Unit – 2

2. A Closer Look at Methods and Classes

2.1 Overloading Methods

- If two or more method in a class have same name but different parameters, it is known as method overloading. In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading

In order to overload a method, the parameter lists of the methods must differ in either of these:

- Number of parameters
- Data type of parameters
- Sequence of data type of parameters

- Method overloading is one of the ways that Java supports polymorphism. When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters.

- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

-

```
// Method overloading by changing data type of parameters
class Calculate {
        void sum (int a, int b) {
                System.out.println("sum is"+(a+b)) ;
        }
        void sum (float a, float b){
                System.out.println("sum is"+(a+b));
        }
        public static void main (String [] args) {
                Calculate cal = new Calculate ();
                cal.sum (8,5); //sum (int a, int b) is method is called.
                cal.sum (4.6f, 3.8f); //sum (float a, float b) is called.
        }
}
```

Output:

sum is 13

Overloading Constructors:

- In addition to overloading normal methods, constructors can also be overloaded.
- Constructor overloading is a concept of having more than one constructor with different parameters list, in such a way so that each constructor performs a different task.
- They are differentiated by the compiler by the number of parameters in the list and their types. Constructor overloading is done to construct object in different ways.

```
class Box
{
        double width;  double height; double depth;
        Box(double w, double h, double d)
        {       width = w;      height = h; depth = d;
        }
        Box()
        {
                width = -1; // use -1 to indicate
                height = -1; // an uninitialized
                depth = -1; // box
        }
        // constructor used when cube is created
        Box(double len)
        {       width = height = depth = len;
        }
        // compute and return volume
        double volume()
        {       return width * height * depth;
        }
}

class OverloadCons
{
                public static void main(String args[])
                {       // create boxes using the various constructors
                        Box mybox1 = new Box(10, 20, 15);
                        Box mybox2 = new Box();
                        Box mycube = new Box(7);
                        double vol;
                        // get volume of first box
                        vol = mybox1.volume();
                        System.out.println("Volume of mybox1 is " + vol);
                        // get volume of second box
                        vol = mybox2.volume();
                        System.out.println("Volume of mybox2 is " + vol);
                        // get volume of cube
                        vol = mycube.volume();
                        System.out.println("Volume of mycube is " + vol);
                }
```

Output:

```
Volume of mybox1 is  3000
Volume of mybox2 is -1
```

```
}
```

2.2 Using Objects as Parameters

So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods.

```
// Objects may be passed to methods.
class Test
{
        int a, b;
        Test(int i, int j)
        {
                a = i;
                b = j;
        }
        // return true if o is equal to the invoking object
        boolean equals(Test o)
        {
                if(o.a == a && o.b == b) return true;
                else return false;
        }
}

class PassOb
{
        public static void main(String args[])
        {
                Test ob1 = new Test(100, 22);
                Test ob2 = new Test(100, 22);
                Test ob3 = new Test(-1, -1);
                System.out.println("ob1 == ob2: " + ob1.equals(ob2));
                System.out.println("ob1 == ob3: " + ob1.equals(ob3));
        }
}
```

Output:
ob1 == ob2:true
ob1 == ob2: false

A closer look at arguments passing

In general, there are two ways that a computer language can pass an argument to a subroutine.

- The first way is call-by-value. This approach copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.

- The second way an argument can be passed is call-by-reference. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

Although Java uses call-by- value to pass all arguments, the precise effect differs between whether a primitive type or a reference type is passed.

- When a primitive type is passed to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method.

```
// Simple types are passed by value.
class Test
{          void meth(int i, int j)
          {       i *= 2;
                  j /= 2;
          }
}
class CallByValue
{
          public static void main(String args[])
          {
                  Test ob = new Test();
                  int a = 15, b = 20;
                  System.out.println("a and b before call: " + a + " " + b); // 15 20
                  ob.meth(a, b);
                  System.out.println("a and b after call: " +a + " " + b); // 15 20
          }
}
```

Output:
a and b before call:15 20
a and b after call:15 20

- When an object is passed to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. When a variable of a class type is created a reference to an object is created. Thus, when this reference is passed to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects act as if they are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument.

```
// Objects are passed by reference.
class Test
{
    int a, b;
    Test(int i, int j)
    {
            a = i;
            b = j;
    }
    // pass an object
    void meth(Test o)
    {
            o.a *= 2;
```

Output:
ob.a and ob.b before call: 15 20
ob.a and ob.b after call:30 10

```
                o.b /= 2;
        }
    }

    class CallByRef
    {
        public static void main(String args[])
        {
                Test ob = new Test(15, 20);
                System.out.println("ob.a and ob.b before call: " +ob.a + " " + ob.b);
    // 15 20
                ob.meth(ob);
                System.out.println("ob.a and ob.b after call: " +ob.a + " " + ob.b);
                  // 30 10
        }
    }
```

2.3 Returning Objects

A method can return any type of data, including class types.

```
// Returning an object.
class Test
{
        int a;
        Test(int i)
        {
                a = i;
        }
        Test incrByTen()
        {
                Test temp = new Test(a+10);
                return temp;
        }
}

class RetOb
{
        public static void main(String args[])
        {
                Test ob1 = new Test(2);
                Test ob2;
                ob2 = ob1.incrByTen();
                System.out.println("ob1.a: " + ob1.a); // 2
                System.out.println("ob2.a: " + ob2.a); //12
                ob2 = ob2.incrByTen();
                System.out.println("ob2.a after second increase: "+ ob2.a); // 22
```

Output:
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22

```
        }
    }
```

2.4 Introducing Access Control

Encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: access control. Through encapsulation, what parts of a program can access the members of a class can be controlled. By controlling access, misuse can be prevented.

For example, allowing access to data only through a well-defined set of methods, misuse of the data can be prevented. The access modifiers in java specify accessibility (scope) of a data member, method, constructor or class. There are four access modifiers in java which are (table 2.1):

1. Default Access Modifier - No Keyword
2. Private Access Modifier - private
3. Protected Access Modifier – protected
4. Public Access Modifier – public

Modifier	Description
Private	Declarations are visible within the class only
Default	Declarations are visible only within the package (package private)
Protected	Declarations are visible within the package or and all sub classes
Public	Declarations are visible everywhere

Table 2.1: Access Modifiers

- When a member of a class is modified by public, then that member can be accessed by any other code.
- When a member of a class is specified as private, then that member can only be accessed by other members of its class.
- When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.
- When private modifier is used, then that member can only be accessed by the members of same class and the members of sub classes.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

1. Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A
{
  void msg()
{
System.out.println("Hello");
}
}

//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
 }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package

Public:

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

```
/save by A.java
```

```
package pack;
public class A
{
public void msg()
{
System.out.println("Hello");
}
}
//save by B.java

package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
```

```
    System.out.println(obj.data);//Compile Time Error
    obj.msg();//Compile Time Error
    }
}
```

Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method. It can't be applied on the class.

It provides more accessibility than the default modifer.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}

//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

2.5 Understanding static

- There will be times when a class member needs to be defined that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However,

it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword static.

- The static can be:
- Variable (also known as a class variable)
- Method (also known as a class method)
- Block
- Nested class

- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.

- The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist. Instance variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

To call static member use the following syntax:

```
clsName.varName;
clsName.methodName(args);
```

Only one copy of variable is created.
For static variable,
int is initialized to 0,
float to 0.0,
boolean initialized to false,
String to null

Example:

```
class Counter
    {       int c1=0;
            static int c2;
            Counter ()
            {       c1++;
                    c2++;
            }
            void disp()
            {       System.out.print("First Counter : "+c1);
                    System.out.println("   Second Counter : "+c2);
       }
    }
    class prg3
    {               public static void main(String args[])
                   {        Counter  o1 = new Counter();
```

Output:

```
First Counter : 1   Second Counter : 1
First Counter : 1   Second Counter : 2
First Counter : 1   Second Counter : 3
```

```
                Counter  o2 = new Counter();
                Counter  o3 = new Counter();
                    o1.disp();
                o2.disp();
                o3.disp();
            }
        }
```

- Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.
- static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

- When we require to do computation in order to **initialize static variables,** we can declare a **static block** that gets executed **exactly once,** when the **class is first loaded or when there is a reference to the class.**

- **If static method and main method is within same class then static method can be invoked without an object otherwise static method is called by classname.staticmethodname.**

```
// Demonstrate static variables, methods, and blocks.
class UseStatic
{
        static int a = 3;
        static int b;
        static void meth(int  x)
        {       System.out.println("x = " + x);
                System.out.println("a = " + a);
                System.out.println("b = " + b);
        }
        static {
                System.out.println("Static block initialized.");
                b = a * 4;
            }
        public static void main(String args[])
        {       meth(42);
        }
}
```

Output:

```
Static block initialized.
        x = 42
        a = 3
        b = 12
```

Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.

- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

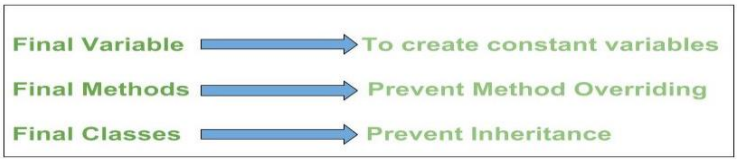
```
Class Student{
    int rollno;
    String name;
    static String college = "ITS";
    //static method to change the value of static variable
    static void change(){
    college = "BBDIT";
    }
    //constructor to initialize the variable
    Student(int r, String n){
    rollno = r;
    name = n;
    }
    //method to display values
    void display(){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to create and display the values of object
public class TestStaticMethod{
   public static void main(String args[]){
   Student.change();//calling change method
   //creating objects
   Student s1 = new Student(111,"Karan");
   Student s2 = new Student(222,"Aryan");
   Student s3 = new Student(333,"Sonoo");
   //calling display method
   s1.display();
   s2.display();
   s3.display();
   }
}
```

Test it Now

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
```

2.6 Introducing final

Final keyword is used in different contexts. First of all, *final* is a non-access modifier applicable **only to a variable, a method or a class**. Following are different contexts where final is used:



- If you make any variable as final, you cannot change the value of final variable(It will be constant).
- If you make any method as final, you cannot override it.
- If you make any class as final, you cannot extend it.

2.6.1 Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

```
Output:Compile Time Error
```

2.6.2 Java final method

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}
```

```
  public static void main(String args[]){
  Honda honda= new Honda();
  honda.run();
  }
}
```

Test it Now

```
Output:Compile Time Error
```

2.6.3 Java final class

If you make any class as final, you cannot extend it.

```
final class Bike{}

class Honda1 extends Bike{
 void run(){System.out.println("running safely with 100kmph");}

 public static void main(String args[]){
 Honda1 honda= new Honda1();
 honda.run();
 }
}
```

Output:compile time error

2.7 Introducing Nested and Inner Classes

Inner classes logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods. A non-static class that is created inside a class but outside a method is called member inner class.

Syntax of Inner class

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called *static nested classes*. Non-static nested classes are called *inner classes*.

```
class OuterClass
{
        static class StaticNestedClass
        {
                ...
        }
        class InnerClass
        {
                ...
        }
}

class Outer
{       int  outer_x = 100;
        void test()
        {       Inner  inner = new Inner();
                inner.display();
        }
        //inner class
        class Inner
        {       void display()
                {     System.out.println("display: outer_x = " + outer_x);
                }
        }
}
class InnerClassDemo
{       public static void main(String args[])
        {       Outer  outer = new Outer();
                outer.test();
        }
}

class Outer
{       int outer_x = 100;
        void test()
        {       Inner  inner = new Inner();
                inner.display();
        }
    void showY()
        {       System.out.println(y);     // error, y not known here!
        }
        class Inner
        {       int y = 10;  //y is local to Inner
```

```
                void display()
                {       System.out.println("display: outer_x = " + outer_x);
                }
        }
}
class InnerClassDemo
{       public static void main(String args[])
        {       Outer  outer = new Outer();
                outer.test();
        }
}
```

2.8 Using Command-Line Arguments

Command line arguments is a methodology using which user will give inputs through the console using commands. The passed information is stored as a string array in the main method. Command Line Arguments can be used to specify configuration information while launching your application.
There is no restriction on the number of java command line arguments. You can specify any number of arguments.

```
//Display all command line arguments
  class CommandLine
  {
        public static void main(String args[])
        {
                for(int i=0; i< args.length; i++)
                System.out.println("args[" + i + "]: " + args[i]);
        }
  }
```

Output:
args[0] :this
args[1]:is
args[2]:a
args[3]:test
args[4]:100
args[5]:-1

Program execution at command prompt:
Java Command Line this is a test 100 -1

```
Output:
Static block initialized.
        x = 42
        a = 3
        b = 12
```

2.9 Varargs: Variable-Length Arguments

Varargs is a short name for variable arguments. In Java, an argument of a method can accept arbitrary number of values. This argument that can accept variable number of values is called varargs.

If we don't know how many argument we will have to pass in the method, varargs is the better approach. // method body

Advantage of Varargs:

We don't have to provide overloaded methods so less code.

```
class VarargsExample1{

static void display(String... values){
 System.out.println("display method invoked ");
}

public static void main(String args[]){

display();//zero argument
display("my","name","is","varargs");//four arguments
}
}
```

```
Output:display method invoked
       display method invoked
```

```
class VarargsExample2{

static void display(String... values){
 System.out.println("display method invoked ");
 for(String s:values){
  System.out.println(s);
 }
}

public static void main(String args[]){

display();//zero argument
display("hello");//one argument
display("my","name","is","varargs");//four arguments
}
}
```

```
Output:display method invoked
       display method invoked
       hello
       display method invoked
       my
       name
       is
```

varargs

Inside display, values is operated on as an array. It tells the compiler that a variable n umber of arguments will be used which will be stored in the array referred to by valu es. In main(), display is called with **different number of arguments** including no ar guments at all. The arguments are **automatically put in an array** and passed to valu es. In case of **no arguments**, the length of the array is zero.

INHERITANCE

Inheritance Basics

What is inheritance?

- Inheritance is a mechanism in which one object acquires all the properties and behaviors of parent object.

(a) (b)

- Inheritance represents the IS-A relationship, Example Rectangle is a shape

Why use Inheritance

- To achieve Runtime Polymorphism using method overriding
- For code reusability.

Methods of Inheritance

- By extending a class
- By implementing an interface

i) Inheritance By extending a class:

Syntax:

classSubclassName extends SuperclassName

{
//methods and fields
}

ii) Inheritance By implementing a interface:

Syntax:

Class SubclassName implements interfaceName
{
//methods and fields
}

Types of Inheritance

On the basis of class, there can be three types of inheritance: single, multilevel and hierarchical, multiple and hybrid is supported through interface only. We will learn about interface later.

Simple Inheritance

```
Class ClassA {
     //methods and fields of ClassA class
}
class ClassB extends ClassA {

     //methods and fields of ClassB class
}
```

Multilevel Inheritance

```
Class ClassA {
     //methods and fields of ClassA class
                          }
class ClassB extends ClassA {

     //methods and fields of ClassB class
}
Class classC extends ClassB{

//methods and fields of ClassC class
}
```

Hierarchical Inheritance

```
Class ClassA {
     //methods and fields of ClassA class
}
class ClassB extends ClassA {
     //methods and fields of ClassB class
}
Class classC extends ClassA{
//methods and fields of ClassC class
}
```

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private.

In a class hierarchy, private members remain private to their class. It is not accessible by any code outside its class, including subclasses.

Example*// Create a super class.*

```
class A {
        int i; // public by
        default private int j;
        // private to A void
        setData (int x, int y)
        {
              i = x; j = y;
        }
 }
class B extends A {          // A's j is not accessible here.
      i
      n
      t
      t
      o
      t
      a
      l
      ;
      v
      o
      i
      d
      s
      u
      m
      (
      )
```

```
        {
            total = i + j;                    // ERROR, j is not accessible here
        }
}
class Access {

        public static void main(String
            args[]) {
          B ObjectB = new B();
           ObjectB.setData(10, 12);
            ObjectB.sum();
            System.out.println("Total is " + ObjectB.total);
        }
}
```

Java has a special access modifier known as ***protected*** which is meant to support Inherit ance in Java. Any protected member including protected method and field are accessible for classes of same package and only Sub class outside the package.

Constructors and Inheritance

When both the superclass and subclass define constructor, the process is a bit more complicated because both the superclass and subclass constructor must be executed .

In this case java's keyword '***super***' can be used.

- First calls the superclass constructor
- Second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to call superclass constructor

A subclass can call a constructor defined by its superclass by use of the following form of super

Syntax

- Super(*parameter_list*);
  - *parameter_list* specifies any parameter needed by the constructor in superclass().
  - super() must be the first statement executed inside a subclass constructor.

Example 1:

```
class SuperClass {
        SuperClass( ) {
                System.out.println("SuperClass is created");
        }
}

class Subclass
        extends
```

```
        SuperClas
        s {
        SubClass(
        ) {
        super( );
                System.out.println("SubClass is created");
        }
}
class Demo {
        public static void
        main(String
        args[]){ Demo
        d=new Demo();
}
}
```

Output : SuperClass is created SubClass is created

Example 2:

```
class Person{
    i
    n
    t
    i
    d
    ;
    S
    t
    r
    i
    n
    g
    n
    a
    m
    e
    ;
    Person(int
        id,Str
        ing
        name
        ){
        this.i
        d=id;
        this.n
        ame=
        name
        ;
```

```
        }
}
class Emp
        exte
        nds
        Pers
        on{
        float
        sala
        ry;
        Emp(int id,String name,float salary){
                super(id,name);      //reusing
                parent const this. salary=salary;
                }
        void display( ){
                System.out.println(id+" "+name+" "+salary); } }
class TestSuper {
        public static void
        main(String[] args){ Emp
        e1=new
        Emp(1,"ankit",45000f);
        e1.display();
        }
}
```

Output:
1 ankit 45000

Using super to access super class members

In Java 'super' keyword can be used to refer super class instance in a subclass. 'super' when used to access members need not be first line.

Syntax :

Super.memberVariable
Super.memberMethod( )

Example: //Accessing overridden instance method

```
class Bank
{
 String
        name
        ;
   void
        show
        ( )
     {
                System.out.println("Customer Name = " + name);
        }
}
class   Customer
extends Bank
```

```
 {
String name;
        void setName(String custName, String bankName)
        {
        super.name = bankName;//sets instance variable of
        super           name = custName;
        }
        void show () {
                super.show();           //calls method of super class
                System.out.println("Customer Name = " + name);
        }
}
class BankAccount{
Customer cust1 = new Customer();
cust1.setName ( "Anil", "ICICI" );
cust1.show ( );
}
```

Creating a Multilevel Hierarchy

You can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another.

Order of constructor execution in multilevel inheritance

- When a class hierarchy is created, then the *"constructors are called in order of derivation, from super class to subclass"*.
- If super is used to call constructor, since super( ) must be the first statement executed in a subclass‟ constructor, this order is the same whether or not super( ) is used.
- If super() is not used, then the ***default constructor*** of each super class will be executed. Subclass constructor is the last to execute.

Example:

cla

s

s

C

l

a

s

s

A

```
{
C
l
a
s
s
A
(
)
{
        System.out.println("Inside A'sconstructor.");
  }
}
class ClassB
   extends
   ClassA {
   ClassB( ) {
        System.out.println("Inside B's constructor.");
      }
}

class ClassC
        extends
        ClassB {
        ClassC( )
        {
        System.out.println("Inside C's constructor.");
      }
}
class Demo {
        public static void
        main(String[] args) { ClassC
        obj = new ClassC( );
        }
}
```

Output: Inside A's constructor.
Inside B's constructor.
Inside C's constructor.

When are Constructors Executed

Constructors are executed in order of derivation.

- Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by subclass.

```
class ClassB
{
        ClassB( ) {
                System.out.println("Inside B's constructor.");
        }
```

```
}
class ClassC
        extends
        ClassB {
        ClassC( )
        {
                System.out.println("Inside C's constructor.");
        }
}
class Demo {
        public static void main(String[ ] args) {
        ClassC obj = new ClassC( ); //constructs a C object
        }
}
```

Output:

```
Inside B's constructor.
Inside C's constructor.
```

Superclass Reference and Subclass Objects

Upper class reference variable can point to Sub Class Object e.g.

SuperClass parentOb = new SubClass();

If you want to store object of Sub class, which is stored in super class reference variable as above, back to Sub class reference then you need to use casting, as shown below :

SubClass childOb = (SubClass) parent; *//since parent variable pointing to SubClass object*

Accessing subclass overloaded methods using super reference

Example

```
class Person{

   public String name;

   public int age;

   public Person(String name, int age){

      this.name = name;

      this.age = age;

   }

   public void displayPerson() {

      System.out.println("Data of the Person class: ");

      System.out.println("Name: "+this.name);

      System.out.println("Age: "+this.age);
```

```
    }
}
public class Student extends Person {
   public String branch;
   public int Student_id;
   public Student(String name, int age, String branch, int
Student_id){
      super(name, age);
      this.branch = branch;
      this.Student_id = Student_id;
   }
   public void displayStudent() {
      System.out.println("Data of the Student class: ");
      System.out.println("Name: "+super.name);
      System.out.println("Age: "+super.age);
      System.out.println("Branch: "+this.branch);
      System.out.println("Student ID: "+this.Student_id);
   }
   public static void main(String[] args) {
      Person person = new Student("Krishna", 20, "IT", 1256);
      person.displayPerson();
   }
}
```

Output

```
Data of the Person class:
Name: Krishna
Age: 20
```

Method Overriding

Overloading	Overriding
Static polymorphism	Runtime polymorphism
Same name different signature	Same name same signature
Return type can be different	Return type also should be same
Can be overloaded in same or in a subclass	Only be overridden in a subclass

- **Rules for overriding**
  1. Instance methods can be overridden only if they are inherited by the subclass.
  2. private, static and final methods cannot be overridden in Java..
  3. Constructors cannot be overridden.

```
class Animal
{
 public void move()
{
        System.out.println("Animals can move");
}
 }

        class Dog
        extends
        Animal
        {
         public void
        move()
        {
        System.out.println("Dogs can walk and run"); }

        }
        public class TestDog{
        public static void main(String args[]){
        Animal bullDog = new Dog(); // Animal reference but Dog object
        bullDog.move(); //Runs the method in Dog class
        }
        }
```

Overridden Methods support polymorphism or (Dynamic Method Dispatch)
Method overriding is a Means of run time polymorphism. Which method to call is based on object type, at runtime time.
Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

Method to execute is based upon the type of the object being referred to at the time the call occurs.

```
class Animal
{
 public void move()
{
System.out.println("Animals can move");
  }
}

class
Dog
extends
Animal
{
public
void
move()
{
System.out.println("Dogs can walk and run");
   }
}

class
Snake
extends
Animal
{
public
void
move()
{
System.out.println("Snake can't walk But run");
}
}
public class TestAnimal
{
public static void
      main(String
      args[])
{
 Animal animal =
      newAnimal();
```

```
Snake cobra
      =
      newSna
      ke();
 Dog
      bullDog
      = new
      Dog();
animal.move();    //Runs
move in Animal animal =
bullDog; // animal refersto
Dog  animal.move();
                          //Runs
move in Dog animal = cobra;
// animal refers to Snake
animal.move();    //Runs
move in Snake
  }
}
```

Why Overridden Methods

It allows a general class to specify methods that will b common to all derivaties, while allowing subclasses to define the specific implementation of some or all of those methods.

Overridden methods are another way that java implements the "one interface, multiple methods"

Using Abstract Classes

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon)

An abstract method has no implementation. It just has a method signature.

Syntax: *abstract return-type function-name( );*

You declare a method abstract by adding the abstract keyword in front of the method declaration.

Example: *abstract public double area( );*

Abstract Class:

- The purpose of an abstract class is to specify the default functionality of an object and let its sub-classes to explicitly implement that functionality.
- ***Java Abstract classes are*** used ***to declare common characteristics of subclasses.***
- It can only be used as a superclass for other classes that extend the abstract class.

Syntax: Abstract classes are declared with the abstract keyword

```
public abstract class MyabstractClass
{
        //code
}
```

An abstract class cannot be instantiated.

MyAbstractClass myClassinstance = new MyAbstractClass( );//not valid

Abstract classes are used to provide a template or design for concrete subclasses down the inheritance tree.

Using final

a) final class
b) final method
c)final variable

final class:

Restrict the inheritance, A final class cannot be subclassed.
All methods in a final class are implicity final.
Syntax:

```
final class className {
        //code
}
```

Example

```
public final class MyFinalClass {
//some code
}
public class WrongSubClass extends MyFinalClass {      // forbidden
}
```

- Final keyword helps to write immutable class.
- Immutable classes are the one which cannot be modified once it gets created
- String is primary example of immutable and final class .

Final method

A final method cannot be overridden by subclasses.
This is used to prevent unexpected behaviour from a subclass altering a method that may becrucial to the function or consistency of the class
You should make a method final in java if you think it's complete and its behavior should remain constant in sub-classes
Final methods are faster than non-final methods because they are not required to be resolved during run-time and they are bonded on compile time.

Final methods are bonded during compile t1ime also called static binding