Chapter 7: Reusing Classes

➔ In case of inheritance we call base class methods from derived class using keyword 'super'. Ex : super.fn().

➔ Compiler put default constructor for each class but when one want to have two constructor with parameter and without parameter, Also when he want to call both of them then he have to explicitly create a default(non-para) constructor.

➔ We already know that when there are more than one constructor present then we can use 'this' keyword to call one constructor from another. In case of inheritance we know that when we create an object of a derived class then super class constructor(default or implicit) will be called automatically. But if user has declared a parameterized constructor and have not declared default constructor separately then there will be error. This error can be avoided by calling parameterized super class constructor using keyword **super** in derived class constructor. Example of these is show below.

```
package ch7ReusingClasses;

class SuperConstructor1
{
	SuperConstructor1(int i)
	{
		System.out.println("inside supcon1");
	}
}
class SuperConstructor2 extends SuperConstructor1
{
	SuperConstructor2(int j)
	{
		super(j);
		System.out.println("inside supcon2");
	}
}

public class SuperConstructor extends SuperConstructor2 {
	/**How super keyword is used inside constructors */
	SuperConstructor(int k)
	{
		super(11);
		System.out.println("inside supcon");
	}
	public static void main(String[] args) {
		SuperConstructor sp = new SuperConstructor(11);
	}
}
Output:
inside supcon1
inside supcon2
inside supcon
```

Note: If we remove super(j) or super(11) in derived class constructor then compiler give error

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
Implicit super constructor SuperConstructor2()(or SuperConstructor2()) is
undefined. Must explicitly invoke another constructor.
```

➔ Also one can avoid using ‘super’ if default constructors are defined by programmer.

➔ We know that inside a same class with many constructors we use ‘this’ to reuse the code and same can be done in inheritance through ‘super’.

➔ (KnS) When a overridden subclass method wants to use functionality of superclass method and wants to add some extra functionality then this could be done using keyword ‘super’ as shown below.

```
package ch7ReusingClasses;

class Animal
{
	void animalWalk()
	{
		System.out.println("All animal walks");
	}
}

class Wolf extends Animal
{
	void animalWalk()
	{
		super.animalWalk();
		System.out.println("Wolf crowls too");
	}
}

public class BaseMethodUsingSuper
{
	public static void main(String[] args)
	{
		Wolf w = new Wolf();
		w.animalWalk();
	}
}
Output:
All animal walks
Wolf crowls too
```

➔ When a base class method is overloaded several times in derived class then redefining that method in derive class will not hide any base class versions. Example

```
package ch7ReusingClasses;

class NameHidMethodCls1
{
	char doh(char c)
	{
		System.out.println("char(doh)");
		return 'a';
	}
	float doh(float i)
	{
		System.out.println("float(doh)");
		return 1.0f;
	}
}
```

```
class NameHidMethodCls2 {}

class NameHidMethodCls3 extends NameHidMethodCls1
{
	void doh(NameHidMethodCls2 nms2)
	{
		System.out.println("nms2(doh)");
	}
}

public class NameHidMethodCls {
	/** How derive class have access to all overloaded
	 * version of methods */
	public static void main(String[] args) {
		NameHidMethodCls3 nms3 = new NameHidMethodCls3();
		nms3.doh(1);
		nms3.doh('a');
		nms3.doh(1.0f);
		nms3.doh(new NameHidMethodCls2());
	}
}
```

Output:

```
float(doh)
char(doh)
float(doh)
nms2(doh)
```

➔ Also one will see that it's far more common to **override** methods of the same name, using exactly the same signature and return type as in the base class. Java SE5 added a new annotation @Override which prevents methods to get **overloaded.** Example

```
class Lisa extends Homer
 {
    @Override void doh(Milhouse m)
    {
       System.out.println("doh(Milhouse)");
    }
}
```

➔ When one has to chose between composition or inheritance it depends on many criteria. But if 'upcasting' is involved then one should use inheritance else one can decide on what to do.

➢ **<u>Order of initialization with inheritance</u> :**

```
package ch7ReusingClasses;

class BaseOrderOfInitilization1
{
	private int i = 9;
	protected int j;
	BaseOrderOfInitilization1()
	{
		System.out.println("i="+i+" "+"j="+j);
```

```
            j=39;
      }
      private static int x1 = printInit("static BaseOrdInit1.x1 initialized");
      static int printInit(String s)
      {
            System.out.println(s);
            return 47;
      }
}

class OrderOfInitialization extends BaseOrderOfInitilization1
{
      private int k = printInit("OrdInt is initialized");
      OrderOfInitialization()
      {
            System.out.println("k="+k);
            System.out.println("j="+j);
      }
      private static int x2 = printInit("static Ordint.x2 initialized");

      public static void main(String[] args) {
            // TODO Auto-generated method stub
            OrderOfInitialization oi = new OrderOfInitialization();
      }
}
Output:
static BaseOrdInit1.x1 initialized
static Ordint.x2 initialized
i=9 j=0
OrdInt is initialized
k=47
j=39
```

➔ (*)From above we could see that the order of initialization takes place as shown below

1. Static reference variables
2. Static primitive variables
3. Non-static reference variables
4. Non-static primitive variables
5. Constructors

Now above execution takes place separately in two different scenerio

A. With separate class : Execution take place from 1 to 5 class wise.

B. With Inheritance : With Inheritance execution from 1 to 2 take place from Base to derive and then 3 to 5 take place class wise.

Note : Base to Derive execution only take place when inheritance is present.

➔ Above can also be explained with the example given below

```
package ch8Polymorphism;

class OrderOfInitForVarNObjVar1

{
	OrderOfInitForVarNObjVar1()
	{

	}
	OrderOfInitForVarNObjVar1(int a)
	{
		System.out.println("OrderOfInitForVarNObjVar1 "+a);
	}
}

class OrderOfInitForVarNObjVar2 extends OrderOfInitForVarNObjVar1
{
	static OrderOfInitForVarNObjVar1 ov1 = new OrderOfInitForVarNObjVar1(1);

	OrderOfInitForVarNObjVar1 ov2 = new OrderOfInitForVarNObjVar1(2);
}

class OrderOfInitForVarNObjVar3 extends OrderOfInitForVarNObjVar2
{
	static OrderOfInitForVarNObjVar1 ov3 = new OrderOfInitForVarNObjVar1(3);

	OrderOfInitForVarNObjVar1 ov4 = new OrderOfInitForVarNObjVar1(4);
}

public class OrderOfInitForVarNObjVar {
	public static void main(String[] args)
	{
		OrderOfInitForVarNObjVar3 ovn1 = new OrderOfInitForVarNObjVar3();

	}
}
Output:
OrderOfInitForVarNObjVar1 1
OrderOfInitForVarNObjVar1 3
OrderOfInitForVarNObjVar1 2
OrderOfInitForVarNObjVar1 4
```

Note: In above example default constructor will be executed for `OrderOfInitForVarNObjVar1` class.