**/\***Use Of **SUPER** Keyword with Instance **Variable**…**\*/**

**class A**

**{**

**int i;**

**}**

**class B extends A**

**{**

**int i; //Overrides ‘i’ Of SuperClass.**

**void set\_data(int a,int b)**

**{**

**super.i = a;**

**i = b;**

**}**

**void get\_data()**

**{**

**System.out.println("SuperClass i : "+super.i);**

**System.out.println("SubClass i : "+i);**

**}**

**}**

**public class Super\_with\_Variables**

**{**

**public static void main(String[] args) {**

**B b = new B();**

**b.set\_data(10,20);**

**b.get\_data();**

**b.i = 87;**

**b.get\_data();**

**}**

**}**

**Output:**

SuperClass i : 10

SubClass i : 20

SuperClass i : 10

SubClass i : 87

**/\***Use **SUPER** Keyword with **Methods**…**\*/**

**class A**

**{**

**int a;int b;**

**A(int a,int b)**

**{**

**this.a = a;**

**this.b = b;**

**}**

**void display()**

**{**

**System.out.println("A : "+a+" B : "+b);**

**}**

**}**

**class B extends A**

**{**

**int c;**

**B(int a,int b,int c)**

**{**

**super(a,b);**

**this.c = c;**

**}**

**void display() //Overrides The Method display() of The SuperClass A....**

**{**

**System.out.println("C : "+c);**

**}**

**}**

**public class Super\_with\_Methods**

**{**

**public static void main(String[] args) {**

**B b = new B(4,5,90);**

**b.display();**

**}**

**}**

**Output:**

A : 4 B : 5                                            C : 90

**/\***When **Constructors** Are Executed..??**\*/**

**class A**

**{**

**A()**

**{**

**System.out.println("Inside A's Constructor.!");**

**}**

**}**

**class B extends A**

**{**

**B()**

**{**

**System.out.println("Inside B's Constructor.!");**

**}**

**}**

**class C extends B**

**{**

**C()**

**{**

**System.out.println("Inside C's Constructor.!");**

**}**

**}**

**public class Constructors\_excutn**

**{**

**public static void main(String[] args) {**

**B b = new B();**

**System.out.println("\n");**

**C c = new C();**

**}**

**}**

**Output:**

Inside A's Constructor.!

Inside B's Constructor.!

Inside A's Constructor.!

Inside B's Constructor.!

Inside C's Constructor.!

**/\*Method Overriding… \*/**

**class A**

**{**

**int a;int b;**

**A(int a,int b)**

**{**

**this.a = a;**

**this.b = b;**

**}**

**void display()**

**{**

**System.out.println("A : "+a+" B : "+b);**

**}**

**}**

**class B extends A**

**{**

**int c;**

**B(int a,int b,int c)**

**{**

**super(a,b);**

**this.c = c;**

**}**

**void display() //Overrides The Method display() of The SuperClass A....**

**{**

**System.out.println("C : "+c);**

**}**

**}**

**public class Overriding\_Demo**

**{**

**public static void main(String[] args) {**

**B b = new B(4,5,90);**

**b.display();**

**}**

**}**

**Output:**

C : 90

**/\*Multiple Generics\*/**

**class A <T1,T2>**

**{**

**T1 ob1;**

**T2 ob2;**

**A (T1 o1,T2 o2)**

**{**

**ob1 = o1;**

**ob2 = o2;**

**}**

**T1 retType1()**

**{**

**return ob1;**

**}**

**T2 retType2()**

**{**

**return ob2;**

**}**

**void display()**

**{**

**System.out.println("Value Of Ob1 : "+ob1+"\nValue Of Ob2 : "+ob2);**

**}**

**void showType()**

**{**

**System.out.println("Typ1 Of T1 : "+ob1.getClass().getName());**

**System.out.println("Type Of T2 : "+ob2.getClass().getName());**

**}**

**}**

**public class Generics\_Demo**

**{**

**public static void main(String[] args) {**

**A<Integer,Character> m1 = new A<Integer,Character>(10,'+');**

**m1.display();**

**m1.showType();**

**}**

**}**

**Why Overridden Methods..???**

Overridden methods allow java to support Run-time polymorphism. Polymorphism is essential for Object-Oriented Programming for one reason. i.e., it allows a general class to specify a method that will be common to all of its derivatives.

Overridden methods are another way that Java implements “one interface, multiple methods” aspect of polymorphism.

Dynamic, run-time polymorphism is one of the most powerful mechanism that Object-Oriented design brings to bear on code reuse & Robustness.

**When Constructors Are Executed..??**

In a class hierarchy, constructors complete their execution in order of derivation, from super class to sub class. Further, since **super ()** must be the first statement in the subclass constructor, these order is same whether or not the **super ()** is used.

If **super ()** is not used than the **default** or **parameter-less** constructor of each super class will be executed.

**Dynamic Method Dispatch:**

Super class reference can hold a subclass object. Such reference can accept/access only the common properties of both the Super & Sub classes and not the Exclusive Property.

**Generics:**

In Java Generic methods & Generic class allow the programmer to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provides Compile time type safety, which allows the programmers to catch invalid types at compile time.

**Generic Class:** Generic class declaration is similar to as that of non-generic except the class name is followed by a type parameter section.

The type parameter section can have one or more type parameters separated by commas.

**Abstract Classes:**

* Abstract Method – Method with No Definition is called as abstract method.
* Abstract class must have at least one abstract method.
* Super class requires/mandates that a method to defined in all of its subclasses.
* Abstract classes cannot Be Instantiated i.e., No Object can be made for this class.

**E.g:**

**abstract class A**

**{**

**int a;**

**A(){}**

**A (int a1)**

**{**

**a = a1;**

**}**

**abstract void calc ();**

**}**

**class B extends A**

**{**

**int b;**

**B(int x,int y)**

**{**

**a=x;**

**b=y;**

**}**

**public void calc()**

**{**

**System.out.println("A : "+a+" B : "+b);**

**}**

**}**

**public class Abstract\_Demo**

**{**

**public static void main (String[]args)**

**{**

**A a;**

**B b1 = new B(10,20);**

**b1.calc();**

**}**

**}**

**Output:**

A : 10 B : 20

**Finding Packages & Class-Path:**

In Java, Packages are mirrored as directories. So How the Java Run-Time system know where to look for the package that you have created…

There are 3 parts:

1. By Default, the Java Run-Time system uses the current working directory as its starting point. Thus if your package is in a subdirectory of current directory, it will be found.
2. You can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
3. You can use the **–classpath** option with **java** or **javac** to specify the path of your classes.

For example, consider the following package specification:

package mypack;

* In order for a program to find **mypack**, the program can be executed from a
* directory immediately above **mypack**, or the **CLASSPATH** must be set to
* Include the path to **mypack**, or the **-classpath** option must specify the path to
* **mypack** when the program is run via **java**.
* When the second two options are used, the class path *must not* include
* **mypack**, itself. It must simply specify the *path to* **mypack**. For example, in a
* Windows environment, if the path to **mypack** is

C:\MyPrograms\Java\mypack

then the class path to **mypack** is

C:\MyPrograms\Java

* package directories below your current development directory, put the **.class**