Chapter 5 : Initialization and cleapup

➔ Constructors don't have return type.

➔ If one is using overloaded methods and if the methods look like sum(int i , int j) and if you have two values say byte a and byte b and if u send sum (a, b) it will still execute because byte will be promoted to int. But the reverse is not possible i.e. if method arguments are byte and if one is sending int then there will be error but one can cast and send. example sum((int)a, (int)b). In general if casting rule applied to method parameters.

Example :

```
package ch5InitializationNCleanUp;

class MethParamCast1
{
	void intMethod(int i, int j)
	{
		int sum = i+j;
		System.out.println(sum);
	}

	void byteMethod(byte i, byte j)
	{
		byte sum = (byte) (i+j);
		System.out.println(sum);
	}

	void byteIntMethod(byte i, int j)
	{
		int sum = i+j;
		System.out.println(sum);
	}
}

public class MethParamCast
{
	public static void main(String[] args)
	{
		int a=10,b=10;
		byte  c=5,d=5;

		MethParamCast1 mpc1 = new MethParamCast1();

		//int method
		System.out.println("int method");
		mpc1.intMethod(a,b);//OK both are int
		mpc1.intMethod(a,c);//OK c will be upcast to int
		System.out.println();

		//byte method
		System.out.println("byte method");
		mpc1.byteMethod(c, d);//OK both are byte
		//mpc1.byteMethod(c, b);//b is not allowed;
		mpc1.byteMethod(c, (byte) b);//OK int b downcast to byte
		System.out.println();

		//byte and int method
		System.out.println("byte and int method");
```

```
            mpc1.byteIntMethod(c, a);//OK as per method parameters
            mpc1.byteIntMethod(c, d);//OK as byte d upcast to int
            mpc1.byteIntMethod((byte)a, b);//OK int a downcast to byte
            mpc1.byteIntMethod((byte) a, c);//OK int a downcast to byte and c will
            be upcast to int
      }
}
Output:
int method
20
15

byte method
10
15

byte and int method
15
10
20
15
```

Note : Upcast is done automatically by java.

➔ While creating a class one don't need to write default constructor because compiler creates that but when one is writing a constructor himself then compiler don't create a default constructor so the programmer has to mention default constructor himself if he wants to create such object. But if he is not using default constructor but only parameterized one then it can be ignored.

```
package ch5InitializationNCleanUp;

class DefConst1
{
      int a;

      DefConst1(int i)
      {
            a=i;
      }
      void printval()
      {
            System.out.println("value of a : "+a);
      }
}

public class Ch5DefaultConstructorNeeded {

      public static void main(String[] args) {
            DefConst1 dc = new DefConst1(1);
            dc.printval();
            //DefConst1 dc1 = new DefConst1();
            //Error 'the constructor DefConst1() is undefined
      }
}
Output:
value of a : 1
```

➔ One cant use different return value types for overloading methods. Only argument list differs.

➔ That is overloaded methods have same method name but argument list will differ. Argument list could differ in

a. Number of Parameters

b. Data types of Parameters(When no. are same)

Example :

```
package ch5InitializationNCleanUp;


class OverloadByMethArgs1
{
	void method1(int i, byte j)
	{
		int sum = i+j;
		System.out.println("int, byte : "+sum);
	}
}

class OverloadByMethArgs2 extends OverloadByMethArgs1
{
	void method1(int i, byte j)//duplicate method
	{
		int sum = i+j;
		System.out.println("derived int, byte : "+sum);
	}
	void method1(int i)//Change in no. of Parameters
	{
		System.out.println("Only int arg : "+i);
	}
	void method1(byte i, int j)//Change in data types of Parameters
	{
		int sum = i+j;
		System.out.println("byte, int : "+sum);
	}
}

public class OverloadByMethArgs {

	public static void main(String[] args)
	{
		int a = 10;
		byte c = 5;

		OverloadByMethArgs1 obm1 = new OverloadByMethArgs1();
		OverloadByMethArgs2 obm2 = new OverloadByMethArgs2();

		obm1.method1(a, c);//int, byte
		obm2.method1(a, c);//int, byte
		obm2.method1(a);//only int
		obm2.method1(c, a);//byte, int
	}
}
```

```
Output:
int, byte : 15
derived int, byte : 15
Only int arg : 10
byte, int : 15
```

➢ **The 'this' keyword :**

➔ 'this' keyword can be used only in non-static methods.

➔ It is used when one needs a reference for the current object.

Example :

```
package ch5InitializationNCleanUp;

class ThisUsageCls1
{
	void method1(ThisUsageCls2 tuc2)
	{
		System.out.println("tuc2.a "+tuc2.a);
	}
}

class ThisUsageCls2 extends ThisUsageCls1
{
	int a = 10;
	void method2()
	{
		method1(this);
	}
	void method3()
	{
		this.method2();
	}
}

public class ThisUsageCls {

	public static void main(String[] args)
	{
		ThisUsageCls2 tuc2 = new ThisUsageCls2();
		tuc2.method2();
	}
}
Output:
tuc2.a 10
```

➔ 'this' can be returned using '`return this`'.

➢ Calling constructor from constructor :

➔ Sometime when one write many constructors then he can call one constructor from another just to avoid duplication of code. Also we know 'this' means 'this object' or 'current object' when used in a method as shown in above example. But when 'this' is used in a constructor it takes different meaning. Example is shown

below.

```
//: initialization/Flower.java
// Calling constructors with "this"
import static net.mindview.util.Print.*;
public class Flower {
int petalCount = 0;
String s = "initial value";
Flower(int petals) {
petalCount = petals;
print("Constructor w/ int arg only, petalCount= "
+ petalCount);
}
Flower(String ss) {
print("Constructor w/ String arg only, s = " + ss);
s = ss;
}
Flower(String s, int petals) {
this(petals);
//! this(s); // Can't call two!
this.s = s; // Another use of "this"
print("String & int args");
}
Flower() {
this("hi", 47);
print("default constructor (no args)");
}
void printPetalCount() {
//! this(11); // Not inside non-constructor!
print("petalCount = " + petalCount + " s = "+ s);
}
public static void main(String[] args) {
Flower x = new Flower();
x.printPetalCount();
}
} /* Output:
Constructor w/ int arg only, petalCount= 47
String & int args
default constructor (no args)
petalCount = 47 s = hi
*///:~
```

➔ Above example one should note that 'this' can't be called twice one after another for a constructor or two constructor can't be called compiler will point to the first 'this' and will say that constructor call should be the first statement. Also one can't call constructor from a method.

Example :

```
package ch5InitializationNCleanUp;

class ConstructorFromMethod1
{
      ConstructorFromMethod1()
      {
```

```
            System.out.println("constructor starts");
            method();
            System.out.println("constructor ends");
      }

      void method()
      {
            System.out.println("inside method");
      }
}

public class ConstructorFromMethod {

      public static void main(String[] args)
      {
            ConstructorFromMethod1 cfm1 = new ConstructorFromMethod1();
      }
}
Output:
constructor starts
inside method
constructor ends
```

Note : One can call method from constructor but not vice-versa.

- **'super' keyword :**

➔ The keyword 'super' can be used in two ways

1. Constructors : For calling super class constructors as super() or super(11);
2. Methods : For calling super class methods as super.method();

➔ Also one should keep in mind that 'this' keyword for constructors can be used only inside a same class. For subclass one should use keyword 'super' for call superclass constructor.

➔ Example 1 :

```
package ch5InitializationNCleanUp;

class ThisNSuperConstructor1
{
      String str="string";
      ThisNSuperConstructor1(int i)
      {
            //this("Java");//Recursive constructor invocation
            ThisNSuperConstructor1(String)
            //ThisNSuperConstructor1(str);//method ThisNSuperConstructor1(str)
            doesn't exist
            System.out.println("Inside ThisNSuperConstructor1 "+i);

      }
      ThisNSuperConstructor1(String s)
      {
            this(20);//You can use 'this' only within the same class
      }
```

```
}

class ThisNSuperConstructor2 extends ThisNSuperConstructor1
{
	ThisNSuperConstructor2(int j)
	{
		super(10);//superclass constructor could be only called through 'super'
		this.str = ("string");
		System.out.println("Inside ThisNSuperConstructor2 "+j);
	}
}

public class ThisNSuperConstructor {

	public static void main(String[] args)
	{
		ThisNSuperConstructor2 sc2 = new ThisNSuperConstructor2(4);
	}
}
Output :
Inside ThisNSuperConstructor1 10
Inside ThisNSuperConstructor2 4
```

- Private Constructors :

If a constructor of a class marked private then subclass cannot extend it in a generic way. To extend such a superclass one needs to define another constructor apart from private constructor which can be called through subclass.

```
package ch7ReusingClasses;

class Fruit
{
	String fruitColor;
	private Fruit()
	{
		//Apple cannot extends fruit if only private constructor
		//is present
	}

	Fruit(String color)
	{//Only if non-private constructor is present then it can be extended
		System.out.println("fruit is "+color);
	}
}

class Apple extends Fruit
{
	Apple()
	{
		super("red");
	}
}

public class PrivateConstructorClass
{
```

```
	public static void main(String[] args)
	{
		Apple apple = new Apple();
	}
}
Output:
fruit is red
```

➔ Example 2 :

```
package ch5InitializationNCleanUp;

class SuperKeywordUsage1
{
	int a;
	SuperKeywordUsage1(int i)
	{
		a = i;
		System.out.println("SuperKeywordUsage1 const a = "+a);
	}

	void method1()
	{
		System.out.println("method1()");
	}
}

class SuperKeywordUsage2 extends SuperKeywordUsage1
{
	SuperKeywordUsage2()
	{
		super(5);
		System.out.println("SuperKeywordUsage2 const a = "+super.a);
	}

	void method2()
	{
		super.method1();
		System.out.println("method2()");
	}
}

public class SuperKeywordUsage {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		SuperKeywordUsage2 sku2 = new SuperKeywordUsage2();
		sku2.method2();
	}
}
Output:
SuperKeywordUsage1 const a = 5
SuperKeywordUsage2 const a = 5
method1()
method2()
```

➔ We know that we can't use 'this' inside a static method. Because we know one cannot call a non-static methods from a static one even they are in the same class and keyword 'this' will give reference of the current object which enable static method to call non-static members which should be avoided.

```
class AClass
{
	void printMsg()
	{
		System.out.println("Inside non static method");
	}
}

class BClass
{
	static void methodstc(AClass ac)
	{
		System.out.println("Inside static method");
		ac.printMsg();
		/*AClass ac1 = new AClass();
		ac1.printMsg();*///This is legal. one can create an object and call f
		rom static
		//this.method2();//cant use this inside static method
		//method2();//cant make a static reference to the non-static method
	}
	static void method1()
	{
		System.out.println("Simple static method");
			//methodstc(new AClass());//static meth can call another //static
			method
	}

	void method2()
	{
		System.out.println("Inside method");
		method1();//non-static methods can call static one.
	}
}

public class ThisStatic {

	public static void main(String[] args) {
		AClass acc = new AClass();
		BClass bc = new BClass();
		bc.methodstc(acc);
		bc.method2();
	}
}
Output:
Inside static method
Inside non-static method
Inside method
Simple static method
```

➔ In the above example one can see that in a static method 'methodstc()' one can create an object and call a non-static method from a static one. I.e. if one gets a reference to an object inside static method either by

creating an object or as a method parameter inside an static method then non-static method can be called using this reference.

➢ **Cleanup : Finalization and garbage collection :**

➔ As we know when one do '**new**' for a class an object got created . This object occupies a memory on heap. When it is not referred by any reference then it will be garbage collected. Sometime the memory allocation it done through other medium then **new.** In that case we need to use finalize() then garbage collector first call finalize method and then will do normal garbage collection. One can mention code to remove that specially allotted memory object in finalize().

➔ But how one can allot memory to some object without having **new.** As everything in java is an object. It happen because of native methods. Languages like C or C++ does that. So now we know that when to use finalize().

➔ Example :

```
class FClass
{
	boolean CheckOut = false;
	FClass(boolean checkout)
	{
		CheckOut = checkout;
	}
	void checkIn()
	{
		CheckOut = false;
	}
	protected void finalize()
	{
		if(CheckOut)
		{
			System.out.println("inside finalize");
		}
	}
}

public class FinalizeClass {

	public static void main(String[] args) {
		FClass fc = new FClass(true);
		fc.checkIn();
		new FClass(true);
		System.gc();
	}
}
Output : inside finalize
```

The termination condition is that all **Book** objects are supposed to be checked in before they are garbage collected, but in **main( )**, a programmer error doesn't check in one of the books. Without **finalize( )** to verify the termination condition, this can be a difficult bug to find.

- **How Garbage Collector works :**

➔ Most common technique for GC is reference counting. What is done here is a reference counter will be given to each object. This reference counter will indicate how many references are referencing to particular object. While JVM does garbage collector it goes through all the objects. When it found that reference counter for any object is 'zero' then that object will get collected. But its an slow technique.

➔ Other fast GC technique JVMs use is Adaptive garbage collector. To do that there are different variants. One of them is Stop-Copy. What is done here is JVM stops the program for GC and go through all the references present on stack and static memory. It will trace all the objects they point to and it also find references present in those objects and also trace those object which they point to. Then it will just copy all the live objects into another part of the heap and remaining objects will be garbage collected.

➔ The disadvantage of stop-copy(also called as copy collector) is that it creates two separate memory locations. Second when program becomes stable and generate no garbage still copy collect creates two memory location which is wasteful and to deal with this GC have another method (here where adaptive part of it come into play) which is called as mark-sweep.

➔ What mark-sweep does is jvm will go through all the references in stack and static memory and trace all the live objects accordingly. When it finds any live object then it will mark it with a flag but no sweep followed yet. Only when all the marking process is completed then the sweep occurs. During the sweep the dead objects are released.

➔ There are other possible speedup possible in JVM like just in time(JIT) compiler. Here this compiler will partially or fully compile the program in native machine code so that it doesn't need to be interpreted by JVM and thus runs faster.

- **Order of Initialization :**

➔ When one creates an object by calling **new()** on it then its constructor will be called but before that all the variables including construction of object variables. Example

```
class OrderOfInitialztn1
{
	int i;
	OrderOfInitialztn1(int a)
	{
		i = a;

		System.out.println("object"+a+" will have "+a+" value");
	}
}
class OrderOfInitialztn2
{
```

```
	int j;
	OrderOfInitialztn1 oi1 = new OrderOfInitialztn1(1);
	OrderOfInitialztn2()
	{
		System.out.println("value of j variable is "+j);
		j = 10;
		System.out.println("value of j variable is "+j);
		System.out.println("inside OrderOfInitialztn2 class");
		oi3 = new OrderOfInitialztn1(33);
	}
	void OrderOfInitFn()
	{
	System.out.println("inside the function");
	}
	OrderOfInitialztn1 oi2 = new OrderOfInitialztn1(2);
	OrderOfInitialztn1 oi3 = new OrderOfInitialztn1(3);
}
public class OrderOfInitialization {
	public static void main(String[] args) {
		OrderOfInitialztn2 ooi1 = new OrderOfInitialztn2();
		ooi1.OrderOfInitFn();
	}
}
Output :
object1 will have 1 value
object2 will have 2 value
object3 will have 3 value
value of j variable is 0
value of j variable is 10
inside OrderOfInitialztn2 class
object33 will have 33 value
inside the function
```

➔ When objects have been created using 'static' reference variables then scenario will be different as shown in the following example.

```
class StaticInitializatn1
{
	static int a;
	StaticInitializatn1(int i)
	{
		a = i;
		System.out.println("value is "+a);
	}
	void SIFunctn()
	{
		System.out.println("method("+a+")");
	}
}
class StaticInitializatn2
{
	static StaticInitializatn1 st1 = new StaticInitializatn1(1);
	StaticInitializatn2()
	{
		st2.SIFunctn();
	}
	static StaticInitializatn1 st2 = new StaticInitializatn1(2);
```

```
}
class StaticInitializatn3
{
	StaticInitializatn1 st3 = new StaticInitializatn1(3);
	static StaticInitializatn1 st4 = new StaticInitializatn1(4);
	StaticInitializatn3()
	{
		st4.SIFunctn();
	}
	static StaticInitializatn1 st5 = new StaticInitializatn1(5);

}
public class StaticInitializatn {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		StaticInitializatn2 stc2 = new StaticInitializatn2();
		StaticInitializatn3 stc3 = new StaticInitializatn3();
		StaticInitializatn2 stc4 = new StaticInitializatn2();
	}
}
Output :
value is 1
value is 2
method(2)
value is 4
value is 5
value is 3
method(3)
method(3)//method(3) because 'a' is static
```

In the above example we can see how the execution flow is going. The order of initialization is statics first, if they haven't been initialized already by previous object creation i.e. static initialization is executed only **once** always, after that the non-static one. Also we can see object creation with static reference is first executed. After that object creation with non-static reference(in example above st3) then constructors is executed.

➔ Static members of a class can be put inside a 'static block' which will be executed once. Example is shown below.

```
class StaticBlock1
{
	StaticBlock1(int i)
	{
		System.out.println("StaticBlock1("+i+")");
	}
	void fn(int a)
	{
		System.out.println("fn("+a+")");
	}
}

class StaticBlock2
{
	static StaticBlock1 st1;
```

```
	static StaticBlock1 st2;
	static
	{
		st1 = new StaticBlock1(1);
		st2 = new StaticBlock1(2);
	}
	StaticBlock2()
	{
		System.out.println("StaticBlock2");
	}
}

public class StaticBlock {
	public static void main(String[] args)
	{
		System.out.println("main method");
		StaticBlock2.st1.fn(1);//an static member can be accessed through class
					name.
	}

}
Output :
main method
StaticBlock1(1)
StaticBlock1(2)
StaticBlock2
fn(1)
```

➔ Similarly we can have non-static block as shown below in a example.

```
class NonStaticBlock1
{
	NonStaticBlock1(int i)
	{
		System.out.println("NonStaticBlock1("+i+")");
	}
	void fn(int a)
	{
		System.out.println("fn("+a+")");
	}
}
class NonStaticBlock2
{
	NonStaticBlock1 st1;
	NonStaticBlock1 st2;
	{
		st1 = new NonStaticBlock1(1);
		st2 = new NonStaticBlock1(2);
	}
	NonStaticBlock2()
	{
		System.out.println("NonStaticBlock2");
	}
}
public class NonStaticBlock {
	public static void main(String[] args)
	{
		NonStaticBlock2 nsb2 = new NonStaticBlock2();
```

```
	}
}
Output:
NonStaticBlock1(1)
NonStaticBlock1(2)
NonStaticBlock2
```

Above we can see that a non-static block starts with '{' and only difference is its not attached with a static keyword. Also we can see the non-static block will be executed before constructors.

➔ If we replace above programme as shown below then output will be different.

```
package ch5InitializationNCleanUp;

class StaticBlock1
{
	StaticBlock1(int i)
	{
		System.out.println("StaticBlock1("+i+")");
	}
	void fn(int a)
	{
		System.out.println("fn("+a+")");
	}
}

class StaticBlock2
{
	static StaticBlock1 st1;
	static StaticBlock1 st2;
	static StaticBlock1 st3;
	static StaticBlock1 st4;

	{
		st1 = new StaticBlock1(1);
		st2 = new StaticBlock1(2);
	}


	static
		{
			st3 = new StaticBlock1(3);
			st4 = new StaticBlock1(4);
		}

	StaticBlock2()
	{
		System.out.println("StaticBlock2");
	}
}

public class StaticBlock {
	public static void main(String[] args)
	{
		System.out.println("main method");
		StaticBlock2 st2 = new StaticBlock2();
		StaticBlock2.st3.fn(1);//an static member can be accessed through class
```

```
                                        name.
      }
}
Output:
main method
StaticBlock1(3)
StaticBlock1(4)
StaticBlock1(1)
StaticBlock1(2)
StaticBlock2
fn(1)
```

Above we could see that initialization of static block takes first before initialization of individual static object references.

Initialization takes place in a following way

1. Static variables(both object reference and primitive variables, it is first initialized because it is connected to class itself.
2. Non-static variables(both primitive variables and object references).
3. Finally constructors will be called from base to derived classes.

➢ **Array Initialization** :

➔ One can take array of type **Object** as a argument list which is also called in C as 'variable argument list' also called as varargs. Varargs include unknown quantities of arguments and as well as unknown types. Example is shown below.

```
class VarArgsClass1{}

public class VarArgsClass
      {
      static void printArray(Object [] args)
      {
            for(Object obj : args)
            System.out.println(obj);
            System.out.println();
      }
      public static void main(String [] args)
      {
      printArray(new Object[]{new Integer(1), new Integer(2), new Integer(3)});
      printArray((Object[])new Integer[]{1,2,3,4});//this works too
      printArray(new Object[]{"one", "two", "three"});
      printArray(new Object[]{new VarArgsClass1(),new VarArgsClass(),
                        new VarArgsClass1()});
      //printArray();//This can't be used here
      }
}
Output:
1
2
3
```

```
1
2
3
4

one
two
three

ch5InitializationNCleanUp.VarArgsClass1@e5b723
ch5InitializationNCleanUp.VarArgsClass@15a8767
ch5InitializationNCleanUp.VarArgsClass1@6f7ce9
```

➔ With Java SE5 now we can use ellipses to define a variable argument list or varargs. Example for the same is shown below.

```
class VarArgsClassEllipse1{}

public class VarArgsClassEllipse {
      static void printArray1(Object...args)
      {
            for(Object obj : args)
            System.out.println(obj);
            System.out.println();
      }
      public static void main(String[] args) {
            printArray1(new Integer(1), new Integer(2), new Integer(3));
            printArray1("one", "two", "three");
            printArray1(1.1, 2.2, 3.3);
            printArray1(new VarArgsClassEllipse1(), new VarArgsClassEllipse1()
            ,new VarArgsClassEllipse1());
            printArray1((Object[])new Integer[]{1,2,3,4});
            printArray1();
      }
}
Output:
1
2
3

one
two
three

1.1
2.2
3.3

ch5InitializationNCleanUp.VarArgsClassEllipse1@82ba41
ch5InitializationNCleanUp.VarArgsClassEllipse1@923e30
ch5InitializationNCleanUp.VarArgsClassEllipse1@130c19b

1
2
3
4
```

From above we can see that how using ellipses we can pass the arguments. Also in the program last but one printArray() method is using argument as autoboxing therefore it needed to be cast with Object so that compiler won't give error.

➔ Also in the previous example we can see that the last printArray() method didn't sent any argument at all but it is still working with Ellipses i.e. we can pass zero argument to the Ellipses. This will be useful when one will have optional trailing argument. Example for it is shown below.

```
public class OptionalTrailingArguments {
static void f(int required, String... trailing) {
System.out.print("required: " + required + " ");
for(String s : trailing)
System.out.print(s + " ");
System.out.println();
}
public static void main(String[] args) {
f(1, "one");
f(2, "two", "three");
f(0);
}
} /* Output:
required: 1 one
required: 2 two three
required: 0
```

➔ I above example one cannot use f(String...training, int required) then compiler will give error that " The variable argument type String of the method 'f' should be the last parameter. I.e. when there are two parameters ellipse or varargs should come after primitive type.

➔ Also there cannot be two varargs together like f(char...arg1, string...arg2).

➔ Ellipses can also be specific like for String array or int array. Example is shown below

Example :

```
package ch5InitializationNCleanUp;

class StringEllipseCls1
{
	void ellipseMeth1(String...str)
	{
		for(String s : str)
			System.out.println(s);
		System.out.println();
	}

	void ellipseMeth2(int...intArr)
	{
		for(int i : intArr)
			System.out.println(i);
		System.out.println();
```

```
	}
}

public class StringEllipseCls {

	public static void main(String[] args) {

		int [] intArr = {1,2,3};
		String [] strArr = {"John","David","Luke"};

		StringEllipseCls1 sec1 = new StringEllipseCls1();
		sec1.ellipseMeth1("one", "two", "three");
		//sec1.ellipseMeth(intArr);//ERROR
		sec1.ellipseMeth1(strArr);

		sec1.ellipseMeth2(intArr);
		//sec1.ellipseMeth2(strArr);//ERROR
	}
}
Output:
one
two
three

John
David
Luke

1
2
3
```

Order of Initialization with Separate Classes :

```
package ch5InitializationNCleanUp;

class OrderOfInitializationSeparateCls1
{
	static int a = 0;
	OrderOfInitializationSeparateCls1(int i)
	{
		a = i;
		System.out.println("value of a = "+a);
	}

	static int method(char c)
	{
		System.out.println("char val is "+c);
		return a;
	}
}

class OrderOfInitializationSeparateCls2
{
	static int i = OrderOfInitializationSeparateCls1.method('a');
	static
	{
```

```
            OrderOfInitializationSeparateCls1 ois14 = new
OrderOfInitializationSeparateCls1(4);
      }
      OrderOfInitializationSeparateCls1 ois11 = new
OrderOfInitializationSeparateCls1(1);
      static OrderOfInitializationSeparateCls1 ois12 = new
OrderOfInitializationSeparateCls1(2);


      static OrderOfInitializationSeparateCls1 ois13 = new
OrderOfInitializationSeparateCls1(3);
      int j = ois12.method('b');

      OrderOfInitializationSeparateCls2()
      {
            System.out.println("inside OrderOfInitializationSeparateCls2 const");
      }
}

class OrderOfInitializationSeparateCls3
{
      static int i = OrderOfInitializationSeparateCls1.method('c');
      static OrderOfInitializationSeparateCls1 ois15 = new
OrderOfInitializationSeparateCls1(5);
      OrderOfInitializationSeparateCls1 ois16 = new
OrderOfInitializationSeparateCls1(6);

      OrderOfInitializationSeparateCls3()
      {
            System.out.println("inside OrderOfInitializationSeparateCls3 const");
      }
}

public class OrderOfInitializationSeparateCls {

      public static void main(String[] args)
      {
            OrderOfInitializationSeparateCls3 ois31 = new
OrderOfInitializationSeparateCls3();
            OrderOfInitializationSeparateCls2 ois21 = new
OrderOfInitializationSeparateCls2();
      }

}
Output:
char val is c
value of a = 5
value of a = 6
inside OrderOfInitializationSeparateCls3 const
char val is a
value of a = 4
value of a = 2
value of a = 3
value of a = 1
char val is b
inside OrderOfInitializationSeparateCls2 const
```

Order Of Initialization with Inheritance :

```
package ch5InitializationNCleanUp;

class OrderOfInitializationSeparateCls1
{
	static int a = 0;
	OrderOfInitializationSeparateCls1(int i)
	{
		a = i;
		System.out.println("value of a = "+a);
	}

	static int method(char c)
	{
		System.out.println("char val is "+c);
		return a;
	}
}

class OrderOfInitializationSeparateCls2 extends OrderOfInitializationSeparateCls1
{
	static int i = OrderOfInitializationSeparateCls1.method('a');
	static
	{
		OrderOfInitializationSeparateCls1 ois14 = new
OrderOfInitializationSeparateCls1(4);
	}
	OrderOfInitializationSeparateCls1 ois11 = new
OrderOfInitializationSeparateCls1(1);
	static OrderOfInitializationSeparateCls1 ois12 = new
OrderOfInitializationSeparateCls1(2);


	static OrderOfInitializationSeparateCls1 ois13 = new
OrderOfInitializationSeparateCls1(3);
	int j = ois12.method('b');

	OrderOfInitializationSeparateCls2()
	{
		super(66);
		System.out.println("inside OrderOfInitializationSeparateCls2 const");

	}
}

class OrderOfInitializationSeparateCls3 extends OrderOfInitializationSeparateCls2
{
	static int i = OrderOfInitializationSeparateCls1.method('c');
	static OrderOfInitializationSeparateCls1 ois15 = new
OrderOfInitializationSeparateCls1(5);
	OrderOfInitializationSeparateCls1 ois16 = new
OrderOfInitializationSeparateCls1(6);

	OrderOfInitializationSeparateCls3()
	{
		System.out.println("inside OrderOfInitializationSeparateCls3 const");
```

```
	}
}

public class OrderOfInitializationSeparateCls {

	public static void main(String[] args)
	{
		OrderOfInitializationSeparateCls3 ois31 = new
OrderOfInitializationSeparateCls3();
		//OrderOfInitializationSeparateCls2 ois21 = new
OrderOfInitializationSeparateCls2();
	}

}
Output:
char val is a
value of a = 4
value of a = 2
value of a = 3
char val is c
value of a = 5
value of a = 66
value of a = 1
char val is b
inside OrderOfInitializationSeparateCls2 const
value of a = 6
inside OrderOfInitializationSeparateCls3 const
```

Order Of Initialization :

1) Static variable or reference or block whichever comes first
2) Non-static variable or references
3) Constructors

A) With Separate classes the order of initialization is from 1 to 3
B) For Inheritance Order of Initialization is 1 from base to derive and 2-3 together from base to derive through each class.