Java Main Revision Points(Part 2)

Types of Variables

There are 2 types of variables

a. Based on the type of value variable holds

i)- primitive type(holds primitive values)

ii)- reference type(holds the address of the objects, or these varaible are used to refer the objects)

b. Based on the behaviour and position of its declaration

i)- instance variable

ii)- static variable

iii)- local variable

i)- instance variable

1)- instance variables are created at the time of object creation and destroyed at the time of object destruction(GC)

2)- instance variables will be stored in the heaparea of the object[seperate copy for each object].

3)- instance varaibles can be accessed directly from instance area in instance method.

4)- instance variables are accessed through reference from static area in static method.

eg#1.

```
class Test
{
    boolean isMarried;
    //Pre-Defined Method[Entry point/Driving Code]
    public static void main(String[] args)
    {
        System.out.println(isMarried);  //CE(point 3)
        System.out.println(new Test().isMarried);  //false(point 4)
    }
}
```

eg#2.

```
class Test
{
    //Array declaration
    int[] arr;
    //Pre-Defined Method[Entry point/Driving Code]
    public static void main(String[] args)
```

```
    {
        System.out.println(new Test().arr);//null
        System.out.println(new Test().arr[0]);  //NPE
    }
}
```

ii)- static variable

1)- static variables are created at the time of loading the .class file and destroyed at the time of unloading the .class file.

2)- static variables will be stored in the methodArea[Common copy for all the objects of the class].

3)- static varaibles can be accessed directly inside instance or static area.

4)- static variables should be accessed using classname or object name,but good practise is through "classname".

In Java, when a class is loaded into memory, static variables are initialized with default values even if you do not explicitly assign values to them. The default values for static variables in Java depend on their data types. Here are the default values for common data types:

Based on the behaviour and its position of declaration

1. **instance variables.[jvm will give default value]**
2. **static variables. [jvm will give default value]**
3. **local variables. [jvm will not give, programmer should initialise before using]**

local variables

=> These variables are also called as "Temporary variables/stack variables".

=> Local variables will be created during the method execution inside stack and they will be destoryed once the control comes out of method execution.

=> Scope of local varible is limited only to that method or to that block, if we try to access them outside the block or method it would result in "CompileTime Error".

intializing it is mandatory, otherwise it would result in "CompileTimeError".

Note:

1)- It is not a convention to initialize the local variable inside the block, because there is no gaurantee that a block will be executed and it will be initialized at runtime.

2)- It is suggestible to initialize the local variable at the time of declaration itself with a default value depending upon the datatype.

eg#1.

class Test

```
{
    //Pre-Defined Method[Entry point/Driving Code]
    public static void main(String[] args)
    {
        int x ;
        if(args.length > 0 )
            x = 10;
        System.out.println(x);//CE
    }
}
```

eg#2.

```
class Test
{
    //Pre-Defined Method[Entry point/Driving Code]
    public static void main(String[] args)
    {
        int x ;
        if(args.length > 0 )
            x = 10;
        else
            x = 20;
        System.out.println(x); //will work fine as per the value of args
    }
}
```

How many access modifers are there in java?

public, private, protected

static, strictfp, synchronized

final, abstract, native

transient, volatile

Note: The only access modifier which is applicable at the local variable level is "final",if we try to use any other access modifier it would result in "CompiletimeError".

Note: combination of varibles can be of the following types

- **a. instance** -> primitive, reference
- **b. static** -> primitive, reference
- **c. local** -> primitive, reference

eg:

```
class Test
{
    int[] a =new int[3]; //instance-reference
    static int x= 100; //static-primitive
    //Pre-Defined Method[Entry point/Driving Code]
    public static void main(String[] args)
    {
        String name = "sachin";  //local-reference
    }
}
```

Polymorphism

1. **Overloading**

   => Two or more methods are said to overloaded methods iff they have the same methodname but change in the argument types.

   => In c language we did'nt had this Overloading concept so as a C programer for same tasks with different argument types, programmer should remembe mulitple method names like

   a. abs() -> for int type

   b. labs() -> for long type

   c. fabs() -> for float type

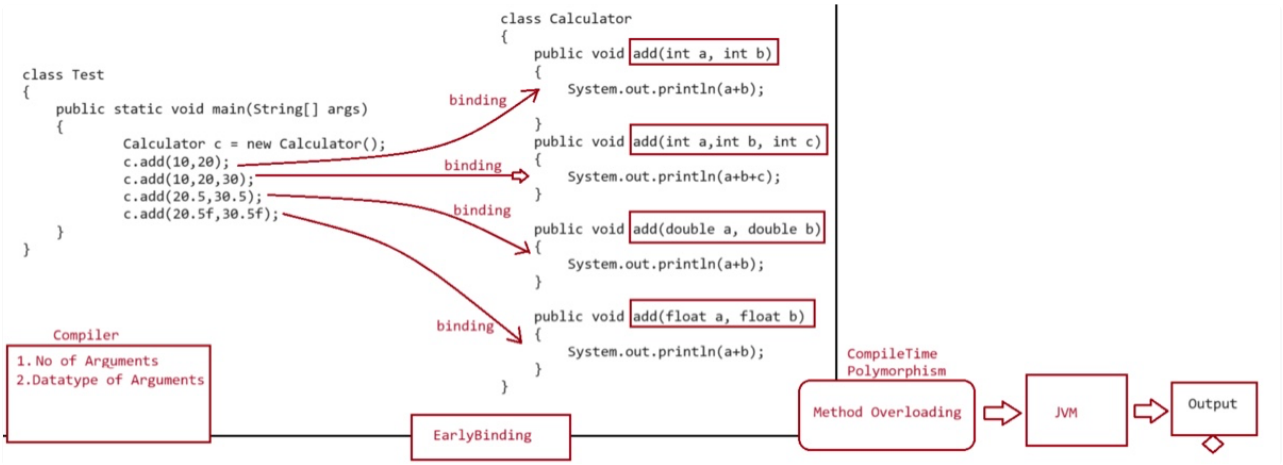
   ;;;

   => Remembering mulitple method names was challenging for developers.

   => In java we have concept of Overloading, where we can write one method name for same tasks with different argument types.

   => overloading concept reduces the complexity of programming in java.

   => Overloading refers to "CompileTime Polymorphism".



Conclusion: In overloading compiler is responsible for method resolution(binding) based on the reference type, so we say overloading as "CompileTime Polymorphism/Eager Binding".

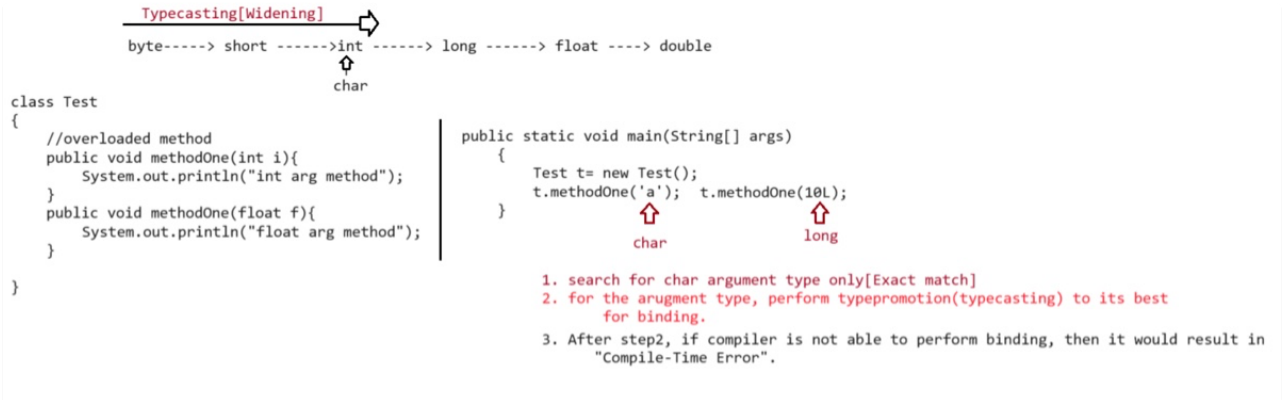
TypePromotion in Overloading

```
public class Test
{
    //overloaded method
    public void methodOne(int i){
        System.out.println("int arg method");
    }
    public void methodOne(float f){
        System.out.println("float arg method");
    }
    public static void main(String[] args)
    {
        Test t= new Test();
        t.methodOne('a');//int arg method
        t.methodOne(10L);//float arg method
        t.methodOne(19.5);//CE
    }
}
```

eg#2.

```
class Test
{
    //Overloaded method
    public void methodOne(int i)
    {
        System.out.println("int arg method");
    }
    public void methodOne(Integer i)
    {
        System.out.println("Integer version");
    }
    public void methodOne(Character c)
    {
        System.out.println("Character version");
    }
    public static void main(String[] args)
    {
        Test t= new Test();
        t.methodOne('a');
```

```
    }
}
```



// primtiive -----> Exact Match(primitive) -----> typecasting -----> bind

// primitive ------> no exact match -> no typecasting -----> Wrapper classes ---→ bind

eg#3.

```
class Test
{
    //Overloaded method
    public void methodOne(String s)
    {
        System.out.println("String version");
    }
    public void methodOne(StringBuilder sb)
    {
        System.out.println("StringBuilder version");
    }
    public void methodOne(Object o)
    {
        System.out.println("Object version");
    }
    public static void main(String[] args)
    {
        Test t= new Test();
        t.methodOne("sachin");//String version
        t.methodOne(new Object());//Object version
        t.methodOne(new StringBuilder("sachin"));//StringBuilder version
        //StringBuilder(child) -> null String(child) -> null , Object(Parent) --> null
        t.methodOne(null);
    }
```

}

Note: While resolving Overloaded methods exact match will get high priority. While resolving Overloaded methods,child class will get more priority than the parent class. While resolving Overloaded methods, if both the class are from child level only, then it would result in "Compiletime Error".

Method Overloading

=> Two or more methods is said to be overloaded method, iff both the methods have same name,but change in the argument datatype.

=> Order of binding the method calls would be based on

a. exact match

b. type promotion

=> In case of reference type, first priority is for child and then for parent.

=> In case of Overloading,binding the method call will be done by the compiler based on the reference type.

=> It is also called as "CompileTime Polymorphism/Early Binding".

eg#1.

```
class Test
{
    public void m1(int i,float f)
    {
        System.out.println("int-float arg method");
    }
    public void m1(float f,int i)
    {
        System.out.println("float-int arg method");
    }
    public static void main(String[] args)
    {
        Test t= new Test();
        t.m1(10,10.5f);//int-float
        t.m1(10.5f,10);//float-int
        t.m1(10,10); //CE: ambigous
        t.m1(10.5f,10.5f);//CE: can't find symbol
    }
}
```

Var-Args(Variable no of arguments method)

-> Untill 1.4V we can't declare a method with variable no of arguments

-> if there is a change in no of arugments compulsorily we have to define a new method

-> This approach increases the length of the code and reduces the readability.

-> From 1.5V version onwards we can declare a method with variable no of arugmentssuch type of methods are calle d as "VAR-ARGS METHODS".

-> Syntax:

```
public XXXXX methodName(XXXX... varaible)
{
}
```

eg#1.

```
class Test{
    public void m1(int... data)
    {
        System.out.println("Var-Arg method");
    }
    public static void main(String[] args) {
        Test t= new Test();
        t.m1();
        t.m1(10);
        t.m1(10,20);
        t.m1(10,20,30);
        t.m1(10,20,30,40);
    }
}
```

Note: Internally var-arg parameter is implemented by using "1-D Array",so var-arg parameter can be accessed through index.

Case1:

Which of the following var-arg declarations are valid?

methodOne(int... x) //valid [recomended]

methodOne(int ...x) //valid

methodOne(int...x) //valid

methodOne(int x...) //invalid

methodOne(int. ..x) //invalid

methodOne(int .x..) //invalid

Case2: we can mix var-args with normal parmeters also, and normal parameter can be different type and var-arg can be different type.

eg:: m1(int a, int... arr)

m1(String name, int... arr)

```
class Test{
    public void m1(String name, int... arr){
        System.out.println(name);
        System.out.println(arr);
    }
    public static void main(String[] args) {
        Test t= new Test();
        t.m1("sachin",20,30,40);
    }
}
```

Output

```
D:\OctBatchMicroservices>javac Test.java
D:\OctBatchMicroservices>java Test
sachin
[I@76ed5528
```

Case3: We can mix var-arg parameter with normal parameter,but in the parameter list the var-arg parameter should be at the last.

```
class Test{
    public void m1(int... arr, int data ){
        System.out.println(data);
        System.out.println(arr);
    }
    public static void main(String[] args) {
        Test t= new Test();
        t.m1(10,20,30,40);
    }
}
```

Output

```
D:\OctBatchMicroservices>javac Test.java
Test.java:2: error: varargs parameter must be the last parameter
public void m1(int... arr, int data ){
```

Case4: In a paramter list,we can have only var-arg parameters,more than one results in "CompiletimeError".

```
class Test{
    public void m1(int... arr1, int... arr2 ){
        System.out.println(arr1);
        System.out.println(arr2);
    }
    public static void main(String[] args) {
        Test t= new Test();
        t.m1(10,20,30,40);
    }
}
```

Output

D:\OctBatchMicroservices>javac Test.java

Test.java:2: error: varargs parameter must be the last parameter

public void m1(int... arr1, int... arr2){

Case5: In general var-arg method will get least priority that is if no other methods are available to bind only then var-arg method will get a chance for binding. This is just like "default" statement in switch case.

```
class Test{
    public void m1(int... arr ){// arr-> 0,1,.... n
        System.out.println("var-arg method");
    }
    public void m1(int i){// i -> 1
        System.out.println("one-arg method");
    }
    public static void main(String[] args) {
        Test t= new Test();
        t.m1(10,20,30,40);
        t.m1(10);
        t.m1();
    }
}
```

output

D:\OctBatchMicroservices>javac Test.java

D:\OctBatchMicroservices>java Test

var-arg method

one-arg method

var-arg method

Case6: For the var-args method we can provide the corresponding type array as argument

```
class Test{
    public void m1(int... arr ){//int[] :: arr-> 0,1,.... n
        System.out.println("var-arg method");
    }
    public static void main(String[] args) {
        Test t= new Test();
        t.m1(10,20,30,40);
        t.m1(new int{100,200,300,400});
    }
}
```

output

D:\OctBatchMicroservices>javac Test.java

D:\OctBatchMicroservices>java Test

var-arg method

var-arg method

Case7:

```
class Test{
    public void m1(int... arr ){//m1(int[]) :: arr-> 0,1,.... n
        System.out.println("Var-arg method");
    }
    public void m1(int[] arr) //m1(int[])
    {
        System.out.println("Array-arg method");
    }
    public static void main(String[] args) {
        Test t= new Test();
        t.m1(10,20,30,40);
        t.m1(new int{100,200,300,400});
    }
}
```

output

D:\OctBatchMicroservices>javac Test.java

Test.java:5: error: cannot declare both m1(int[]) and m1(int...) in Test

public void m1(int[] arr)//m1(int[])

^

1 error

Case 8: wherever there is [] array,we can replace it with "..." also to provide the arguments in flexible manner(var-args,arrays)

Case 9: Wherever there is ..., if we repalce it "[]", we don't get flexibility to provide the arguments in var-args and arrays style.

eg::

```
class Test{
    public void m1(int... arr ){// arr-> 0,1,.... n
        System.out.println("Var-arg method");
    }
    public static void main(String[] args) {
        Test t= new Test();
        t.m1(10,20,30,40);
        t.m1(new int{100,200,300,400});
    }
}
```

output

D:\OctBatchMicroservices>javac Test.java

D:\OctBatchMicroservices>java Test

Array-arg method

Array-arg method

eg::

```
class Test{
    public void m1(int[] arr ){// arr-> 0,1,.... n
        System.out.println("Array-arg method");
    }
    public static void main(String[] args) {
        Test t= new Test();
        t.m1(10,20,30,40);
        t.m1(new int{100,200,300,400});
    }
}
```

D:\OctBatchMicroservices>javac Test.java

Test.java:7: error: method m1 in class Test cannot be applied to given types;

t.m1(10,20,30,40);

eg::

```
class Test{
    public static void main(String... args) {
        System.out.println("Var-Arg main method");
    }
}
```

output

D:\OctBatchMicroservices>javac Test.java

D:\OctBatchMicroservices>java Test

Var-Arg main method

Case 8:

```
class Test{
    public void m1(int[]... twoDarr){
        System.out.println(twoDarr);
        for (int[] oneDArr: twoDarr )
        {
            for (int data: oneDArr )
            {
                System.out.print(data+"\t");
            }
            System.out.println();
        }
    }
    public static void main(String... args) {
        Test t =new Test();
        int[] arr1 = {10,20,30};
        int[] arr2 = new int{100,200,300};
        t.m1(arr1,arr2);
    }
}
```

//m1(int... x) ===> arguments var-args,array ---> x []

//m1(int[]... x) ===> arguments 1D-array var-args ----> x [][]

Output

D:\OctBatchMicroservices>javac Test.java

D:\OctBatchMicroservices>java Test

[[I@76ed5528

10 20 30

100 200 300

Rule1: Binding of method call will happen based on the reference, not on the runtime object

```
class Animal{}
class Monkey extends Animal{}
class Test{
    public void talk(Monkey m){
        System.out.println("Monkey version");
    }
    public void talk(Animal a){
        System.out.println("Animal version");
    }
    public static void main(String... args) {
        Test t = new Test();
        Animal a =new Animal();
        t.talk(a);//Animal version

        Monkey m =new Monkey();
        t.talk(m);//Monkey version

        Animal a1= new Monkey();// a1 -> Animal type(compiler will bind) a1→ Monkey(runtime object::JVM)
        t.talk(a1);  //Animal version
    }
}
```

Output

D:\OctBatchMicroservices>javac Test.java

D:\OctBatchMicroservices>java Test

Animal version

Monkey version

Animal version

Can we have normal method with the name same as classname and also constructor?

Ans. Yes, it is possible, but the constructor will be called during the creation of object where as normal method should be called by the programmer explicitly.

Can we Overload main()?

Ans. yes we can overload main(),but jvm will always call main() with the following signature

public static void main(String[] args).

static

=> This access modifier is appicable at

a. class

b. variable

c. method

d. block

variable :

1)- memory for static variables will be give in "MethodArea".

2)- if we don't initialize the static variables, then memory for static variables will be taken care by "JVM".

method :

if we mark a method as static, then those methods can be called in 2 ways.

a. using ClassName(best practise)

b. using objectName

block :

1)- we can mark a block with static access modifier.

2)- This block is mainly meant for "initializing the static variables".

3)- This block will be executed only once, so we normally keep "Driving code" in static block.

Read Indirect Write Only (RIWO)

Direct Read => Within a static block, if we are reading a variable then such type of read is called as "Direct read".

Indirect Read => If we are calling a static method,and with in the static method if we are reading a static variable then such type of read is called "Indirect Read".

If a variable is in RIWO state ,then we can't perform direct read operation, if we try to do it would result in CompileTime Error.

```
class Test
{
  1 static int i = 10; 7
  2 static
    {
        methodOne(); 8
        System.out.println("First static block");10
    }
  3 public static void main(String[] args)
    {
        methodOne(); 13
        System.out.println("Inside main method"); 15
    }
  4 public static void methodOne()
    {
        System.out.println(j); 9 , 14
    }
  5 static
    {
        System.out.println("Second static block");11
    }
  6 static int j = 20; 12
}
Output
  0
First static block
Second static block
  20
Inside main method
```

Steps followed by JVM

1. Identification of static members from top to bottom

   i = 0
   j = 0 | RIWO — Read Indirect Write only — Step 1 to 6

2. execution of static varaibles assignments and static block execution from top to bottom

   i = 10
   j = 20 | R&W — Read and Write only — Step 7 to 12

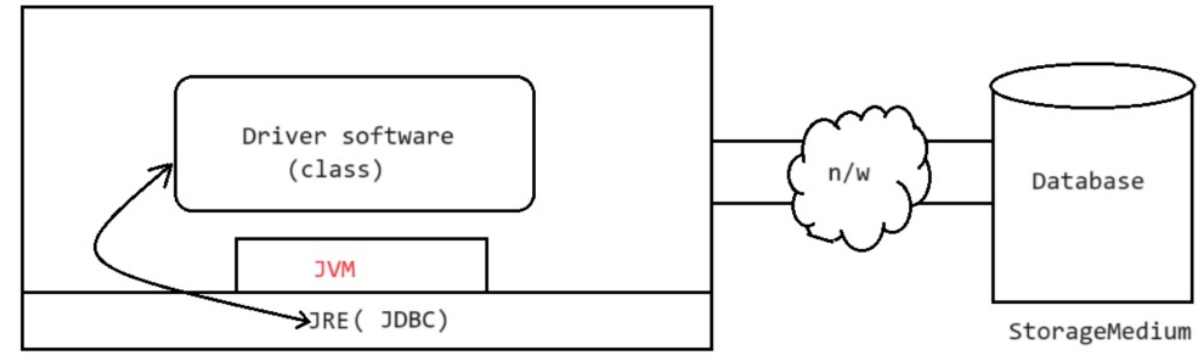
3. Execution of main method — Step 13 to 15

Usage of static block in realtime

static block : It is called as "One Time Execution block". It will be executed during the loading of .class file.

Note1:

Every driver software internally contains static block to register the driver with DriverManager, which helps the programmer to get JDBC environment in JRE, to do this we need static block.



Capable of setting environment for java program execution

```
class Driver
{
    static
    {
       //Register the driver with DriverManager(Setting JDBC environment)
    }
}
```

Note2:

Can we write any statements to the console without writing statement inside main()?

Answer : yes , by using static block.

eg#1.(in normal method)

```
class Test
{
    static int i = methodOne();
    public static void main(String[] args)
    {
    }
    public static int methodOne()
    {
        System.out.println("Hello i can print");
        System.exit(0);
        return 10;
    }
}
```

Output

Hello i can print

eg#2.(in constructor with static variable)

```
class Test
{
    static Test t = new Test();
    public Test()
    {
        System.out.println("Hello i can print");
        System.exit(0);//shutdown jvm
    }
    public static void main(String[] args)
    {
    }
}
```

Output

Hello i can print

Note: It is mandatory to write main() inside every class for the execution to happen, if we don't write then class will not be loaded.

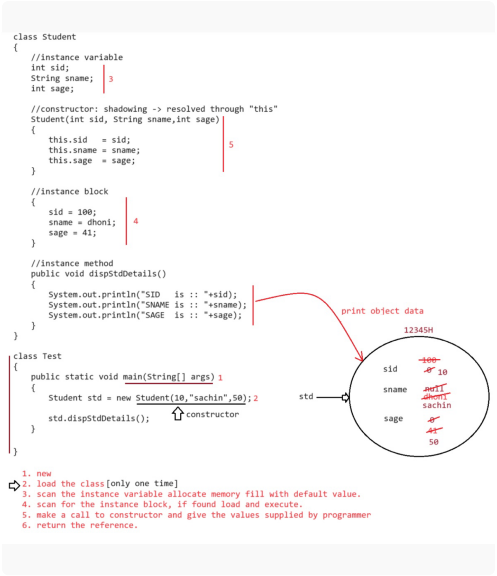
System.exit(0) -> This line if jvm executes, then jvm will shutdown itself, by skipping the remaining statements in the program.

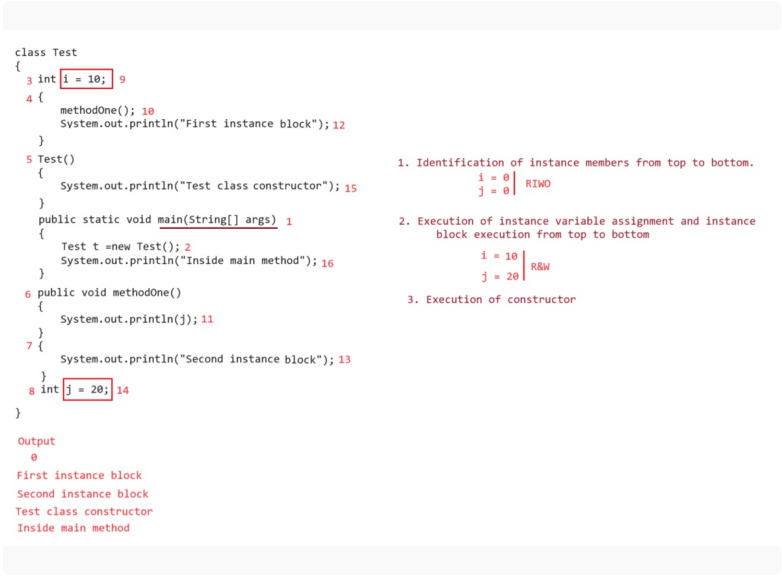
Instance control flow

instance block : This block gets executed at the time of creating an object,but before the call to constructor. This block will be executed for every object we create,but before the call to a constructor.

Note:

1. **Constructor :** initialize the object with the required values.
2. **instance block :** Apart from initalization, if we want to perform any other activities then we need to go for "instance block".
3. **Order of execution :** first instance block then constructor.





Write a program to count the no of objects created for a class?

i)- //instance block

{

count++;

}

ii)- constructor with var-args

Test(int... args){ **//var-args**

count++;

}

final variables

a. final instance variable

b. final static variable

c. final local varaible

Note: A variable is said to be final iff the value of the variable is "fixed", we can't change the value once it is initialized. If a variable is final, then compiler will get to know the value of the variable and these values will be used during the evaluation of an Expression.

If an instance variable is marked as final, then for such instance variables, jvm will not give default value, programmer should supply the value for instance variables, otherwise it would result in "CompileTimeError".

eg#2.

```
class Test
{
    final int i;
    public static void main(String[] args)
    {
        System.out.println(new Test().i);
    }
}
```

Output

CompileTimeError:: variable i not initialized in the default constructor

Places to initalize the value for final instance variable

1. **At the time of declaration**

eg#1

```
class Test
{
    final int i=100;
    public static void main(String[] args)
    {
        System.out.println(new Test().i);//100
    }
}
```

2. **Inside instance block**

eg#1

```
class Test
{
    final int i;
```

```
    {
        i = 100;
    }
    public static void main(String[] args)
    {
        System.out.println(new Test().i);//100
    }
}
```

3. **Inside constructor**

```
eg#1
class Test
{
    final int i;
    Test()
    {
        i = 100;
    }
    public static void main(String[] args)
    {
        System.out.println(new Test().i);//100
    }
}
```

Apart from these 3 places, if we try to do initalization then it would result in "CompiletimeError".

static variable

eg#1.

If an static variable is marked as final, then for such static variables, jvm will not give default value, programmer should supply the value for static variables, otherwise it would result in "CompileTimeError".

eg#2.

```
class Test
{
    final static int i;
    public static void main(String[] args)
    {
        System.out.println(i);
    }
```

```
}
```

Output : CE: variable i not initialized in the default constructor

Places to initialize final static variable

1. **At the time of declaration**

eg#1.

```
class Test
{
    final static int i=100;
    public static void main(String[] args)
    {
        System.out.println(i);//100
    }
}
```

2. **Inside static block**

eg#2.

```
class Test
{
    final static int i;
    static
    {
        i = 100;
    }
    public static void main(String[] args)
    {
        System.out.println(i);//100
    }
}
```

If a variable is marked as static and final, then for those variables initialization should be completed before class loading completes,otherwise it would result in "CompileTimeError".

final local variable

eg#1.

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

```
        int i;
        System.out.println(i);
    }
}
```

Output: error: variable i might not have been initialized

eg#2.

```
class Test
{
    public static void main(String[] args)
    {
        final int i;
        System.out.println(i);
    }
}
```

Output: error: variable i might not have been initialized

The only place where we can initialize the local variable is at the time of declaration.

eg#1.

```
class Test
{
    public static void main(String[] args)
    {
        final int i=100;
        System.out.println(i);//100
    }
}
```

eg#2.

```
class Test
{
    public static void main(String[] args)
    {
        methodOne(100,200);
    }
    public static void methodOne(final int i, int j)
    {
        i = 1000;
        j = 2000;
```

```
        System.out.println(i + " " + j);
    }
}
```

Output :: error: final parameter i may not be assigned i = 1000;

Note: The only access modifier applicable at local variable level is "final".

Inheritance

=> In java inheritance can be promoted in 2 ways

a. IS-A relationship

b. HAS-A relationship

=> IS-A relationship is referred to "Inheritance".

methodTwo → child specific method, methodOne→ parent method which is inherited to child

```
    public static void main(String[] args)
    {
        Parent p= new Parent();
        p.methodOne();
        p.methodTwo(); //CE: can't find symbol

        System.out.println();

        Child c = new Child();
        c.methodOne();
        c.methodTwo();

        System.out.println();

        Parent p1 =new Child();
        p1.methodOne();
        p1.methodTwo(); //CE: can't find symbol

        System.out.println();

        Child c1 = new Parent();  //incompatible types
    }
```

Note:

1)- Parent class reference can be used to collect child class objects, but by using parent class reference we can call only parent class methods but not child class specialized methods.

2)- Child class reference can't be used to hold parent class objects.

3)- Whatever the parent class has under public category(variables, methods, etc.), by default will be available to child class.

4)- Whatever the child class has under public category(variables, methods, etc.), by default won't be available to the parent class.

Note:

1. For all java classes,, the most commonly requried functionality is defined inside object class so Object class is called as "root" for all java classes.
2. For all java exceptions and errors, the most commonly required functionality is defined inside "Throwable" class, so Throwable class acts as root for "Exception Hierarchy".
3. **For compiler and jvm by default all the classes available in a package/folder called " java.lang.* " will available.**

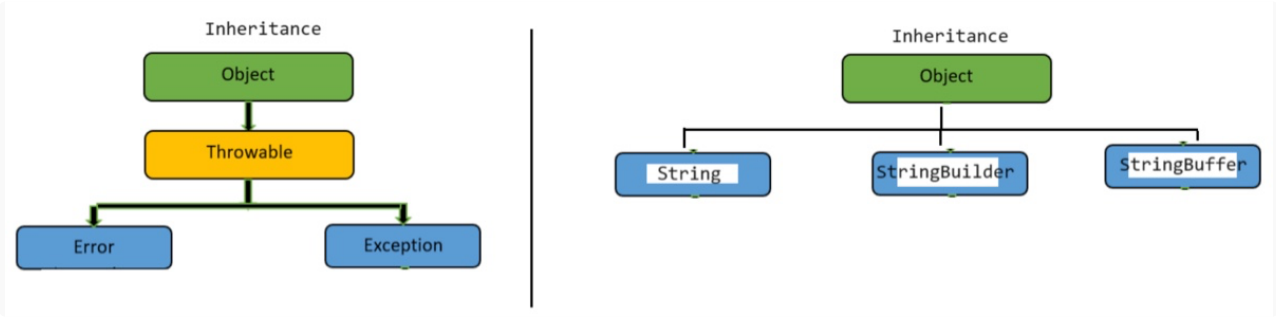
```
class Object
{
    public final native java.lang.Class<?> getClass();
    public native int hashCode();
    protected native java.lang.Object clone() throws
    java.lang.CloneNotSupportedException;

    //used while working with String,StringBuilder,StringBuffer
    public boolean equals(java.lang.Object);

    //toString() gets called automatically when we print the reference of the object[callback method/magic method]
    public java.lang.String toString();

    //Methods related to MultiThreading
    public final native void notify();
    public final native void notifyAll();
    public final void wait() throws java.lang.InterruptedException;
    public final native void wait(long) throws java.lang.InterruptedException;
    public final void wait(long, int) throws java.lang.InterruptedException;
    //Method related to GarbageCollector
    protected void finalize() throws java.lang.Throwable;
}
```

```
class String extends Object
{
    //String class specific methods
    @Override
    public String toString()
    {
        //print the content of the Strings
    }
}
class StringBuffer extends Object
{
    //StringBuffer class specific methods
    @Override
    public String toString()
    {
        //print the content of the Strings
    }
}
class StringBuilder extends Object
{
    //StringBuilder class specific methods
    @Override
    public String toString()
    {
        //print the content of the Strings
    }
}
System.out.println(str);    //str.toString():: sachin
Student std = new Student(10,"sachin");
System.out.println(std);    //std.toString():: hashCode
System.out.println();
String str = new String("sachin");
System.out.println(str);    //str.toString():: sachin
```




```
//Default Constructor
Farmer()
{
   super();
}
```

During Compilation it will be added by compiler

```
}
```

Note::

1)- When an object of parent class is created ,the parent class constructor will be called.

2)- When an object of child class is created ,both parent and child class constructor will be called because of super().

Snippet

1)-

```
public class Boxer1{
    Integer i;
    int x;
    public Boxer1(int y) {
        x = i+y;   // x = null + 4
        System.out.println(x);
    }
    public static void main(String[] args) {
        new Boxer1(new Integer(4));
    }
}
```

What is the result?

A. The value "4" is printed at the command line.

B. Compilation fails because of an error in line 5.

C. Compilation fails because of an error in line 9.

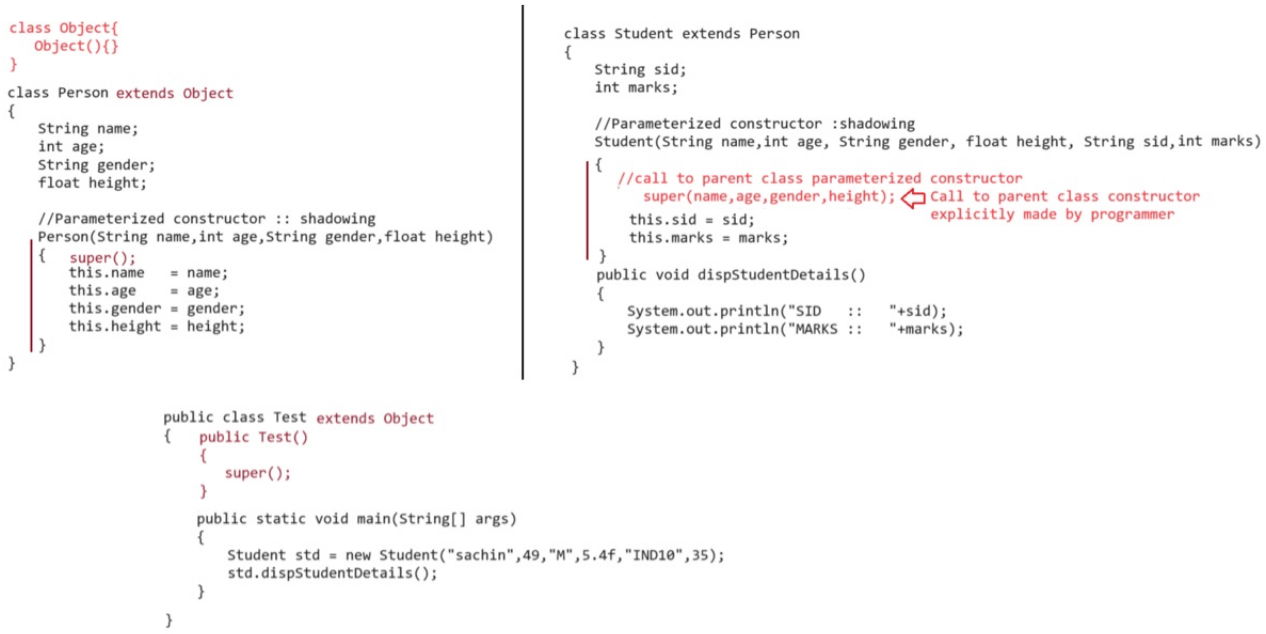
D. A NullPointerException occurs at runtime.

E. A NumberFormatException occurs at runtime.

F. An IllegalStateException occurs at runtime.

Answer: D

ALL THE INHERITANCE SNIPPETS ARE NOT SOLVED(SEARCH ABOVE ONE WHICH IS LAST SOLVED)



In case of Parent class, if parent class has a parameterized constructor then in child class constructor compulsorily there should be a call to parent class parameterized constructor otherwise the code would result in "CompileTime Error".

Constructor OverLoading

eg#2.

//Constructor Overloading

this() :: It is used to make a call to current class constructors only.

```
class Demo
{
    Demo(int i)
    {
        this(10.5f);
        System.out.println("int arg constructor");
    }
    Demo(float f)
    {
        System.out.println("float arg constructor");
    }
```

```
    Demo()
    {
        this(10);
        System.out.println("zero arg constructor");
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Demo d1= new Demo();//float-arg/int-arg/zero-arg constructor
        System.out.println();
        Demo d2= new Demo(10);//float-arg/int-arg constructor
        System.out.println();
        Demo d3= new Demo(10.5f);//float-arg constructor
    }
}
```

Output

float arg constructor

int arg constructor

zero arg constructor

float arg constructor

int arg constructor

float arg constructor

Few Caeses →

1)- class Test

{

}

Compiler

class Test exteds Object

{

    Test()

    {

        super();

    }

```
}
```

2)- public class Test

```
{

}
```

Compiler

```
public class Test exteds Object
{
    public Test()
    {
        super();
    }
}
```

3)- class Test

```
{
    Test(int i)
    {
        this();
    }
    Test()
    {
    }
}
```

Compiler

```
class Test extends Object
{
    Test(int i)
    {
        this();
    }
    Test()
    {
        super();
    }
}
```

Conclusions of this() vs super()

Case1:

We have to take super() or this() only in the 1st line of the constructor, if we are taking anywhere else it would result in "CompileTimeError".

Case2:

We can't use either super() or this() both simultaneously

Case3:

super() or this() should always be the first statement inside the constructor but we can't use inside the method, if we try to use it would result in "CompileTime Error".

Q> Whenever we are creating an object of child class object then automatically parent class object will be created?

Answer: false. only child class object will be created but not the parent class.

PROOF:: eg#1.

```
class Parent
{
    Parent()
    {
        System.out.println(this.hashCode()); //366712642
    }
}
class Child extends Parent
{
    Child()
    {
        System.out.println(this.hashCode()); //366712642
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Child c = new Child();
        System.out.println(c.hashCode());   //366712642
    }
}
```

Recursive call

Case1: recursive method call is always "RuntimeException"(not always --> if we have wrote conditions to break the recursion then it will not lead to StackOverflowError) where as recursive constructor invocation is a "CompiletimeError".

eg#1

```
public class Test
{
    public static void methodOne()
    {
        methodTwo();
    }
    public static void methodTwo()
    {
        methodOne();
    }
    public static void main(String[] args)
    {
        methodOne();
        System.out.println("hello");
    }
}
```

Output:: java.lang.StackOverflowError

eg#2.

```
public class Test
{
    Test(int i)
    {
        this();
    }
    Test()
    {
        this(10);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println("hello");
    }
```

}

CE: recursive constructor invocation

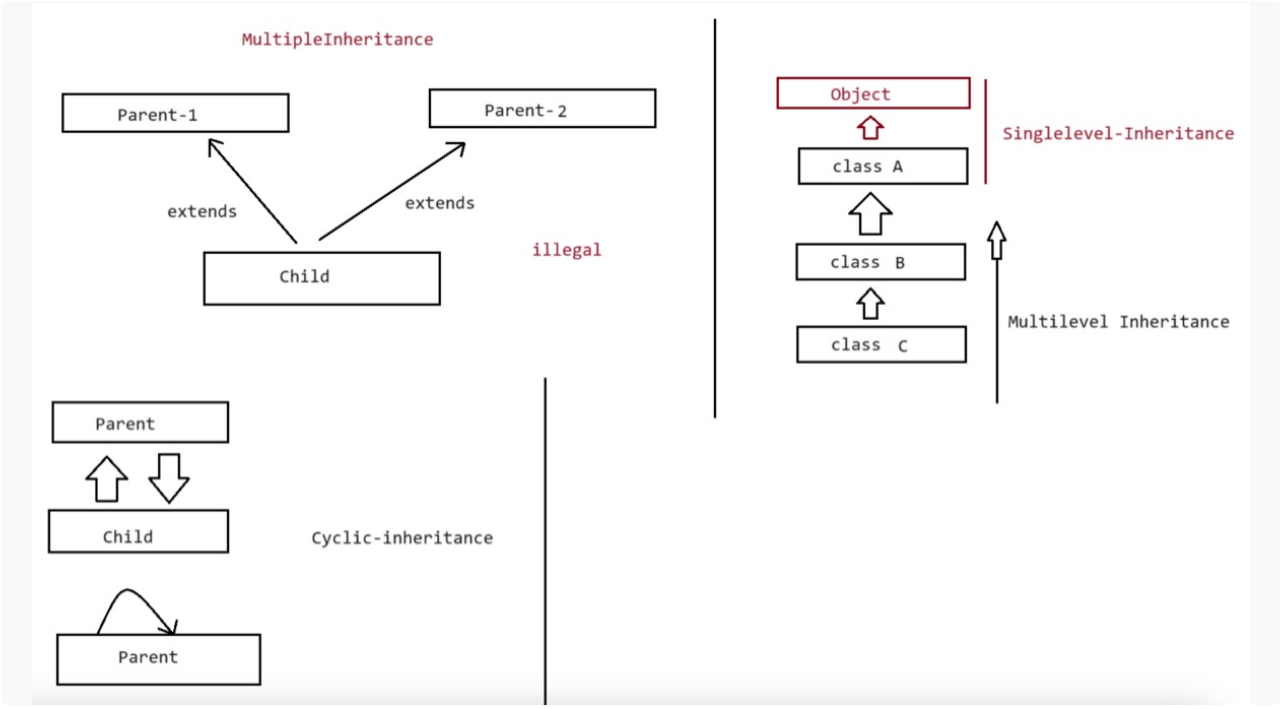
Q>

Given

```
public class Hello {
    String title;
    int value;
    public Hello() {
        title += " World";
    }
    public Hello(int value) {
        this.value = value;
        title = "Hello";
        Hello();            //we cannot call a constructor (CE)
    }
}
```

Types of inheritance supported by java

1. **Multiple inheritance :** Not supported because of ambiguity in java through "classes".**(if same methods occur in both the classes than after Multiple inheritance compiler will confuse which method to call)**
2. **Multilevel inheritance :** Getting properties from parent to child in hierarchial way is refered as "MultiLevel Inheritance".
3. **Cyclic inheritance :** Not supported in java becoz of constructor invocation in loop.



Note: In case of Overriding

1. **Compiler will use reference of the type to check whether the respective method is available in the class or not.**
2. **JVM will use the current object and respective method of that object will be called.**

In java Polymorphism is one of the pillar of oops. We have 2 types of polymorphism

a. Compiletimebinding / Earlybinding / staticbinding

eg: Overloading, method-hiding

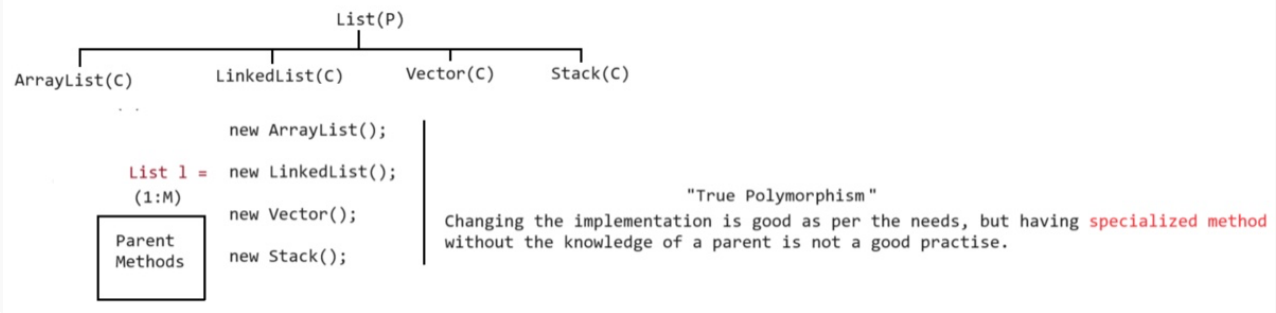
b. Runtime / dynamic / latebinding

eg: Overriding

1)- In case of Overriding JVM will play a vital role of binding the method call to method body, so we say overriding as "RuntimePolymorphism". In case of Overriding compiler will just check the reference type and see whether the method signature is present in the class or not.

2)- Parent p1 = new Child();

This syntax is really important in terms of code readability and consistency. The main idea is that everything should be in parent class and child class should either use it or override it if child is not satisfied with it. So there should be no child specific methods(as we can't access it with p1). With p1 we can access all the methods which are in parent and the overridden methods if any will be taken from child(logic). We cant access child specific methods with p1. Overloading increases number of lines in code while with overiding we can decrease those lines.And JVM will make a decision which method to execute.



In case of Overriding lines of code would be less, but because of JVM playing a role the actions will be peformed based on the runtime object.

Since private class of a parent cannot be overridden and if we try to do so by mentioning @Override, it will lead to CE. But if we remove @Override and compile than program will be compiled successfully(that method will not be overridden but that method will be treated as a specialized method of child class.)Hence it is always a good practice to mention @Override(Rule 2 in following points). Another reason → if I write @Overide over methodOne (static method in parent)in Child class then it will lead to CE(as static methods cannot be overridden) otherwise it is method hiding --> Importance of mentioning @Overide

Rule1:

In case of MethodOverriding, method signature(Method Signature is the combination of a method's name and its parameter list.) should be same in child class while overriding.

It is possible to change the return type also if it is of reference type[Relationship should be "IS-A"(inheritance)].

If the return type is of primitive type, then we can't change the returntype, if we try to change it would result in "CE".

For this there are 4 examples given in last one we can use void to bypass the above rule(check notes if in confusion → Rule4 is being used).

Rule2:

Private methods are not visible in child class, so Overriding them in the child class is not possible.

Note: **final access modifier can be applied to**

a. class :: These classes won't participate in inheritance.(this is required because some classes are helper classes whose properties we can directly use(So we cannot inherit and use it on our classes)(eg: String is a final class(if it is not final than someone can inherit the properties and overwrite them and say he has created those methods from scratch hence final is used.)))

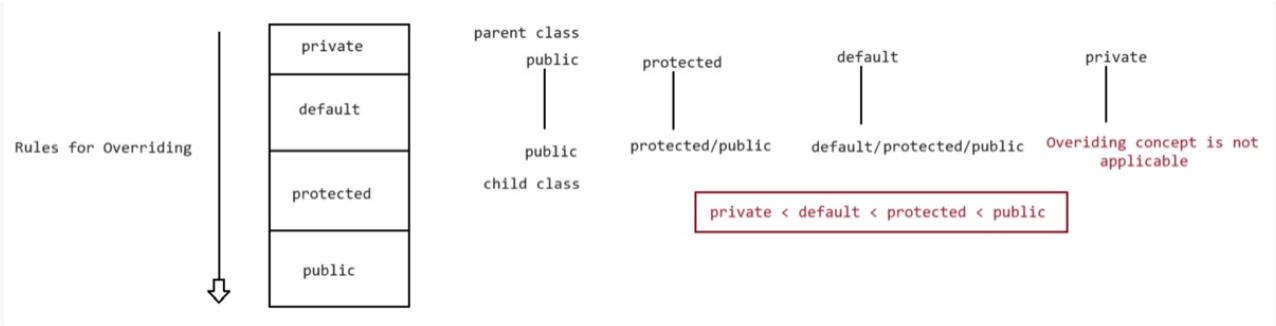
b. method :: These methods implementation can't be changed , but it will be inherited to child class.

c. variable :: it would be treated as compile time constants whose value should not be changed during the execution.

Rule3:

1. Parent class final methods can't be changed to non-final in child class during overriding.
2. Parent class non-final methods can be made as final in child class during overriding.

Rule4: In case of Overriding, we can increase the privilege of the access modifier ,if we try to decrease it would result in "CompileTimeError".



Overriding w.r.t static methods

1)- We can't override static method as non-static(meaning→ parent-static and in child-non-static while overriding)

2)- Non-static method can't be made static in Overriding.

3)- **static methods can't be Overriden** --> (**Reason from chatgpt** --> **The concept of method overriding is closely tied to polymorphism and the dynamic dispatch of methods during runtime. However, static methods operate at the class level rather than the instance level, and their behavior is determined at compile-time.**)

Again if I write @Overide over methodOne(static method in parent) in Child class then it will lead to CE(as static methods cannot be overridden) otherwise it is method hiding --> Importance of mentioning @Overide.

Difference b/w methodhiding and methodoverloading

MethodHiding

1. both child class and parent class methods should be static.
2. Method resoultion will be taken care by compiler based on the reference type.
3. Method hiding is considered as "static binding/early binding".

MethodOverrding

1. both child class and parent class methods should be non-static.
2. Method resoultion will be taken care by JVM based on the runtime object.
3. Method Overriding is considered as "runtime binding/late binding".

eg#1.

class Parent

```
{
    public static void methodOne(){
        System.out.println("From Parent...");
    }
}
class Child extends Parent
{
    public static void methodOne(){
        System.out.println("From Child...");
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Parent p = new Parent();
        p.methodOne();//From Parent...

        Child c = new Child();
        c.methodOne();//From Child...

        Parent p1 = new Child();
        p1.methodOne();  //From Parent...   ///. ---> very very imp --> method hiding
    }
}
```

without static methods and + writing @override we will get "From Child..." as output for the final part

Overriding w.r.t var-args method → A var-arg method should be overriden as "var-arg" method only. if we try to override with normal method then it would become overloading but not overriding.

eg#1.

```
class Parent
{
    //var-arg method
    public void methodOne(int... i){
        System.out.println("From Parent...");
    }
}
```

```
class Child extends Parent //overloading not overriding
{
    //normal method
    public void methodOne(int i){
        System.out.println("From Child...");
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Parent p = new Parent();
        p.methodOne(10);   //From Parent...

        Child c = new Child();
        c.methodOne(10);   //From Child...

        Parent p1 = new Child();
        p1.methodOne(10);   //From Parent...
    }
}
```

eg#2.

```
class Parent
{
    //var-arg method
    public void methodOne(int... i){
        System.out.println("From Parent...");
    }
}
class Child extends Parent // Overriding
{
    @Override
    public void methodOne(int... i){
        System.out.println("From Child...");
    }
}
public class Test
```

```
{
    public static void main(String[] args)
    {
        Parent p = new Parent();
        p.methodOne(10);   //From Parent...

        Child c = new Child();
        c.methodOne(10);   //From Child...

        Parent p1 = new Child();
        p1.methodOne(10);   //From Child...
    }
}
```

Overriding w.r.t variables

=> Overriding is not applicable for variables.

=> Variable resolution is always taken care by compiler based on reference type.

```
class Parent
{
    int x= 888;
}
class Child extends Parent
{
    int x= 999;
}
public class Test
{
    public static void main(String[] args)
    {
        Parent p = new Parent();
        System.out.println(p.x);   //888

        Child c = new Child();
        System.out.println(c.x);   //999

        Parent p1 = new Child();
        System.out.println(p1.x);   //888
    }
```

}

instance control flow in parent to child relationship

Whenever we are creating an object of child class the following sequence of events will take place

a. Identification of instance variable from Parent to Child.

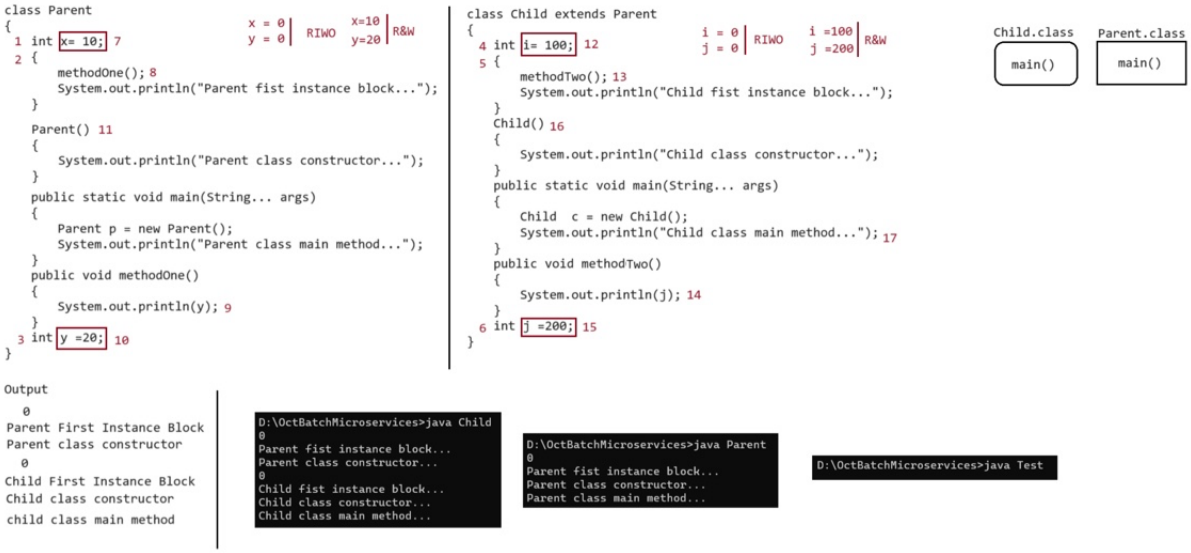
b. Execution of instance variable assignments and instance block only in Parent class.

c. Execution of parent class constructor.

d. Execution of instance variable assignments and instance block only in child class.

e. Execution of child class constructor.

1)- for instance block and variables



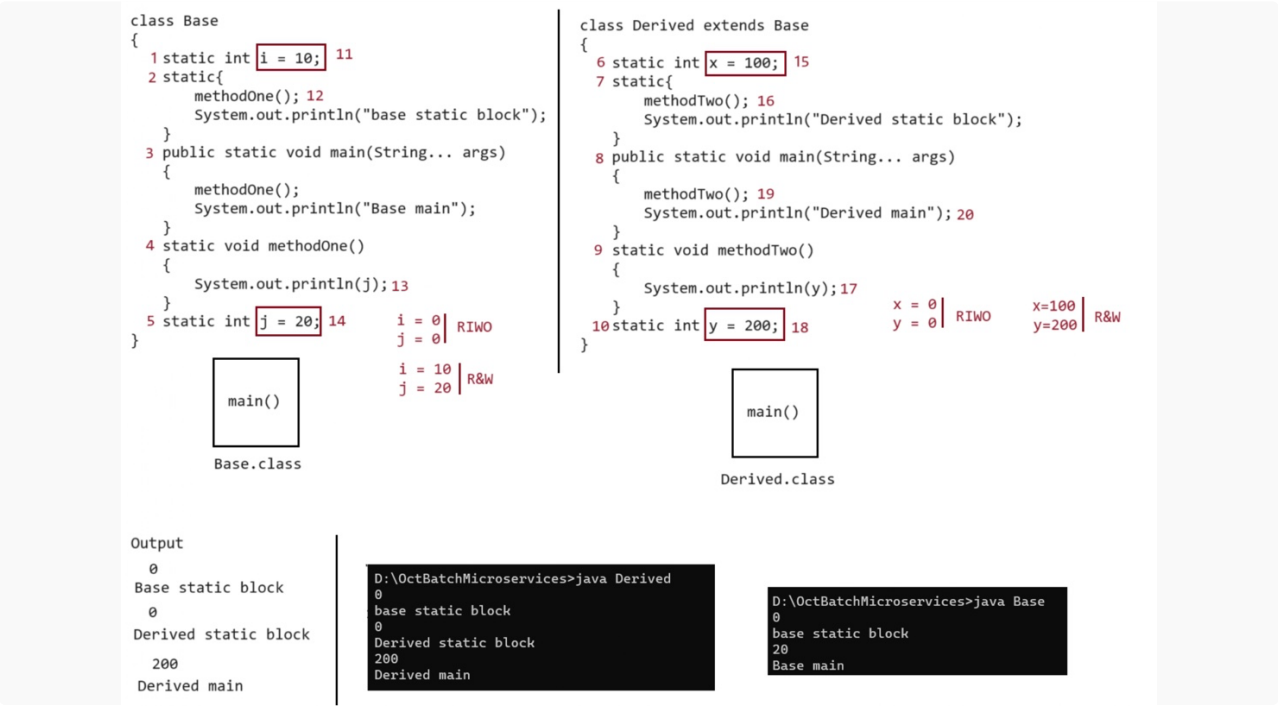
2)- static control flow in inheritance

Whenever we are executing child class the following sequence of events takes place

a. Identification of static memebers from parent to child.

b. Execution of static variable assignments and static block execution from parent to child

c. Execution of child class main().



Creating an object everytime without its need is it a good programming practice?

Ans. No, it is not a good programming practise because to create an object

1. loading of .class file should be happen[All the activities of static control flow should happen]

b. instance control flow from parent to child should happen.

Since it is time consuming event, only when it is required we need to use "new" keyword in java.

Encapsulation = Datahiding + abstraction.

Encapsulation

Binding of data and corresponding methods into single unit is called "Encapsulation".

Abstraction

Hiding internal implementation,but exposing the set of services is called "Abstraction".

Datahiding => Our internal data should not go to outside world directly that is outside person can't access internal data directly.

To promote datahiding, we need to use "access modifiers"

eg: private,protected

It promotes security.

Dependancy Injection

=> It refers to process of injecting the values to the instance variables of a class.

=> we can perform depedancy injection in 2 ways

a. through setter method.

b. through constructor

Pillars of oops

a. Encapsulation => Datahiding(private, setXXXX(),getXXXX()) + abstraction[abstract,interface]

b. Inheritance => ReUsablitiy(IS-A,HAS-A)

c. Polymorphism => Code Flexiblity(Overloading,Overriding, MethodHiding)

What is TightCoupling and What is Loose Coupling?

TightCoupling(if i use Tiger t as parameter than i will not able to pass any other animal to it other than Tiger hence it is tightly coupled)

eg#1.

```
class Forest {
    //Method Over-loading :: Tight Coupling
    public void allowAnimal(Tiger t)
    {
        t.eat();
        t.sleep();
        t.breathe();
    }
    public void allowAnimal(Deer d)
    {
        d.eat();
        d.sleep();
        d.breathe();
    }
    public void allowAnimal(Monkey m)
    {
        m.eat();
        m.sleep();
        m.breathe();
    }
}
```

Loose Coupling

eg#1.

//Helper class

```
class Forest {
    /* RunTime Polymphism[1:M] = new Tiger(); Animal ref = new Deer(); = new Monkey(); */
```

```
    //Method-Overriding :: LooseCoupling
    public void allowAnimal(Animal ref)
    {
        ref.eat();
        ref.sleep();
        ref.breathe();
        System.out.println();
    }
}
```

Abstract class in java

=> If we don't want an object to be created for a particular class, then such class we need to mark as " abstract".

=> abstract access modifer is applicable at

a. class level : prevents object creation.

b. method level : prevents giving the implementation for body of a method.

c. variable level : not applicable at variable level.

Through abstract keyword we can promote "abstraction".

=> By referring to abstract class, we would get to know only the services name(methodnames),but not the internal implementation given by developers, this mechanism only we call it as "abstraction".

=> for an abstract class,"instantiation is not possible",but we can create a reference for an " abstract class".

=> If a class contains only one abstract methods also,then we need to mark the class as "abstract".

=>In java abstraction shows incompleteness

=>We cannot make object of abstract class but we can use it as a refrence. Which makes sence as if we create object of abstract class than there is no way we can access its methods as they are also abstract or incomplete.

=> Abstraction is not applicable at variable level as abstract means incompleteness and variable represents a memory location which cannot be incomplete logically.

=> Can parent class be final --> no, then no further class can inherit its properties and define implemention for abstract methods. Then that class will be of no use.

Note:

1. An abstract class can contain "concrete methods" also along with abstract methods.
2. During inheritance, concrete methods can be "Overriden".
3. An abstract class need not have any abstract methods also.
4. For an abstract class object can't be created, only reference can be created.

5. **If we dont want to write the defination of an abstract method than we need to include the defination of that method in current class(not mandatory) and also write abstract in front of the current class.**

Note: Illegal combinations of access modifier at methods level

a. abstract and final -------> [Illegal]

b. abstract and static -------> [Illegal]

Question.

1. Will constructor be called at the time of Object creation?

   Ans. yes

2. Can we create an object for abstract class?

   Ans. No.

3. Does abstract class have constructor?

   Ans. yes.

4. If object can't be created for an abstract class, then why do we need a constructor?

   Ans. Even though we can't create an object for abstract class, still constructor is required, because in inheritance the child class object will be initialized by making a call to parent class constructor. constructor in abstract class is required to initialize the object completely.

5. **When we create an object of child class, will object of parent class also be created?**

Ans. No, parent class constructor will be executed to make the child object complete(initialized).

eg#1.

```
//Person class
abstract class Person
{
    String name;
    int age;
    char gender;
    Person(String name,int age, char gender)
    {
        System.out.println("HashCode is :: "+this.hashCode());
        this.name =name;
        this.age = age;
        this.gender =gender;
    }
    public void dispDetails(){
        System.out.println("Name is :: "+name);
        System.out.println("Age is :: "+age);
```

```
        System.out.println("Gender is :: "+gender);
    }
}
//Concrete class
class Student extends Person
{
    int sid;
    float avg;
    Student(String name,int age,char gender,int sid, float avg)
    {
        super(name,age,gender);
        System.out.println("HashCode is :: "+this.hashCode());
        this.sid = sid;
        this.avg = avg;
    }
    public void dispDetails(){
        super.dispDetails();
        System.out.println("SID is :: "+sid);
        System.out.println("AVG is :: "+avg);
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Person p;
        p = new Student("sachin",51,'M',10,57.5f);
        System.out.println("HashCode is :: "+p.hashCode());
        p.dispDetails();
    }
}
```

Output

HashCode is :: 366712642

HashCode is :: 366712642

HashCode is :: 366712642

Name is :: sachin

Age is :: 51

Gender is :: M

SID is :: 10

AVG is :: 57.5

Interfaces in java

Definition-1

=> Any Service Requirement Specification is called "interface".

eg#1. SunMS is responsible to define JDBC API and Database vendors is responsible to provide implementation to it.

Definition-2

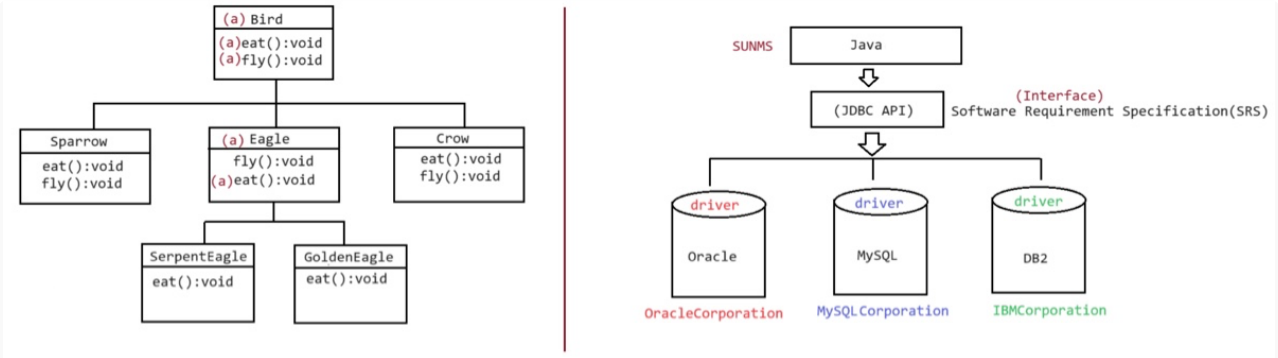
=> It refers to the contract b/w client and the service provider

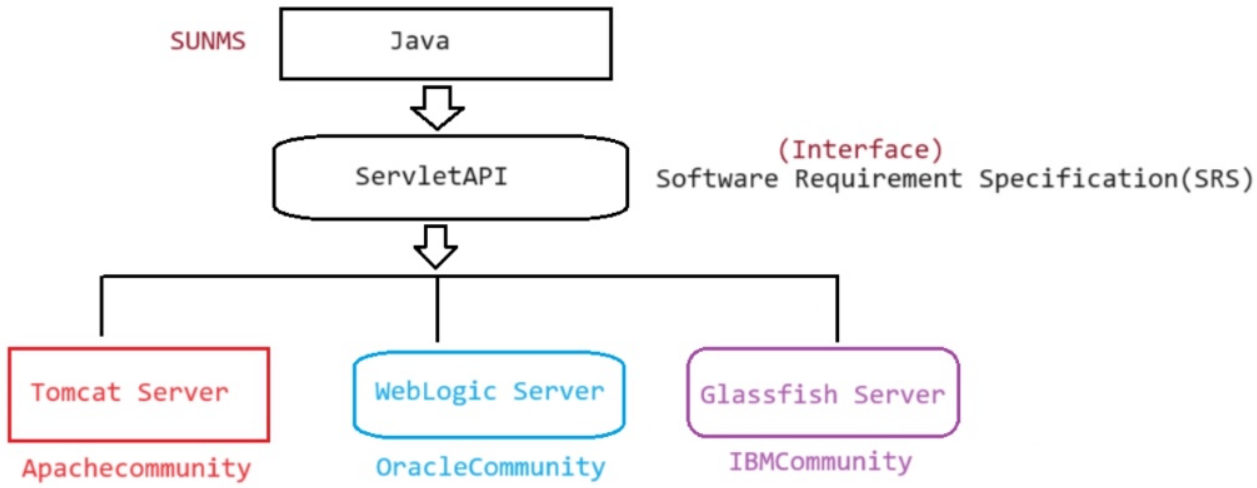
eg#2. ATM GUI screen describes what bank people are offering,at the same time GUI SCREEN represents what customer is expecting So GUI screen acts like a bridge b/w client and the service provider.

Definition-3

=> Inside interface every method present are "public and abstract".

=> Since it is public and abstract,we say interface as "pure abstract class".





Very very imp--> helpers class-->

Helper class are the classes which are required to reduce the redundant code. So we pass the refrence to this class and than call respective methods.

The refrence is of parent type in helper's class parameter to make the code general for all types.

Now when we call this class we make an object and than call it.

Cant we make it abstract?

Ans - but than we will not be able to make its object and access it. Than no problem mark the methods of this class as static and then we can directly access it with class name. In this way we can avoid object creation as well which is a costly process--> very important for optimisation.

At method level using abstract and static(both) is illegal because we can access that method with class name and since its implementation is not there hence illegal.

Interface is called "pure abstract class" because in interface we will have only abstract methods no concrete methods are allowed while in abstract class we can have both.

Rules of Interface

case 1:

1. **Whenever we are implementing an interface, we need to give body for all the abstract methods present inside the interface.If we fail to give body for all the methods present inside the interface, then we need to mark the class as "abstract".**
2. For an interface instantiation(creation of object) is not possible.
3. For an interface, creation of reference is possible.
4. **Through interface we achieve :: TruePolymorphism(Overriding).[The key to achieving true polymorphism here is that at compile time, Java only knows that you have a reference variable of type `Shape`. However, when you call `calculateArea()`, the actual method invoked depends on the object's type at runtime (either `Square` or `Circle` object). This allows for flexible and dynamic behavior based on the actual object instance.]**

Case2: Whenever we are implementing an interface method compulsory, it should be declared as public otherwise we will get "CompileTime Error"**(rule where we cannot assign weaker access privileges during overriding).**

case3:

=> Relationship b/w interface to class is always "implements".

=> Relationship b/w interface to interface is always "extends".

=> If we implemented the interface which has extended from one more interface, then as a programmer the implementation class should give body for all the abstract methods present in the interface, if not we need to mark the class as "abstract", otherwise the code would result in "CompileTime Error".

case4:

At a time one class can extend from how many classes?

Answer. One because java doesn't support multiple inheritance through class to avoid "Ambiguity problem".

At a time one class can implement multiple interfaces?

Answer: Yes possible, so we can say mulitple inheritance is supported in java through "interfaces" and "Ambiguity problem " won't occur because Compiler will keep the method signature in the implementation class only if it is not available.

As noticed in the below example ICalculator1 and ICalculator2 both have void add(int a,int b) method,but compiler will keep only one method void add(int a,int b) in the implementation class through which "Ambiguity problem" will not occur in interfaces.

At a time can one class implement an interface and extends a class?

Answer: yes,but first we need to have extends and followed by implements.

++

To promote loose coupling we follow the rule of

interface ->abstract class -> class

++

Which of the following is true?

a. A class can extend any no of class at a time.

b. An interface can extend only one interface at at time.

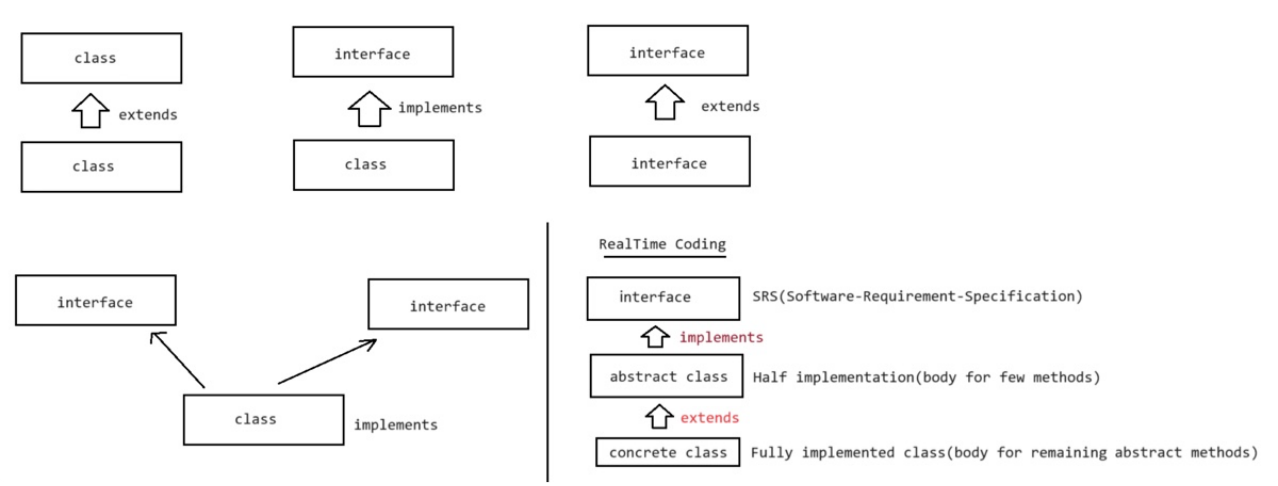
c. A class can implement only one interface at at a time.

d. A class can extend a class and can implement an interface but not both simultaneously.

e. An interface can implements any no of Interfaces at a time.

f. None of the above

Answer: f



Imp diagram(last one specifically)

Interface variables

=> Inside the interface we can define variables.

=> Inside the interface variables is to define requirement level constants.

=> Every variable present inside the interface is by default public static final.

public :: To make it available for implementation class Object.

static :: To access it without using implementation class Name.

final :: Implementation class can access the value without any modification.

variable declaration inside interface

a. int x=10;

b. public int x=10;

c. static int x=10;

d. final int x=10;

e. public static int x=10;

f. public final int x=10;

g. static final int x=10;

h. public static final int x=10;

Answer: All are valid

Note:

Since the variable defined in interface is public static final, we cannot use modifiers like private, protected, transient, volatile. since the variable is static and final, compulsorily it should be initialized at the time of declaration otherwise it would result in compile time error.

eg::

interace IRemote{ int x;}// compile time error.

Eg.1

```
interface IRemote
{
    //public static final
    int MIN_VOLUME = 0;
    int MAX_VOLUME = 100;
}
public class Test implements IRemote
{
    public static void main(String[] args)
    {
        int MIN_VOLUME = -5;
        System.out.println(MIN_VOLUME);
        System.out.println(IRemote.MIN_VOLUME);
        System.out.println(Test.MIN_VOLUME);
    }
}
```

Output

-5

0

0

Interface Naming Conflicts

Case 1::

If 2 interfaces contain a method with same signature and same return type in the implementation class only one method implementation is enough.

eg#1.

```
interface IRight
{
    public void methodOne();
}
interface ILeft
{
    public void methodOne();
}
public class Test implements ILeft,IRight
```

```
{
    @Override
    public void methodOne()
    {
        System.out.println("Impl for MethodOne...");
    }
    public static void main(String[] args)
    {
        Test t =new Test();
        t.methodOne();
    }
}
```

Output

Impl for MethodOne...

Case2:

If 2 interfaces contain a method with same name but different arguments in the implementation class we have to provide implementation for both methods **and these methods acts as a Overload methods.**

eg#1.

```
interface IRight
{
    public void methodOne();
}
interface ILeft
{
    public void methodOne(int i);
}
public class Test implements ILeft,IRight
{
    @Override
    public void methodOne()
    {
        System.out.println("Impl for MethodOne...");
    }
    @Override
    public void methodOne(int i)
    {
```

```
        System.out.println("Impl for MethodOne with One argument");
    }
    public static void main(String[] args)
    {
        Test t =new Test();
        t.methodOne();
        t.methodOne(10);
    }
}
```

Output

Impl for MethodOne...

Impl for MethodOne with One argument

case3:

If two interfaces contains a method with same signature but different return types then it is not possible to implement both interface simultaneously.

eg#1.

```
interface IRight
{
    public void methodOne();
}
interface ILeft
{
    public int methodOne();
}
public class Test implements ILeft,IRight
{
    @Override
    public void methodOne()
    {
        System.out.println("Impl for MethodOne...");
    }
    @Override
    public int methodOne()
    {
        System.out.println("Impl for MethodOne with One argument");
    }
```

```
    public static void main(String[] args)
    {
        Test t =new Test();
        //Overloading
        t.methodOne();
        t.methodOne();
    }
}
```

Output

CE: ambigous method call.

Can a java class implement any no of interfaces simultaneously?

Answer.yes, except if two interfaces contains a method with same signature but different return types.

Variable naming conflicts::

Two variables can contain a variable with same name and there may be a chance variable naming conflicts but we **can resolve variable naming conflicts by using interface names.**

eg#1.

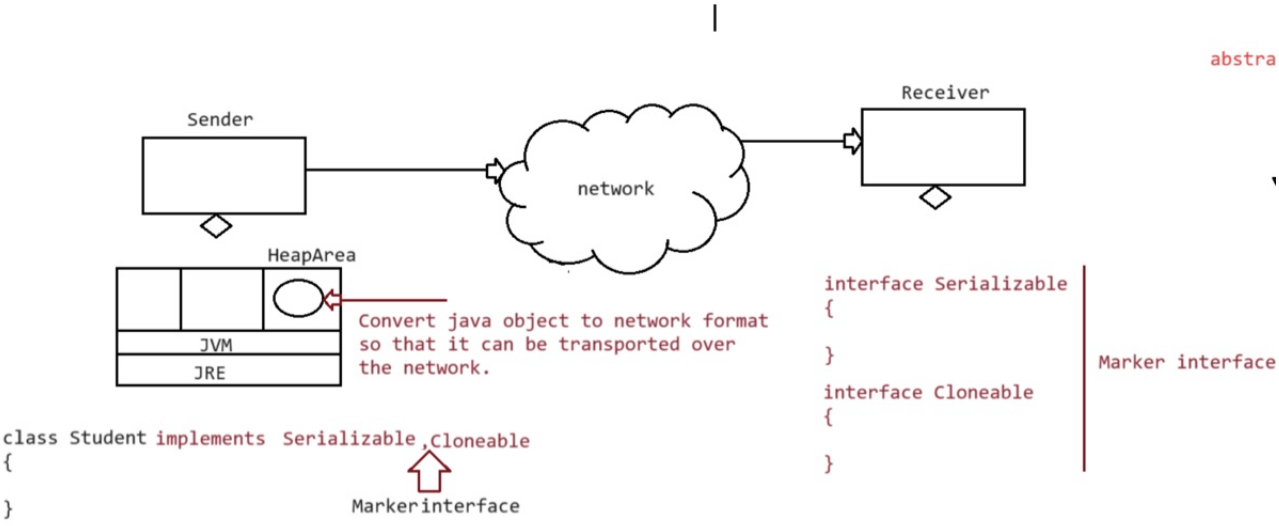
```
//SRS :: methods -> public abstract
//SRS :: variables -> public static final
interface IRight
{
    int x = 888;
}
interface ILeft
{
    int x = 999;
}
public class Test implements ILeft,IRight
{
    public static void main(String[] args)
    {
        //System.out.println(x);
        //System.out.println(Test.x);
        System.out.println(IRight.x);
        System.out.println(ILeft.x);
```

```
    }
}
```

Output

888

999



Explanation of the image -->

1)- If we want to send an object from sender to receiver over a network than we can't directly send it first we need to convert it into network format.

2)- To achieve we need to implement an interface where we need not to implement anything all will be done by jvm and such type of interface is known as **Marker Interface.**

3)- Serializable --> interface

4)- Since this pre-defined interface don't have I infront of it to avoid confusion we can add that.

5)- Now suppose we have sent this object but at receiver end there was some problem and he didn't receive the object than we again have to create an object and send it which is a costly process, instead we can create a copy of the object and send it. This can be achieved with Cloneable interface again it will be done by jvm.

Marker Interface(name --> as we are just marking its name with class, not providing any implementation from our side)

=> If an interface does not contain any methods and by implementing that interface if our Object will get some ability such type of interface are called "Marker Interface"/"Tag Interface"/"Ability Interface".

=> example --> Serializable, Cloneable, SingleThreadModel, RandomAccess.

Example1 -->

By implementing Serializable interface we can send that object across the network and we can save state of an object into the file.

Example2 -->

By implementing SingleThreadModel interface servlet can process only one client request at a time so that we can get "Thread Safety".

Example3 -->

By implementing Cloneable Interface our object is in a position to provide exactly duplicate cloned object.

Without having any methods in marker interface how objects will get ability?

Ans. JVM is responsible to provide required ability.

Why JVM is providing the required ability to Marker Interfaces?

Ans. To reduce the complexity of the programming.

Can we create our own marker interface?

Ans. Yes, it is possible but we need to cusomtize JVM.

Adapter class

1)- It is a simple java class that implements an interface only with empty implememtation for every method.

2)- If we implement an interface compulsorily we should give the body for all the methods whether it is required or not. This approach increases the length of the code and reduces readability.

eg::

```
interface X{
    void m1();
    void m2();
    void m3();
    void m4();
    void m5();
}
class Test implements X{
    public void m3(){
        System.out.println("I am from m3()");
    }
    public void m2(){}
    public void m3(){}
    public void m4(){}
    public void m5(){}
}
```

1)- In the above approach, even though we want only m3(), still we need to give body for all the abstract methods, which increase the length of the code, to reduce this we need to use "Adapater class".

2)- Instead of implementing the interface directly we opt for "Adapter class". Adapter class are such classes which implements the interface and gives dummy implementation for all the abstract methods of interface.

3)- So if we extends Adapter classes then we can easily give body only for those methods which are interested in giving the body.

eg::

```
interface X{
    void m1();
    void m2();
    void m3();
    void m4();
    void m5();
}
abstract class AdapaterX implements X{
    public void m1(){}
    public void m2(){}
    public void m3(){}
    public void m4(){}
    public void m5(){}
}
class TestApp extends AdapterX{
    public void m3(){
        System.out.println("I am from m3()");
    }
}
```

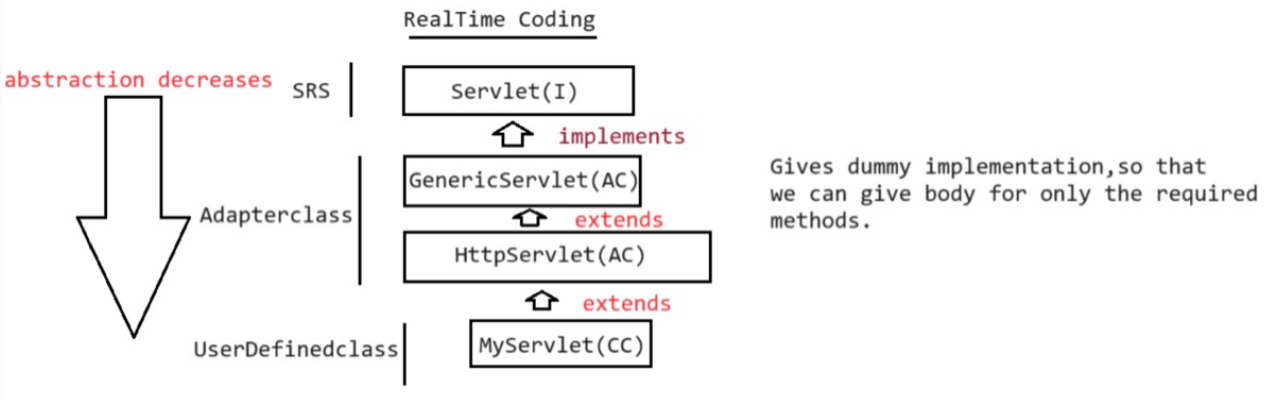
eg:: interface Servlet{....}

abstract class GenericServlet implements Servlet{}

abstract class HttpServlet extends GenericServlet{}

class MyServlet extends HttpServlet{}

Note:: Adapter class and Marker interface are big utilities (or design) to programmer to simplify programming.



Interview Questions

Difference b/w Abstract class and Interface?

Interface::

1)- If we dont know anything about implementation just we have requirement specification then we should go for interface.

2)- Every method present inside the interface is always public and abstract whether we are declaring or not.

3)- We can't declare interface methods with the modifiers like private,protected,final,static,synchronized,native,strictfp.

4)- Every interface variable is always public static final whether we are declaring or not.

5)- Every interface variable is always public static final we can't declare with the following modifiers like private,protected,transient,volatile.

6)- For every interface variable compulsorily we should perform initialisation at the time of declaration, otherwise we get compile time error.

7)- Inside interface we can't write static and instance block.

8)- Inside interface we can't write constructor.

Abstract class::

1)- If we are talking about implementation but not completely then we should go for abstract class.

2)- Every method present inside abstract class need not be public and abstract.

3)- There are not restrictions on abstract class method modifiers.

4)- Every abstract class variable need not be public static final.

5)- No restriction on access modifiers

6)- Not required to perform initialisation for abstract class variables at the time of declaration.

7)- Inside abstract class we can write static and instance block.

8)- Inside abstract class we can write constructor.

Q>Every method present inside the interface is abstract,but in abstract class also we can take only abstract methods also then what is the need of interface concept?

Ans. we can replace interface with abstract class,but it is not a good programming practise.if we try to do, it would result in "missusing the role" of abstract class and it would also create peformance issue.

Using interface

```
interface ISample
{
}
class SampleImpl implements ISample
{
}
```

1. ISample sample=new SampleImpl(); **//one level chaining :: SampleImpl ---> Object[Performance is relatively high]**
2. **While Implementing ISample, we can also get the benefit from Another class[Inheritance : Reusability].**

Using Abstract class

```
abstract Sample
{
}
class SampleImp extends Sample
{
}
```

1. Sample sample=new SampleImp(); **//Multi level chaining:: SampleImp ---> Sample ---> Object [Performance is low]**
2. **While extending Sample, we can't get the benefit of other classes[Inheritance can't be used here]**

Q> Why abstract class can contains constructor and interface doesn't contains constructor?

=> Constructor :: To initialize the instance variable of an object, meaning is to provide values for instance variables.

:: In abstract class we have instance variable so we need constructor for initializing the instance variables.

:: In case of interface, we don't have instance variable we have variables which are of type public static final, these variables are initialized at the time of declaration only.

so we dont' need constructors in interface.

Q> When to go for interface and when to go for abstract class?

interface -> To promote 100 percent abstraction we need to go for "interface" or to provide Software Requirement Specification we need to go for "interface".

abstract class -> If we are taking about implementation that is partial implementation, then we need to go for "abstract class".

Which of the following are valid?

1. The purpose of the constructor is to create the object.
2. The purpose of the constructor is to initialize the object,not to create the object.
3. Once constructor completes then only object creation completes.
4. First object will be created and then constructor will be executed.
5. The purpose of the new keyword is to create object and the purpose of constructor is to initalize the object.
6. **We can't create Object for abstract class directly but indirectly we can create.**
7. Whenever we are creating child class object automatically parent class object will be created.
8. Whenever we are creating child class object automatically abstract class constructor(provided if it is parent) will be executed.
9. Whenever we are creating child class object automatically parent constructor will be executed but parent object wont be created.
10. Either directly or indirectly we can't create Object for abstract class and hence constructor concept is not applicable for abstract class.
11. Interface can contain constructor.

Valid : 2,4,5,8,9

Invalid : 1,3,6,7,10,11