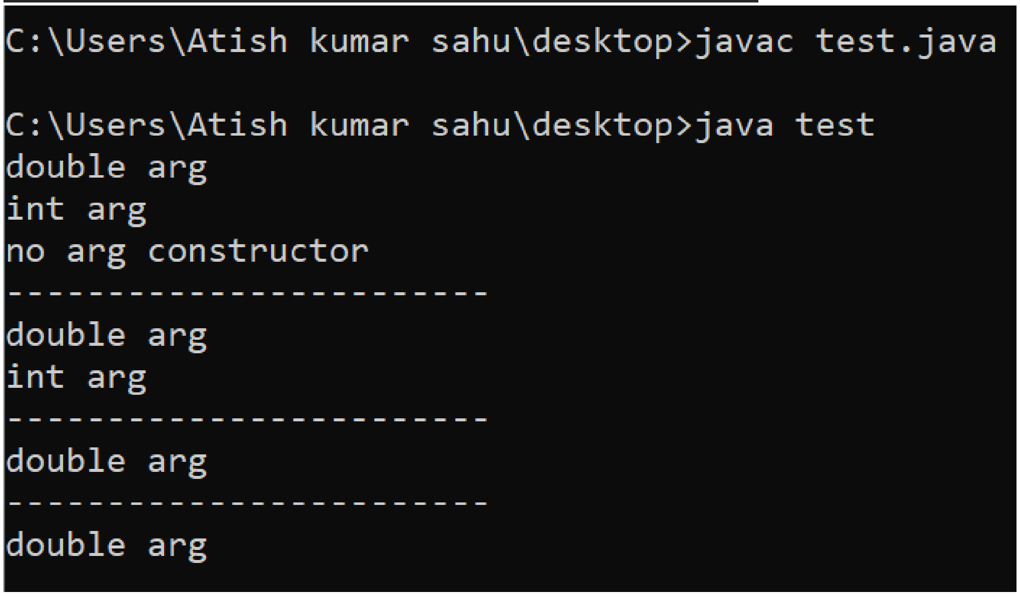
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java child
b = 20
```

OVERLOADED CONSTRUCTOR:

With in a class we can declare multiple constructors and all these constructors having same name but different type of arguments. Hence all these constructor considered as overloaded constructors. Hence overloading concept applicable for constructors.



```
class test
{
//overloaded constructor
test()
{
this(10);
System.out.println("no arg constructor");
}
test(int i)
{
this(10.5);
System.out.println("int arg");
}
test(double d)
{
System.out.println("double arg");
}

public static void main(String args[])
{
test t1 = new test();
System.out.println("-----------------------");
test t2 = new test(10);
System.out.println("-----------------------");
test t3 = new test(10.5);
System.out.println("-----------------------");
test t4 = new test(10l);
}
}
```

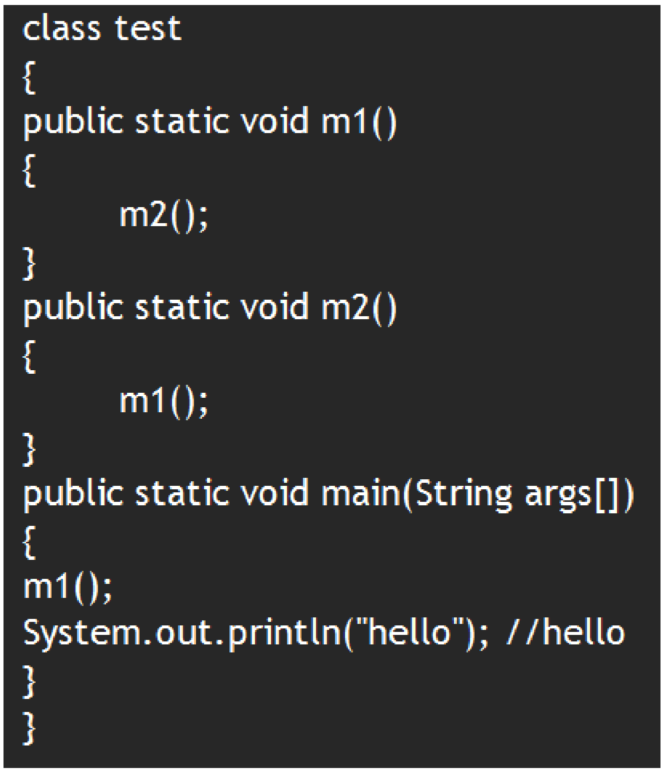


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
double arg
int arg
no arg constructor
------------------------
double arg
int arg
------------------------
double arg
------------------------
double arg
```

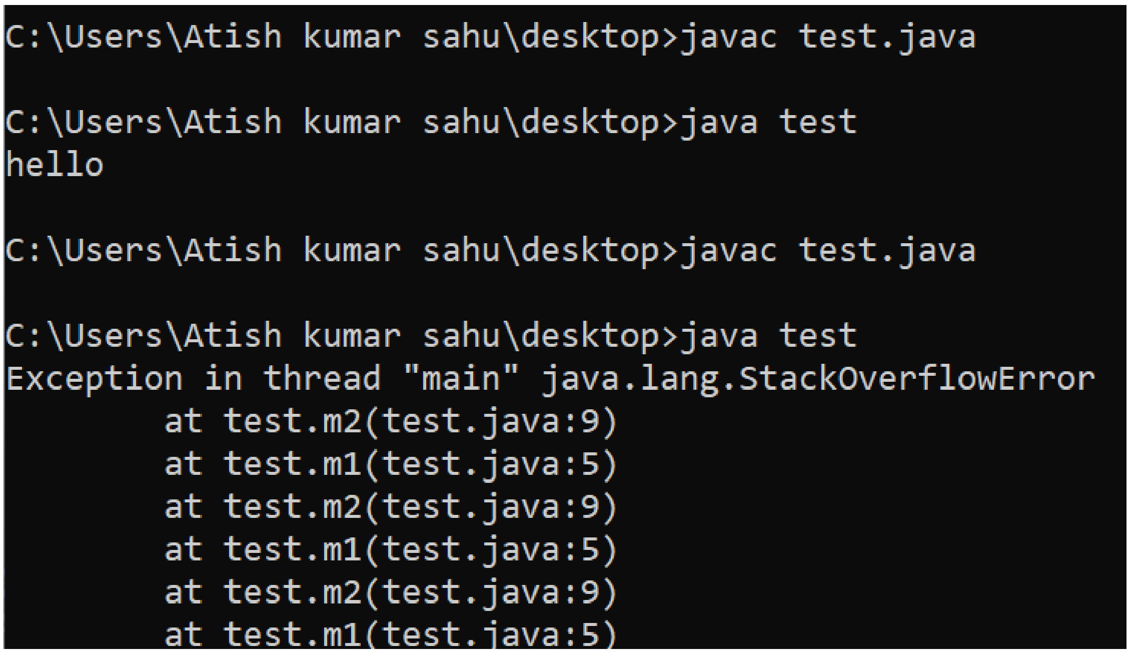
For constructor inheritance and overriding concepts are not applicable. But overloading concept is applicable. Every class in java including abstract class can contain constructor but interface concept can not contain constructor.

```
class test          abstract class test         interface test = wrong statement
{                   {                           {
test()              test()                      test()
{                   {                           {
}                   }                           }
}                   }                           }
```

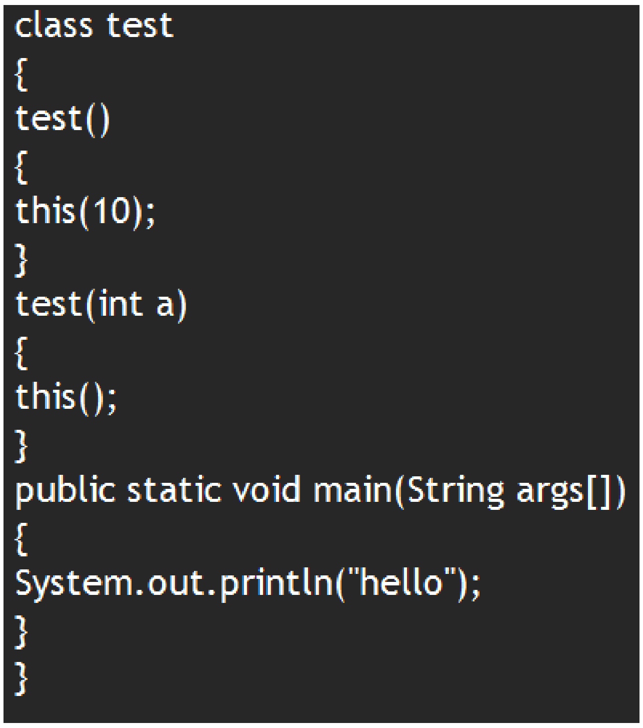
<u>CASE 01:</u>



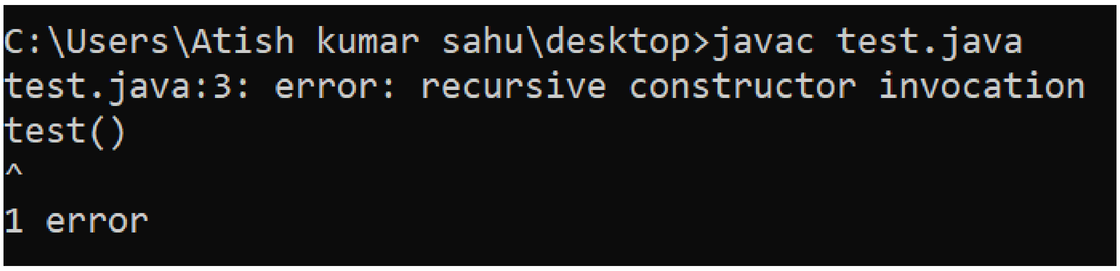
```
class test
{
public static void m1()
{
       m2();
}
public static void m2()
{
       m1();
}
public static void main(String args[])
{
m1();
System.out.println("hello"); //hello
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
hello

C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
Exception in thread "main" java.lang.StackOverflowError
        at test.m2(test.java:9)
        at test.m1(test.java:5)
        at test.m2(test.java:9)
        at test.m1(test.java:5)
        at test.m2(test.java:9)
        at test.m1(test.java:5)
```



```
class test
{
test()
{
this(10);
}
test(int a)
{
this();
}
public static void main(String args[])
{
System.out.println("hello");
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:3: error: recursive constructor invocation
test()
^
1 error
```

Recurssive method call is a runtime exception saying stack overflow error. But in our above program if there is a chance of recurrsive constructor invocation then the code won't compile and we will get compile time error.

<u>CASE 02:</u>

```
class p
{       }
class c extends p
{       }
correct
```

```
class p
{
p()     {       }
}
class c extends p
{       }
correct
```

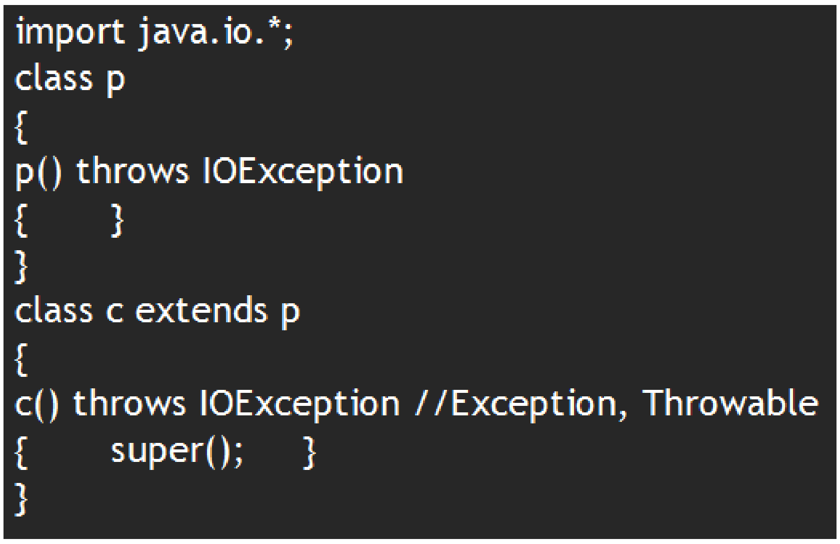
```
class p
{
        p(int i)        {       }
}
class c extends p
{       }
incorrect
```

1. if parent class contains any argument constructor then while writing child classes we have to take special care with respect to constructors.

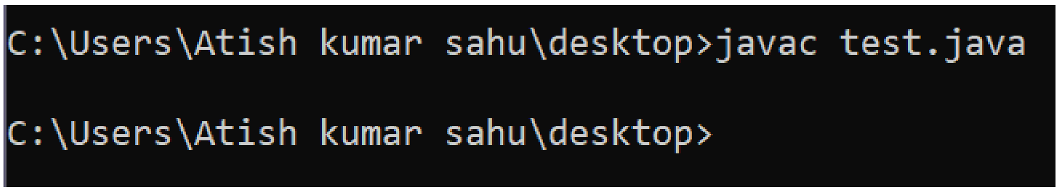
*2. whenever we are writing any argument constructor, it is highly recommended to write no argument constructor also. ******

CASE 03:

```
/*incorrect program*/
class p
{
        p() throws IOException
        {
        }
}
class c extends p
{
        c()
        {
        super();
        }
}
```



```
import java.io.*;
class p
{
p() throws IOException
{      }
}
class c extends p
{
c() throws IOException //Exception, Throwable
{      super();    }
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>
```

If parent class constructor throws any checked exception compulsory child class constructor should throw the same checked exception or its parent. Otherwise the code won’t compile.

WHICH OF THE FOLLOWING STATEMENTS ARE VALID?

1. Main purpose of constructor is to create an object. -> invalid

2. The main purpose of constructor is to perform initialization of object. -> valid

3. The name of the constructor need not be same as class name. -> invalid

4. Return type concept applicable for constructors but only void. -> invalid

5. We can apply any modifier for constructor. -> invalid

6. Default constructor generated by jvm. -> false

7. Compiler is responsible for generating default constructor. -> true

8. Compile will always generate default constructor. -> false

9. If we are writing no arg constructor then compiler will generate default constructor. -> invalid.

10. Every no argumemt constructor is always default constructor. -> false.

11. Default constructor is always no arg constructor. -> true

12. The first line inside every constructor should be either super() or this() if we are not writing anything then compiler will generate this(). -> invalid.

13. For constructors both overloading and overriding concepts are applicable. -> false only overload is valid

14. For constructor inheritance concept applicable but not overriding. -> false

15. Only concrete class can contain constructor but abstract classes can not. -> invalid

16. Interface can contain constructors. -> invalid

17. Recurrsive constructor invocation is a runtime exception. -> invalid

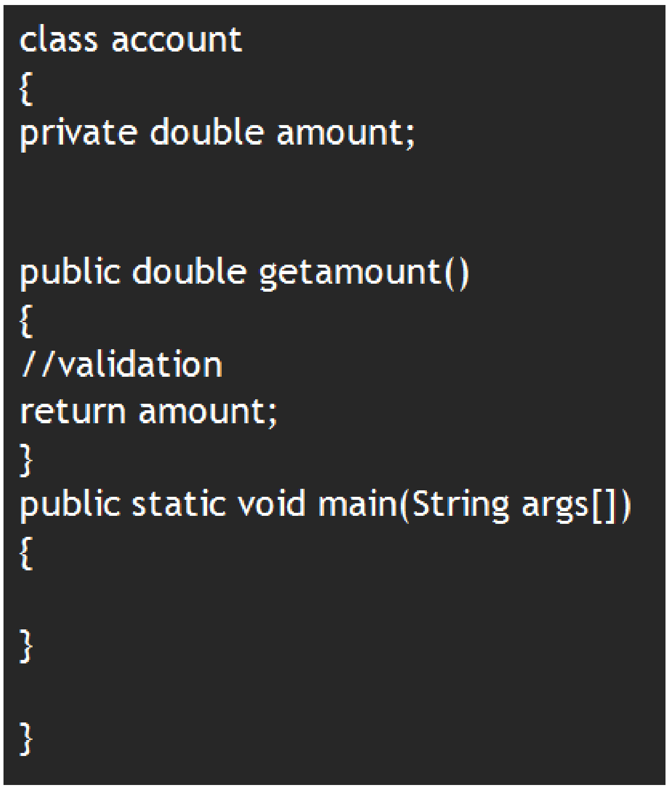
It is a compile time error.

18. If parent class constructor thrwos checked exception then compulsory child class constructor should throw the same checked exception or its chiild. -> invalid it throw parent.

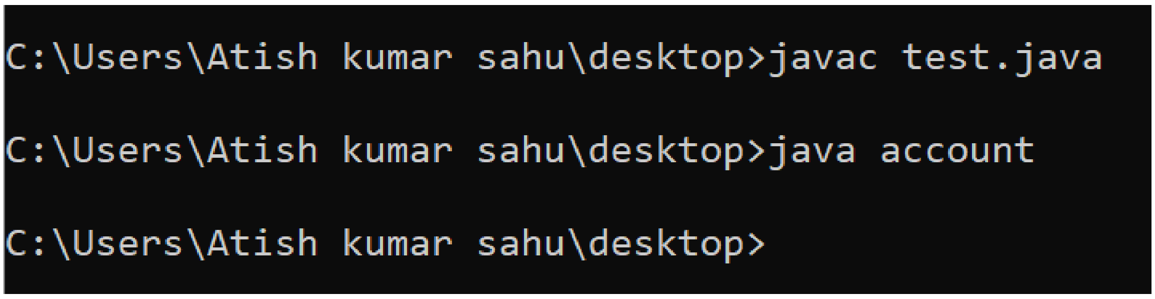
DATA HIDING

DATA HIDING:

Outside person can’t access our internal data directly or our internal data should not go out directly. This oop feature is nothing but data hiding. After validation/authentication outside person can access our internal data. Example: after providing proper username and password we can able to access our Gmail inbox information. Example: even though we are valid customer of the bank we can able to access our account information and we can’t access other’s account information. By declaring data member(variable) as private we can achieve data hiding. The main advantage of data hiding is security. ***It is highly recommended data member(variables) as private***.



```
class account
{
private double amount;


public double getamount()
{
//validation
return amount;
}
public static void main(String args[])
{

}

}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java account

C:\Users\Atish kumar sahu\desktop>
```

ABSTRACTION

ABSTRACTION:

Hiding internal implementation and just highlight the setup services what we are offering is a concept of abstraction. Through bank ATM GUI screen bank people are highlighting the stup services what they are offering without highlighting internal implementation. The maim advantages of abstraction are

1. We can achieve security because we are not highlighting our internal implementation.

2. Without affecting outside person we can able to perform anytype of changes in our internal system and hence enhancement will become easy.

3. It improves maintainability of the application. It improves easyness to use our system.

By using interfaces and abstract classes we can implement abstraction.

ENCAPSULATION

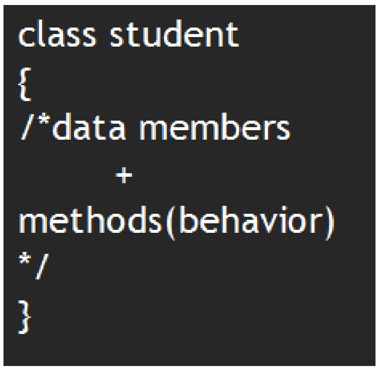
ENCAPSULATION:

The process of binding data and corresponding methods into a single units is nothing but encapsulation. If any component follows data hiding and abstraction such type of component is said to be encapsulated component.

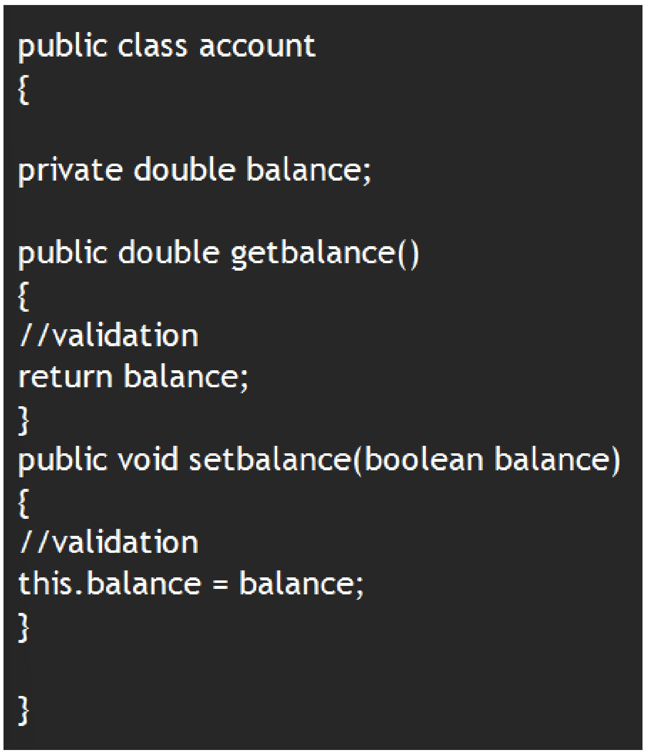
Encapsulation = Data Hiding + Abstraction

The main advantage of encapsulation are we can achieve security, enhancement will become easy, it improves maintainability of application.

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



```
class student
{
/*data members
       +
methods(behavior)
*/
}
```



```
public class account
{

private double balance;

public double getbalance()
{
//validation
return balance;
}
public void setbalance(boolean balance)
{
//validation
this.balance = balance;
}

}
```

TIGHTLY ENCAPSULATED CLASS:

A class is said to be tightly encapsulated if and only if each and every variable declared as private. Whether class contain corresponding getter and setter methods are not and wether these methods are declared as public or not these things we are not required to check.

```
class account
{

private double balance;

public double getbalance()
{
return balance;
}

}
```

Q. WHICH OF THE FOLLOWING CLASSES ARE TIGHTLY ENCAPSULATED?

```
class a
{
private int x = 10; //tight encap
}
class b extends a
{
int y = 20; //no tight encap
}
class c extends a
{
private int z = 30; //tight encap
}
```

Class a & class c both are the tightly encapsulated.

Q. WHICH OF THE FOLLOWING CLASSES ARE TIGHTLY ENCAPSULATED?

```
class a
{
int x = 10; //no tight
}
class b extends a
{
private int y = 20; //no tight
}
class c extends b
{
private int z = 30; //no tight
}
```

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

Data hiding, Abstraction, Encapsulation, Tightly Encapsulated class are for security.

IS A RELATIONSHIP

IS A RELATIONSHIP:

It is also known as "*Inheritance*". The main advantage of IS A RELATIONSHIP is "*Code Reusabilty*". By using "*extends*" keyword we can implement IS A RELATIONSHIP.

```
class p
{
	public void m1()
	{
	System.out.println("parent");
	}
}
class c extends p
{
	public void m2()
	{
	System.out.println("child");
	}
}
```

```
class test
{
Public static void main(String args[])
{
//01
p p = new p();
p.m1();//valid
p.m2();//invalid

//02
c c = new c();
c.m1();//valid
c.m2();//valid

//03
p p1 = new c();
p1.m1();//valid
p1.m2();//invalid

//04
c c1 = new p();//invalid
}
}
```

CONCLUSION:

01. Whatever methods child has by default not available to the parent, and hence on the parent reference we can't call child specific methods.

02. Whatever methods parent has by default available to the child, and hence on the child reference we can call both parent and child class methods.

03. Parent reference can be used to hold child object but by using that reference we can't call child specific methods. But we can call the methods present in parent class.

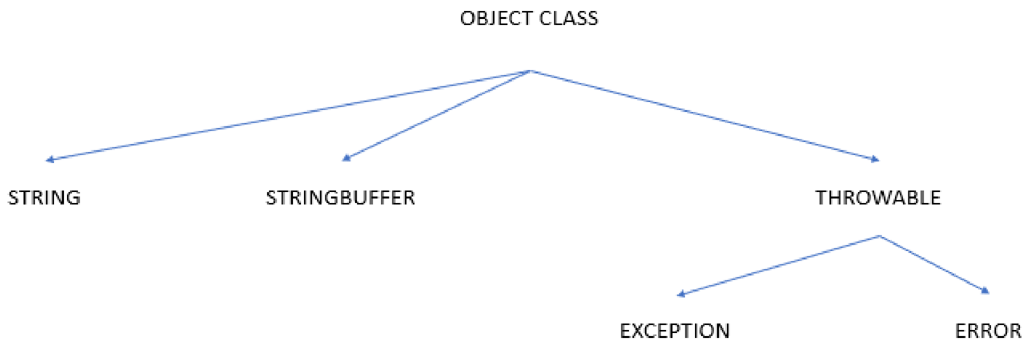
04. Parent reference can be used to hold child object but child reference can not be used to hold parent object.

INHERITANCE:

The most common methods which are applicable for anytype of child, we have to define in parent class. the specific methods which are applicable for a particular child we have to define in child class.

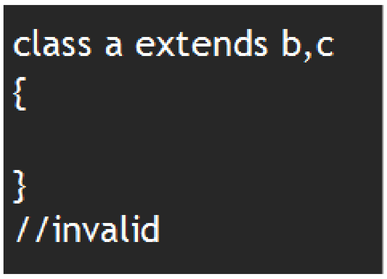
Total java API is implemented based on inheritance concept. The most common methods which are applicable for any java object are defined in object class and hence every class in java is the child class of object either directly or indirectly so that object class methods by default available to every java class without rewriting due to this object class access root for all java classes.

Throwable class defines the most common methods which are required for every exception and error classes. Hence this class access root for java exception hierarchy.



MULTIPLE INHERITANCE:

A java class can't extend more than one class at a time. Hence java won't provide support for multiple inheritance in classes.



```
class a extends b,c
{

}
//invalid
```

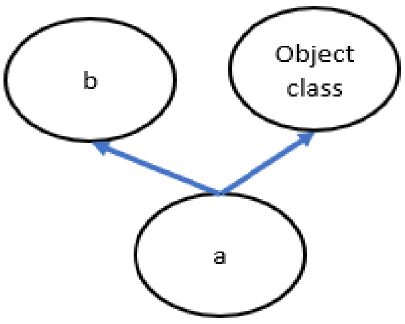
IF OUR CLASS DOESN'T EXTEND ANY OTHER CLASS THEN ONLY OUR CLASS IS DIRECT CHILD CLASS OF OBJECT.

class a {   } -> "class a" is the class of the object class.

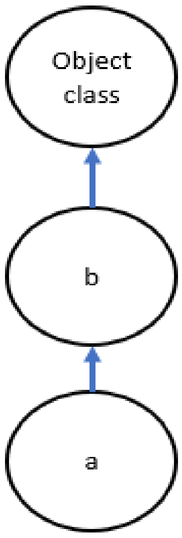
IF OUR CLASS EXTENDS ANY OTHER CLASS THEN OUR CLASS IS INDIRECT CHILD CLASS OF OBJECT.

EITHER DIRECTLY OR INDIRECTLY JAVA WON'T PROVIDE SUPPORT FOR INHERITANCE WITH RESPECT TO CLASSES.

```
class a extends b
{

}
```



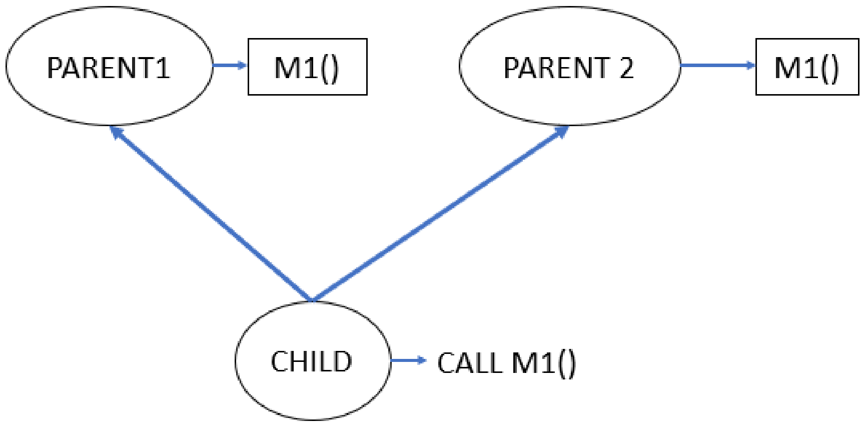
Multiple inheritance which is not allowed



Multi level inheritance which is allowed in java

Q. WHY JAVA WON'T PROVIDE SUPPORT FOR MULTIPLE INHERITANCE?

ANS: There may be a chance of ambiguity problem hence java won't provide support for multiple inheritance.



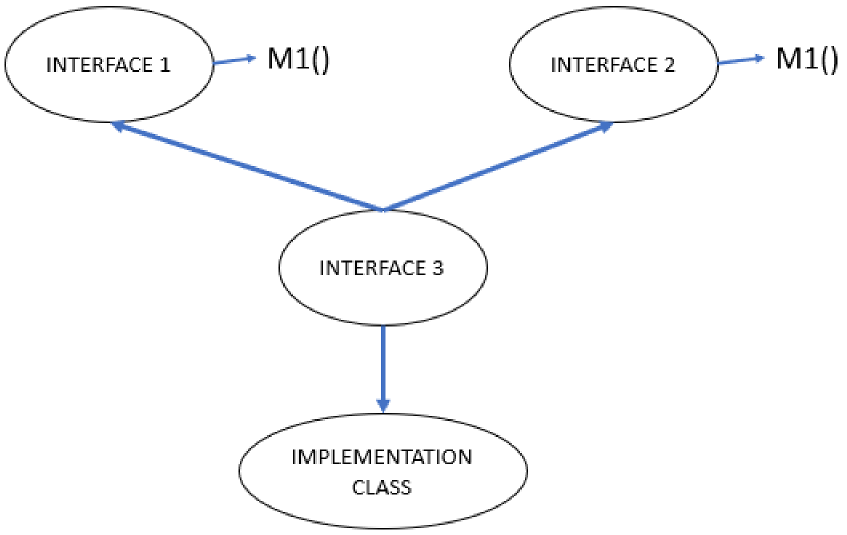
Ambiguity Problem

BUT INTERFACE CAN EXTEND ANY NUMBER OF INTERFACES SIMULTANEOUSLY HENCE JAVA PROVIDE SUPPORT FOR MULTIPLE INHERITANCE WITH RESPECT TO INTERFACES.

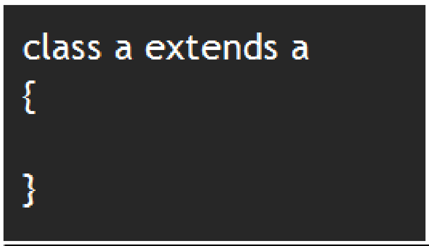
interface a { }   interface b { }   interface c extends a, b { }

Q. WHY AMBIGUITY PROBLEMWON'T BE THERE IN INTERFACES?

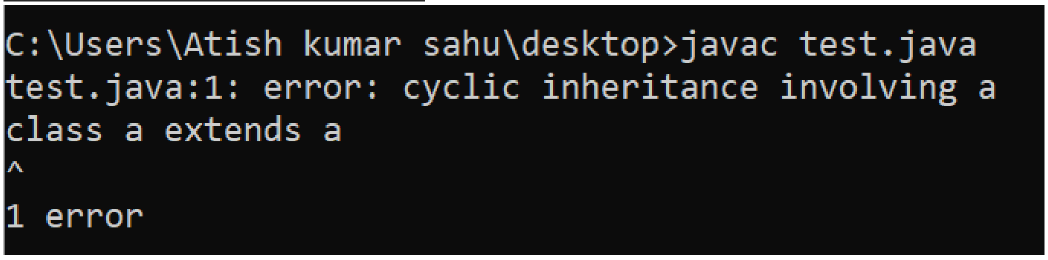
ANS: Even though multiple method declarations are available but implementation is unique. And hence there is no chance of ambiguity problem in interfaces.



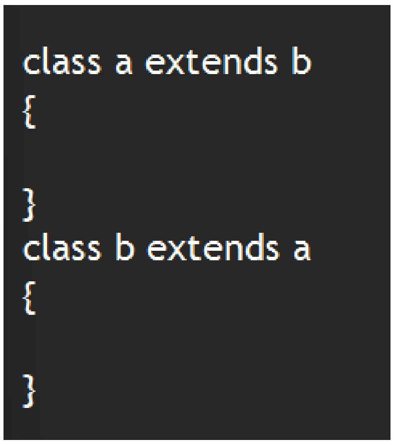
STRICTLY SPEAKING THROUGH INTERFACES WE WON'T GET ANY INHERITANCE.



```
class a extends a
{

}
```



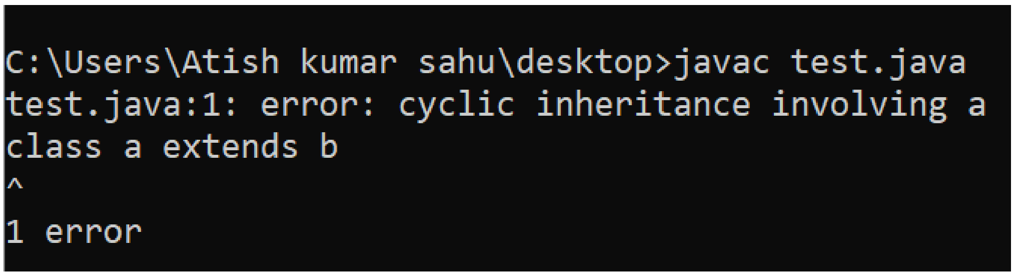
```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:1: error: cyclic inheritance involving a
class a extends a
^
1 error
```



```
class a extends b
{

}
class b extends a
{

}
```

CYCLIC INHERITANCE:

It is an inheritance is not allowed in java of course it is not required.



```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:1: error: cyclic inheritance involving a
class a extends b
^
1 error
```

HAS A RELATIONSHIP:

Has a relationship is also known as composition and agreegation. There is no specific keyword to implement “has a relation” but most of the time we are depending on new keyword. The main advantage of has a relationship is reusability of code.

Example:

```
class Enginee
{
/*enginee specific functionality */
}
class car
{
 Enginee e = new Enginee();
}
```

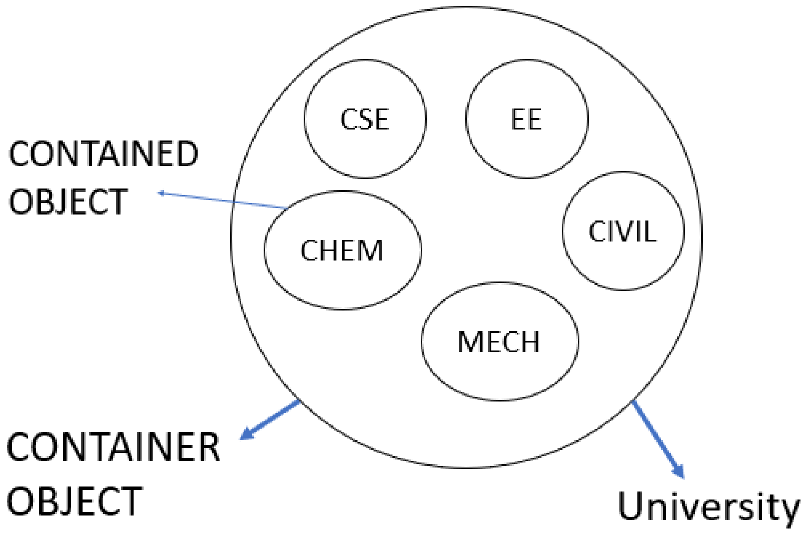
Here class car has a Enginee reference.

DIFFERENCE BETWEEN COMPOSITION AND AGREEGATION:

COMPOSITION:

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

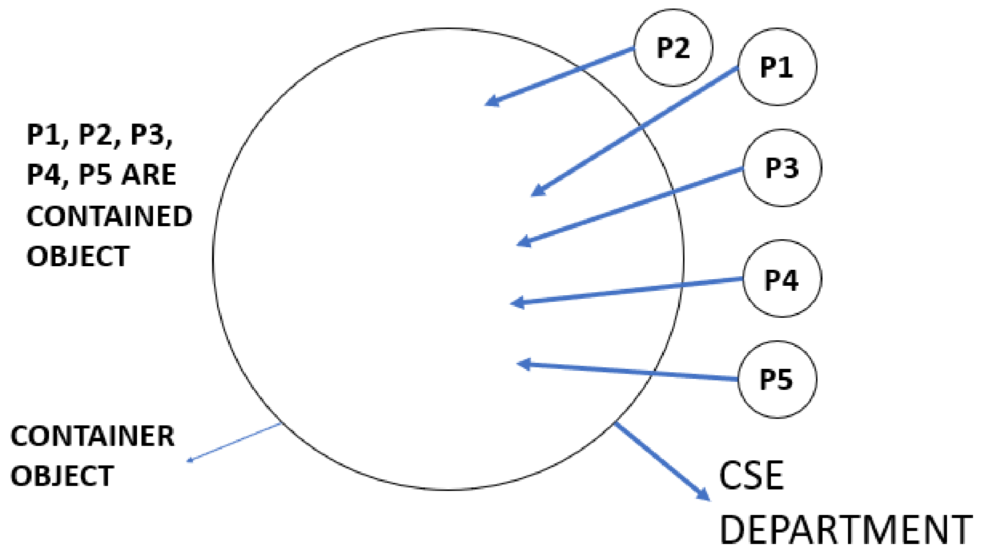
Example: university consist of several departments without existing university there is no chance of existing department hence university and department are strongly associated and the strong association is nothing but composition.



AGREEGATION:

Without existing container object if there is a chance of existing contained object then container and contained objects are weakly associated and the weak associate is nothing but aggreegation.

Example: in a particular department consist of several professors. Without existing department there may be a chance of existing professor objects. Hence department and professor objects are weakly associtaed and the weak association is nothing but agreegation.



IN COMPOSITION OBJECTS ARE STRONGLY ASSOCIATED WHERE AS IN AGREEGATION OBJECTS ARE WEAKLY ASSOCIATED.

IN COMPOSITION CONTAINER OBJECT HOLDS DIRECTLY CONTAINED OBJECTS WHERE AS IN AGREEGATION CONTAINER OBJECT HOLDS. JUST REFERENCES OF CONTAINED OBJECTS.

IS A RELATONSHIP VS HAS A RELATIONSHIP:

If we want total functionality of a class automatically then we should go for is a relationship.

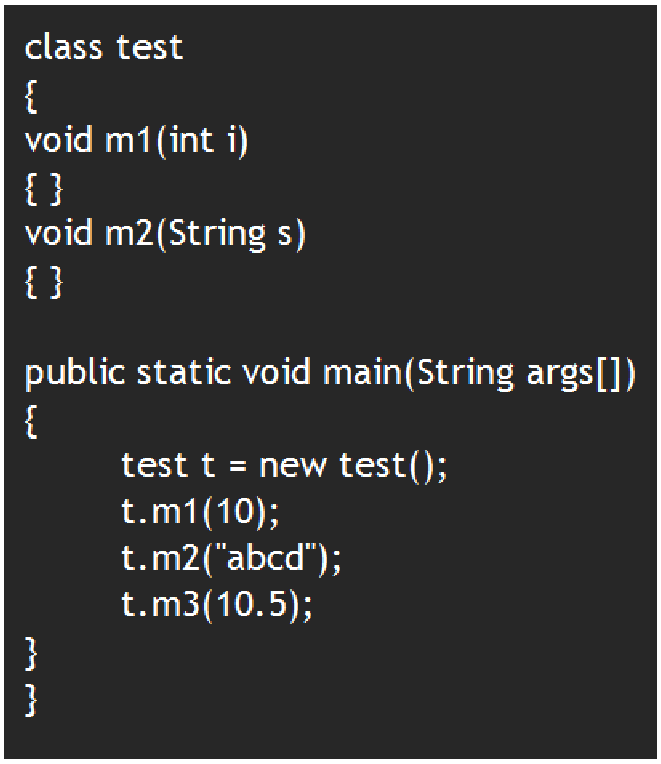
Example: Person class function completely required for student class.

If we want part of the functionality then we should go for has a relationship.

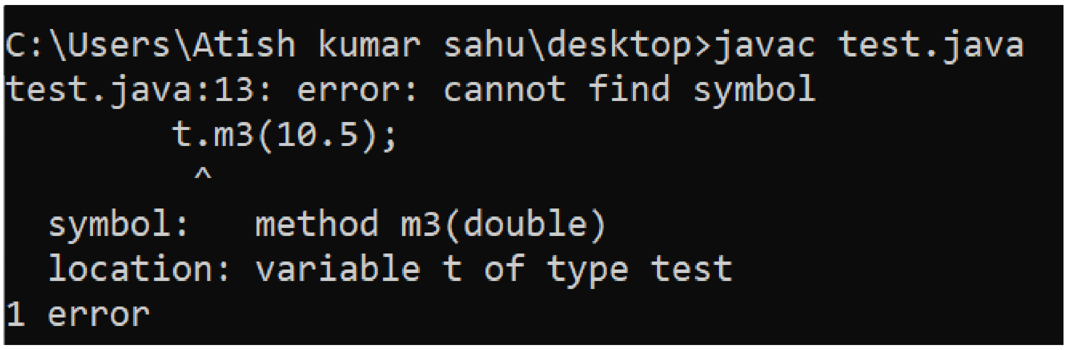
Example: a class contain 100 methods and in another demo class we required ony 20 methods of class who contains 100 methods.

METHOD SIGNATURE:

In java method signature consist of method names followed by argument types. return type is not a part of method signature in java. public static int m1(int I, float F){ } -> m1(int, float)

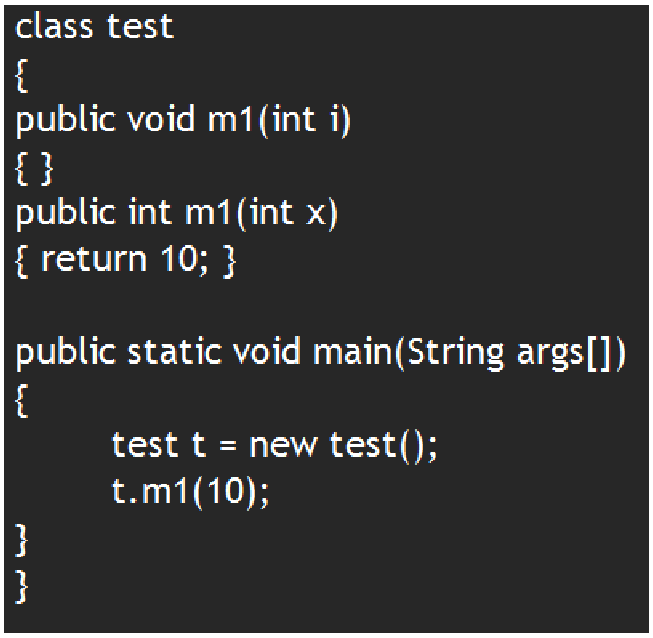


```
class test
{
void m1(int i)
{ }
void m2(String s)
{ }

public static void main(String args[])
{
        test t = new test();
        t.m1(10);
        t.m2("abcd");
        t.m3(10.5);
}
}
```

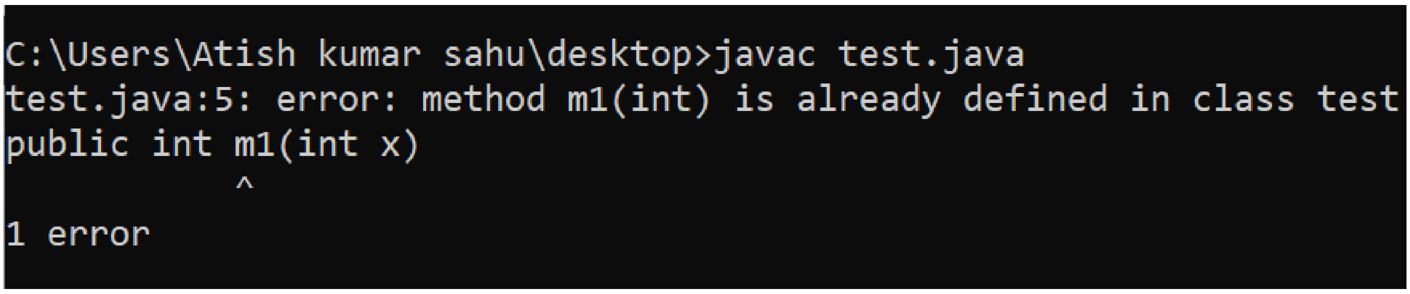


```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:13: error: cannot find symbol
        t.m3(10.5);
         ^
  symbol:   method m3(double)
  location: variable t of type test
1 error
```

Compiler will use method signature to resolve method calls.



```
class test
{
public void m1(int i)
{ }
public int m1(int x)
{ return 10; }

public static void main(String args[])
{
        test t = new test();
        t.m1(10);
}
}
```



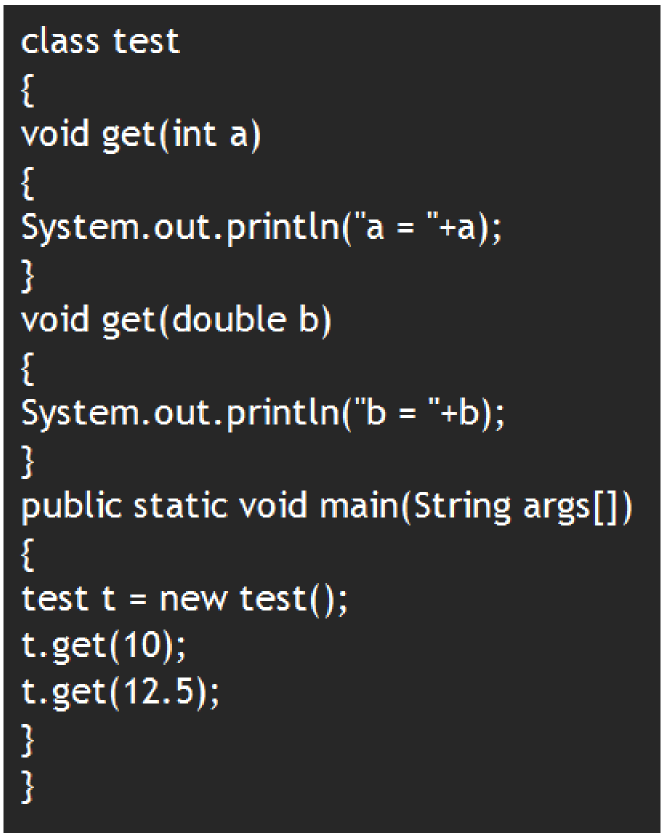
```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:5: error: method m1(int) is already defined in class test
public int m1(int x)
           ^
1 error
```

WITHIN A CLASS TWO METHODS OF SAME SIGNATURE NOT ALLOWED.

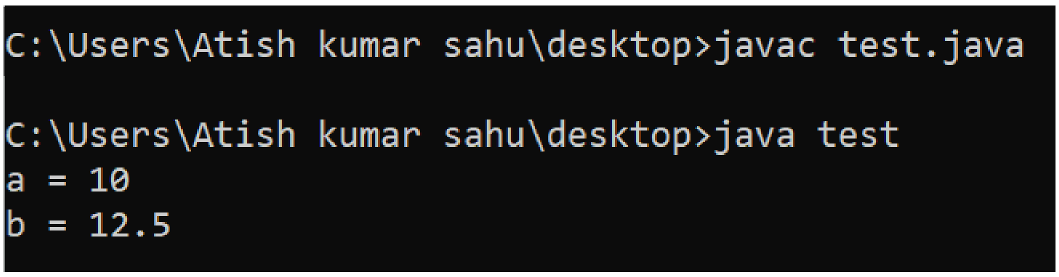
OVERLOADING

OVERLOADING:

TWO METHODS ARE SAID TO BE OVERLOADED IF AND ONLY IF BOTH METHODS HAVING SAME NAME BUT DIFFERENT ARGUMENT TYPES. IN C THERE IS NO OVERLOADING.



```
class test
{
void get(int a)
{
System.out.println("a = "+a);
}
void get(double b)
{
System.out.println("b = "+b);
}
public static void main(String args[])
{
test t = new test();
t.get(10);
t.get(12.5);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
a = 10
b = 12.5
```

/*for java abs() you have to import java.lang.Math.abs(); S.O.P(Math.abs(variable)); */

In C language method overloading concept is not available. Hence we can’t declare multiple methods with same name but different argument types. If there is change in argument type compulsory we should go for new method name which increases complexity of programming. But in java we can declare multiple methods with same name but different argument types such type of methods are called overloaded methods.

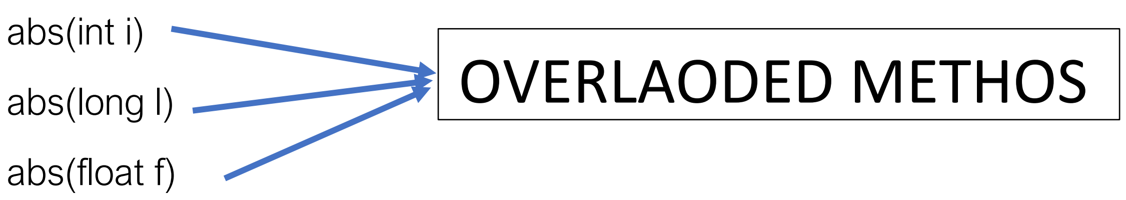
<u>IN C:</u>

Abs(int i) == abs(10);

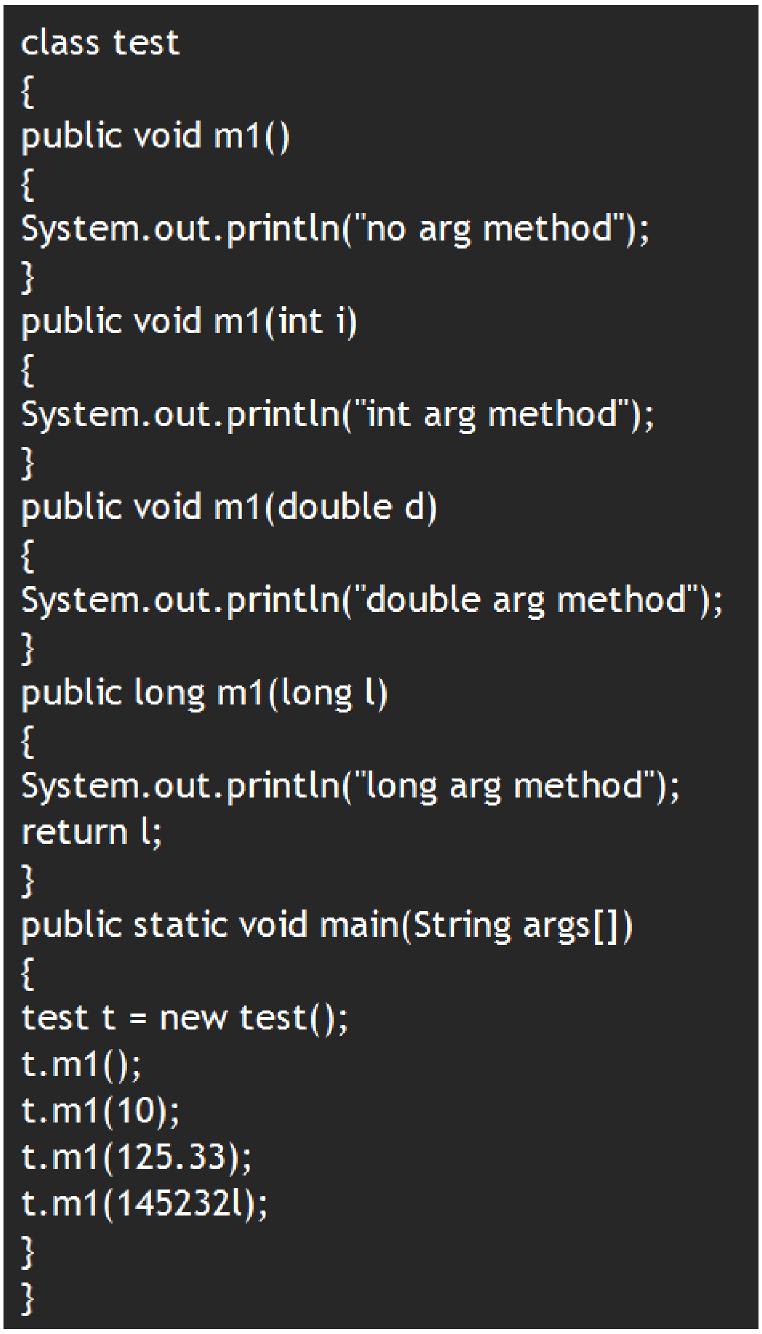
Labs(long l) == Labs(10l);

Fabs(float f) = Fabs(10.5f);

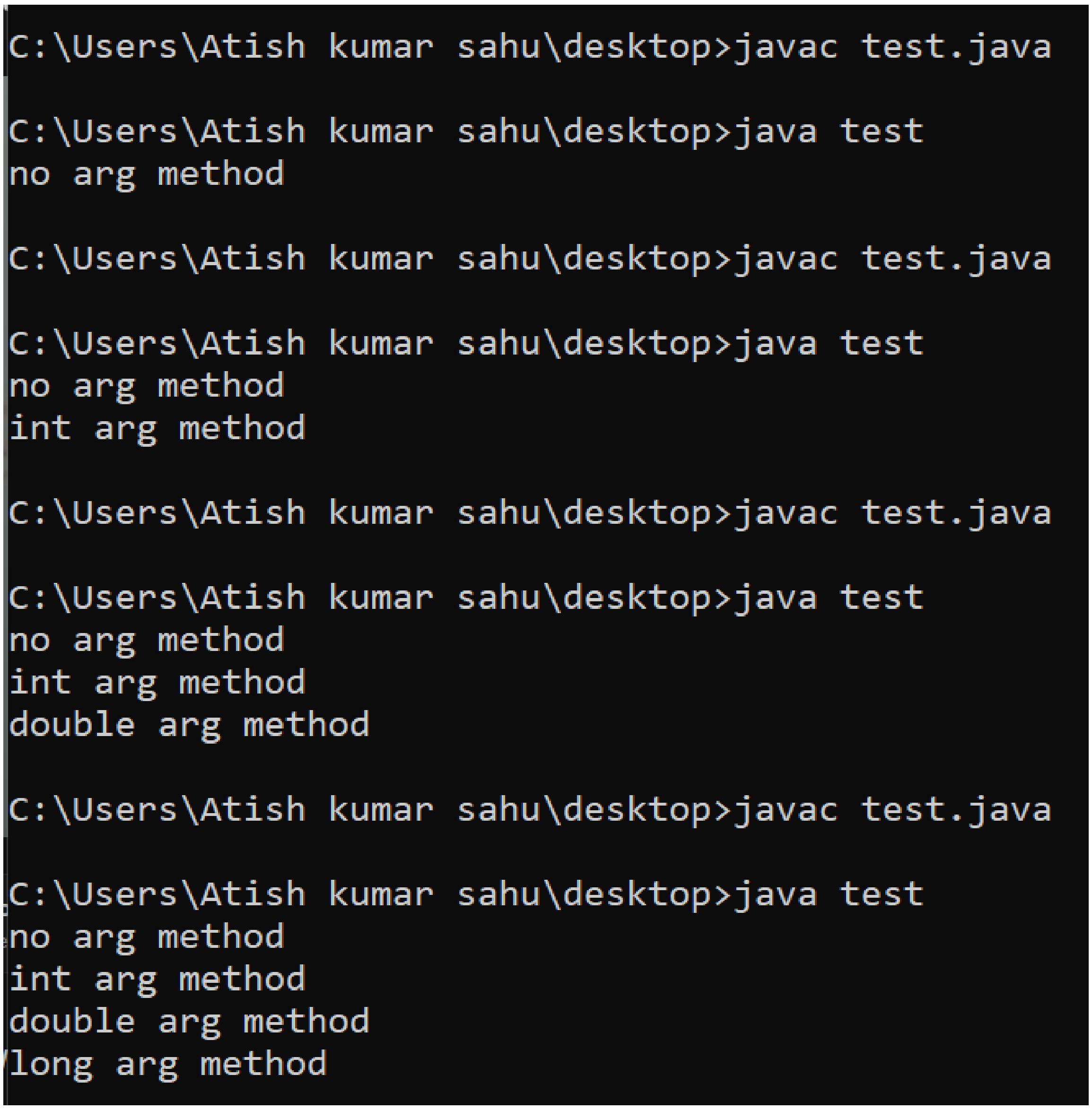
<u>IN JAVA:</u>



Having overloading concept in javareduces complexity of programming.



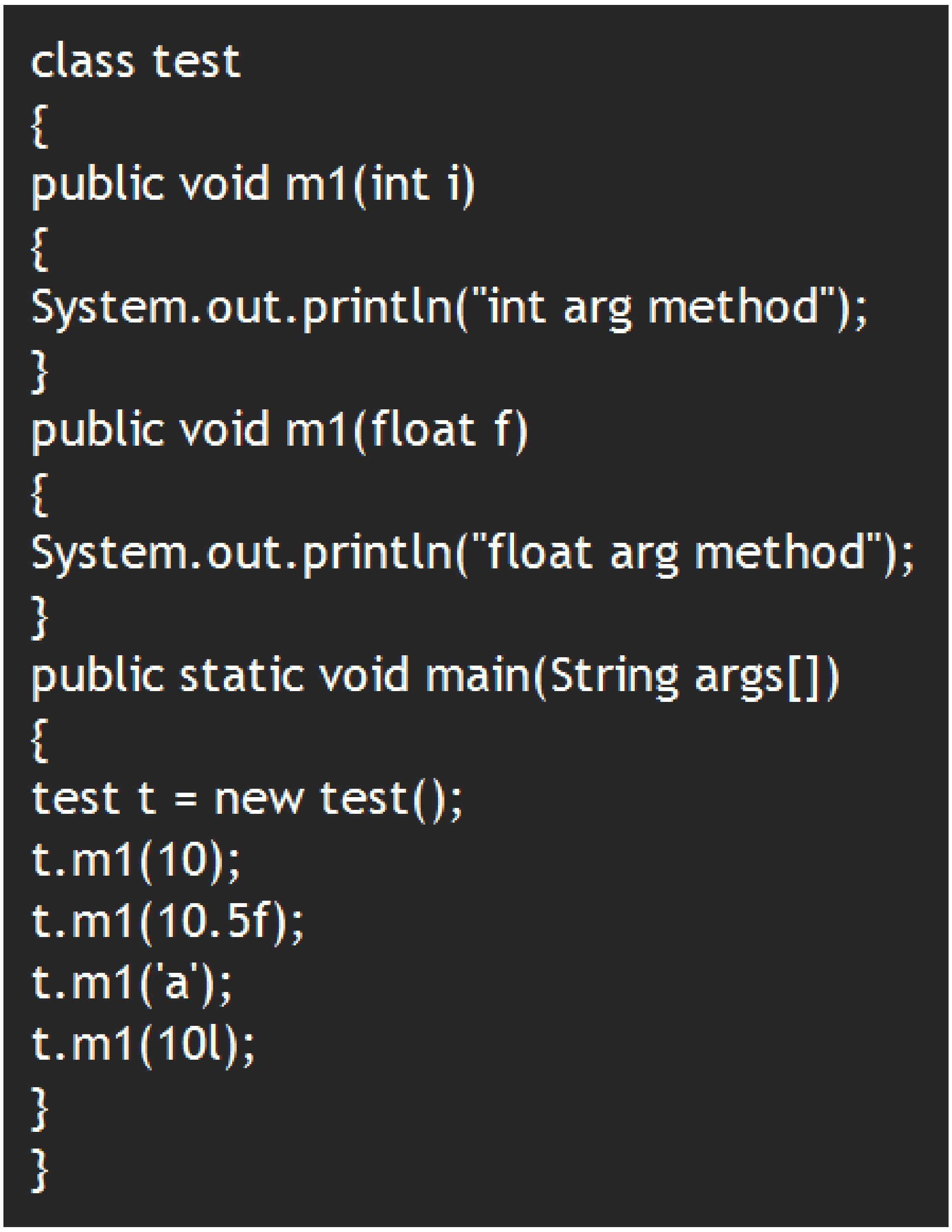
```
class test
{
public void m1()
{
System.out.println("no arg method");
}
public void m1(int i)
{
System.out.println("int arg method");
}
public void m1(double d)
{
System.out.println("double arg method");
}
public long m1(long l)
{
System.out.println("long arg method");
return l;
}
public static void main(String args[])
{
test t = new test();
t.m1();
t.m1(10);
t.m1(125.33);
t.m1(145232l);
}
}
```



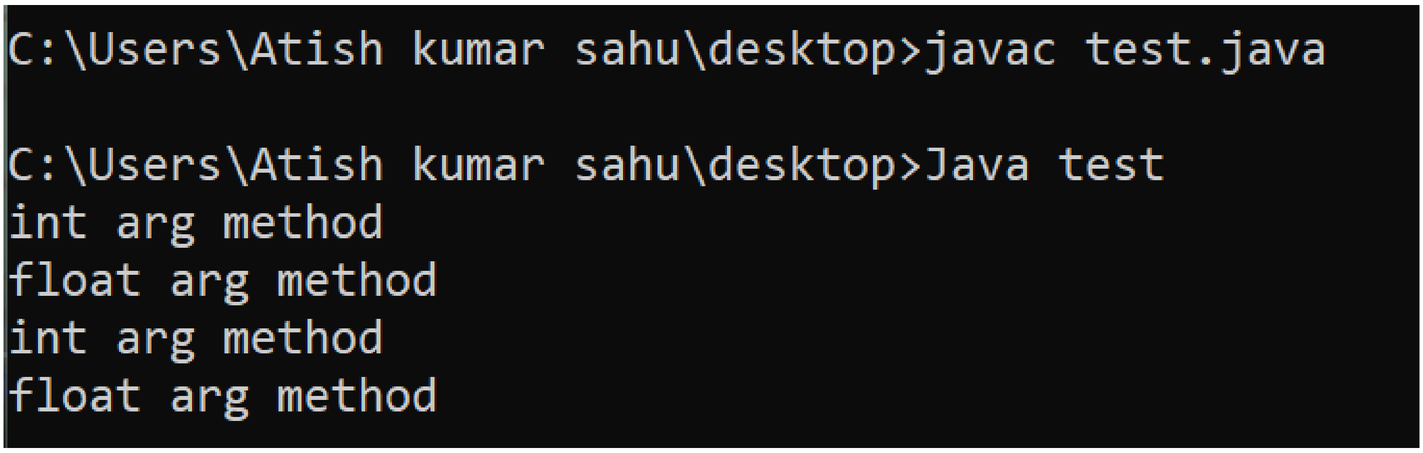
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
no arg method

C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
no arg method
int arg method

C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
no arg method
int arg method
double arg method

C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
no arg method
int arg method
double arg method
long arg method
```

IN OVERLOADING METHOD RESOLUTION ALWAYS TAKES CARE BY COMPILER BASED ON REFERENCE TYPE HENCE OVERLOADING IS ALSO CONSIDERED AS "COMPILE TIME POLYMORPHISM", "STATIC POLYMORPHISM", "EARLY BINDING".

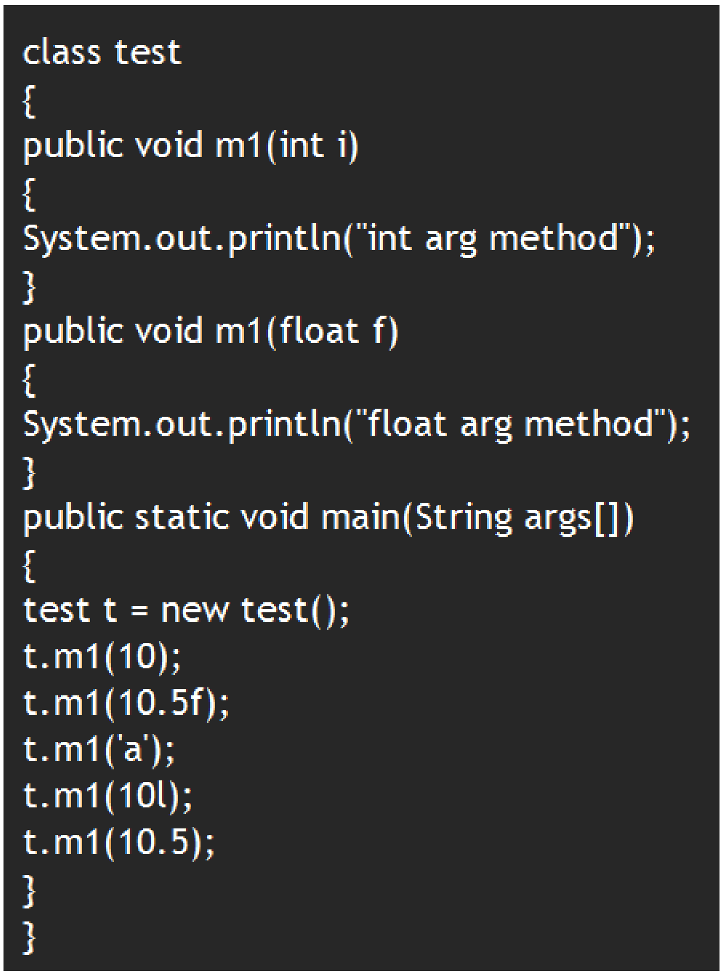
CASE 01:



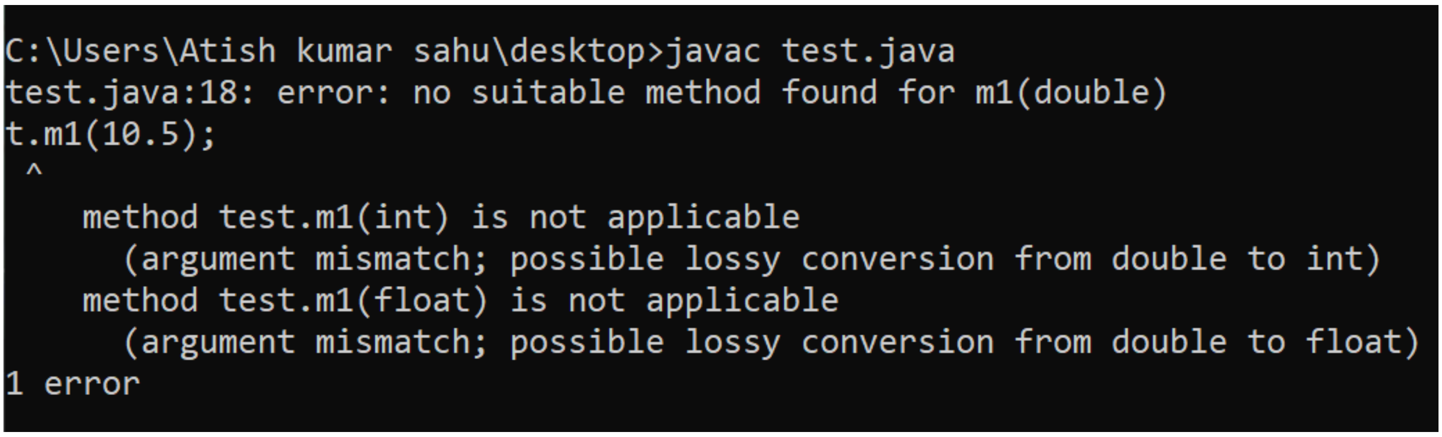
```
class test
{
public void m1(int i)
{
System.out.println("int arg method");
}
public void m1(float f)
{
System.out.println("float arg method");
}
public static void main(String args[])
{
test t = new test();
t.m1(10);
t.m1(10.5f);
t.m1('a');
t.m1(10l);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>Java test
int arg method
float arg method
int arg method
float arg method
```



```
class test
{
public void m1(int i)
{
System.out.println("int arg method");
}
public void m1(float f)
{
System.out.println("float arg method");
}
public static void main(String args[])
{
test t = new test();
t.m1(10);
t.m1(10.5f);
t.m1('a');
t.m1(10l);
t.m1(10.5);
}
}
```



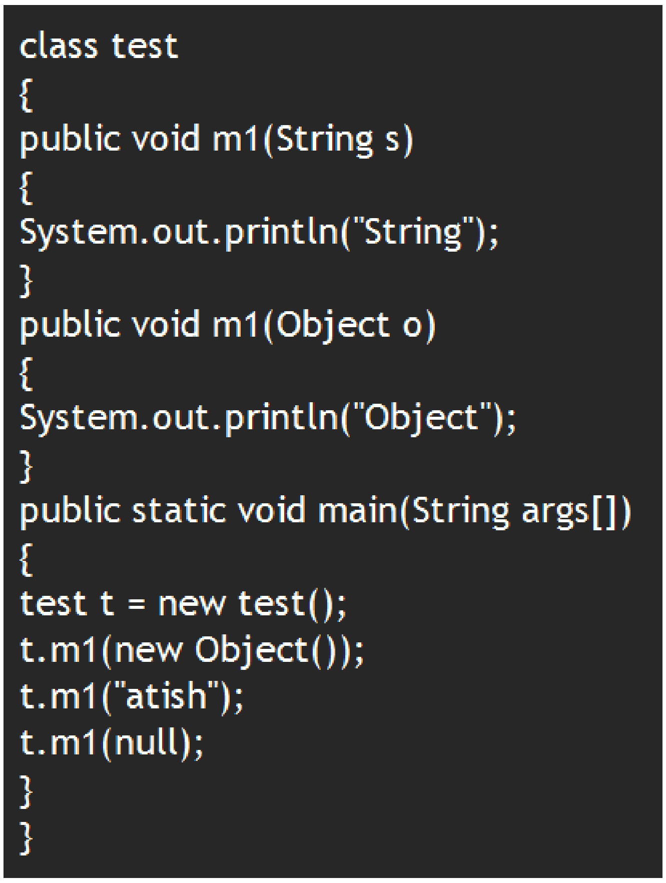
```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:18: error: no suitable method found for m1(double)
t.m1(10.5);
  ^
    method test.m1(int) is not applicable
      (argument mismatch; possible lossy conversion from double to int)
    method test.m1(float) is not applicable
      (argument mismatch; possible lossy conversion from double to float)
1 error
```

AUTOMATIC PROMOTION IN OVERLOADING CONCEPT. WHILE RESOLVING OVERLOADED METHOD IF EXACT METHOD IS NOT AVAILABLE THEN WE WON'T GET ANY COMPILE TIME ERROR IMMEDIATELY. FIRST IT WILL PROMOTE ARGUMENT TO THE NEXT LEVEL AND CHECK WHETHER MATCHED METHOD IS AVAILABLE OR NOT. IF MATCHED METHOD IS AVAILABLE THEN IT WILL BE CONSIDERED. AND THE MATCHED METHOD IS NOT AVAILABLE THEN COMPILER PROMOTES ARGUMENT ONCE AGAIN TO THE NEXT LEVEL. THIS PROCESS WILL CONTINUED UNTILL ALL POSSIBLE PROMOTIONS STILL IF THE MATCHED METHOD IS NOT AVAILABLE THEN WE WILL GET COMPILE TIME ERROR. THE FOLLOWING ARE ALL POSSIBLE PROMOTIONS IN OVERLOADING.

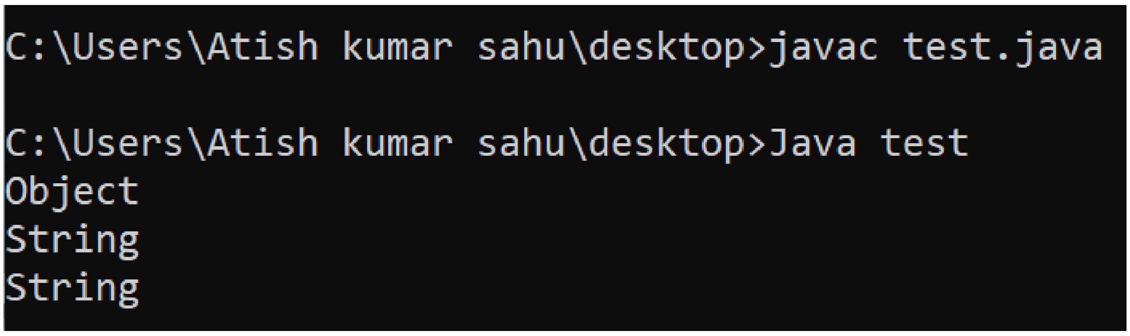
BYTE -> SHORT-> INT-> LONG-> FLOAT-> DOUBLE

CHAR -> INT ↗ *THIS PROCESS IS CALLED AUTOMATIC PROMOTION.*

CASE 02



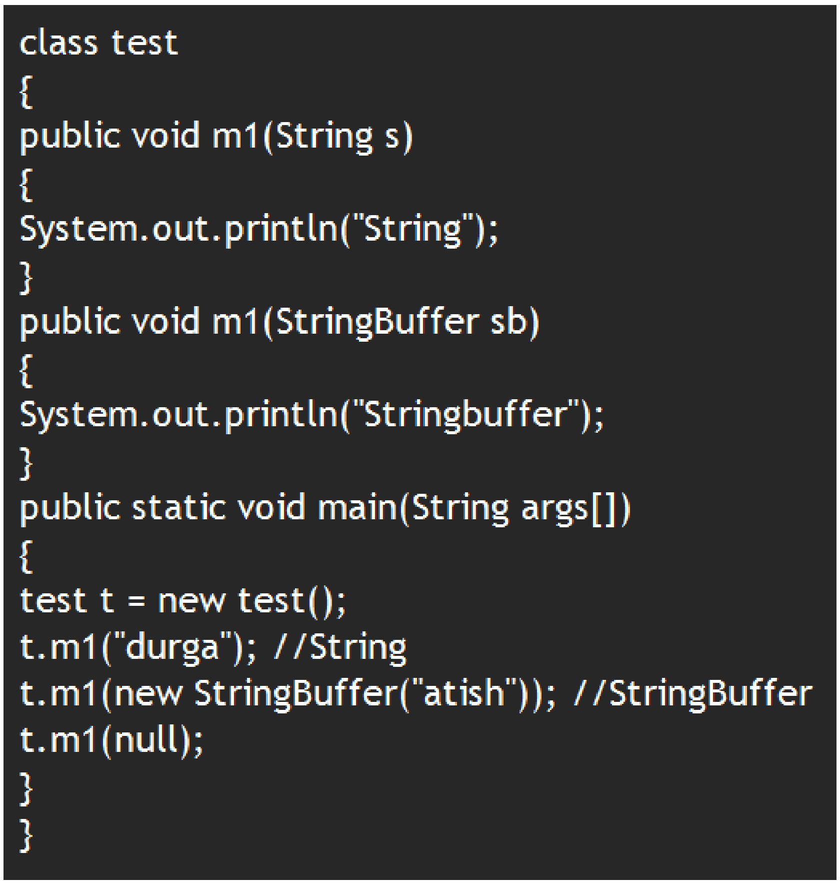
```
class test
{
public void m1(String s)
{
System.out.println("String");
}
public void m1(Object o)
{
System.out.println("Object");
}
public static void main(String args[])
{
test t = new test();
t.m1(new Object());
t.m1("atish");
t.m1(null);
}
}
```



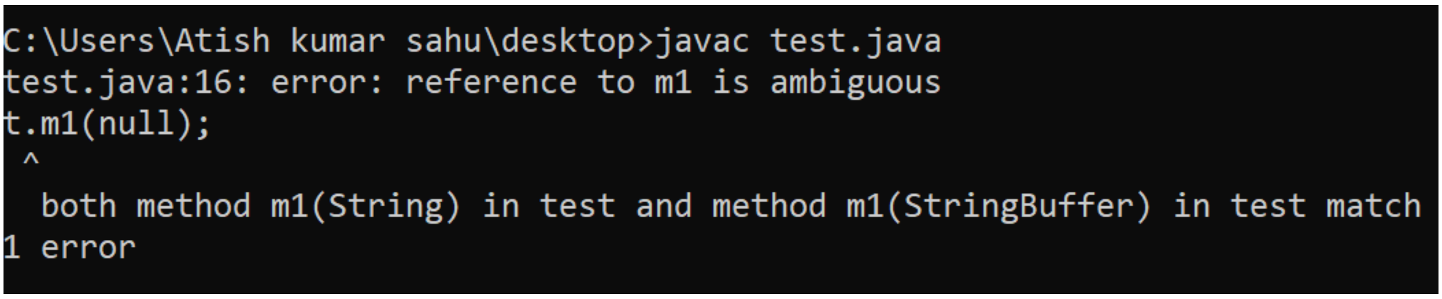
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>Java test
Object
String
String
```

WHILE RESOLVING OVERLOADED METHODS COMPILER WILL ALWAYS GIVES THE PRECEDENCE FOR CHILD TYPE ARGUMENT THEN COMPARED WITH PARENT TYPE ARGUMENT.

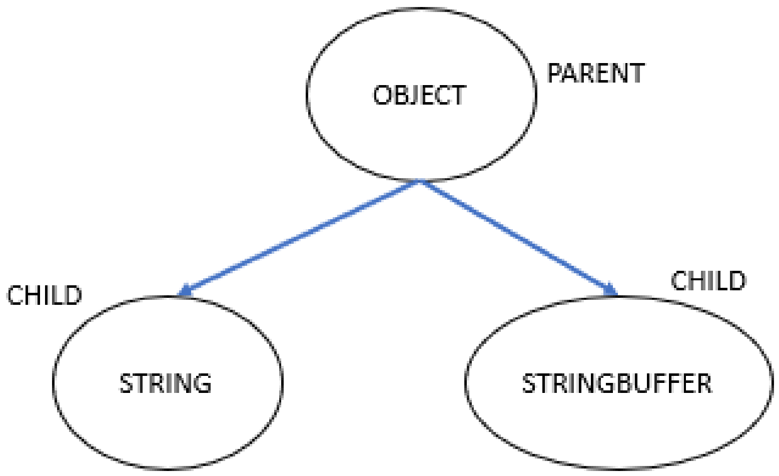
CASE 03



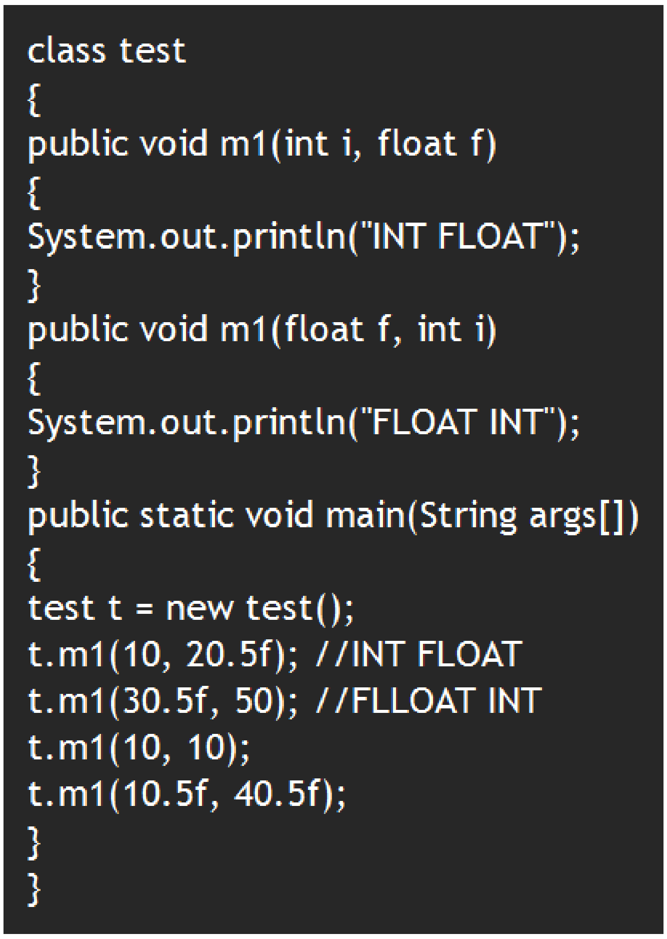
```
class test
{
public void m1(String s)
{
System.out.println("String");
}
public void m1(StringBuffer sb)
{
System.out.println("Stringbuffer");
}
public static void main(String args[])
{
test t = new test();
t.m1("durga"); //String
t.m1(new StringBuffer("atish")); //StringBuffer
t.m1(null);
}
}
```



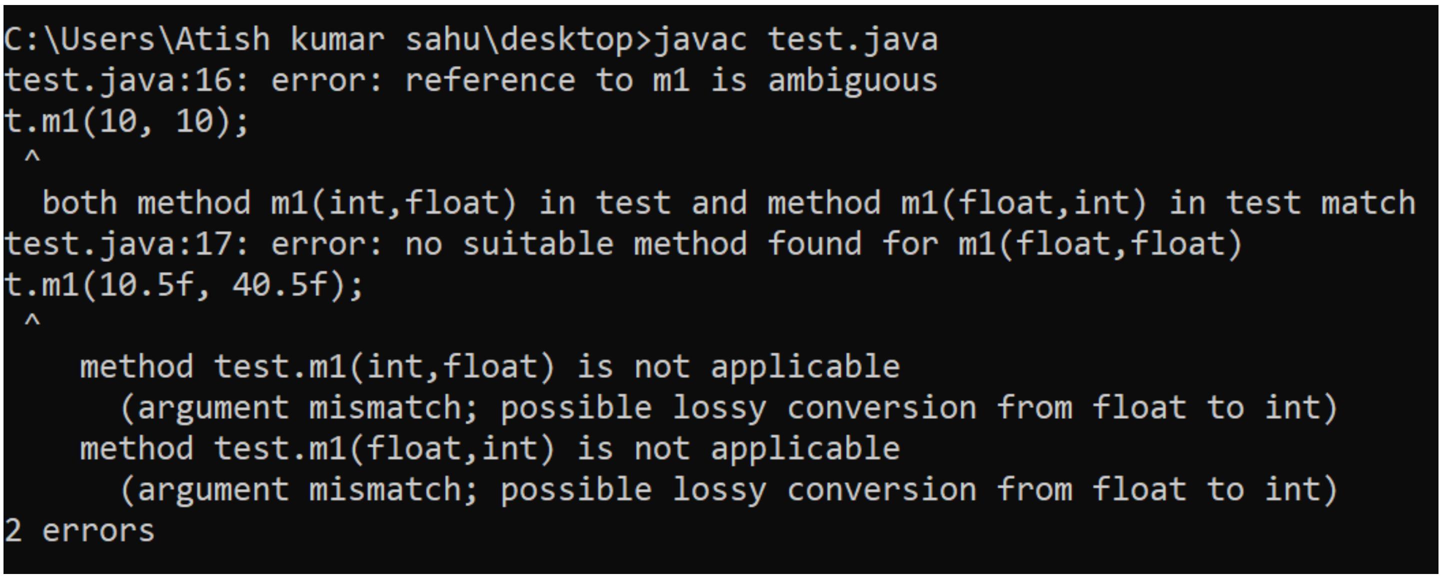
```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:16: error: reference to m1 is ambiguous
t.m1(null);
 ^
  both method m1(String) in test and method m1(StringBuffer) in test match
1 error
```



CASE 04

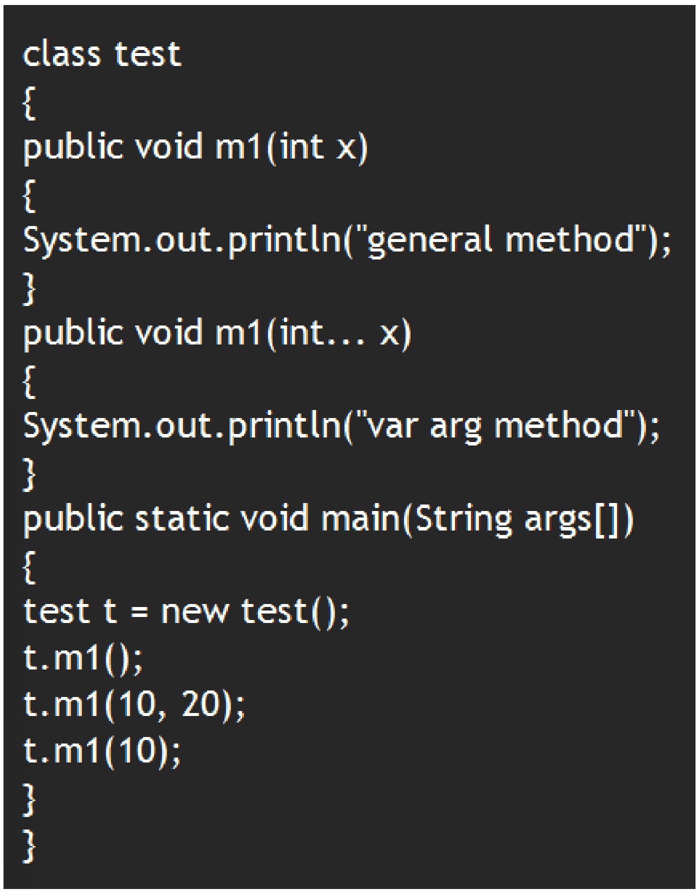


```
class test
{
public void m1(int i, float f)
{
System.out.println("INT FLOAT");
}
public void m1(float f, int i)
{
System.out.println("FLOAT INT");
}
public static void main(String args[])
{
test t = new test();
t.m1(10, 20.5f); //INT FLOAT
t.m1(30.5f, 50); //FLLOAT INT
t.m1(10, 10);
t.m1(10.5f, 40.5f);
}
}
```

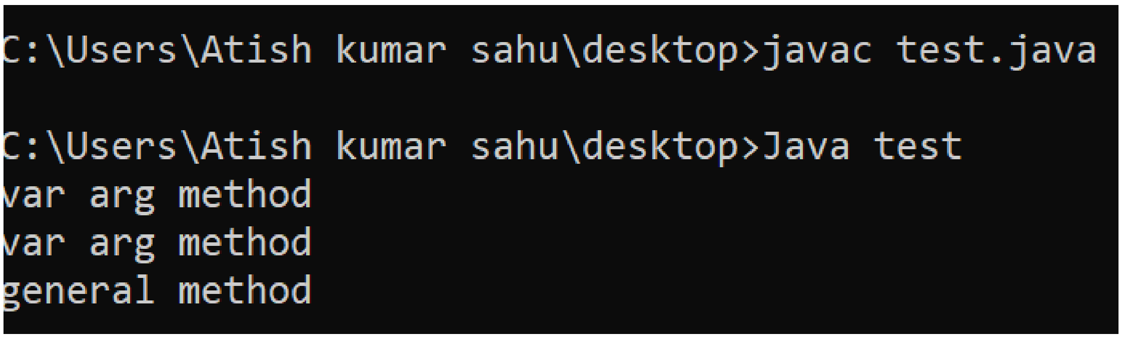


```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:16: error: reference to m1 is ambiguous
t.m1(10, 10);
 ^
  both method m1(int,float) in test and method m1(float,int) in test match
test.java:17: error: no suitable method found for m1(float,float)
t.m1(10.5f, 40.5f);
 ^
    method test.m1(int,float) is not applicable
      (argument mismatch; possible lossy conversion from float to int)
    method test.m1(float,int) is not applicable
      (argument mismatch; possible lossy conversion from float to int)
2 errors
```

CASE 05



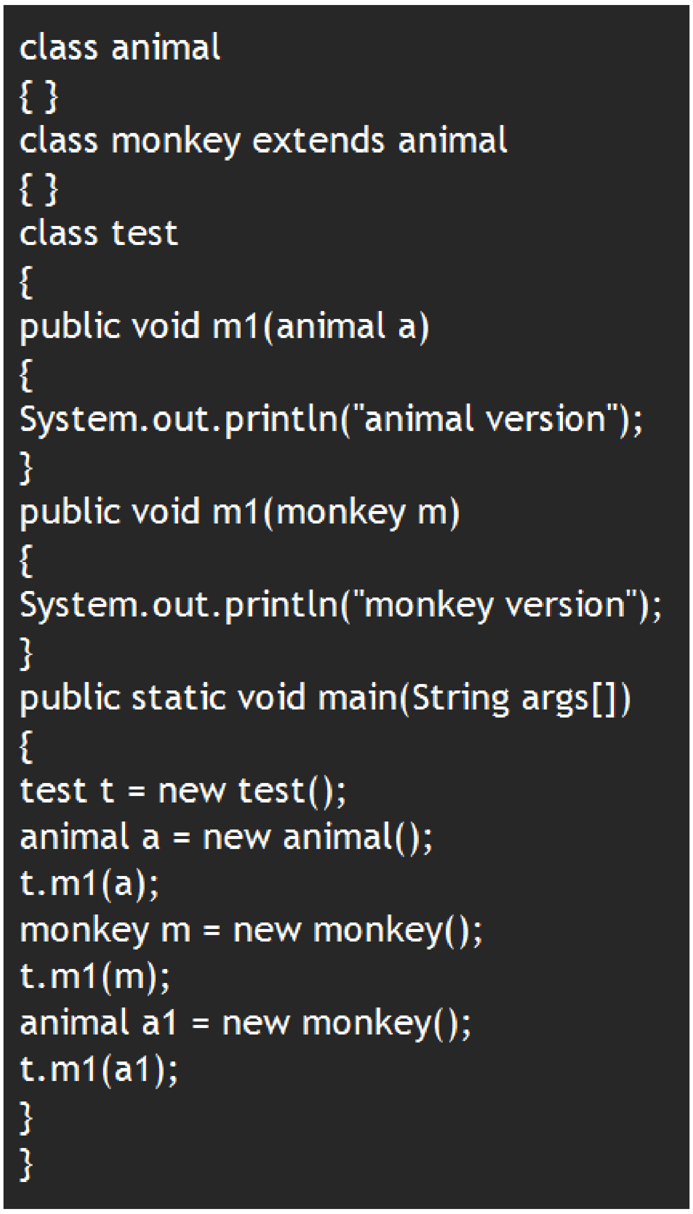
```
class test
{
public void m1(int x)
{
System.out.println("general method");
}
public void m1(int... x)
{
System.out.println("var arg method");
}
public static void main(String args[])
{
test t = new test();
t.m1();
t.m1(10, 20);
t.m1(10);
}
}
```



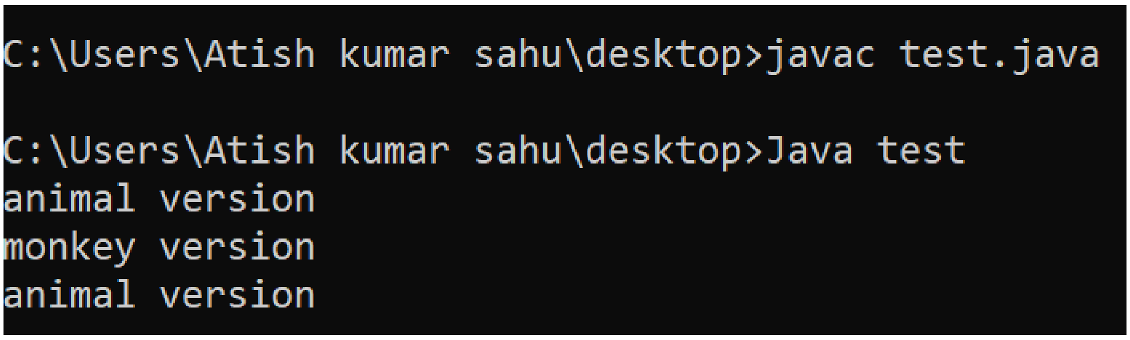
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>Java test
var arg method
var arg method
general method
```

IN GENERAL VAR ARG METHOD WILL GET LEAST PRIORITY THAT IS IF NO OTHER METHOD MATCHED THEN ONLY VAR ARG METHOD WILL GET THE CHANCE. IT IS EXACTLY SAME AS DEFAULT CASE INSIDE SWITCH.

CASE 06



```
class animal
{ }
class monkey extends animal
{ }
class test
{
public void m1(animal a)
{
System.out.println("animal version");
}
public void m1(monkey m)
{
System.out.println("monkey version");
}
public static void main(String args[])
{
test t = new test();
animal a = new animal();
t.m1(a);
monkey m = new monkey();
t.m1(m);
animal a1 = new monkey();
t.m1(a1);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>Java test
animal version
monkey version
animal version
```

IN OVERLOADING METHOD RESOLUTION ALWAYS TAKES CARE BY COMPILER BASED ON REFERENCE TYPE. IN OVERLOADING RUNTIME OBJECT WON'T PLAY ANY ROLE.

OVERRIDING:

OVERRIDING:

Whatever methods parent has bydefault available to child through inheritance. If child class not satisfied with parent class implementation, then child is allowed to redefine that method based on its requirements. This process is called overriding. The parent class method which is overriden is called overriden method and the child class method which is overriding is called overriding method.

```
class p
{
public void m1()
{
System.out.println("this is parent method 01");
}
public void m2()
{
System.out.println("this is parent method 02"); //overriden method
}

}
class c extends p
{
public void m2()
{
System.out.println("this is overriden parent method 02"); //overriding method
}

}
```

```
class test
{
public static void main(String args[])
{
p pp = new p();
pp.m2();
c cc = new c();
cc.m2();
p pp1 = new c();
pp1.m2();
}
}
```

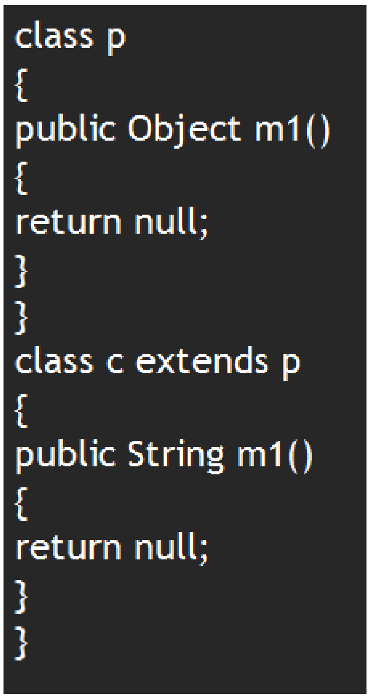
****

In overriding method resolution always takes care by JVM based on runtime object and hence overriding is also considered as "RUNTIME POLYMORPHISM", "DYNAMIC PLOYMORPHISM", "LATE BINDING".

RULES FOR OVERRIDING CONCEPT:

1. In overriding method names and argument types must be matched. That is method signature must be same.

2. In overriding return type must be same but this rule is applicable until 1.4 version only from 1.5 version onwards we can take covariant return types. according to this *child class method return type need not be same as parent method return type its child type also allowed.*



```
class p
{
public Object m1()
{
return null;
}
}
class c extends p
{
public String m1()
{
return null;
}
}
```

It is invalid until 1.4 version. From 1.5 version onwards it is valid.

NOTE:

If you want to run a java program in different version the code is

“javac -source version number classname.java”

PARENT CLASS METHOD RETURN TYPE = CHILD CLASS METHOD RETURN TYPE

OBJECT = OBJECT, STRING, STRINGBUFFER -> VALID

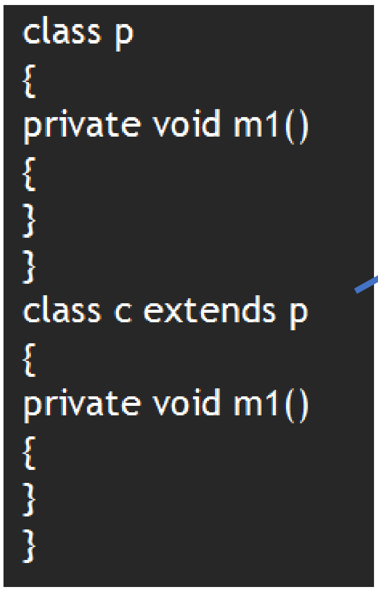
NUMBER = NUMBER, INTEGER, DOUBLE,…. -> VALID

STRING = OBJECT -> INVALID

DOUBLE = INT -> INVALID

COVARIANT RETURN TYPE CONCEPT APPLICABLE ONLY FOR OBJECT TYPES BUT NOT PREMITIVE TYPES.

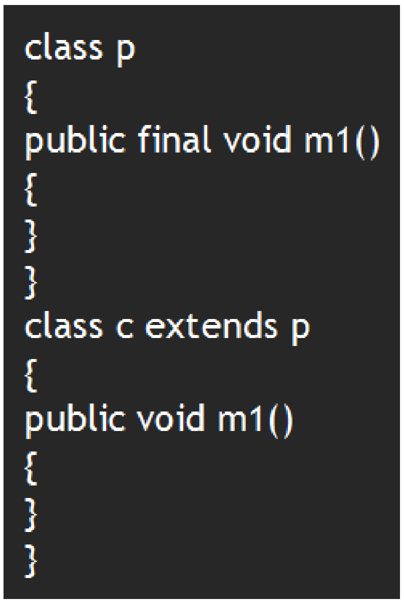
3. Parent class private methods not available to the child and hence overriding concept not applicable for private methods. Based on our requiredment we can define exact same private method in child class it is valid but it is not allowed for overriding concept.



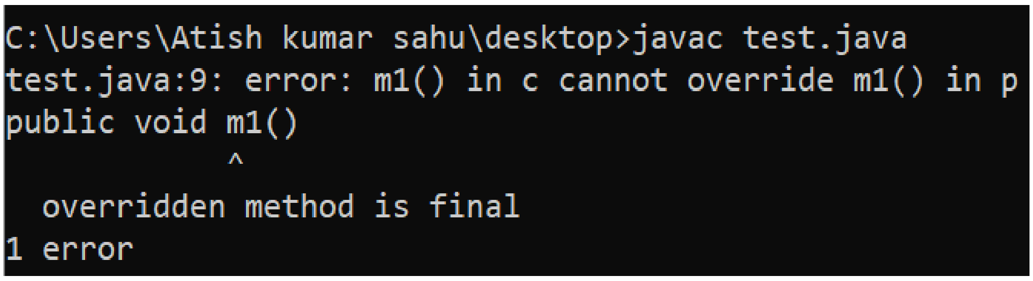
```
class p
{
private void m1()
{
}
}
class c extends p
{
private void m1()
{
}
}
```

VALID BUT NOT OVERRIDING.

4. We can't override parent class final method in child class. if we trying to override we will get compile time error.

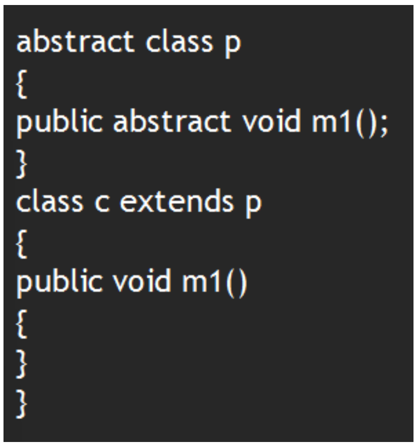


```
class p
{
public final void m1()
{
}
}
class c extends p
{
public void m1()
{
}
}
```



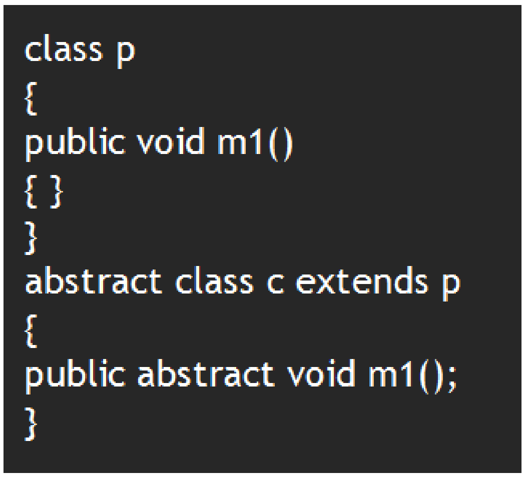
```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:9: error: m1() in c cannot override m1() in p
public void m1()
            ^
  overridden method is final
1 error
```

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



```
abstract class p
{
public abstract void m1();
}
class c extends p
{
public void m1()
{
}
}
```

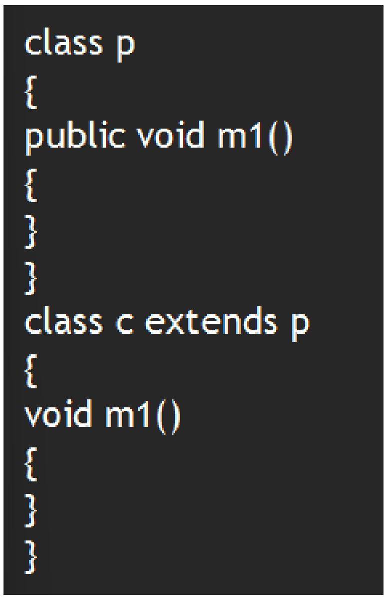
6. We can override non abstract method as abstract. Advantage of this approach is we can stop the availablity of parent method implementation to the next level child classes.



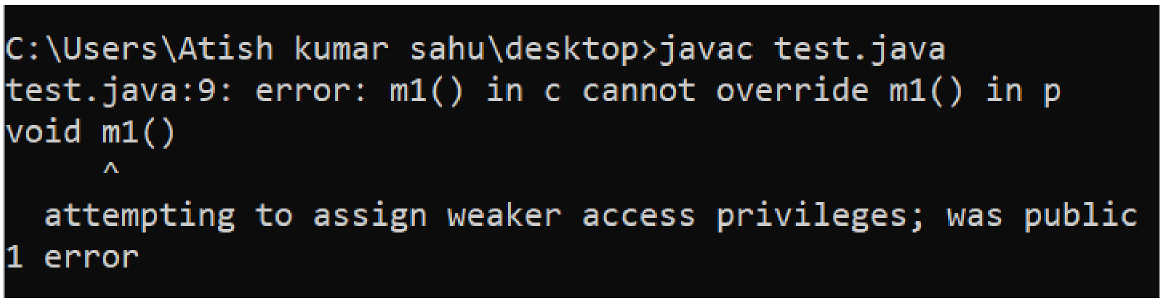
```
class p
{
public void m1()
{ }
}
abstract class c extends p
{
public abstract void m1();
}
```

<u>PARENT CLASS</u>	<u>CHILD CLASS</u>	
FINAL	FINAL/NON-FINAL	INVALID
NON-FINAL	FINAL	VALID
ABSTRACT	NON-ABSTRACT	VALID
NON-ABSTRACT	ABSTRACT	VALID
SYNCHRONIZED	NON-SYNCHRONIZED	VALID
NON-SYNCHRONIZED	SYNCHRONIZED	VALID
NATIVE	NON-NATIVE	VALID
NON-NATIVE	NATIVE	VALID
STRICTFP	NON-STRICTFP	VALID
NON-STRICTFP	STRICTFP	VALID

In overriding the following modifiers won't keep any restriction. Synchronized, native, strictfp.



```
class p
{
public void m1()
{
}
}
class c extends p
{
void m1()
{
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:9: error: m1() in c cannot override m1() in p
void m1()
     ^
  attempting to assign weaker access privileges; was public
1 error
```

7. IN ABOVE TWO IMAGE WHILE OVERRIDING WE CAN'T REDUCE SCOPE OF ACCESS MODIFIER. BUT WE CAN INCREASE THE SCOPE.

PRIVATE < DEFAULT < PROTECTED < PUBLIC

PRIVATE IS MOST RESTRICTED MODIFIER AND PUBLIC IS LESS RESTRICTED MODIFIER.

<u>PARENT CLASS METHOD</u>	<u>CHILD CLASS METHOD</u>
Public	Public
Protected	Protected, Public
Default	Default, Protected, Public
Private ->	Overriding concept is not applicable for private methods

1. VALID

P: public void m1() throws Exception

C: public void m1()

2. INVALID

P: public void m1()

C: public void m1() throws Exception

3. VALID

P: public void m1() throws Exception

C: public void m1() throws IOException

4. INVALID

P: public void m1() throws IOException

C: public void m1() throws Exception

5. VALID

P: public void m1() throws IOException

C: public void m1() throws FileNotFoundException, EOFException

6. INVALID

P: public void m1() throws IOException

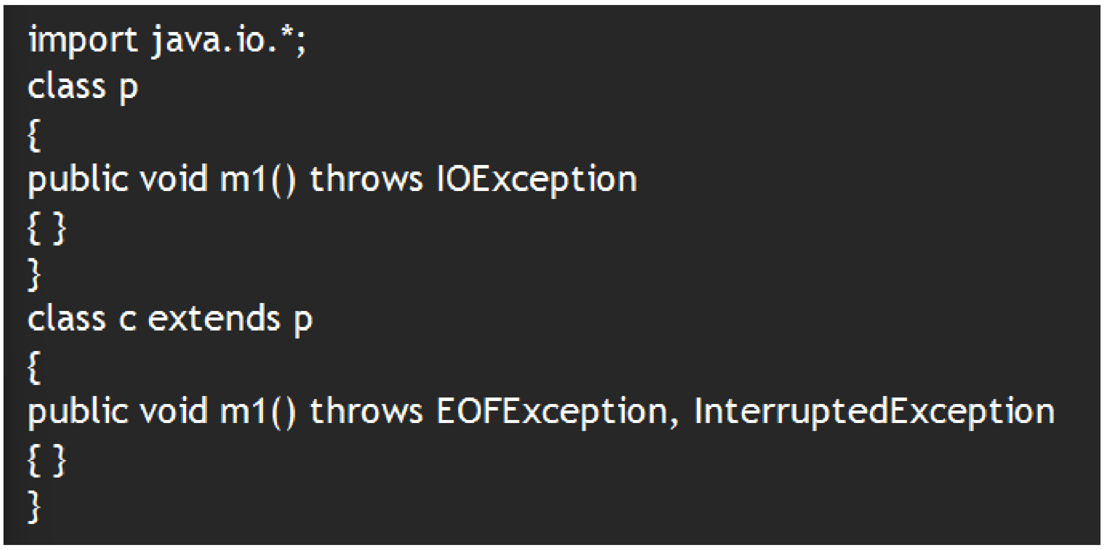
C: public void m1() throws EOFexception, InterruptedException

7. VALID

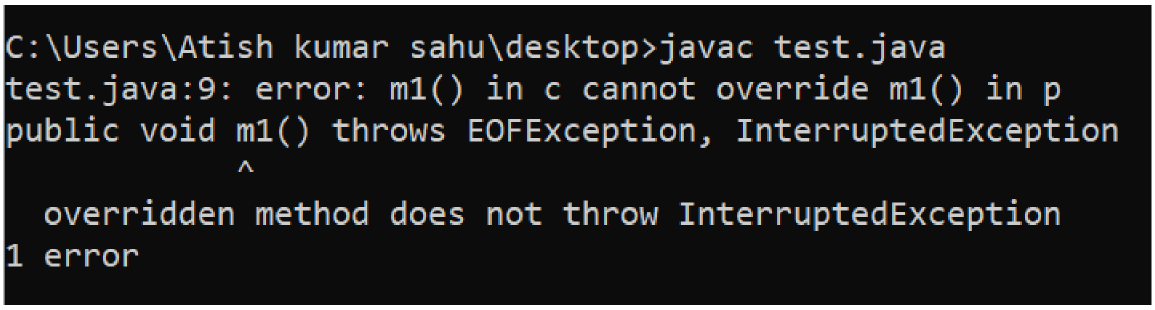
P: public void m1() throws IOException

C: public void m1() throws AE, NE, CCE

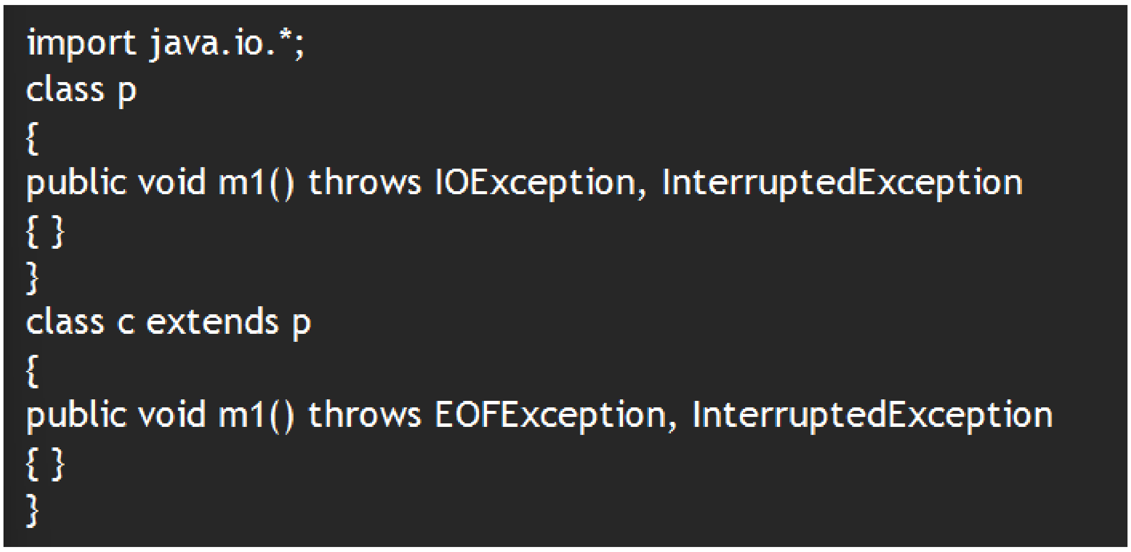
8. *From above 7 options we understand. If child class method throws any checked exception compulsory parent class method should throw the same checked exception or it's parent. Otherwise we will get compile time error. But there are no restriction for unchecked exception.*



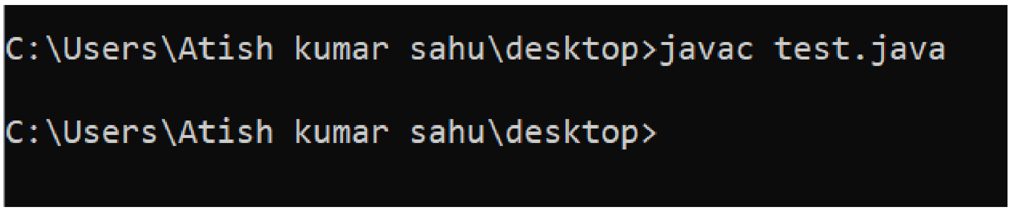
```java
import java.io.*;
class p
{
public void m1() throws IOException
{ }
}
class c extends p
{
public void m1() throws EOFException, InterruptedException
{ }
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:9: error: m1() in c cannot override m1() in p
public void m1() throws EOFException, InterruptedException
            ^
  overridden method does not throw InterruptedException
1 error
```



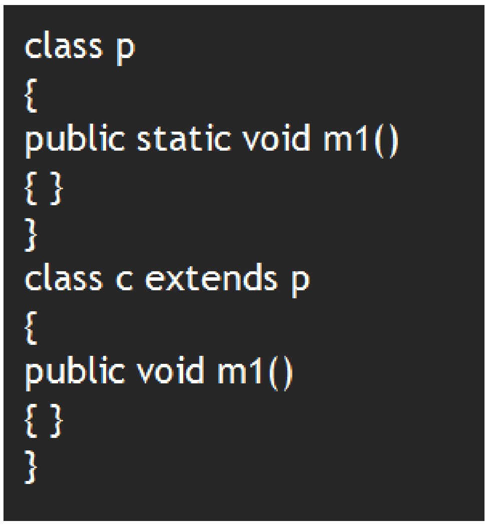
```java
import java.io.*;
class p
{
public void m1() throws IOException, InterruptedException
{ }
}
class c extends p
{
public void m1() throws EOFException, InterruptedException
{ }
}
```



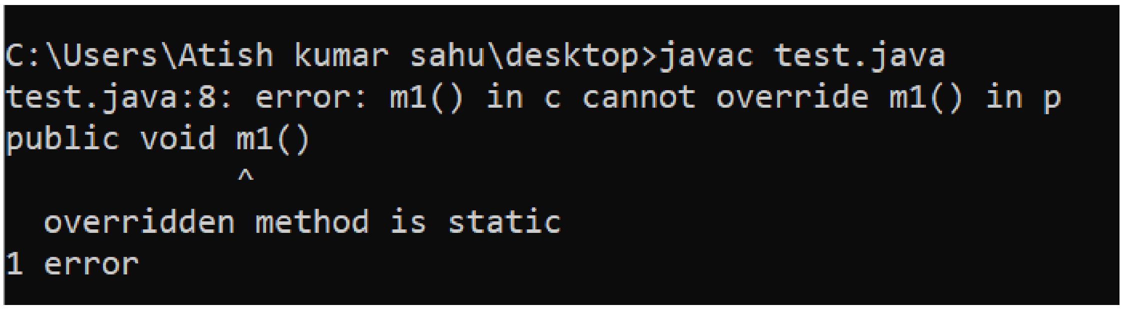
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>
```

OVERRIDING WITH RESPECT TO STATIC METHODS:

1. We can’t override a static method as non-static. Otherwise wen will get compile time error.

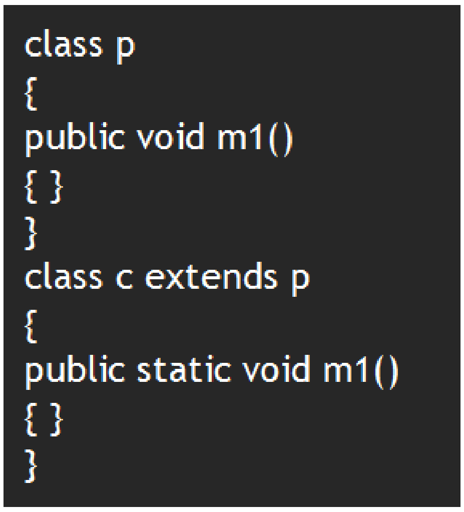


```
class p
{
public static void m1()
{ }
}
class c extends p
{
public void m1()
{ }
}
```

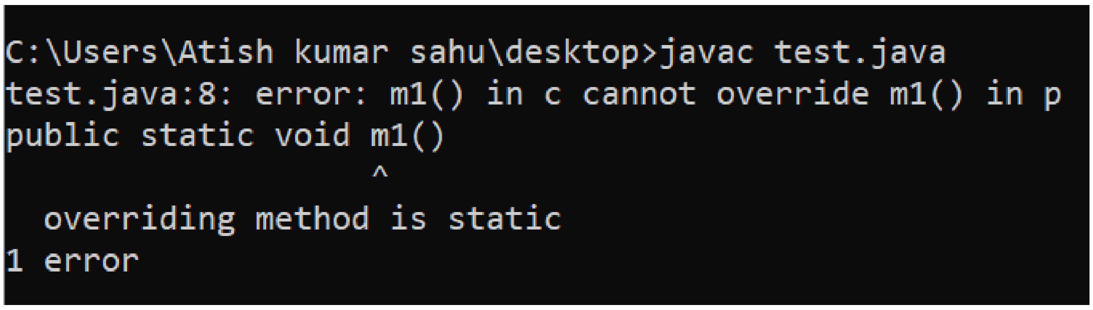


```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:8: error: m1() in c cannot override m1() in p
public void m1()
            ^
  overridden method is static
1 error
```

2. We can’t override a non-static method as static.

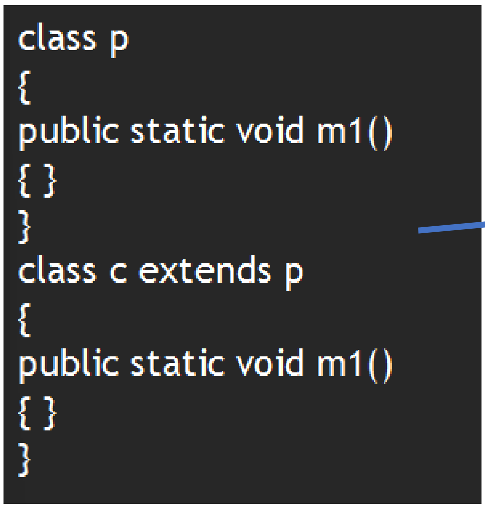


```
class p
{
public void m1()
{ }
}
class c extends p
{
public static void m1()
{ }
}
```



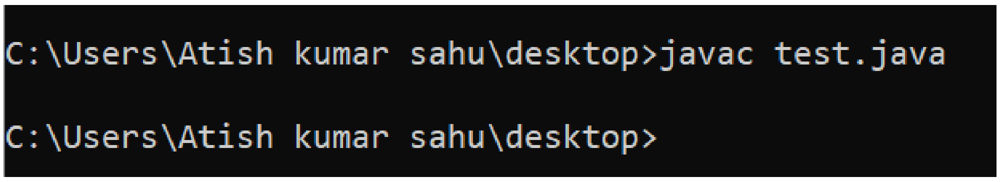
```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:8: error: m1() in c cannot override m1() in p
public static void m1()
                   ^
  overriding method is static
1 error
```

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



```
class p
{
public static void m1()
{ }
}
class c extends p
{
public static void m1()
{ }
}
```

It is not OVERRIDING it is METHOD HIDING.



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>
```

<u>METHOD HIDING VS METHOD OVERRIDING:</u>

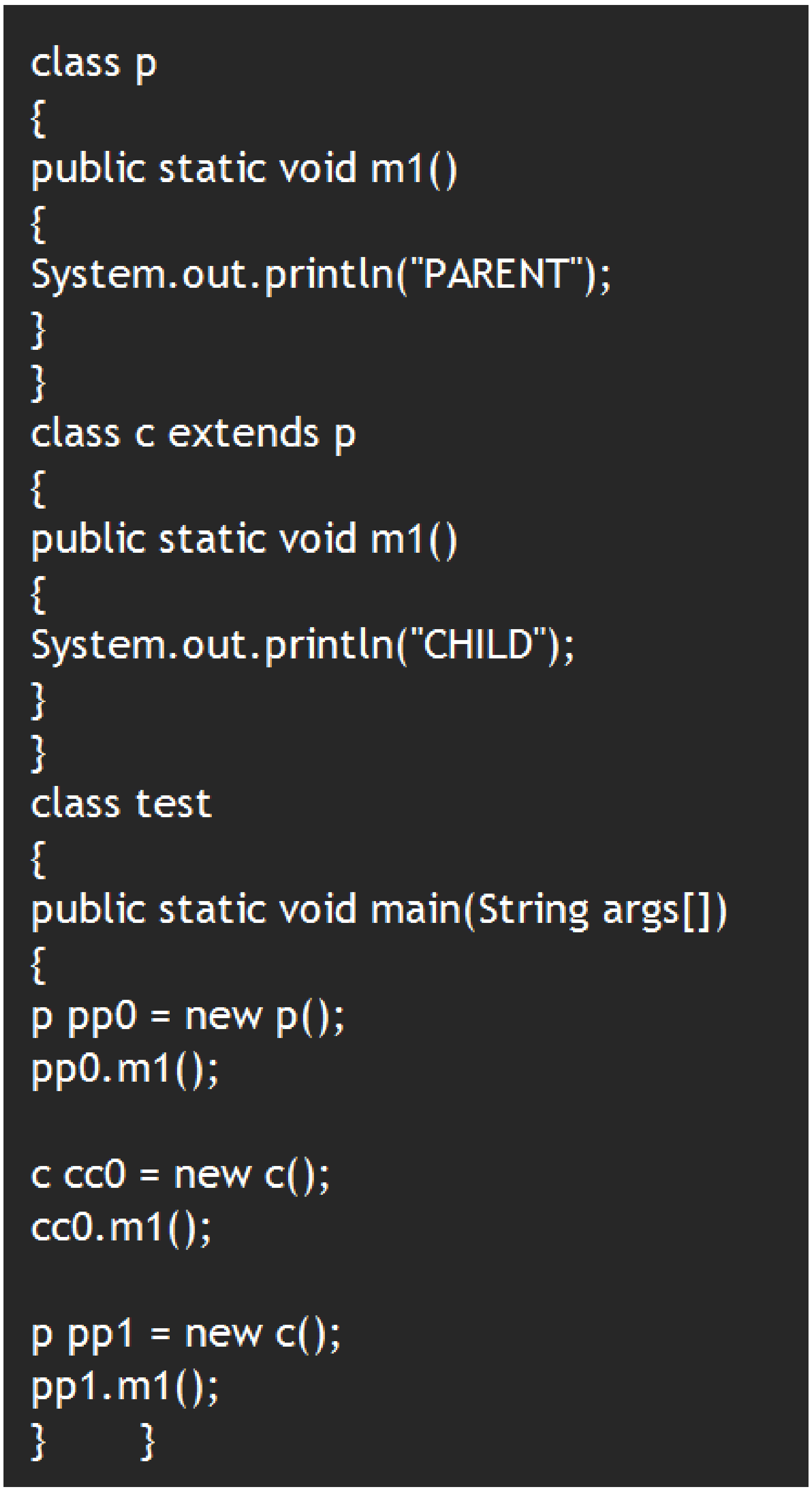
All rules of METHOD HIDING is exactly same as OVERRIDING except the following differences.

1. In METHOD HIDING both parent and child class method should be static. In OVERRIDING both parent and child class method should be non-static.

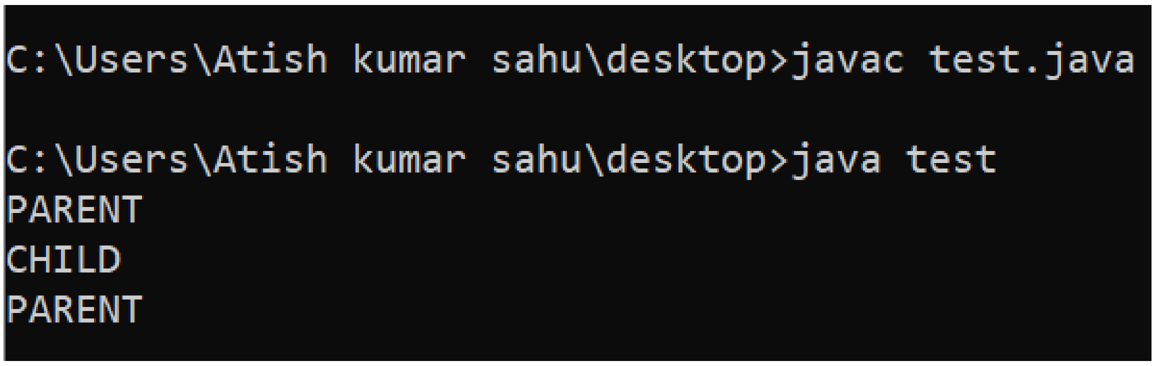
2. In METHOD HIDING method resolution is maintained by compiler based on reference type. in OVERRIDING method resolution is maintained by JVM based on runtime object.

3. METHOD HIDING is also known as "COMPILE TIME POLYMORPHISM", "STATIC POLYMORPHISM", "EARLY BINDING". OVERRIDING is also known as RUNTIME POLYMORPHISM, DYNAMIC POLYMORPHISM, LATE BINDING.

METHOD HIDING:

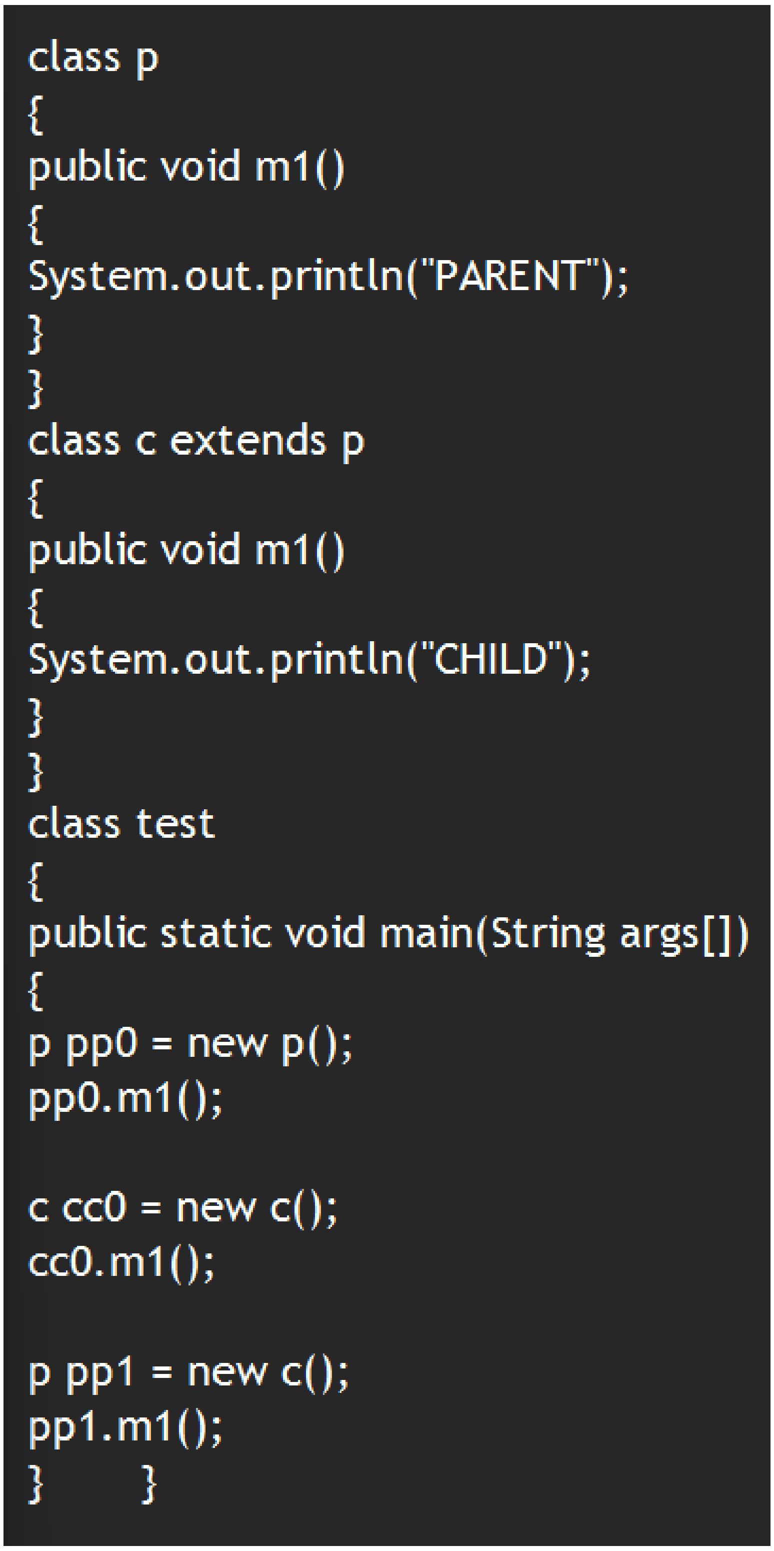


```java
class p
{
public static void m1()
{
System.out.println("PARENT");
}
}
class c extends p
{
public static void m1()
{
System.out.println("CHILD");
}
}
class test
{
public static void main(String args[])
{
p pp0 = new p();
pp0.m1();

c cc0 = new c();
cc0.m1();

p pp1 = new c();
pp1.m1();
}      }
```

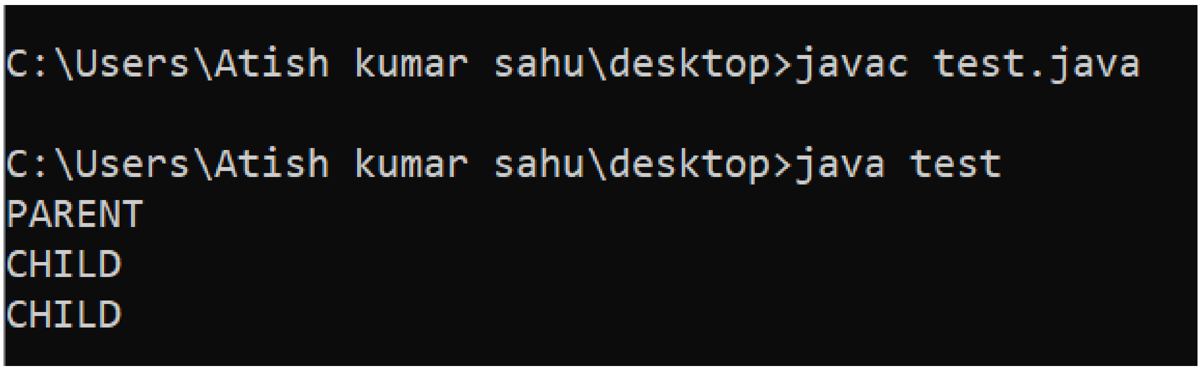


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
PARENT
CHILD
PARENT
```

OVERRIDING:



```java
class p
{
public void m1()
{
System.out.println("PARENT");
}
}
class c extends p
{
public void m1()
{
System.out.println("CHILD");
}
}
class test
{
public static void main(String args[])
{
p pp0 = new p();
pp0.m1();

c cc0 = new c();
cc0.m1();

p pp1 = new c();
pp1.m1();
}     }
```



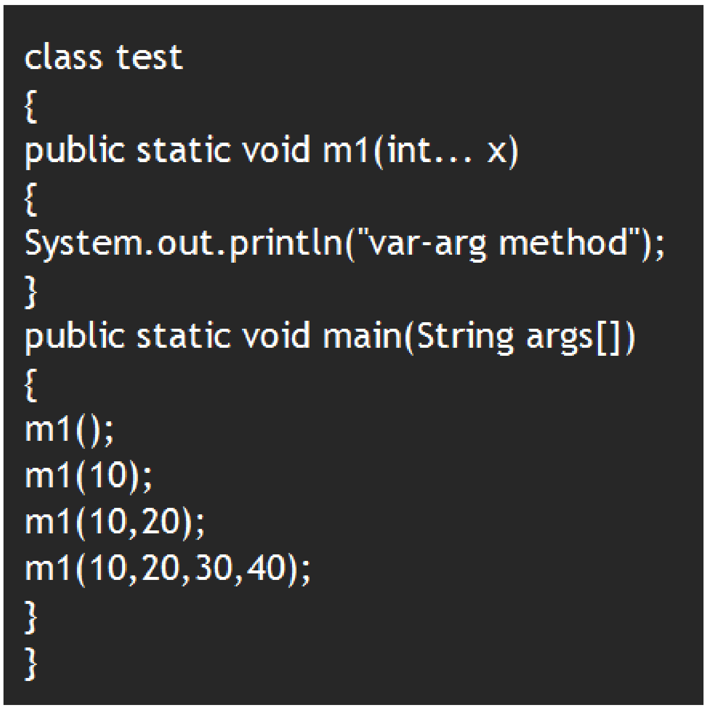
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
PARENT
CHILD
CHILD
```

VAR-ARG METHODS

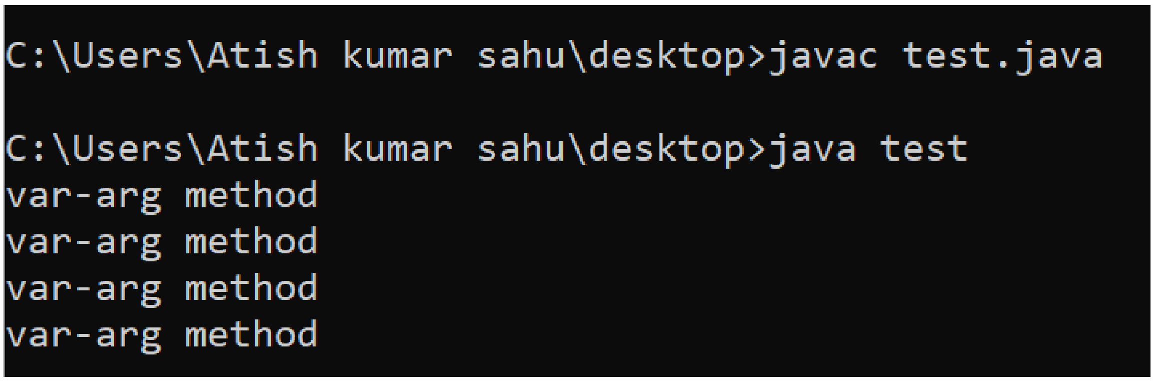
VAR-ARG METHODS: (VARIABLE NUMBER OF ARGUMENT METHODS)

Untill 1.4 version we can’t decalre a method with variable number of arguments. If there is a change in number of arguments compulsory we should go for new method. It increases length of the code and reduces redability. To overcome this problem sun people introduced var-arg method in 1.5 version. According to this we can declare a method which can take variable number of arguments. Such type of methods are called var-arg methods.

We can declare a var-arg method as follows “***m1(int… x)***” we can call this method by passing any number of int values including 0 number. m1();, m1(10);, m1(10,20,30,40);

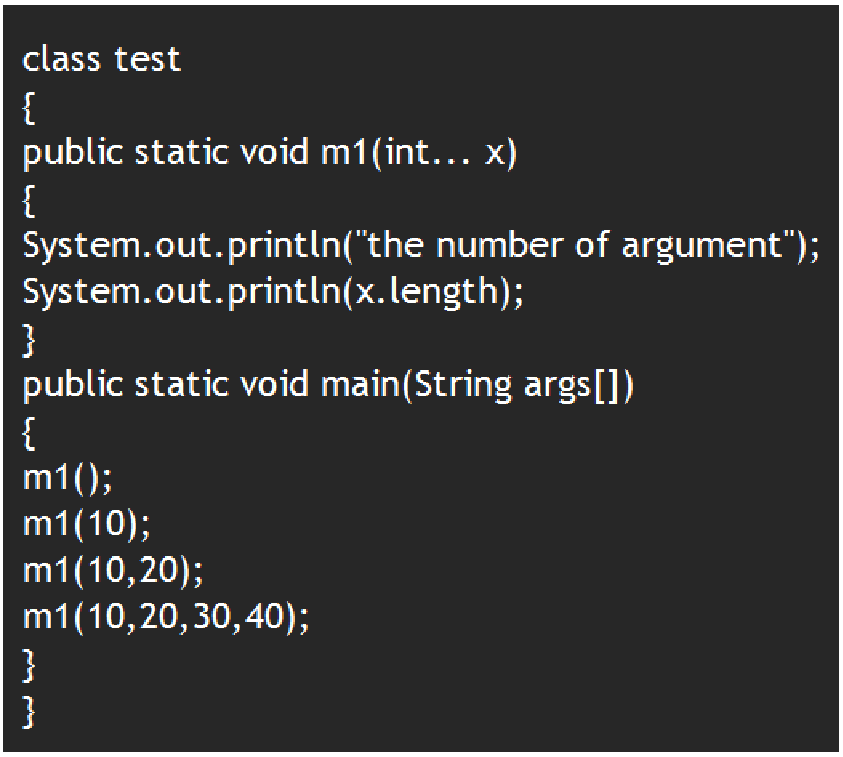


```
class test
{
public static void m1(int... x)
{
System.out.println("var-arg method");
}
public static void main(String args[])
{
m1();
m1(10);
m1(10,20);
m1(10,20,30,40);
}
}
```

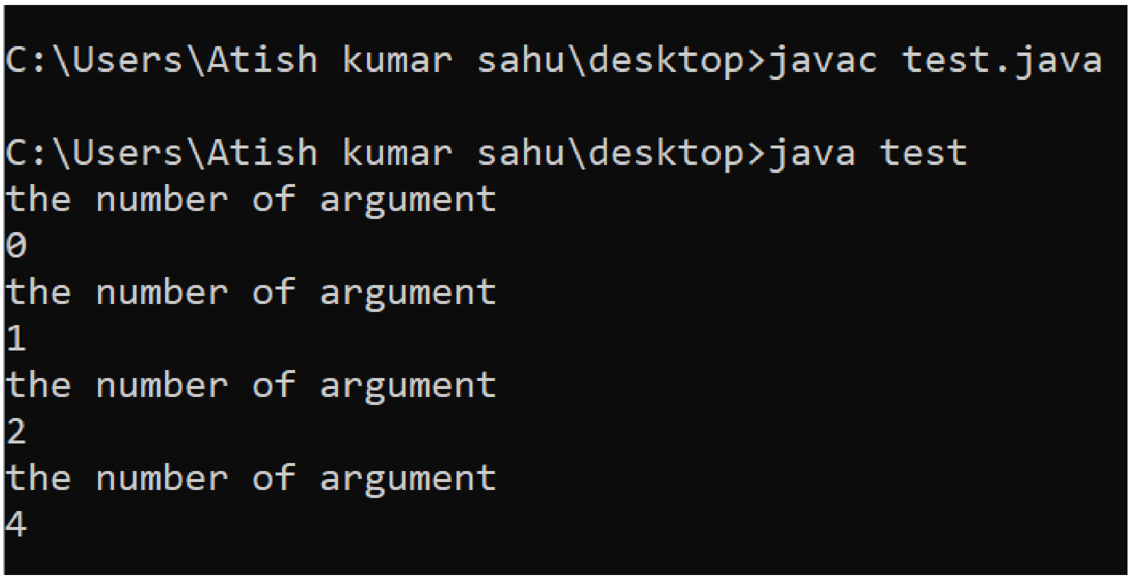


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
var-arg method
var-arg method
var-arg method
var-arg method
```

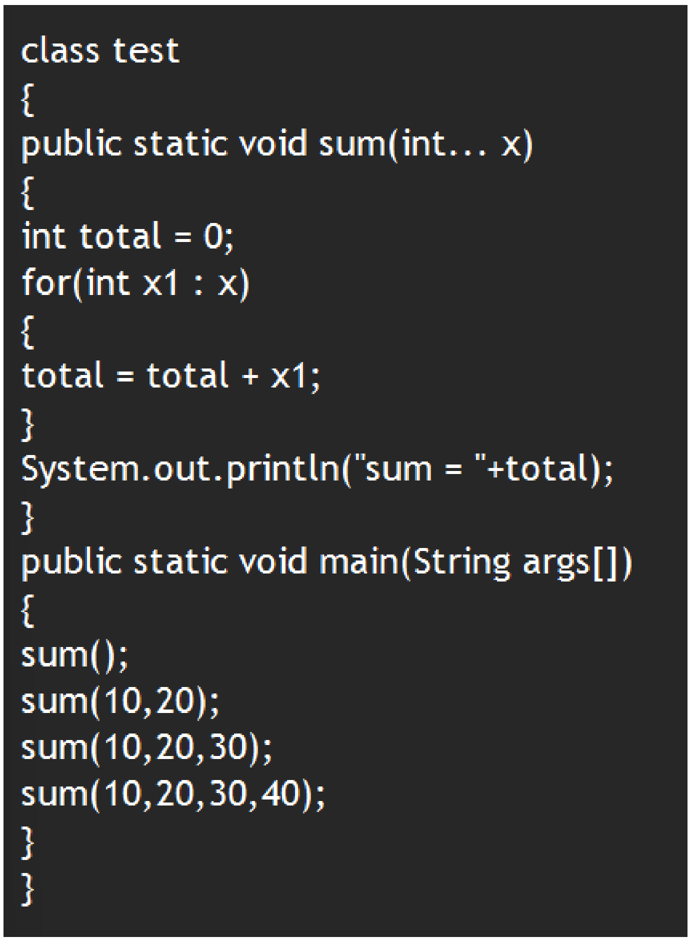
Internally var-arg parameter will be converted into one-dimensional array. hence with in the var-arg method we can differentiate values by using index.



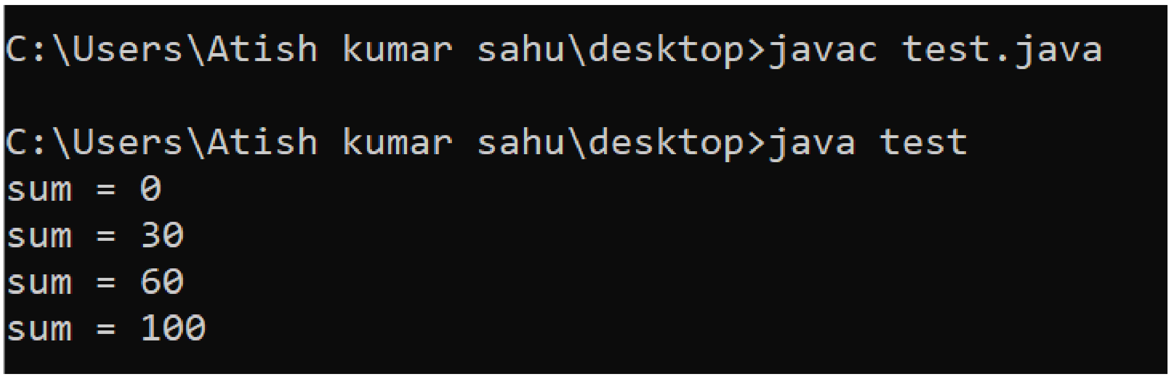
```java
class test
{
public static void m1(int... x)
{
System.out.println("the number of argument");
System.out.println(x.length);
}
public static void main(String args[])
{
m1();
m1(10);
m1(10,20);
m1(10,20,30,40);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
the number of argument
0
the number of argument
1
the number of argument
2
the number of argument
4
```



```
class test
{
public static void sum(int... x)
{
int total = 0;
for(int x1 : x)
{
total = total + x1;
}
System.out.println("sum = "+total);
}
public static void main(String args[])
{
sum();
sum(10,20);
sum(10,20,30);
sum(10,20,30,40);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
sum = 0
sum = 30
sum = 60
sum = 100
```

VAR-ARG CASE 01:

Q. WHICH ARE THE FOLLOWING ARE VALID VAR-ARG METHOD DECLARATION?

```
public static void m1(int... x)//valid
{ }
public static void m2(int    ...x)//valid
{ }
public static void m3(int...x)//valid
{ }
```

```
//public static void m4(int    x...)//invalid
{ }
//public static void m5(int.   ..x)//invalid
{ }
//public static void m6(int    .x..)//invalid
{ }
```

VAR-ARG CASE 02:

We can mix var-arg parameter with normal parameter.

```
class test
{
public static void m1(int x, int... y)
{ }
public static void m2(String s, double... d)
{ }
public static void main(String args[])
{

}
}
```

VAR-ARG CASE 03:

If we mix normal parameter with var-arg parameter then var-arg parameter should be last parameter.

```
public static void m1(int... y, int x) //invalid
{ }
public static void m2(double... d, String s) //invalid
{ }
```

VAR-ARG CASE 04:

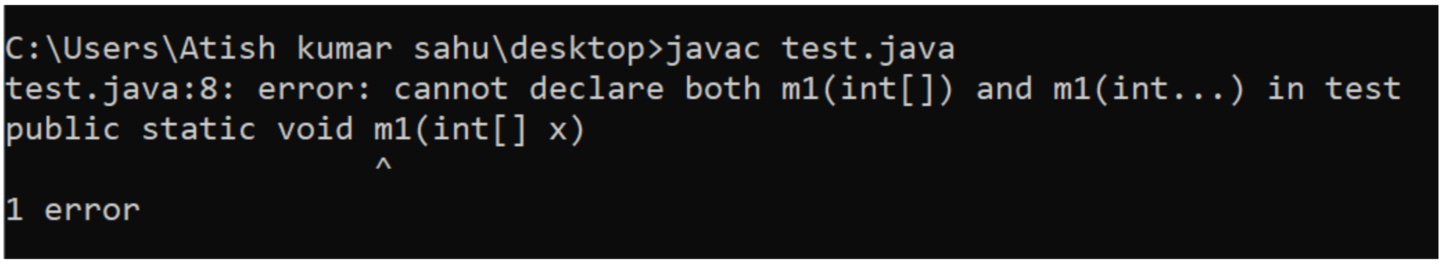
Inside var-arg method we can take only one var-arg parameter. We can’t take more than one var-arg parameter.

```
public static void m1(int... x, double... d) //invalid
{ }
```

VAR-ARG CASE 05:

Inside a class we can’t declare var-arg method and corresponding one-dimensional array method simultaniously otherwise we will get compile time error.

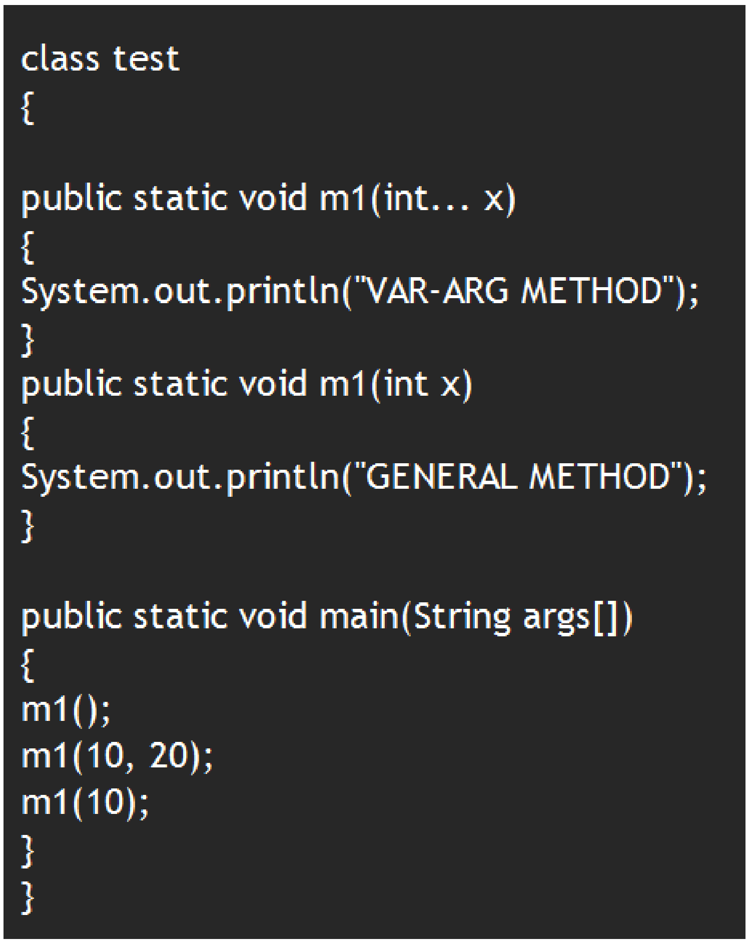
```
class test
{

public static void m1(int... x)
{
System.out.println("int...");
}
public static void m1(int[] x)
{
System.out.println("int[]");
}

}
```



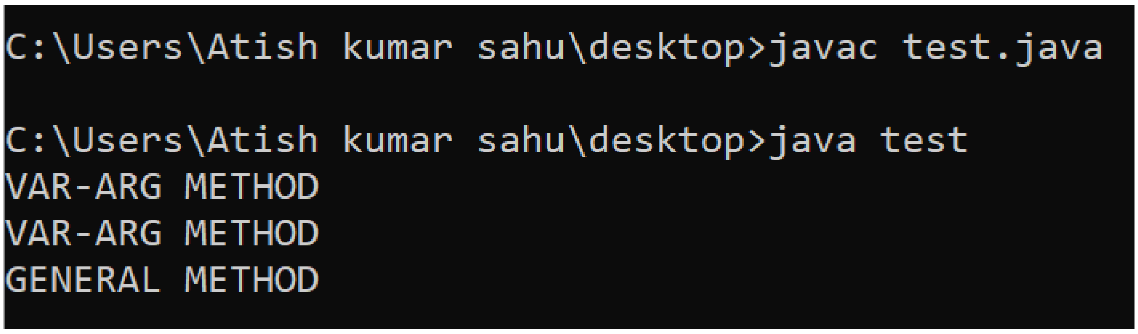
```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:8: error: cannot declare both m1(int[]) and m1(int...) in test
public static void m1(int[] x)
                   ^
1 error
```

VAR-ARG CASE 06:

In general var-arg method will get least priority that is if no other method matched then only var-arg method will get the chance. It is exactly same as default case inside switch concept.



```java
class test
{

public static void m1(int... x)
{
System.out.println("VAR-ARG METHOD");
}
public static void m1(int x)
{
System.out.println("GENERAL METHOD");
}

public static void main(String args[])
{
m1();
m1(10, 20);
m1(10);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
VAR-ARG METHOD
VAR-ARG METHOD
GENERAL METHOD
```

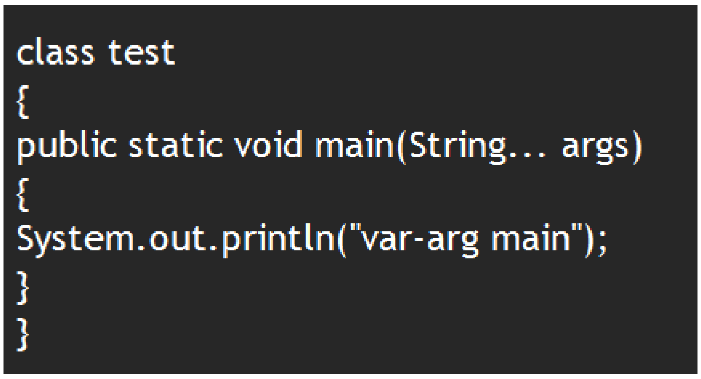
EQUIVALANCE BETWEEN VAR-ARG PARAMETER AND ONE-DIMENSIONAL ARRAY:

CASE 01:

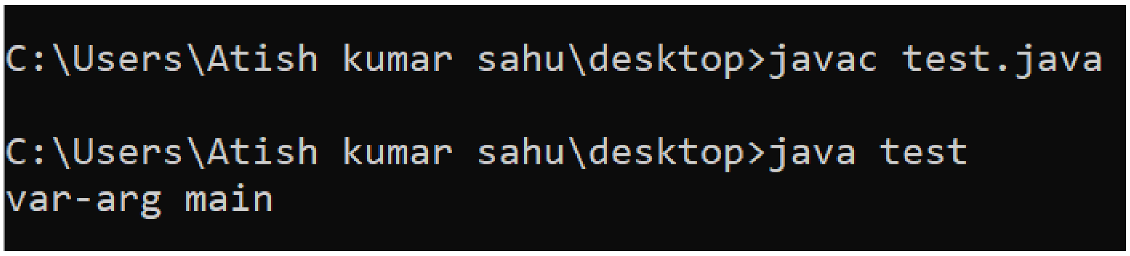
Where ever one-dimensional array present we can replace with var-arg parameter.

m1(int [] x) => m1(int… x)

EX: main(String args[]) => main(String… args)



```
class test
{
public static void main(String... args)
{
System.out.println("var-arg main");
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
var-arg main
```

CASE 02:

Wherever var-arg parameter present we can’t replace with one-dimensional array.

m1(int… x) => m1(int[] x) -> invalid statement

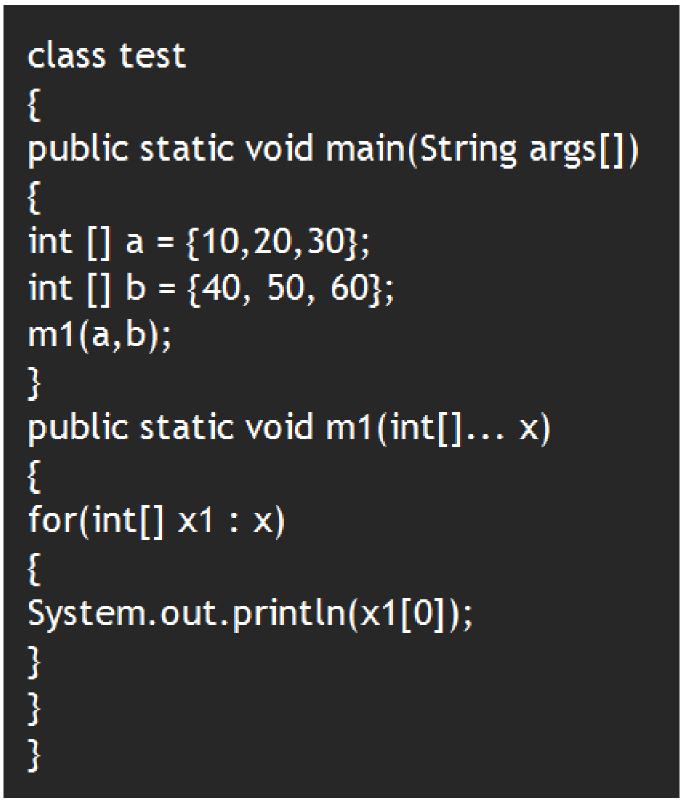
NOTE:

m1(int… x) => int[] x

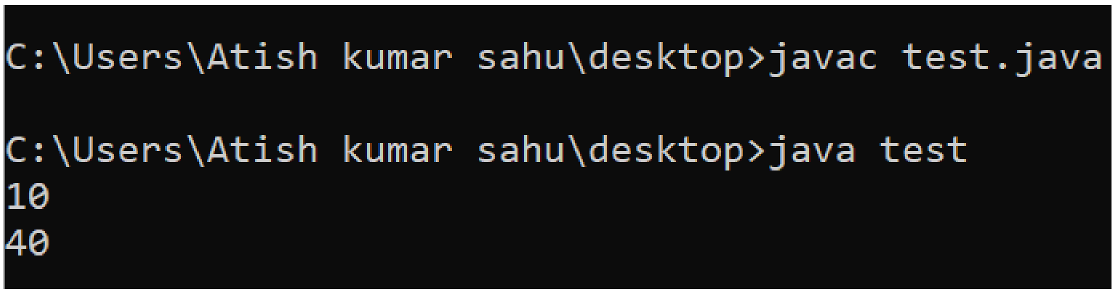
we can call this method by passing a group of int values and x will become one-dimensional array.

m1(int[]… x) => int [][] x

we can call this method by passing a group of one-dimensional int array and x will become two dimensional int array.



```java
class test
{
public static void main(String args[])
{
int [] a = {10,20,30};
int [] b = {40, 50, 60};
m1(a,b);
}
public static void m1(int[]... x)
{
for(int[] x1 : x)
{
System.out.println(x1[0]);
}
}
}
```

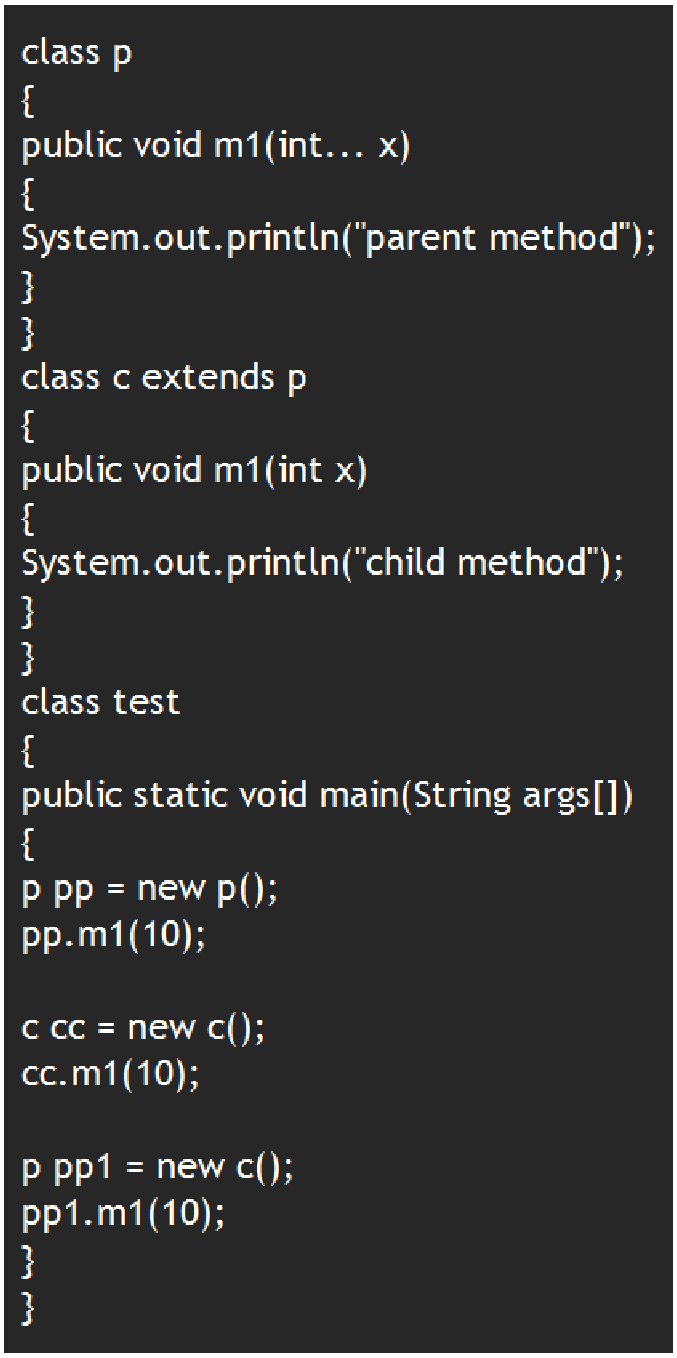


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
10
40
```

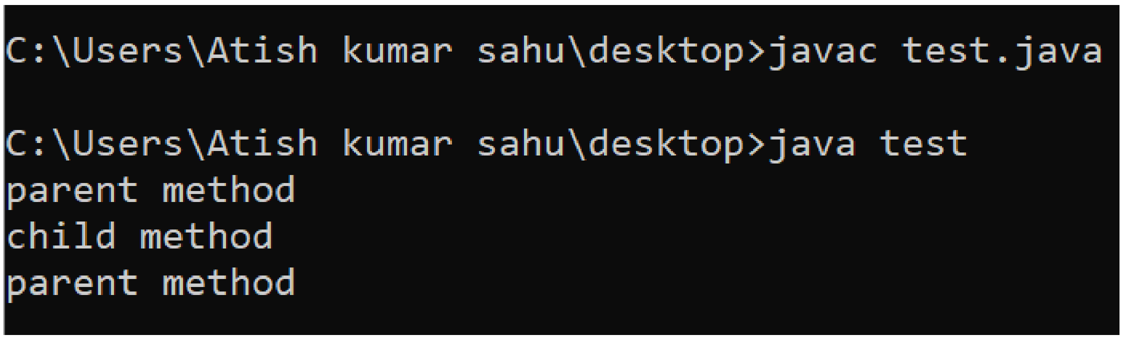
OVERRIDING WITH RESPECT TO VAR-ARG METHOD

CASE 01:

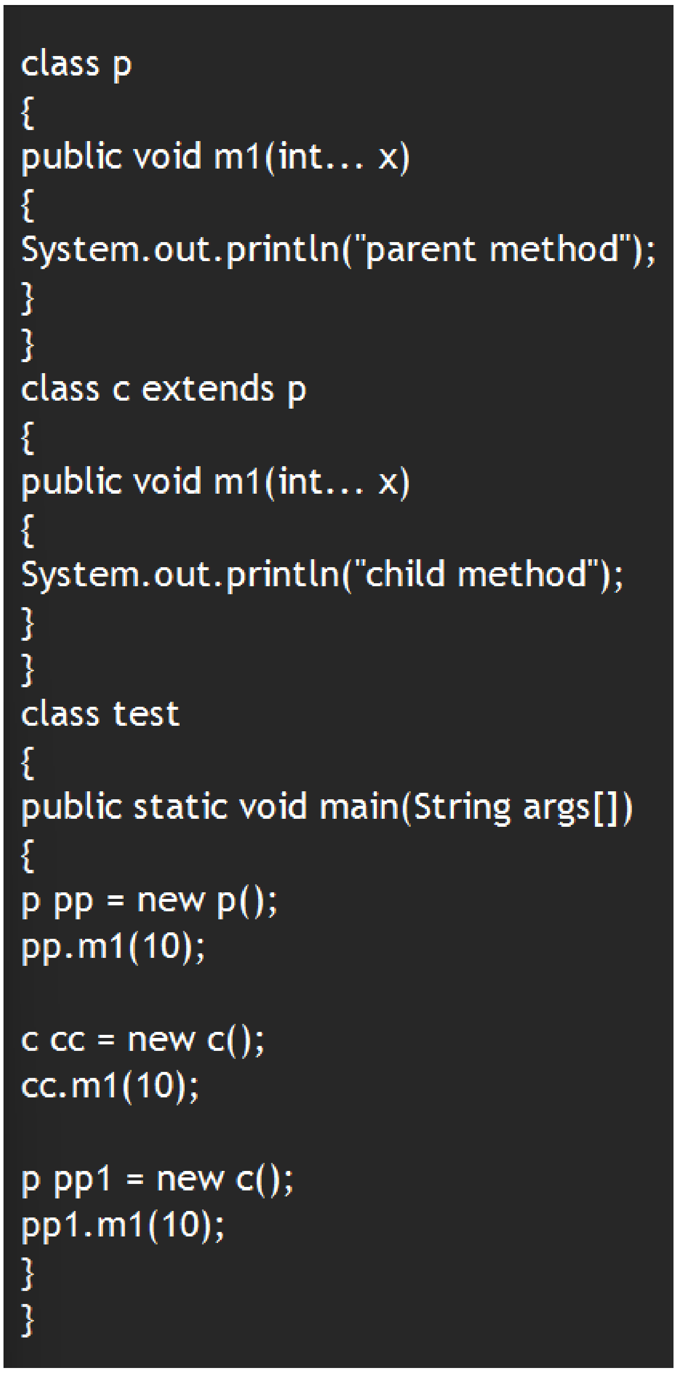
We can override var-arg method with another var-arg method only. If we are trying to override with normal method, then it will become overloading but not overriding. In the below program if we replace child method with var-arg method then it will become overriding. In this case out put is “parent, child, child”.



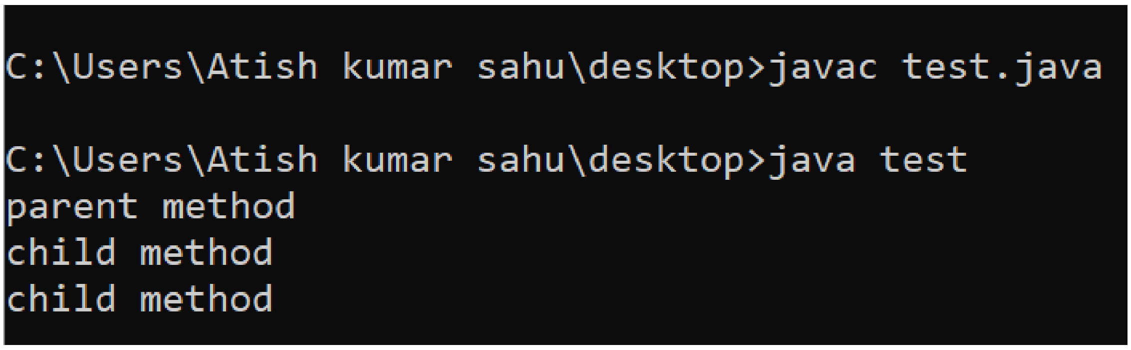
```
class p
{
public void m1(int... x)
{
System.out.println("parent method");
}
}
class c extends p
{
public void m1(int x)
{
System.out.println("child method");
}
}
class test
{
public static void main(String args[])
{
p pp = new p();
pp.m1(10);

c cc = new c();
cc.m1(10);

p pp1 = new c();
pp1.m1(10);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
parent method
child method
parent method
```



```
class p
{
public void m1(int... x)
{
System.out.println("parent method");
}
}
class c extends p
{
public void m1(int... x)
{
System.out.println("child method");
}
}
class test
{
public static void main(String args[])
{
p pp = new p();
pp.m1(10);

c cc = new c();
cc.m1(10);

p pp1 = new c();
pp1.m1(10);
}
}
```

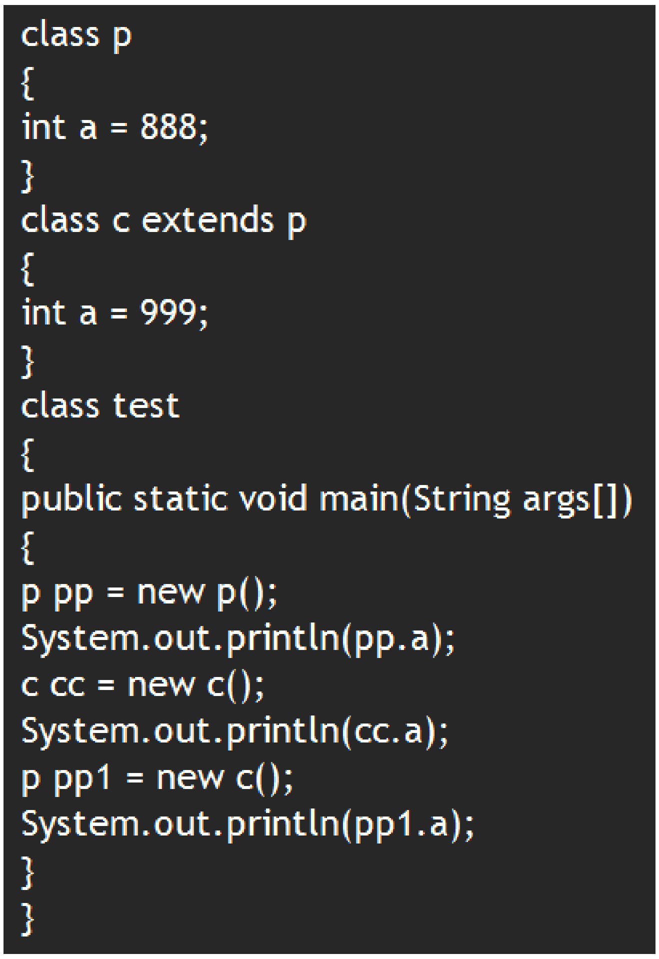


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
parent method
child method
child method
```

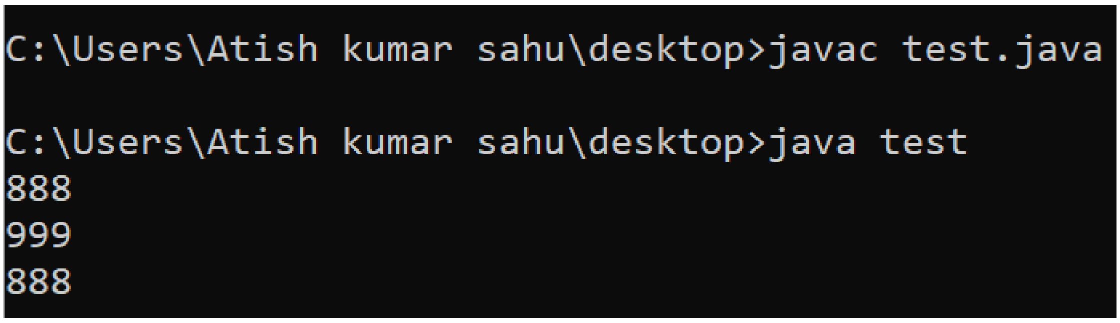
OVERRIDING WITH RESPECT TO VARIABLES

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

Parent class: non-static   Child class: non-static   = 888, 999, 888



```
class p
{
int a = 888;
}
class c extends p
{
int a = 999;
}
class test
{
public static void main(String args[])
{
p pp = new p();
System.out.println(pp.a);
c cc = new c();
System.out.println(cc.a);
p pp1 = new c();
System.out.println(pp1.a);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
888
999
888
```

P: static	c: non-static	= 888, 999, 888
P: non-static	c = static	= 888, 999, 888
P: static	c = static	= 888, 999, 888

OVERLOADING VS OVERRIDING

1. METHOD NAME:

In overloading method name must be same.

In overriding method name must be same.

2. ARGUMENT TYPES:

In overloading argument types must be different (atleast order).

In overriding argument types must be same (including order).

3. METHOD SIGNATURES:

In overloading method signatures must be different.

In overriding method signatures must be same.

4. RETURN TYPES:

In overloaiding return types has no restriction.

In overriding return type is must be same until 1.4v. from 1.5v covariant return types are allowed.

5. PRIVATE, STATIC, FINAL METHOD:

In overloading private, static, final method can be overlaoded.

In overriding private, static, final methods can’t be overriden.

6. ACCESS MODIFIERS:

In overloading there is no restriction on acccess modifier.

In overriding the scope of access modifier can’t be reduced but we can increase.

7. THROWS CLASS:

In overloading there is no restriction on throws class.

in overriding if child class method throws any checked exception compulsory parent class method should throw the same checked exception or it’s parent. But no restriction for unchecked exception.

8. METHOD RESOLUTION:

In overloading method resolution always takes care by compiler based reference type.

in overriding method resolution always takes care by JVM based on runtime object.

9. ALSO KNOWN AS:

Overloading also known as “Compile Time Polymorphism” “Static Polymorphism” “Early Binding”. Overriding also known as “Runtime Polymorphism” “Dynamic Polymorphism” “Late Binding”.

IN OVERLOADING WE HAVE TO CHECK ONLY METHOD NAMES(MUST BE SAME) AND ARGUMENT TYPES(MUST BE DIFFERENT). WE ARE NOT REQUIRED TO CHECK REMAINING LIKE RETURN TYPES, ACCESS MODIFIERS, ETC. IN OVERRIDING EVERY THING WE HAVE TO CHECK LIKE METHOD NAME, ARGUMENT TYPES, RETURN TYPES, ACCESS MODIFIERS, THROWS CLASS, ETC.

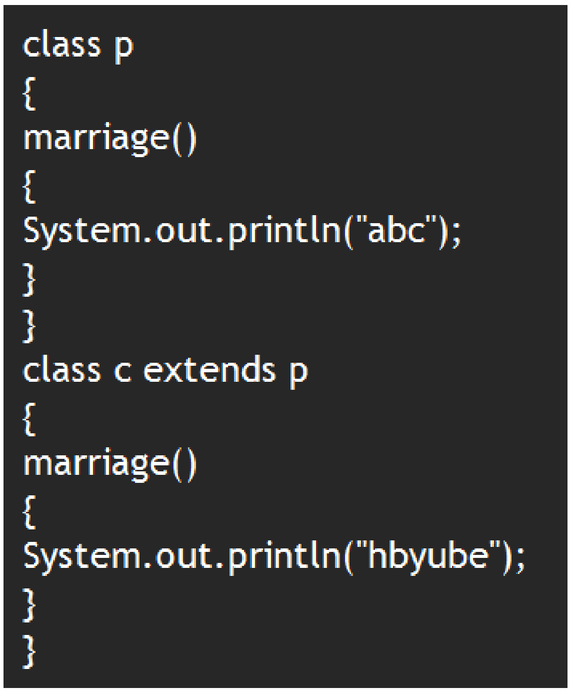
Q. Consider the following method in parent class “public void m1(int x) throws IOException” in the child class which of the following methods we can take?

1. public void m1(int i) -> overriding
2. public static int m1(long l) -> overloading
3. public static void m1(int i) -> overriding but it is invalid non-static to static can’t override
4. public void m1(int i) throws Exception -> invalid overriding concept of throws class
5. public static abstract void m1(double d) -> invalid, static & abstract is illegal combination

POLYMORPHISM:

One name but multiple forms is the concept of polymorphism. Ex1: method name is the same but we can apply for different types of arguments is known as overloading. Ex: abs(int), abs(long), abs(float).

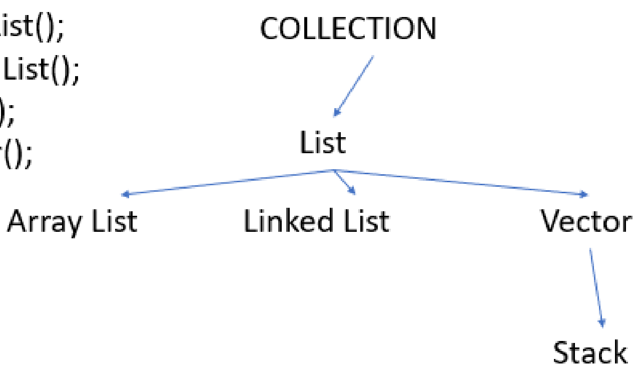
Ex2: method signature is same but in parent class one type of implementation and in child class another type of implementation is known as overriding.



```
class p
{
marriage()
{
System.out.println("abc");
}
}
class c extends p
{
marriage()
{
System.out.println("hbyube");
}
}
```

Ex: usage of parent reference to hold child object is the concept of polymorphism.

```
List l = new ArrayList();
List l = new LinkedList();
List l = new Stack();
List l = new Vector();
```



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

P p = new C();

p.m1(); -> valid

p.m2(); -> invalid can’t find symbol.

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

C c = new c();

c.m1(); -> valid

c.m2(); -> valid

WHEN WE SHOULD GO FOR PARENT REFERENCE TO HOLD CHILD OBJECT?

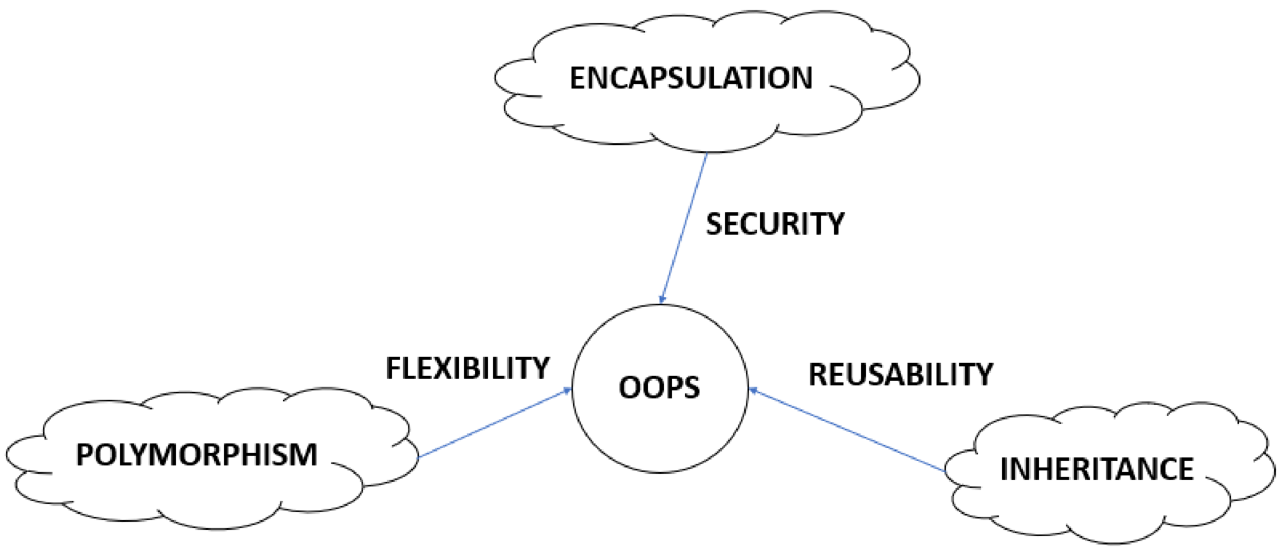
If we don’t now exact runtime type of object then we should go for parent reference. For example the first element present in the array list can be any type it may be student object/ customer object/ string object/stringbuffer object. Hence the return type of get method is object, which can hold any object. “Object O = l.get(0);”

C c = new C()

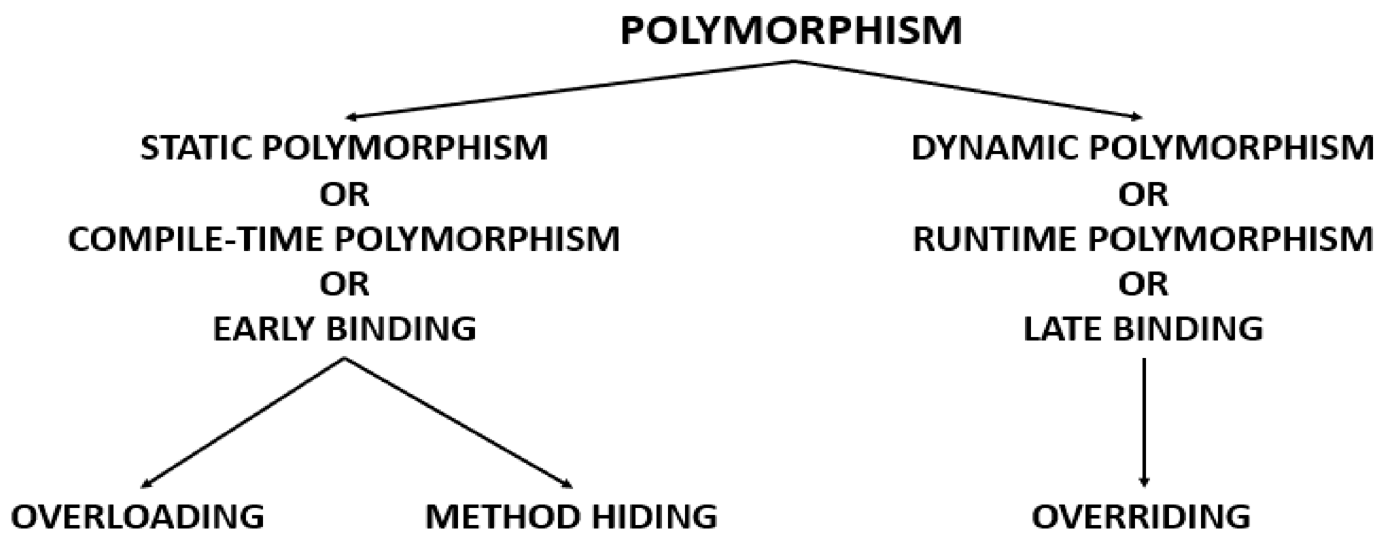
AL l = new AL(); We can use this approach if we know exact runtime type of object. By using child reference we can call both child and parent class methods. This is the advantage of this approach. We can use child reference to hold only particular child class object. This is the disadvantage of this approach.

P p = new C();

List l = new AL(); We can use this approach if we don’t know exact runtime type of object. By using parent reference we can call only methods available in parent class. and we can’t call child specific methods. This is the disadvantage of this approach. We can use parent reference to hold any child class object this is the advantage of this approach.



3 PILLARS OF OBJECT-ORIENTED PROGRAMMING

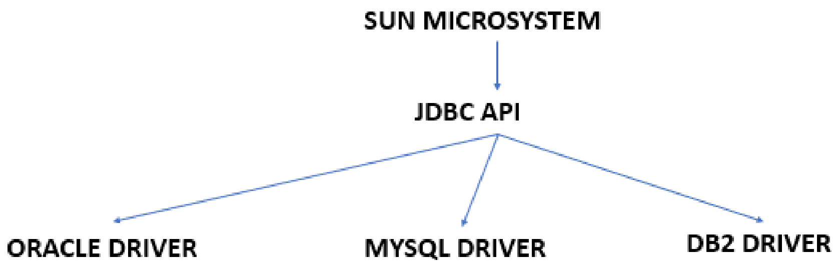


INTERFACE

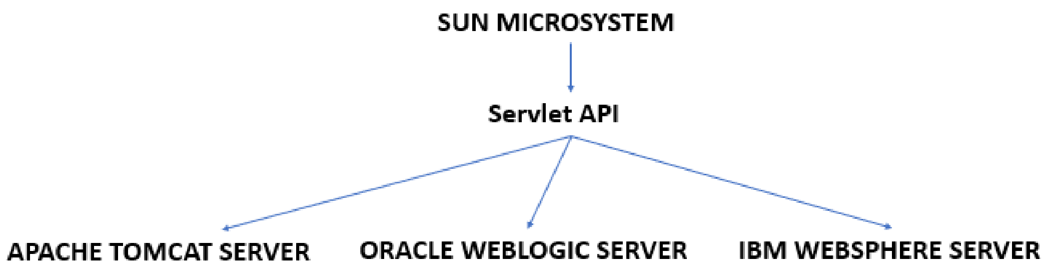
INTERFACE:

1. Any service requirement specification (SRS) is considered as an interface.

Ex: JDBC API access requirement specification to develop database driver. Database vendor is responsible to implement this JDBC API.

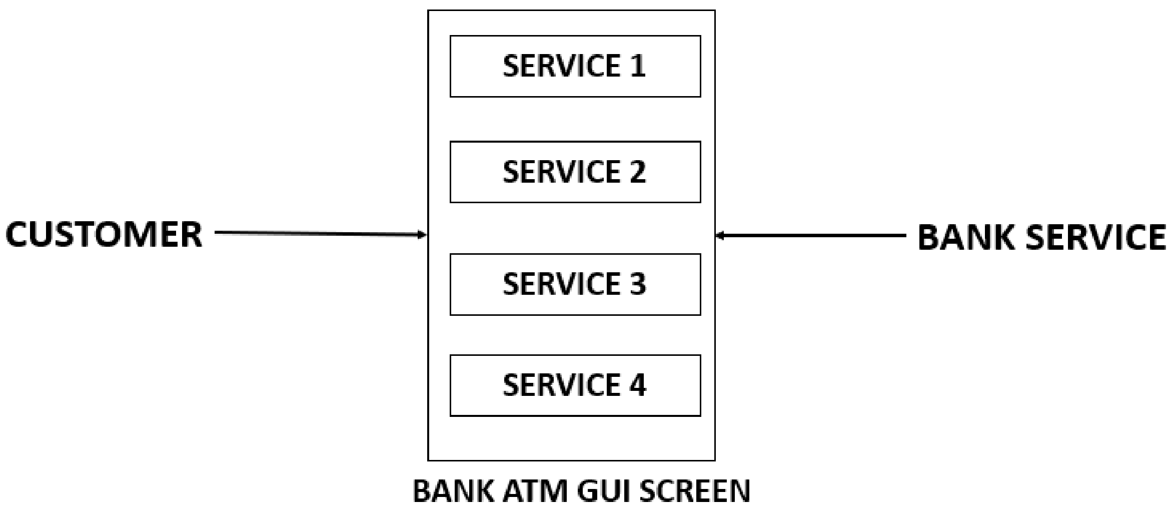


Ex: Servlet API access requirement specification to develop web server. Web server vendor is responsible to implement Servlet API.



2. From client point of view an interface defines the setup services what he is expecting from service provider point of view a interface defines the setup services what he is offerring hence any contract between client and service provider is considered as an interface.

Ex: Through bank ATM GUI screen bank people are highlighting the setup services what they are offering at the same time. The same GUI screen represents the setup services what customer is expecting hence this GUI screen access contract between customer and bank people.

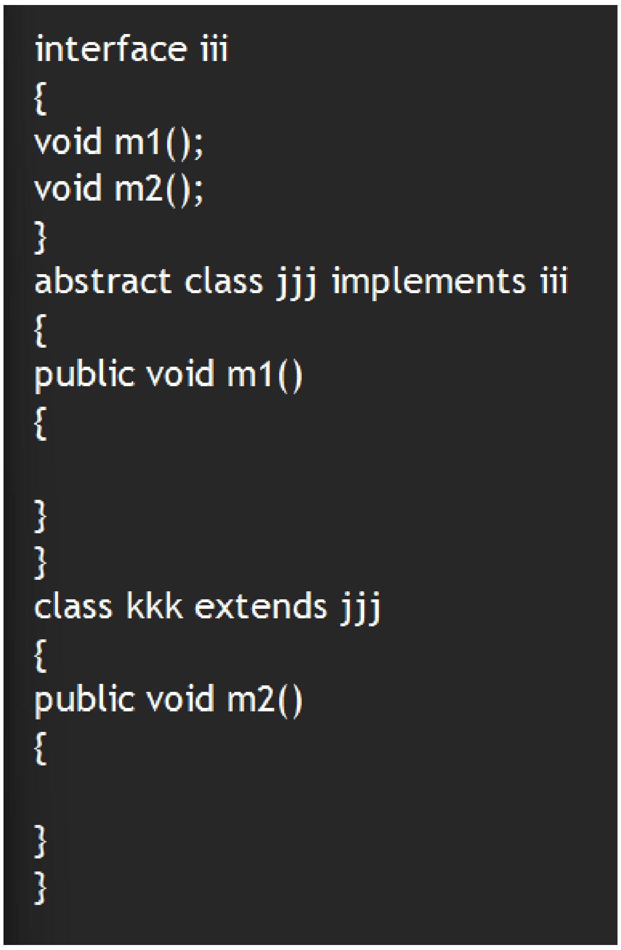


3. Inside interface every method is always abstract whether we are declaring or not. Hence interface is considered as 100% pure abstract class.

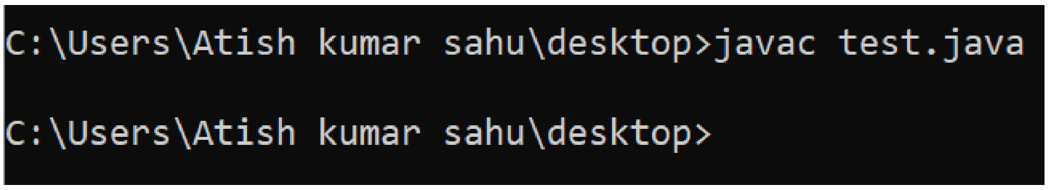
ANY SERVICE REQUIRED SPECIFICATION OR ANY CONTRACT BETWEEN CLIENT AND SERVICE PROVIDER OR 100% PURE ABSTRACT CLASS IS NOTHING BUT INTERFACE.

Whenever we are implementing an interface for each and every method of that interface we have to provide implementation otherwise we have to declare class as abstract then next level child class is responsible to provide implementation.

Every interface method is always public and abstract whether we are declaring or not. Hence whenever we are implementing interface method compulsory we should declare as "public" otherwise we will get compli time error.



```
interface iii
{
void m1();
void m2();
}
abstract class jjj implements iii
{
public void m1()
{

}
}
class kkk extends jjj
{
public void m2()
{

}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>
```

EXTENDS VS IMPLEMENTS:

1. A class can extends only one class at a time.

2. An interface can extends any number of interfaces simultaneously.

```
interface a
{ }
interface b
{ }
interface c extends a, b
{ }
```

3. A class can implements any number of interfaces simultaneously.

4. A class can extend another class and can implement any number of interfaces simultaneously.

```
class a extends b implements c,d,e
{ }
```

Q. WHICH OF THE FOLLOWING STATEMNENTS ARE VALID?

1. a class can extends any number of classes at a time. -> invalid

2. a class can implement only one interface at a time. -> invalid

3. interface can extends only one interface at a time. -> invalid

4. an interface can implements any interfaces simultaneously. -> invalid

5. a class can extend another class or can implement an interface but not both simultaneously. -> invalid

Q. consider the following expression "X extends Y" for which of the following possibilities the above expression is valid?

1. both X and Y should be class

2. both X and Y should interfaces

3. both X and Y can be either class or interfaces -> valid

4. no restriction

Q. X extends Y, Z -> X, Y, Z should be interfaces.

Q. X implements Y, Z -> X should be class Y,Z should be interfaces.

Q. X extends Y implements Z -> X, Y should be class and Z should be interface.

Q. X implements Y extends Z -> invalid statement extends first and then implements.

INTERFACE METHODS:

Every method present inside interface is always public and abstract whether we are declaraing or not.

```
interface a
{
public abstract void m1();
}
```

To make the method available to every implementation class we should declare the method as public.

Implementation class is responsible to provide implementation so we should declare the method as abstract which is inside in interface. Hence inside interface the following method declarations are equal.

"void m1();"

"public void m1();"

"abstract void m1();"

"public abstract void m1();"

THESE METHODS ARE VALID METHODS IN INTERFACE.

As every interface method is always public and abstract we can't declare interface method with the following modifiers. PRIVATE, PROTECTED, STATIC, FINAL, SYNCHRONIZED, STRICTFP, NATIVE.

Q. WHICH OF THE FOLLOWING METHOD DECLARATION ARE ALLOWED INSIDE INTERFACE?

public void m1() { } -> INVALID

private void m1(); -> INVALID

protected void m1(); -> INVALID

static void m1(); -> INVALID

public abstract native void m1(); -> INVALID

abstract public void m1(); -> VALID

public abstract void m1(); -> VALID

INTERFACE VARIABLES:

An interface can contain variables. The main purpose of interface variable is to define requirement level constants.

Every variable in interface is public static final whether we declaring or not.

To make a variable available to every implementation class the interface variable should be public.

Without exicting object also implementation class has to access the variable which is inside interface. Hence it is refer to make the variable as a static variable.

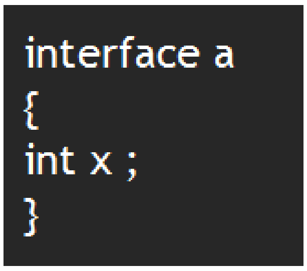
The variable in interface may be used by multiple implementation class and there is a chance that any class trying to change the value of variable. If one implementation class changes value then remaining implementation class will be affected to restrict this every interfcae variable is always final.

Hence with in the interface the following variable declarations inside interface are valid and equal.

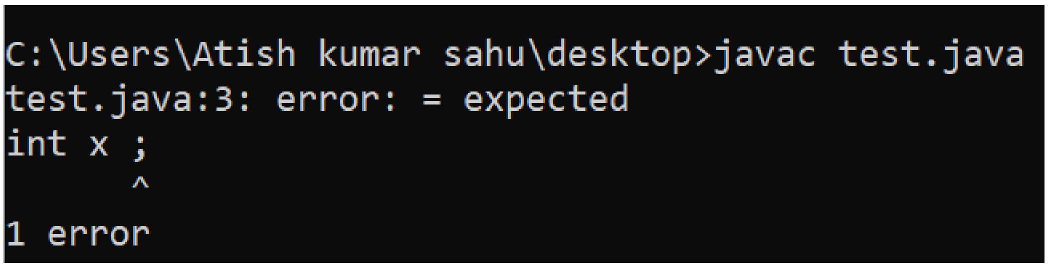
```
interface a
{
public int A = 10;

static int B = 25;

final int C = 39;

public static int D = 55;

public final int E = 47;

static final int F = 66;

public static final int H = 150;

int G = 100;
}
```

As every interface variable is always public static final we can't declare with the following modifiers "PRIVATE", "PROTECTED", "TRANSIENT", "VOLATILE".

For interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.



```
interface a
{
int x ;
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:3: error: = expected
int x ;
     ^
1 error
```

Q. INSIDE INTERFACE WHICH OF THE FOLLOWING VARIABLE DECLARATIONS ARE ALLOWED?

int x; -> invalid

private int x = 140;   -> invalid

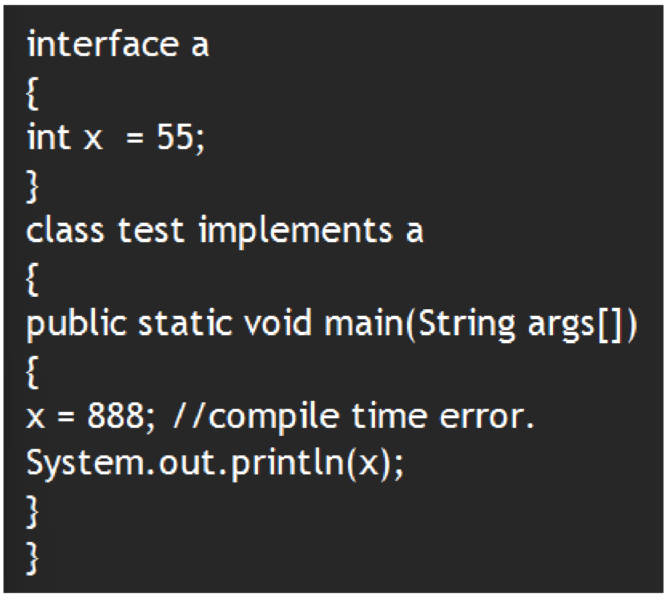
protected int a = 185;   -> invalid

volatile int x = 555;   -> invalid

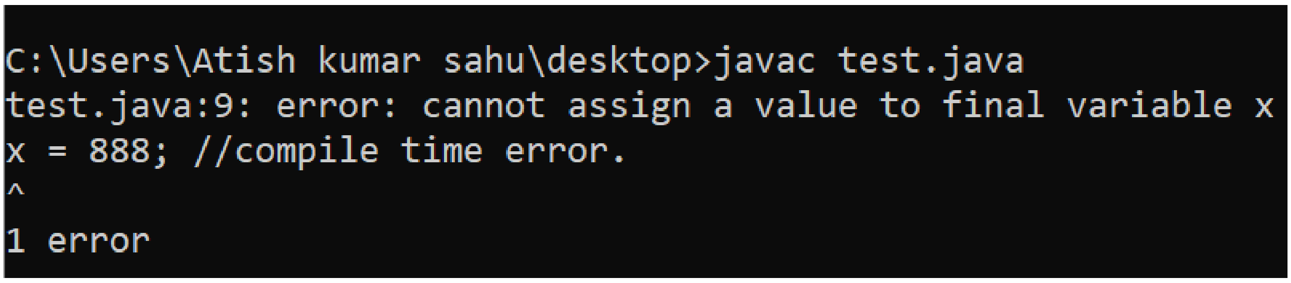
transient int d = 858   -> invalid

public static int x = 5458;   -> valid

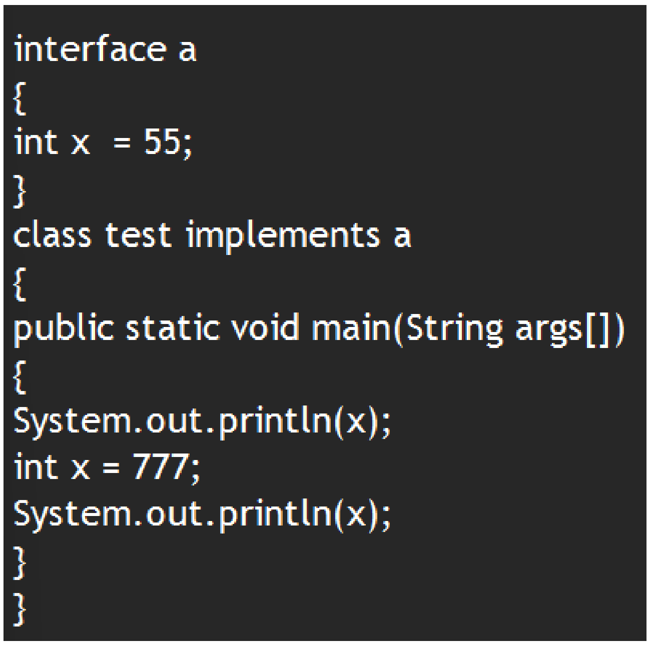
inside implementation class we can access interface varibales but we can't modify values.



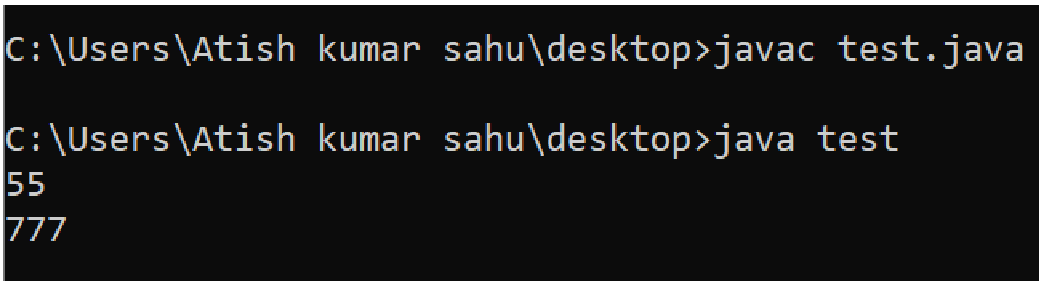
```
interface a
{
int x  = 55;
}
class test implements a
{
public static void main(String args[])
{
x = 888; //compile time error.
System.out.println(x);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:9: error: cannot assign a value to final variable x
x = 888; //compile time error.
^
1 error
```



```
interface a
{
int x  = 55;
}
class test implements a
{
public static void main(String args[])
{
System.out.println(x);
int x = 777;
System.out.println(x);
}
}
```



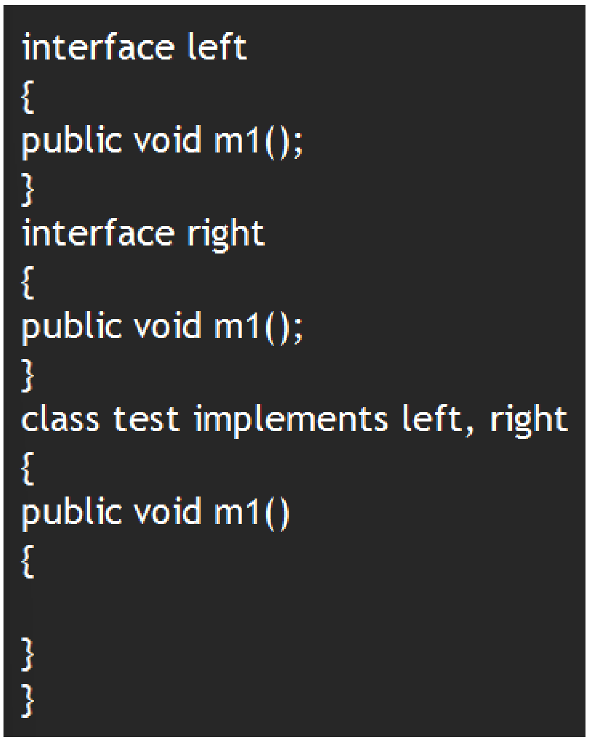
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
55
777
```

<u>INTERFACE NAMING CONFLICTS:</u>

<u>METHOD NAMNG CONFLICTS:</u>

<u>CASE 01:</u>

If two interfaces contains a method with same signature same return type, then in the implementation class we have to provide implementation for only one method.



```
interface left
{
public void m1();
}
interface right
{
public void m1();
}
class test implements left, right
{
public void m1()
{

}
}
```

CASE 02:

If two interfaces contains a method with same name but different argument types then in the implementation class we have to provide implementation for both methods and these method access overloaded methods.

```
interface left
{
public void m1();
}
interface right
{
public void m1(int i);
}
class test implements left, right
{
public void m1()
{

}
public void m1(int i)
{

}
}
```

CASE 03:

If two interfaces contains a method with same signature but different return types then it is impossible to implements both interfaces simultaneoulsy. If return types are not covarient. We can’t write any java class which implements simultaneously.

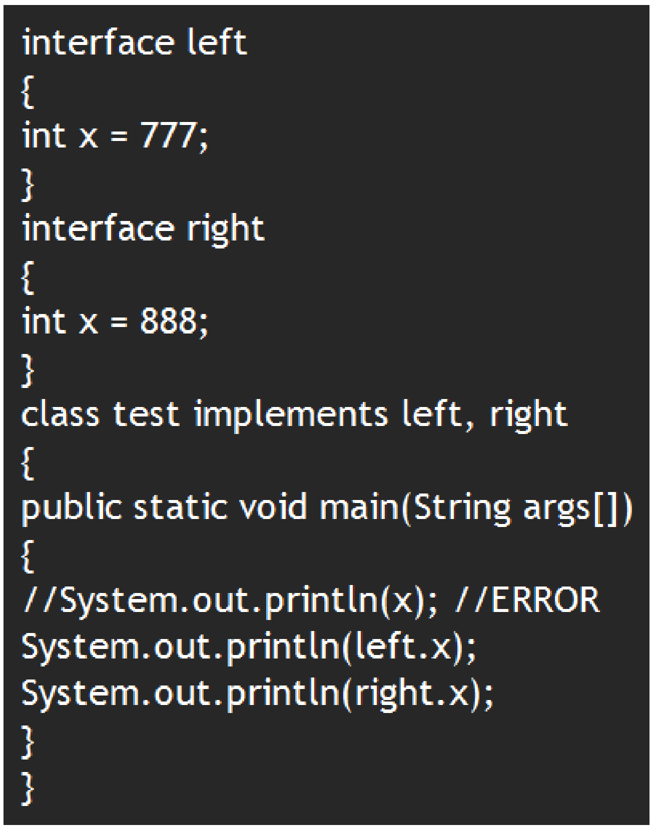
```
interface left
{
public void m1();
}
interface right
{
public int m1();
}
```

Q. IS A JAVA CLASS CAN IMPLEMENT ANY NUMBER OF INTERFACES SIMULTANEOUSLY?

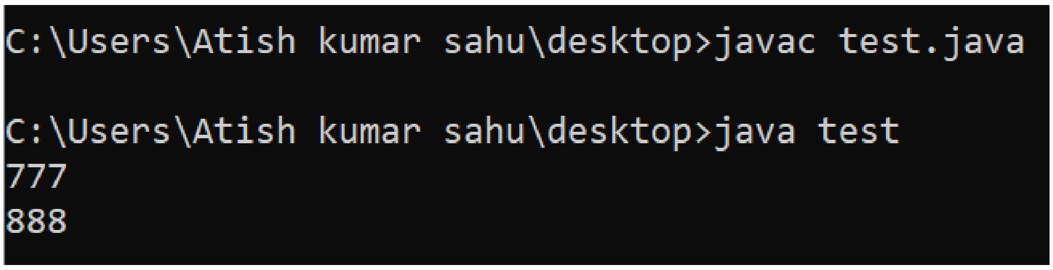
ANS: Yes, except a particular case. If two interfaces contains a method with same signature but different return types then it is immposible to implements both interface simultaneously.

INTERFACE VARIABLE NAMING CONFLICTS:

Two interfaces can contain a variable with same name and there may be a chance of variable naming conflicts but we can solve this problem by using interfaces names.



```
interface left
{
int x = 777;
}
interface right
{
int x = 888;
}
class test implements left, right
{
public static void main(String args[])
{
//System.out.println(x); //ERROR
System.out.println(left.x);
System.out.println(right.x);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
777
888
```

MARKER INTERFACE:

If an interface doesn't contain any methods and by implementing that interface if our object will get some ability such type of interfaces are called marker interfaces or "ability interface" or "Tag interface". Ex: Serializable, Clonable, RandomAccess, SingleThreadModel. These are marked for some ability.

Ex1: by implementing serializable interface our objects can be saved to the file and can travel across the network.

Ex2: by implementing clonable interface our objects are in a position to produce exactly duplicate cloned objects.

Q. WITHOUT HAVING ANY METHODS HOW THE OBJECTS WILL GET SOME ABILITY IN MARKER INTERFACES?

ANS: internally JVM is responsible to provide required ability.

Q. WHY JVM IS PROVIDING REQUIRED ABILITY IN MARKER INTERFACES?

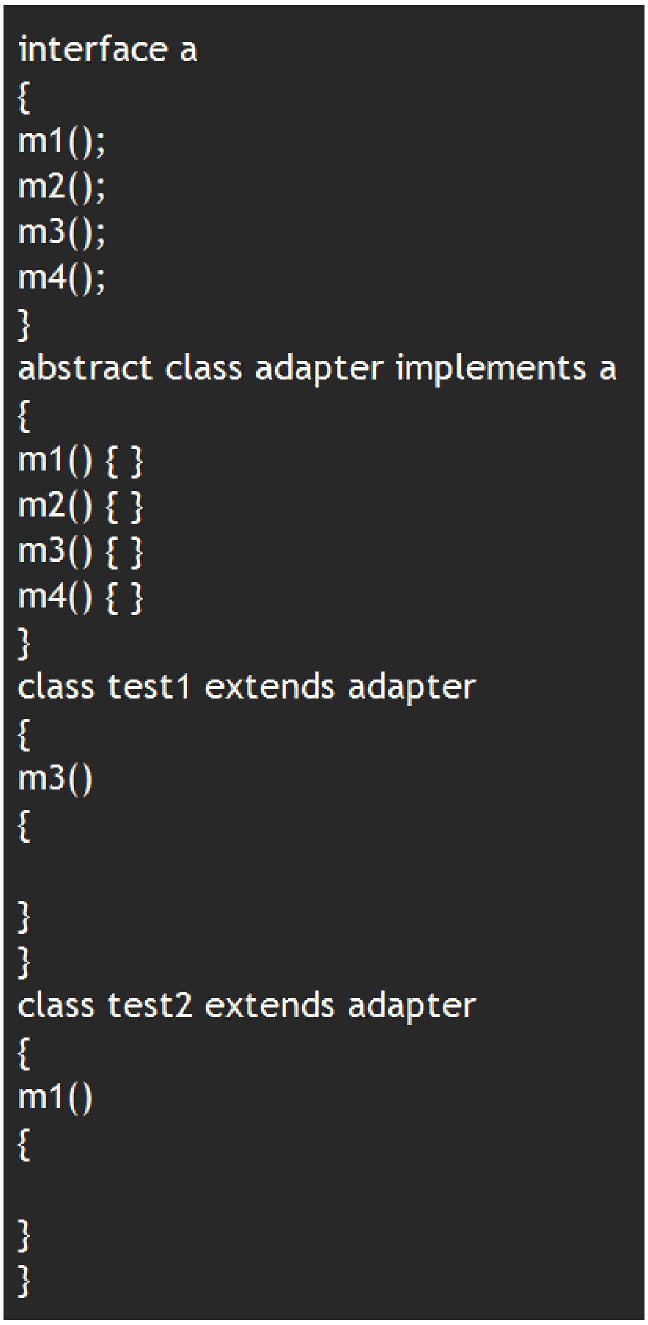
ANS: to reduce complexity of programming and to make JAVA language as simple.

Q. IS IT POSSIBLE TO CREATE OUR OWN MARKER INTERFACE?

ANS: yes, but cuustomization of JVM is required.

ADAPTOR CLASS:

Adaptor class is a simple java class that implements an interface with only empty implementation. If we implement an interface for each and every method of interface compulsory we should provide implementation. Whether it is required or not required. The problem in this approach is it increases length of the code and reduces redability. We can solve this problem by using adaptor classes. Instead of implementing interface if we extends adaptor class we have to provide implementation only for required methods and we are not responsible to provide implementation for each and every method of the interface. So that length of the code will be reduced.



```
interface a
{
m1();
m2();
m3();
m4();
}
abstract class adapter implements a
{
m1() { }
m2() { }
m3() { }
m4() { }
}
class test1 extends adapter
{
m3()
{

}
}
class test2 extends adapter
{
m1()
{

}
}
```

We can develop a servlet in the following 3 ways. By implementing servlet interface, by extending genric servlet, by extending http servlet. If we implement servlet interface for each and every method of that interface we should provide implementation. It increases length of the code and reduces redability.

Instead of implementing servlet interface directly if we extend generic servlet we have to provide implementation only for service method. And all remaining methods we are not required to provide implementation. Hence more or less generic servlet access adaptor class for servlet interface.

NOTE:

MARKER INTERFACE AND ADAPTOR CLASSES SIMPLIFIES COMPLEXITY OF PROGRAMMING AND THESE ARE BEST UTILITIES TO PROGRAMMER AND PROGRAMMER LIFE WILL BECOME SIMPLE.

INTERFACE VS ABSTRACT CLASS VS CONCRETE CLASS:

1. if we don't know anything about implementation just we have requirement specification then we should go for interface. Ex: servlet

2. if we are talking about implementation but not completely(partial implementation) then we should go for abstract class. Ex: genric servlet, http servlet

3. if we are talking about implementation completely and ready to provide service then we should go for concrete class. Ex: my own servlet.

$$\begin{tikzcd}
\text{Interface} \arrow[r] & \text{servlet} \arrow[r] & \text{plan} \\
\text{Abstract class} \arrow[u] \arrow[r] & \text{genric and http servlet} \arrow[u] \arrow[r] & \text{partial completion} \arrow[u] \\
\text{Concrete class} \arrow[u] \arrow[r] & \text{my own servlet} \arrow[u] \arrow[r] & \text{full completion} \arrow[u]
\end{tikzcd}$$

DIFFERENCES BETWEEN INTERFACE AND ABSTRACT CLASS:

If we don't know anything about implementation and just we have requirement specification then we should go for interface. If we are talking about implementation but not completely(partial implementation) then we should go for abstract class.

Inside interface every method is always public and abstract whether we are declaring or not hence interface is considered as 100% pure abstract class. every method present inside abstract class need not be public and abstract and we can take concrete method also.

As every interface method is always public and abstract and hence we can't declare with the following modifiers PRIVATE, PROTECTED, FINAL, STATIC, SYNCHRONIZED, NATIVE, STRICTFP. There are no restriction on abstract class method modifiers.

Every variable present inside interface is always public static final whether we are decclaring or not. Every variable present inside abstract class need not be public static final.

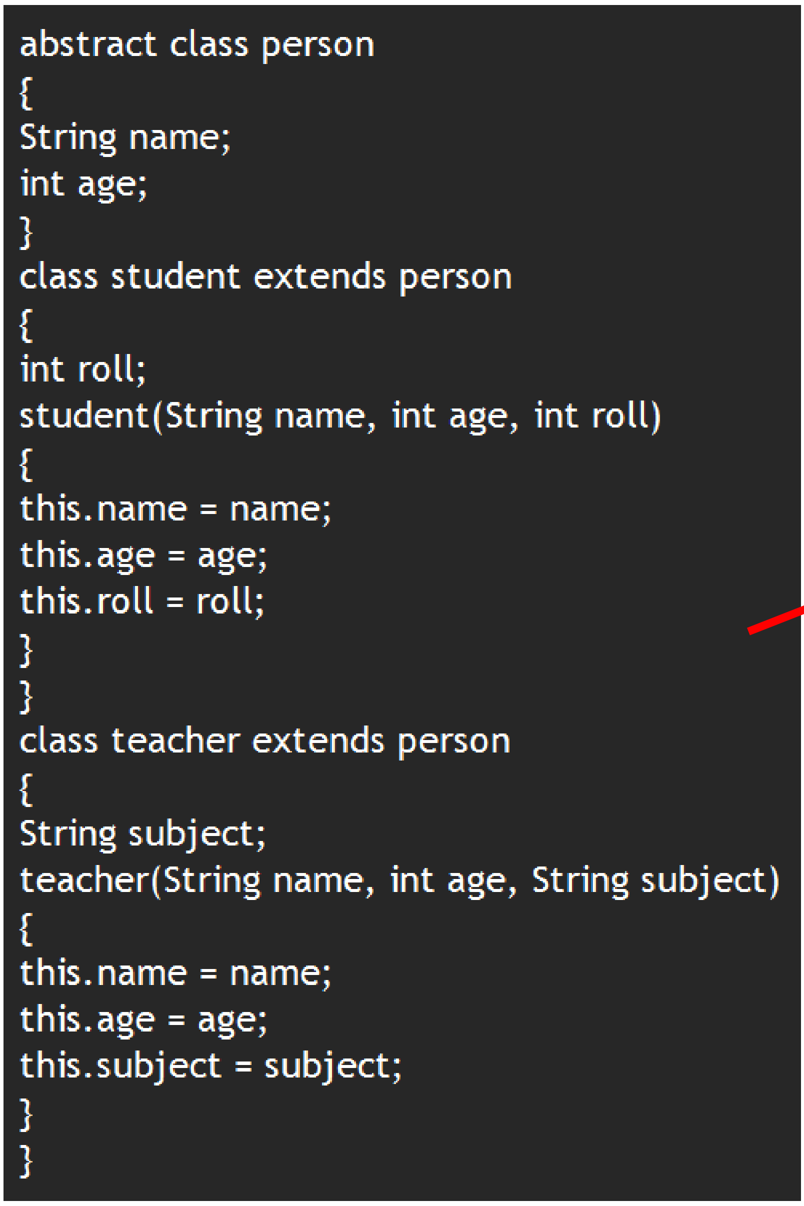
As every interface variable is always public static final we can't declare with the following modifiers PRIVATE, PROTECTED, VOLATILE, TRANSIENT. There are no restriction on abstract class variable modifiers.

For interface variables compulsory we should perform initialization at the time of declaration only otherwise we will get compile time error. for abstract class variables we are not required to perform initialization at the time of declaration.

Inside interface we can't declare static and instance blocks. Inside abstract class we can declare static and instance blocks. Inside interface we can't declare constructors. Inside abstract class we can declare constructor.

Q. ANYWHERE WE CAN'T CREATE OBJECT FOR ABSTRACT CLASS, BUT ABSTRACT CLASS CAN CONTAIN CONSTRUCTOR. WHAT IS THE NEED?

ANS: abstract class constructor will be executed whenever we are creating child class object to perform initialization of child class object.

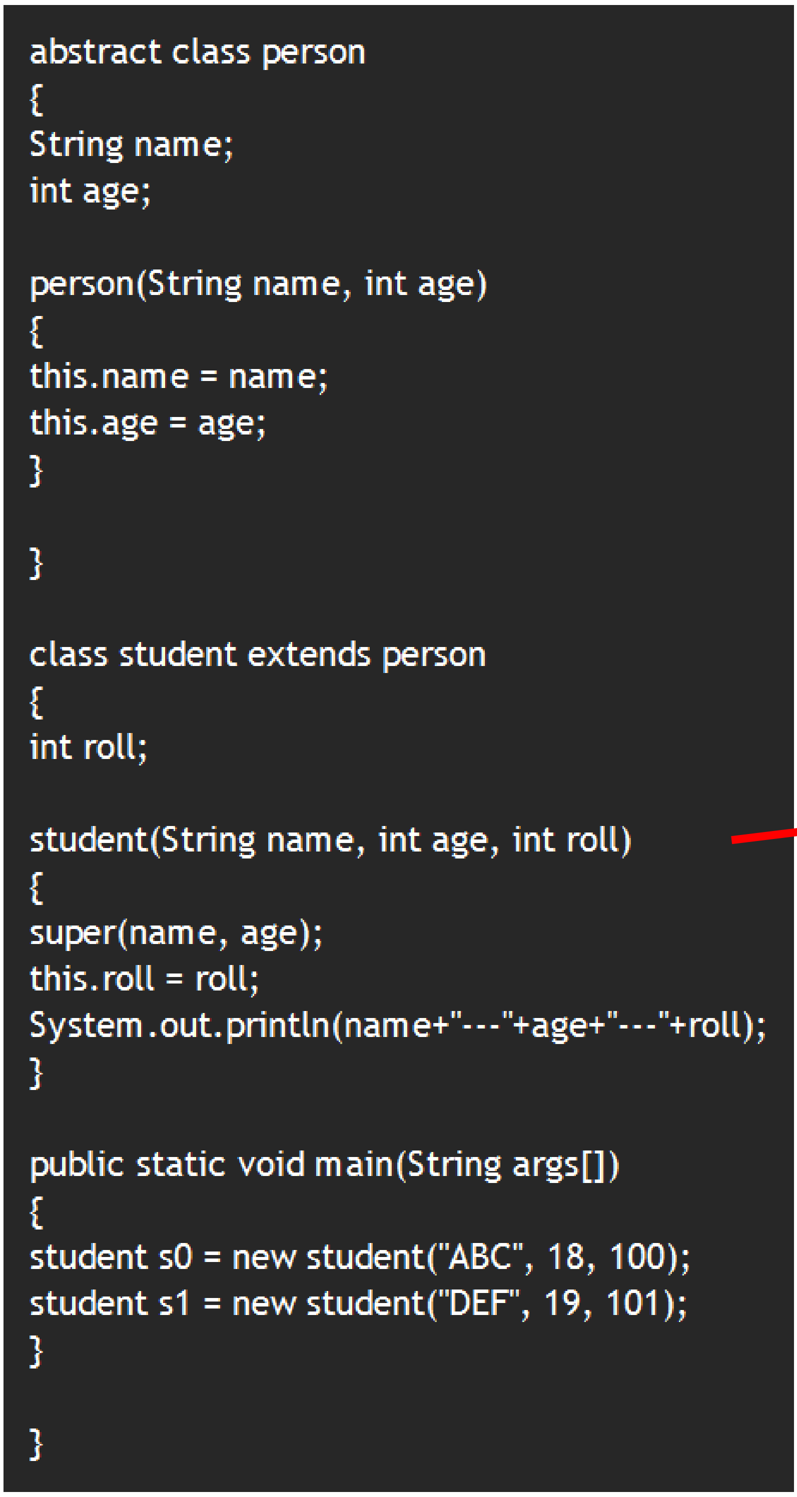


```
abstract class person
{
String name;
int age;
}
class student extends person
{
int roll;
student(String name, int age, int roll)
{
this.name = name;
this.age = age;
this.roll = roll;
}
}
class teacher extends person
{
String subject;
teacher(String name, int age, String subject)
{
this.name = name;
this.age = age;
this.subject = subject;
}
}
```

WORST KIND OF PROGRAMMING

WITHOUT HAVING CONSTRUCTOR IN ABSTRACT CLASS.

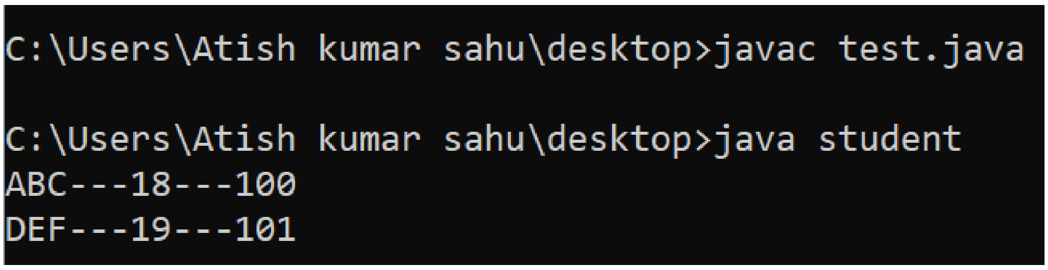
CODE IS MORE AND REDUNDANCY PROBLEM IS THERE.



```
abstract class person
{
String name;
int age;

person(String name, int age)
{
this.name = name;
this.age = age;
}

}

class student extends person
{
int roll;

student(String name, int age, int roll)
{
super(name, age);
this.roll = roll;
System.out.println(name+"---"+age+"---"+roll);
}

public static void main(String args[])
{
student s0 = new student("ABC", 18, 100);
student s1 = new student("DEF", 19, 101);
}

}
```

BEST KIND OF PROGRAMMING

EITHER DIRECTLY OR INDIRECTLY WE CAN'T CREATE OBJECT FOR ABSTRACT CLASS.



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java student
ABC---18---100
DEF---19---101
```

WITH CONSTRUCTOR INSIDE ABSTRACT CLASS.

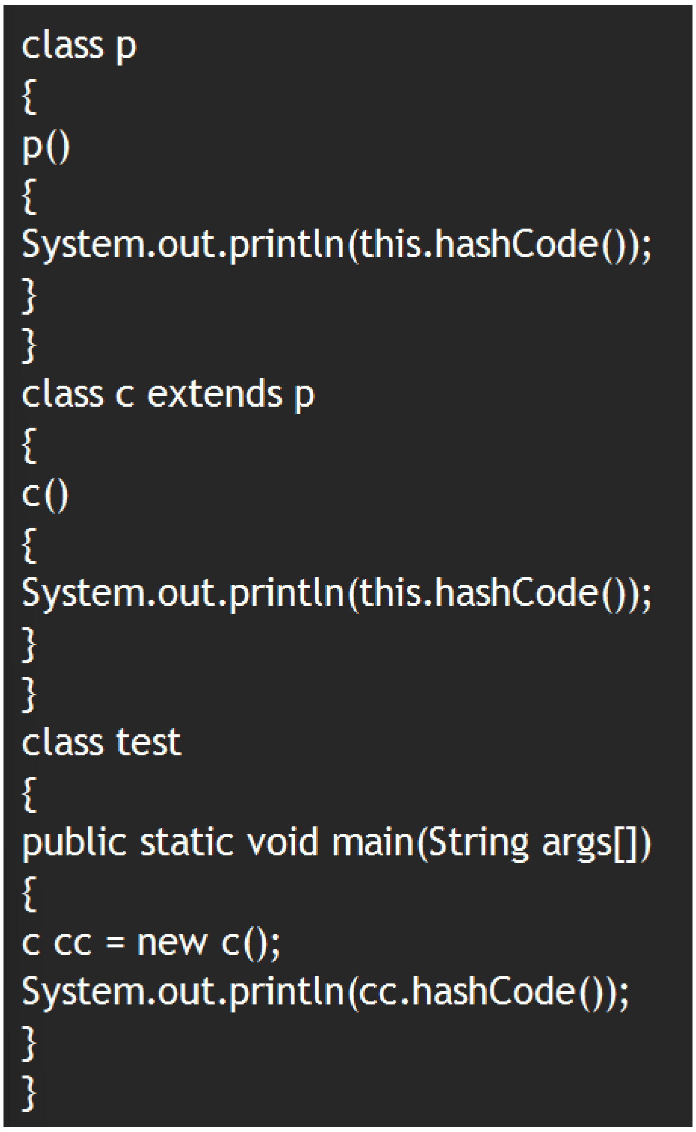
CODE RESUABILITY AND LESS CODE ADVANTAGES.

Q. ANYWHERE WE CAN'T CREATE OBJECTS FOR ABSTRACT CLASS AND INTERFACE BUT ABSTRACT CLASS CAN CONTAIN CONSTRUCTOR BUT INTERFACE DOESN'T CONTAIN CONSTRUCTOR. WHAT IS THE REASON?

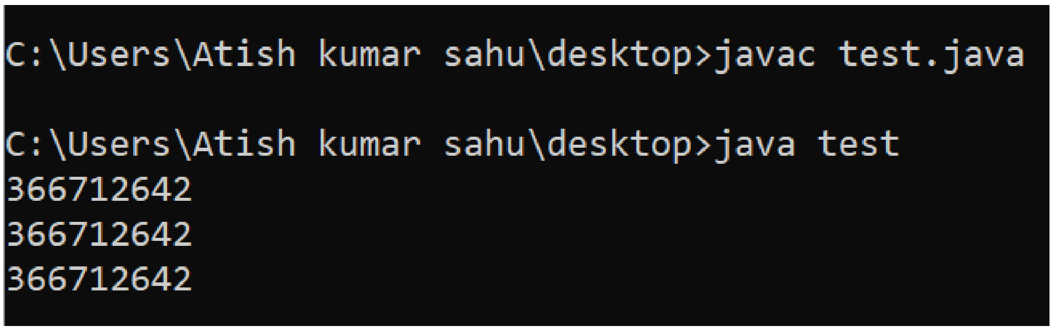
ANS: the main purpose of constructor is to perform intialization of instance variables. abstract class can contain instance variables which are required for child object to perform initialization of those instance variables constructor is required for abstract class.

But every variable present inside interface is always public static final whether we are declaring or not, and there is no chance of existing instance variable inside interface hence constructor concept is not required for interface.

Whenever we are creating child class object parent object won't be created just parent class constructor will be executed for the child object purpose only.



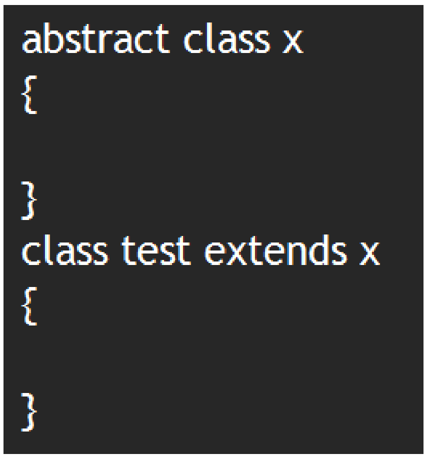
```
class p
{
p()
{
System.out.println(this.hashCode());
}
}
class c extends p
{
c()
{
System.out.println(this.hashCode());
}
}
class test
{
public static void main(String args[])
{
c cc = new c();
System.out.println(cc.hashCode());
}
}
```



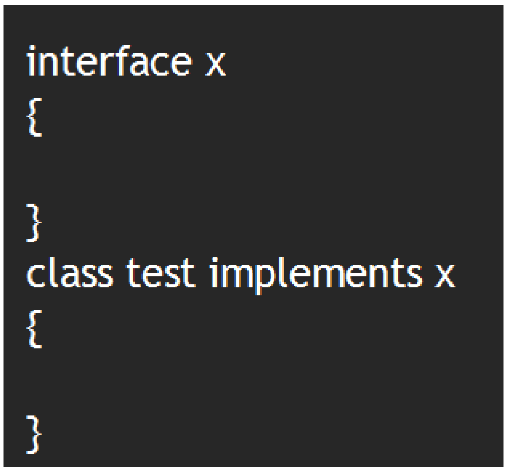
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
366712642
366712642
366712642
```

Q. INSIDE INTERFACE EVERY METHOD IS ALWAYS ABSTRACT AND WE CAN TAKE ONLY ABSTRACT METHODS IN ABSTRACT CLASS ALSO THEN WHAT IS THE DIFFERENCE BETWEEN INTERFACE AND ABSTRACT CLASS. IS IT POSSIBE TO REPLACE INTERFACE WITH ABSTRACT CLASS?

ANS: we can replace interface with abstract class, but it is not a good programming practice. This is something like recruiting IAS officer for sweeping activity. If everything is abstract then it is highly recommended to go for interface but not for abstract class.



```
abstract class x
{

}
class test extends x
{

}
```



```
interface x
{

}
class test implements x
{

}
```

While extending abstract class it is not possible to extend any other class and hence we are missing inheritance benefit. While implementing interface we can extends some other class and hence we won't miss any inheritance benefit.

In abstract case the object creation is costly example "test t = new test(); -> 2 min".

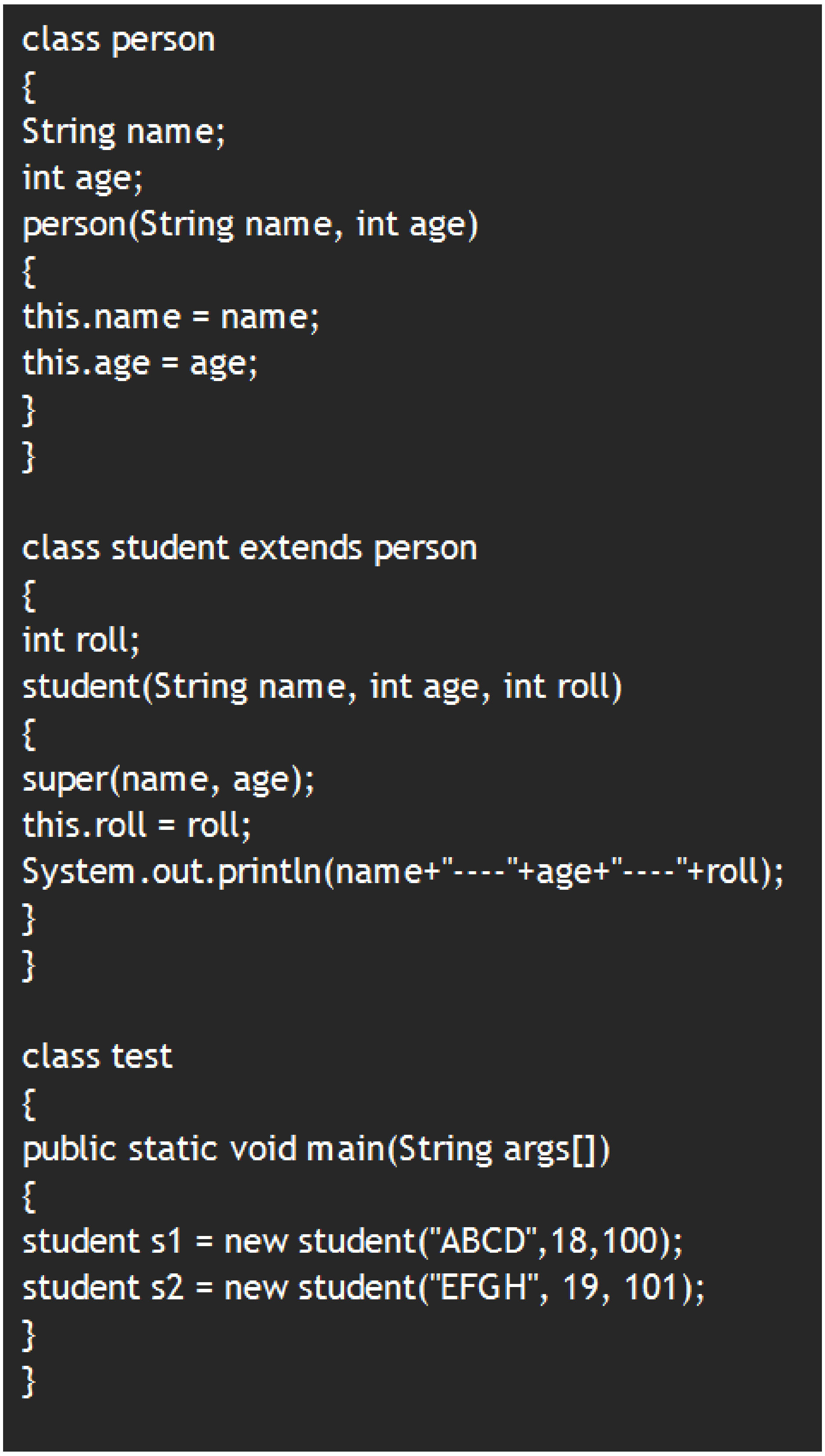
In interface case the object creation is not costly example: "test t = new test(); -> 2 sec".

Q. WHENEVER WE ARE CREATING CHILD OBJECT WHETHER PARENT OBJECT WILL BE CREATED OR NOT?

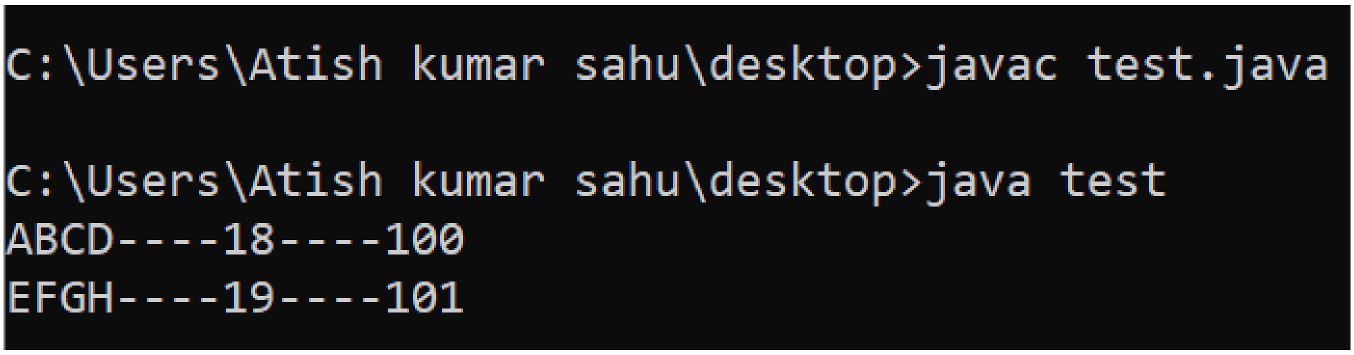
ANS: NO

Q. WHENEVER WE ARE CREATING CHILD CLASS OBJECT WHAT IS THE NEED OF EXECUTING PARENT CLASS CONSTRUCTOR?

ANS: Whenever we are creating child class object automatically parent constructor will be executed to perform initialization for the instance variables which are inheriting from parent.



```
class person
{
String name;
int age;
person(String name, int age)
{
this.name = name;
this.age = age;
}
}

class student extends person
{
int roll;
student(String name, int age, int roll)
{
super(name, age);
this.roll = roll;
System.out.println(name+"----"+age+"----"+roll);
}
}

class test
{
public static void main(String args[])
{
student s1 = new student("ABCD",18,100);
student s2 = new student("EFGH", 19, 101);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
ABCD----18----100
EFGH----19----101
```

<u>COUPLING & COHESION:</u>

<u>COUPLING:</u>

The degree of the dependency between the component is called “COUPLING”. If dependency is more then it is considered as “TIGHTLY COUPLING” and if dependency is less then it is considered as “LOOSELY COUPLING”.

```
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 100;
}
```

The above components are said to be tightly coupled with each other because dependency between the component is more. Tightly coupling is not a good programming practice because it has several disadvantages

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

2. it reduces reusability.

3. it reduces maintainability of the application.

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

COHESION:

For every component a clear well defined functionality is defined then that component is said to be follow high cohesion. High cohesion is always a good programming practice because it has several advantages.

1. without affecting remaining component we can modify any component hence enhancement will become easy.

2. it promotes reusability of code(whereever validation is required we can reuse the same validate servlet without rewriting).

3. It improves maintainablity of the application.

NOTE:

LOOSELY COUPLING AND HIGH COHESION ARE GOOD PROGRAMMING PRACTICES.

OBJECT TYPE-CASTING:

We can use parent reference to hold child object. Ex: “Object O = new String (“ATISH”);”. We can use interface reference to hold implemented class object. Example: “Runnable r = new Thread();”.

Object o = new String(“AAAA”);

StringBuffer sb = (StringBuffer)o;

A b = (C) d;

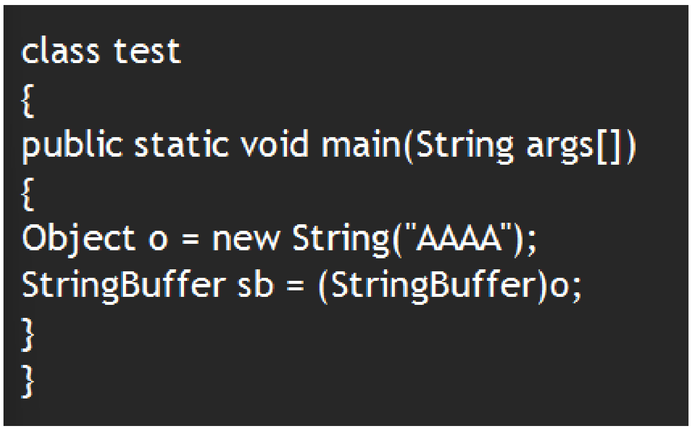
A = CLASS/INTERFACE NAME	b = name of reference variable
C = CLASS/INTERFACE NAME	d = name of reference variable

*** COMPILE TIME CHECKING 01:

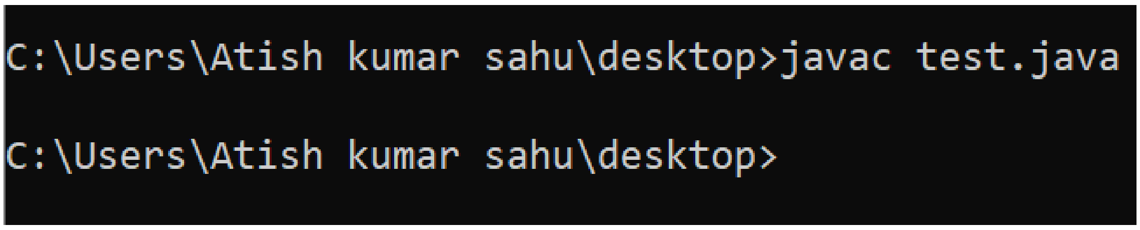
The type of “d” and “C” must have some relation either child to parent or parent to child or same type. Otherwise we will get compile time error saying “inconvertable type found : d type required : C.”

Object O = new String(“AAAA”);

StringBuffer sb = (StringBuffer)O;



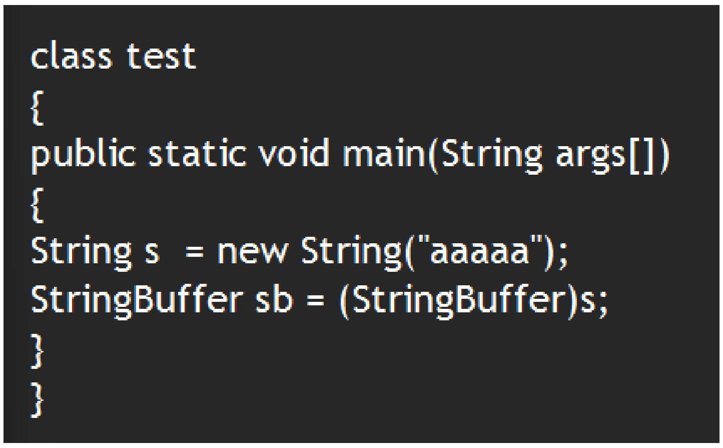
```
class test
{
public static void main(String args[])
{
Object o = new String("AAAA");
StringBuffer sb = (StringBuffer)o;
}
}
```



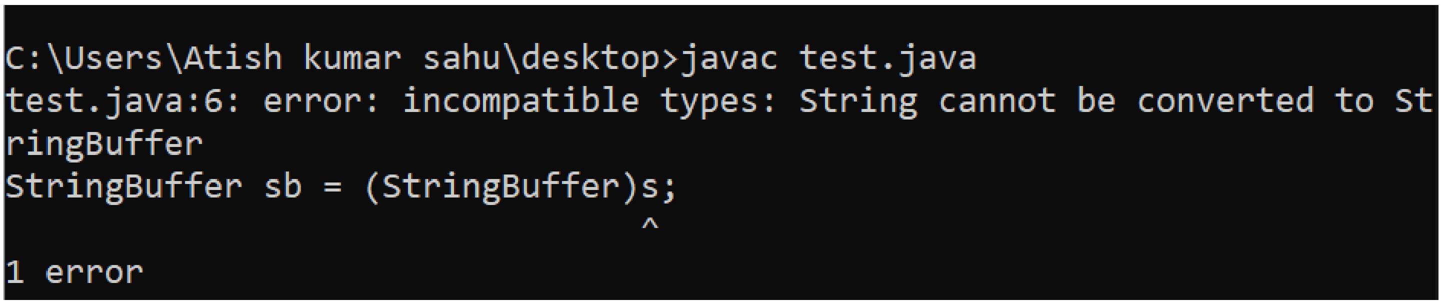
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>
```

String s = new String(“aaaa”);

StringBuffer sb = (StringBuffer)s;



```
class test
{
public static void main(String args[])
{
String s  = new String("aaaaa");
StringBuffer sb = (StringBuffer)s;
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:6: error: incompatible types: String cannot be converted to St
ringBuffer
StringBuffer sb = (StringBuffer)s;
                                ^
1 error
```

*** COMPILE TIME CHECKING 2:

A b = (C) d; -> "C" must be either same or derieved type of "A" otherwise we will get compile time error saying incompatible types found: C required: A

Example1:

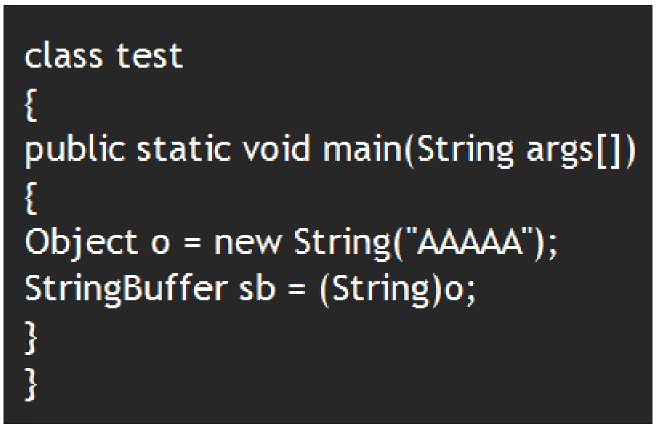
Object O = new String("aaaaa");

StringBuffer SB = (StringBuffer)O;

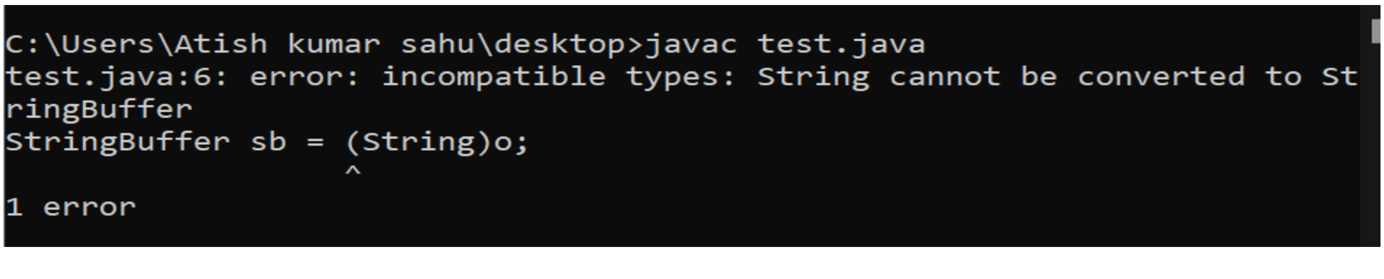
Example2:

Object o = new String("aaaa");

StringBuffer sb = (String)o;



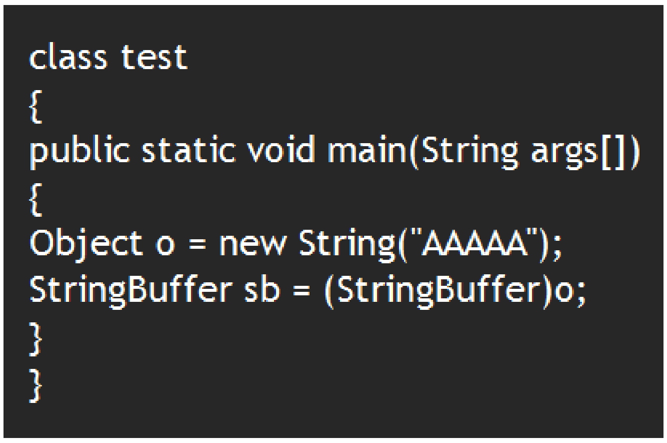
```
class test
{
public static void main(String args[])
{
Object o = new String("AAAAA");
StringBuffer sb = (String)o;
}
}
```



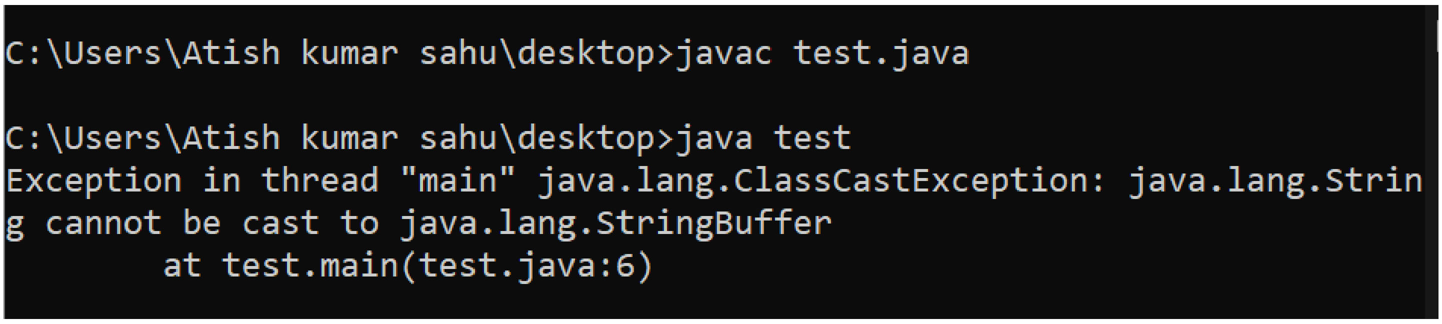
```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:6: error: incompatible types: String cannot be converted to St
ringBuffer
StringBuffer sb = (String)o;
                  ^
1 error
```

*** RUNTIME CHECKING:

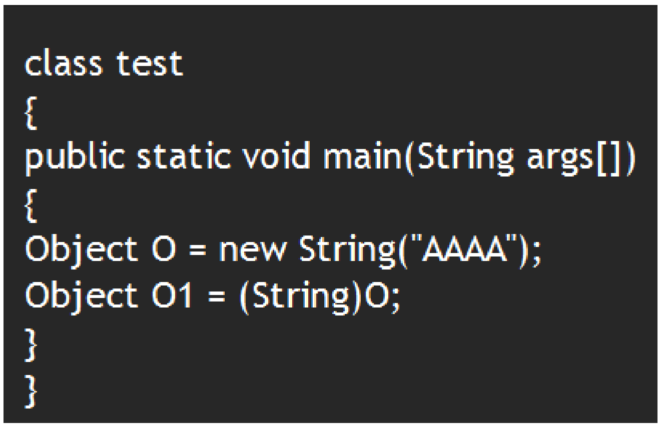
Runtime object type of “d” must be either same or derived type of “C” otherwise we will get runtime exception saying “ClassCastException”.



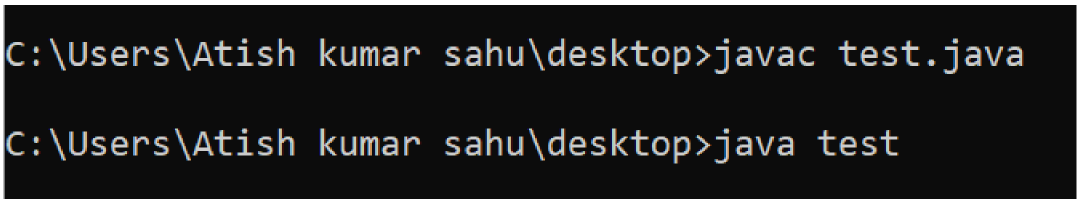
```
class test
{
public static void main(String args[])
{
Object o = new String("AAAAA");
StringBuffer sb = (StringBuffer)o;
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
Exception in thread "main" java.lang.ClassCastException: java.lang.Strin
g cannot be cast to java.lang.StringBuffer
        at test.main(test.java:6)
```



```
class test
{
public static void main(String args[])
{
Object O = new String("AAAA");
Object O1 = (String)O;
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
```

$$\begin{tikzcd}[column sep=small]
& & & \text{OBJECT} \arrow[dll] \arrow[drr] & & & \\
& \text{BASE 1} \arrow[dl] \arrow[dr] & & & & \text{BASE 2} \arrow[dl] \arrow[dr] & \\
\text{DERVED 1} & & \text{DERVED 2} & & \text{DERVED 3} & & \text{DERVED 4}
\end{tikzcd}$$

Base2 b = new Derived4();

1. Object o = (Base2)b; -> valid
2. Object o = (Base1)b; -> invalid -> inconvertable types found “Base2” required “Base1”.
3. Object o = (Derived3)b; -> invalid -> class cast exception
4. Base b1 = (Base1)b; -> invalid -> inconvertable types found “Base2” required “Base1”.
5. Base1 b1 = (Derived4)b; -> invalid -> incompatable types found “Derived4” required “B1”
6. Base b1 = (Derived1)b; -> invalid -> inconvertable types found “Base2” required “Der1”.

Strictly speaking through type casting we are not creating any new object. For the existing object we are providing another types of reference variable. That is we are performing type casting but not object casting.

String S = new String(“aaaa”);
Object O = (Object)S;

$\rightarrow$ $\boxed{\text{Object O = new String(“aaaa”);}}$

Integer I = new Integer(111);
Number n = (Number)I;
Object o = (Object)n;

$\rightarrow$ $\boxed{\text{Number n = new Integer(111);}}$

$\rightarrow$ $\boxed{\text{Object o = new Integer(111);}}$

System.out.println(i == n);
System.out.println(n ==o);

C c = new C();
(B)c -> B b = new C();
(A)(B)c = A a = new C();

$$\boxed{\begin{tikzcd}
\text{A} \\
\text{B} \arrow[u] \\
\text{C} \arrow[u]
\end{tikzcd}}$$

EXAMPLE 1:

C c = new C(); -> WHICH METHOD CALLS ARE VALID AND WHICH ARE INVALID?

```
P -> m1(){ }

C -> M2(){ }
```

c.m1(); -> valid

c.m2(); -> valid

((p)c).m1(); -> P p = new C(); -> p.m1(); -> valid

((p)c).m2(); -> P p = new C(); -> p.m2(); -> invalid

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

EXAMPLE 2:

```
A -> m1(){ SOP(“A”);}

B -> M1(){ SOP(“B”);}

C -> M1(){ SOP(“C”);}
```

C c = new C();

c.m1(); -> C

((B)c).m1(); -> C

((A)((B)c)).m1(); -> C

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

EXAMPLE 3:

```
A -> static m1(){ SOP(“A”);}

B -> static M1(){ SOP(“B”);}

C -> static M1(){ SOP(“C”);}
```

C c = new C();

c.m1(); -> C

((B)c).m1(); -> B

((A)((B)c)).m1(); -> A

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

EXAMPLE 4:

```
A -> int x = 777;

B -> int x = 888;

C -> int x = 999;
```

C c = new C();

SOP(c.x); -> 999

SOP(((b)c).x); -> 888

SOP((A((B)c)).x); -> 777

Variable resolution is always based on reference type but nor based on runtime object.

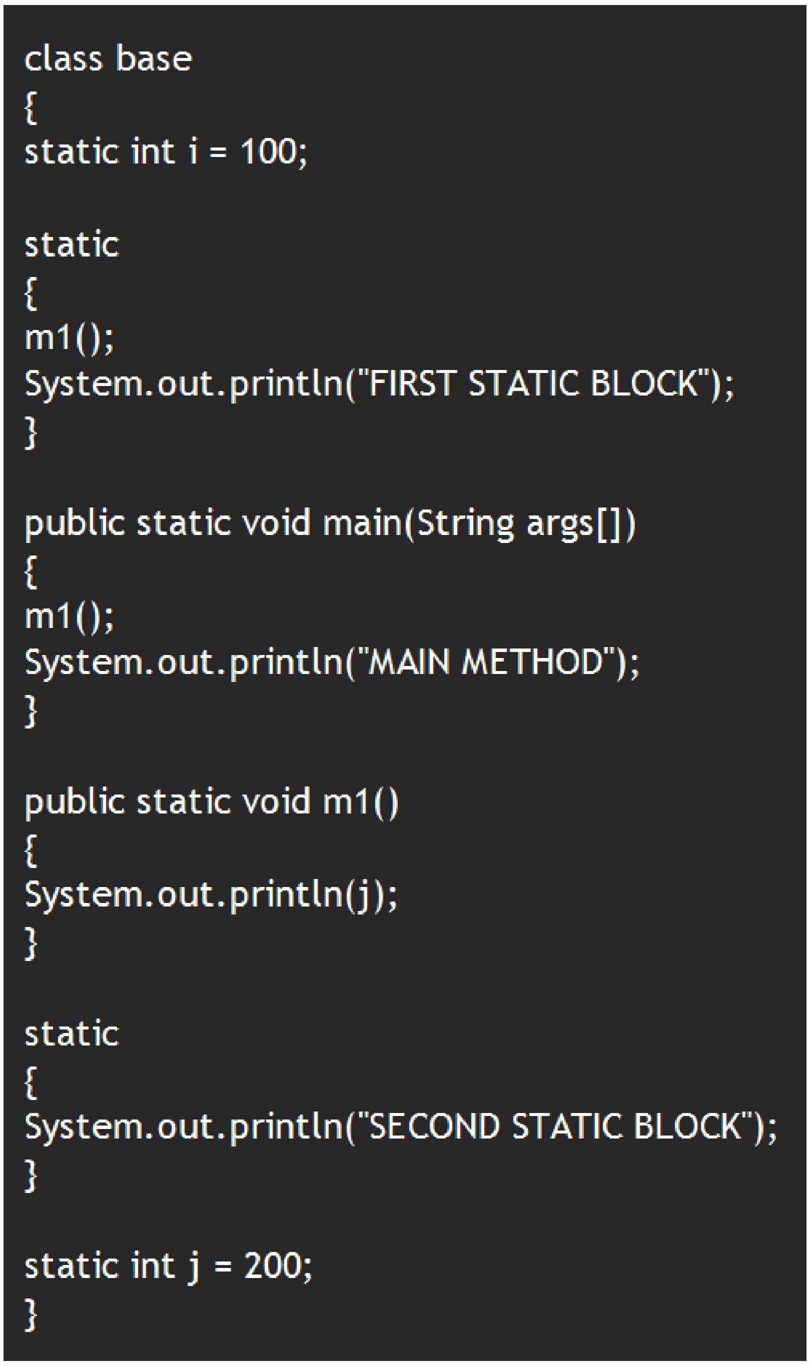
STATIC CONTROL FLOW

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

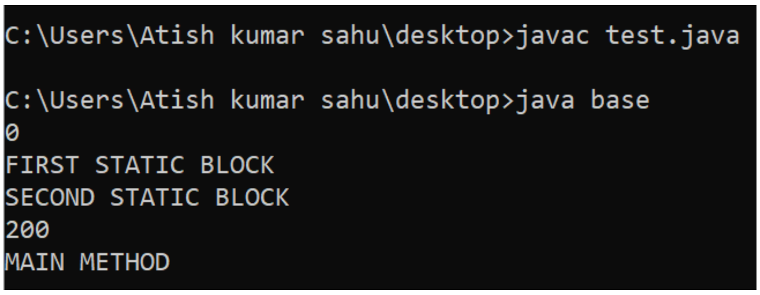
1. identification of static members from top to bottom.

2. execution of static variable assignments and static blocks from top to buttom.

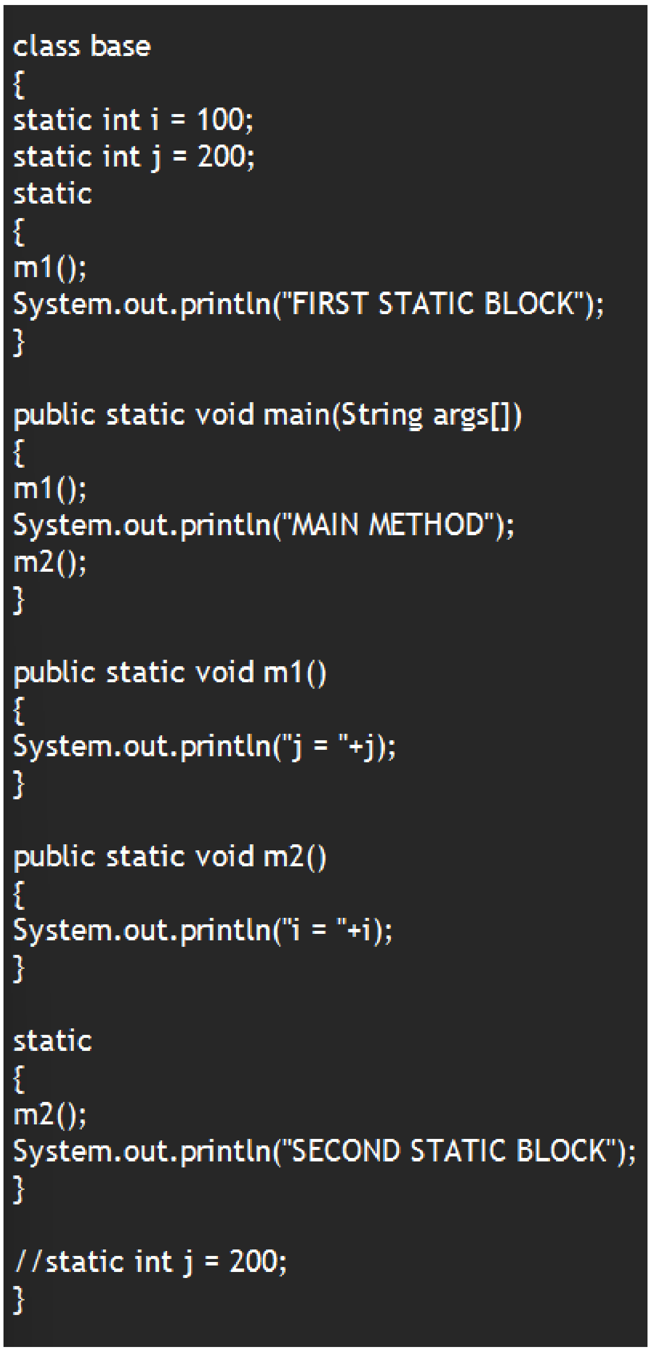
3. execution of main method.



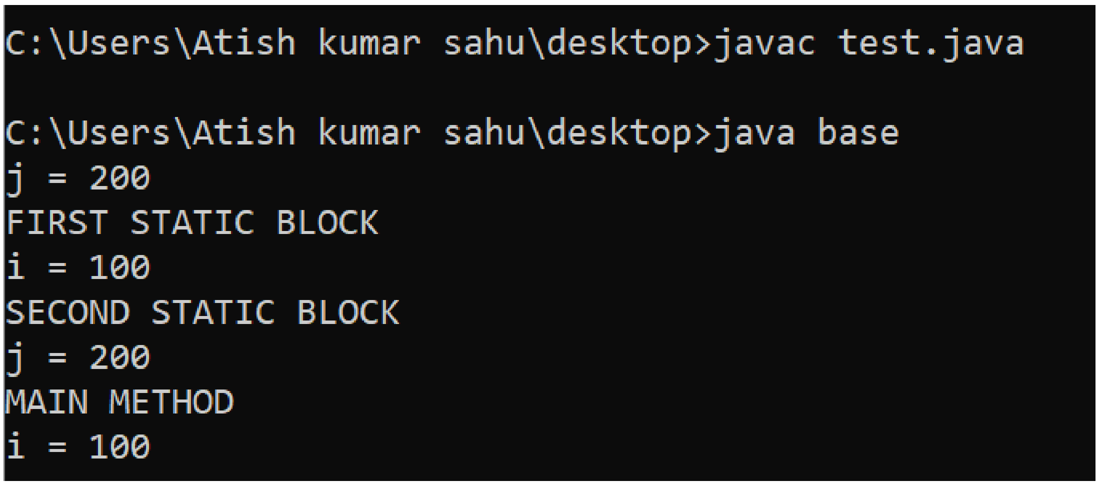
```
class base
{
static int i = 100;

static
{
m1();
System.out.println("FIRST STATIC BLOCK");
}

public static void main(String args[])
{
m1();
System.out.println("MAIN METHOD");
}

public static void m1()
{
System.out.println(j);
}

static
{
System.out.println("SECOND STATIC BLOCK");
}

static int j = 200;
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java base
0
FIRST STATIC BLOCK
SECOND STATIC BLOCK
200
MAIN METHOD
```



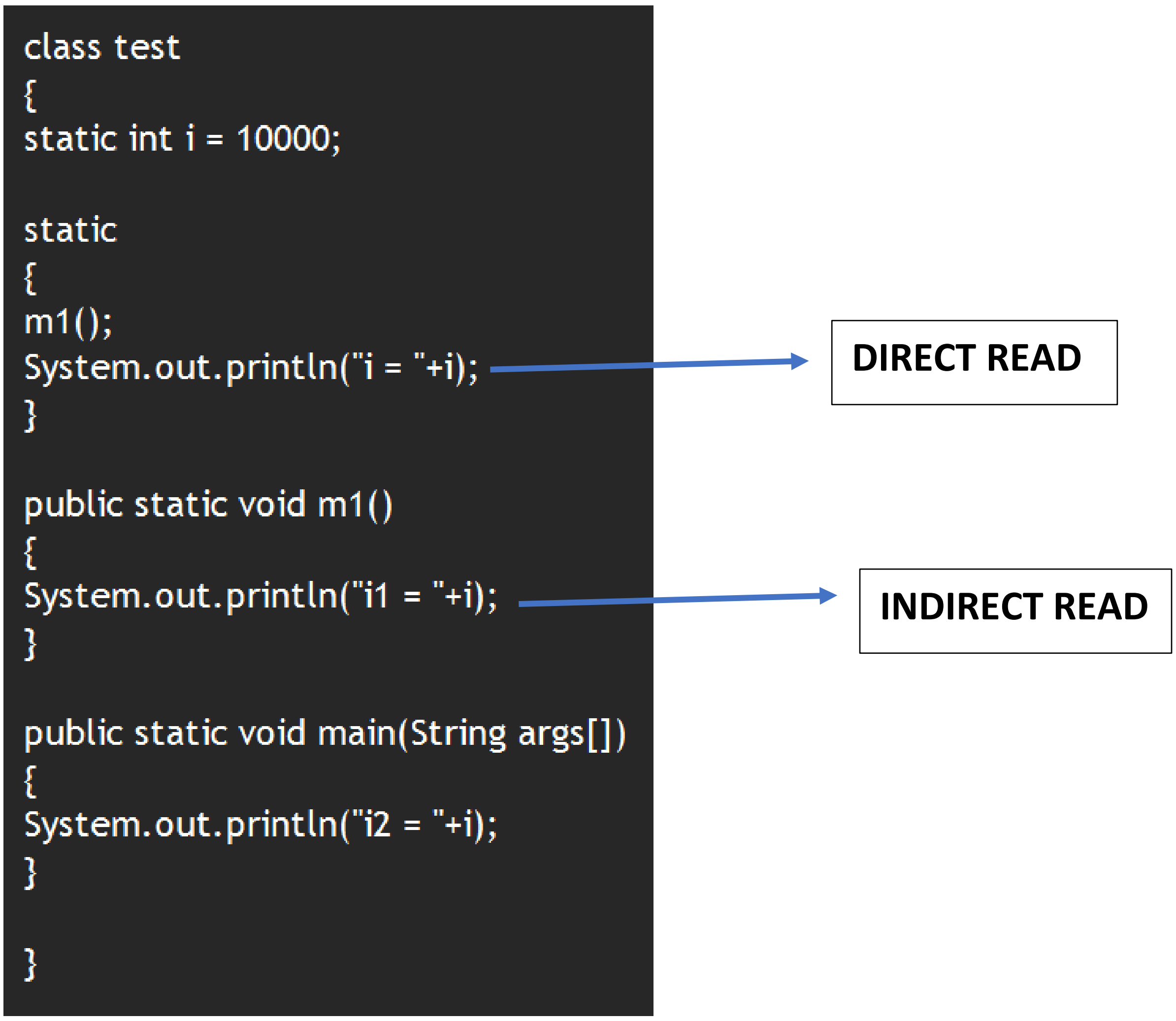
```
class base
{
static int i = 100;
static int j = 200;
static
{
m1();
System.out.println("FIRST STATIC BLOCK");
}

public static void main(String args[])
{
m1();
System.out.println("MAIN METHOD");
m2();
}

public static void m1()
{
System.out.println("j = "+j);
}

public static void m2()
{
System.out.println("i = "+i);
}

static
{
m2();
System.out.println("SECOND STATIC BLOCK");
}

//static int j = 200;
}
```



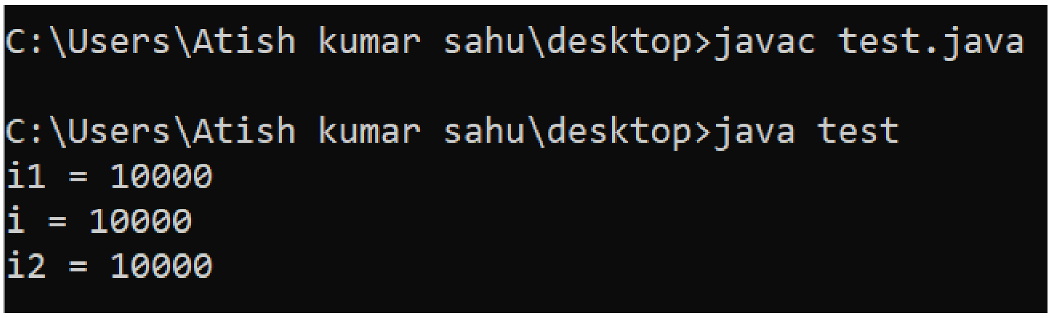
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java base
j = 200
FIRST STATIC BLOCK
i = 100
SECOND STATIC BLOCK
j = 200
MAIN METHOD
i = 100
```

READ INDIRECTLY WIRTE ONLY:

Inside a static block if we are trying to read a variable that read operation is called direct read. If we are calling a method and with in that method if we are trying to read a variable that read operation is called indirect read.

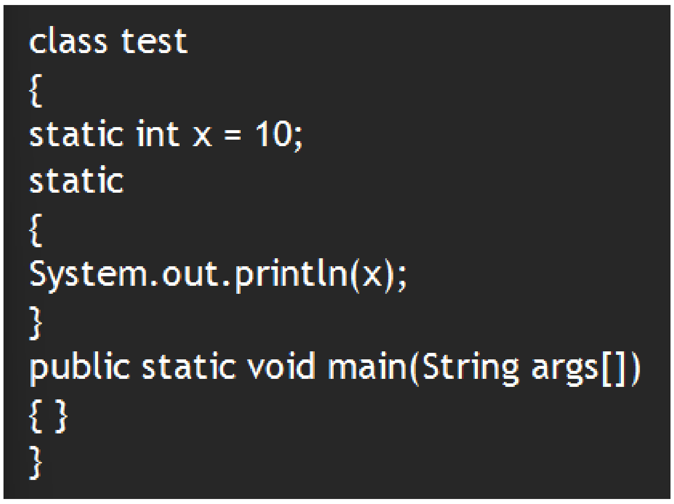


```
class test
{
static int i = 10000;

static
{
m1();
System.out.println("i = "+i);
}

public static void m1()
{
System.out.println("i1 = "+i);
}

public static void main(String args[])
{
System.out.println("i2 = "+i);
}

}
```

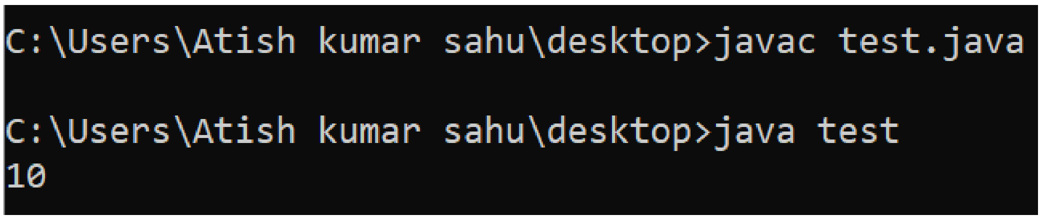


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
i1 = 10000
i = 10000
i2 = 10000
```

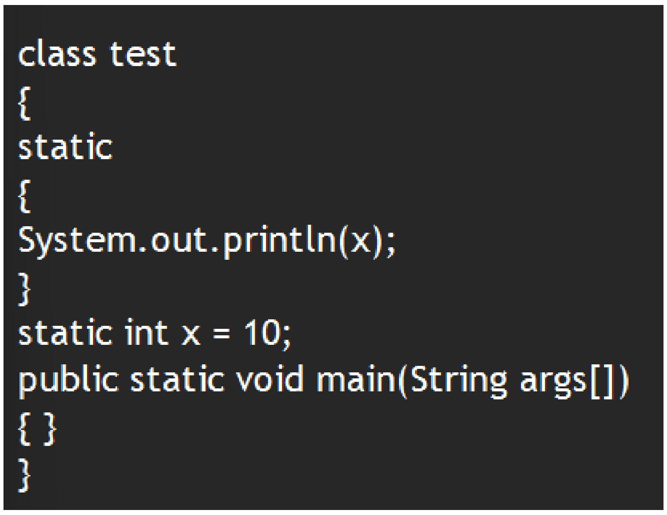
If a variable is just identified by the JVM and original value not yet assigned then the variable is said to be in “Read Indirectly and Write Only state (RIWO)”. If a variable is in read indirectly write only state then we can’t perform direct read, but we can perform indirect read. If we are trying to read directly we will get compile time error saying. Illegal forward reference.



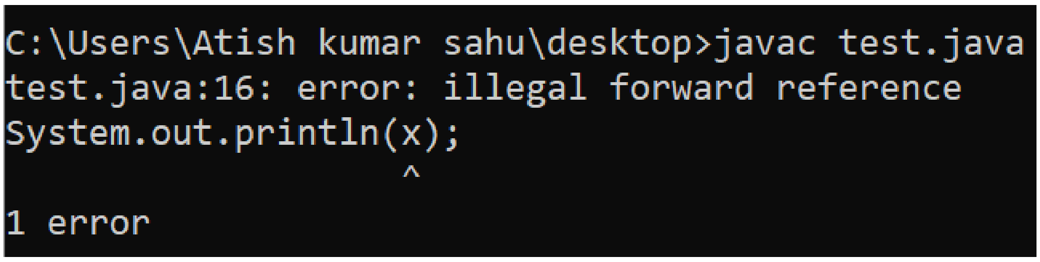
```
class test
{
static int x = 10;
static
{
System.out.println(x);
}
public static void main(String args[])
{ }
}
```



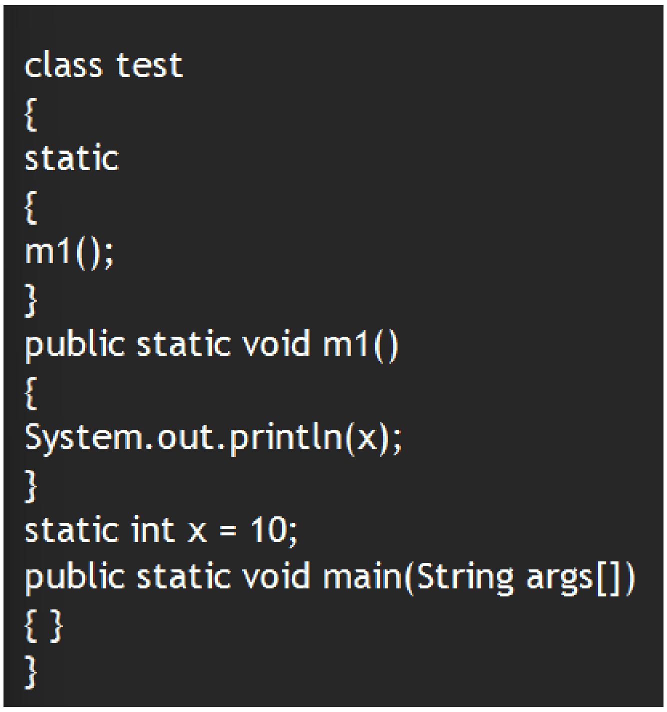
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
10
```



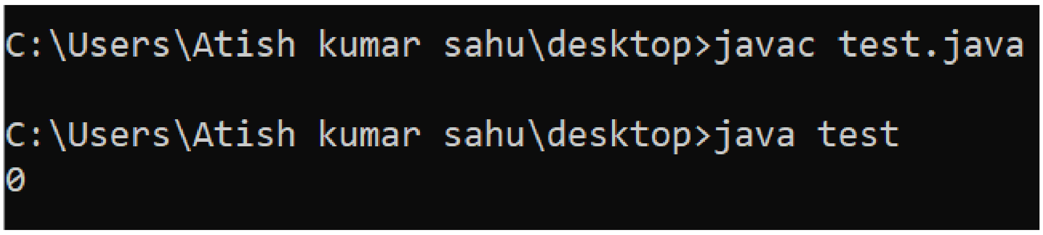
```
class test
{
static
{
System.out.println(x);
}
static int x = 10;
public static void main(String args[])
{ }
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:16: error: illegal forward reference
System.out.println(x);
                   ^
1 error
```



```
class test
{
static
{
m1();
}
public static void m1()
{
System.out.println(x);
}
static int x = 10;
public static void main(String args[])
{ }
}
```

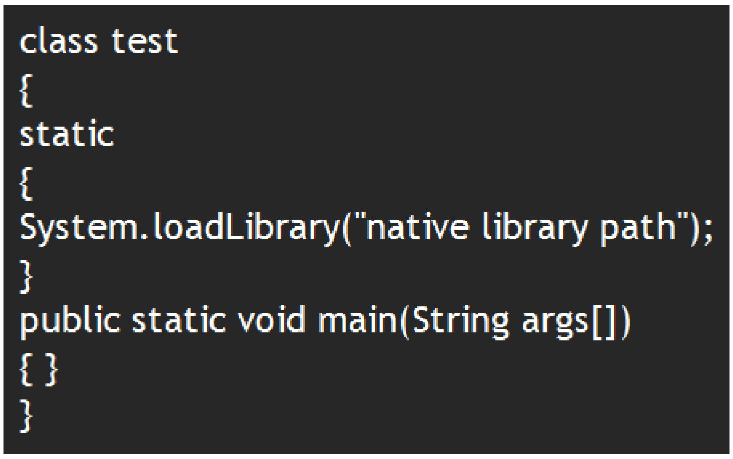


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
0
```

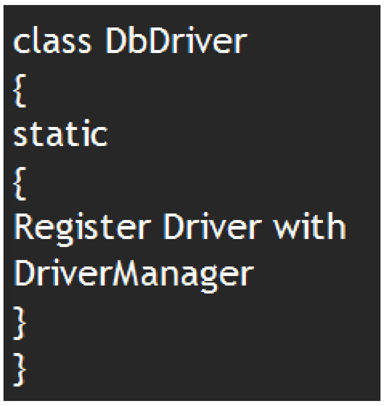
STATIC BLOCK:

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

At the time of java class loading the corresponding native library should be loaded. Hence we have to define this activity inside static block.



```
class test
{
static
{
System.loadLibrary("native library path");
}
public static void main(String args[])
{ }
}
```



```
class DbDriver
{
static
{
Register Driver with
DriverManager
}
}
```

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

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

Q. WITHOUT WRITING MAIN METHOD IS IT POSSIBLE TO PRINT SOME STATEMENT TO THE CONSOLE?

ANS: YES, BY USING STATIC BLOCK.

```
class test
{
static
{
System.out.println("HERE I CAN PRINT");
System.exit(0);
}
}
```

Q. WITHOUT WRITING MAIN METHOD AND STATIC BLOCK IS IT POSSIBLE TO PRINT SOME STATEMENTS TO CONSOLE?

ANS: YES, THERE ARE MULTIPLE WAYS USING STATIC BLOCK.

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

```
class test
{
static test t = new test();
{
System.out.println("i can print");
System.exit(0);
}
}
```

```
class test
{
static test t = new test();
test()
{
System.out.println("atish");
System.exit(0);
}
}
```

FROM JAVA 1.7 VERSION ONWARDS MAIN METHOD IS MANDATORY TO START A PROGRAM EXECUTION HENCE FROM 1.7 VERSION ONWARDS WITHOUT WRITING MAIN METHOD IT IS IMPOSSIBLE TO PRINT SOME STATEMENT TO THE CONSOLE.

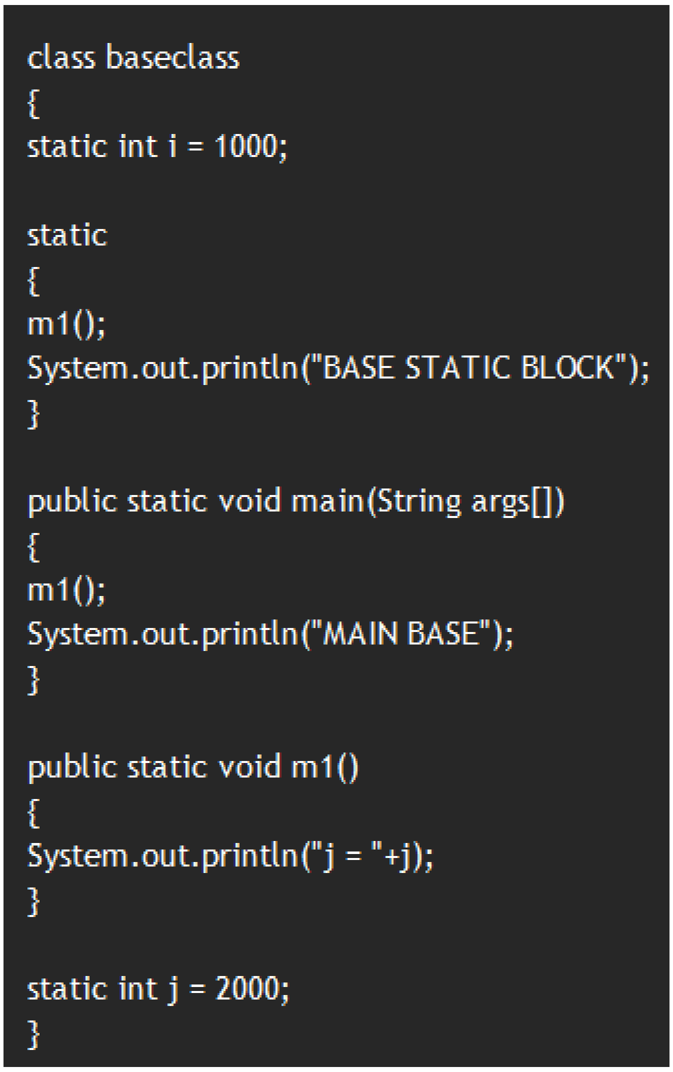
STATIC CONTROL FLOW IN PARENT TO CHILD RELATIONSHIP:

Whenever we are executing child class the following sequence of events will be executed automatically as a part of static control flow.

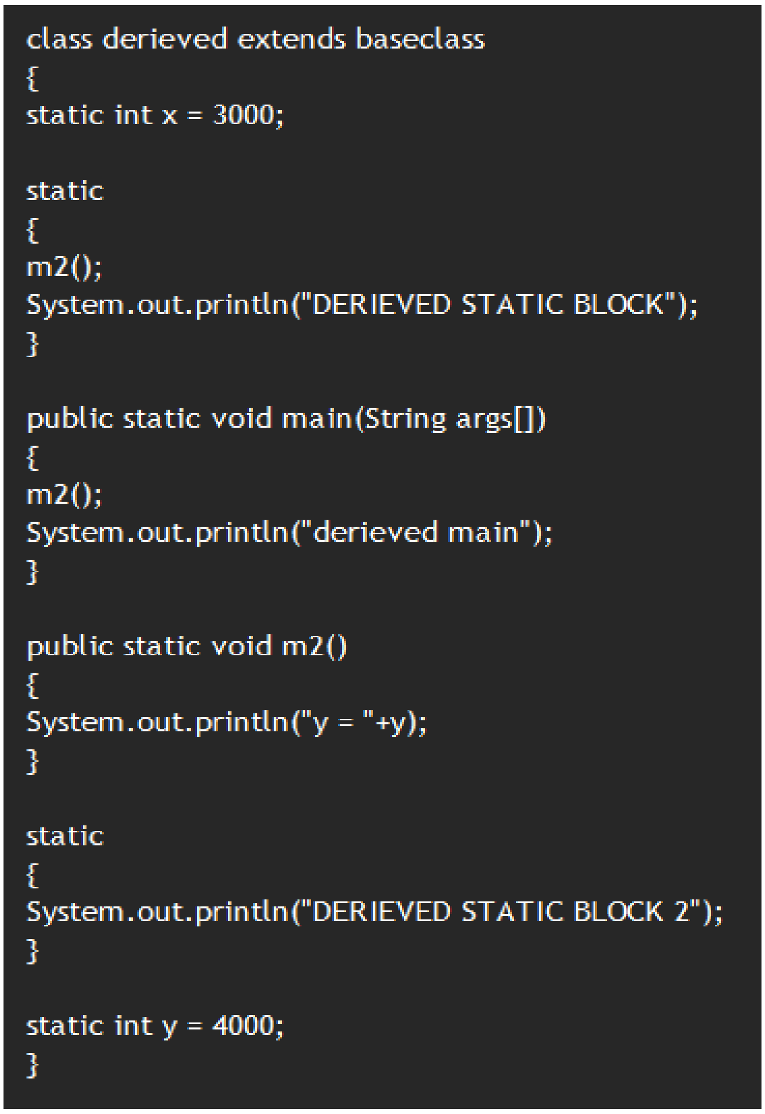
1. identification of static members from parent to child.

2. execution of static variable assignments and static blocksfrom parent to child.

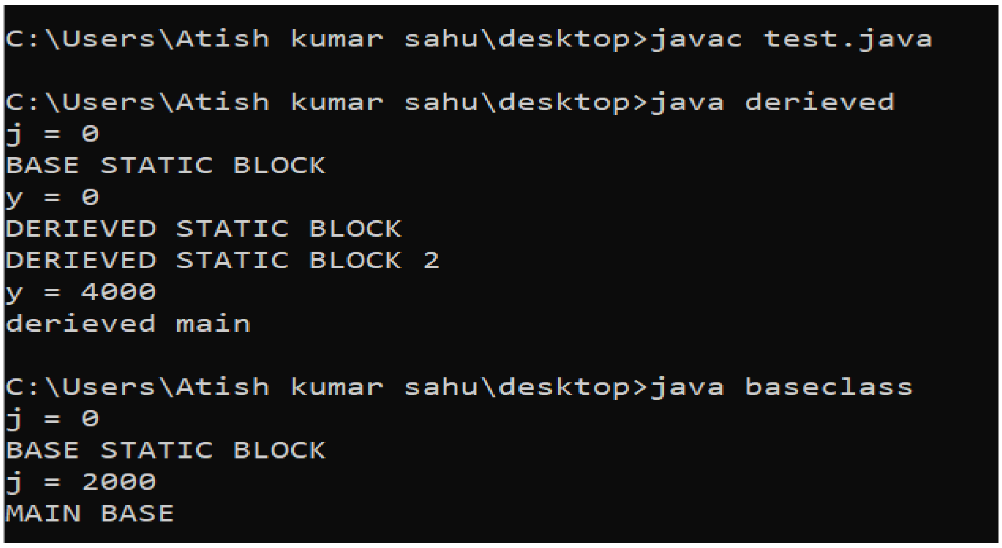
3. execution of only child class main method.



```
class baseclass
{
static int i = 1000;

static
{
m1();
System.out.println("BASE STATIC BLOCK");
}

public static void main(String args[])
{
m1();
System.out.println("MAIN BASE");
}

public static void m1()
{
System.out.println("j = "+j);
}

static int j = 2000;
}
```



```
class derieved extends baseclass
{
static int x = 3000;

static
{
m2();
System.out.println("DERIEVED STATIC BLOCK");
}

public static void main(String args[])
{
m2();
System.out.println("derieved main");
}

public static void m2()
{
System.out.println("y = "+y);
}

static
{
System.out.println("DERIEVED STATIC BLOCK 2");
}

static int y = 4000;
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java derieved
j = 0
BASE STATIC BLOCK
y = 0
DERIEVED STATIC BLOCK
DERIEVED STATIC BLOCK 2
y = 4000
derieved main

C:\Users\Atish kumar sahu\desktop>java baseclass
j = 0
BASE STATIC BLOCK
j = 2000
MAIN BASE
```

IF CHILD CLASS DOES NOT CONTAIN MAIN METHOD,

THEN PARENT CLASS MAIN METHOD WILL EXECUTED.

WHENEVER WE ARE LOADING CHILD CLASS AUTOMATICALLY PARENT CLASS WILL BE LOADED BUT WHENEVER WE ARE LOADING PARENT CLASS CHILD CLASS WON'T BE LOADED(BECAUSE PARENT CLASS MEMBERS BY DEFAULT AVAILABLE TO CHILD CLASS WHERE AS CHILD CLASS MEMBERS BY DEFAULT WON'T AVAILABLE TO THE PARENT).

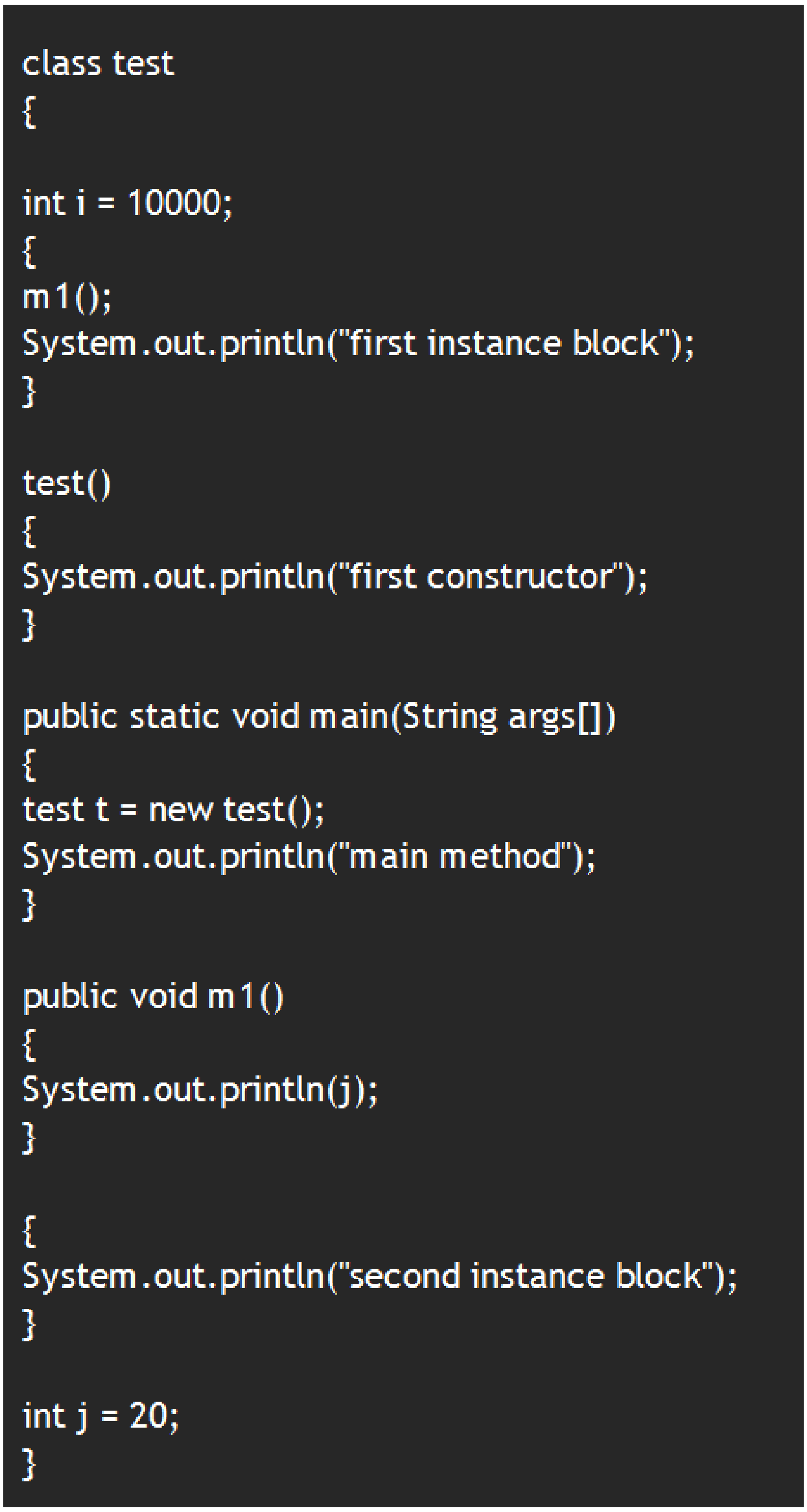
INSTANCE CONTROL FLOW:

Whenever we are executing a java class first static control flow will be executed. In the static control flow if we are creating an object the following sequence of events will be executed as a part of instance control flow.

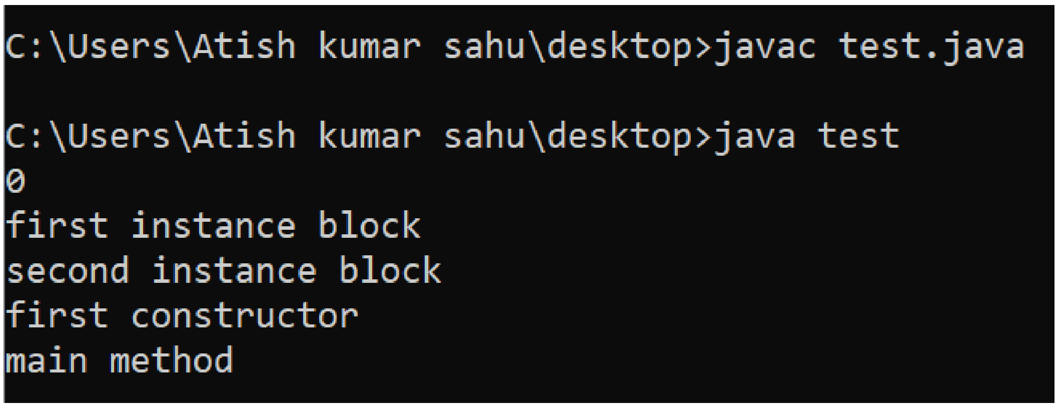
1. identification of instance member from top to buttom.

2. execution of instance variable variable assignments and instance blocks from top to buttom.

3. execution of constructor.



```
class test
{

int i = 10000;
{
m1();
System.out.println("first instance block");
}

test()
{
System.out.println("first constructor");
}

public static void main(String args[])
{
test t = new test();
System.out.println("main method");
}

public void m1()
{
System.out.println(j);
}

{
System.out.println("second instance block");
}

int j = 20;
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
0
first instance block
second instance block
first constructor
main method
```

If we comment the object creation line at main method then the output is only "System.out.println("main method");"

Static control flow is one time activity which will be performed at the time of class loading but instance control flow is not an one time activity and it will be performed for every object creation.

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

<u>INSTANCE CONTROL FLOW IN PARENT TO CHILD OBJECT:</u>

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

1. identification of instance member from parent to child.

2. execution of instance variable assignments and instance blocks only in parent class.

3. execution of parent constructor.

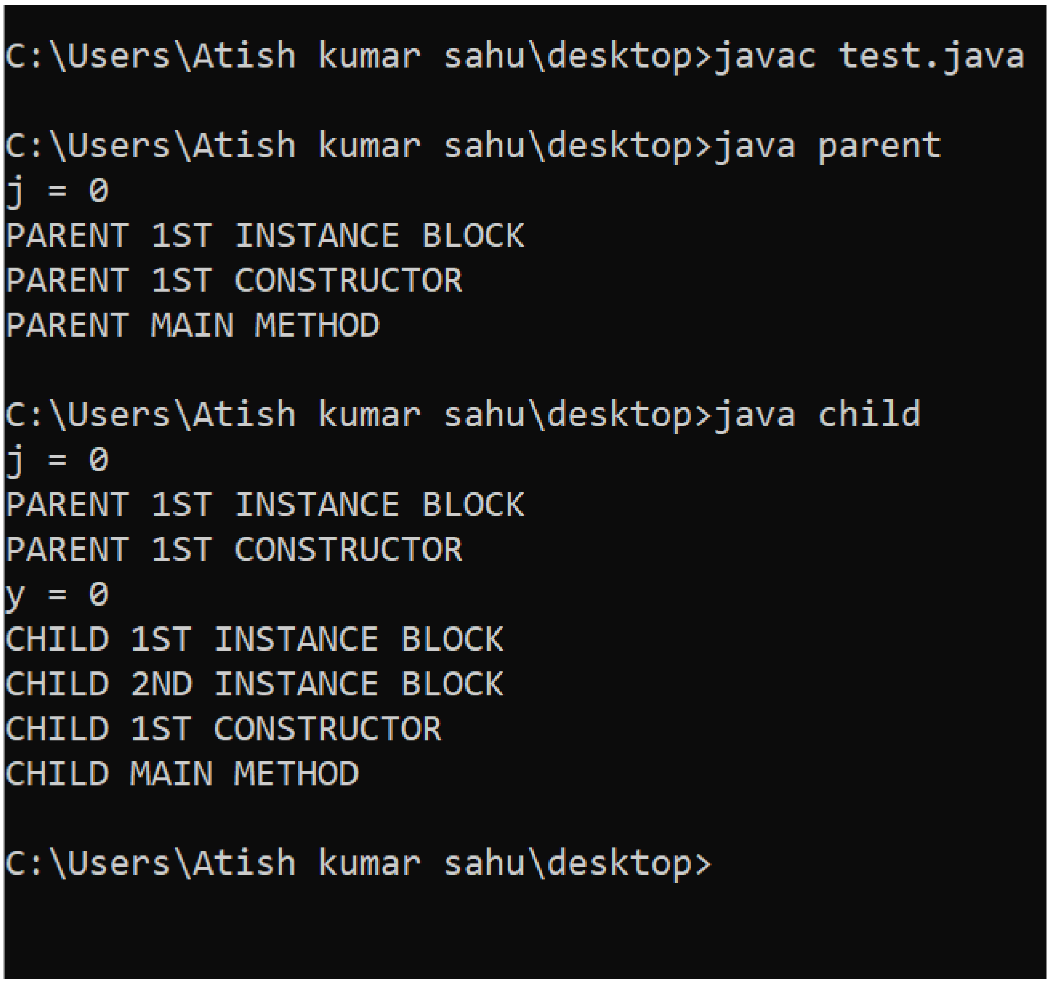
4. execution of instance variable assignments and instance blocks in child class.

5. execution of child constructor.

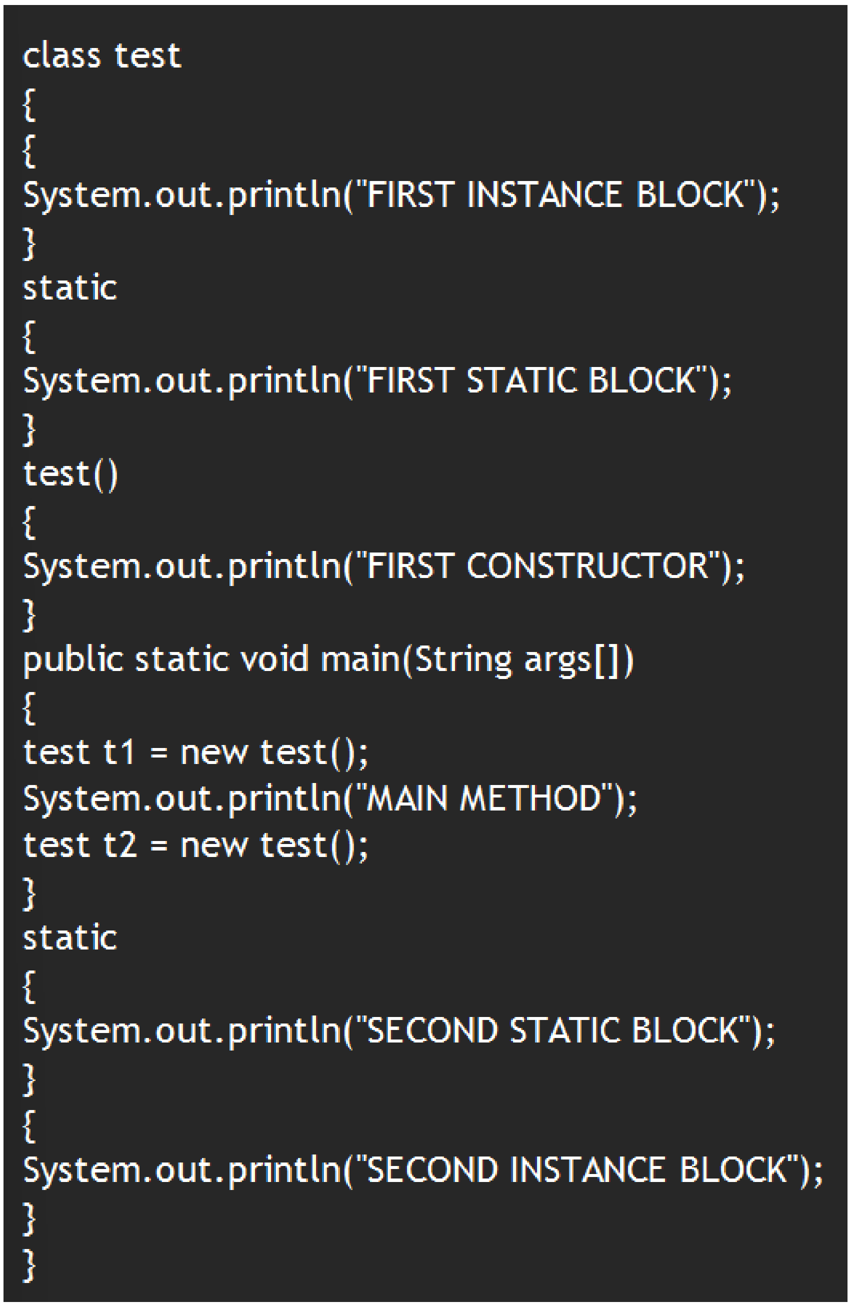
6. execution of child main class.

```
class parent
{
int i = 1000;
{
m1();
System.out.println("PARENT 1ST INSTANCE BLOCK");
}
parent()
{
System.out.println("PARENT 1ST CONSTRUCTOR");
}
public static void main(String args[])
{
parent p1 = new parent();
System.out.println("PARENT MAIN METHOD");
}
public void m1()
{
System.out.println("j = "+j);
}
int j = 2000;
}
```

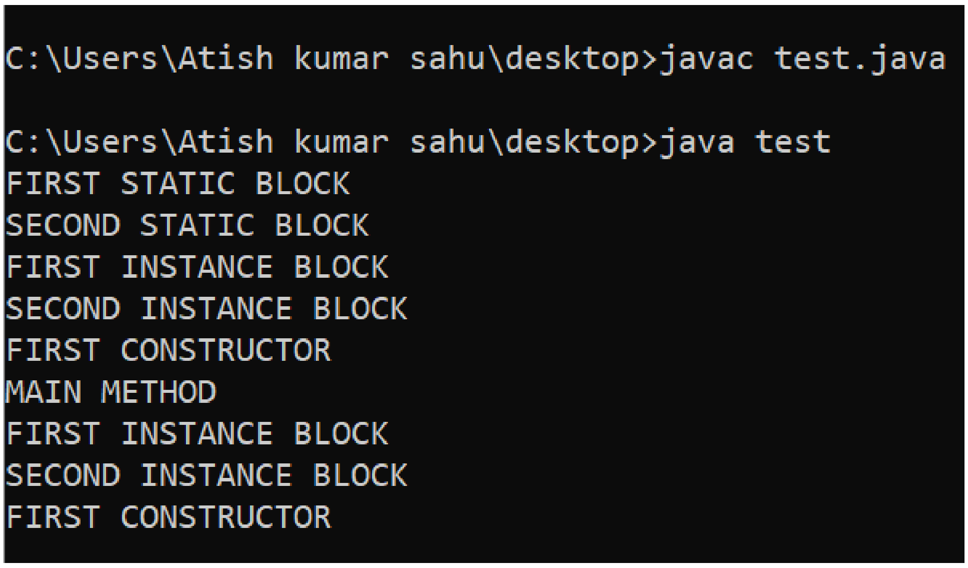
```
class child extends parent
{
int a = 3000;
{
m2();
System.out.println("CHILD 1ST INSTANCE BLOCK");
}
child()
{
System.out.println("CHILD 1ST CONSTRUCTOR");
}
public static void main(String args[])
{
child c1 = new child();
System.out.println("CHILD MAIN METHOD");
}
public void m2()
{
System.out.println("y = "+y);
}
{
System.out.println("CHILD 2ND INSTANCE BLOCK");
}
int y = 4000;
}
```



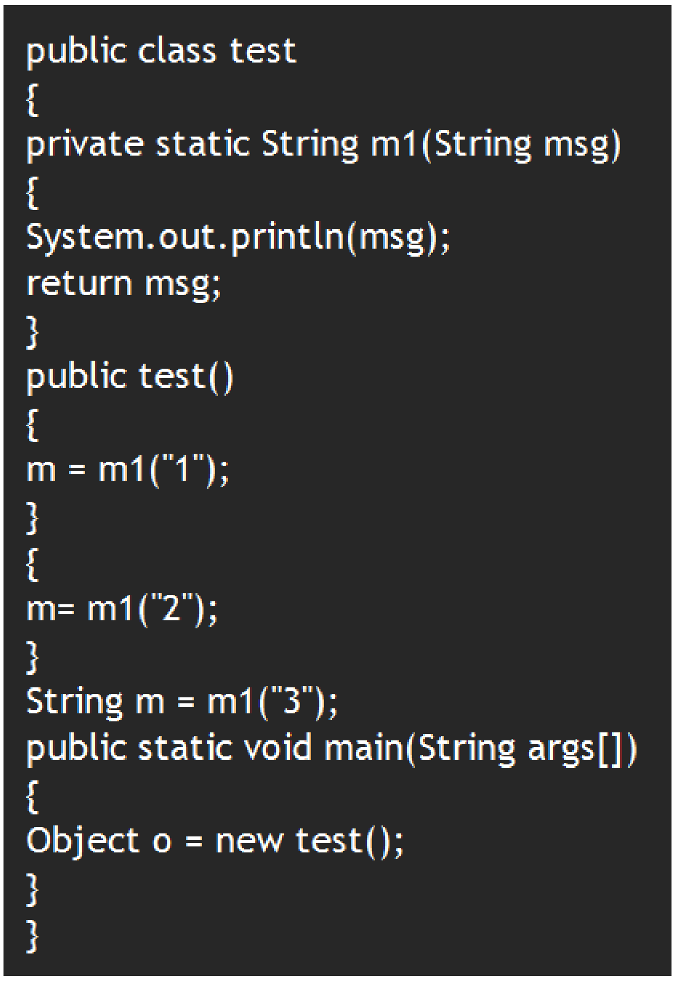
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java parent
j = 0
PARENT 1ST INSTANCE BLOCK
PARENT 1ST CONSTRUCTOR
PARENT MAIN METHOD

C:\Users\Atish kumar sahu\desktop>java child
j = 0
PARENT 1ST INSTANCE BLOCK
PARENT 1ST CONSTRUCTOR
y = 0
CHILD 1ST INSTANCE BLOCK
CHILD 2ND INSTANCE BLOCK
CHILD 1ST CONSTRUCTOR
CHILD MAIN METHOD

C:\Users\Atish kumar sahu\desktop>
```



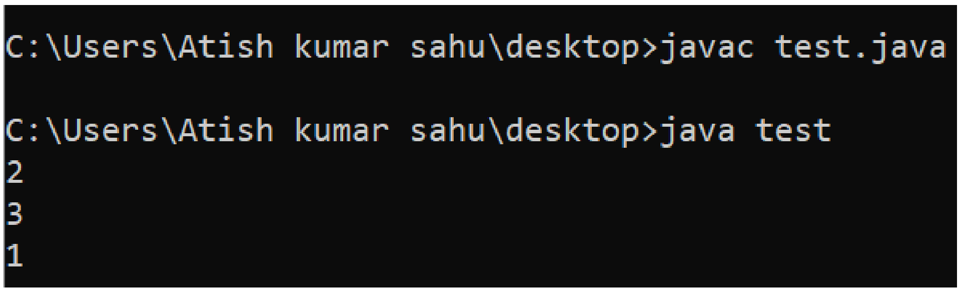
```
class test
{
{
System.out.println("FIRST INSTANCE BLOCK");
}
static
{
System.out.println("FIRST STATIC BLOCK");
}
test()
{
System.out.println("FIRST CONSTRUCTOR");
}
public static void main(String args[])
{
test t1 = new test();
System.out.println("MAIN METHOD");
test t2 = new test();
}
static
{
System.out.println("SECOND STATIC BLOCK");
}
{
System.out.println("SECOND INSTANCE BLOCK");
}
}
```



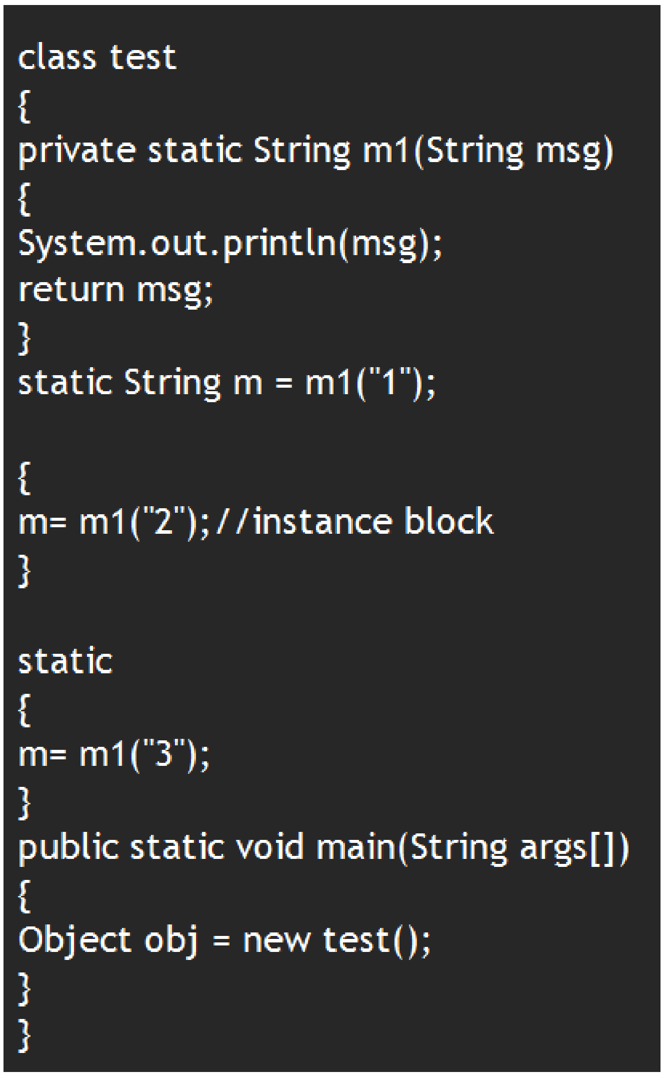
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
FIRST STATIC BLOCK
SECOND STATIC BLOCK
FIRST INSTANCE BLOCK
SECOND INSTANCE BLOCK
FIRST CONSTRUCTOR
MAIN METHOD
FIRST INSTANCE BLOCK
SECOND INSTANCE BLOCK
FIRST CONSTRUCTOR
```



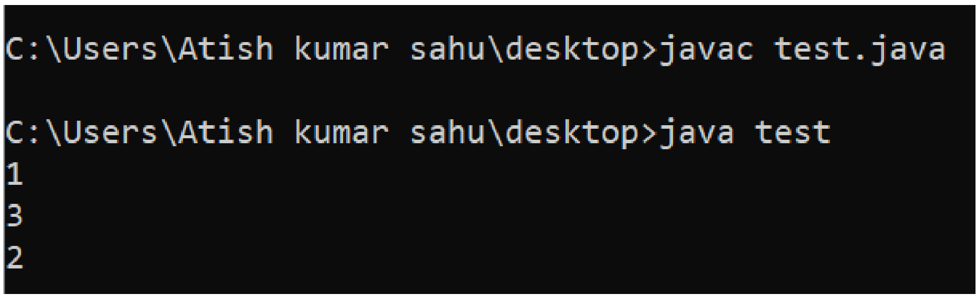
```java
public class test
{
private static String m1(String msg)
{
System.out.println(msg);
return msg;
}
public test()
{
m = m1("1");
}
{
m= m1("2");
}
String m = m1("3");
public static void main(String args[])
{
Object o = new test();
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
2
3
1
```

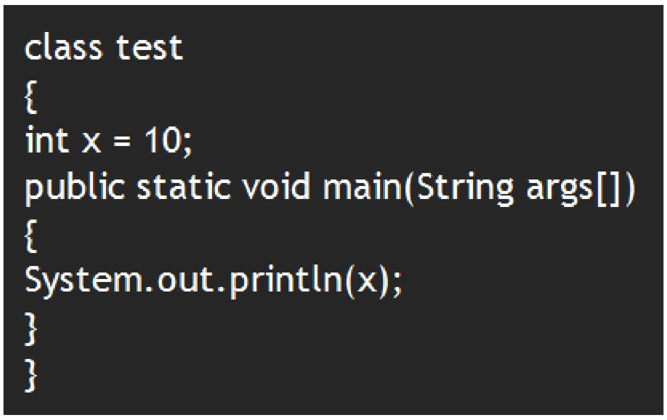


```
class test
{
private static String m1(String msg)
{
System.out.println(msg);
return msg;
}
static String m = m1("1");

{
m= m1("2");//instance block
}

static
{
m= m1("3");
}
public static void main(String args[])
{
Object obj = new test();
}
}
```

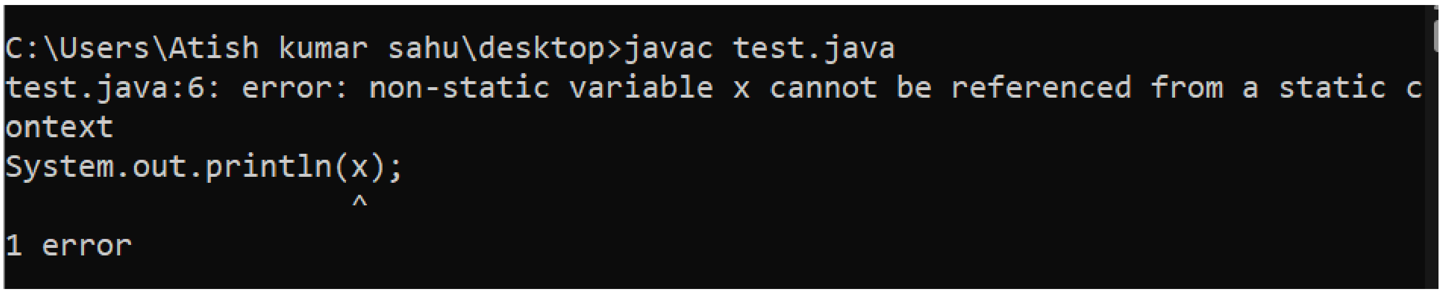


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
1
3
2
```

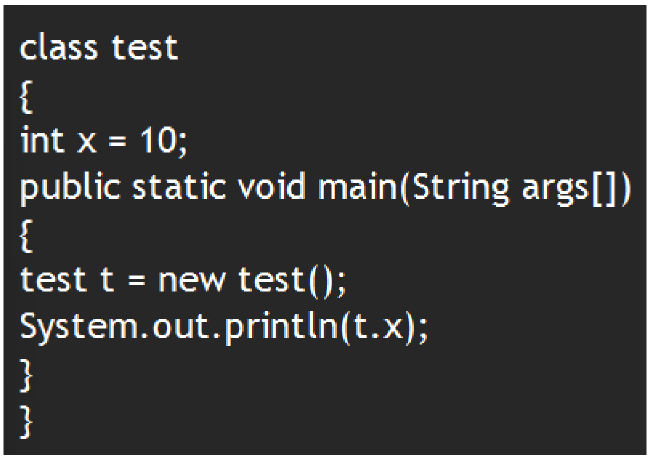
From static area we can’t access instance members directly because while executing static area JVM may not identify instance members.



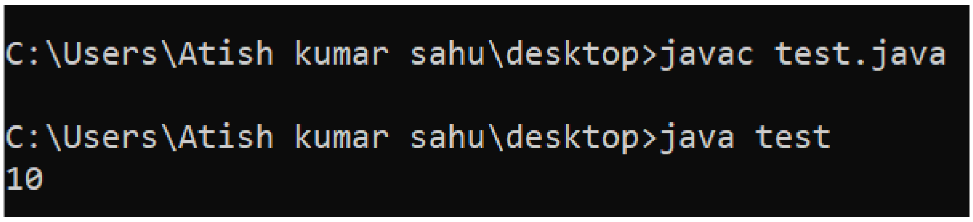
```
class test
{
int x = 10;
public static void main(String args[])
{
System.out.println(x);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:6: error: non-static variable x cannot be referenced from a static c
ontext
System.out.println(x);
                   ^
1 error
```



```
class test
{
int x = 10;
public static void main(String args[])
{
test t = new test();
System.out.println(t.x);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
10
```

**** IMPORTANT QUESTION ****

Q. IN HOWMANY WAYS WE CAN CREATE AN OBJECT IN JAVA? OR IN HOWMANY WAYS WE CAN GET OBJECT IN JAVA?

ANS.

1. BY USING NEW OPERATOR.

```
test t = new test();
```

2. BY USING NEW INSTANCE METHOD.

```
test t = (test)Classforname(“test”).newInstance();
```

3. BY USING FACTORY METHOD.

```
Runtime r = Runtime.getRuntime();

DateFormat df = DateFormat.getInstance();
```

4. BY USING CLONE METHOD.

```
test t1 = new test();

test t2 = (test)t1.clone();
```

5. BY USING DESERIALIZATION.

```
FileInputStream fis = new FileInputStream(“abc.ser”);

ObjectInputStream ois = new ObjectInputStream(fis);

Test t1 = (Test)ois.readObject();
```

SINGLETON CLASS

SINGLETON CLASS:

For any java class if we are allowed to create only one object such type of class is called singleton class. Example: Runtime, BusinessDelegate, ServiceLocator

ADVANTAGE:

If several people have same requirement then it is not recommended to create seprate object for every requirement. We have to create only one object and we can reuse the same object for every similar requirement so that performance and the memroy utilization will be improved. This is the central idea of singleton classes.

Runtime r1 = new Runtime.getRuntime();

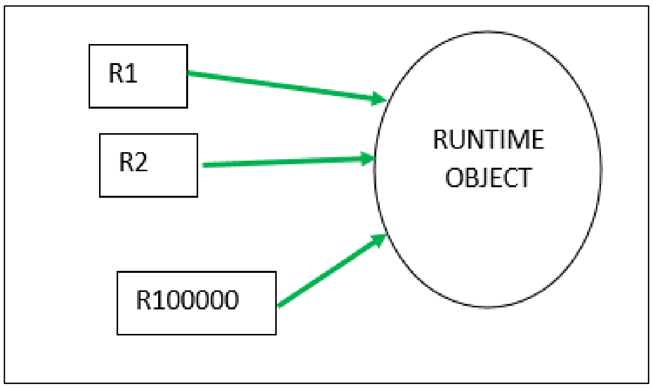
Runtime r2 = new Runtime.getRuntime();

|

|

|

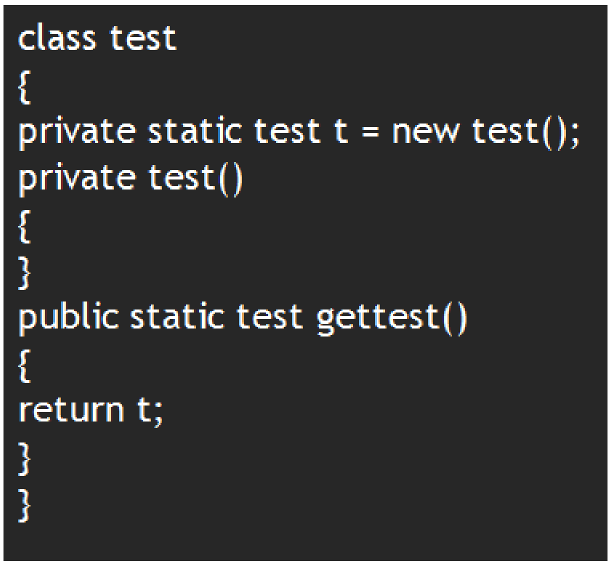
Runtime r100000 = new Runtime.getRuntime();



HOW TO CREATE OUR OWN SINGLETON CLASSES:

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

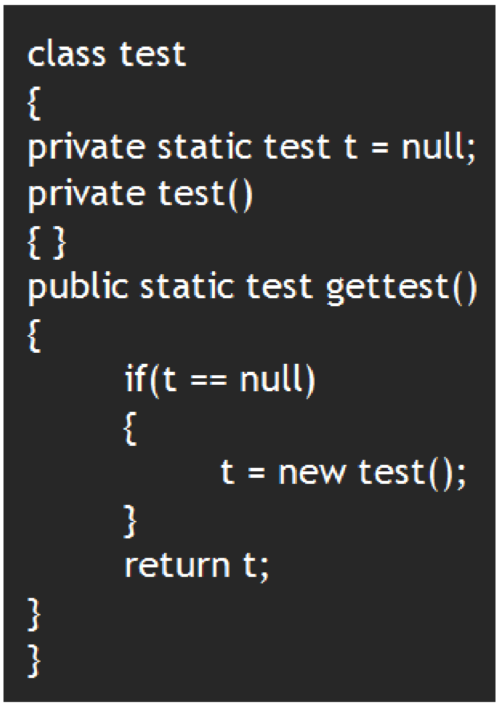
Approach 01:



```
class test
{
private static test t = new test();
private test()
{
}
public static test gettest()
{
return t;
}
}
```

Runtime class is internally implemented by using this approach.

Approach 02:

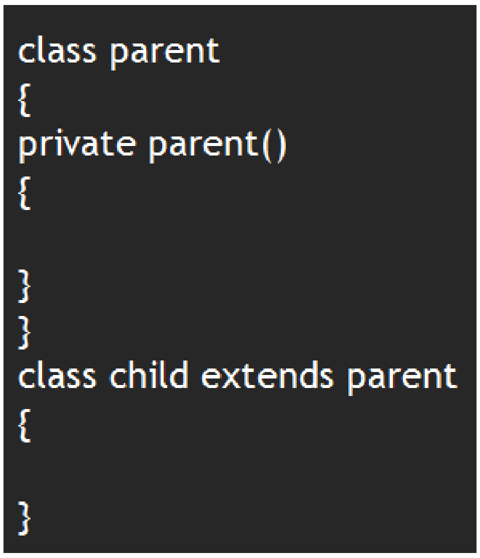


```
class test
{
private static test t = null;
private test()
{ }
public static test gettest()
{
        if(t == null)
        {
                t = new test();
        }
        return t;
}
}
```

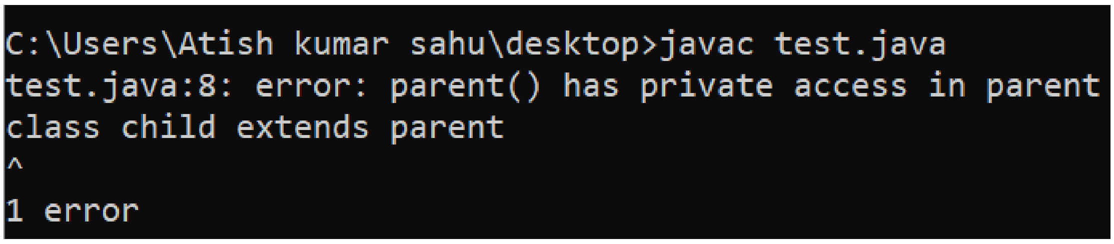
At any point of time for a test class we can create only one object hence test calss is singleton class.

Q. IF A CLASS IS NOT FINAL BUT WE ARE NOT ALLOWED TO CREATE CHILD CLASS HOW IT IS POSSIBLE?

ANS: by decalring every parent calss constructor as private we can restrict child class creation. For the below class it is impossible to create child class.



```
class parent
{
private parent()
{

}
}
class child extends parent
{

}
```



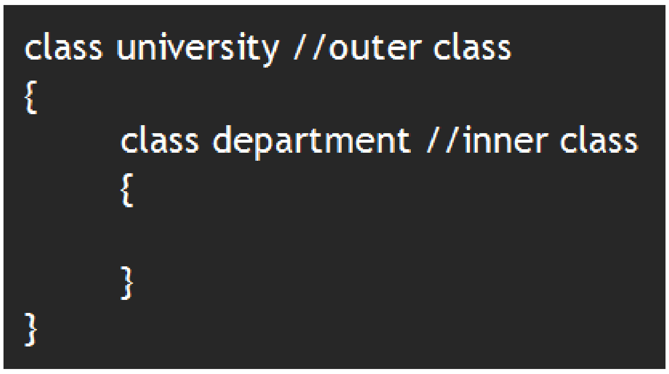
```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:8: error: parent() has private access in parent
class child extends parent
^
1 error
```

INNER CLASS

INNER CLASS:

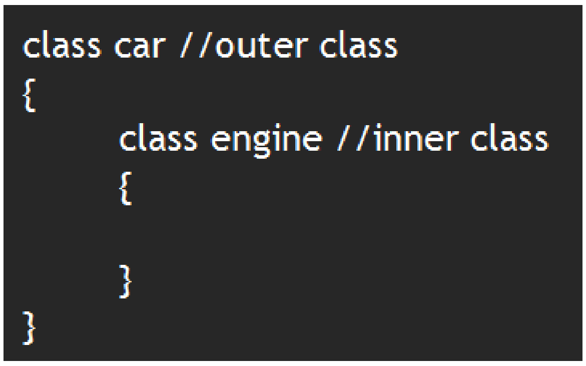
Sometimes we can declare a class inside another class such type of classes are called inner classes. Inner calsses concept **INTRODUCED IN JAVA 1.1 VERSION TO FIX GUI(GRAPHICAL USER INTERFACE) BUGS AS A PART OF EVENTHANDLING**, but because of powerful features and benefits of inner classes slowly programmers started using in regular coding also. Without existing one type of object if there is no chance of existing another type of object then we should go for inner classes.

Example 01: university consist of several departments without existing university there is no chance of existing department hence we have to declare department class inside university class.



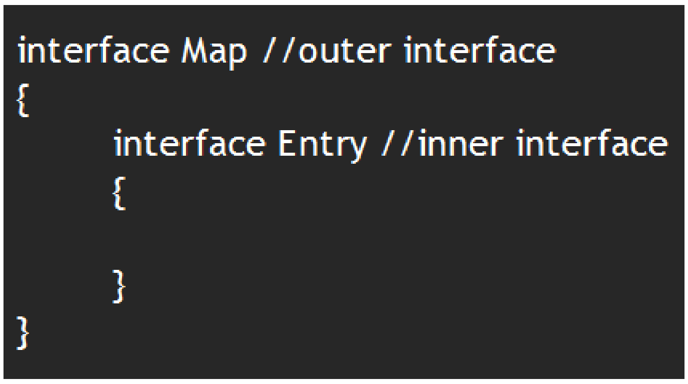
```
class university //outer class
{
	class department //inner class
	{

	}
}
```

Example 02: without existing car object there is no chance of exisitng engine object. Hence we have to declare engine class inside car class.



```
class car //outer class
{
	class engine //inner class
	{

	}
}
```

Example 03: map is a group of key value pairs and each key value pair is called an entry. Without existing map object there is no chance of existing entry object hence interface entry is define inside map interface.



```
interface Map //outer interface
{
	interface Entry //inner interface
	{

	}
}
```

NOTE:

WITHOUT EXISTING OUTER CLASS OBJECT THERE IS NO CHANCE OF EXISTING INNER CLASS OBJECT. THE RELATION BETWEEN OUTER CLASS AND INNER CLASS IS NOT IS A RELATION AND IT IS HAS A RELATIONSHIP (COMPOSITION OR AGREEGATION CONCEPT).

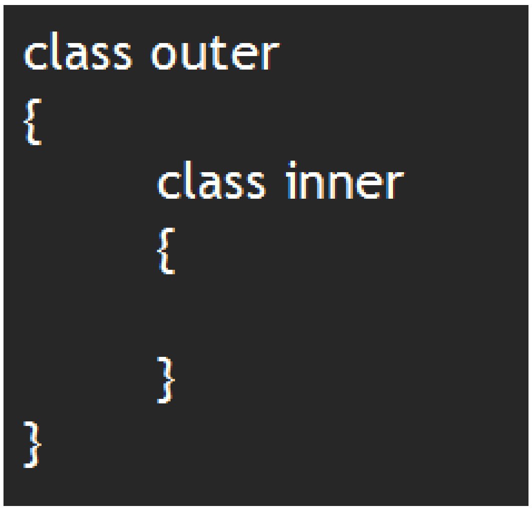
TYPES OF INNER CLASS:

Based on position of declaration and behavior all inner classes are divided into four types.

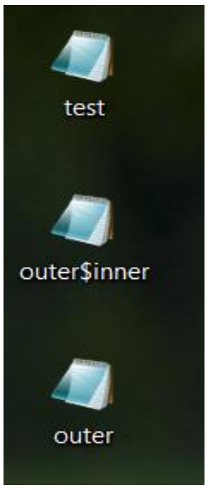
1. NORMAL OR REGULAR INNER CLASS
2. METHOD LOCAL INNER CLASS
3. ANONYMOUS INNER CLASSES
4. STATIC NESTED CLASSES

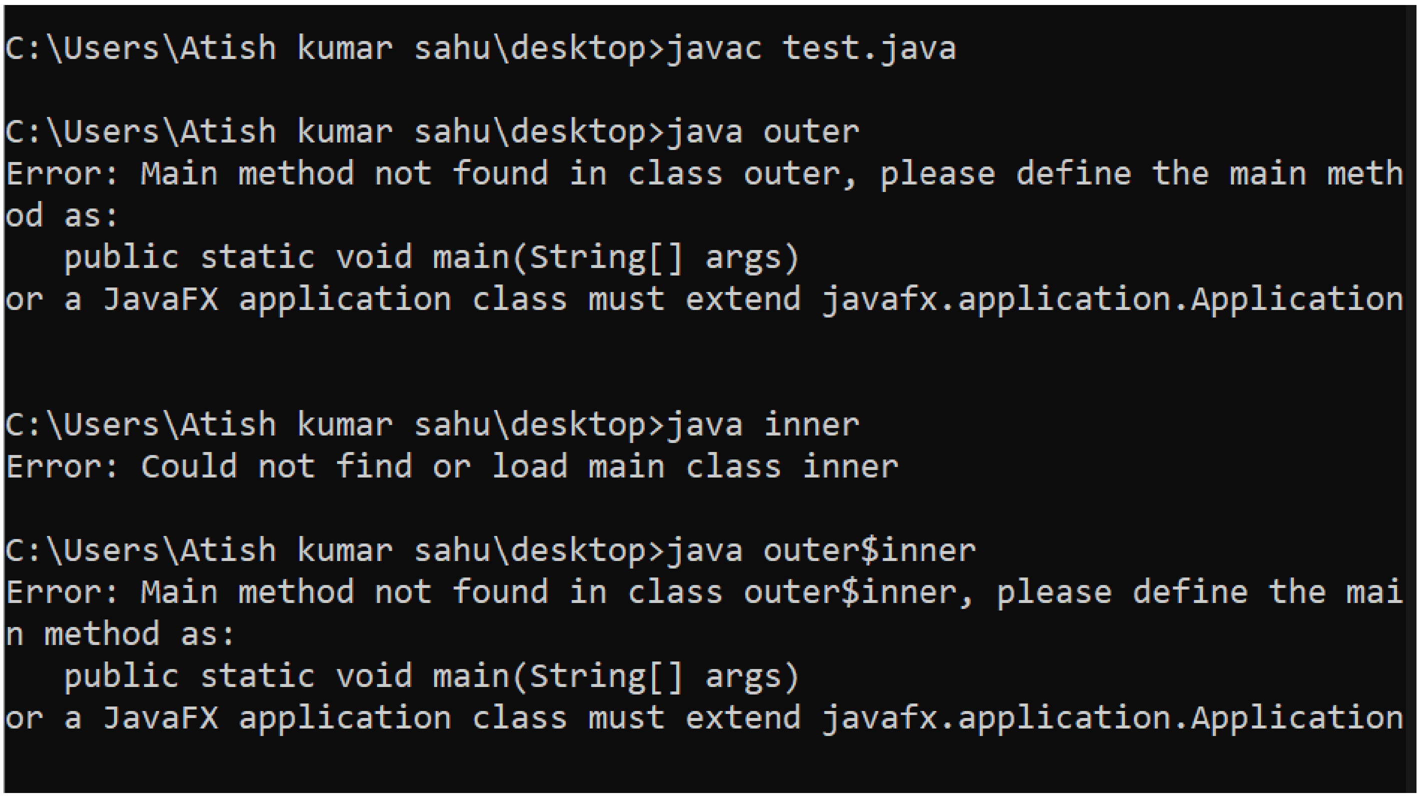
1. NORMAL OR REGULAR INNER CLASS:

If we are declaring any named class directly inside a class without using static modifier such type of inner class is called. Normal or regular inner class.

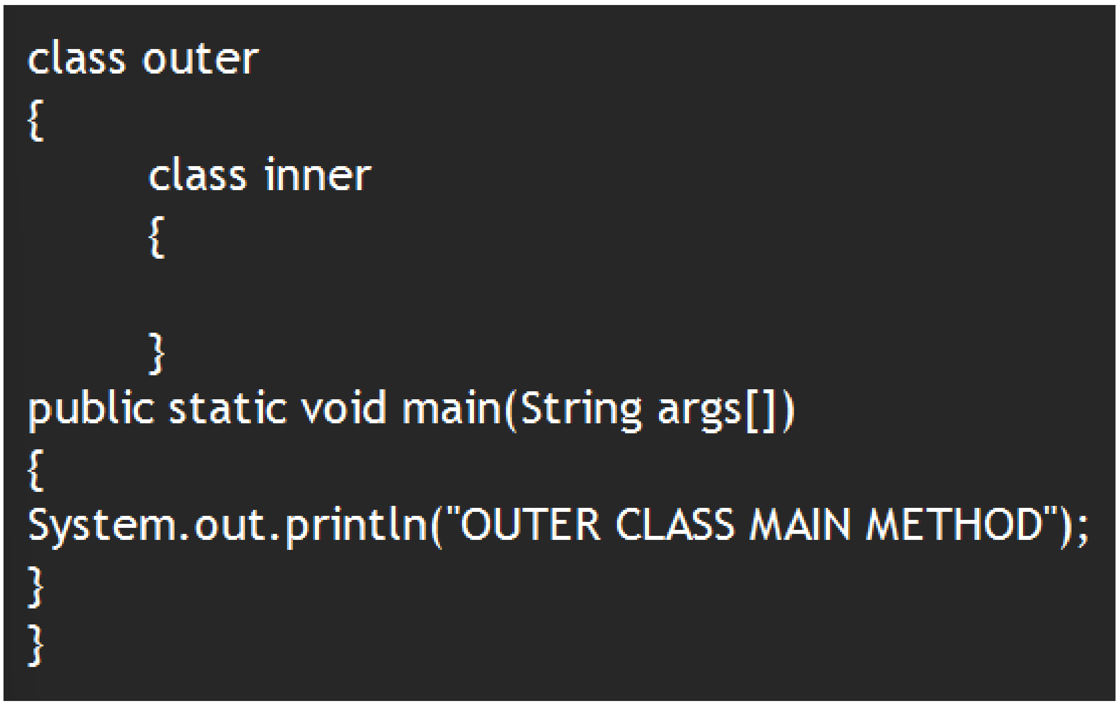


```
class outer
{
        class inner
        {

        }
}
```

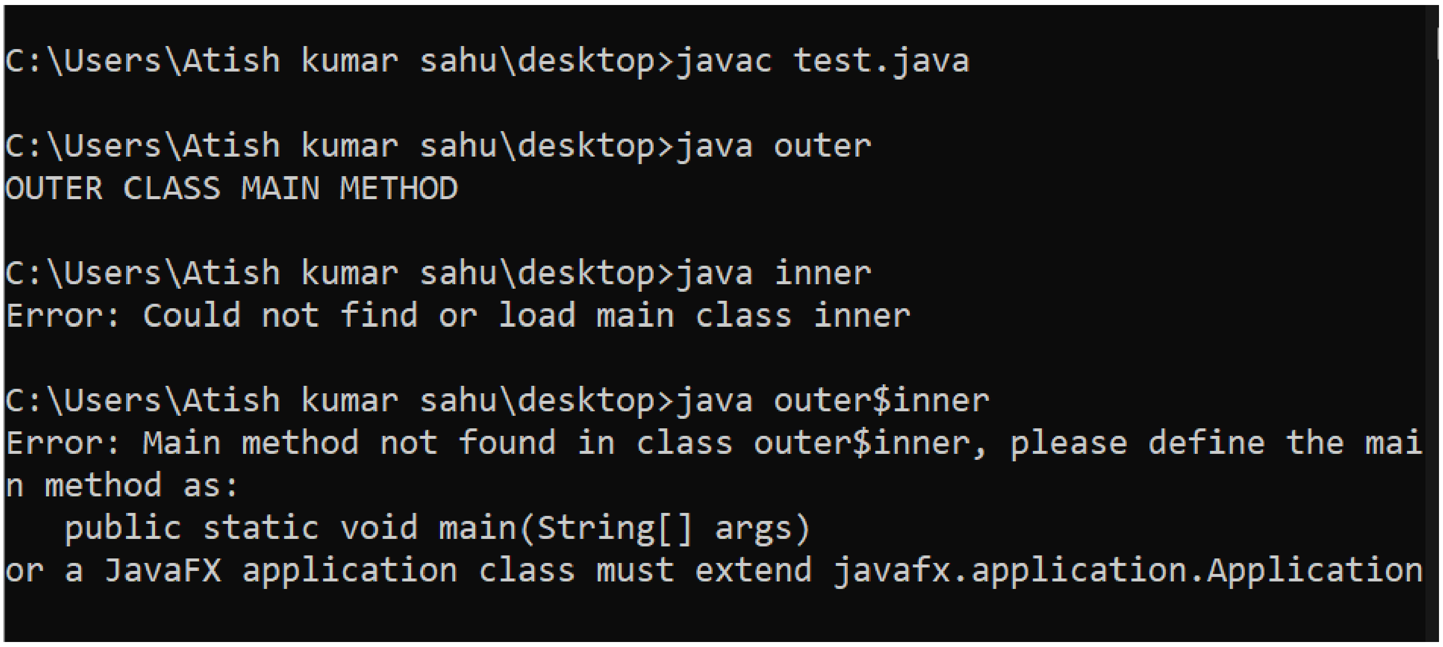




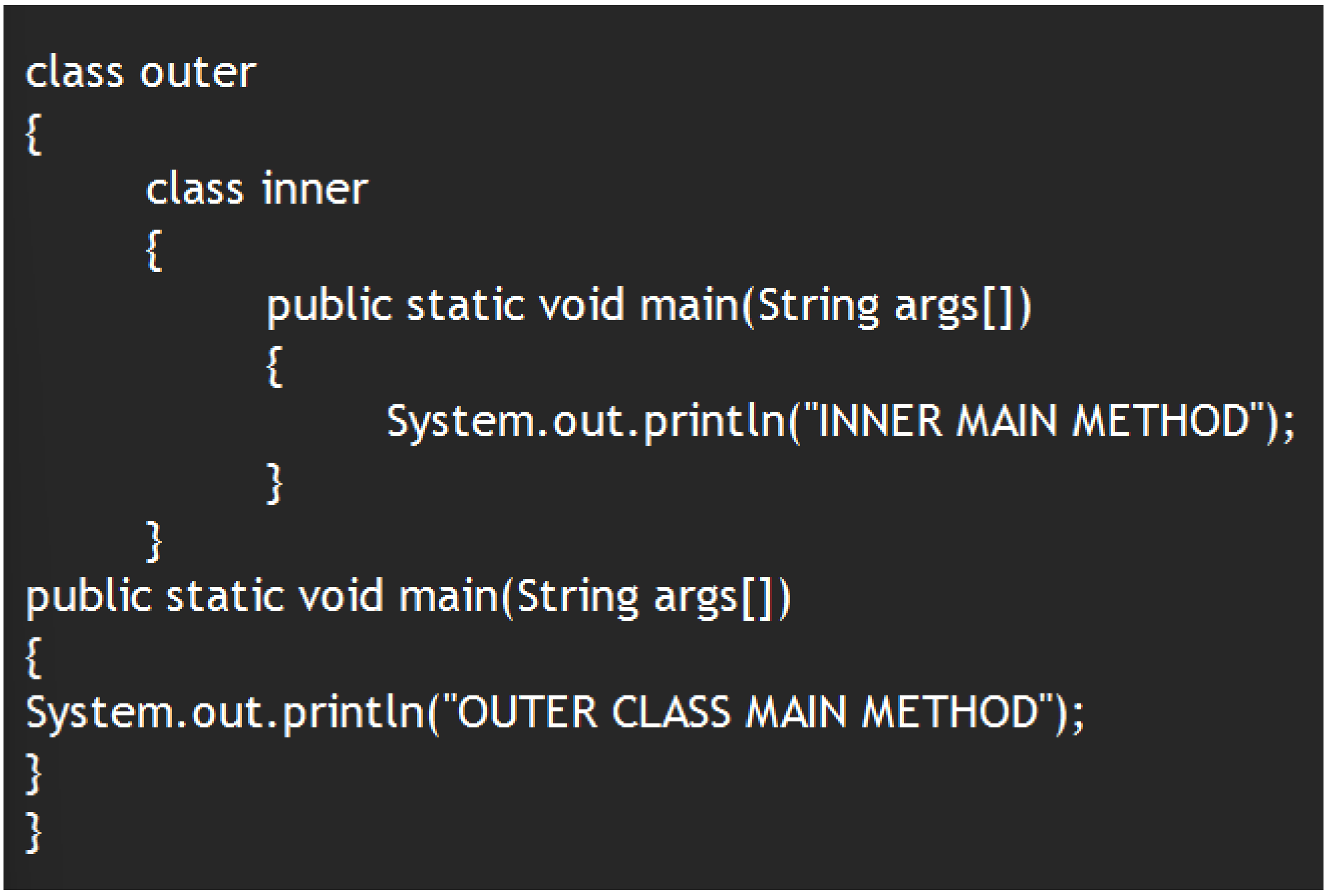
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java outer
Error: Main method not found in class outer, please define the main meth
od as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application


C:\Users\Atish kumar sahu\desktop>java inner
Error: Could not find or load main class inner

C:\Users\Atish kumar sahu\desktop>java outer$inner
Error: Main method not found in class outer$inner, please define the mai
n method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```



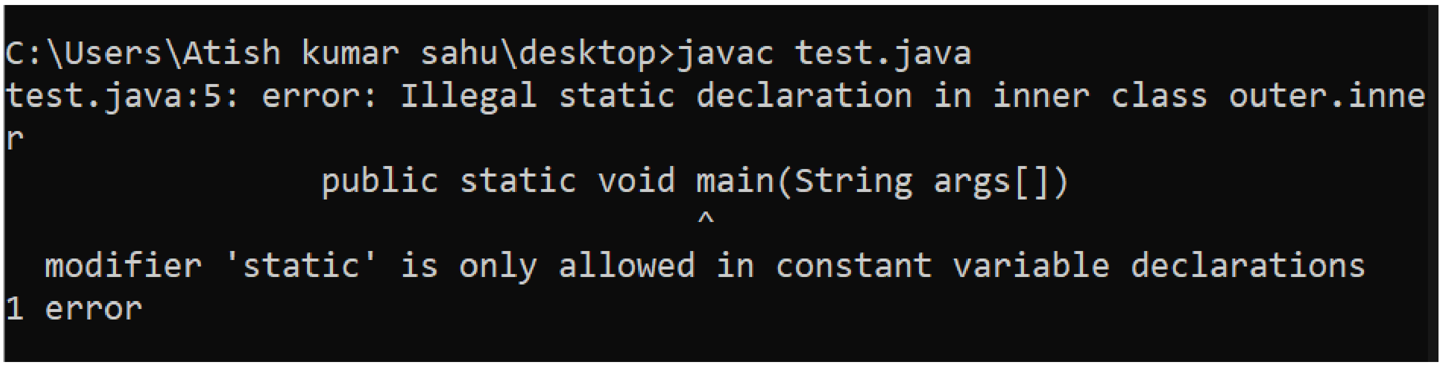
```
class outer
{
        class inner
        {

        }
public static void main(String args[])
{
System.out.println("OUTER CLASS MAIN METHOD");
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java outer
OUTER CLASS MAIN METHOD

C:\Users\Atish kumar sahu\desktop>java inner
Error: Could not find or load main class inner

C:\Users\Atish kumar sahu\desktop>java outer$inner
Error: Main method not found in class outer$inner, please define the mai
n method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```



```
class outer
{
        class inner
        {
                public static void main(String args[])
                {
                        System.out.println("INNER MAIN METHOD");
                }
        }
public static void main(String args[])
{
System.out.println("OUTER CLASS MAIN METHOD");
}
}
```

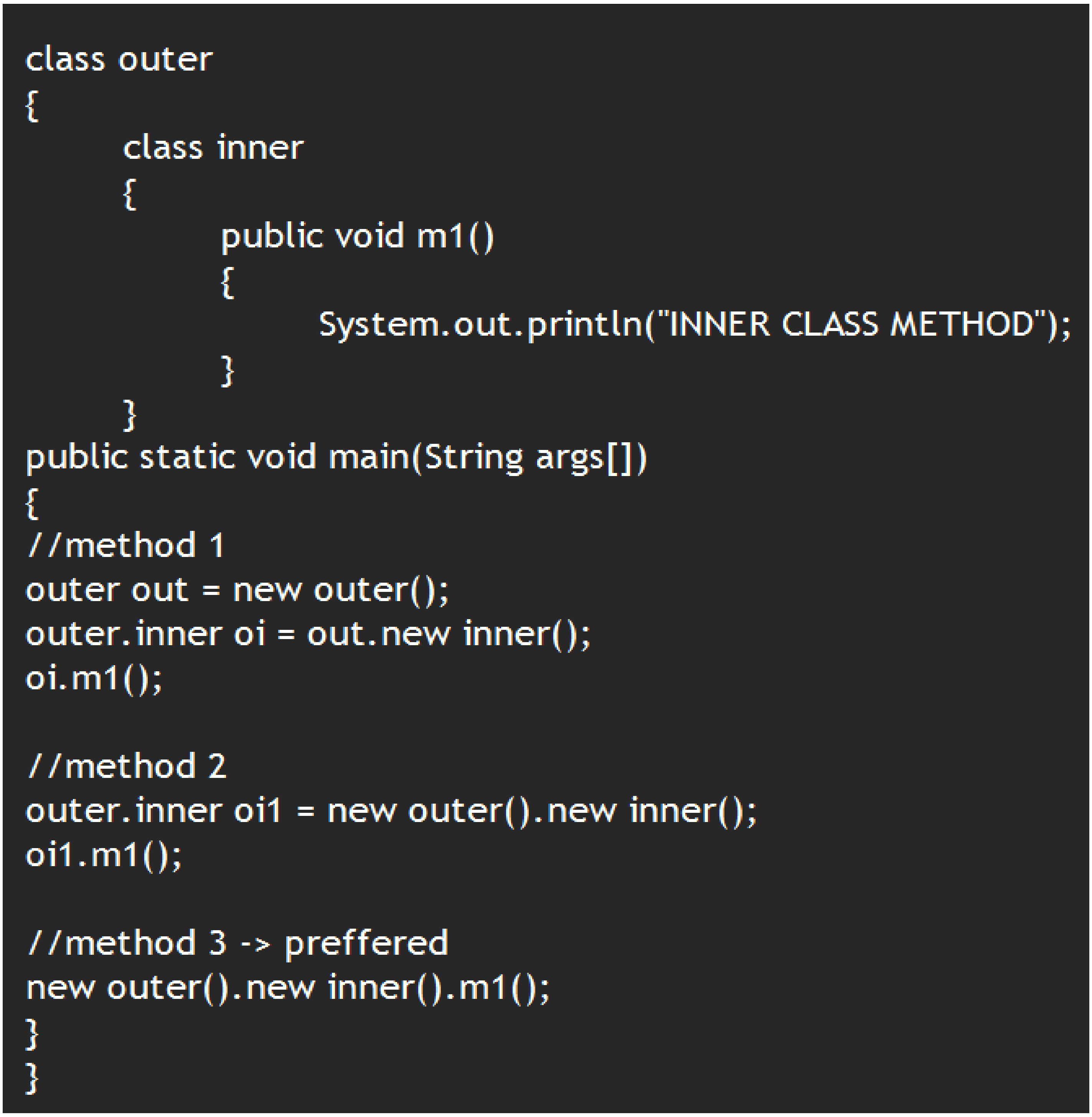


```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:5: error: Illegal static declaration in inner class outer.inne
r
                public static void main(String args[])
                                   ^
  modifier 'static' is only allowed in constant variable declarations
1 error
```

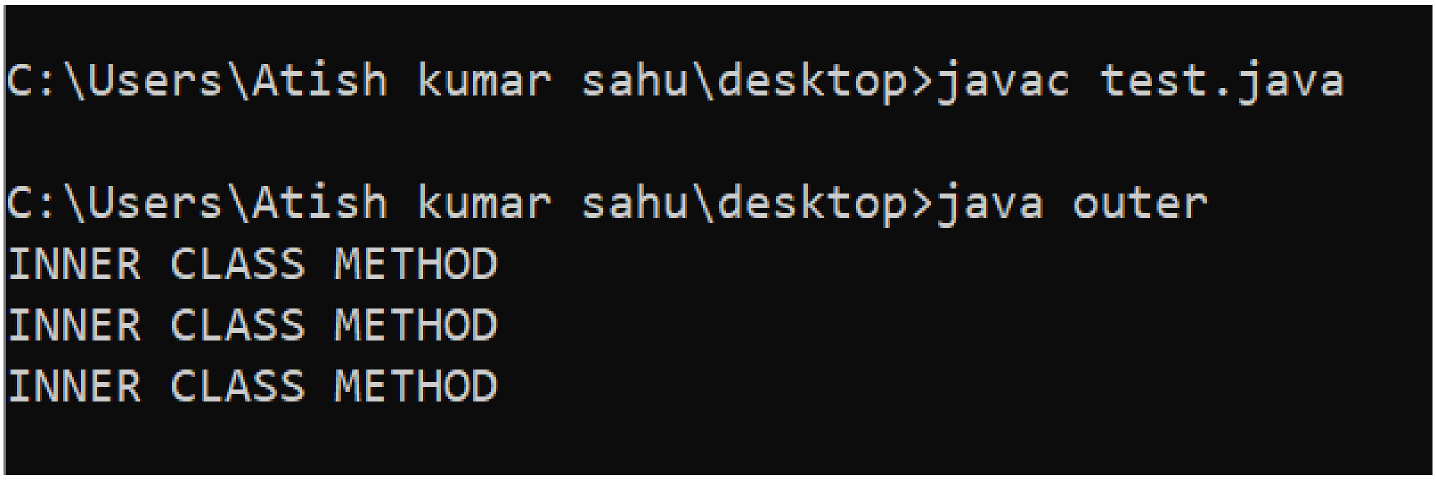
Inside inner class we can not declare any static members. Hence we can't declare main method and we can't run inner class directly from command prompt.

CASE 01:

ACCESSING INNER CLASS CODE FROM STATIC AREA OF OUTER CLASS.



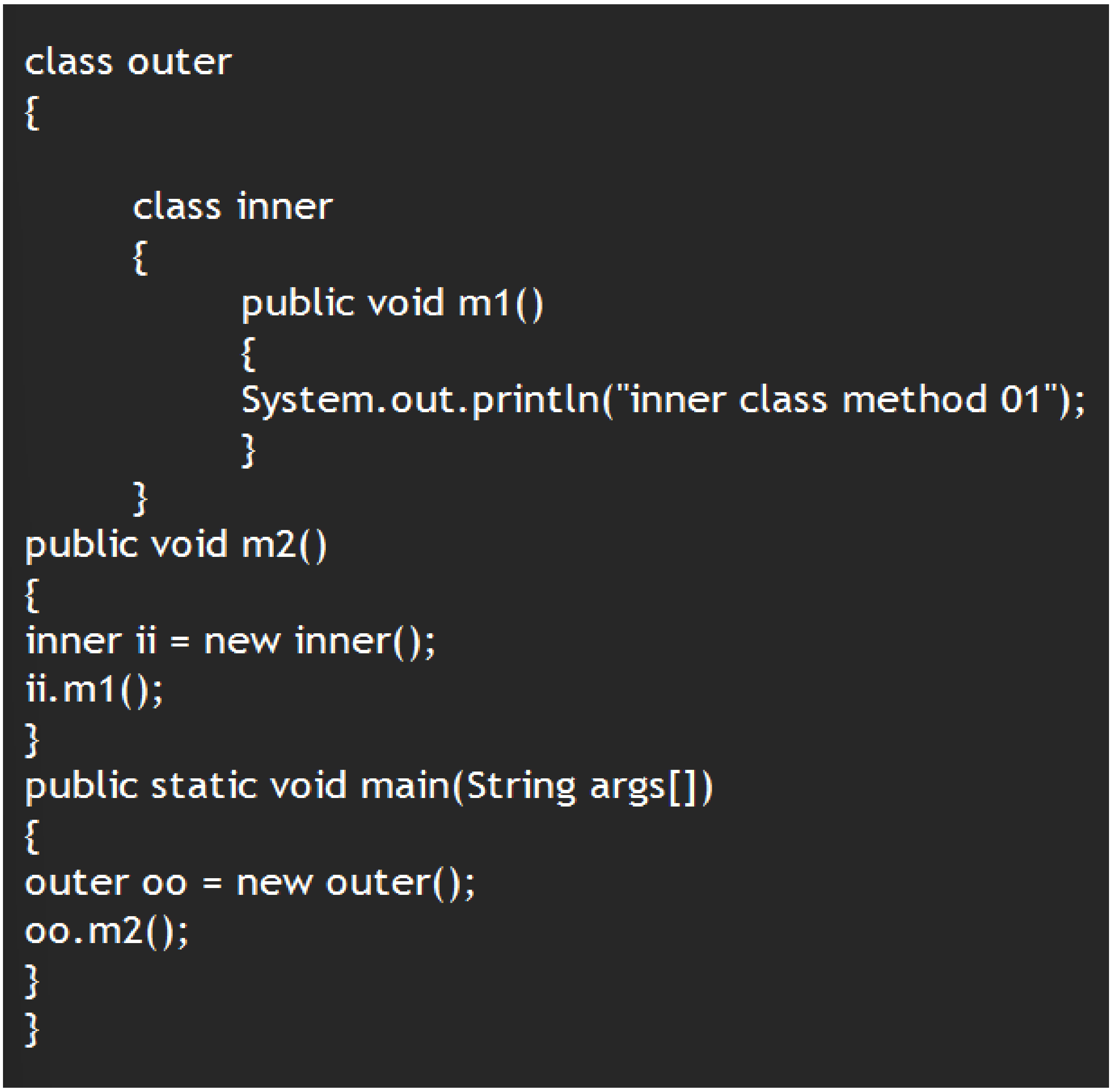
```
class outer
{
	class inner
	{
		public void m1()
		{
			System.out.println("INNER CLASS METHOD");
		}
	}
public static void main(String args[])
{
//method 1
outer out = new outer();
outer.inner oi = out.new inner();
oi.m1();

//method 2
outer.inner oi1 = new outer().new inner();
oi1.m1();

//method 3 -> preffered
new outer().new inner().m1();
}
}
```



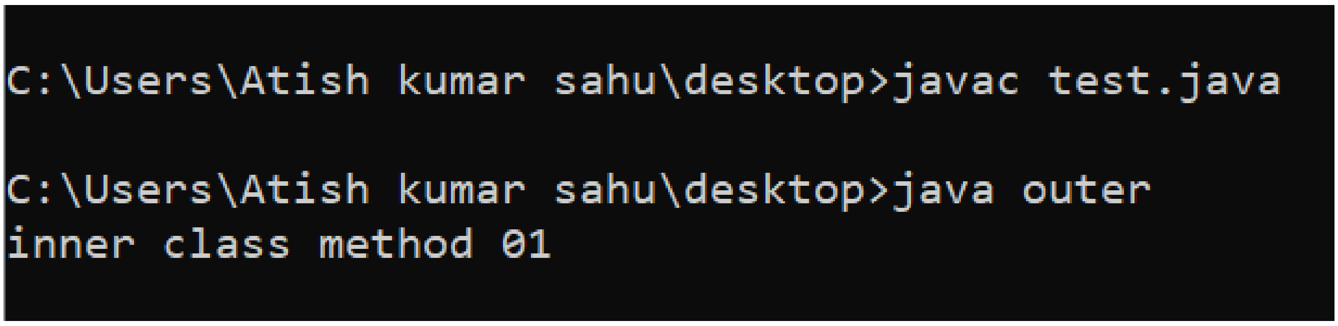
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java outer
INNER CLASS METHOD
INNER CLASS METHOD
INNER CLASS METHOD
```

CASE 02:

ACCESSING INNER CLASS CODE FROM INSTANCE AREA OF OUTER CLASS. WHICH IS VERY EASY.



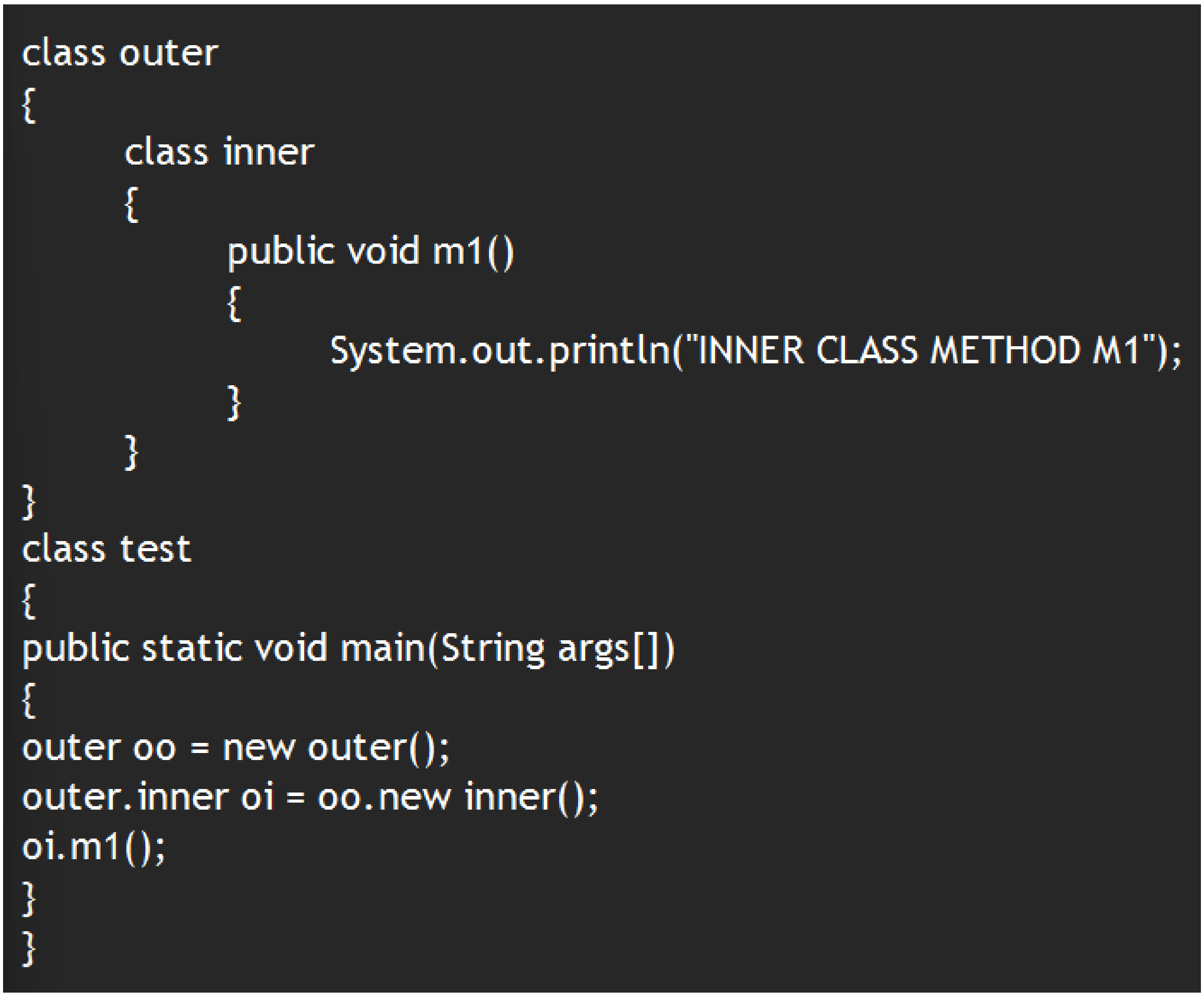
```
class outer
{

        class inner
        {
                public void m1()
                {
                System.out.println("inner class method 01");
                }
        }
public void m2()
{
inner ii = new inner();
ii.m1();
}
public static void main(String args[])
{
outer oo = new outer();
oo.m2();
}
}
```



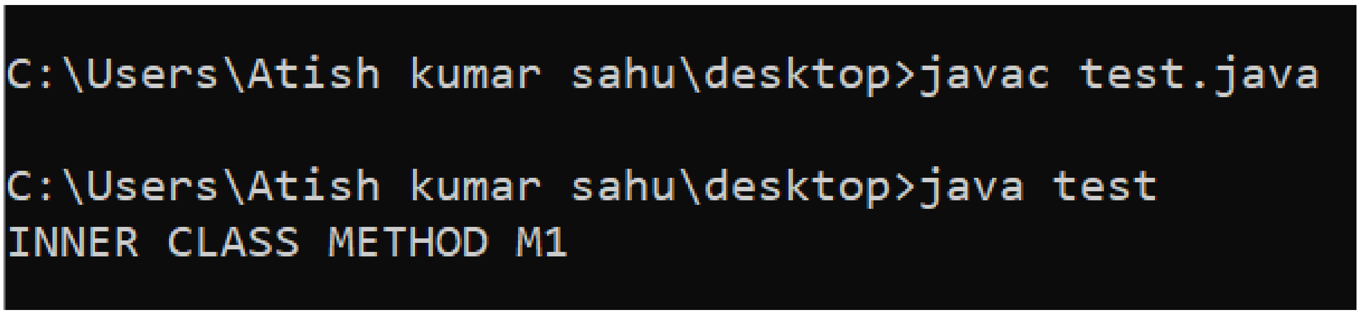
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java outer
inner class method 01
```

CASE 03:

ACCESSING INNER CLASS CODE FROM OUTSIDE OF OUTER CLASS.



```
class outer
{
	class inner
	{
		public void m1()
		{
			System.out.println("INNER CLASS METHOD M1");
		}
	}
}
class test
{
public static void main(String args[])
{
outer oo = new outer();
outer.inner oi = oo.new inner();
oi.m1();
}
}
```

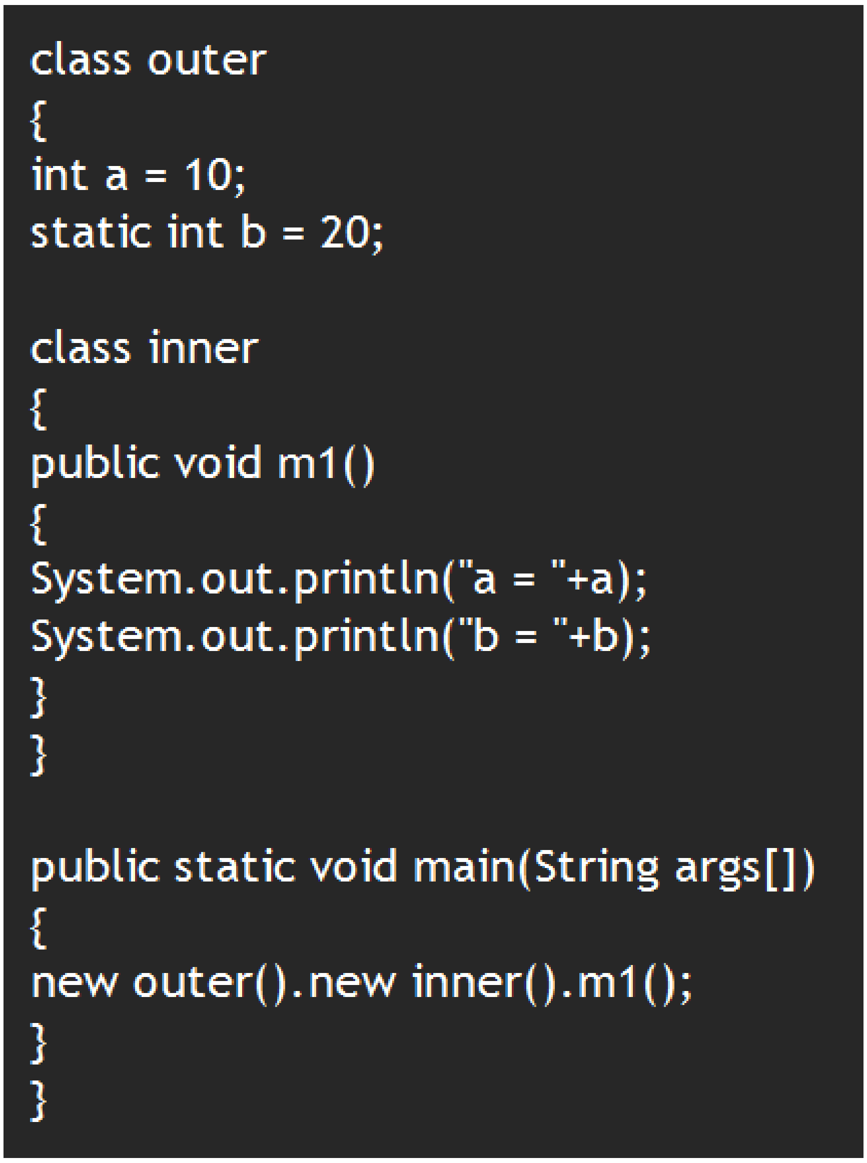


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
INNER CLASS METHOD M1
```

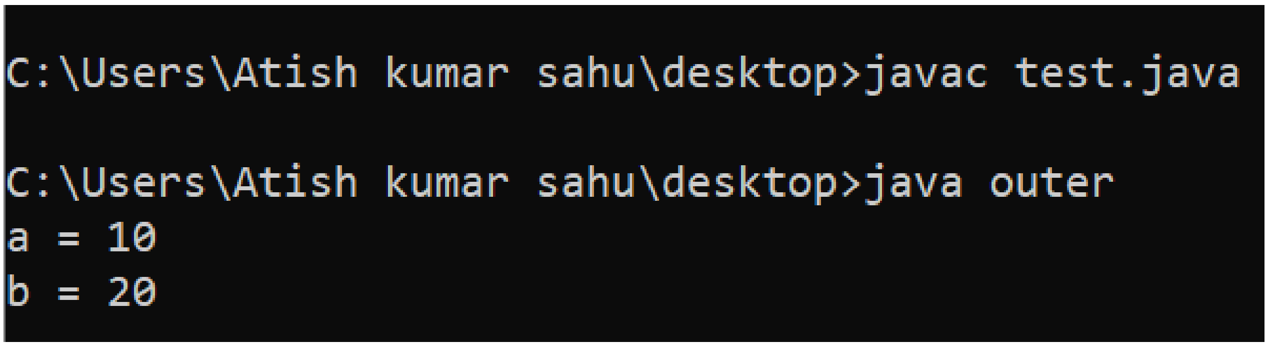
<u>ACCESSING INNER CLASS CODE:</u>

1. from static area of outer class or from outside of outer class the method is "outer o = new outer();" "outer.inner oi = o.new inner();"

2. from instance area of outer class the method is "inner I = new inner();" "I.m1();" after that you can call the insatnce area in main method and you can easily access inner class code.

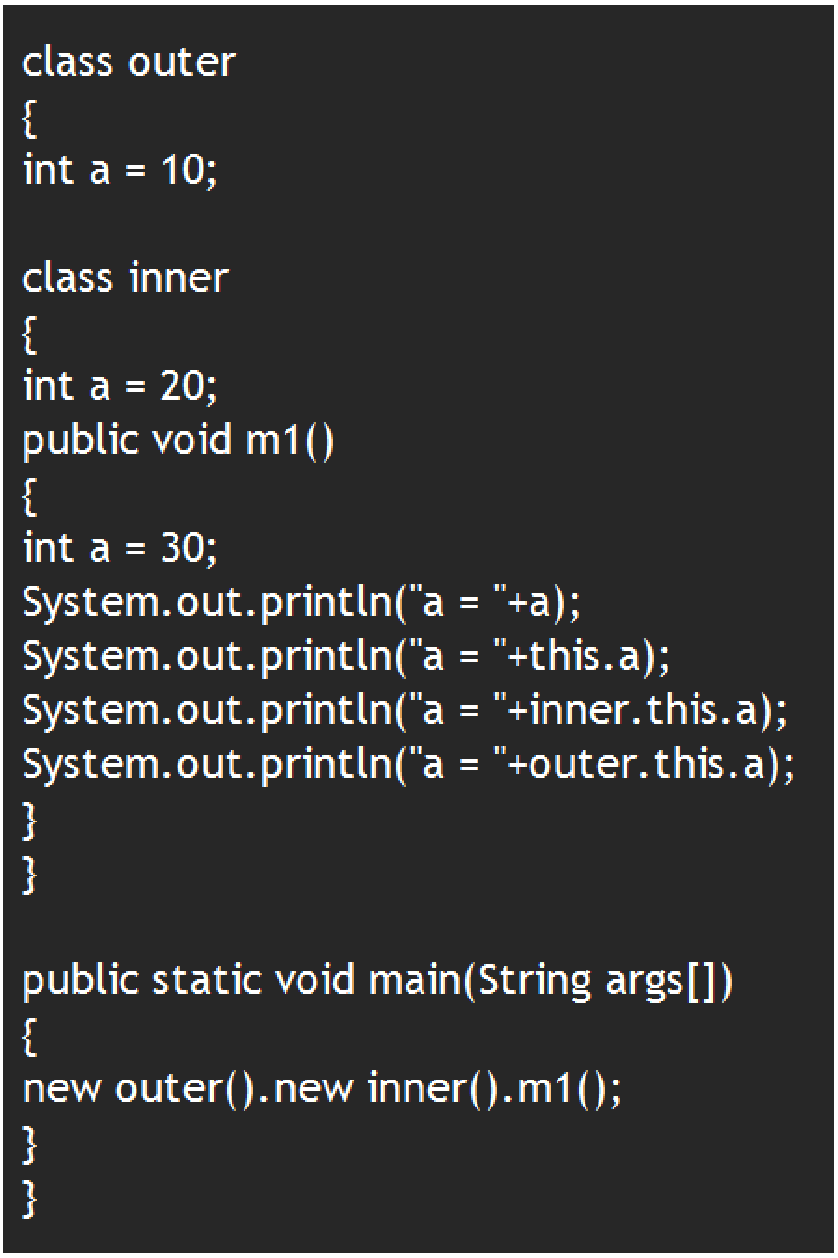


```
class outer
{
int a = 10;
static int b = 20;

class inner
{
public void m1()
{
System.out.println("a = "+a);
System.out.println("b = "+b);
}
}

public static void main(String args[])
{
new outer().new inner().m1();
}
}
```

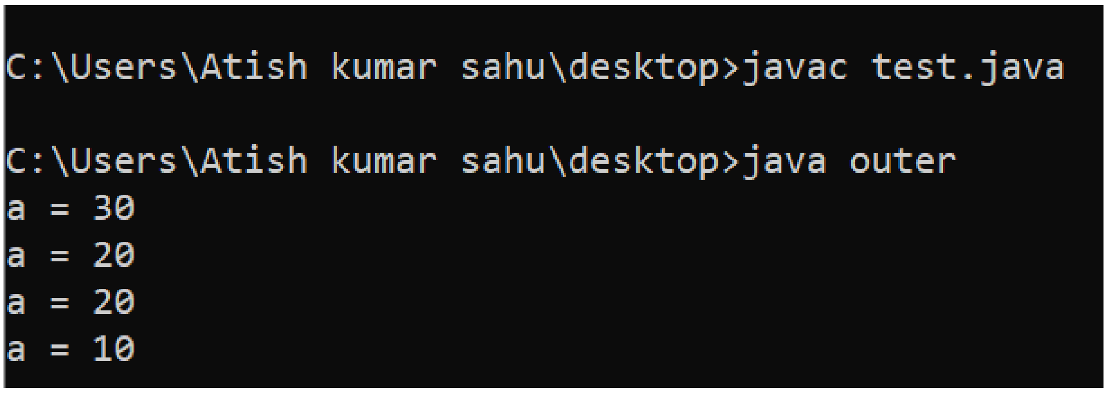


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java outer
a = 10
b = 20
```

From normal or regular inner class we can access both static and non-static members of outer class directly.



```
class outer
{
int a = 10;

class inner
{
int a = 20;
public void m1()
{
int a = 30;
System.out.println("a = "+a);
System.out.println("a = "+this.a);
System.out.println("a = "+inner.this.a);
System.out.println("a = "+outer.this.a);
}
}

public static void main(String args[])
{
new outer().new inner().m1();
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java outer
a = 30
a = 20
a = 20
a = 10
```

With in the inner class “this” always refers current inner class object. If we want to refer current outer class object we have to use “OuterClassName.this”.

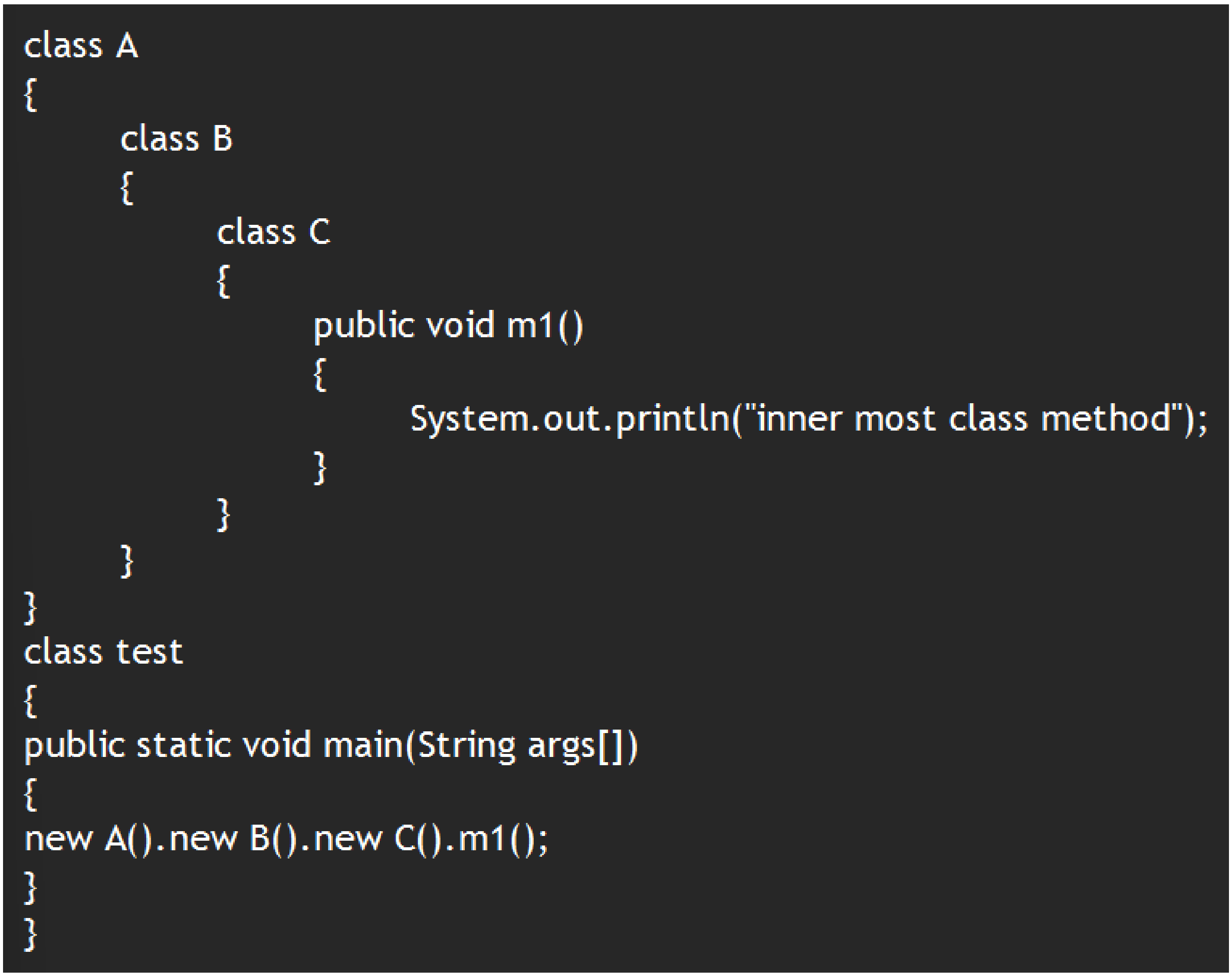
NOTE:

THE ONLY APPLICABLE MODIFIERS FOR OUTER CLASSES ARE “PUBLIC”, “DEFAULT”, “FINAL”, “ABSTRACT”, “STRICTFP”.

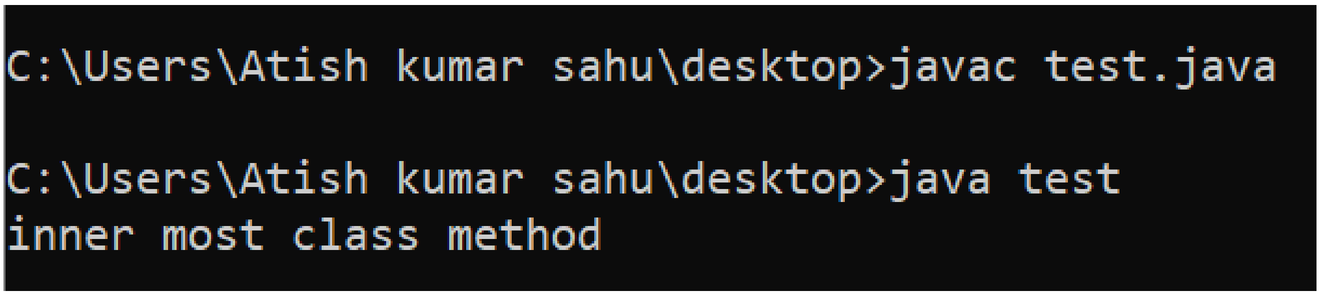
BUT FOR INNER CLASSES APPLICABLE MODIFIERS ARE “PUBLIC”, “DEFAULT”, “FINAL”, “ABSTRACT”, “STRICTFP”, “PRIVATE”, “PROTECTED”, “STATIC”.

NESTING OF INNER CLASSES:

Inside inner class we can declare another inner class that is nesting of inner classes is possible.



```
class A
{
	class B
	{
		class C
		{
			public void m1()
			{
				System.out.println("inner most class method");
			}
		}
	}
}
class test
{
public static void main(String args[])
{
new A().new B().new C().m1();
}
}
```



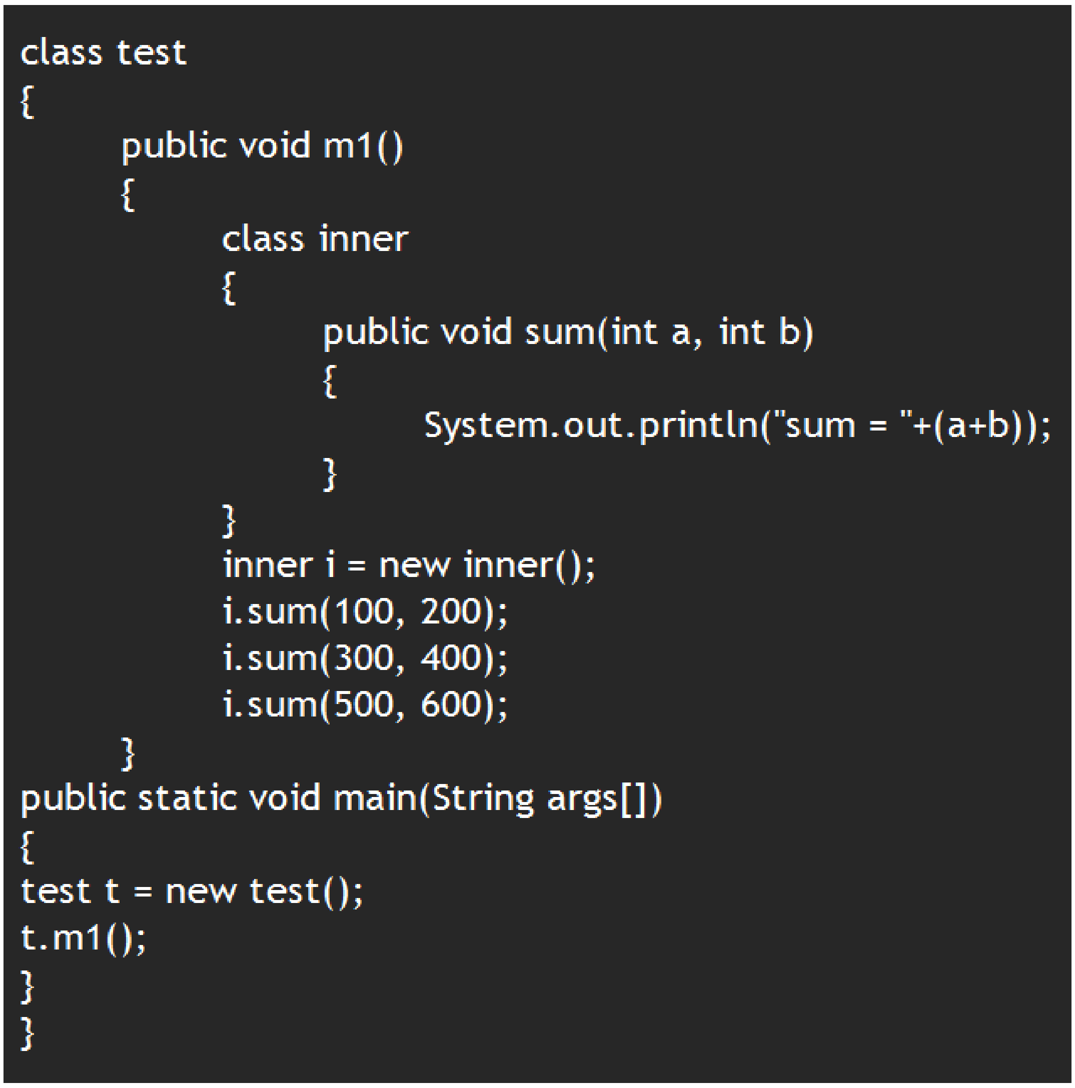
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
inner most class method
```

2. METHOD LOCAL INNER CLASSES:

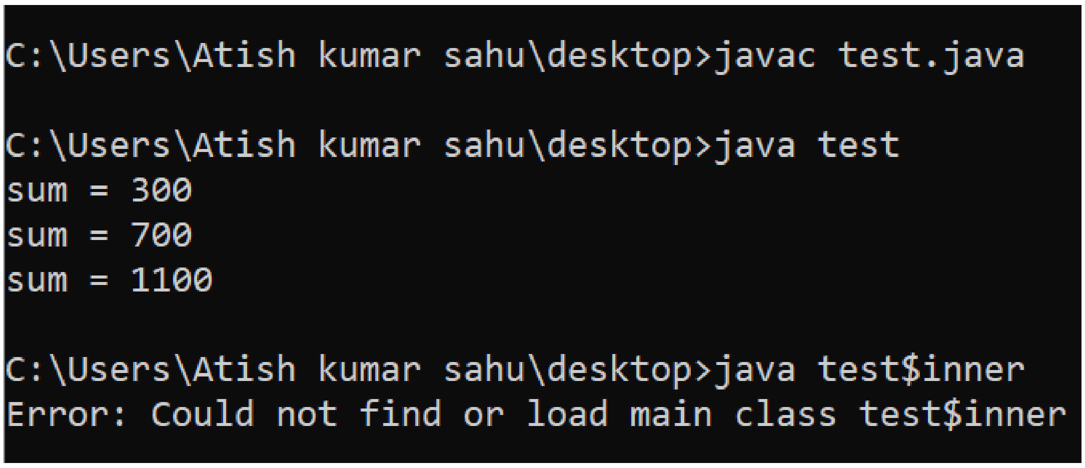
Sometimes we can declare a class inside a method, such type of inner classes are called method local inner classes.

The main purpose of method local inner class is to define method specific repeated required functionality. Method local inner classes are best suitable to meet nested method requirements.

We can access method local inner classes only with in the method where we declared outside of the method we can't access. Because of its less scope method local inner classes are most rarely used type of inner classes.



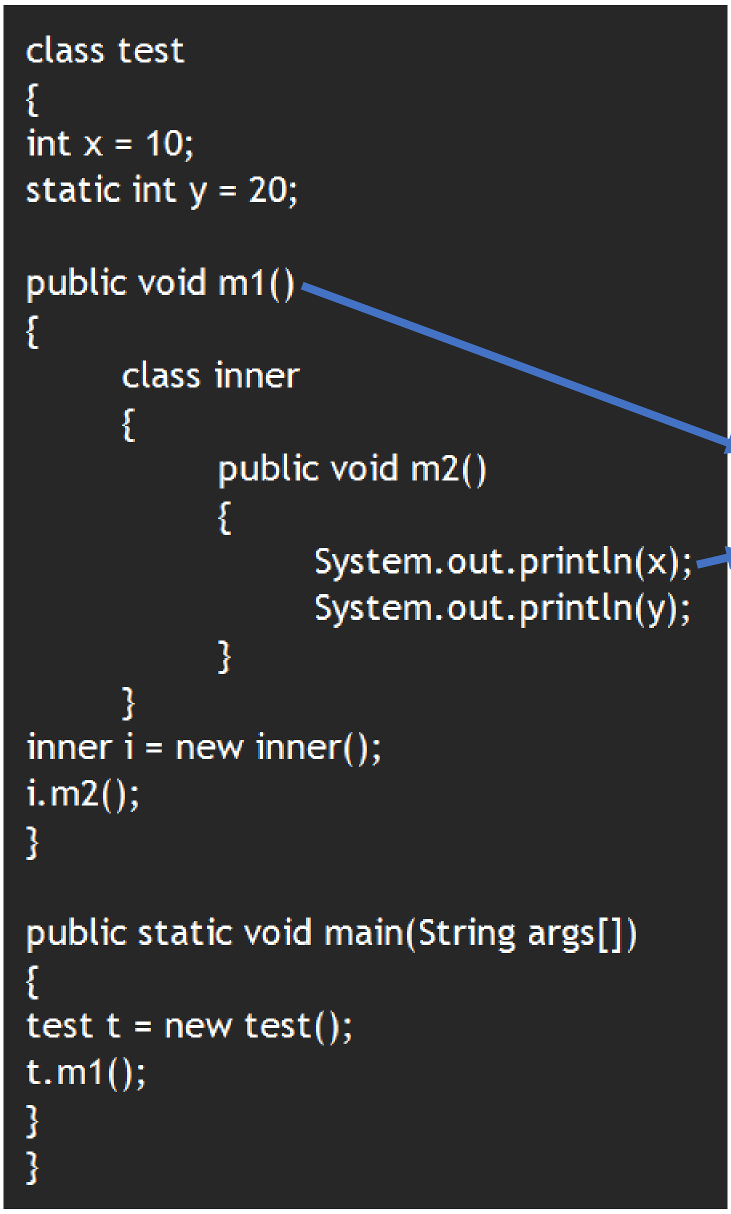
```
class test
{
        public void m1()
        {
                class inner
                {
                        public void sum(int a, int b)
                        {
                                System.out.println("sum = "+(a+b));
                        }
                }
                inner i = new inner();
                i.sum(100, 200);
                i.sum(300, 400);
                i.sum(500, 600);
        }
public static void main(String args[])
{
test t = new test();
t.m1();
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
sum = 300
sum = 700
sum = 1100

C:\Users\Atish kumar sahu\desktop>java test$inner
Error: Could not find or load main class test$inner
```

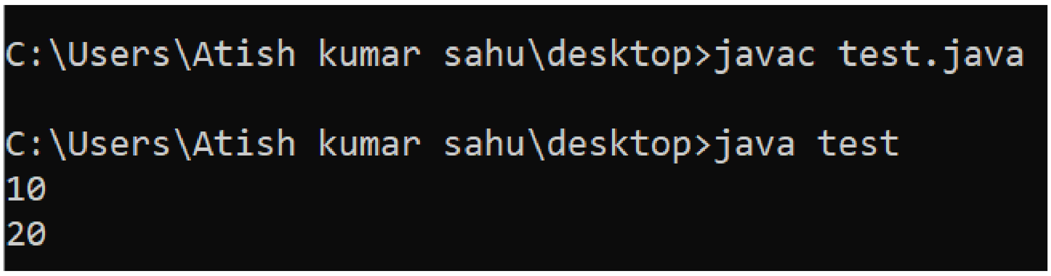
We can declare method local inner class inside both instance and static methods. If we declare inner class inside instance method then from that method local inner class we can access both static and non static members of outer class directly.

If we decalre inner class inside a static method then we can access only static member of outer class directly from that method local inner class.

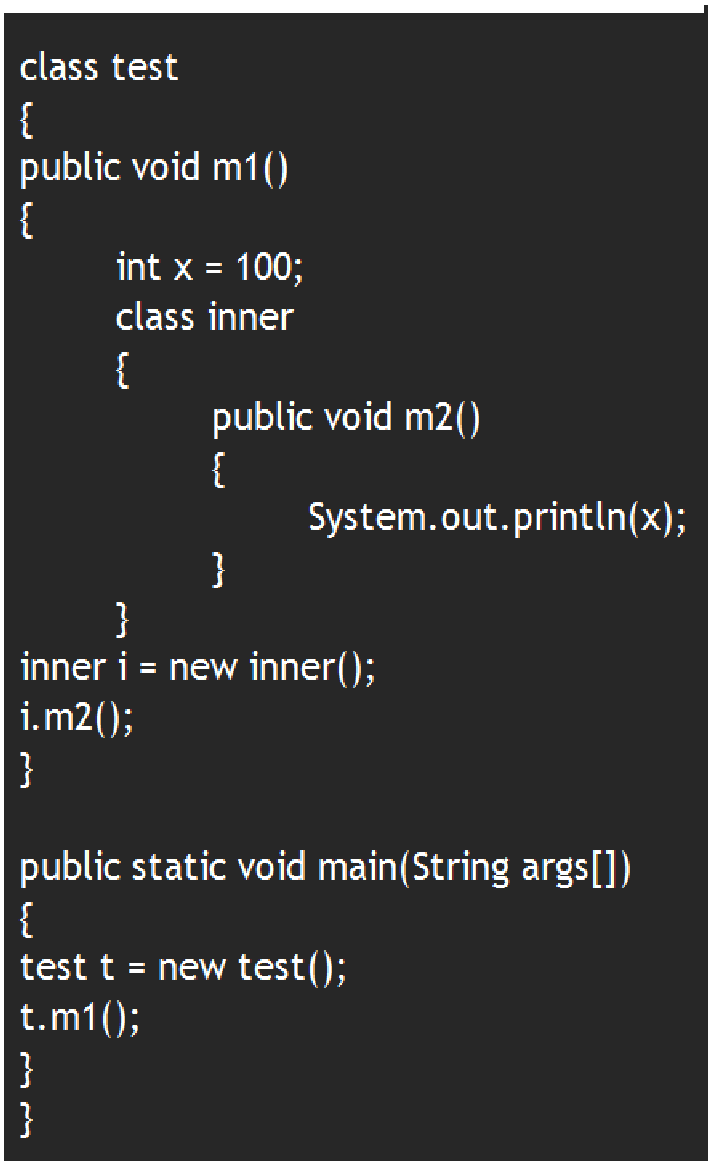


```
class test
{
int x = 10;
static int y = 20;

public void m1()
{
        class inner
        {
                public void m2()
                {
                        System.out.println(x);
                        System.out.println(y);
                }
        }
inner i = new inner();
i.m2();
}

public static void main(String args[])
{
test t = new test();
t.m1();
}
}
```

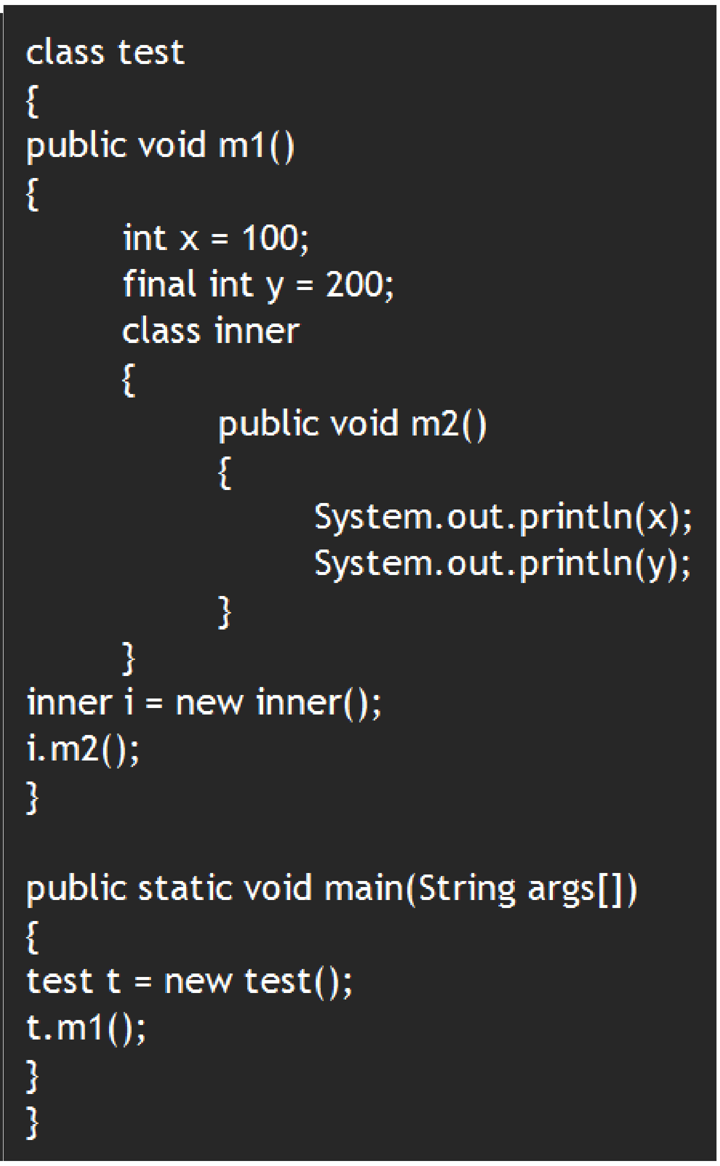
If we declare m1() method as static then "System.out.println(x);" we will get compile time error saying "<u>non-static variable x can't be referenced from a static context.</u>"



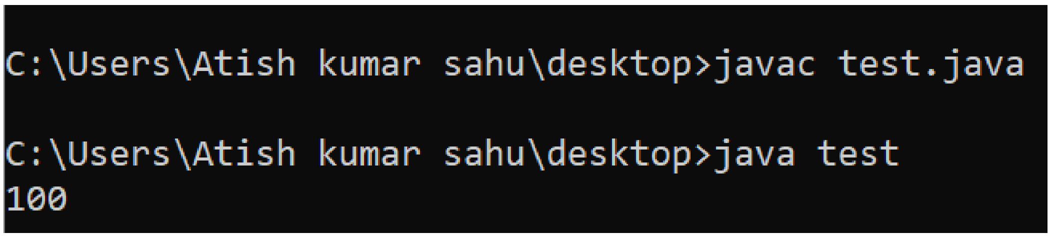
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
10
20
```



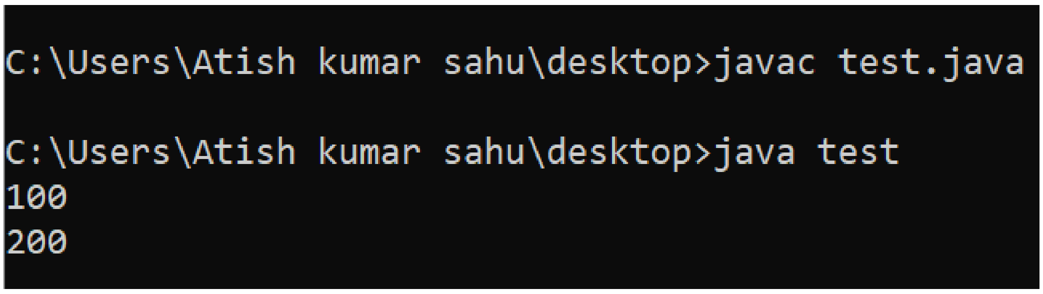
```
class test
{
public void m1()
{
        int x = 100;
        class inner
        {
                public void m2()
                {
                        System.out.println(x);
                }
        }
inner i = new inner();
i.m2();
}

public static void main(String args[])
{
test t = new test();
t.m1();
}
}
```



```
class test
{
public void m1()
{
        int x = 100;
        final int y = 200;
        class inner
        {
                public void m2()
                {
                        System.out.println(x);
                        System.out.println(y);
                }
        }
inner i = new inner();
i.m2();
}

public static void main(String args[])
{
test t = new test();
t.m1();
}
}
```



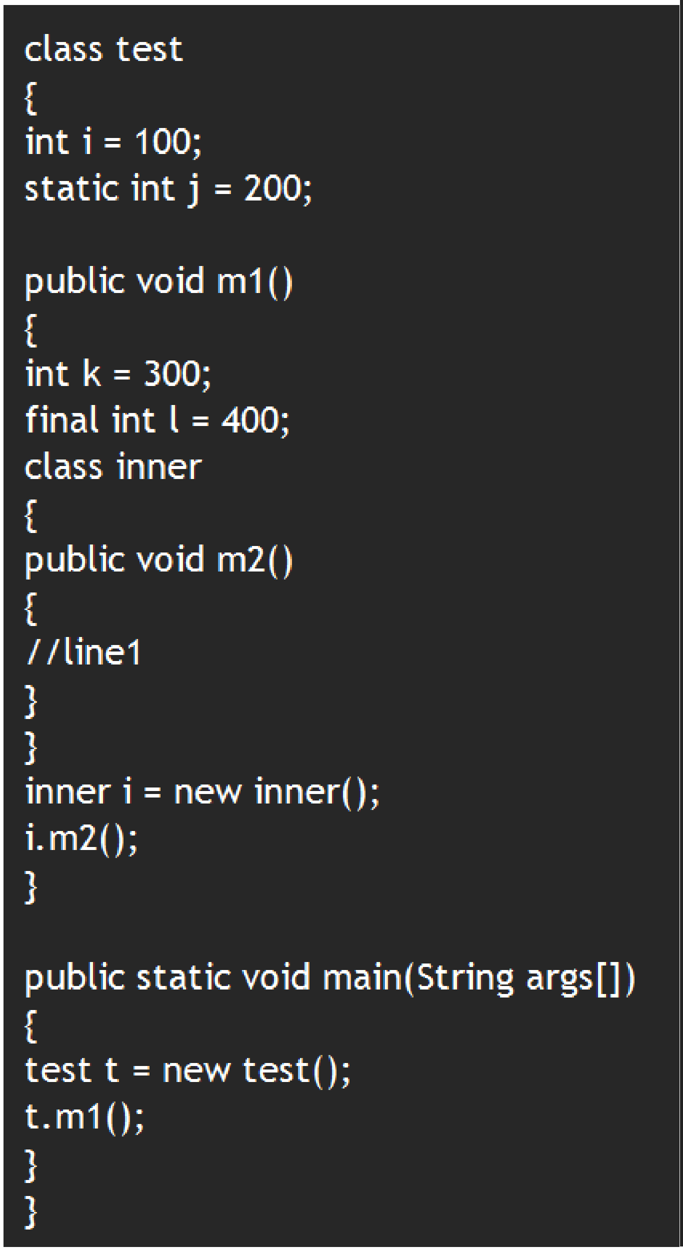
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
100
```



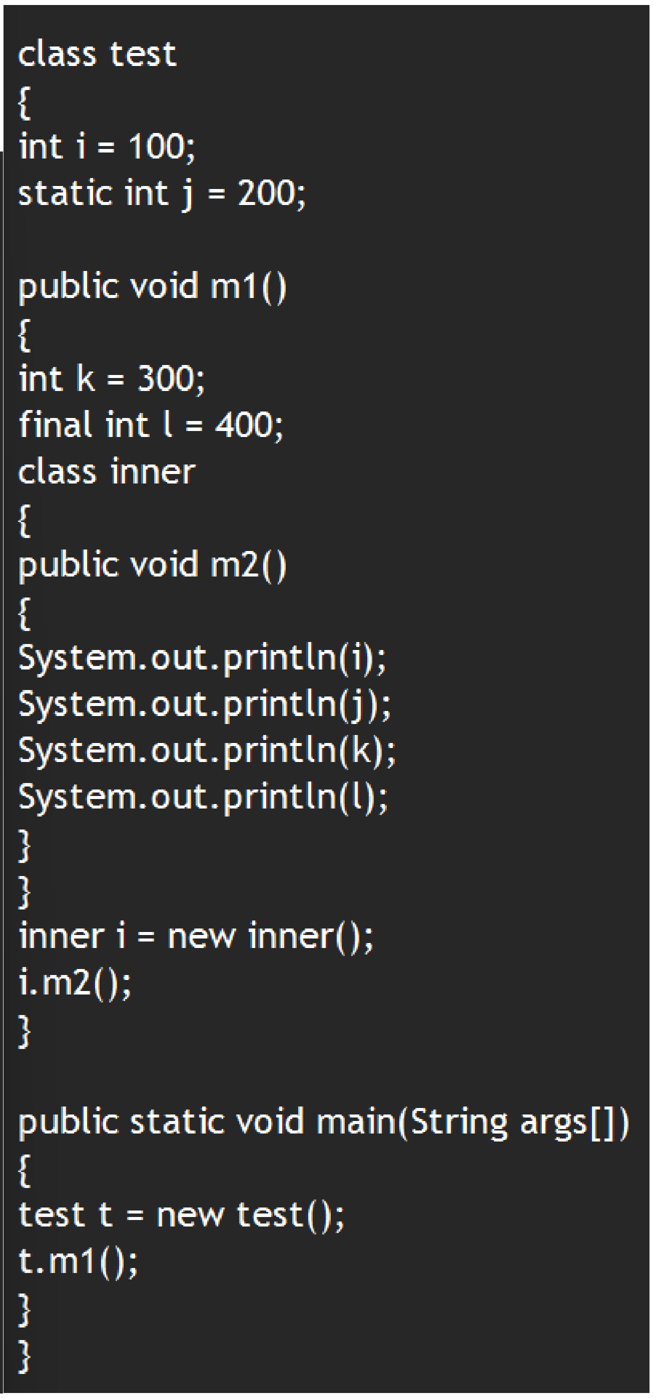
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
100
200
```

From method local inner class we can access local variables of method in which we declare inner class. if the local variable declared as final then we can access.

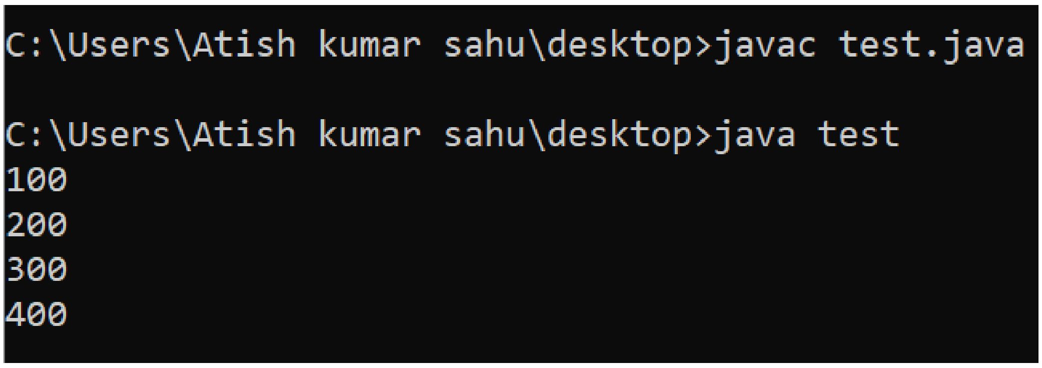
Q1. CONSIDER THE FOLLOWING CODE AT LINE 1 WHICH OF THE FOLLOWING VARIABLES WE CAN ACCESS DIRECTLY?



```java
class test
{
int i = 100;
static int j = 200;

public void m1()
{
int k = 300;
final int l = 400;
class inner
{
public void m2()
{
//line1
}
}
inner i = new inner();
i.m2();
}

public static void main(String args[])
{
test t = new test();
t.m1();
}
}
```

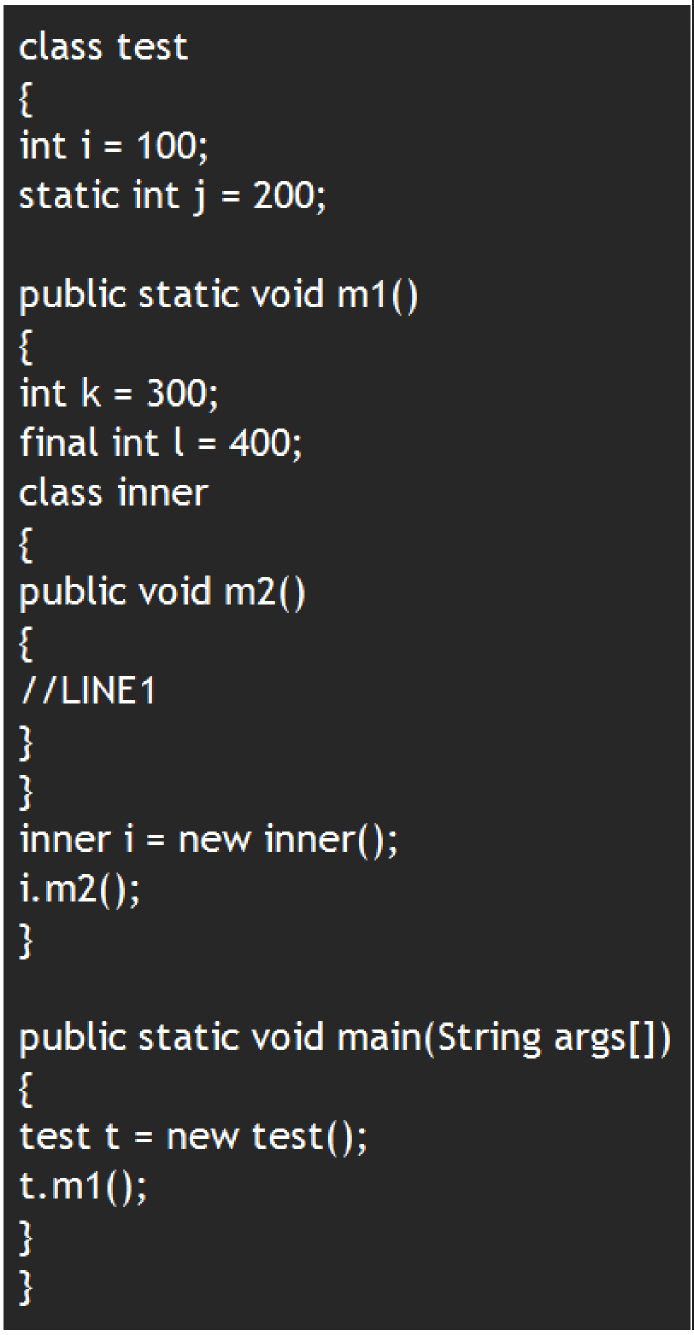


```java
class test
{
int i = 100;
static int j = 200;

public void m1()
{
int k = 300;
final int l = 400;
class inner
{
public void m2()
{
System.out.println(i);
System.out.println(j);
System.out.println(k);
System.out.println(l);
}
}
inner i = new inner();
i.m2();
}

public static void main(String args[])
{
test t = new test();
t.m1();
}
}
```

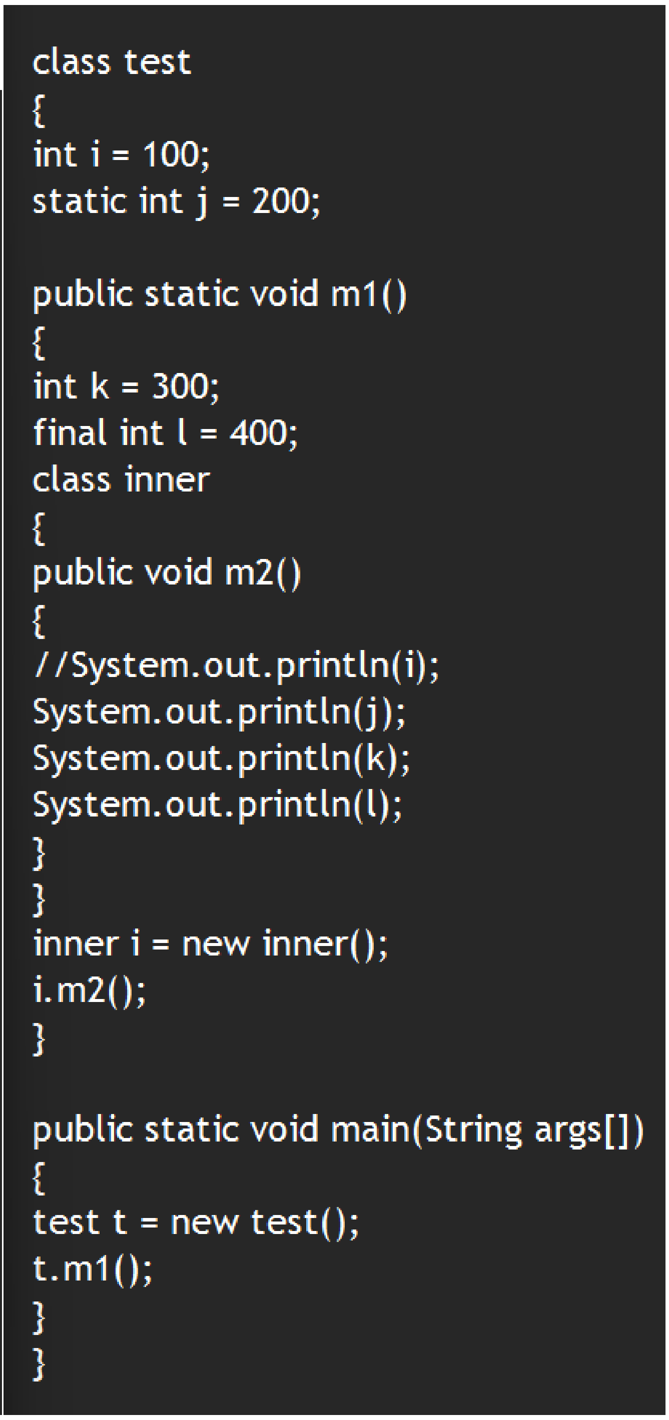


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
100
200
300
400
```

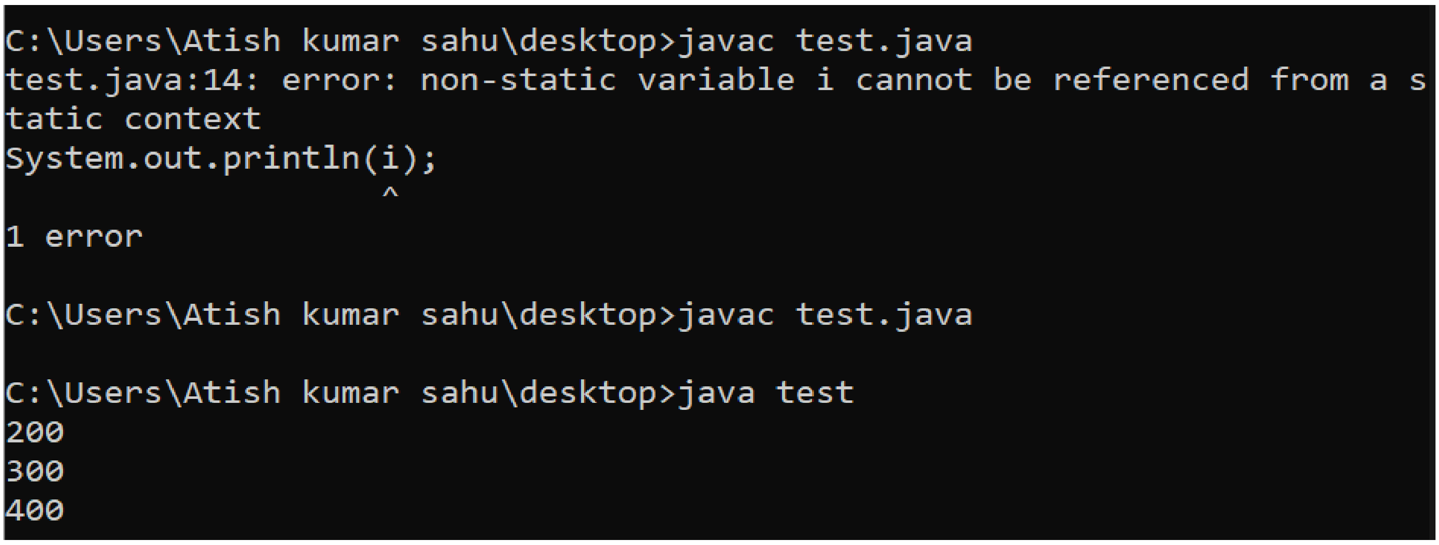
Q. IF WE DECLARE M1() METHOD AS STATIC THEN AT LINE 1 WHICH VARIABLES WE CAN ACCESS DIRECTLY?



```
class test
{
int i = 100;
static int j = 200;

public static void m1()
{
int k = 300;
final int l = 400;
class inner
{
public void m2()
{
//LINE1
}
}
inner i = new inner();
i.m2();
}

public static void main(String args[])
{
test t = new test();
t.m1();
}
}
```



```
class test
{
int i = 100;
static int j = 200;

public static void m1()
{
int k = 300;
final int l = 400;
class inner
{
public void m2()
{
//System.out.println(i);
System.out.println(j);
System.out.println(k);
System.out.println(l);
}
}
inner i = new inner();
i.m2();
}

public static void main(String args[])
{
test t = new test();
t.m1();
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:14: error: non-static variable i cannot be referenced from a s
tatic context
System.out.println(i);
                   ^
1 error

C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
200
300
400
```

Q. IF WE DECALRE M2() METHOD AS STATIC THEN WHAT WILL HAPPEN?

ANS: we will get compile time error because we can’t declare static members inside inner classes.

THE ONLY APPLICABLE MODIFIERS FOR METHOD LOCAL INNER CLASSES ARE “FINAL”, “ABSTRACT”, “STRICTFP”. IF WE ARE TRYING TO APPLY ANY OTHER MODIFIER THEN WE WILL GET COMPILE TIME ERROR.

3. ANONYMOUS INNER CLASS:

Sometimes we can declare inner class without name such type of inner classes are called anonymous inner class.

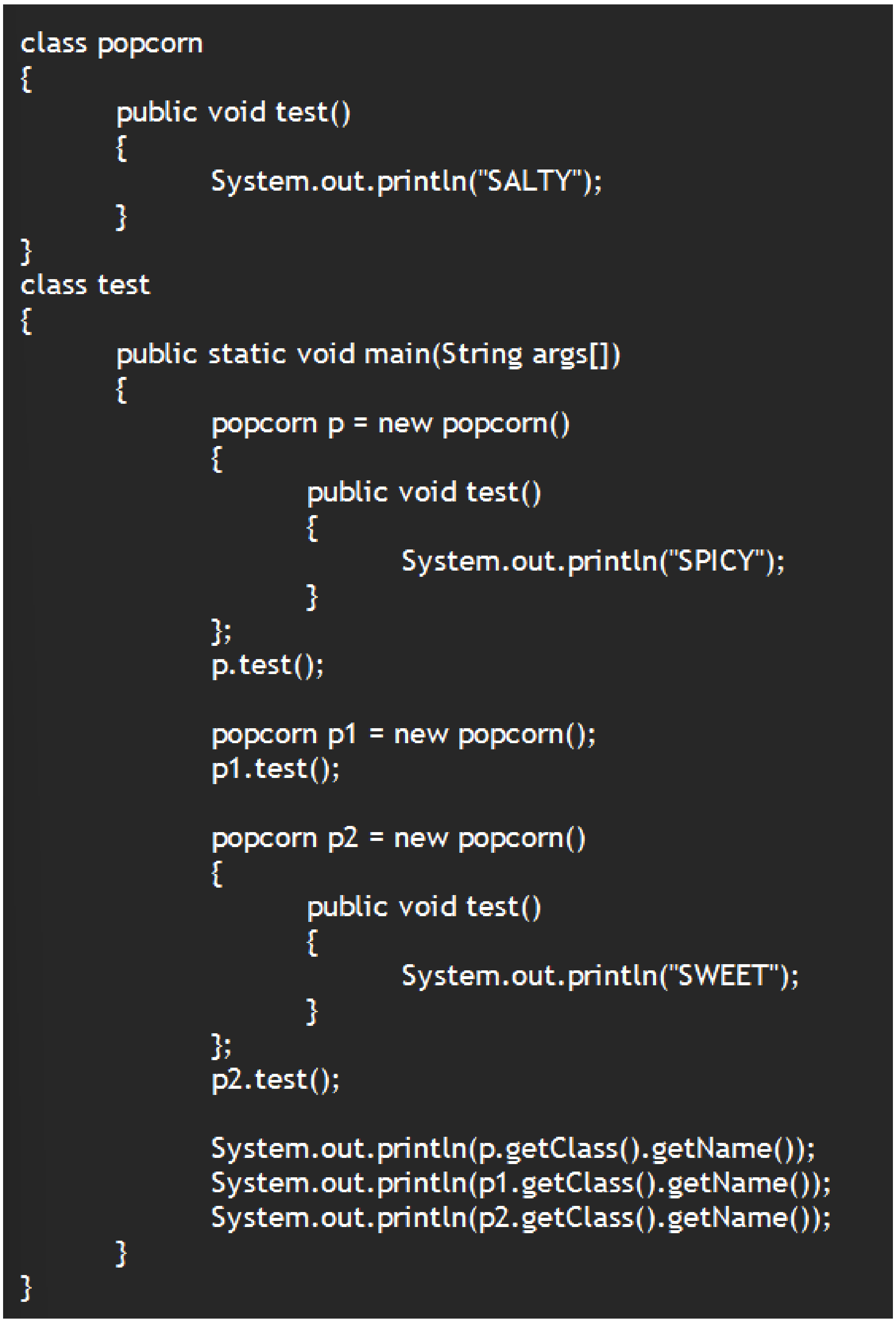
The main purpose of anonymous inner class is just for instant use(one time uses). There are three types of anonymous inner class are there.

1. ANONYMOUS INNER CLASS THAT EXTENDS A CLASS

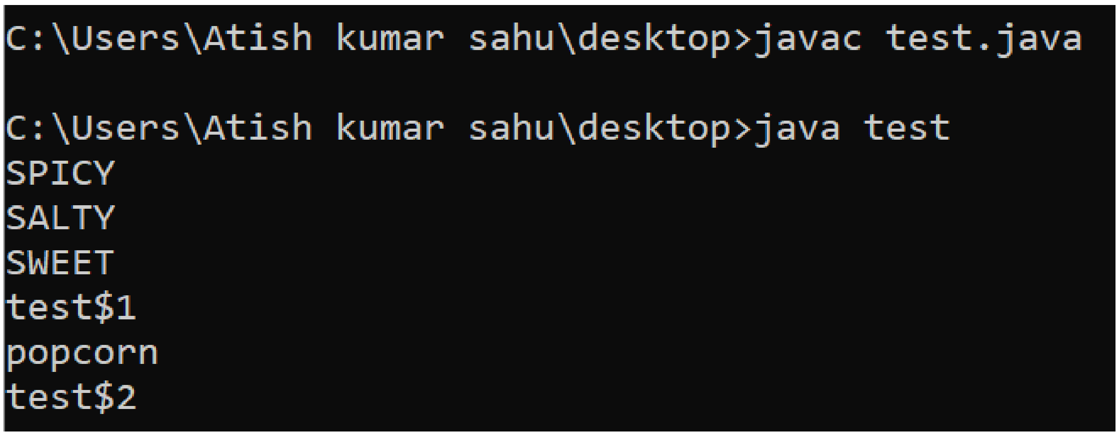
2. ANONYMOUS INNER CLASS THAT IMPLEMENTS INTERFACE

3. ANONYMOUS INNER CLASS THAT DEFINE INSIDE ARGUMENTS

1. ANONYMOUS INNER CLASS THAT EXTENDS A CLASS:



```
class popcorn
{
	public void test()
	{
		System.out.println("SALTY");
	}
}
class test
{
	public static void main(String args[])
	{
		popcorn p = new popcorn()
		{
			public void test()
			{
				System.out.println("SPICY");
			}
		};
		p.test();

		popcorn p1 = new popcorn();
		p1.test();

		popcorn p2 = new popcorn()
		{
			public void test()
			{
				System.out.println("SWEET");
			}
		};
		p2.test();

		System.out.println(p.getClass().getName());
		System.out.println(p1.getClass().getName());
		System.out.println(p2.getClass().getName());
	}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
SPICY
SALTY
SWEET
test$1
popcorn
test$2
```

1. “popcorn p = new popcorn();” in this statement we are creating popcorn object.

2. “popcorn p = new popcorn() { };” in this statement 2 activity happens

2.1 we are declaring a class that extends popcorn without name (anonymous inner class)

2.2 for that child class we are creating a object with parent reference.

3. “popcorn p = new popcorn() { public void test() { S.O.P(“SPICY”); } };” in this statement 3 activities are happens.

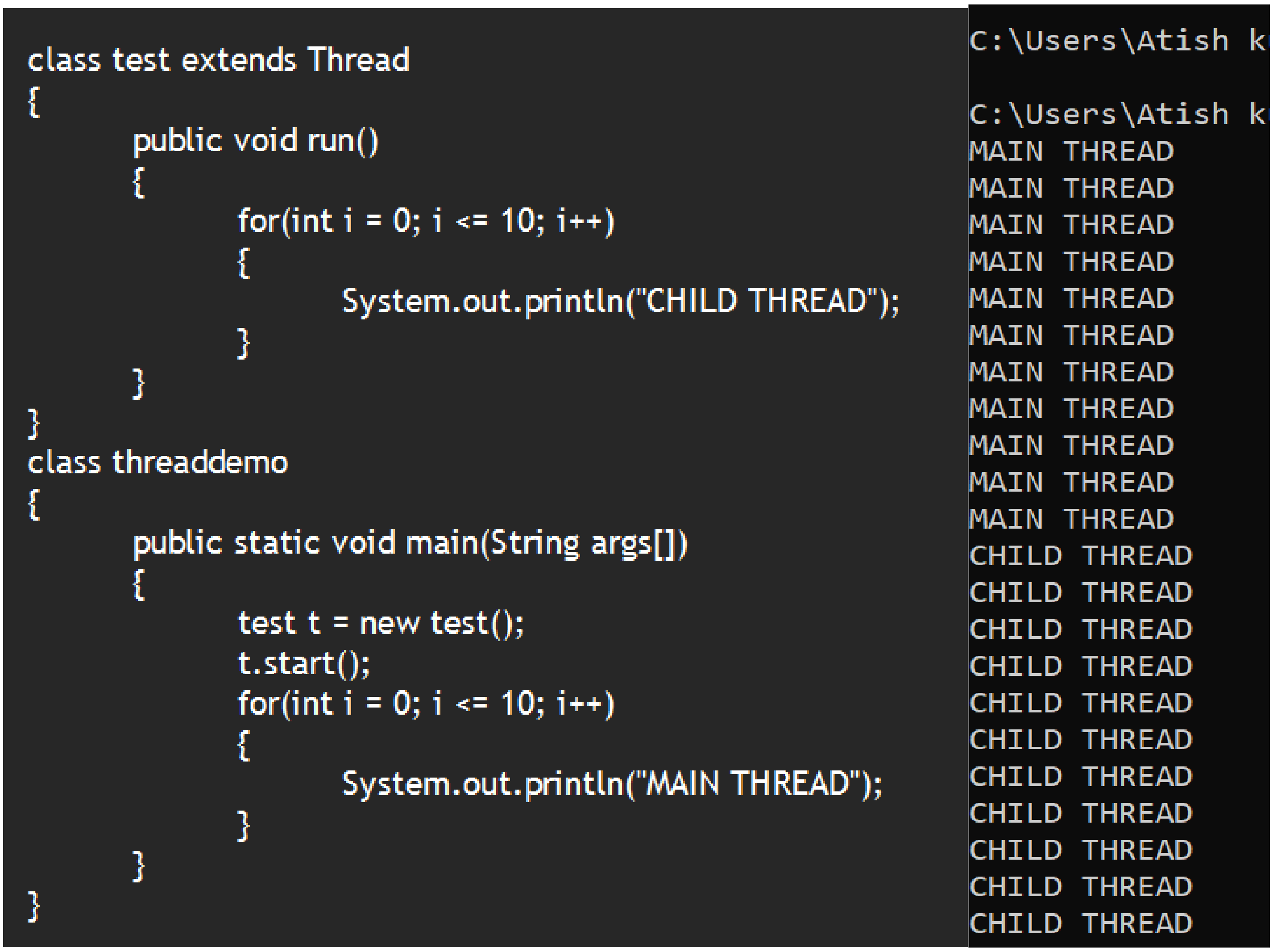
3.1 we are declaring a class that extends popcorn without name(anonymous inner class).

3.2 in that child class we are overridng test() method.

3.3 for that child class we are creating an object with parent reference.

<u>DEFINING A THREAD BY EXTENDING THREAD CLASS:</u>

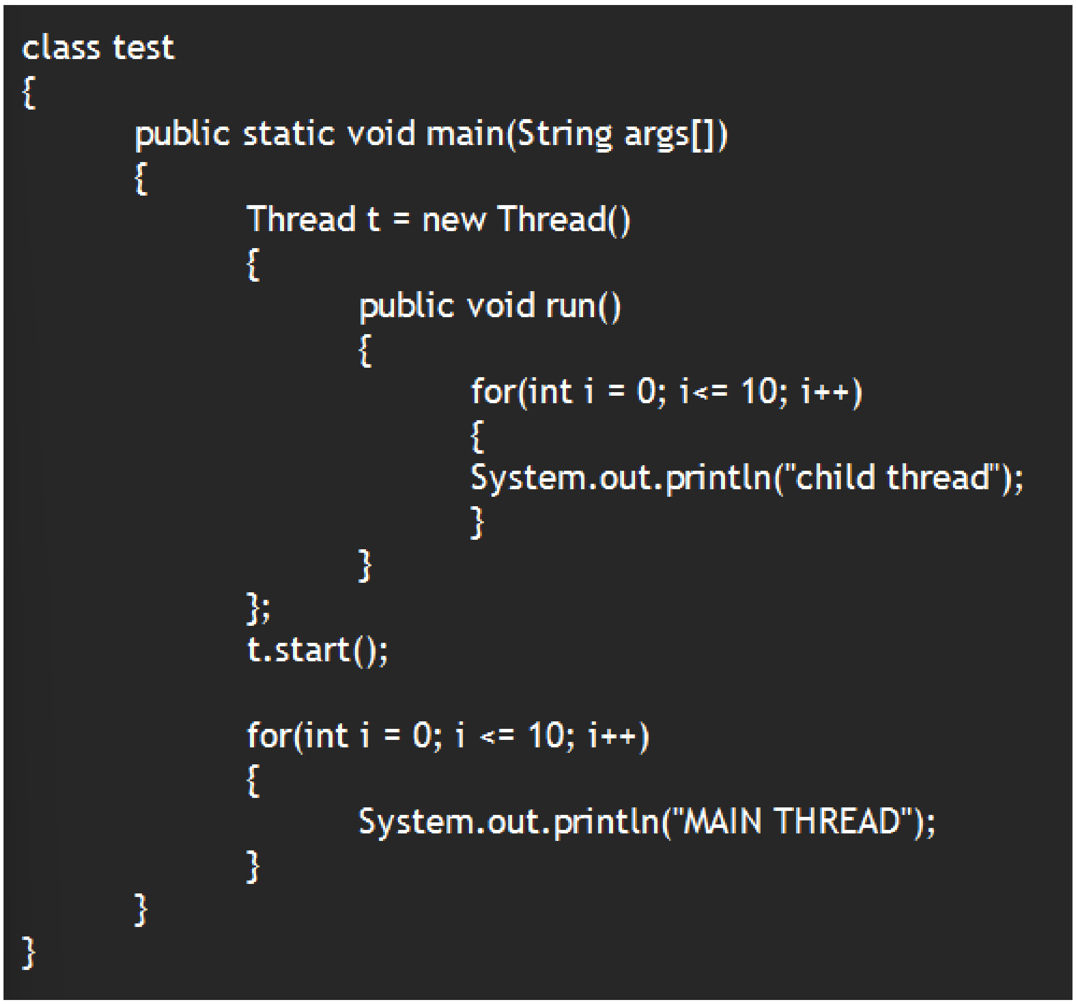
<u>NORMAL METHOD:</u>



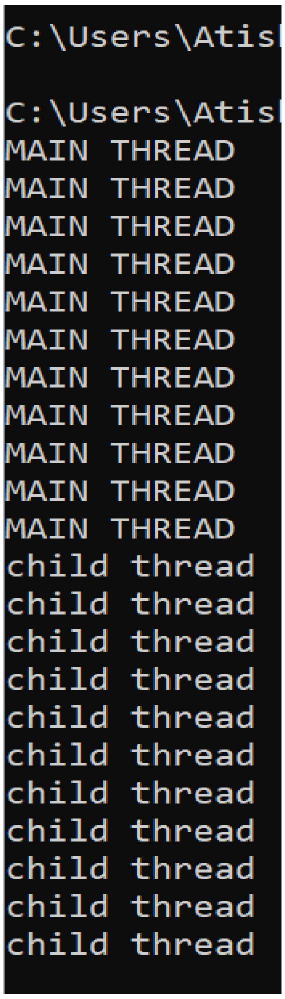
```
class test extends Thread
{
	public void run()
	{
		for(int i = 0; i <= 10; i++)
		{
			System.out.println("CHILD THREAD");
		}
	}
}
class threaddemo
{
	public static void main(String args[])
	{
		test t = new test();
		t.start();
		for(int i = 0; i <= 10; i++)
		{
			System.out.println("MAIN THREAD");
		}
	}
}
```

```
C:\Users\Atish k

C:\Users\Atish k
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
CHILD THREAD
CHILD THREAD
CHILD THREAD
CHILD THREAD
CHILD THREAD
CHILD THREAD
CHILD THREAD
CHILD THREAD
CHILD THREAD
CHILD THREAD
CHILD THREAD
```

ANONYMOUS INNER CLASS APPROACH:



```
class test
{
	public static void main(String args[])
	{
		Thread t = new Thread()
		{
			public void run()
			{
				for(int i = 0; i<= 10; i++)
				{
				System.out.println("child thread");
				}
			}
		};
		t.start();

		for(int i = 0; i <= 10; i++)
		{
			System.out.println("MAIN THREAD");
		}
	}
}
```

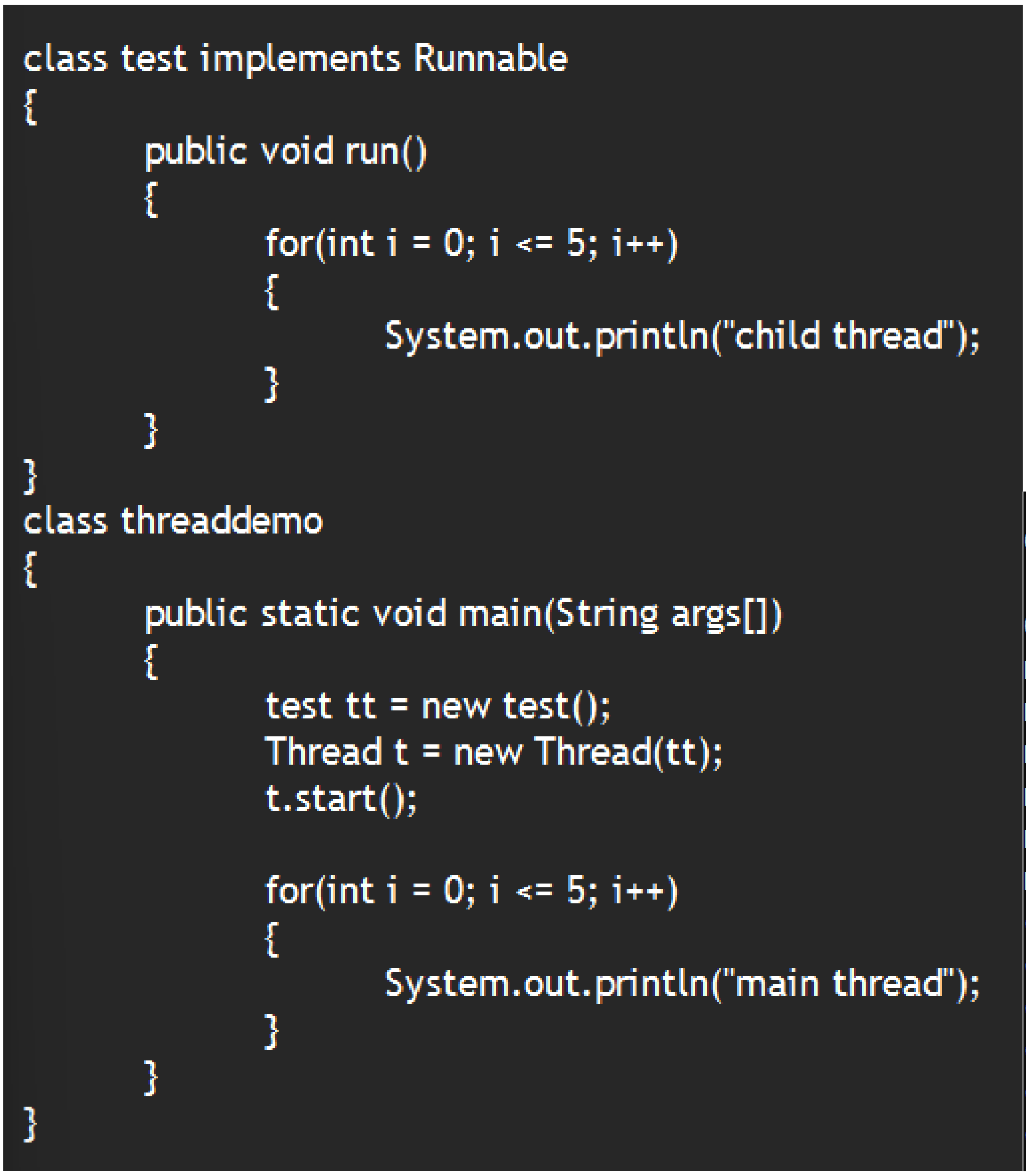


```
C:\Users\Atis

C:\Users\Atis
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
MAIN THREAD
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
```

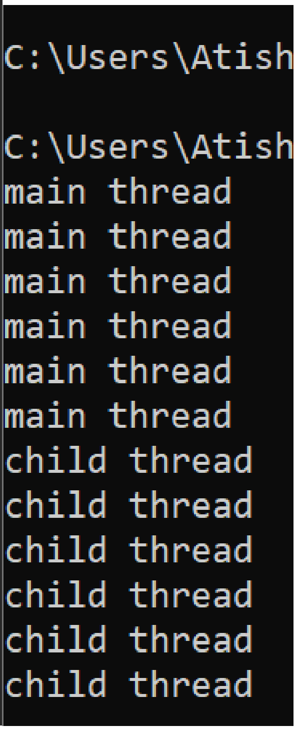
ANONYMOUS INNER CLASS THAT IMPLEMENTS AN INTERFACE:

DEFINING A THREAD BY IMPLEMENTING RUNNABLE INTERFACE:

NORMAL METHOD:

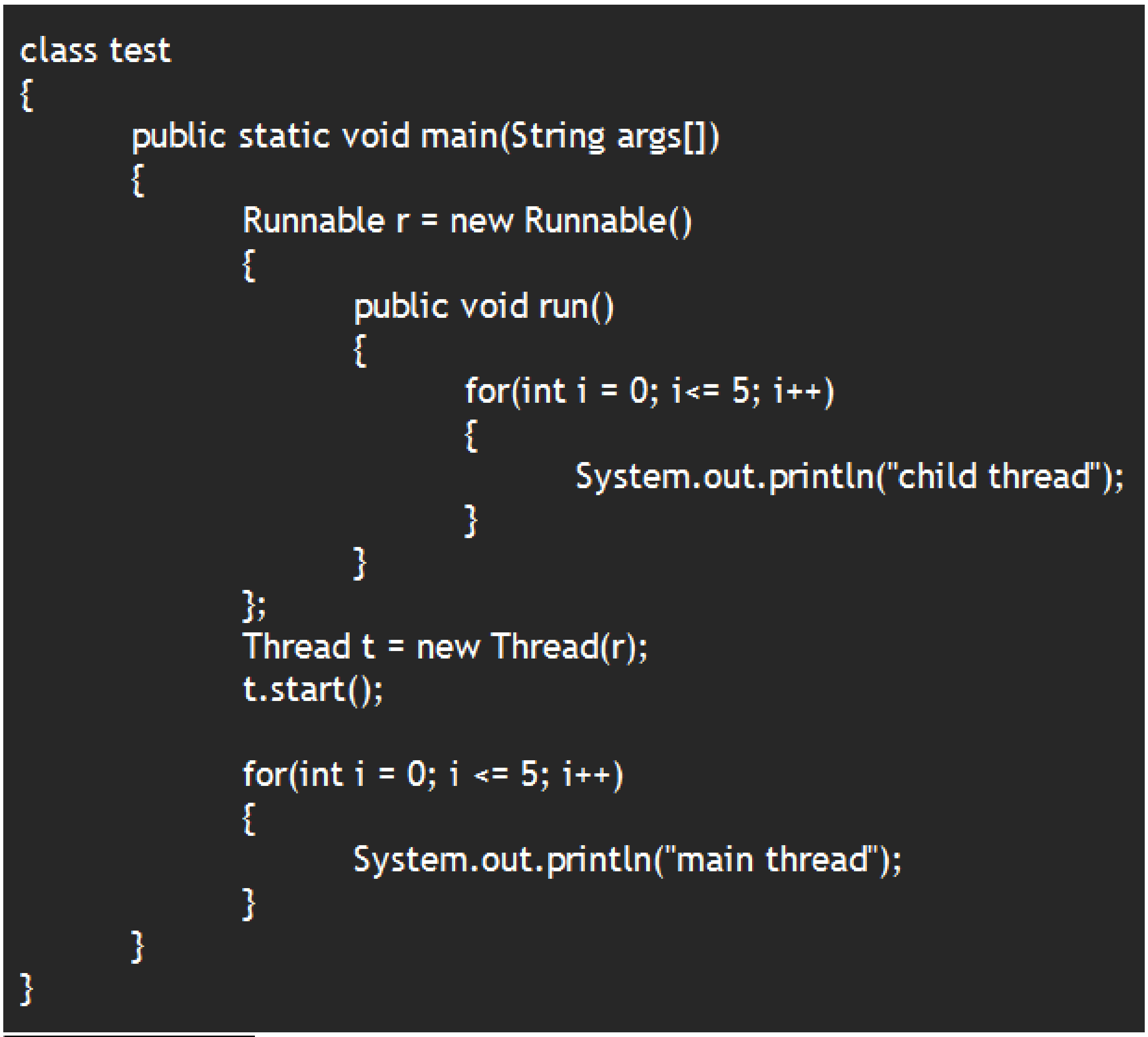


```
class test implements Runnable
{
	public void run()
	{
		for(int i = 0; i <= 5; i++)
		{
			System.out.println("child thread");
		}
	}
}
class threaddemo
{
	public static void main(String args[])
	{
		test tt = new test();
		Thread t = new Thread(tt);
		t.start();

		for(int i = 0; i <= 5; i++)
		{
			System.out.println("main thread");
		}
	}
}
```

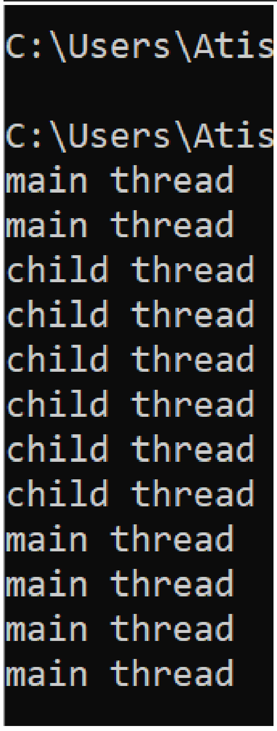


```
C:\Users\Atish

C:\Users\Atish
main thread
main thread
main thread
main thread
main thread
main thread
child thread
child thread
child thread
child thread
child thread
child thread
```

ANONYMOUS INNER CLASS METHOD:

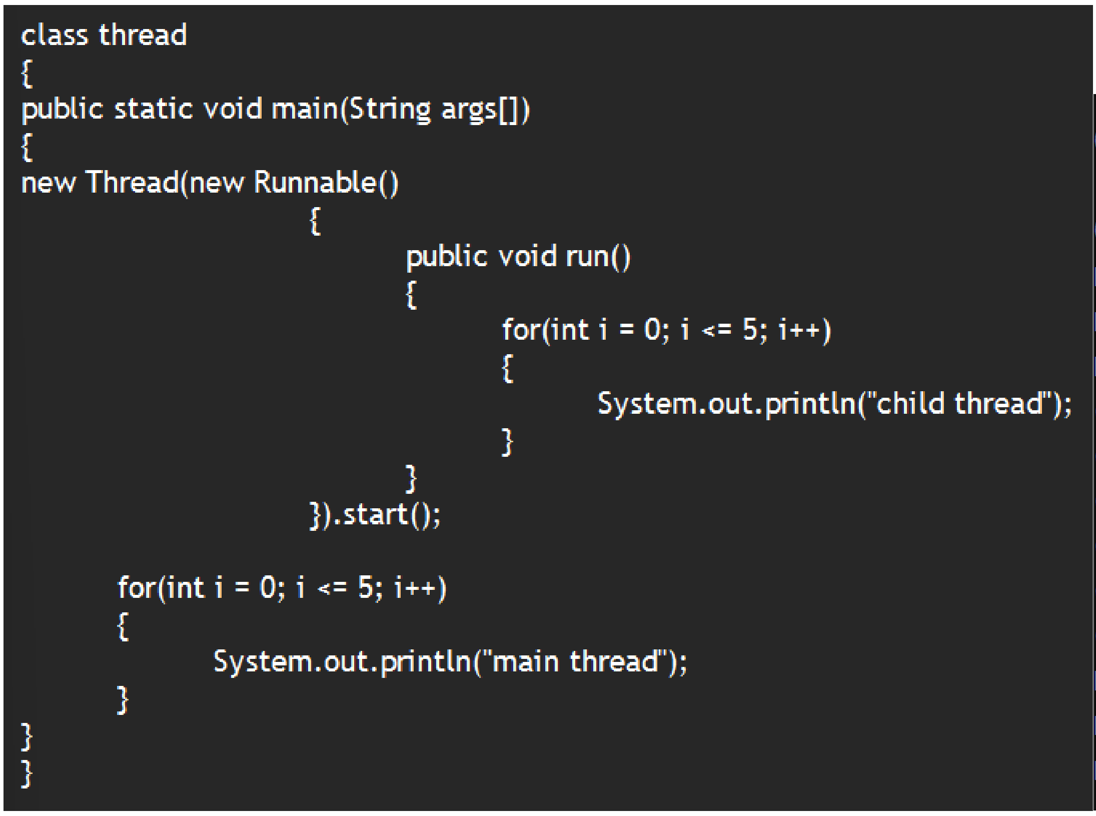


```
class test
{
        public static void main(String args[])
        {
                Runnable r = new Runnable()
                {
                        public void run()
                        {
                                for(int i = 0; i<= 5; i++)
                                {
                                        System.out.println("child thread");
                                }
                        }
                };
                Thread t = new Thread(r);
                t.start();

                for(int i = 0; i <= 5; i++)
                {
                        System.out.println("main thread");
                }
        }
}
```

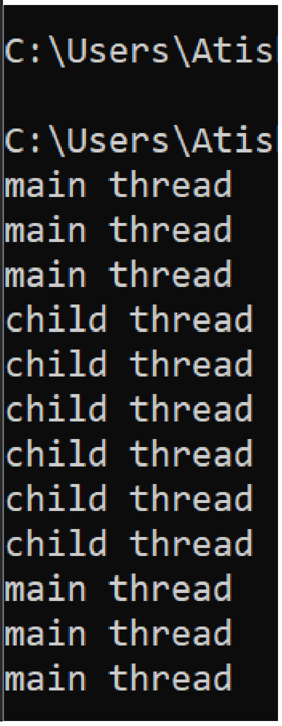


```
C:\Users\Atis

C:\Users\Atis
main thread
main thread
child thread
child thread
child thread
child thread
child thread
child thread
main thread
main thread
main thread
main thread
```

ANONYMOUS INNER CLASS THAT DEFINE INSIDE ARGUMENTS:



```
class thread
{
public static void main(String args[])
{
new Thread(new Runnable()
                        {
                                public void run()
                                {
                                        for(int i = 0; i <= 5; i++)
                                        {
                                                System.out.println("child thread");
                                        }
                                }
                        }).start();

        for(int i = 0; i <= 5; i++)
        {
                System.out.println("main thread");
        }
}
}
```



```
C:\Users\Atis

C:\Users\Atis
main thread
main thread
main thread
child thread
child thread
child thread
child thread
child thread
child thread
main thread
main thread
main thread
```

NORMAL JAVA CLASS VS ANONYMOUS INNER CLASS:

1. A normal java class can extends only one class at a time of course anonymous inner class also can extends only one class at a time.

2. A normal java class can implement any number of interfaces simultaneously. In case of anonymous inner class can implements only one interface at a time.

3. A normal java class can extends a class and can implements any number of interfaces simultaneously. In case of anonymous inner class can extend a class or can implement an interface but not both simultaneously.

4. In normal java class we can write any number of constructors simultaneously. In case of anonymous inner classes we can't write any constructor explicitly(because the name of class and name of the constructor must be same but anonymous inner class not have nay name).

5. If the requirement is standard and required severla times then we should go for normal top level class. if the requirement is temporary and required only once (instant use) then we shuld go for anonymous inner class.

Q. WHERE ANONYMOUS INNER CLASSES ARE BEST SUITABLE?

ANS: we can use anonymous inner classes freequently in GUI based application to implement event handling.

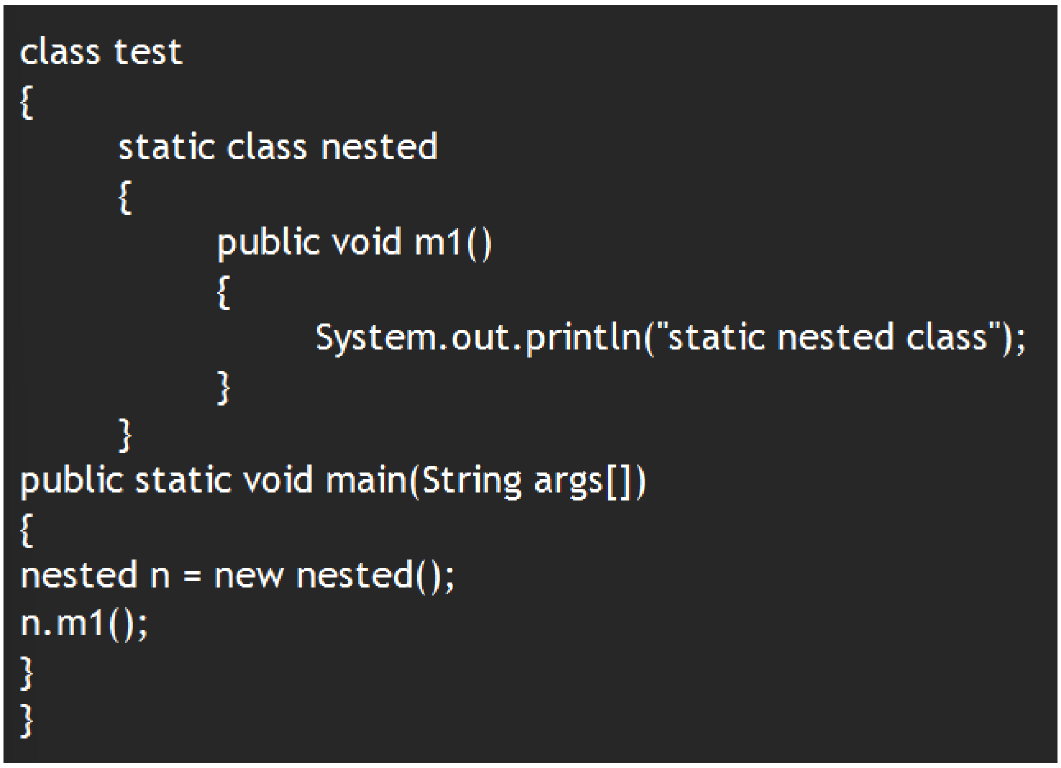
4. STATIC NESTED CLASSES:

Sometimes we can inner class with static modifiers such type of inner classes are called statiic nested classes. In case of normal or regular inner class without existing outer class object there is no cha ce of existing inner class object that is inner class object is strongly associated with outer class object.

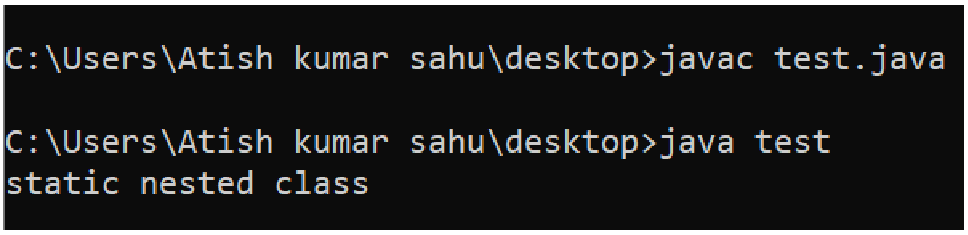
But in the case of static nested classes without existing outer class object there may be a chance of existing nested class object hence static nested class object is not strongly associated with outer class object.

If we want to create nested class object from outside of the outer class then we can create as folllows

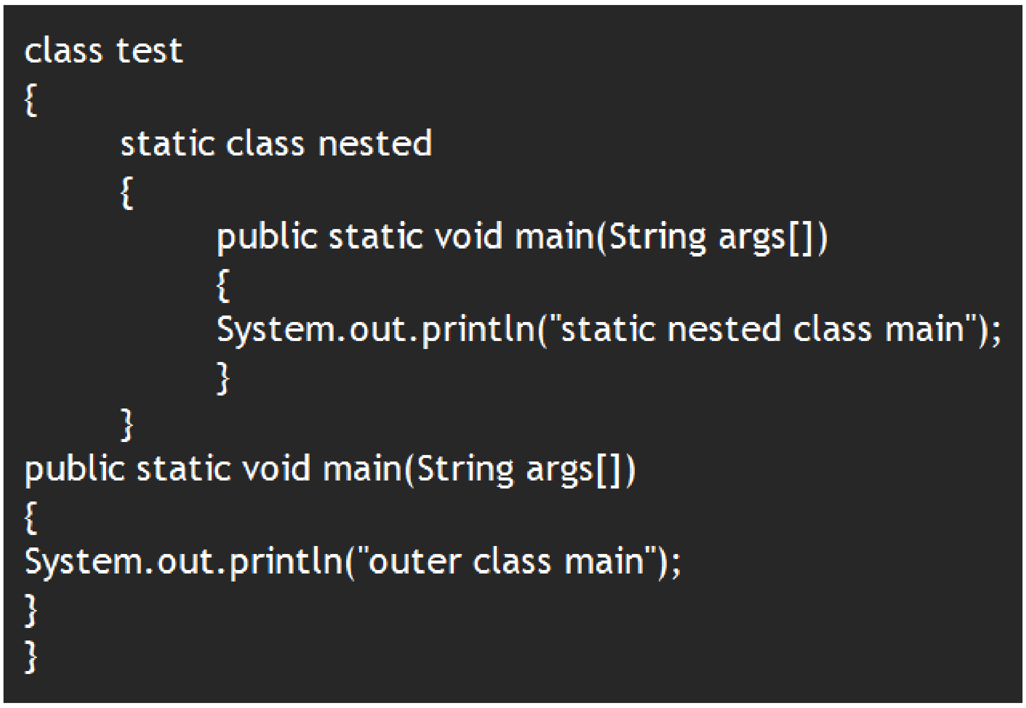
“test.nested n = new test.nested();”



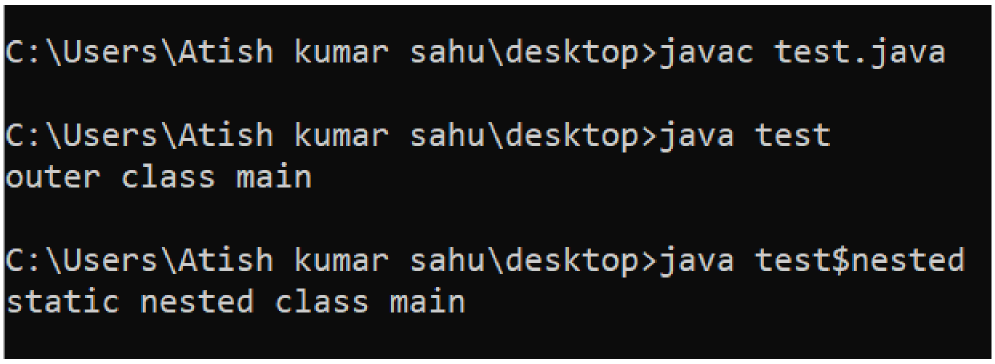
```
class test
{
        static class nested
        {
                public void m1()
                {
                        System.out.println("static nested class");
                }
        }
public static void main(String args[])
{
nested n = new nested();
n.m1();
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
static nested class
```



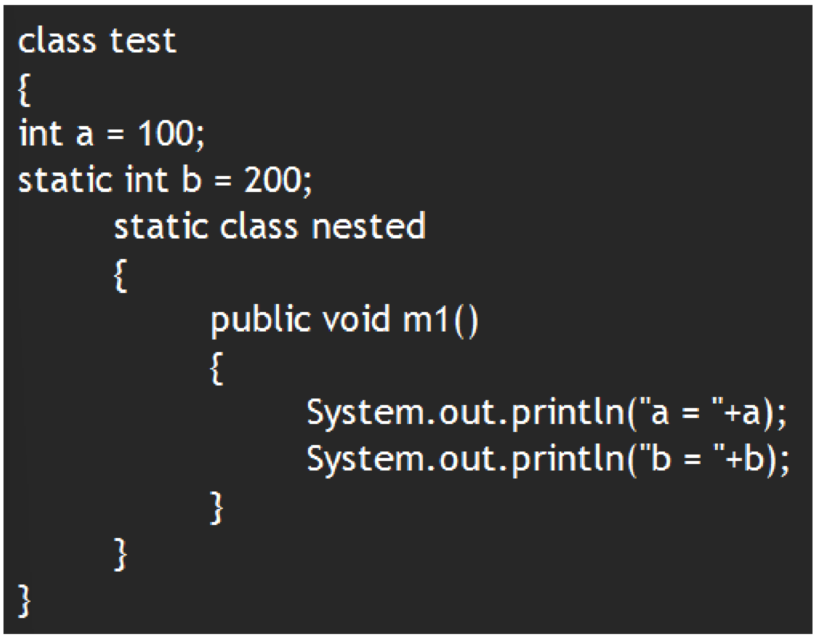
```
class test
{
        static class nested
        {
                public static void main(String args[])
                {
                System.out.println("static nested class main");
                }
        }
public static void main(String args[])
{
System.out.println("outer class main");
}
}
```



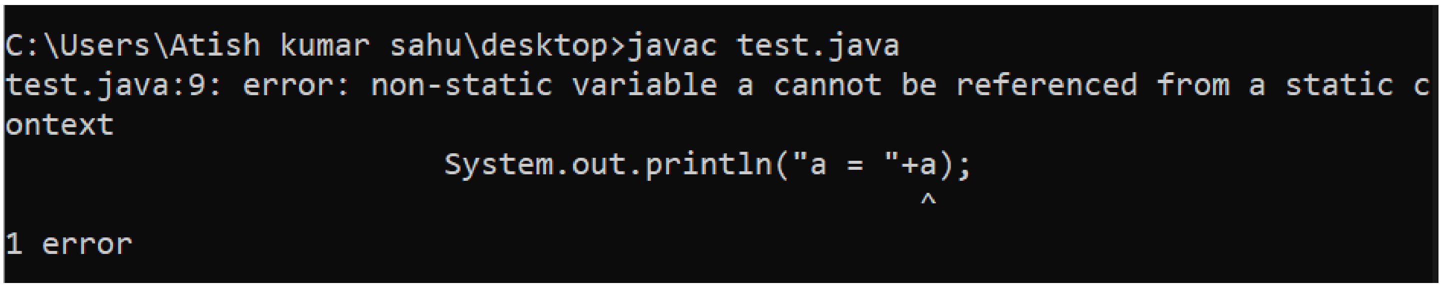
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
outer class main

C:\Users\Atish kumar sahu\desktop>java test$nested
static nested class main
```

In normal or regular inner class we can't declare any static members. But in static nested classes we can declare static members including main method. Hence we can invoke static nested class directly from command prompt.

From normal or regular inner classes we can access both static and non-static members of outer class directly. But from static nested classes we can access static members of outer class directly and we can't access non-static members.



```
class test
{
int a = 100;
static int b = 200;
	static class nested
	{
		public void m1()
		{
			System.out.println("a = "+a);
			System.out.println("b = "+b);
		}
	}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:9: error: non-static variable a cannot be referenced from a static c
ontext
                        System.out.println("a = "+a);
                                                  ^
1 error
```

NORMAL OR REGULAR INNER CLASS VS STATIC NESTED CLASS:

1. without existing outer class object there is no chance of existing inner class object that is inner class object is strongly associated with outer class object.

Without existing outer class object there may be a chance of existing static nested class object. Hence static nested class object is not strongly not associated with outer class object.

2. in normal or regular inner classes we can't declare static members. In static nested classes we can declare static members.

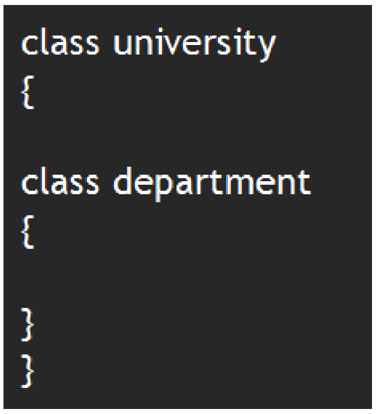
3. in normal or regular inner class we can't declare main method. And hence we can't invoke inner class directly from command prompt. In static nested classes we can declare main method and hence we can invoke nested class directly from command prompt.

4. from normal or regular inner classes we can access both static and non-static members of outer class directly. From static nested classes we can access only static members of outer class.

VARIOUS COMBINATIONS OF NESTED CLASSES AND INTERFACES:

CASE 01: CLASS INSIDE A CLASS:

Without existing one type of object if there is no chance of existing another type of object then we can declare a class inside a class. example: university consist of severla departments. Without existing university there is no chance of existing department. Hence we have to declare department class inside university class.



```
class university
{

class department
{

}
}
```

CASE 02: INTERFACE INSIDE A CLASS:

Inside a class if we required multiple implementation of interface and all these implementation are related to a particular class then we can define interface inside a class.

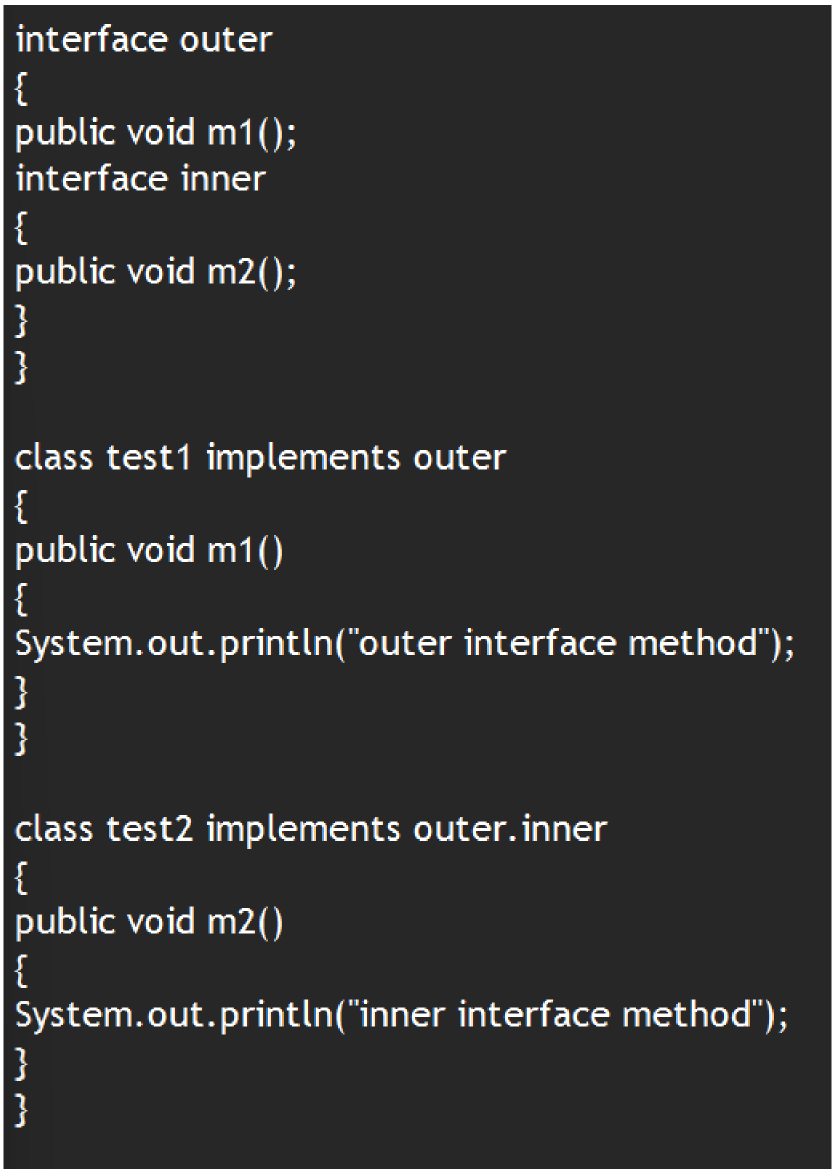
```
class test
{

interface vehicle
{
public int getWheel();
}
class bus implements vehicle
{
public int getWheel()
{
return 6;
}
}
class auto implements vehicle
{
public int getWheel()
{
return 3;
}
}

}
```

CASE 03: INTERFACE INSIDE INTERFACE:

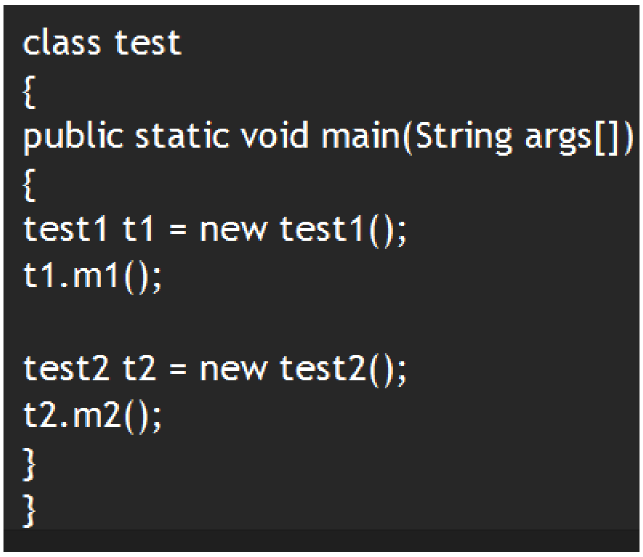
We can declare interface inside interface. Example: a map is a group of key value pairs and each key value pair is called an entry. Without existing map object there is no chance of existing entry object. Hence interface entry is defined inside map interface.

```
interface map
{
interface entry
{
}
}
```

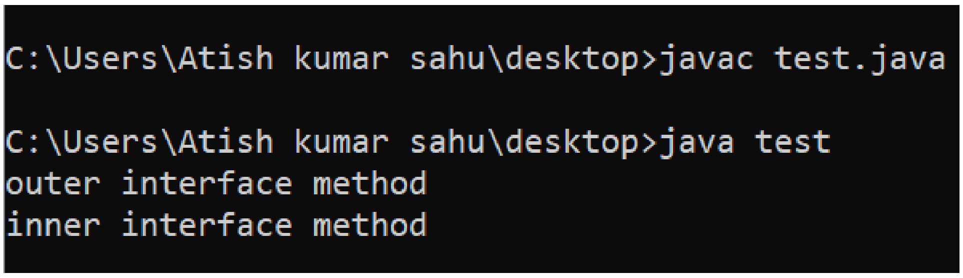
Every interface present inside interface is always public and static whether we are declaring or not. Hence we can implement inner interface directly without implementing outer interface. Similarly whenever we are implementing outer interface we are not required to implement inner interface. It means we can implement outer and inner interfaces independently.



```
interface outer
{
public void m1();
interface inner
{
public void m2();
}
}

class test1 implements outer
{
public void m1()
{
System.out.println("outer interface method");
}
}

class test2 implements outer.inner
{
public void m2()
{
System.out.println("inner interface method");
}
}
```



```
class test
{
public static void main(String args[])
{
test1 t1 = new test1();
t1.m1();

test2 t2 = new test2();
t2.m2();
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
outer interface method
inner interface method
```

CASE 04: CLASS INSIDE INTERFACE:

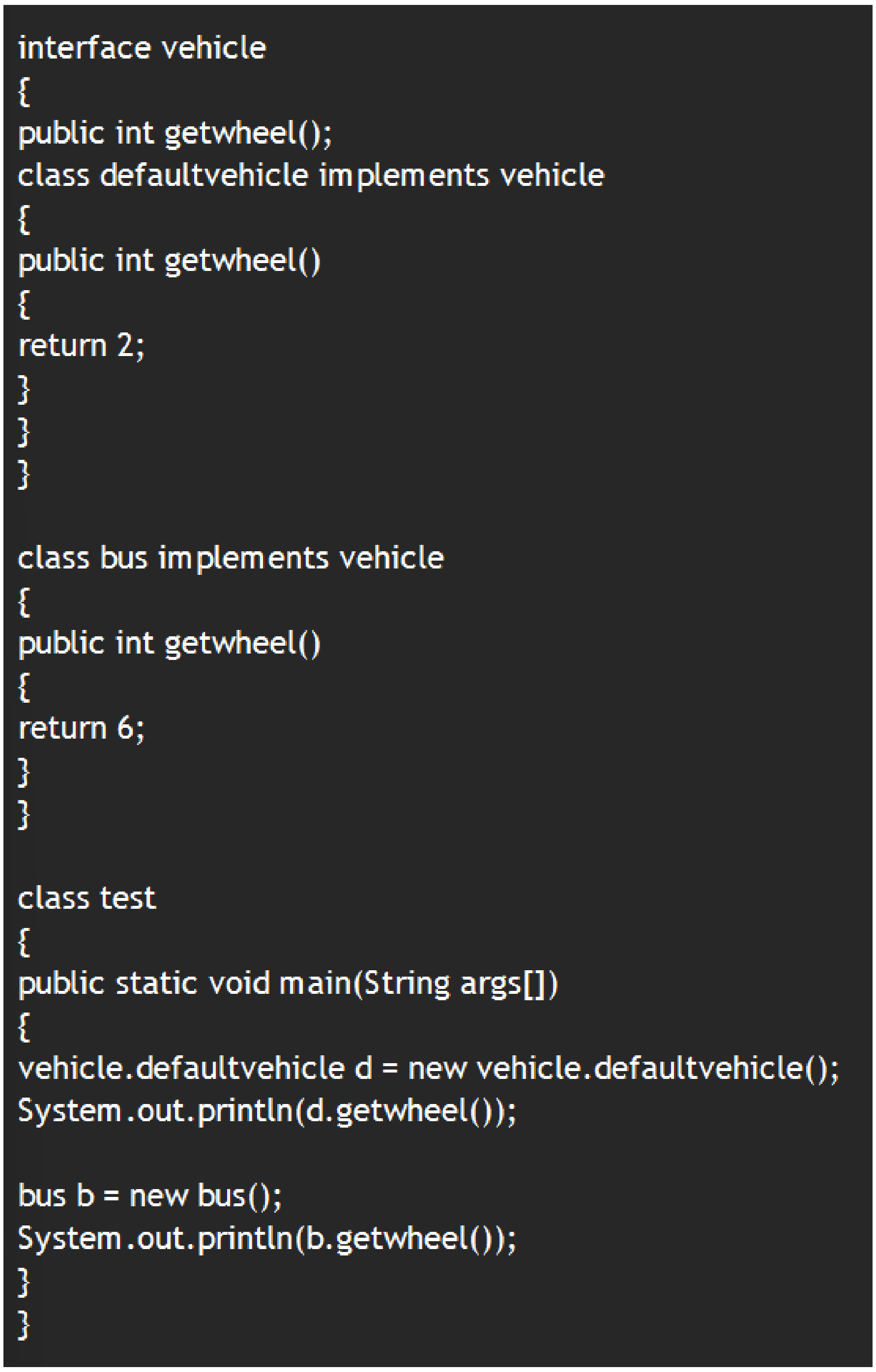
If functionality of a class is closely associated with interface then it is highly recommended to declare class inside interface.

In below example email details is required only for email service and we are not using anywhere else. Hence email details class is recommended to declare inside email service interface.

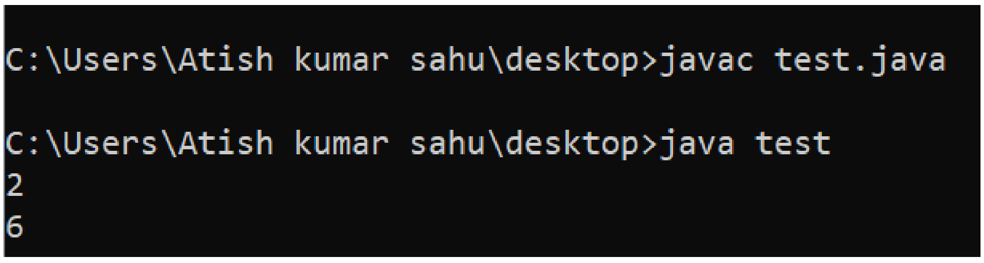
Example:

```
interface mailservice
{
public void sendmail(emaildetails e);
class emaildetails
{
String tolist;
String cclist;
String subject;
String body;
}
}
```

We can also implement a class inside interface to provide default implementation for that interface.



```
interface vehicle
{
public int getwheel();
class defaultvehicle implements vehicle
{
public int getwheel()
{
return 2;
}
}
}

class bus implements vehicle
{
public int getwheel()
{
return 6;
}
}

class test
{
public static void main(String args[])
{
vehicle.defaultvehicle d = new vehicle.defaultvehicle();
System.out.println(d.getwheel());

bus b = new bus();
System.out.println(b.getwheel());
}
}
```



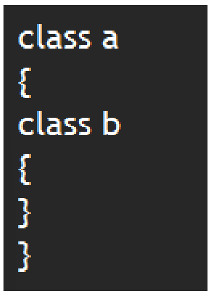
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
2
6
```

In the above example default vehicle is the default implementation of vehicle interface where as bus is customized implementation of vehicle interface.

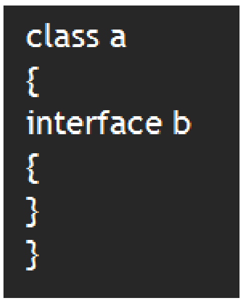
THE CLASS WHICH IS DECLARED INSIDE INTERFACE IS ALWAYS PUBLIC STATIC WHETHER WE ARE DECLARING OR NOT. HENCE WE CAN CREATE CLASS OBJECT DIRECTLY WITHOUT HAVING OUTER INTERFACE TYPE OBJECT.

CONCLUSION:

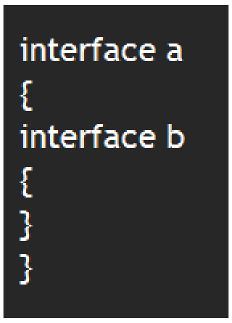
1. among classes and interfaces we can declare anything inside anything.



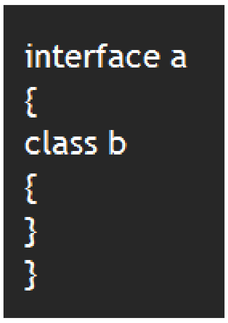
```
class a
{
class b
{
}
}
```



```
class a
{
interface b
{
}
}
```



```
interface a
{
interface b
{
}
}
```



```
interface a
{
class b
{
}
}
```

2. the interface which is declared inside interface is always public and static whether we are declaring or not.

3. the class which is declared inside interface is always public and static whether we are declaring or not.

4. the interface which is declared inside a class is always static but need not be public.

JAVA.LANG PACKAGE

INTRODUCTION:

For writing any java program whether it is simple or complex the most commonly required classes and interfaces are grouped into a separate package which is nothing but “java.lang” package. We are not require to import “java.lang” package explicitly because all classes and interfaces present in lang package by default available to every java program.

java.lang.Object:

The most commonly required methods for every java class (whether it is predefined class or customized class) or defined in separate class which is nothing but Object class. every class in java is the child class of object either direcly or indirectly so that object class methods by default available to every java class. Hence object class is considered as root of all java classes.

NOTE:

If our class doesnot extend any other class then only our class is the direct child class of Object. class A { } -> A is child of Object. If our class extends any other class then our class is indirect child class of object. class A extends B -> A is child of B and B child of Object which is a multilevel inheritance. Either directly or indirectly java won’t provide support for multiple inheritance with respect to classes.

Object class defines the following 11 methods

1. public String toString()
2. public native int hashCode()
3. public boolean equals(Object o)
4. protected native Object clone() throws CloneNotSupportedException
5. protected void finalize() throws Throwable
6. public final class getClass()
7. public final void wait() throws InterruptedException
8. public final native void wait(long ms) throws InterruptedException
9. public final void wait(long ms, int s) throws InterruptedException
10. public native final void notify()
11. public native final void notifyAll()

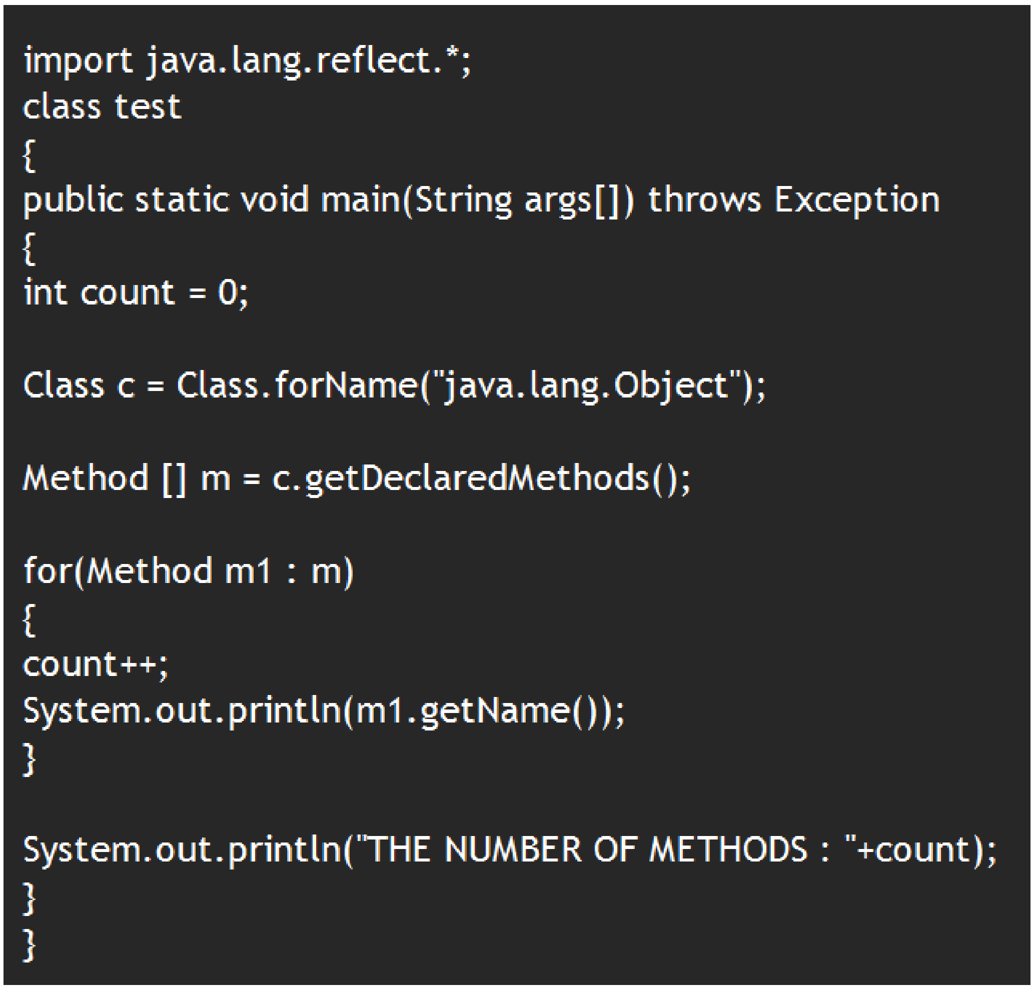
Strictly speaking object class contains 12 methods the extra method is “private static native void registerNatives();” this method internally required for object class and not available for child classes hence we are not required to ocnssider this method.

toString():

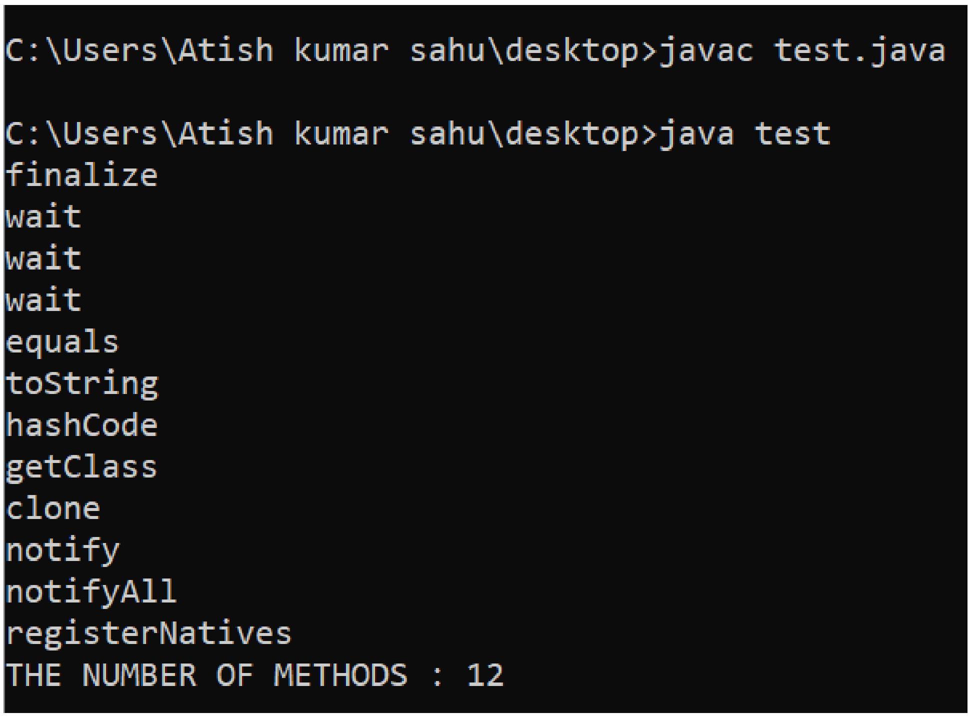
We can use toString() to get String representation of an Object. “String s = Obj.toString();” whenever we are trying to print Object reference internally toString() method will be called.

Student s = new Student();

SOP(s); -> SOP(s.toString());

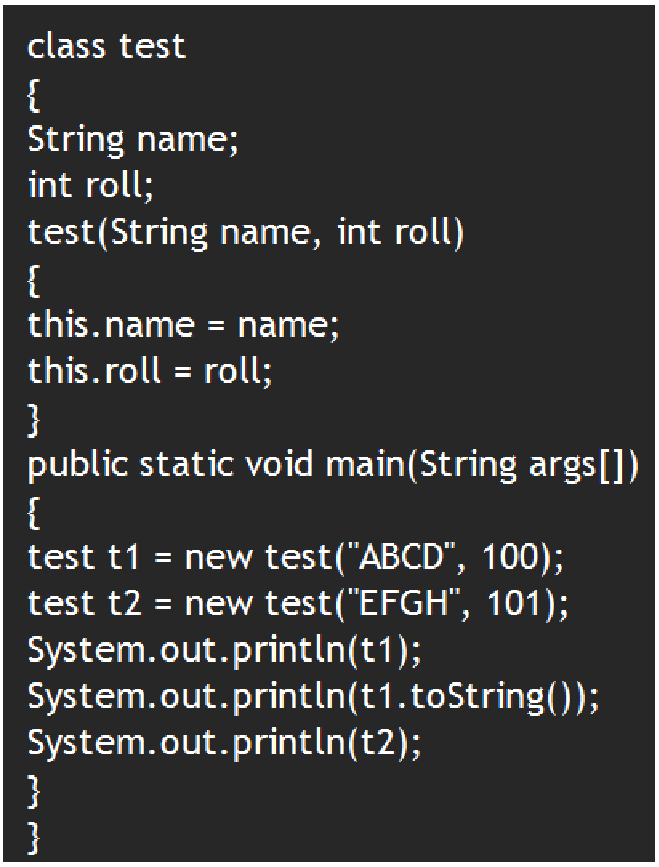


```
import java.lang.reflect.*;
class test
{
public static void main(String args[]) throws Exception
{
int count = 0;

Class c = Class.forName("java.lang.Object");

Method [] m = c.getDeclaredMethods();

for(Method m1 : m)
{
count++;
System.out.println(m1.getName());
}

System.out.println("THE NUMBER OF METHODS : "+count);
}
}
```

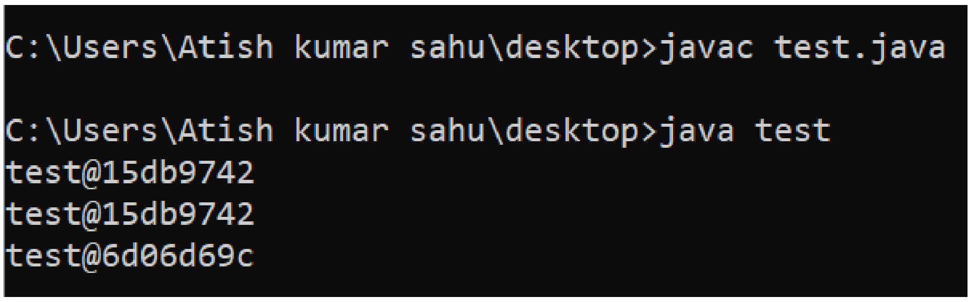


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
finalize
wait
wait
wait
equals
toString
hashCode
getClass
clone
notify
notifyAll
registerNatives
THE NUMBER OF METHODS : 12
```

If our class doesnot contain toString() method then object class toString() method will be executed.



```
class test
{
String name;
int roll;
test(String name, int roll)
{
this.name = name;
this.roll = roll;
}
public static void main(String args[])
{
test t1 = new test("ABCD", 100);
test t2 = new test("EFGH", 101);
System.out.println(t1);
System.out.println(t1.toString());
System.out.println(t2);
}
}
```

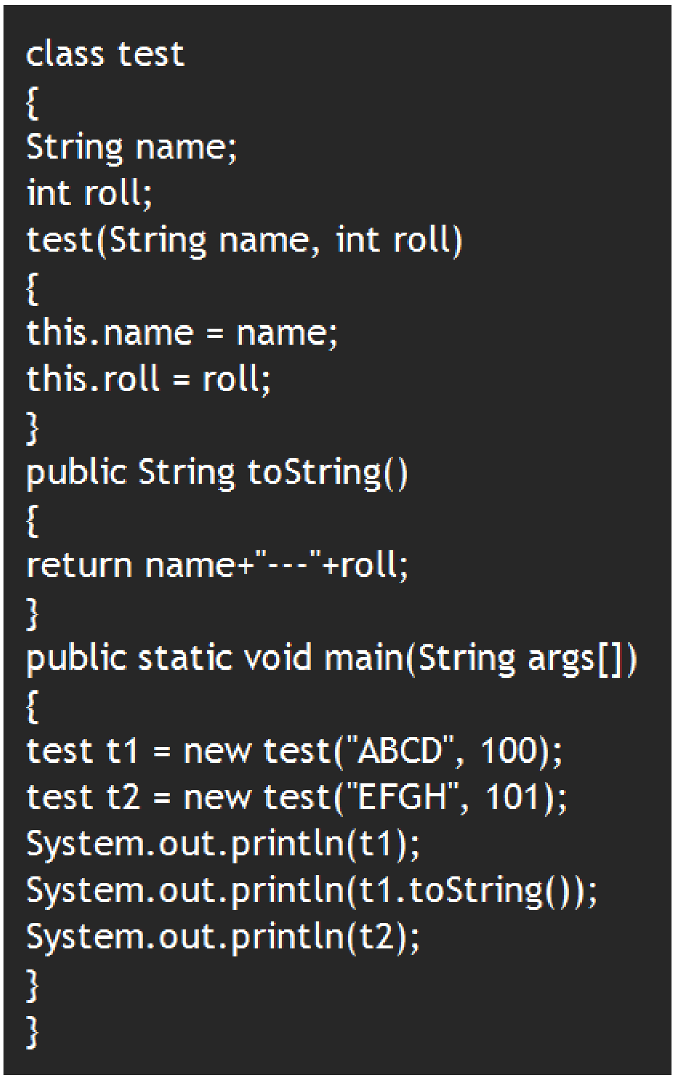


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
test@15db9742
test@15db9742
test@6d06d69c
```

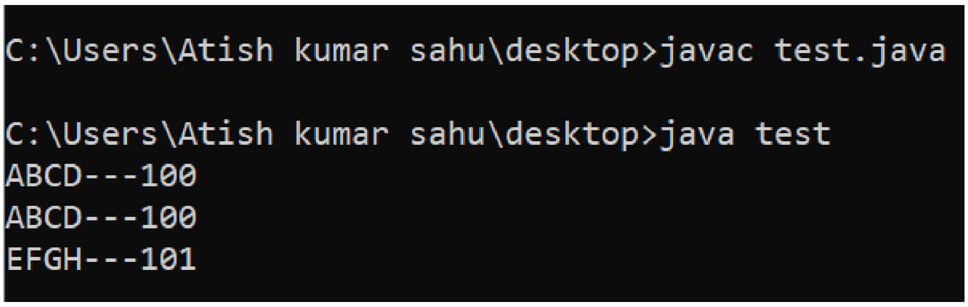
In the above example object class toString() got executed which is implemented as follows.

```
public String toString()

{

return getClass().getName()+ “@” +Integer.toHexString(hashCode());

}
```

classname@hascode_in_hexadecimal_form = test@15db9742

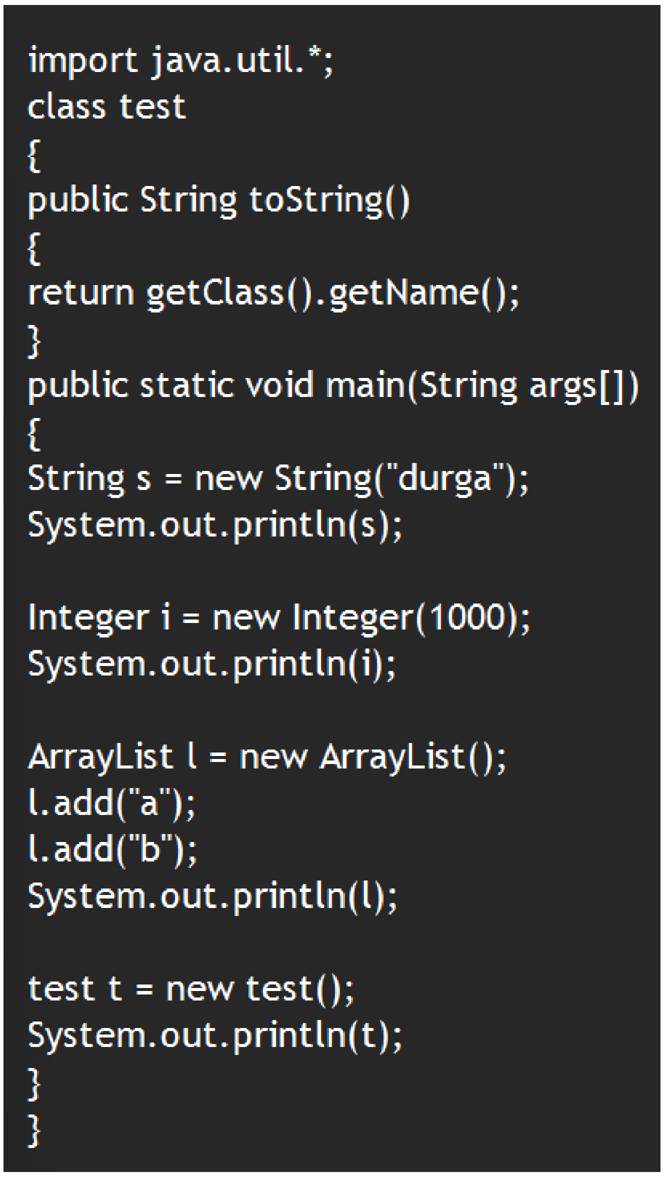


```
class test
{
String name;
int roll;
test(String name, int roll)
{
this.name = name;
this.roll = roll;
}
public String toString()
{
return name+"---"+roll;
}
public static void main(String args[])
{
test t1 = new test("ABCD", 100);
test t2 = new test("EFGH", 101);
System.out.println(t1);
System.out.println(t1.toString());
System.out.println(t2);
}
}
```

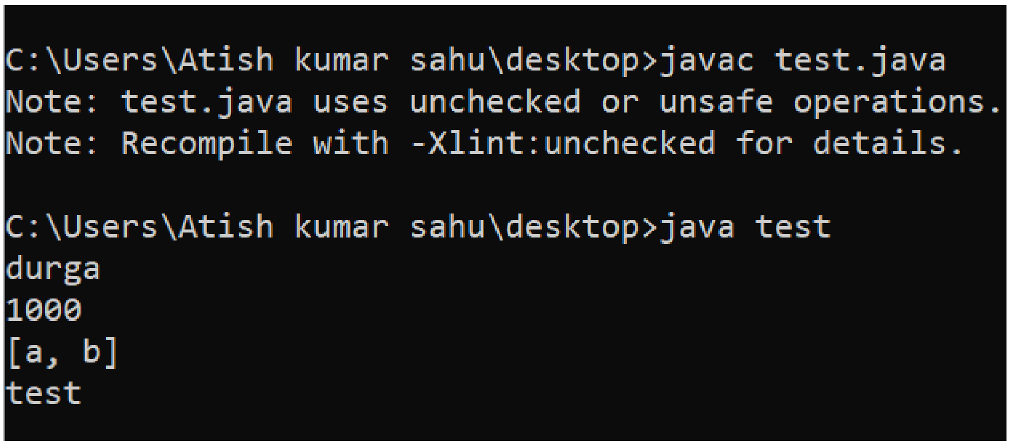


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
ABCD---100
ABCD---100
EFGH---101
```

Based on our requirement we can override toString() method to provide our own String Representation. Example: whenever we are trying to print test object reference to print student's name and roll no we have to override toString() method as follows.



```
import java.util.*;
class test
{
public String toString()
{
return getClass().getName();
}
public static void main(String args[])
{
String s = new String("durga");
System.out.println(s);

Integer i = new Integer(1000);
System.out.println(i);

ArrayList l = new ArrayList();
l.add("a");
l.add("b");
System.out.println(l);

test t = new test();
System.out.println(t);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java
Note: test.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:\Users\Atish kumar sahu\desktop>java test
durga
1000
[a, b]
test
```

In all wrraper classes in all collection classes String class, StringBuffer and StringBuilder classes toString() method is overriden for meaningful String representation hence it is highly recommended to override toString() method in our class also.

hashCode():

LINEAR SEARCH COMPLEXITY = O(n);

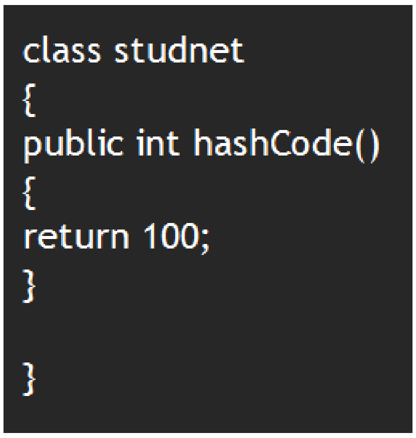
BINARY SEARCH COMPLEXITY = O($\log_2$ n)

HASHING ALGORITHM COMPLEXITY = O(1)

For every object a unique number is generated by JVM which is nothing but hashCode. hashCode won't represent address of object. JVM will use hashCode while saving object into hashing related data structure like hash table, hash map, hash set, etc. the main advantage of saving objects based on hashCode is search operation will become easy(the most powerful search algorithm upto today is hashing).

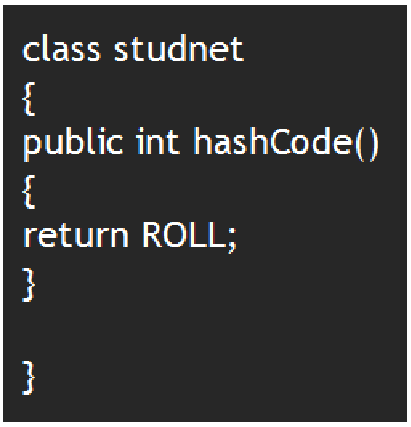
If we are giving the chance to object class hashCode method it will generates hashCode based on address of the object. it doesn't mean hashCode represents address of the object. based on our requirement we can override hashCode method in our class to generate our own hashCode.

Overriding hashCode method is said to be proper if and only if for every object we have to generate a unique number as hash code.



```
class studnet
{
public int hashCode()
{
return 100;
}

}
```

The above code is a wrong way of code. This is a wrong way of overriding hashCode method because for all students objects we are generating same number as hashcode.

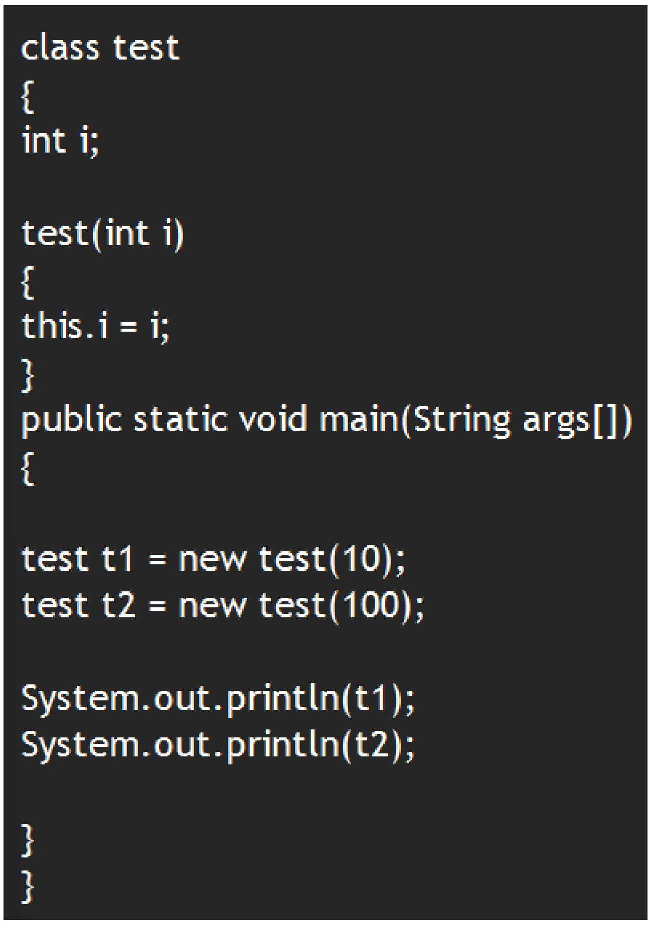


```
class studnet
{
public int hashCode()
{
return ROLL;
}

}
```

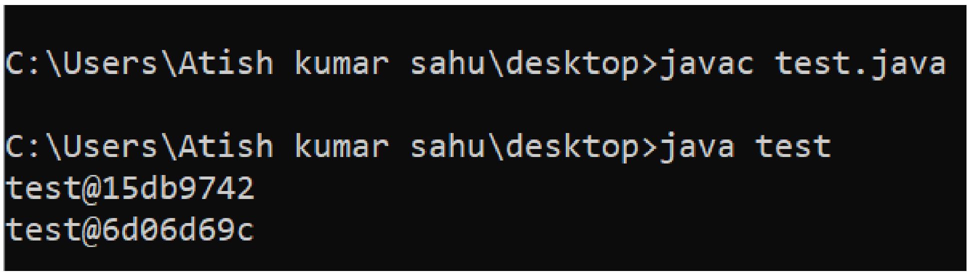
The above code is a correct way of code. This is a coreect way of overriding hashCode method because for all students we generates a different unique number as a hashCode for every objects.

toString vs hashCode():

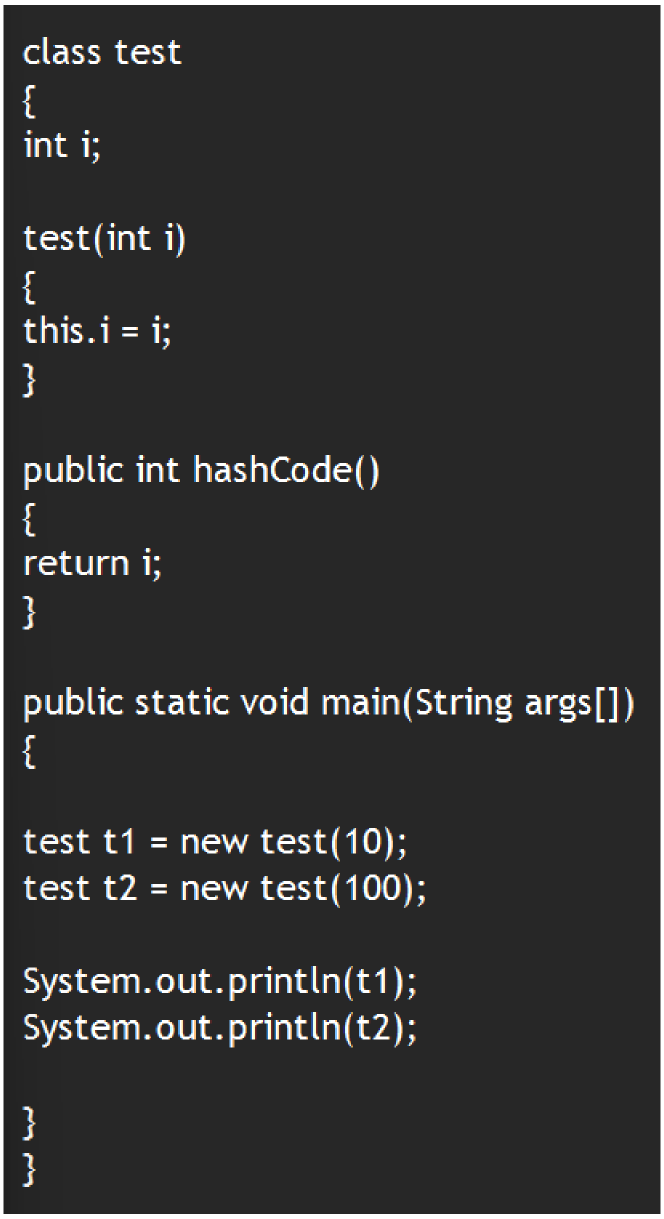
if we are giving the chance to Object class toString() method it will internally calls hashCode() method. If we are overriding toString() method then our toString() method may not call hashCode().



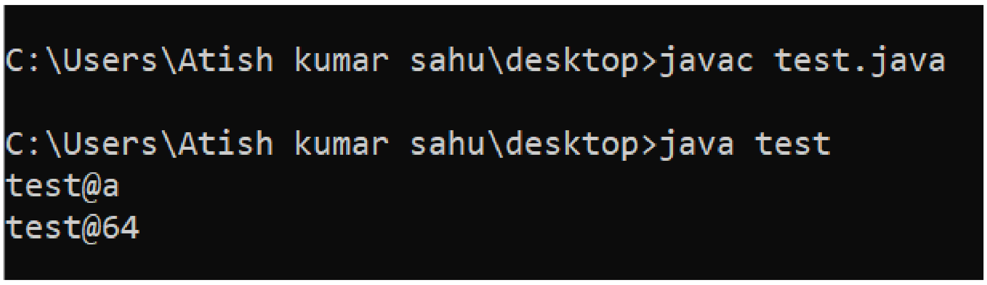
```
class test
{
int i;

test(int i)
{
this.i = i;
}
public static void main(String args[])
{

test t1 = new test(10);
test t2 = new test(100);

System.out.println(t1);
System.out.println(t2);

}
}
```



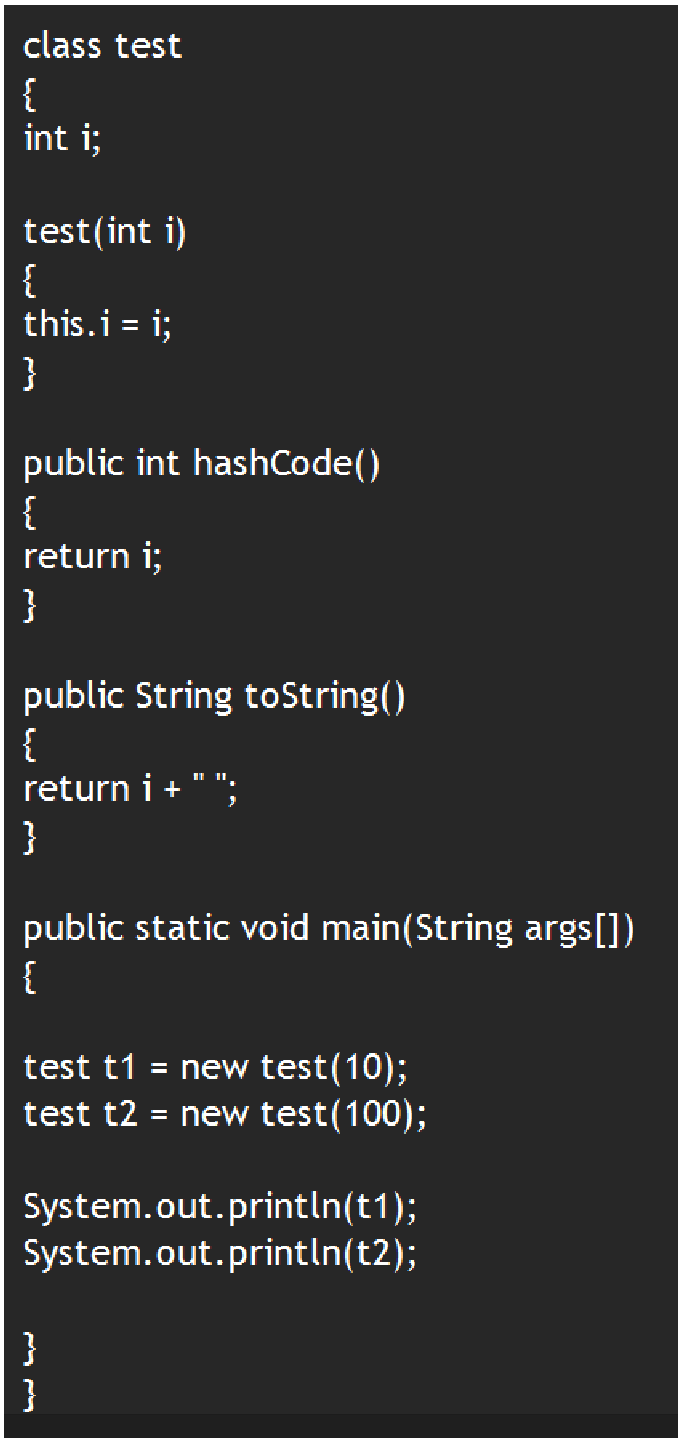
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
test@15db9742
test@6d06d69c
```



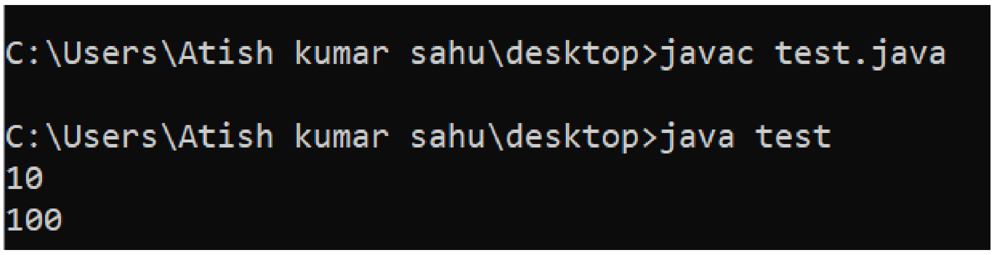
```
class test
{
int i;

test(int i)
{
this.i = i;
}

public int hashCode()
{
return i;
}

public static void main(String args[])
{

test t1 = new test(10);
test t2 = new test(100);

System.out.println(t1);
System.out.println(t2);

}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
test@a
test@64
```



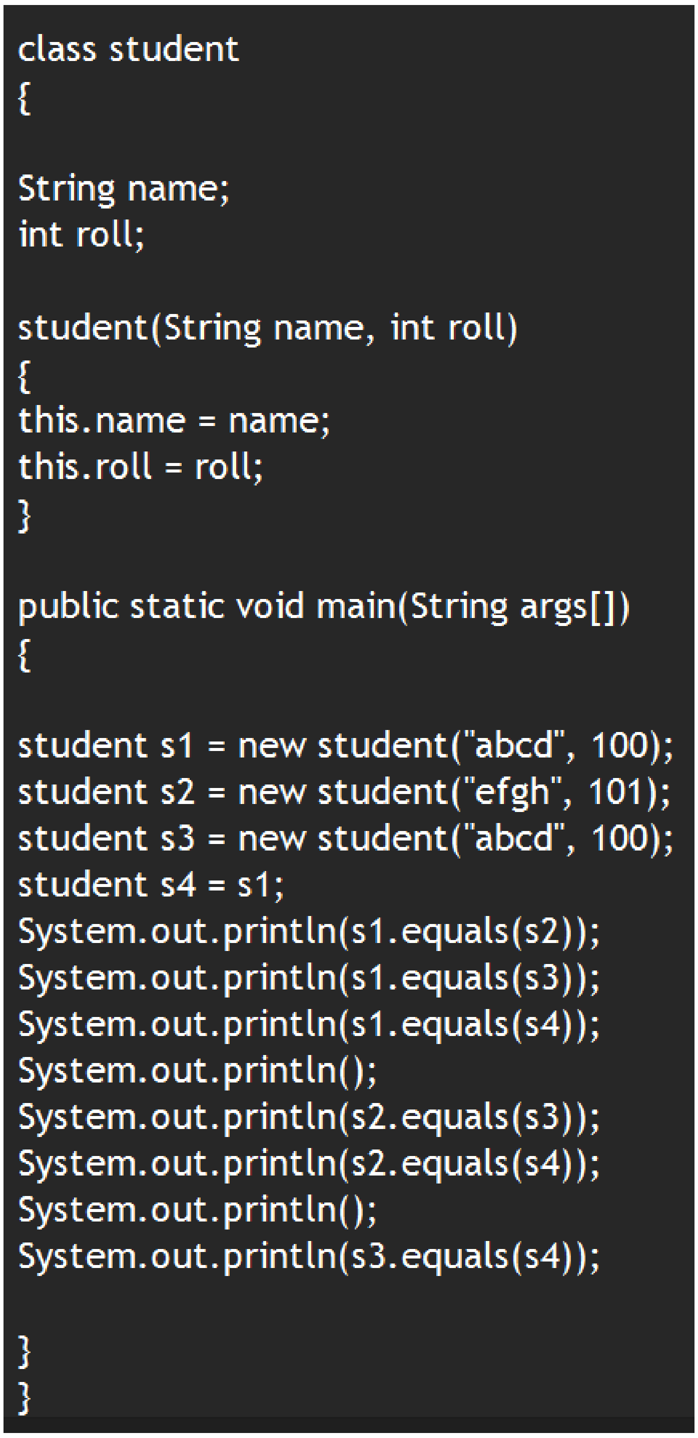
```
class test
{
int i;

test(int i)
{
this.i = i;
}

public int hashCode()
{
return i;
}

public String toString()
{
return i + " ";
}

public static void main(String args[])
{

test t1 = new test(10);
test t2 = new test(100);

System.out.println(t1);
System.out.println(t2);


}
}
```



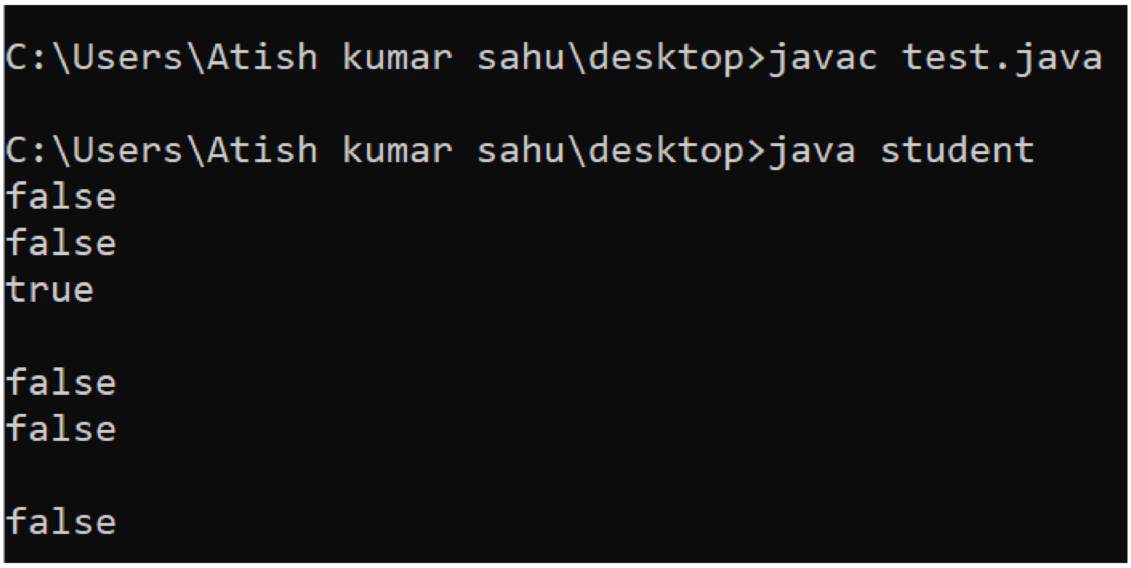
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
10
100
```

equals():

we can use equals() method to check equality of two objects. Example: “obj1.equals(obj2);”. If our class doesn’t contain equals method then object class equals method will be executed.

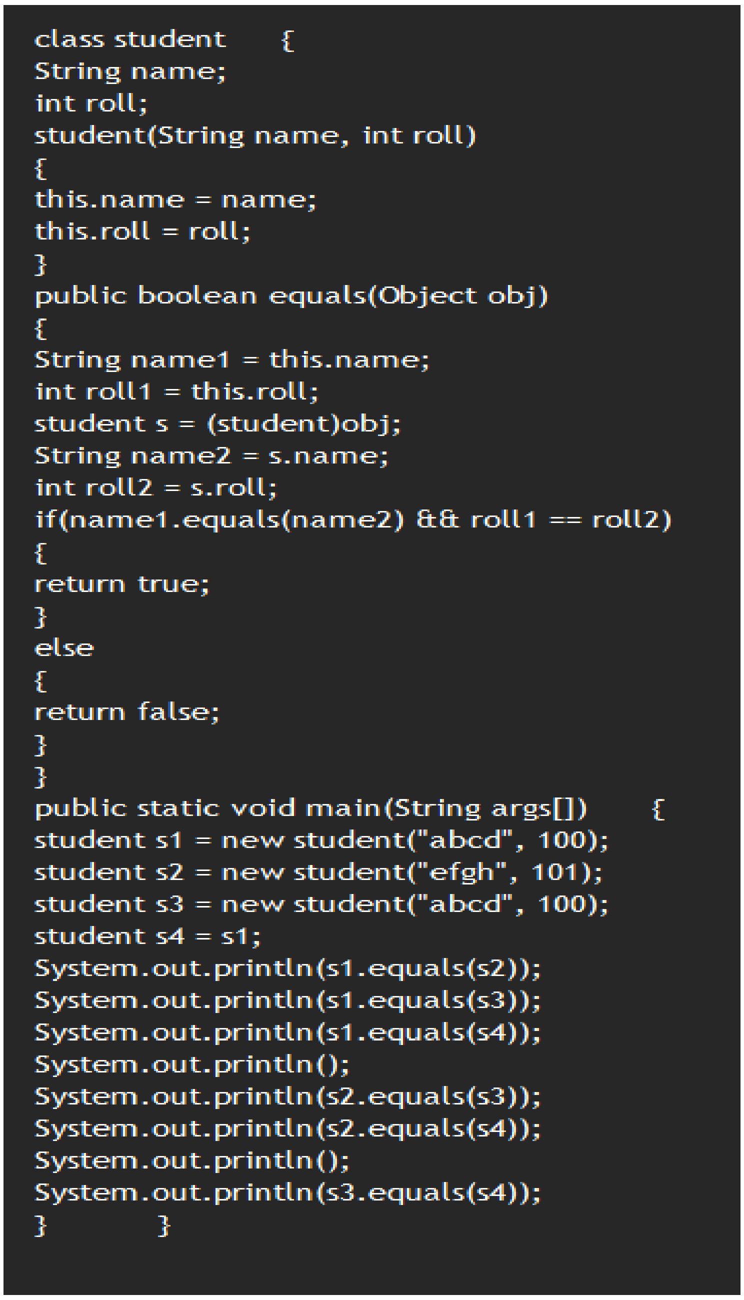


```
class student
{

String name;
int roll;

student(String name, int roll)
{
this.name = name;
this.roll = roll;
}

public static void main(String args[])
{

student s1 = new student("abcd", 100);
student s2 = new student("efgh", 101);
student s3 = new student("abcd", 100);
student s4 = s1;
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
System.out.println(s1.equals(s4));
System.out.println();
System.out.println(s2.equals(s3));
System.out.println(s2.equals(s4));
System.out.println();
System.out.println(s3.equals(s4));

}
}
```

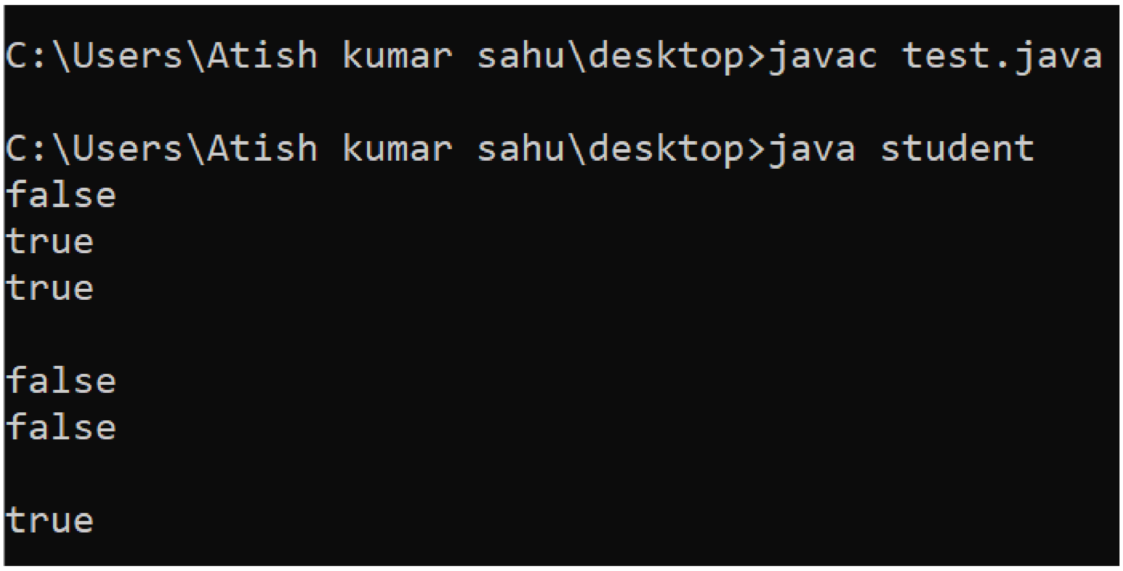


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java student
false
false
true

false
false

false
```

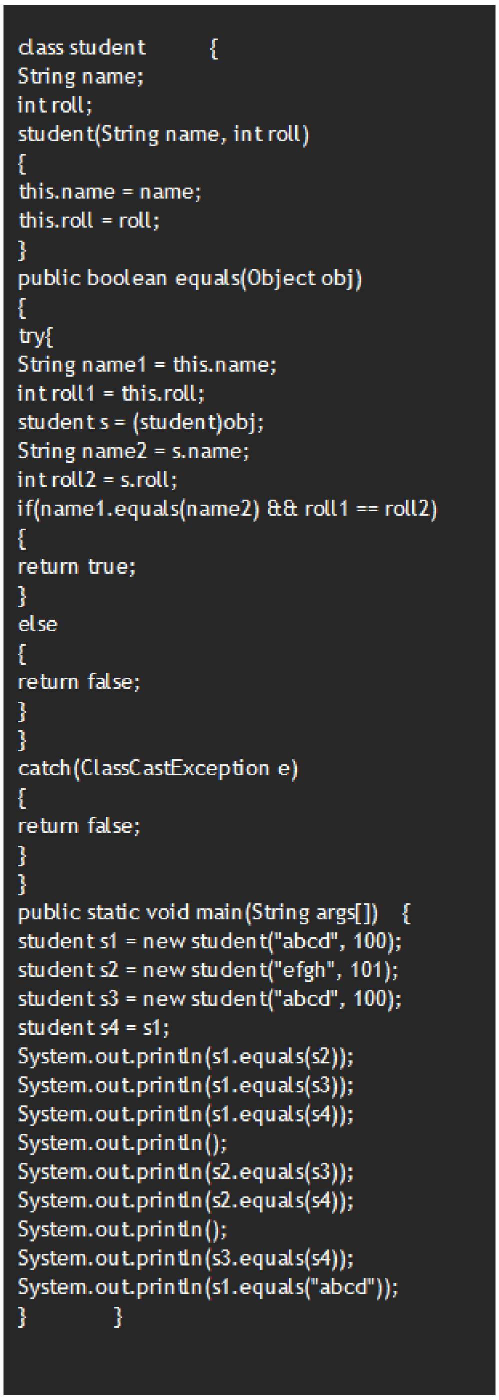
In the above example Object class equals() method got executed which is meant for reference comparison(address comparison) that is if two references pointing to the same object then only equals() method returns true. Based on our requirement we can override equals method for content comparison.



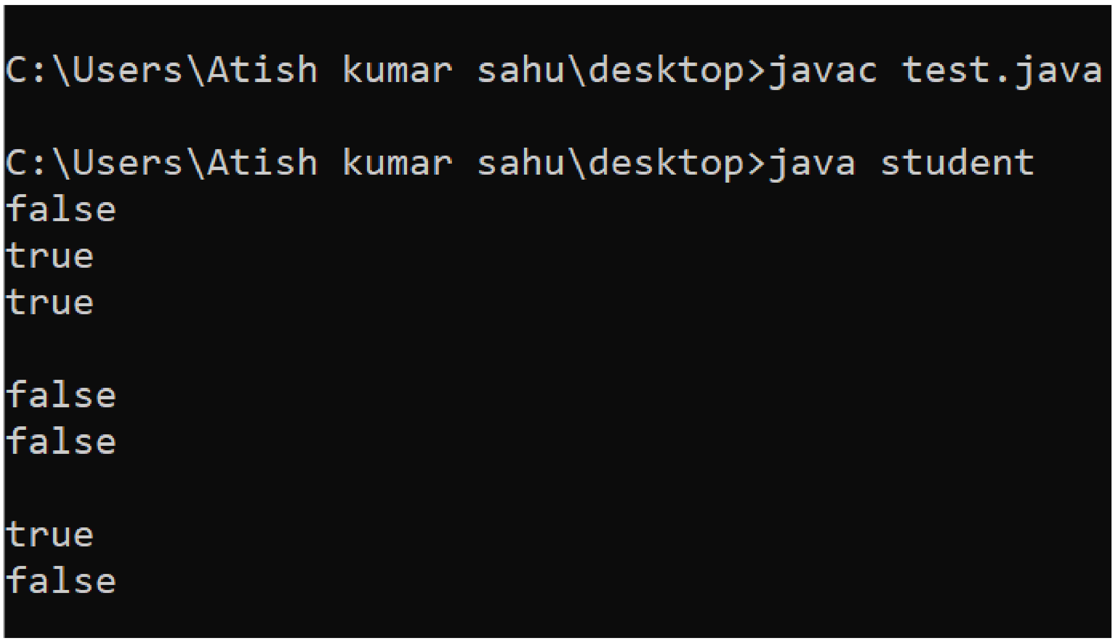
```
class student    {
String name;
int roll;
student(String name, int roll)
{
this.name = name;
this.roll = roll;
}
public boolean equals(Object obj)
{
String name1 = this.name;
int roll1 = this.roll;
student s = (student)obj;
String name2 = s.name;
int roll2 = s.roll;
if(name1.equals(name2) && roll1 == roll2)
{
return true;
}
else
{
return false;
}
}
public static void main(String args[])      {
student s1 = new student("abcd", 100);
student s2 = new student("efgh", 101);
student s3 = new student("abcd", 100);
student s4 = s1;
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
System.out.println(s1.equals(s4));
System.out.println();
System.out.println(s2.equals(s3));
System.out.println(s2.equals(s4));
System.out.println();
System.out.println(s3.equals(s4));
}       }
```



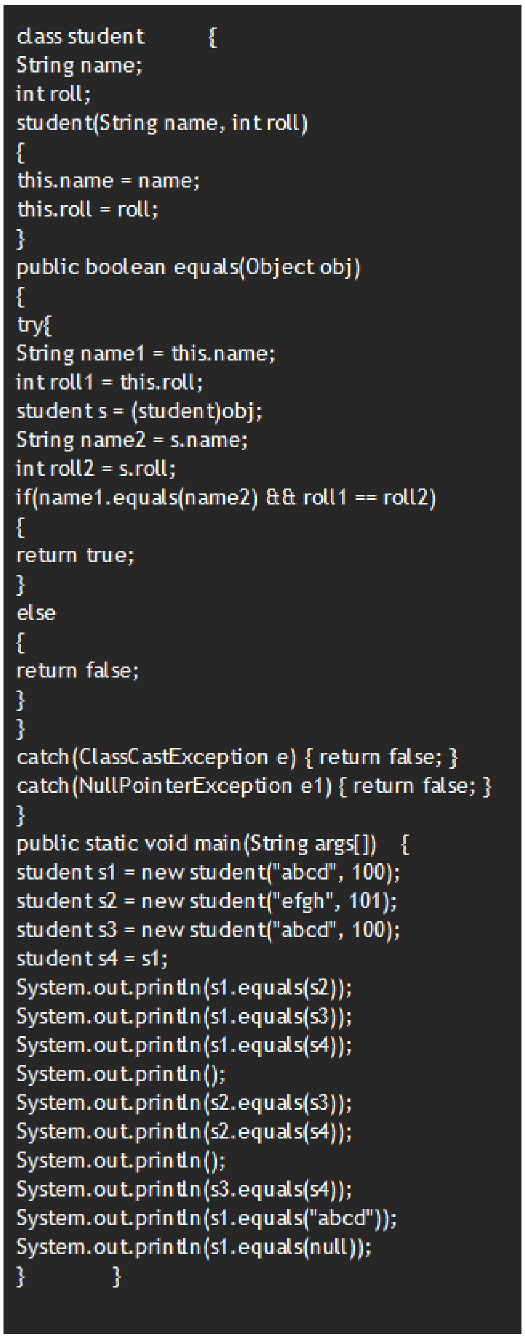
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java student
false
true
true

false
false

true
```



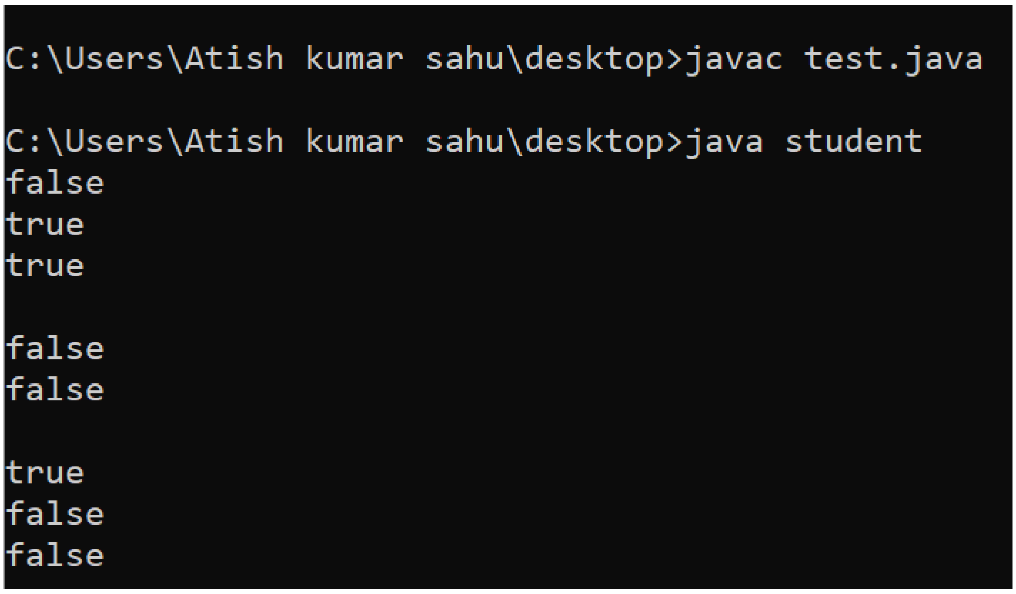
```
class student        {
String name;
int roll;
student(String name, int roll)
{
this.name = name;
this.roll = roll;
}
public boolean equals(Object obj)
{
try{
String name1 = this.name;
int roll1 = this.roll;
student s = (student)obj;
String name2 = s.name;
int roll2 = s.roll;
if(name1.equals(name2) && roll1 == roll2)
{
return true;
}
else
{
return false;
}
}
catch(ClassCastException e)
{
return false;
}
}
public static void main(String args[])    {
student s1 = new student("abcd", 100);
student s2 = new student("efgh", 101);
student s3 = new student("abcd", 100);
student s4 = s1;
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
System.out.println(s1.equals(s4));
System.out.println();
System.out.println(s2.equals(s3));
System.out.println(s2.equals(s4));
System.out.println();
System.out.println(s3.equals(s4));
System.out.println(s1.equals("abcd"));
}          }
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java student
false
true
true

false
false

true
false
```



```
class student      {
String name;
int roll;
student(String name, int roll)
{
this.name = name;
this.roll = roll;
}
public boolean equals(Object obj)
{
try{
String name1 = this.name;
int roll1 = this.roll;
student s = (student)obj;
String name2 = s.name;
int roll2 = s.roll;
if(name1.equals(name2) && roll1 == roll2)
{
return true;
}
else
{
return false;
}
}
catch(ClassCastException e) { return false; }
catch(NullPointerException e1) { return false; }
}
public static void main(String args[])   {
student s1 = new student("abcd", 100);
student s2 = new student("efgh", 101);
student s3 = new student("abcd", 100);
student s4 = s1;
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
System.out.println(s1.equals(s4));
System.out.println();
System.out.println(s2.equals(s3));
System.out.println(s2.equals(s4));
System.out.println();
System.out.println(s3.equals(s4));
System.out.println(s1.equals("abcd"));
System.out.println(s1.equals(null));
}         }
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java student
false
true
true

false
false

true
false
false
```

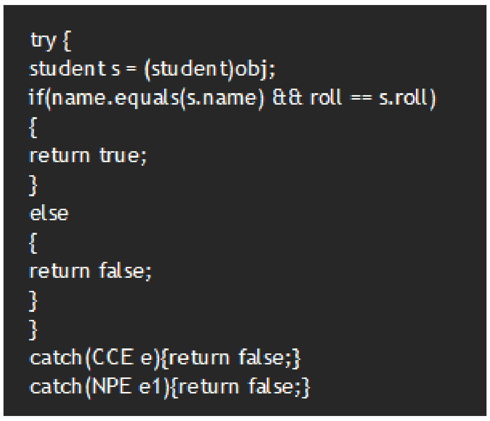
While overriding equals method for content comparison we have to take care about the following

1. What is the meaning of equality that is (whether we have to check only names or only roll or both)

2. If we are passing different type of object our equals method should not rise classcastexception that is we have to handle classcastexception to return false value.

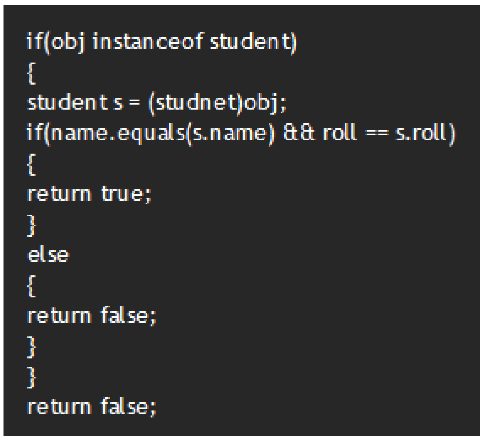
3. If we are passing null argument then our equals method should not rise nullpointerexception that is we have to handle null pointer exception. To return false.

The following is the correct way of overriding equals method for student class content comparison.



```
try {
student s = (student)obj;
if(name.equals(s.name) && roll == s.roll)
{
return true;
}
else
{
return false;
}
}
catch(CCE e){return false;}
catch(NPE e1){return false;}
```

= simplified version of equals() method.



```
if(obj instanceof student)
{
student s = (studnet)obj;
if(name.equals(s.name) && roll == s.roll)
{
return true;
}
else
{
return false;
}
}
return false;
```

-> more simplified version of equals().

```
if(obj ==this)
return true;
if(obj instanceof student)
{
student s = (studnet)obj;
if(name.equals(s.name) && roll == s.roll)
{
return true;
}
else
{
return false;
}
}
return false;
```

To make above equals method more efficient we have to write the following code at the begging inside equals method.

If(obj == this)

return true;

according to this if both references pointing to same object then without performing any comparison equals() return true directly.

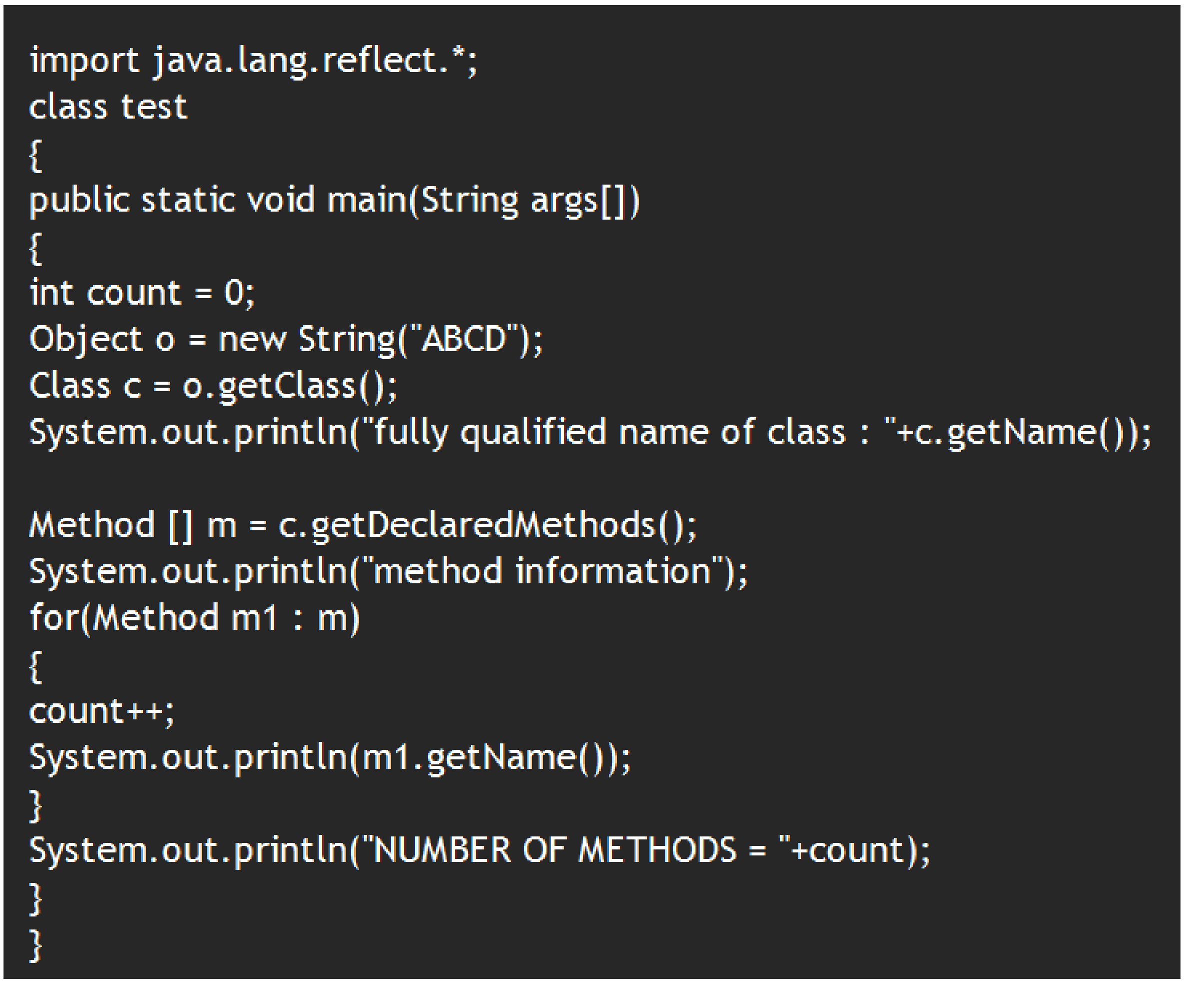
```
String s1 = new String("abcd");
String s2 = new String("abcd");
SOP(s1 == s2); //false
SOP(s1.equals(s2)); //true

StringBuffer sb1 = new StringBuffer("ABCD");
StringBuffer sb2 = new StringBuffer("ABCD");
SOP(sb1 == sb2); //false
SOP(sb1.equals(sb2)); //false
```

in String class equals() method is overriden for content comparison. Hence, even though objects are different if content is same then equals() returns true.

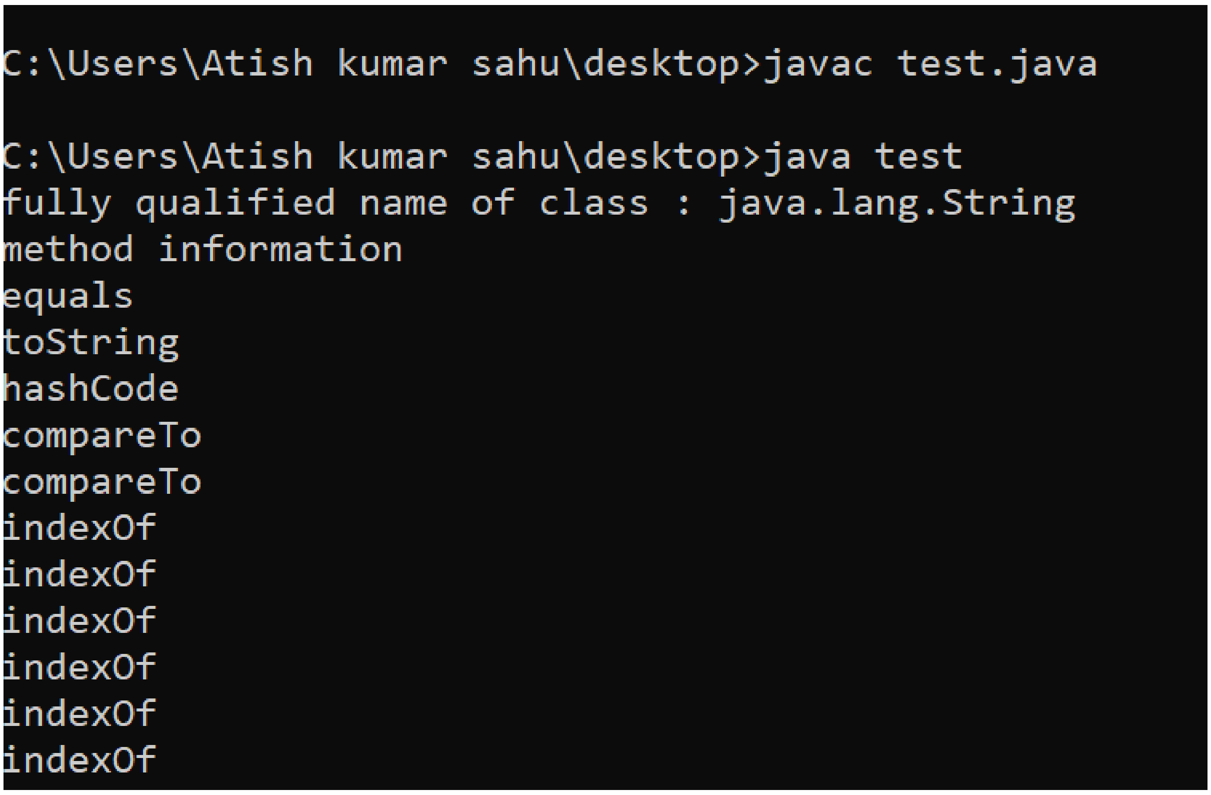
In StringBuffer class equals() method is not overriden for content comparison hence if objects are different equals() method return false even though content is same.

getClass():

we can use getClass() method to get runtime class definition of an object. “public final class getClass()” by using this class class-object we can access class level properties like fully qualified name of the class, methods information, constructor information, etc.



```
import java.lang.reflect.*;
class test
{
public static void main(String args[])
{
int count = 0;
Object o = new String("ABCD");
Class c = o.getClass();
System.out.println("fully qualified name of class : "+c.getName());

Method [] m = c.getDeclaredMethods();
System.out.println("method information");
for(Method m1 : m)
{
count++;
System.out.println(m1.getName());
}
System.out.println("NUMBER OF METHODS = "+count);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
fully qualified name of class : java.lang.String
method information
equals
toString
hashCode
compareTo
compareTo
indexOf
indexOf
indexOf
indexOf
indexOf
indexOf
```

To display database vendor specific connection interface implemented class name

Connection con = DriverManager.getConnection();

System.out.println(con.getClass().getName());

<u>NOTE:</u>

1. After loading every (.class) file jvm will create an object of the type java.lang.Class in the heap area. Programmer can use this class object to get class level information.

2. We can use get class method very frequently in reflections.

<u>finalize():</u>

it is related to garbage collector. Just before destroying an object garbage collector calls finalize() method to perform clean up activities. Once finalize method completes automatically garbage collector destroys that object.

<u>wait(), notify(), notifyAll():</u>

these are related to multi-threading concepts. We can use these methods for inter thread communication. The thread which is expecting updation, it is responsible to call wait() method, then immediately the thread enter into waiting state. The thread which is responsible to perform updation, after performing updation, the thread can call notify method. the waiting thread will get that notification and continue its execution with those updates.

WRAPPER CLASS:

The main objectives of wrapper classes are To wrap primitive into object form do that we can handle primitives also just like objects. To define several utility methods which are required for the primitivies.

CONSTRUCTORS:

Almost all wrapper classes contains 2 constructors one can take coressponding primitives as argument and the other can take String as argument.

Ex1:

Integer I = new Integer(10);

Integer I = new Integer(“10”);

Ex2:

Double D = new Double(10.5);

Double D = new Double(“10.5”);

Ex3:

Integer I = new Integer(“ten”); -> runtime exception = NumberFormatException

If String argument not representing a number then we will get runtime exception saying “NumberFormatException”.

Float class contains 3 constructors with float, double, String arguments.

Float F = new Float(10.5f); -> valid

Float F = new Float(“10.5f”); -> valid

Float F = new Float(10.5); -> valid

Float F = new Float(“10.5”); -> valid

Character class contains only one constructor which can take char argument

Character ch = new Character(‘a’);

Boolean class contains 2 constructors one can take boolean primitive as argument and other can take String argument. If we pass boolean primitive as argument the only allowed values are true or false. Where case is important and content is also important

Boolean b = new Boolean(true); -> valid

Boolean b = new Boolean(false); -> valid

Boolean B = new Boolean(True); -> invalid

Boolean B = new Boolean(ABCD); -> invalid

If we are passing String type as argument then case and content both are not important. If the content is case insensitive String of “true” then it is treated as true. Otherwise it is treated as false.

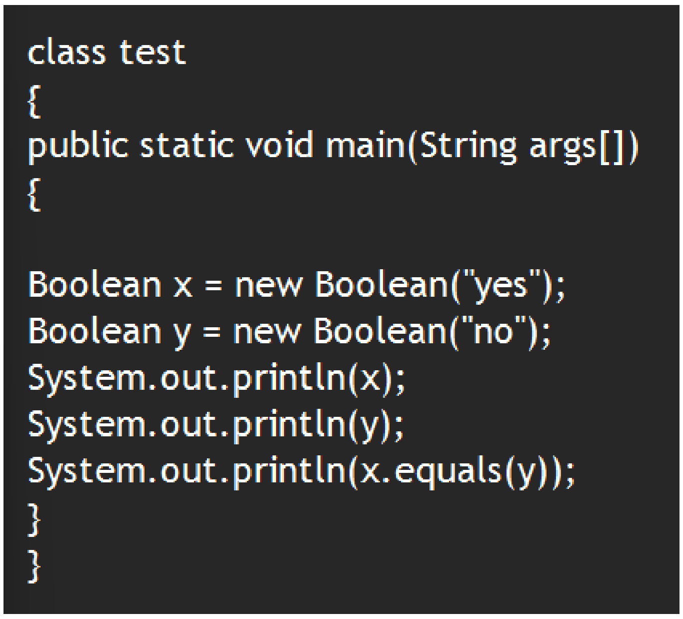
Boolean B = new Boolean(“true”); -> true

Boolean B = new Boolean(“True”); -> true

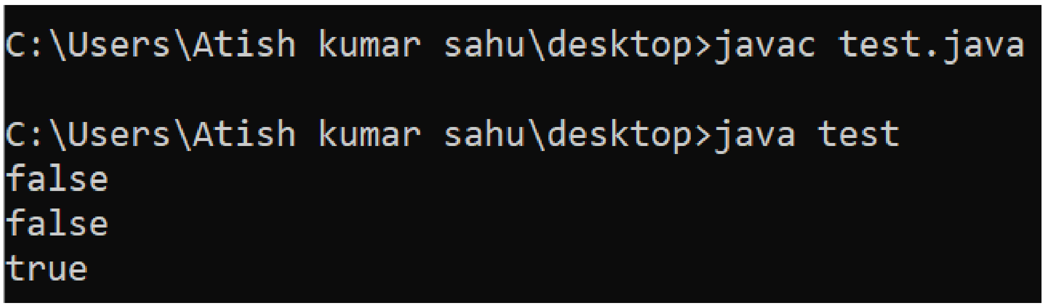
Boolean B = new Boolean(“TRUE”); -> true

Boolean B = new Boolean(“ABCD”); -> false

Boolean B = new Boolean(“EFGH”); -> false



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

Boolean x = new Boolean("yes");
Boolean y = new Boolean("no");
System.out.println(x);
System.out.println(y);
System.out.println(x.equals(y));
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
false
false
true
```

<u>WRAPPER CLASS</u>	<u>CORRESPONDING CONSTRUCTOR ARGUMENTS</u>
Byte	byte or String
Short	short or String
Integer	int or String
Long	long or String
Float	float or String or double ***
Double	double or String
Character	char ***
Boolean	boolean or String ***

In all wrapper classes toString() method is overriden to return content directly. In all wrapper classes equals() method is overriden for content comparison.

<u>WRAPPER CLASS UTILITY METHODS:</u>

1. valueOf()
2. xxxValue()
3. parsexxx()
4. toString()

<u>1. valueOf();</u>

We can use value of methods to create wrapper object for the given primitive or String. Every wrapper class except Character class contains a static value of methods to create Wrapper object for the given String.

"public static wrapper valueOf(String s)"

<u>example:</u>

Integer I = Integer.valueOf("10");

Double D = Double.ValueOf("10.5");

Boolean B = Boolean.valueOf("abcd");

<u>Example:</u>

Integer I = Integer.valueOf("1111");

SOP(I);

Integer I = Integer.valueOf("1111",2); -> consider as binary

SOP(I); = 15

Every integral type wrapper class Byte, Short, Integer, Long contains the following valueOf() method to create wrapper object for the given specified radix String.

"public static wrapper valueOf(String s, int radix);"

the allowed range of radix is 2 to 36

<u>example:</u>

Integer I = Integer.valueOf("100",2);

SOP(I); 4

Integer I = Integer.valueOf("101",4);

SOP(I); 17

Every wrapper class including character class contains a static value of methods to create wrapper object for the given primitive.

“public static wrapper valueOf(primitive p);”

<u>Example:</u>

Integer I = Integer.valueOf(10);

Character C = Character.valueOf(‘a’);

Boolean B = Boolean.valueOf(10.5);

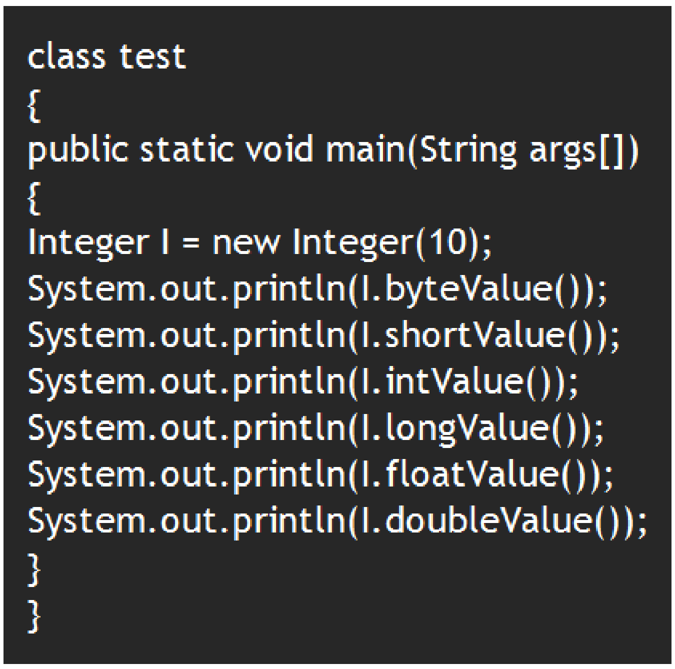
Primitive/String ------valueOf()-------wrapper Object

<u>2. xxxValue():</u>

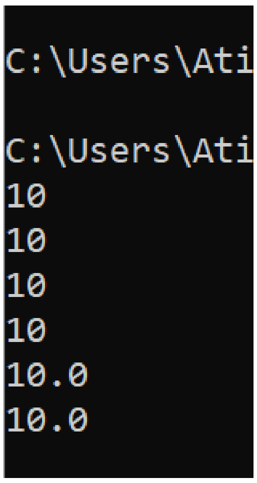
We can use xxxvalue() methods to create to get primitive for the given wrapper object.

Every number type wrapper class (byte, short, int, long, float, double) contains the following 6 methods to get primitive from the given wrapper object.

public byte byteValue();	public short shortValue();	public char charValue()
public int intValue();	public long longValue();	public boolean booleanValue()
public float floatValue();	public double doubleValue();	



```
class test
{
public static void main(String args[])
{
Integer I = new Integer(10);
System.out.println(I.byteValue());
System.out.println(I.shortValue());
System.out.println(I.intValue());
System.out.println(I.longValue());
System.out.println(I.floatValue());
System.out.println(I.doubleValue());
}
}
```



```
C:\Users\Ati

C:\Users\Ati
10
10
10
10
10.0
10.0
```

Character class contains charValue() method to get char primitive for the given character Object. **“public char charValue()”**.

Boolean class contains booleanValue() method to get boolean primitive for the given boolean Object. **“public boolean booleanValue()”**

In total 38 = 6*6 + 1 + 1 xxxValue() methods are possible.

wrapperObject ------xxxValue()------- primitive

3. parsexxx():

We can use parsexxx() methods to convert String to primitive.

Form1:

Every wrapper class except Character class contains the following parsexxx() method to find primitive for given String Object.

“**public static primitive parseXxx(String s);**”

Example:

int I = Integer.parseInt(“10”);

double d = Double.parseDouble(“10.5”);

boolean b = Bollean.parseBoolean(“true”);

form2:

every integral type wrapper class (Byte, Short, Integer, Long) contains the following parseXxx method to convert specified radix String to primitive.

“**public static primitive parseXxx(String s, int radix);**”

The allowed range of radix is 2 to 36

int I = Integer.parseInt(“1111”,2);

SOP(I); //15

String --------parseXxx() method-------- primitive

4. toString():

We can use toString() method to convert wrapper Object or primitive to String.

Form1:

Every wrapper class contains the following toString() method to convert wrapper Object to String type.

“**public String toString();**” It is the overriding version of object class toString() method. whenever we are trying to print wrapper Object reference internally this two String Method will be called.

Integer i = new Integer(10);

String s = i.toString();

SOP(s); SOP(i); SOP(i.toString); -> 10;

String s = Integer.ToString(10); -> 10 is primitive value;

From2:

Every wrapper class including Character class contains the following static toString() method to convert primitive to String.

“public static String toString(primitive p)”

String s = Integer.toString(10);

String s = Boolean.toString(true);

String s = Character.toString(‘a’);

From3:

“public static String toString(primitive p, int radix)” -> radix from 2 to 36

Integer and Long classes contains the following toString() method to convert primitive to specified radix String.

From4: toXxxString()

String s = Integer.toString(15,2);

String s = Integer.toBinaryString(15);

String s = Integer.toOctalString(15);

String s = Integer.toHexString(15);

Integer and Long classes contains the following toXxxString() methods.

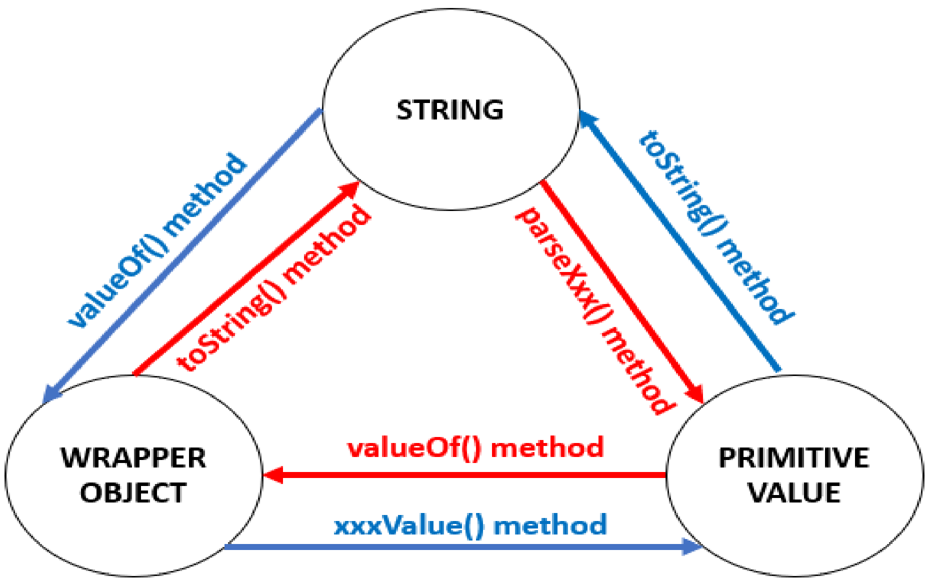
“public static String toBinaryString(primitive p)”

“public static String toOctalString(primitive p)”

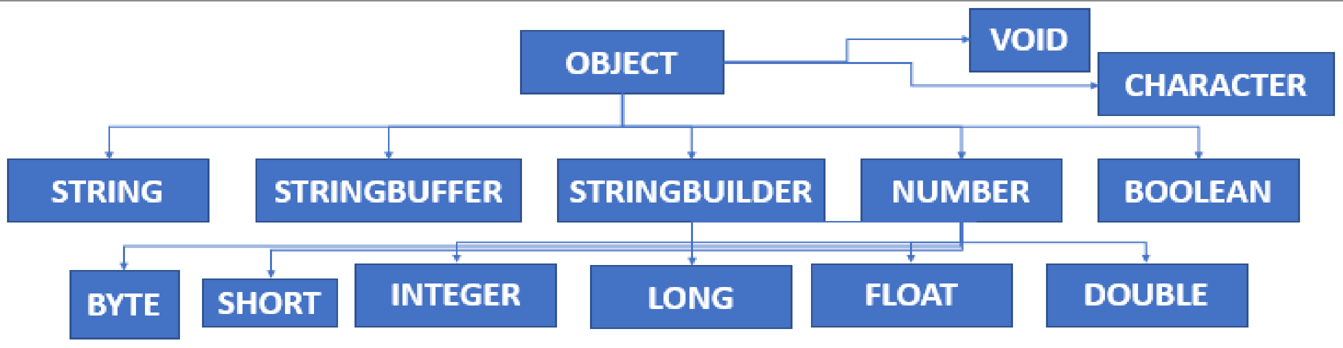
“public static String toHexString(primitive p)”

Wrapper Object/primitive -----toString() method------ String

DANCING BETWEEN STRING, WRAPPER OBJECT AND PRIMITIVE:



PARTIAL HIRARCHY OF JAVA.LANG PACKAGE:



The wrapper classes which are not child class of number are BOOLEAN & CHARACTER.

The wrapper classes which are not direct child class of Object are BYTE, SHORT, INTEGER, LONG, FLOAT, DOUBLE.

String , StringBuffer, StringBuilder and all wrapper classes are final classes.

In addition to String Objects all wrapper class objects are also immutable.

Some times void class is also considered as wrapper class.

Void class:

It is a final class and it is the direct child class of object. it doesn't contain any methods and it contains only one variable "Void.type". in general we can use void class in reflections to check whether the method return type is void or not.

"If(getMethod(m).getReturnType() == Void.type){ }"

Void is the class representation of void keyword in java.

AUTOBOXING AND AUTOUNBOXING: java 1.5 v

AUTOBOXING:

Automatic conversion of premitive to wrapper object by compiler is called autoboxing.

Example:

Integer i = 10; -> compiler converts int to Integer automatically by autoboxing.

After compilation the above example will become "Integer I = Integer.valueOf(10);" that is internally autoboxing concept is implemented by using valueOf() methods.

AUTOUNBOXING:

Automatic conversion of wrapper object to primitive by compiler is called autounboxing.

Example:

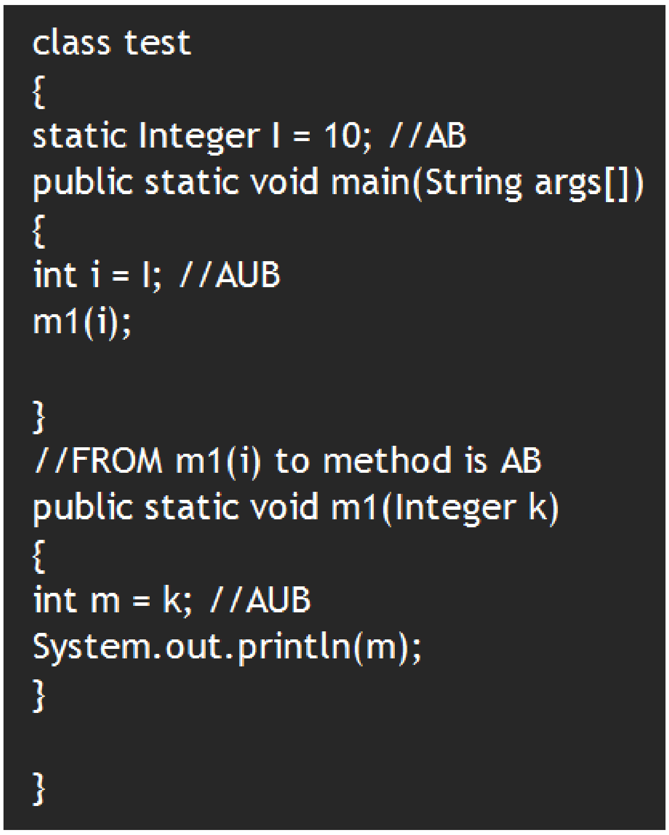
Integer I = new Integer(10);

int i = I; -> compiler converts Integer to int automatically by autounboxing.

After compilation the above line will become "int i= I.intValue();" that is internally autounboxing concept is implemented by using xxxValue() methods.

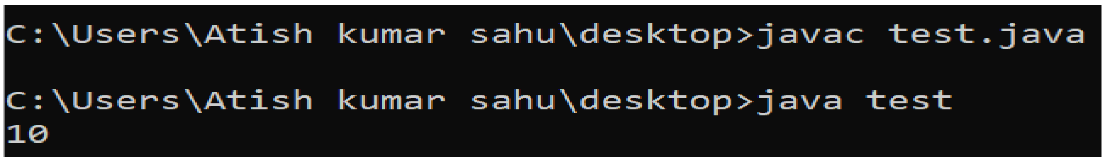
Primitive value ------autoboxing -> valueOf() method--------- wrapper object

Wrapper object --------autounboxing -> xxxValue() method-------- primitive value



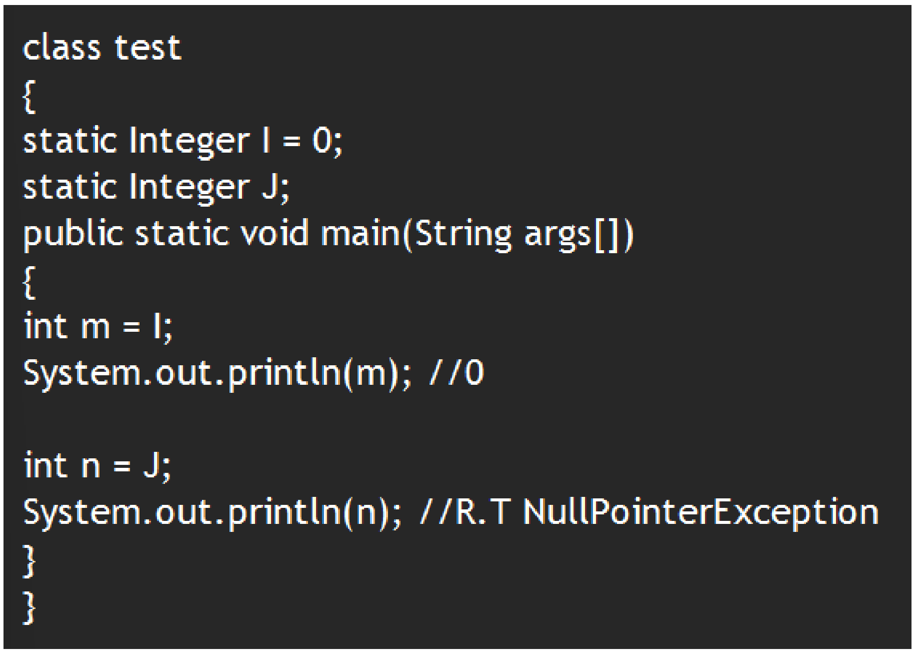
```
class test
{
static Integer I = 10; //AB
public static void main(String args[])
{
int i = I; //AUB
m1(i);

}
//FROM m1(i) to method is AB
public static void m1(Integer k)
{
int m = k; //AUB
System.out.println(m);
}

}
```

valid in 1.5 version invalid in 1.4 version.

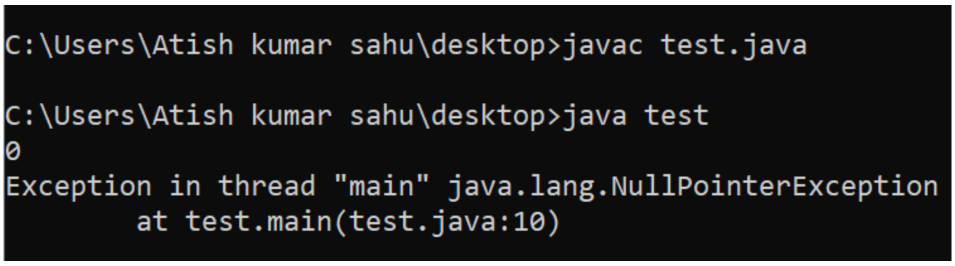


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
10
```

Just because of autoboxing and autounboxing we can use primitive and wrapper objects interchangeable from 1.5 version onwards.

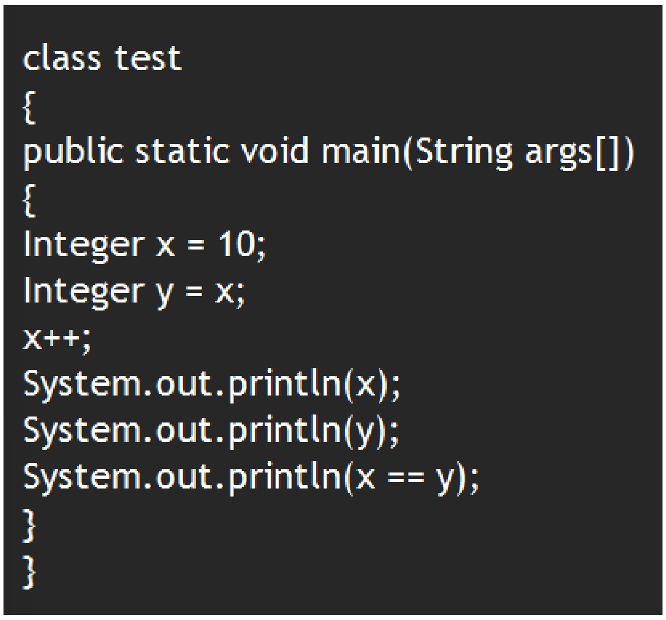


```
class test
{
static Integer I = 0;
static Integer J;
public static void main(String args[])
{
int m = I;
System.out.println(m); //0

int n = J;
System.out.println(n); //R.T NullPointerException
}
}
```



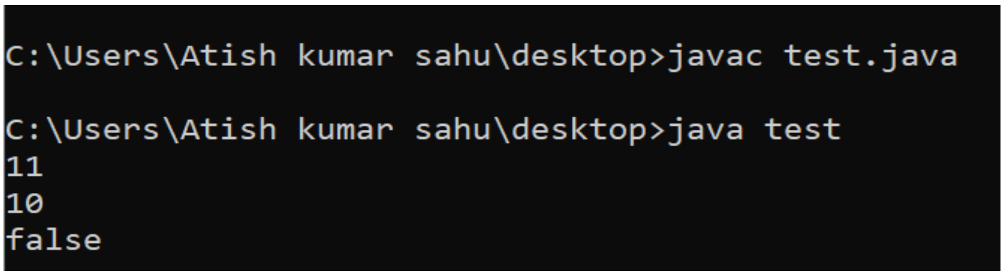
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
0
Exception in thread "main" java.lang.NullPointerException
        at test.main(test.java:10)
```

On null reference if we are trying to perform autounboxing then we will get runtime exception saying “NullPointerException”.

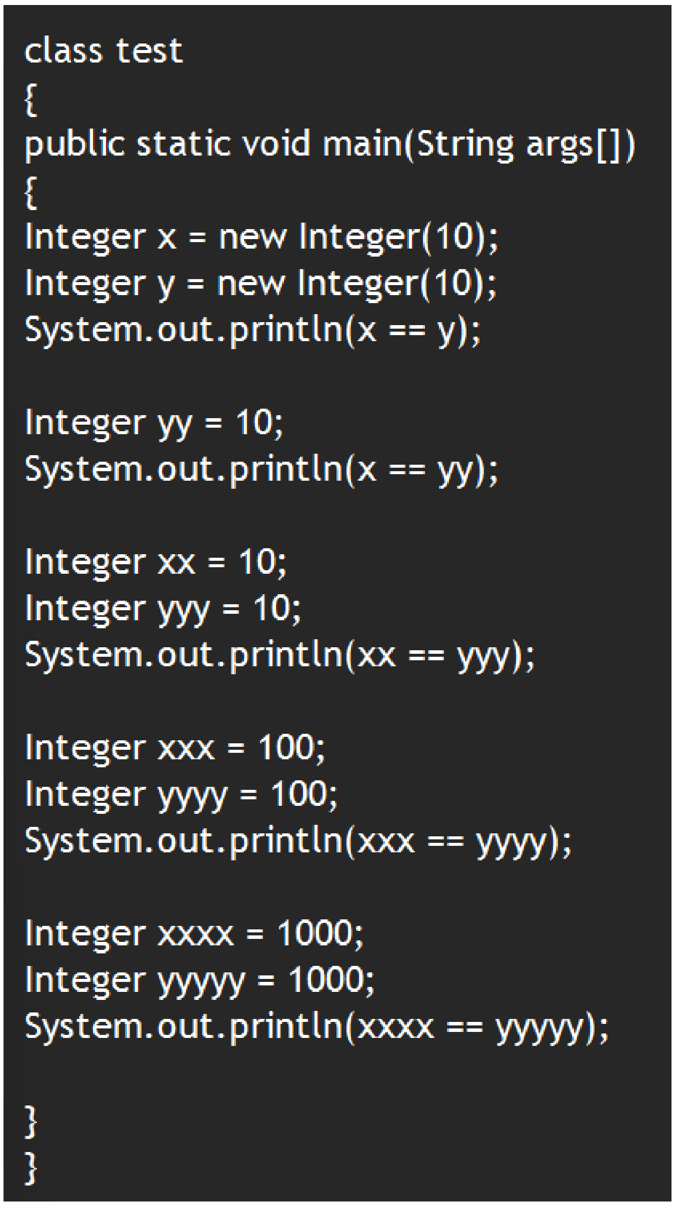


```
class test
{
public static void main(String args[])
{
Integer x = 10;
Integer y = x;
x++;
System.out.println(x);
System.out.println(y);
System.out.println(x == y);
}
}
```

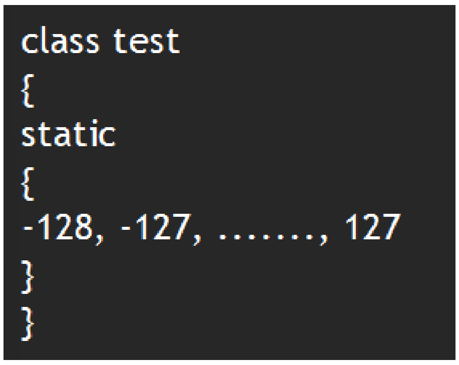
ALL WRAPPER CLASS OBJECTS ARE IMMUTABLE THAT IS ONCE WE CREATE WRAPPER CLASS OBJECT WE CAN’T PERFORM ANY CHANGES IN THAT OBJECT. IF WE ARE TRYING TO PERFORM ANY CHANGES WITH THOSE CHANGES A NEW OBJECT WILL BE CREATD.



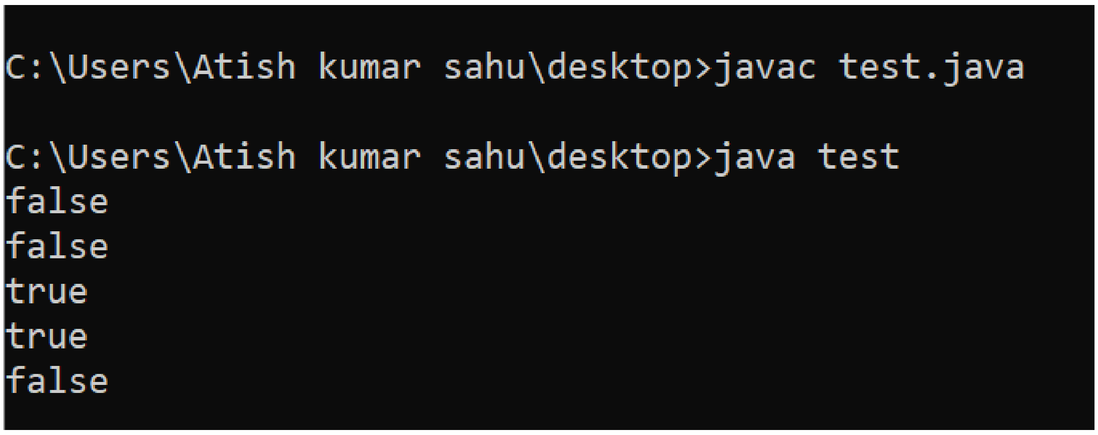
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
11
10
false
```



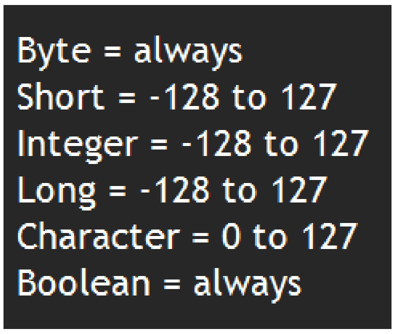
```
class test
{
public static void main(String args[])
{
Integer x = new Integer(10);
Integer y = new Integer(10);
System.out.println(x == y);

Integer yy = 10;
System.out.println(x == yy);

Integer xx = 10;
Integer yyy = 10;
System.out.println(xx == yyy);

Integer xxx = 100;
Integer yyyy = 100;
System.out.println(xxx == yyyy);

Integer xxxx = 1000;
Integer yyyyy = 1000;
System.out.println(xxxx == yyyyy);

}
}
```



```
class test
{
static
{
-128, -127, ......., 127
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
false
false
true
true
false
```



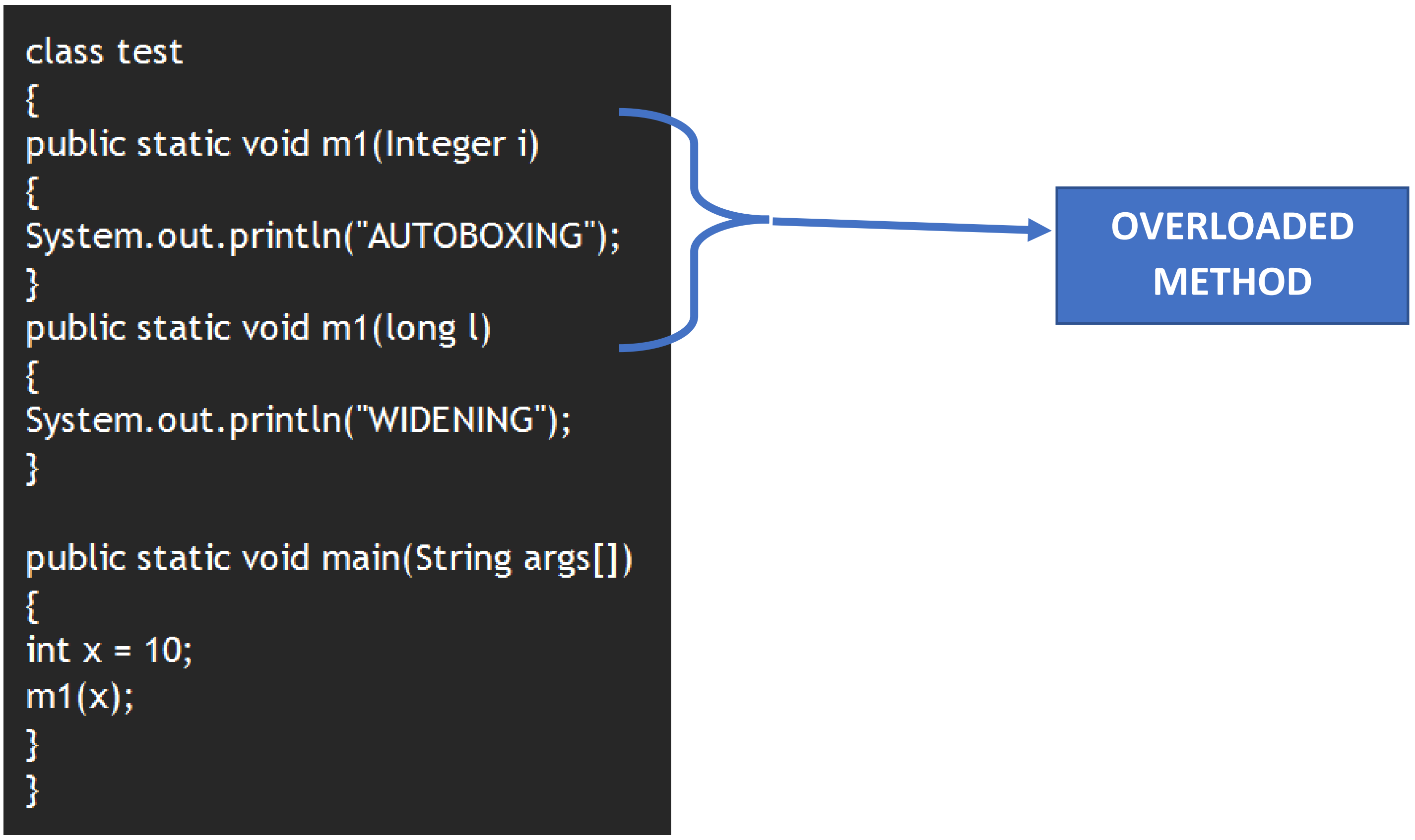
```
Byte = always
Short = -128 to 127
Integer = -128 to 127
Long = -128 to 127
Character = 0 to 127
Boolean = always
```

Internally to provide support for autoboxing a buffer of wrapper object will be created at the time of wrapper class loading. By autoboxing if an object is required to create first JVM will check whether this object already present in the buffer or not if it is already present in the buffer then existing buffer object will be used. If it is not already available in the buffer then JVM will create a new object. but buffer concept is available only in the following ranges. Except this range in all remaining cases a new object will be created.

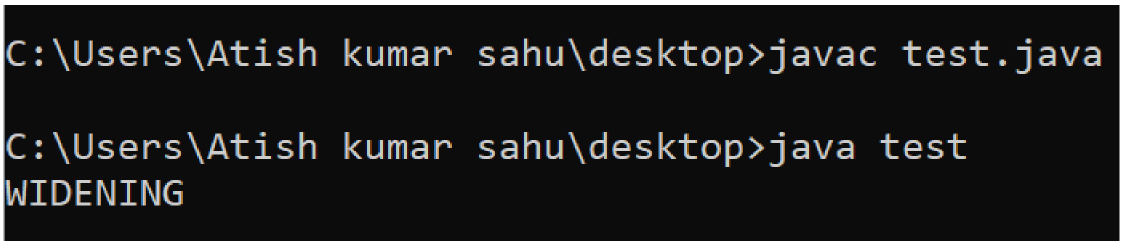
Internally autoboxing concept is implemented by using valueOf() methods. Hence buffering concept is applicable for valueOf() methods also.

<u>OVERLOADING WITH RESPRCT TO AUTOBOXING, WIDENING, VAR-ARG METHOD:</u>

<u>CASE 01: AUTOBOXING VS WIDENING:</u> Widening dominates autoboxing.

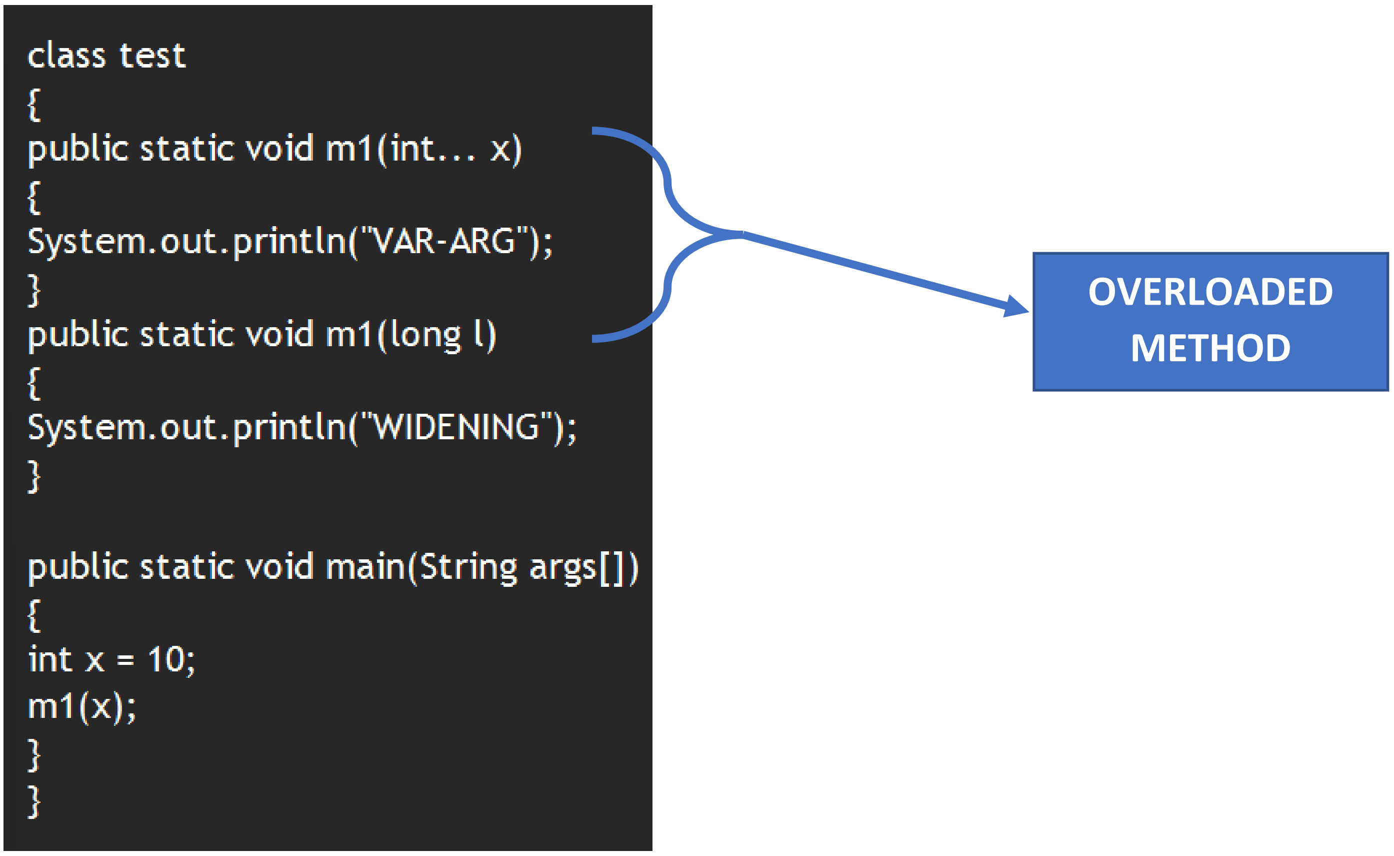


```
class test
{
public static void m1(Integer i)
{
System.out.println("AUTOBOXING");
}
public static void m1(long l)
{
System.out.println("WIDENING");
}

public static void main(String args[])
{
int x = 10;
m1(x);
}
}
```

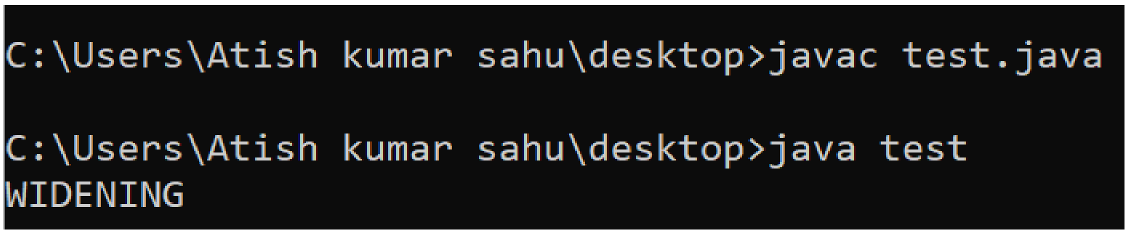


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
WIDENING
```

<u>CASE 02: WIDENING VS VAR-ARG METHOD:</u> widening dominates var-arg method.

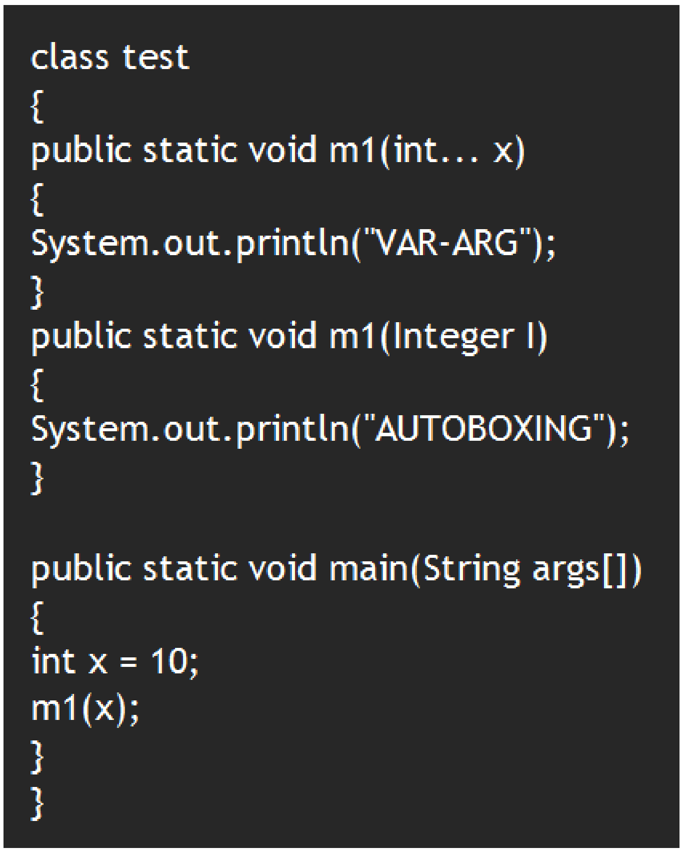


```
class test
{
public static void m1(int... x)
{
System.out.println("VAR-ARG");
}
public static void m1(long l)
{
System.out.println("WIDENING");
}

public static void main(String args[])
{
int x = 10;
m1(x);
}
}
```

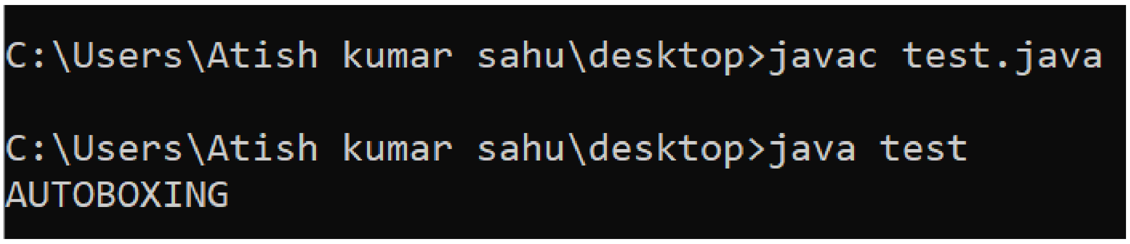


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
WIDENING
```

CASE 03: AUTOBOXING VS VAR-ARG METHOD:



```
class test
{
public static void m1(int... x)
{
System.out.println("VAR-ARG");
}
public static void m1(Integer I)
{
System.out.println("AUTOBOXING");
}

public static void main(String args[])
{
int x = 10;
m1(x);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
AUTOBOXING
```

Autoboxing dominates var-arg method. in general var-arg method will get least priority that is if no other method matched then only var-arg method will get the chance. it is exactly same as default case inside switch.

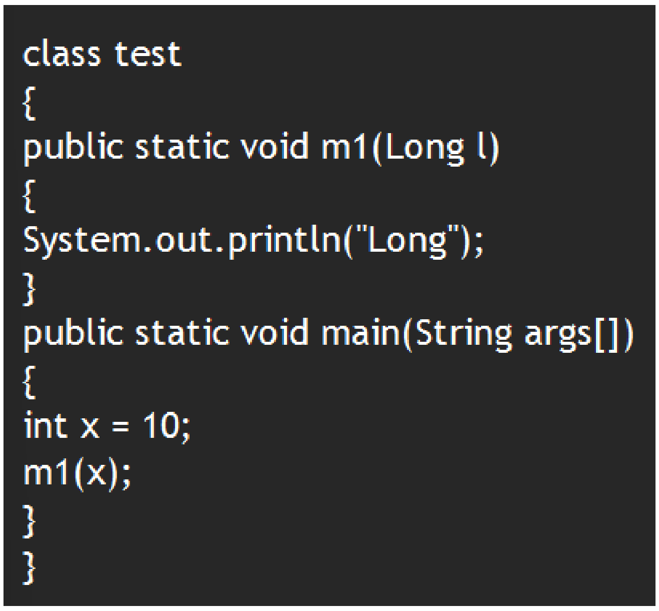
***** while resolving overloaded methods compiler will always gives precedence in the following order.

1. widening

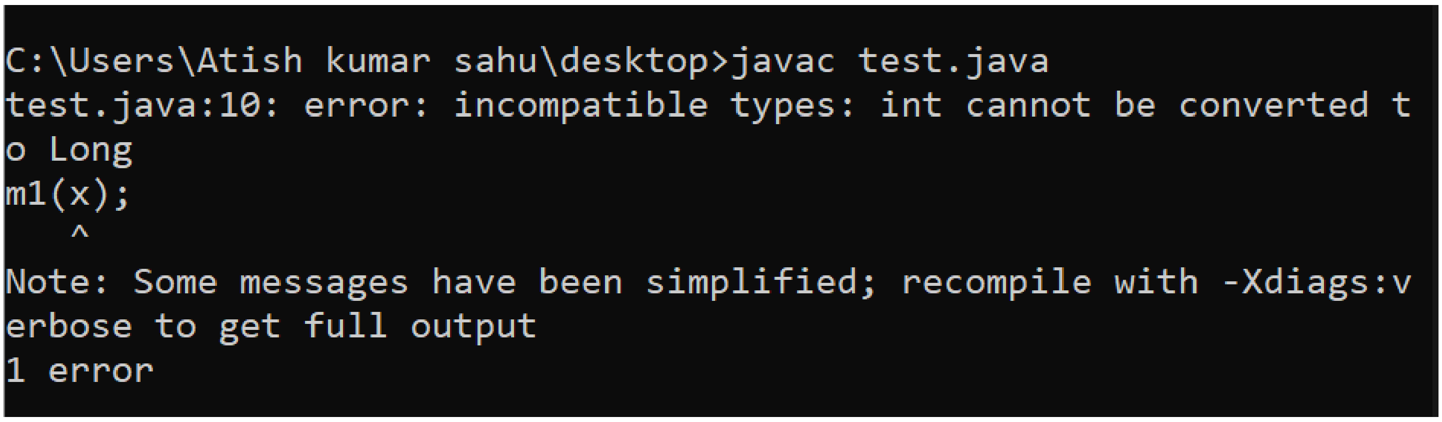
2. autoboxing

3. var-arg method

CASE 04:



```
class test
{
public static void m1(Long l)
{
System.out.println("Long");
}
public static void main(String args[])
{
int x = 10;
m1(x);
}
}
```



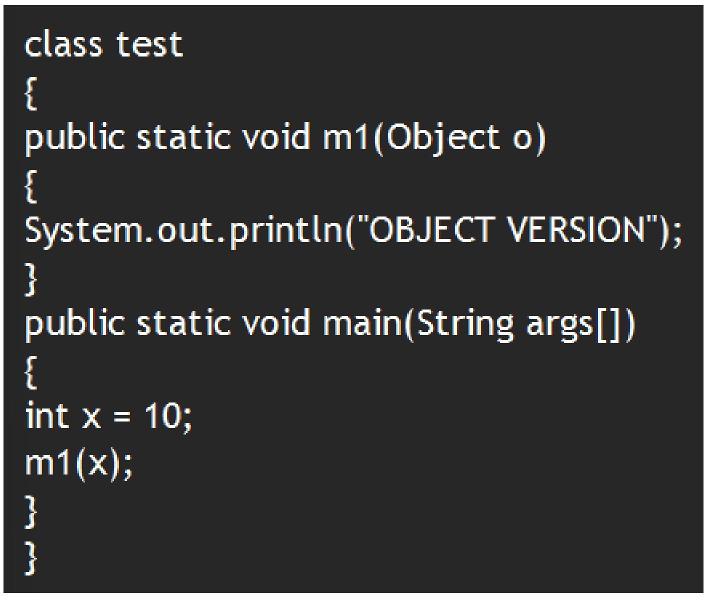
```
C:\Users\Atish kumar sahu\desktop>javac test.java
test.java:10: error: incompatible types: int cannot be converted t
o Long
m1(x);
   ^
Note: Some messages have been simplified; recompile with -Xdiags:v
erbose to get full output
1 error
```

Widening followed by autoboxing is not allowed in java. where as autoboxing followed by widening is allowed.

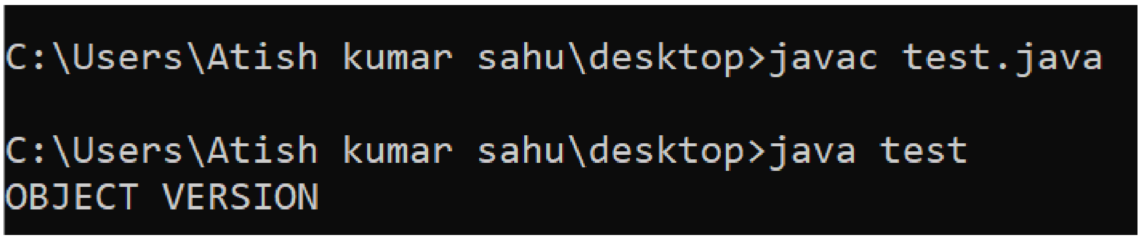
Long l = 10; -> COMPILE TIME ERROR

long l = 10; -> CORRECT METHOD

CASE 05:



```
class test
{
public static void m1(Object o)
{
System.out.println("OBJECT VERSION");
}
public static void main(String args[])
{
int x = 10;
m1(x);
}
}
```

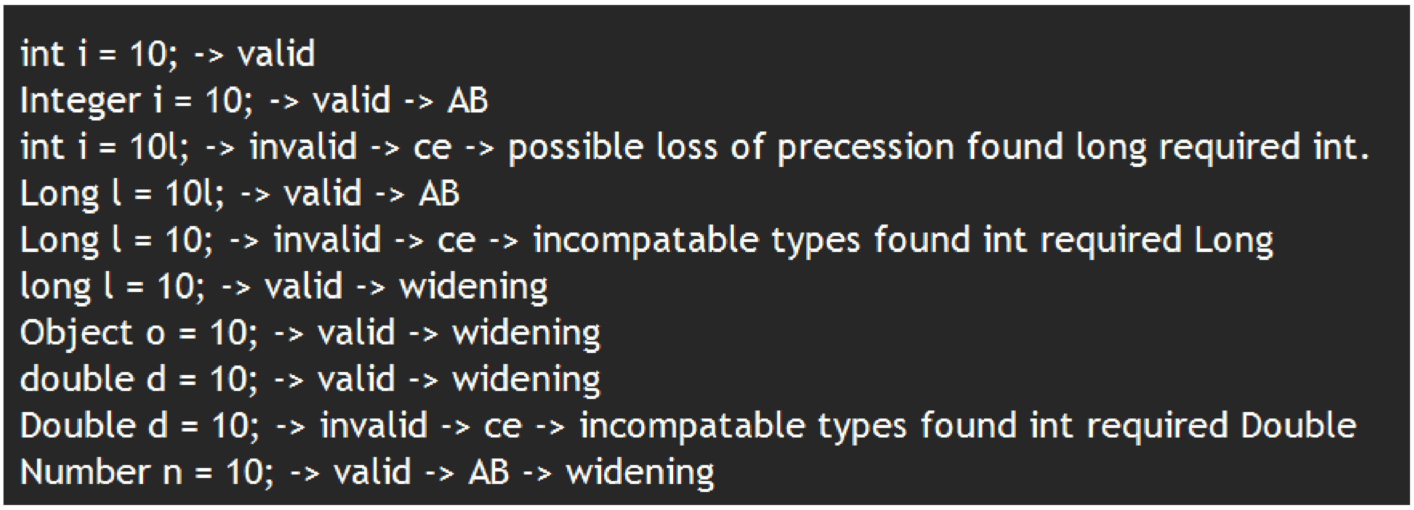


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
OBJECT VERSION
```

Object o = 10; -> valid statement

Number n = 10; -> valid statement

Q. WHICH OF THE FOLLOWINGS ARE LEGAL?



```
int i = 10; -> valid
Integer i = 10; -> valid -> AB
int i = 10l; -> invalid -> ce -> possible loss of precession found long required int.
Long l = 10l; -> valid -> AB
Long l = 10; -> invalid -> ce -> incompatable types found int required Long
long l = 10; -> valid -> widening
Object o = 10; -> valid -> widening
double d = 10; -> valid -> widening
Double d = 10; -> invalid -> ce -> incompatable types found int required Double
Number n = 10; -> valid -> AB -> widening
```

RELATION BETWEEN (==) OPERATOR AND EQUAL() METHODS:

1. if two objects are equal by (==) operator then these objects are always equal by equals() method. that is if r1 == r2 is true, then r1.equals(r2) is always true.

2. if two objects are not equal by (==) operator then we can't conclude anything about equals() method. it may returns true or false. That is if r1 == r2 is false, then r1.equals(r2) may returns true or false. We can't expect exactly.

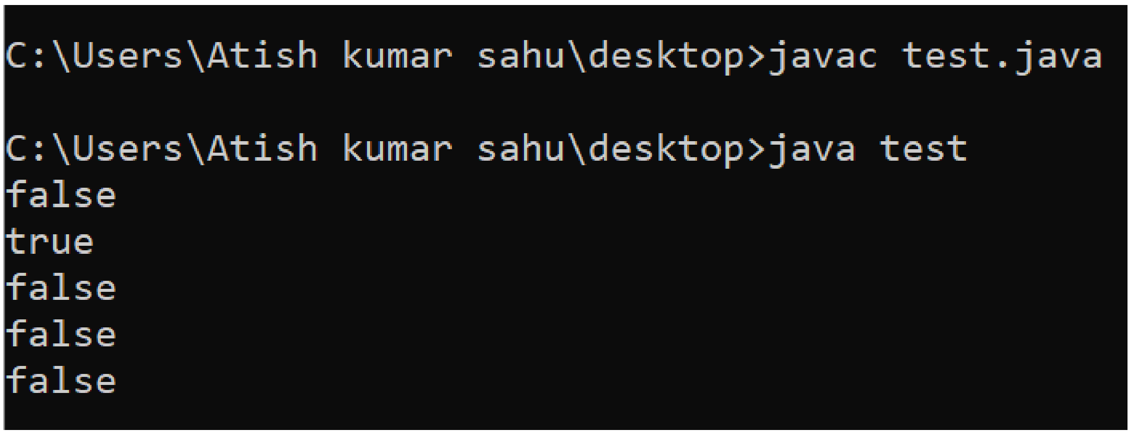
3. if two objects are equal by equals() method then we can't conclude anything about (==) operator. It may returns true or false. That is if r1.equals(r2) is true then we can't conclude anything about r1 == r2. It may returns true or false.

4. if two objects are not equal by equals() then these objects are always not equal by (==) operator. If r1.equals(r2) is false then r1 == r2 is always false.

DIFFERENCES BETWEEN (==) OPERATOR AND EQUALS():

To use (==) operator compulsory there should be some relation between argument types (either child to parent, parent to child, same type) otherwise we will get compile time error saying "incomparable types". if there is no relation between argument types then equals() method won't rise any compile time and runtime errors and simply it returns false.

```
class test
{
public static void main(String args[])
{
String s1 = new String("ABCD");
String s2 = new String("ABCD");

StringBuffer sb1 = new StringBuffer("ABCD");
StringBuffer sb2 = new StringBuffer("ABCD");

System.out.println(s1 == s2);
System.out.println(s1.equals(s2));

System.out.println(sb1 == sb2);
System.out.println(sb1.equals(sb2));

//System.out.println(s1 == sb1);
/*error: incomparable types: String and StringBuffer System.out.println(s1 == sb1);*/
System.out.println(s1.equals(sb1));
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
false
true
false
false
false
```

1. (==) operator is an operator in java applicable for both primitives and object types.

equals() method is applicable only for object types but not for primitives.

2. In the case of Object references (==) operator ment for reference comparison(address comparison).

Bydefault equals() method present in object class also ment for reference reference comparison.

3. We can’t override (==) operator for content comparison.

We can override equals() method for content comparison.

4. To use (==) operator compulsory there should be some relation between argument types(either child to parent, parent to child, or same type) otherwise we will get compile time error saying incomparable types.

If there is no relation between argument types then equlas() method won’t rise any compile time or runtime error and simply returns false.

***** In general we can use (==) operator for reference comparison and equals() method for content comparison. *****

***** for any object reference “r” r == null & r.equals(null) always returns false.

Example: Thread t = new Thread(); SOP(t == null); SOP(t.equals(null)); -> false; *****

Hashing related data structure follow the following fundamental rule. 2 equivalent objects should be placed in same bucket but all objects present in the same bucket need not be equal.

CONTRACT BETWEEN EQUALS() METHODS AND HASHCODE():

1. if two objects are equal by equals() method then their hashcode must be equal. That is two equivalent objects should have same hashcode. That is if “r1.equals(r2)” then “r1.hashcode() == r2.hashcode()” is always true.

2. object class equlas() and hashcode() method follows above contract. Hence whenever we are overriding equals() method compulosry we should override hashcode() method to satisfy above contract. (2 equivalent object should have same hashcode()).

3. If 2 objects are not equal by equals() method there is no restriction on hashcode may be equal or may not be equal.

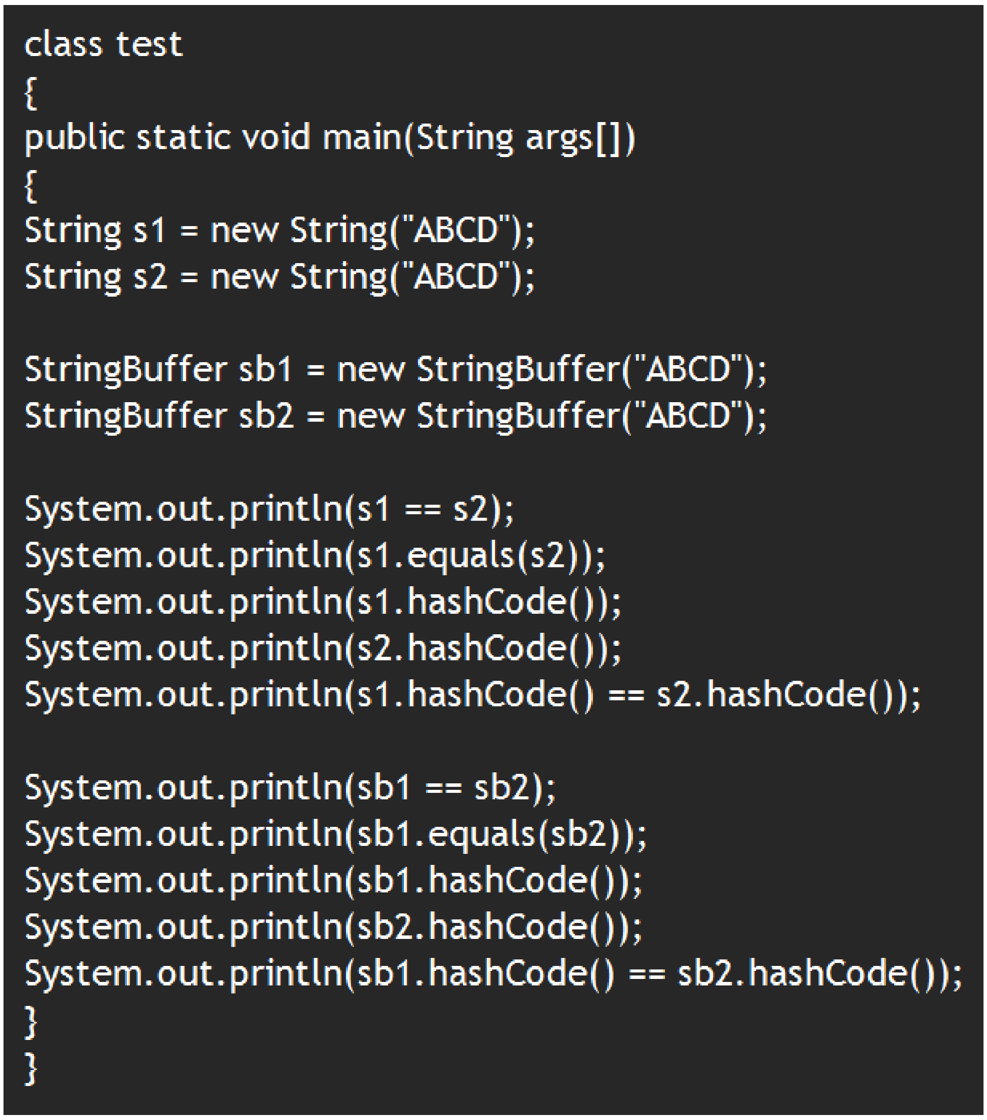
4. If hashcodes of 2 objects are equal then we can’t conclude anything about equals() method. it may returns true or false.

5. If hashcode of 2 objects are not equal then these objects are always not equal by equals() method.

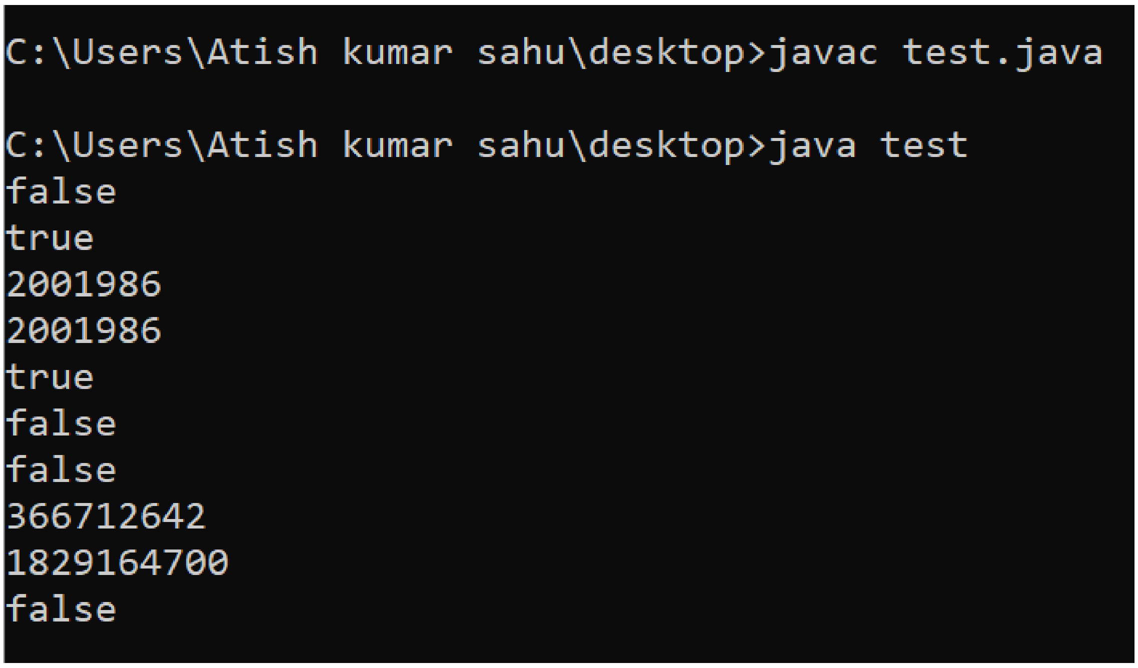
***** to satisfy contract between equals() and hashcode() whenever we are overriding equals() method compulsory we have to override hashcode() method. otherwise we won’t get any cmoplie time and runtime error. but it is not a good programming practice. *****

In string class equals() method is overriden for content comparison and hence hashcode method is also overriden to generate hashcode based on content.

In stringBuffer equals() method is not overriden for content comparison nad hence hashcode() method is also not overriden.



```
class test
{
public static void main(String args[])
{
String s1 = new String("ABCD");
String s2 = new String("ABCD");

StringBuffer sb1 = new StringBuffer("ABCD");
StringBuffer sb2 = new StringBuffer("ABCD");

System.out.println(s1 == s2);
System.out.println(s1.equals(s2));
System.out.println(s1.hashCode());
System.out.println(s2.hashCode());
System.out.println(s1.hashCode() == s2.hashCode());

System.out.println(sb1 == sb2);
System.out.println(sb1.equals(sb2));
System.out.println(sb1.hashCode());
System.out.println(sb2.hashCode());
System.out.println(sb1.hashCode() == sb2.hashCode());
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
false
true
2001986
2001986
true
false
false
366712642
1829164700
false
```

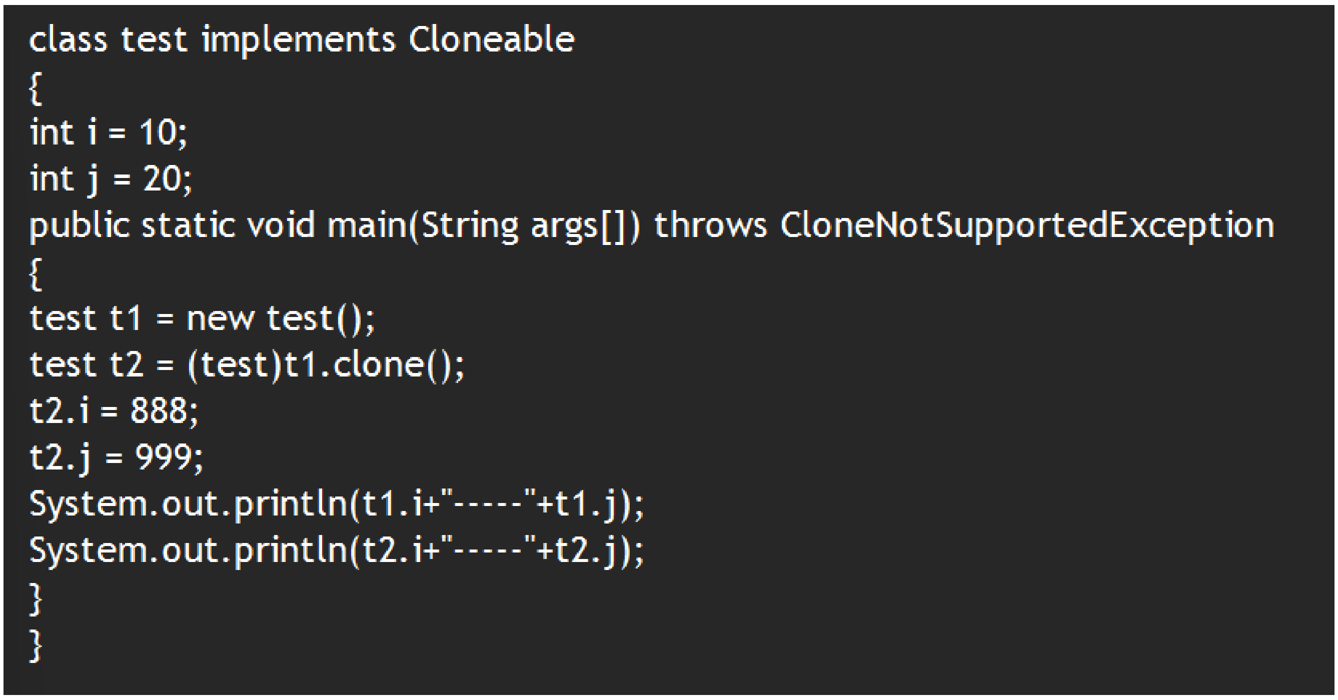
Based on which parameter we override equals() method, it is highly recommended to use same parameter while overriding hashcode() method also.

In all collection classes, in all wrapper classes and in String class equals() method is overriden for content comparison. Hence it is highly recommended to override equals() method in our class also for content comparison.

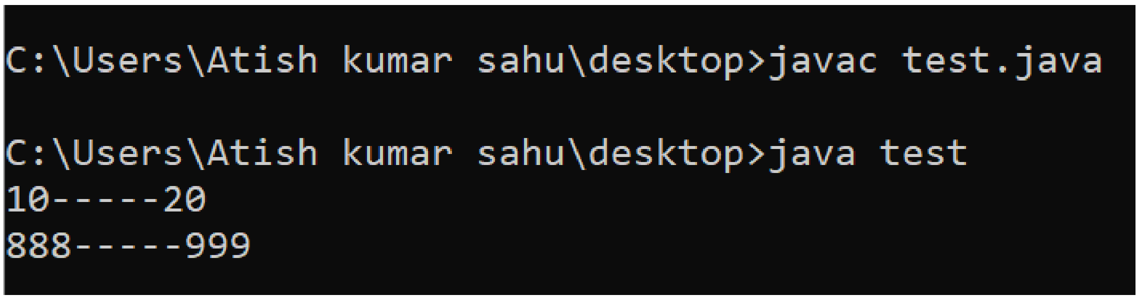
CLONE():

The process of creating exactly duplicate object is called cloning. The main purpose of cloning is to maintain backup copy and to preserve state of an object. we can perform cloning by using clone() method of Object class.

"protected native Object clone() throws CloneNotSupportedException"



```
class test implements Cloneable
{
int i = 10;
int j = 20;
public static void main(String args[]) throws CloneNotSupportedException
{
test t1 = new test();
test t2 = (test)t1.clone();
t2.i = 888;
t2.j = 999;
System.out.println(t1.i+"-----"+t1.j);
System.out.println(t2.i+"-----"+t2.j);
}
}
```

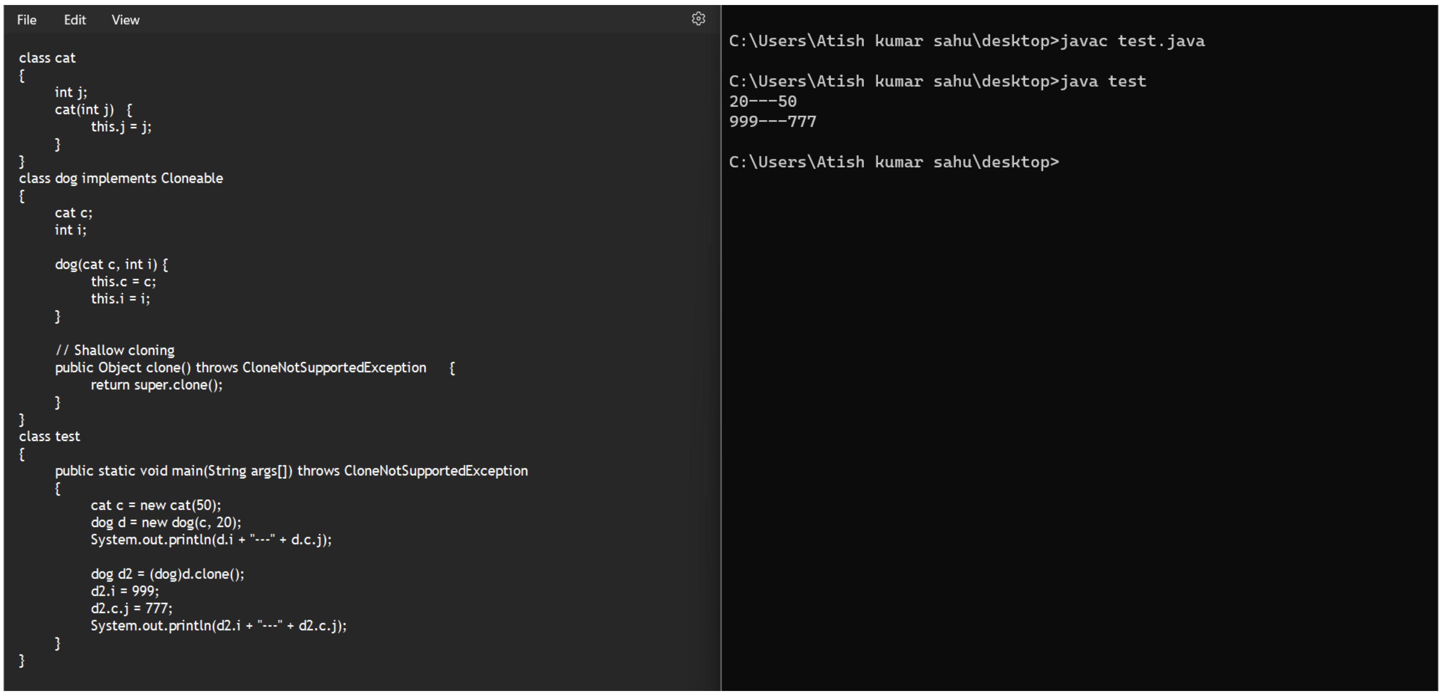


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
10-----20
888-----999
```

We can perform cloning only for cloneable objects. An object is said to be cloneable if and only if the corresponding class implements cloeable interface. Cloneable interface present in java.lang package and it doesn't contain any methods. It is a marker interface. If we are trying to perform cloning for non-cloneable objects then we will get runtime exception saying "CloneNotSupportedException".

SWALLOW CLONING VS DEEP CLONING:

The process of creating bitwise copy of an object is called shallow cloning. if the main object contain primitive variables then exactly duplicate copies will be created in the cloned object. if the main object contain any reference variable then corresponding object won't be created just duplicate reference variable will be created pointing to old contained object. Object class clone method meant for shallow cloning.



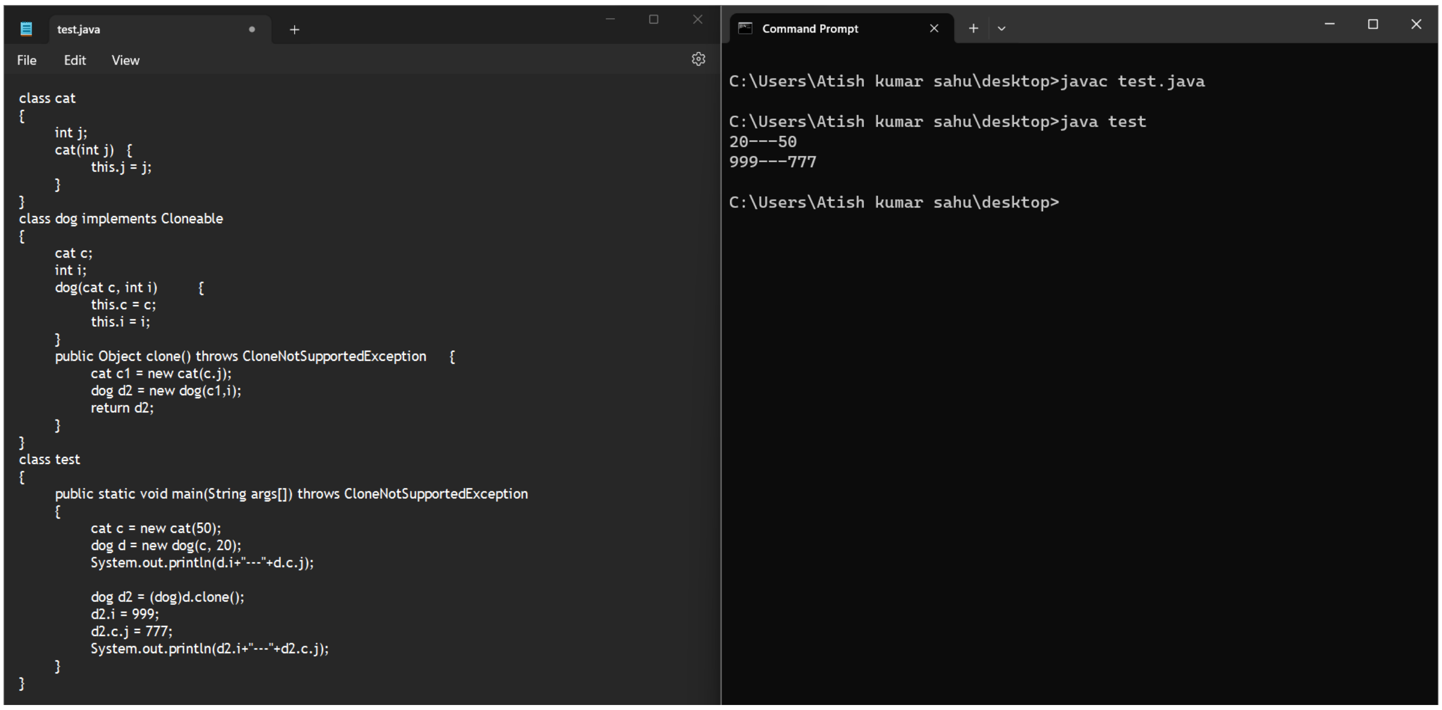
```
class cat
{
	int j;
	cat(int j)  {
		this.j = j;
	}
}
class dog implements Cloneable
{
	cat c;
	int i;

	dog(cat c, int i) {
		this.c = c;
		this.i = i;
	}

	// Shallow cloning
	public Object clone() throws CloneNotSupportedException    {
		return super.clone();
	}
}
class test
{
	public static void main(String args[]) throws CloneNotSupportedException
	{
		cat c = new cat(50);
		dog d = new dog(c, 20);
		System.out.println(d.i + "---" + d.c.j);

		dog d2 = (dog)d.clone();
		d2.i = 999;
		d2.c.j = 777;
		System.out.println(d2.i + "---" + d2.c.j);
	}
}
```

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
20---50
999---777

C:\Users\Atish kumar sahu\desktop>
```

In shallow cloning by using cloned object reference if we perform any change to the contained object then those changes will be reflected to main object. to overcome this problem we should go for deep cloning.

DEEP CLONING:

The process of creating exactly duplicate independent copy including contained Object is called deep cloning. in deep cloning if the main object contain any primitive variables then in the clone object duplicate copy will be created if main object contain any reference variable then the coressponding contained object also will be created in the cloned. By default object class clone method meant for shallow cloning but we can implement deep cloning explicitly by overriding clone method in our class.



```
class cat
{
	int j;
	cat(int j)  {
		this.j = j;
	}
}
class dog implements Cloneable
{
	cat c;
	int i;
	dog(cat c, int i)	{
		this.c = c;
		this.i = i;
	}
	public Object clone() throws CloneNotSupportedException    {
		cat c1 = new cat(c.j);
		dog d2 = new dog(c1,i);
		return d2;
	}
}
class test
{
	public static void main(String args[]) throws CloneNotSupportedException
	{
		cat c = new cat(50);
		dog d = new dog(c, 20);
		System.out.println(d.i+"---"+d.c.j);

		dog d2 = (dog)d.clone();
		d2.i = 999;
		d2.c.j = 777;
		System.out.println(d2.i+"---"+d2.c.j);
	}
}
```

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
20---50
999---777

C:\Users\Atish kumar sahu\desktop>
```

By using cloned object reference if we perform any change to the contained object then those changes won’t be reflected to the main object.

Q. WHICH CLONING IS BEST? If object contains only primitive variables then shallow cloning is the best choice. If the object contains reference variables then deep cloning is the best choice.