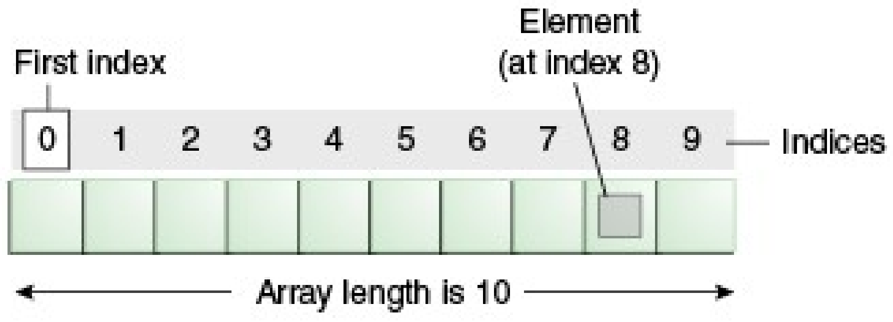
Arrays

- An *array* is a container object that holds a fixed number of values of a single type.
- The length of an array is established when the array is created. After creation, its length is fixed.
- Array elements are automatically initialized with the default value of their type, When an array is created



DECLARING A VARIABLE TO REFER TO AN ARRAY

```
int[] anArray;
byte[] anArrayOfBytes;
short[] anArrayOfShorts;
long[] anArrayOfLongs;
float[] anArrayOfFloats;
double[] anArrayOfDoubles;
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

```
dataType[] arr;
dataType []arr;
dataType arr[];
```

```
// this form is discouraged
float anArrayOfFloats[];
```

CREATING, INITIALIZING, AND ACCESSING AN ARRAY

One way to create an array is with the new operator.

```
int[] anArray;
```

```
// create an array of integers
anArray = new int[10];
```

The statement allocates an array with enough memory for 10 integer elements and assigns the array to the **anArray** variable.

CREATING, INITIALIZING, AND ACCESSING AN ARRAY

Initialization

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
```

Accessing Values

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
```

CREATING, INITIALIZING, AND ACCESSING AN ARRAY

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = { 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};
```

```
int[] anArray = new int[]{ 100, 200, 300, 400, 500, 600, 700, 800,
900, 1000};
```

Quiz

Select which of the following are valid array definition

1. int[] a;

    a = new int[5];

2. int a[] = new int[5]

3. int a[5] = new int[5];

4. int a[] = {1,2,3};

5. int[] a = new int[]{1,2,3};

6. int[] a = new int[5]{1,2,3,4};

Try it and Tell me

```
public class ArrayDemo
{
 public static void main(String[] args)
  {
    int[] x;
    x = new int[5];
    x[0] = 11;
    x[4] = 22;
    System.out.println("Element at index 0: " + x[0]);
    System.out.println("Element at index 1: " + x[1]);
    System.out.println("Element at index 4: " + x[4]);
  }
}
```

Do It Yourself

Write a program to print all the arguments passed to a ***FindArguments.java*** *program.*

Input: *Number of arguments can be 0 to 12*

Output Format:

The Number of Arguments are :

The Following are the Arguments:

Example: *java FindArguments Hi Sai Kiran*

Output:

```
The Number of Arguments are : 3
The Following are the Arguments:
1 Hi
2 Sai
3 Kiran
```

ARRAY COPY

- To copy array elements from one array to another array, we can use ***arraycopy*** static method from System class.
- **Syntax**:

```
        public static void arraycopy(Object s, int  sIndex,
            Object d, int dIndex, int length)
```

- Ex:

```
  Int[] source = {1, 2, 3, 4, 5, 6};
  int[] dest      = new int[10];

  System.arraycopy(source,0, dest,0,source.length);
```

ARRAY COPY

```
public class ArrayLengthDemo
 {
    public static void main(String[] args)
     {
        int[] source = {100, 200, 300};
        int[]dest = new int[3];
        System.arraycopy(source, 0, dest, 0, source.length);
        for (int i =0;   i <dest.length;   i++)
          System.out.println("Element at index " + i + ": " +dest[i]);
    }
 }
```

Do It Yourself

Initialize an integer array with ascii values and print the corresponding character values in a single row.

Do It Yourself

Write a program to initialize an integer array with the command line arguments and print the sum and average of the array.

The number of Arguments is limited to maximum of 10.

Do It Yourself

Write a program to initialize an integer array and find the maximum and minimum value of an array

The number of Arguments is limited to maximum of 10.

Integer.MAX_VALUE

Integer.MIN_VALUE

Do It Yourself

Write a program to find the largest 2 numbers and the smallest 2 numbers in the array initialized from the command line arguments.

The number of Arguments is limited to maximum of 10.

Test Values: 1 2 3 4 5 6 7 8 9 10

o/p: Largest is 10 Second Largest is 9

Smallest is 1 Second Smallest is 2

Test Values: 12 1 2 3 4 5 6 7 8 9

Test Values: 1 2 3 4 5 6 7 8 9

Do It Yourself

Write a program to find the Maximum Sum by considering any 9 out of 10 values.

The number of Arguments is 10.

Test Values: 4 2 5 8 9 10 12 14 16 30

o/p: Maximum Sum of 9 numbers is 108

Do It Yourself

Write a program to initialize an integer array with values and check if a given number is present in the array or not. If the number is not found, it will print -1 else it will print the index value of the given number in the array

Ex1) Array elements are {1,4,34,56,7}

and the search element is 90

O/P: -1

Ex2)Array elements are {1,4,34,56,7}

and the search element is 56

O/P: 4

Do It Yourself

Write a program to initialize an array and print them in a sorted fashion.

The number of Arguments is limited to maximum of 10.

Do It Yourself

Write a program to print the sum of the elements of the array with the given condition. If the array has 6 and 7 in succeeding orders, ignore 6 and 7 and the numbers between them for the calculation of sum.

Eg1) Array Elements - 10,3,6,1,2,7,9

O/P: 22   [i.e 10+3+9]

Eg2) Array Elements - 7,1,2,3,6

O/P:19

Eg3) Array Elements - 1,6,4,7,9

O/P:10

Ex4: Array Elements – 1,2,3,4,6,5,4,3

o/p:28

Ex 5: 6 7 6 1 2 3 4 5 6 7

Ex 6: 1 6 2 7 1 6 3 4 7 6 6 7 7

Do It Yourself

Write a Java program to insert an element (specific position) into an array.

Initialize the array with the following values

Int []array=new int[10]{1,2,3,4,5,6,7,8,9}

Accept two command line arguments First one is value and second is position

Ex: java InsPos 10 4

Required: insert 10 at position 4

Output: 1,2,3,10,4,5,6,7,8,9

Do It Yourself

Write a program to remove the duplicate elements in an array and print

The number of Arguments is limited to maximum of 10.

Eg) Array Elements--12,34,12,45,67,89

O/P: 12,34,45,67,89

Do It Yourself

Write a program to print the element of an array that has occurred the highest number of times

The number of Arguments is limited to maximum of 10.

Eg) Array -> 10,20,10,30,40,100,99

O/P:10

Do It Yourself

Write a Java program to find the common elements between two arrays.

Create and initialize two arrays.

Example:

Array1 = {1,2,3,4,5,6,7,8,9}

Array2={4,10,12,5,23,6,15,19,8}

Output: Common Elements are : 4,5,6,8

TWO DIMENSIONAL ARRAY

Two-dimensional arrays are arrays of arrays

The 1st dimension represent rows or number of one dimension, the 2nd dimension represent columns or number of elements in the each one dimensions.

datatype[][] arrayname;

- Initializing two-dimensional arrays:

  int[][] y = new int[3][3];

TWO DIMENSIONAL ARRAY

- You can initialize the row dimension without initializing the columns but not vice versa

```
int[][] x = new int[3][];
int[][] x = new int[][3];   //error
```

- The length of the columns can vary for each row and initialize number of columns in each row

```
int [][]x = new int [2][];
        x[0] = new int[5];
        x[1] = new int [3];
```

TWO DIMENSIONAL ARRAY

- The curly braces { } may also be used to initialize two dimensional arrays

Ex:
```
int[][] y = { {1,2,3}, {4,5,6}, {7,8,9} };

int[][] y = new int[3][] { {1,2,3}, {4,5,6}, {7,8,9} };
```

Ex2:
```
int [][]x = new int [3][];

   x[0]   =  new int[]{0,1,2,3};
   x[1]   =  new int []{0,1,2};
   x[2]   =  new int[]{0,1,2,3,4};
```

CREATING, INITIALIZING, AND ACCESSING AN ARRAY

```
class TwoDimDemo {
  public static void main(String[] args) {
    int [][] x = new int[3][];
    x[0] = new int[3];
    x[1] = new int[2];
    x[2] = new int[5];
    for(int i=0;  i < x.length; i++) {
      for (int j=0; j < x[i].length; j++) {
        x[i][j] = i;  System.out.print(x[i][j]);
        }
        System.out.println();
        }
    }
}
```

Do It Yourself

Write a program to reverse the elements of a given 2*2 array. Four integer numbers needs to be passed as Command Line arguments.

Example1: C:\>java Sample 1 2 3

O/P Expected : Please enter 4 integer numbers

Example2: C:\>java Sample 1 2 3 4

O/P Expected :

```
The given array is :
1 2
3 4
The reverse of the array is :
4 3
2 1
```

Do It Yourself

Write a program to find greatest number in a 3*3 array. The program is supposed to receive 9 integer numbers as command line arguments.

Example1: C:\>java Sample 1 2 3

O/P Expected : Please enter 9 integer numbers

Example2:C:\>java Sample 1 23 45 55 121 222 56 77 89

O/P Expected :

```
The given array is :
1 23 45
55 121 222
56 77 89
The biggest number in the given array is 222
```

Inheritance

- Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

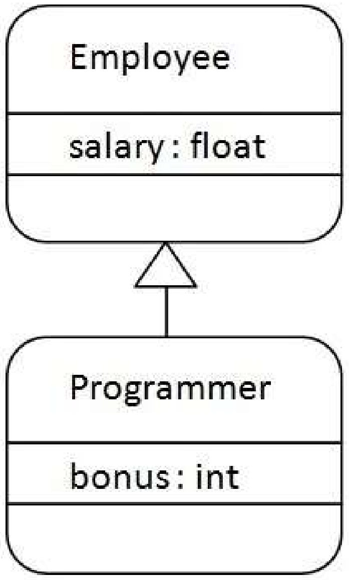
Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Syntax Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

```
class Employee
 {
 }

class Programmer extends Employee
 {
 }
```



Try it

```
class A{
  int m, n;
  void display1( ){
    System.out.println("m and n are:"+m+" "+n);
  }
}
class B extends A{
  int c;
  void display2( ){
    System.out.println("c :" + c);
  }
  void sum(){
    System.out.println("m+n+c = " + (m+n+c));
  }
}
```

Try it

```
class InheritanceDemo{
  public static void main(String args[ ])
  {
    B s2=new B();
    s2.m = 7;
    s2.n = 8;
    s2.c = 9;
    s2.display1();
    s2.display2();
    System.out.println("sum of m, n and c in object B  is:");
    s2.sum();
  }
}
```

Try it

TRY IT NOW!

```
class Employee
 {
   int Salary;
   Employee(){
    Salary=1000;}
 }
class Programmer extends Employee
 {
   int bonus=0;
   Programmer(int sal,int bon)
    {
         Salary = sal;
         bonus=bon;
    }
 }
```

Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

```
class Main
 {
   public static void main(String args[])
    {
         Programmer pg=new Programmer(2000,200);
         System.out.println(pg.Salary);
         System.out.println(pg.bonus);
    }
 }
```

Try it

```
class Main
 {
   public static void main(String args[])
     {
           Programmer pg=new Programmer(2000,200);
           System.out.println(pg.Salary);
           System.out.println(pg.bonus);
           Employee emp=new Employee();
           System.out.println(emp.Salary);
     }
 }
```

Try it

```
class Main
 {
   public static void main(String args[])
     {
          Programmer pg=new Programmer(2000,200);
          System.out.println(pg.Salary);
          System.out.println(pg.bonus);
          Employee emp=new Employee();
          System.out.println(emp.Salary);
          System.out.println(emp.bonus);
     }
 }
```

Private Members - Inheritance

A subclass includes all of the members of its superclass

But, it cannot directly access those members of the super class that have been declared as **private**.

```
class A{
 int money;
 private int pocketMoney;
 void fill (int money, int pocketMoney)
  {
    this.money = money;
    this.pocketMoney = pocketMoney;
  }
}
```

```
class B extends A{
  int total;
   void sum( ){
     total = money + pocketMoney;
   }
}
```

Try it

```
class A{
 int money;
  private int pocketMoney;
  void fill(int money, int pocketMoney)
  {
     this.money = money;
     this.pocketMoney = pocketMoney;
  }
  public int getPocketMoney(){
    return pocketMoney;
  }
}
```

```
class B extends A{
  int total;
   void sum( ){
      total = money + getPocketMoney();
   }
}

class C
 {
   public static void main(String args[])
    {
           B s1=new B();
           s1.fill(1000,500);
           s1.sum();
           System.out.println(s1.total);
    }
 }
```

Constructors - Inheritance

- The creation and initialization of the superclass object is a prerequisite to the creation of the subclass object.
- When a subclass object is created,
  - It creates the <u>superclass object</u>
  - Invokes the relevant superclass constructor.
    - The initialized superclass attributes are then inherited by the subclass object
  - finally followed by the creation of the <u>subclass object</u>
    - initialization of its own attributes through a relevant constructor subclass

Constructors - Inheritance

```
class A
 {
   int a;
   A()
    {
      a=10;
      System.out.println("in Class A Constructor");
    }
 }
class B extends A
 {
   int b;
   B()
    {
      b=20;
      System.out.println("in Class B Constructor");
    }
 }
```

```
class C
 {
   public static void main(String args[])
    {
          B s1=new B();
          System.out.println(s1.a +" " +s1.b);
    }
 }
```

Constructors - Inheritance

```
class A
 {
   int a;
   A()
    {
      a=10;
      System.out.println("in Class A Constructor");
    }
   A(int val)
    {
          a=val;
    }
 }
```

```
class B extends A
 {
   int b;
   B()
    {
      b=20;
      System.out.println("in Class B Constructor");
    }
   B(int val)
    {
       b=val;
       System.out.println("a is " +a +" b is " +b);

    }
 }
```

B b2=new B(50);

Super Keyword in Java

- The **super** keyword in java is a reference variable that is used to refer parent class objects.
- **Use of super with variables:** This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM.
- **Use of super with constructors:** super keyword can also be used to access the parent class constructor. One more important thing is that, ‘’super’ can call both parametric as well as non parametric constructors depending upon the situation.
- **Use of super with methods:** This is used when we want to call parent class method.

Super with Constructors - Inheritance

```
class A
 {
   int a;
   A()
    {
      a=10;
      System.out.println("in Class A Constructor");
    }
   A(int val)
    {
         a=val;
    }
 }
```

```
class B extends A
 {
   int b;
   B()
    {
      b=20;
      System.out.println("in Class B Constructor");
    }
   B(int val)
    {
      super(val*2);
      b=val;
      System.out.println("a is " +a +" b is " +b);

    }
 }
```

```
B b2=new B(50);
```

Super with Constructors - Inheritance

```
class A
 {
   A(int a)
    {
          System.out.println(a);
    }
 }
class B extends A
 {
   B(String str)
    {
          super(10);
          System.out.println(str);
    }
 }
```

```
class C
 {
   public static void main(String args[])
    {
          B obj=new B("Hello");
    }
 }
```

Super with Constructors - Inheritance

```
class A
 {
   A(int a)
    {
         System.out.println(a);
    }
  A(String str)
    {
         System.out.println(str);
    }
 }
```

```
class B extends A
 {
   B(String str)
    {
         super(10);
         System.out.println(str);
    }
 }
```

```
class C
 {
   public static void main(String args[])
    {
         B obj=new B("Hello");
    }
 }
```

Super with Variables - Inheritance

```
class A
 {
    int val=0;
 }
class B extends A
 {
    int val=0;
    void set(int val)
      {
        val=val+10;
        this.val=val+20;
        super.val=val*2;
        System.out.println("arg Val is"+val +" B val
          is "+this.val +" A val is "+super.val);

      }
 }
```

```
class C
 {
    public static void main(String args[])
      {
        B obj=new B();
        obj.set(20);
      }
 }
```

Super with Variables - Inheritance

```
class Vehicle
{
    int maxSpeed = 120;
}

/* sub class Car extending vehicle */
class Car extends Vehicle
{
    int maxSpeed = 180;

    void display()
    {
        /* print maxSpeed of base class (vehicle) */
        System.out.println("Maximum Speed: " + super.maxSpeed);
    }
}
```

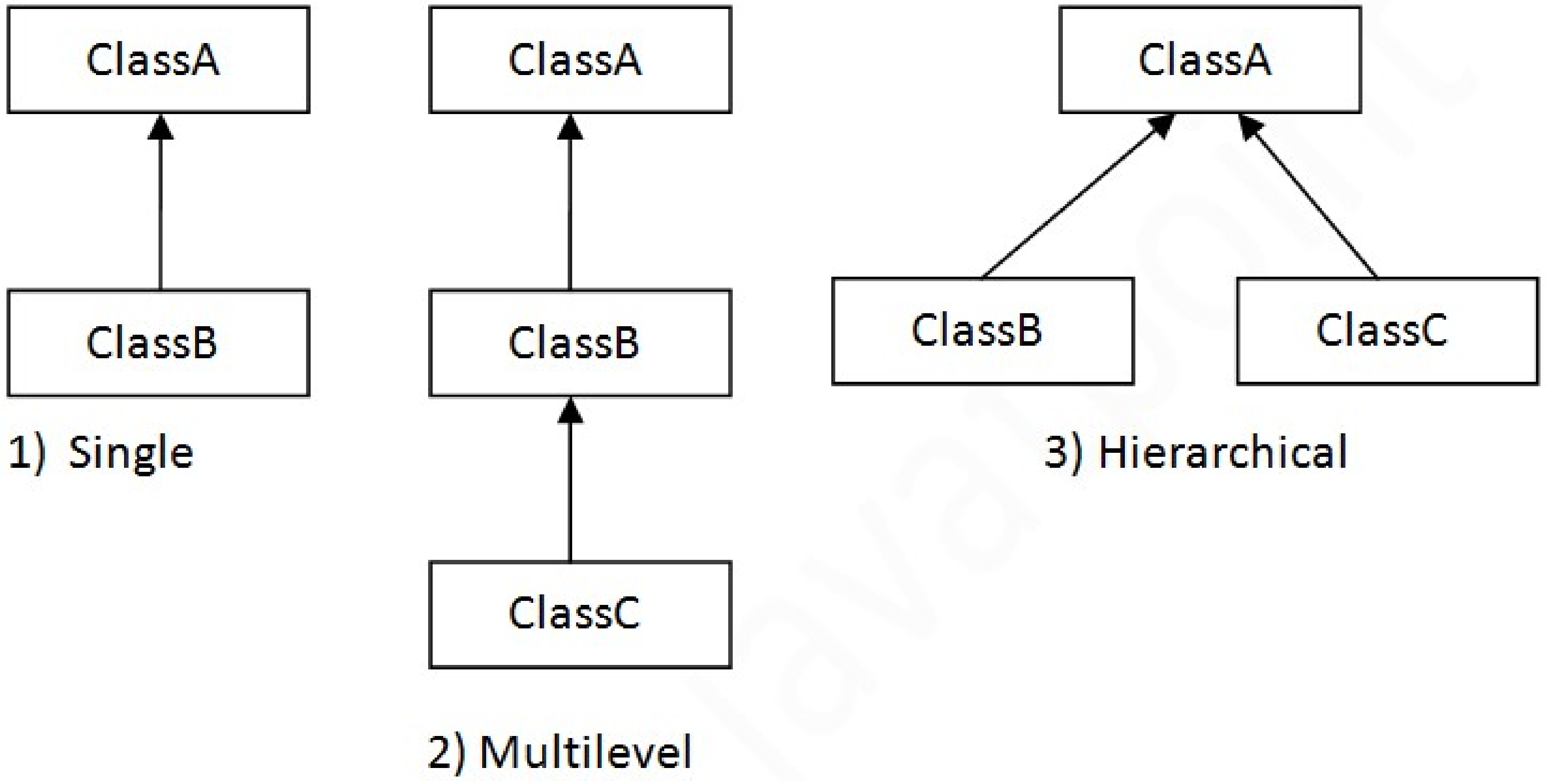
Super with Method- Inheritance

whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword.

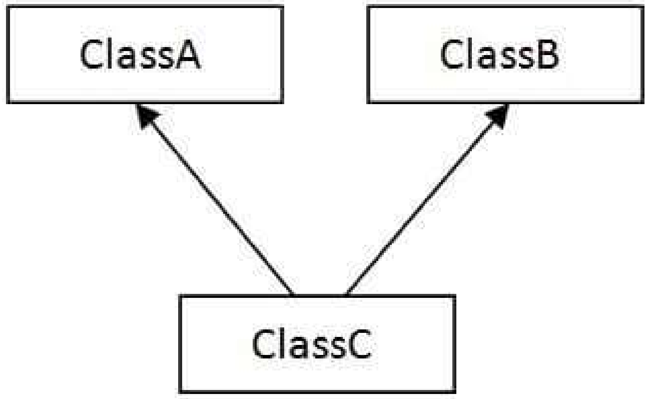
```
/* Base class Person */
class Person
{
    void message()
    {
        System.out.println("This is person class");
    }
}
```

```
/* Subclass Student */
class Student extends Person
{
    void message()
    {
        System.out.println("This is student class");
    }

void display()
    {
        message();
       super.message();
    }
}
```

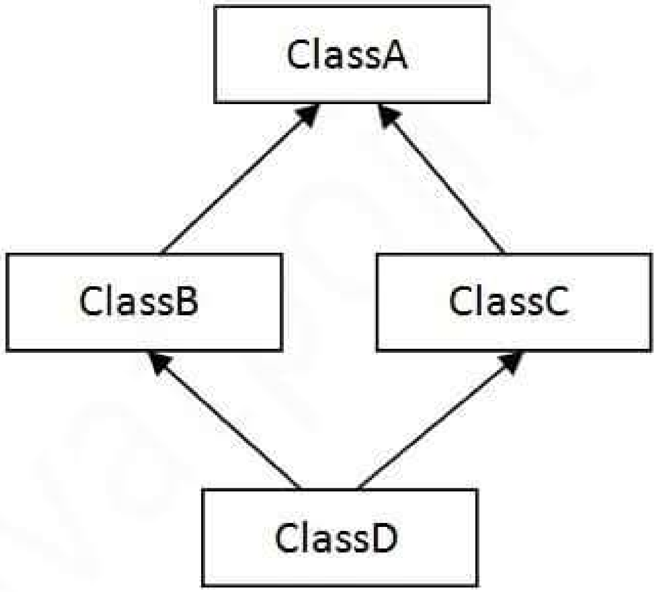
Types of Inheritance



Types of Inheritance



4) Multiple



5) Hybrid

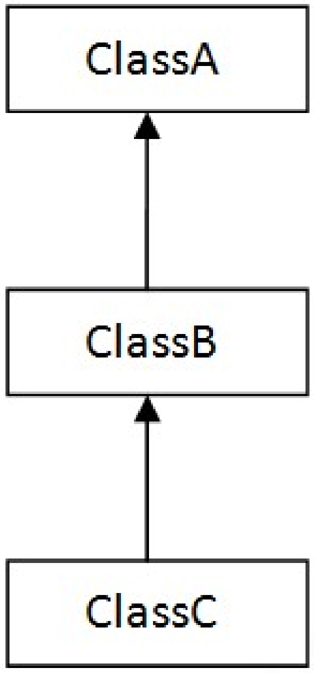
Single Inheritance

```
class Animal{
void eat()
 {
    System.out.println("eating...");
  }
}
class Dog extends Animal{
void bark()
  {
     System.out.println("barking...");
  }
}
```

```
class TestInheritance{
public static void main(String args[])
  {
    Dog d=new Dog();
    d.bark();
    d.eat();
  }
}
```

Multilevel Inheritance

- Java allows us to define multiple layers hierarchy
- We can define a superclass and a subclass, with the subclass in turn becoming a superclass for another subclass
- Consider the following example
  - Employee
  - Manager **is a** Employee
  - Director **is a** Manager



2) Multilevel

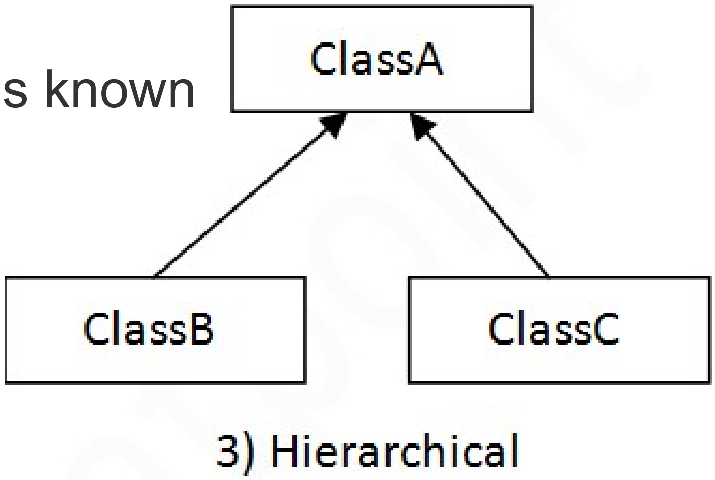
Multilevel Inheritance

```
class Animal{
void eat(){
System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){
System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){
System.out.println("weeping...");}
}
```

```
class TestInheritance2{
public static void main(String args[])
{
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}
}
```

Hierarchical Inheritance

- When two or more classes inherits a single class, it is known as *hierarchical inheritance*.

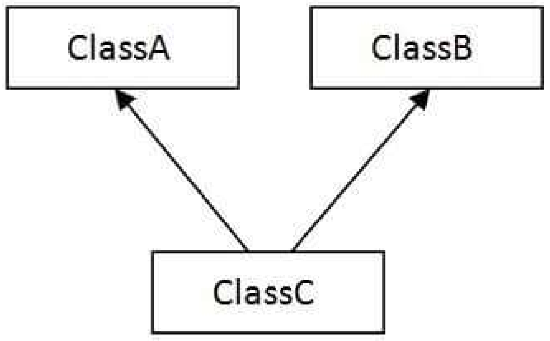


```
class Animal{
void eat(){
System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){
System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){
System.out.println("meowing...");}
}
```

```
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Multiple Inheritance

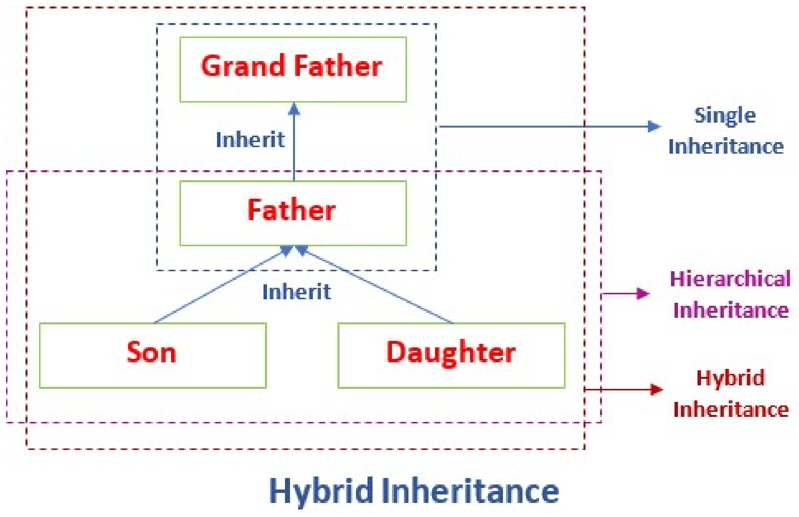
- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes.
- If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.



4) Multiple

Hybrid Inheritance

- Hybrid means consist of more than one. Hybrid inheritance is the combination of two or more types of inheritance.



DIY

Create a class named ‘Animal’ which includes methods like eat() and sleep().

Create a child class of Animal named ‘Bird’ and override the parent class methods. Add a new method named fly().

Create an instance of Animal class and invoke the eat and sleep methods using this object.

Create an instance of Bird class and invoke the eat, sleep and fly methods using this object.

DIY

Create a class called Person with a member variable name. Save it in a file called Person.java

Person

```
   private name
   public setName(String name);
   public String getName();
```

.

DIY

Create a class called Employee who will inherit the Person class.

The other data members of the employee class are annual salary (double), the year the employee started to work, and the national insurance number which is a String. Save this in a file called Employee.java

Employee extends Person

private double salary

private String doj

private int no

public addSalary(Salary)

public double getSalary()

public addDoj(DOJ)

public addNo(no)

DIY

A HighSchool application has two classes: the Person superclass and the Student subclass.

Using inheritance, you will create two new classes, Teacher and CollegeStudent.

A Teacher will be like Person but will have additional properties such as salary (the amount the teacher earns) and subject (e.g. “Computer Science”, “Chemistry”, “English”, “Other”).

The CollegeStudent class will extend the Student class by adding a year (current level in college) and major (e.g. “Electrical Engineering”, “Communications”, “Undeclared”).

Inheritance (IS-A) and Association (HAS-A) in Ja

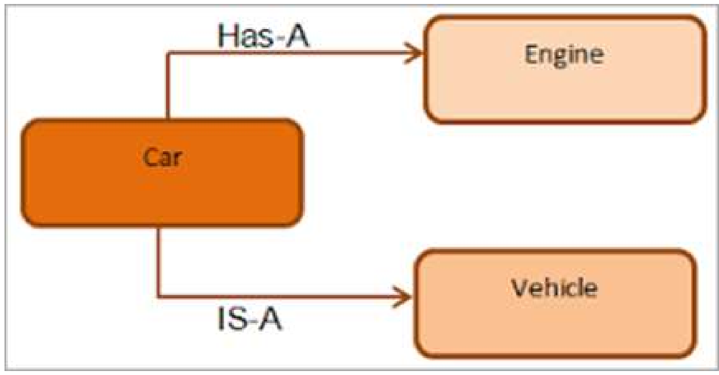
- In object-oriented programming, objects are related to each other and use the common functionality between them.

Inheritance (IS-A)

- An IS-A relationship signifies that one object is a type of another. It is implemented using ‘extends’ and ‘implements’ keywords.

Association (HAS-A)

- A HAS-A relationship signifies that a class has a relationship with another class.



Association

- Association is a relationship between two objects
- The association between objects could be
  - one-to-one
  - one-to-many
  - many-to-one
  - many-to-many
- Types of Association
  - Aggregation
  - Composition
- ***Example:*** A Student and a Faculty are having an association

Types of Association

- An **aggregation** container class and referenced class can have an independent existence.
- A **composition** reference class cannot exist if the container class is destroyed.
- A car has its parts e.g., engines, wheels, music player, etc.
- The **car cannot function without an engine** a**nd wheels** but can function without a music player.
- Here the engine and car have a composition relation, and the car and music player have an aggregation relationship.
- In the case of Aggregation, an object can exist without being part of the main object.

Aggregation

- Aggregation is a special case of association
- **`Its a directional association, which means it is strictly a one way association.`** Aggregation is also called a “Has-a” relationship.
- Consider two classes **Student** class and **Address** class.
  - Every student has an address so the relationship between student and address is a Has-A relationship.
  - But if you consider its vice versa then it would not make any sense as an Address doesn’t need to have a Student necessarily.

Aggregation

```
class Address
{
  int streetNum;
  String city;
  String state;
  String country;
  Address(int street, String c, String st, String coun)
  {
     this.streetNum=street;
     this.city =c;
     this.state = st;
     this.country = coun;
  }
}
```

```
class StudentClass
{
  int rollNum;
  String studentName;
  //Creating HAS-A relationship with Address class
  Address studentAddr;
  StudentClass(int roll, String name, Address addr){
     this.rollNum=roll;
     this.studentName=name;
     this.studentAddr = addr;
  }
```

Need for Aggregation

- To maintain code **re-usability**.
- Suppose there are two other classes College and Staff along with above two classes Student and Address.
- In order to maintain Student's address, College Address and Staff's address we don't need to use the same code again and again.
- Student Has-A Address (Has-a relationship between student and address)
- College Has-A Address (Has-a relationship between college and address)
- Staff Has-A Address (Has-a relationship between staff and address)

Polymorphism

- Polymorphism means "many forms“
- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*.
- **Polymorphism in Java** is the ability of an object to take many forms.
- The human body has different organs. Every organ has a different function to perform. So we have a standard method function that performs differently depending upon the organ of the body.

Types of Polymorphism

Polymorphism in Java can be performed by two different methods:

1. Method Overloading
2. Method Overriding

Method overloading is an example of **Static Polymorphism**.

Method Overriding is an example of **Dynamic/Runtime Polymorphism.**

Method Overloading

Method overloading is defined as a process that can create multiple methods of the same name in the same class, and all the methods work in different ways.

Method overloading occurs when there is more than one method of the same name in the class.

```
public void intSquare ( int number )
public void doubleSquare(double number)
public void longSquare(long number)
```

```
public void Square ( int number )
public void Square(double number)
public void Square(long number)
```

How to Overload

In java, we do method overloading in three ways: –

1. By changing the number of parameters.
2. By changing data types.
3. Sequence of Data type of parameters.

Method overloading **cannot** be done by changing the return type of methods.

Changing the data types:

```
class Sum
 {
    static int add(int a, int b)
     {
        return a+b;
     }
   static double add(double a, double b)
     {
        return a+b;
     }
  }
```

```
class TestOverloading2
 {
    public static void main(String[] args)
     {
        System.out.println(Sum.add(17,13));
        System.out.println(Sum.add(10.4,10.6));
     }
 }
```

Changing the sequence of data types:

```
class Divi
 {
   static double divide(int a, double b)
    {
       return (a/b);
    }
  static double divide(double a, int b)
    {
       return (a/b);
    }
 }
```

```
class TestOverloading2
 {
   public static void main(String[] args)
    {
      System.out.println(Divi.divide(17,2.0));
      System.out.println(Divi.divide(10.4,2));
    }
 }
```

Changing the Return types:

```
class Sample{
   int disp(int x){
      return x;
   }
   double disp(int y){
      return y;
   }
```

```
public static void main(String args[])
{
   Sample s = new Sample();
   System.out.printIn("Value of x : " + s.disp(5));
   System.out.printIn("Value of y : " + s.disp(6.5));
   }
}
```

Method Overloading

```
class CalculateSquare
 {
    public void square()    {
      System.out.println("No Parameter Method Called");
    }
   public int square( int number )  {
     int square = number * number;
    System.out.println("Method with Integer Argument
           Called:"+square);
    }
   public float square( float number ) {
      float square = number * number;
      System.out.println("Method with float Argument
           Called:"+square);
    }
```

```
public static void main(String[] args)
  {
    CalculateSquare obj = new CalculateSquare();
    obj.square();
    obj.square(5);
    obj.square(2.5);
  }
 }
```

Benefits of Overloading

- Method overloading increases the readability of the program.
- This provides flexibility to programmers so that they can call the same method for different types of data.
- This makes the code look clean.

Type Promotion

When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion,

for example: byte data type can be promoted to short, a short data type can be promoted to int, long, double etc.

```
class Demo{
  void disp(int a, double b){
        System.out.println("Method A");
  }
  void disp(double a, double b){
        System.out.println("Method B");
  }

  public static void main(String args[]){
        Demo obj=new Demo();
        obj.disp(100.0, 20);
        obj.disp(100, 20);
        obj.disp(100, 20.0);
  }
}
```

Is it Valid???

```
int mymethod(int a, int b, float c)
int mymethod(int var1, int var2, float var3)
```

```
int mymethod(int a, int b)
int mymethod(float var1, float var2)
```

```
int mymethod(int a, int b)
int mymethod(int num)
```

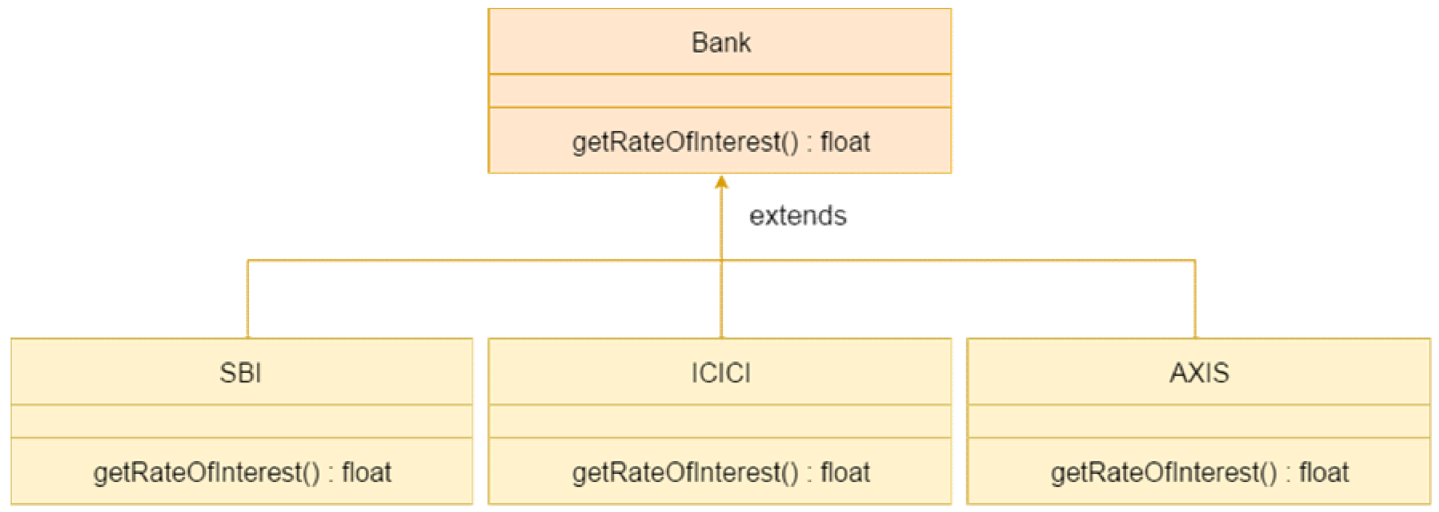
```
float mymethod(int a, float b)
float mymethod(float var1, int var2)
```

```
int mymethod(int a, int b)
float mymethod(int var1, int var2)
```

Method Overriding

- Declaring a method in sub class which is already present in parent class(Same Prototype) is known as method overriding.
- Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class.
- In this case the method in parent class is called overridden method and the method in child class is called overriding method.
- When an overridden method is called from an object of the subclass, it will always refer to the version defined by the subclass

Method Overriding


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

Method Overriding

```
class Shapes {
  public void area() {
    System.out.println("The formula for area of ");
  }
}
class Triangle extends Shapes {
  public void area() {
    System.out.println("Triangle is ½ * base * height ");
  }
}
```

```
class Circle extends Shapes {
  public void area() {
    System.out.println("Circle is 3.14 * radius * radius ");
  }
}
```

```
class Main {
  public static void main(String[] args) {
    Shapes myShape = new Shapes();
    Shapes myTriangle = new Triangle();
    Shapes myCircle = new Circle();
    myShape.area();
    myTriangle.area();
    myShape.area();
    myCircle.area();
  }
}
```

Advantage Method Overriding

- The main advantage of method overriding is that the class can give its own specific implementation to an inherited method **without even modifying the parent class code**.
- This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

upcasting

- A superclass reference variable can refer to a subclass object. This is also known as upcasting.
- Java uses this fact to resolve calls to overridden methods at run time.
- In dynamic method dispatch the object can call the overriding methods of child class and all the non-overridden methods of base class but it cannot call the methods which are newly declared in the child class.

upcasting

```
class A
{
    void m1()
    {
        System.out.println("Inside A's m1 method");
    }
}
class B extends A
{
  void m1()
    {
        System.out.println("Inside B's m1 method");
    }
   void disp()
    {
            System.out.println("Inside Disp Method");
    }
}
```

```
class Dispatch
{
    public static void main(String args[])
    {
        A a = new A();
        B b = new B();
        A ref;
        ref = a;
        ref.m1();
        ref = b;
        ref.m1();

     }
}
```

upcasting

```
class A
{
    void m1()
    {
        System.out.println("Inside A's m1 method");
    }
}
class B extends A
{
  void m1()
    {
        System.out.println("Inside B's m1 method");
    }
   void disp()
    {
            System.out.println("Inside Disp Method");
    }
}
```

```
class Dispatch
{
    public static void main(String args[])
    {
        A a = new A();
        B b = new B();
        A ref;
        ref = a;
        ref.m1();
        ref = b;
        ref.m1();
        ref.disp();
    }
}
```

Method Overriding – Dynamic Method Dispatch

- **Dynamic method dispatch** is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

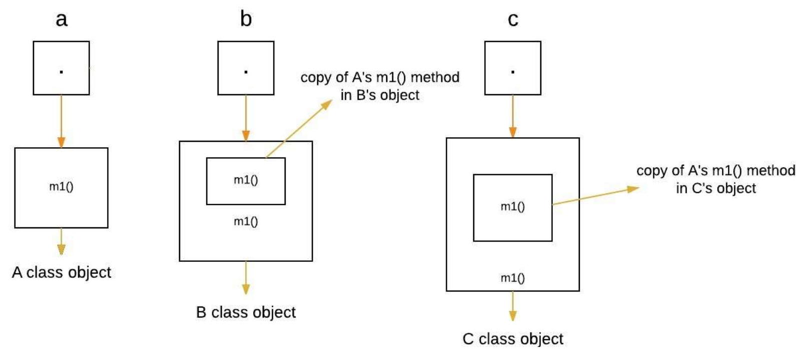
Method Overriding – Dynamic Method Dispatch

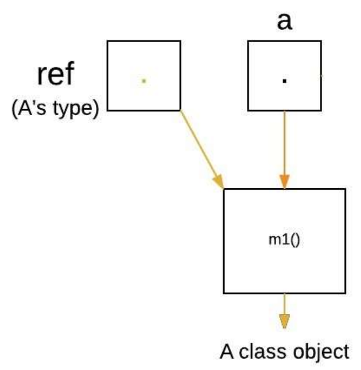
```
class A
{
    void m1()
    {
        System.out.println("Inside A's m1 method");
    }
}

class B extends A
{
   void m1()
    {
        System.out.println("Inside B's m1 method");
    }
}
```

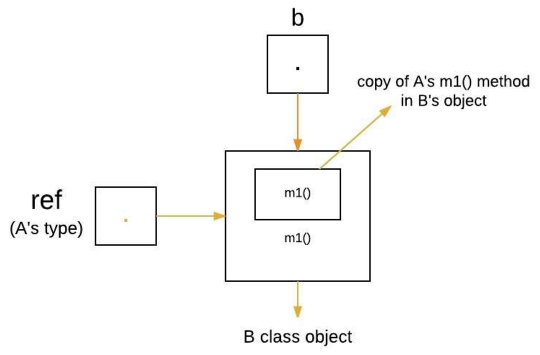
```
class C extends B
{
void m1()
    {
        System.out.println("Inside C's m1 method");
    }
}
```

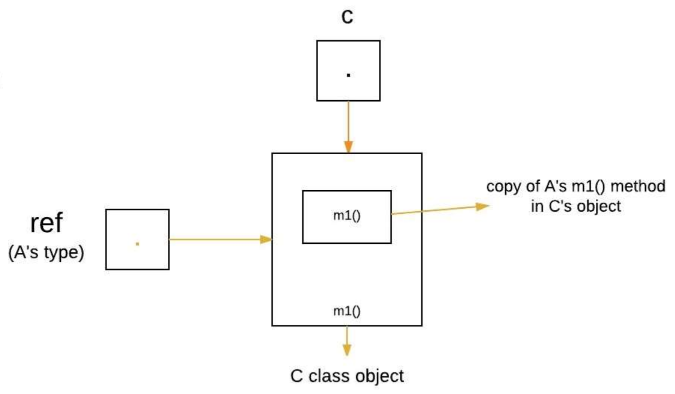
Method Overriding – Dynamic Method Dispatch

```
class Dispatch
{
   public static void main(String args[])
   {
      A a = new A();
      B b = new B();
      C c = new C();
      A ref;
      ref = a;
      ref.m1();
      ref = b;
      ref.m1();
      ref=c;
      ref.m1();
      }
 }
```


a
b
c
m1()
A class object
m1()
m1()
copy of A's m1() method in B's object
B class object
m1()
m1()
copy of A's m1() method in C's object
C class object







Is it Valid???

Can we override static method?

No, a static method cannot be overridden. It is because the static method is bound with class whereas instance method is bound with an object.

Difference

Method Overloading	Method Overriding
Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
Method overloading is the example of *compile time polymorphism.*	Method overriding is the example of *run time polymorphism.*
In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

DIY

Create a base class Fruit which has name ,taste and size as its attributes.

A method called eat() is created which describes the name of the fruit and its taste.

Inherit the same in 2 other class Apple and Orange and override the eat() method to represent each fruit taste.

DIY

Write a program to create a class named shape. It should contain 2 methods- draw() and erase() which should print “Drawing Shape” and “Erasing Shape” respectively.

For this class we have three sub classes- Circle, Triangle and Square and each class override the parent class functions- draw () and erase ().

The draw() method should print “Drawing Circle”, “Drawing Triangle”, “Drawing Square” respectively.

The erase() method should print “Erasing Circle”, “Erasing Triangle”, “Erasing Square” respectively.

Create objects of Circle, Triangle and Square in the following wav and observe the polymorphic nature of the class by calling draw() and erase() m

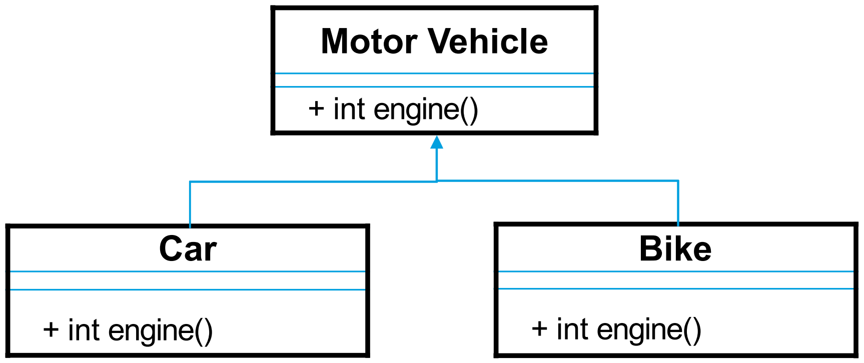
Shape c=new Circle();

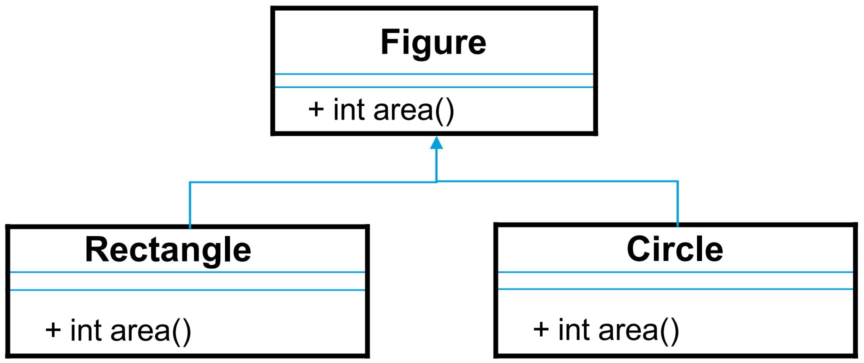
Shape t=new Triangle();

Shape s=new Square();

Abstract Classes

- **Abstract classes** define a generalized form **that will be shared by all of its subclasses**, so that **each subclass can provide specific implementations** of such methods.





- In the above example area() for Figure being more generic we cannot define it. At the level of rectangle or Circle we can give the formula for area.

Abstract Classes

- Often, you would want to define a superclass that declares the structure of a given abstraction without providing the implementation of every method
- The objective is to:
  - Create a superclass that only defines a generalized form that will be shared by all of its subclasses
  - Leaving it to each subclass to provide for its own specific implementations
  - Such a class determines the nature of the methods that the subclasses ***must implement***
  - Such a superclass is unable to create a meaningful implementation for a method or methods

Abstract Classes

- The class **Figure** in the previous example is such a superclass.
  - Figure is a pure geometrical abstraction
  - You have only kinds of figures like **Rectangle**, **Triangle** etc. which actually, are subclasses of class **Figure**
  - The class **Figure** has no implementation for the **area( )** method, as there is no way to determine the area of a **Figure**
  - The **Figure** class is therefore a partially defined class with no implementation for the **area( )** method
    The definition of **area()** is simply a placeholder

Abstract Method

- **abstract method** – It’s a method declaration with no definition
- a mechanism which shall ensure that a subclass must compulsorily override such methods.
- Abstract method in a superclass has to be overridden by all its subclasses.
- The subclasses cannot make use of the abstract method that they inherit directly(without overriding these methods).
- These methods are sometimes referred to as subclasses responsibility as they have no implementation’ specified in the superclass.

Abstract Classes

- To use an abstract method, use this general form:

  abstract type name(parameter-list);

- Abstract methods do not have a body
- Abstract methods are therefore characterized by the lack of the opening and closing braces that is customary for any other normal method
- This is a crucial benchmark for identifying an abstract class area method of Figure class made Abstract.

  ```
  public abstract int area();
  ```

Abstract Classes

- Any class that contains one or more abstract methods **must** also be declared abstract
  - It is perfectly acceptable for an abstract to implement a concrete method.
  - You cannot create objects of an abstract class
  - That is, an abstract class cannot be instantiated with the new keyword
  - Any subclass of an abstract class must **either implement all of the abstract methods in the superclass or be itself declared abstract.**

Revised Figure Class – using abstract

- There is no meaningful concept of area() for an undefined two-dimensional geometrical abstraction such as a Figure
- The following version of the program declares area( ) as abstract inside class Figure.
- This implies that class Figure be declared abstract, and all subclasses derived from class Figure must override area( ).

Improved Version

```
abstract class Figure{
   double dimension1;
   double dimension2;
  Figure(double x, double y)
  {
    dimension1 =  x;
    dimension2 = y;
  }
  abstract double area();
}
```

Improved Version

```
class Rectangle extends Figure{
  Rectangle(double x, double y)
    {
       super(x,y);
     }

  double area(){
    System.out.print("Area of rectangle is :");
    return dimension1 * dimension2;
  }
}
```

Improved Version

```
class Triangle extends Figure{
  Triangle(double x, double y){
   super(x,y); }
  double area(){
    System.out.print("Area for triangle is :");
    return dimension1 * dimension2 / 2;
   }
  }
```

Improved version

```
class FindArea{
 public static void main(String args[]){
   Figure fig;
   Rectangle r = new Rectangle(9,5);
   Triangle t  = new Triangle(10,8);
   fig = r;
   System.out.println("Area  of  rectangle  is  :"  +
 fig.area());
   fig = t;
   System.out.println("Area  of  triangle  is  :"  +
 fig.area());
 }
}
```

Quiz

What will be the output for the below code ?

```
class Gbase{
public abstract void testBase();
}
public class Sample extends GBase{
   public static void main() {
      Sample ob = new Sample();
      ob.testBase();
   }
}
```

Quiz

What will be the output for the below code ?

```
class abstract GBase{
public void testBase(){
System.out.println("Hello World");
}
}
public class Sample extends GBase{
   public static void main() {
      GBase ob = new GBase();
      ob.testBase();
   }
}
```

DIY

1.1. Create a class called **GeneralBank** which acts as base class for all banks. This class has functionality **getSavingInterestRate** and **getFixedInterestRate** methods, which return the saving a/c rate of interest and fixed account rate of interest the specific bank gives. Since GeneralBank cannot say what percentage which bank would give, make it abstract.

1.2. Create 2 subclasses of GeneralBank called **ICICIBank** and **KotMBank**. Override the methods from base class. ICICI - Savings 4% Fixed 8.5% and KotMBank. - Savings 6% Fixed 9%

1.3. Create a main method to test the above classes. Try one by one and absorb your finding.

a) ICICIBank object reference instantiated with ICICIBank class.

b) KotMBank object reference instantiated with KotMBank class.

c) GeneralBank object reference instantiated with KotMBank class.

d) GeneralBank object reference instantiated with ICICIBank class.

DIY

Create an abstract class **Compartment** to represent a rail coach. Provide an abstract function **notice** in this class.

Derive **FirstClass**, **Ladies**, **General**, **Luggage** classes from the compartment class.

Override the **notice** function in each of them to print notice suitable to the type of the compartment.

Create a class **TestCompartment**. Write main function to do the following:

Declare an **array** of Compartment of size 10.

Create a compartment of a type as decided by a randomly generated integer in the range 1 to 4.

Check the polymorphic behavior of the notice method.

DIY

Create an abstract class **Instrument** which is having the abstract function **play**.

Create three more sub classes from Instrument which is **Piano**, **Flute**, **Guitar**.

Override the play method inside all three classes printing a message

“Piano is playing tan tan tan tan ” for Piano class

“Flute is playing toot toot toot toot” for Flute class

“Guitar is playing tin tin tin ” for Guitar class

You must not allow the user to declare an **object of Instrument class**.

Create an array of **10 Instruments**.

Assign different type of instrument to Instrument reference.

Check for the **polymorphic behavior** of play method.

Use the **instanceof** operator to print that which object stored at which index of instrument array.

Final Keyword

- The **final** keyword used in context of behavioral restriction on:
  - variables
  - methods
  - classes
- Using final on variables to make them behave as constants.
- When a variable is made final – it can be initialized only once either by
  - Declaration and initialization

    final int x=10;
  - Using constructor
- System allows you to set the value only once; after which it can’t be changed.

Quiz

What will be the output for the below code ?

```
public class Sample {
       final double pi;
       public Sample()
   {
       pi = 3.14;
   }
   public Sample(double pi)
   {
       this.pi = pi;
   }
```

```
public static void main() {
    Sample ob = new Sample(22/7);
    System.out.println(ob.pi);
    }

}
```

Final Keyword

- The **final** keyword used in context of behavioral restriction on:
  - variables
  - methods
  - classes
- Using final on variables to make them behave as constants which we have seen in earlier module.
- When a variable is made final – it can be initialized only once either by
  - Declaration and initialization

    final int x=10;
  - Using constructor
- System allows you to set the value only once; after which it can't be changed.

Final Keyword

```
class GBase {
public final void display(String s)
{
   System.out.println(s);
}  }
class Sample extends GBase{
     public void display(String s)
   {
      System.out.println(s);
   }
   public static void main(String args[]) {
   Sample ob = new Sample();
   ob.display("TRY ME");
}  }
```

Output:

Compile Time Error : **Cannot override the final method from GBase**

Final Keyword

- Using final to Prevent Inheritance
  - Sometimes you will want to prevent a class from being inherited.
  - This can be achieved by preceding the class declaration with final.
  - Declaring a class as final implicitly declares all of its methods as final too.
  - It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide concrete and complete implementations.

Final Keyword

```
final class GBase {
public void display(String s)
{
  System.out.println(s);
} }
class Sample extends GBase{ public void display(String s)
{
  System.out.println(s);
}
  public static void main(String args[]) {
  Sample ob = new Sample();  ob.display(“TRY ME”);
} }
```

Output:

```
Compile Time Error : The type Sample
cannot subclass the final class
GBase
```

Quiz

What will be the output for the below code ?

```
class abstract GBase{
public final void testBase(){
System.out.println(“Hello World”);
}
}
```

Introduction to Interfaces

Interface

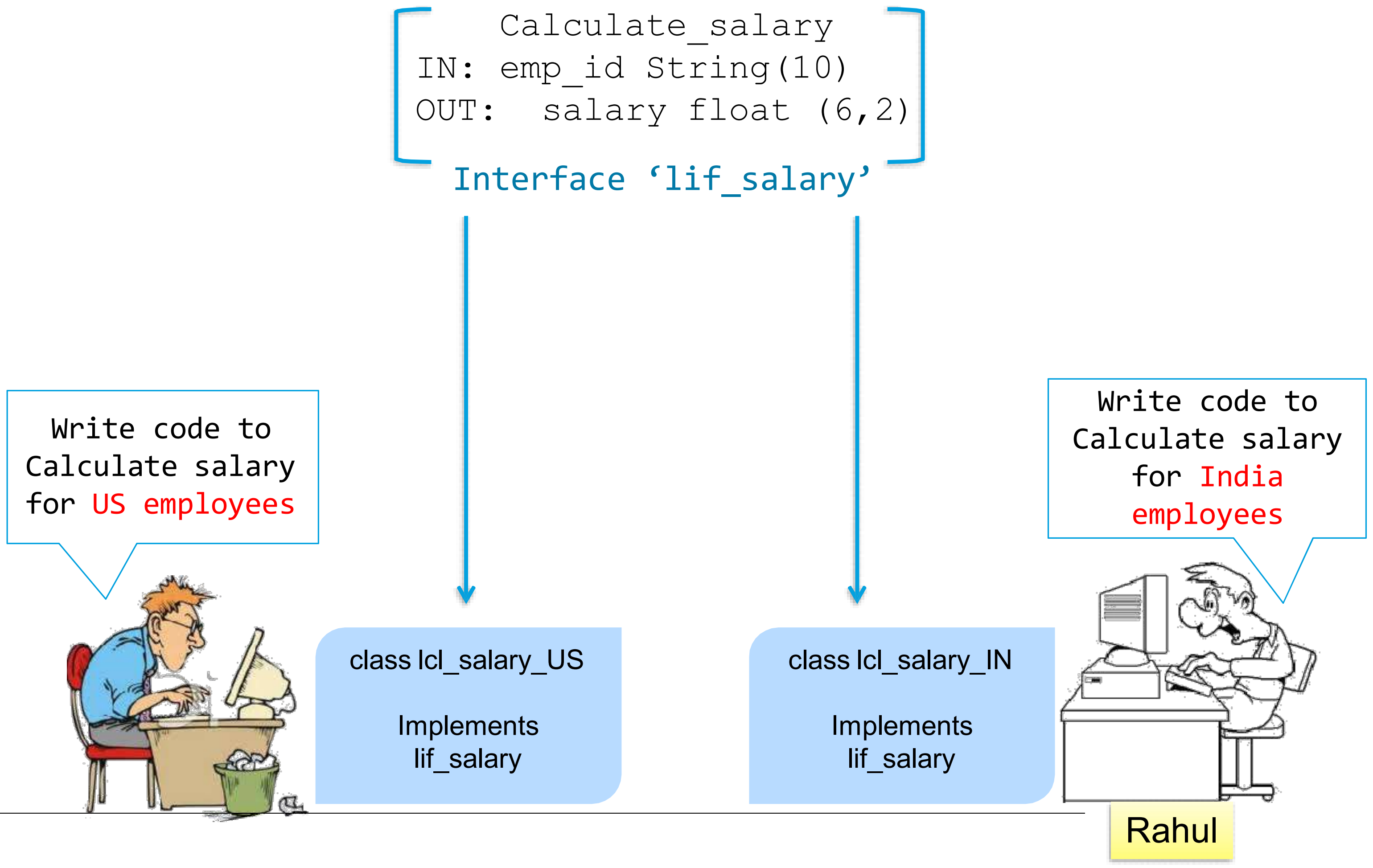
An interface is a named collection of method declarations (without implementations)

- An interface can also include constant declarations
- An interface is syntactically similar to an abstract class
- An interface is a collection of abstract methods and final variables
- A class implements an interface using the **implements** clause

Interface

- An interface defines a protocol of behavior
- An interface lays the specification of what a class is supposed to do
- How the behavior is implemented is the responsibility of each implementing class
- Any class that implements an interface adheres to the protocol defined by the interface, and in the process, implements the specification laid down by the interface

Example



```
   Calculate_salary
IN: emp_id String(10)
OUT:  salary float (6,2)
```

Interface 'lif_salary'
Write code to Calculate salary for US employees
Write code to Calculate salary for India employees
class lcl_salary_US Implements lif_salary
class lcl_salary_IN Implements lif_salary
Rahul

Need for Interface

- Interfaces allow you to implement common behaviors in different classes that are not related to each other
- Interfaces are used to describe behaviors that are not specific to any particular kind of object, but common to several kind of objects

Need for Interface

- Defining an interface has the advantage that an interface definition stands apart from any class or class hierarchy
- This makes it possible for any number of independent classes to implement the interface
- Thus, an interface is a means of specifying a consistent specification, the implementation of which can be different across many independent and unrelated classes to suit the respective needs of such classes
- Interfaces reduce coupling between components in your software

Need for Interface

- Java does not support multiple inheritance
- This is a constraint in class design, as a class cannot achieve the functionality of two or more classes at a time
- Interfaces help us make up for this loss as a class can implement more than one interface at a time
- Thus, interfaces enable you to create richer classes and at the same time the classes need not be related

Interface members

- All the methods that are declared within an interface are always, by default, **public** and **abstract**
- Any variable declared within an interface is always, by default, **public static** and **final**

Quiz

What is the behavior which is common among the entities depicted in the pictures above?

Yes..You are right. All of them can fly.

Requirement : You have to develop 3 classes, Bird, Superman and Aircraft with the condition that all these classes must have a method called fly().

What is the mechanism, using which you can ensure that the method fly() is implemented in all these classes?

An Abstract class or An Interface?

Defining Interface

- An interface is syntactically similar to a class
- It’s general form is:

```
public interface FirstInterface
{
        int addMethod(int x, int y);
        float divMethod(int m, int n);
        void display();
        int VAR1 = 10;
        float VAR2 = 20.65;
}
```

Implementing Interfaces

- A class can implement more than one interface by giving a comma- separated list of interfaces

```
class MyClass implements FirstInterface{
   public int addMethod(int a, int b){
      return(a+b);
   }
   public float divMethod(int i, int j){
      return(i/j);
   }
   public void display(){
    System.out.println(“Variable 1 :” +VAR1);
    System.out.println(“Variable 2 :” +VAR2);
   }
}
```

Quiz

Will the following code compile successfully ?

```
interface I1 {
     private int a=100;
     protected void m1();
}

class A1 implements I1 { public void m1()
     {
     System.out.println("In m1 method");
     }
}
```

It will throw compilation errors.. Why?

Quiz

Will the following code compile successfully ?

```
interface I1 {
     static int a=100; static void m1();
}

class A1 implements I1 { public void m1() {
     System.out.println("In m1 method");
     }
}
```

It will throw compilation error.. Why?

Applying Interfaces

- Software development is a process where constant changes are likely to happen
- There can be changes in requirement, changes in design, changes in implementation
- Interfaces support change
- Programming through interfaces helps create software solutions that are reusable, extensible, and maintainable

Applying Interfaces

```
interface IntDemo{
void display();
}
class classOne implements IntDemo{
 void add(int x, int y){
  System.out.println("The sum is :" +(x+y));
 }
 public void display(){
  System.out.println("Welcome to Interfaces");
 }
}
```

Applying Interfaces

```
class classTwo implements IntDemo{
void multiply(int i,int j, int k) {
    System.out.println("The result:" +(i*j*k) );
  }
  public void display(){
  System.out.println("Welcome to Java ");
  }
}
class DemoClass{
  public static void main(String args[]) { classOne
    c1= new classOne();
   c1.add(10,20);
    c1.display();
    classTwo c2 = new classTwo();
    c2.multiply(5,10,15);
    c2.display();
  }
}
```

Interface References

- When you create objects, you refer them through the class references. For example :
  - ClassOne c1= new classOne();

  /*Here, c1 refers to the object of the class classOne. */
- You can also make the interface variable refer to the objects of the class that implements the interface
- The exact method will be invoked at run time
- It helps us achieve run-time polymorphism

Interface References

```
class DemoClass{
 public static void main(String args[]){
   IntDemo c1= new classOne();
   c1.display();
   c1 = new classTwo();
   c1.display();
  }
}
```

Extending Interfaces

- Just as classes can be inherited, interfaces can also be inherited
- One interface can extend one or more interfaces using the keyword **extends**
- When you implement an interface that extends another interface, you should provide implementation for all the methods declared within the interface hierarchy

Extending Interfaces

```
interface I1 {
     int a=100;
     void m1();
}
interface A1 extends I1 {
public void m2();
}

class Aimp implements I1 {
public void m1() {
     System.out.println("In m1 method");
     }
       }
```

Abstract Classes v/s Interfaces

Abstract Classes

- Abstract classes can have non-final non-static variables.
- Abstract Classes can have abstract methods as well as concrete methods.
- You can declare any member of an abstract class as private, default, protected or public. Members can also be static.
- Abstract class is extended by another class using "extends" keyword.

Interfaces

- Variables declared within an interface are always static and final.
- Interfaces can have only method declarations(abstract methods). You cannot define a concrete method.
- Interface members are by default public. You cannot have private or protected members. Interface methods cannot be static.
- An interface is "implemented" by a java class using "implements" keyword .

Abstract Classes v/s Interfaces

Abstract Classes

- An abstract class can extend another class and it can implement one or more interfaces.
- An abstract class can have constructors defined within it.
- An abstract class cannot be instantiated using “new” Keyword
- You can execute(invoke) an abstract class, provided it has public static void main(String[] args) method declared within it.

Interfaces

- An interface can extend one or more interfaces but cannot extend a class. It cannot implement an interface.
- You cannot define constructors within an interface.
- An interface cannot be instantiated.
- You cannot execute an interface