Marwadi
education foundation

Unit – 5

Object Oriented Thinking

Prepared By

Prof. Ravikumar Natarajan

Assistant Professor, CE Dept.

KNOWLEDGE IS THE CURRENCY
FOR THE 21st CENTURY

Elements / Features / Concepts / Principles of OOPs

1. **Object** – Instance of class | - Run time entities which occupies memory
2. **Classes** – Collection of attributes, methods.
3. **Instance** – Obj. created at run time
4. **Inheritance** – Provides reusability |
5. **Data abstraction** – Information hiding | refers to particular feature and hiding its background details | used in software design phase.
6. **Encapsulation** – Binding data and method together | used in s/w implementation | Inherited
7. **Polymorphism** –Ability to take more than one form | Types: compile time & run time
8. **Message passing** – An object sends data to another object.

Characteristics of Java

1. **Simple** – No pointer concept so easy to debug – auto memory allocation & De-allocation
2. **Portable** – JVM | run in any platform
3. **Object Oriented** – Almost everything in java is object – it is an entity has attributes, functions to manipulate.
4. **Platform independent** – WORA | write once, Run anywhere - JVM
5. **Dynamic and distributed** – java classes can be distributed in networks – java.net package.
6. **Multithreaded** – concurrency (run multiple pgm at same time) – Parallel execution – built in thread class.
7. **Robust** and **secure** – good exception handling, explicit methods – array bound checking
8. **Interpreted** language –source code stored in .java – compiled file in .class(bytecode) – JVM interprets and executes the program.
9. **High performance** – Byte codes are highly optimized | JVM executes it faster
10. **Architecture-neural** – Independent of hardware

Abstraction

- Class abstraction is the separation of class implementation from the use of a class.
- The details of implementation are encapsulated and hidden from the user, which is called as class encapsulation.

- **Ways to achieve Abstraction**
- There are two ways to achieve abstraction in java

- Abstract class (0 to 100%)
- Interface (100%)

Abstraction

- **Encapsulation vs Data Abstraction**

- Encapsulation is data hiding(information hiding) while Abstraction is detailed hiding(implementation hiding).

- While encapsulation groups together data and methods that act upon the data, data abstraction deal with exposing the interface to the user and hiding the details of implementation.

Advantages of Abstraction

- It reduces the complexity of viewing the things.
- Avoids code duplication and increases reusability.
- Helps to increase the security of an application or program as only important details are provided to the user.

Class abstraction and Encapsulation

- Class abstraction is the separation of class implementation from the use of a class.
- The details of implementation are encapsulated and hidden from the user, which is called as class encapsulation.

Class abstraction and Encapsulation

- Java provides many levels of abstraction.
- Class abstraction separates class implementation from how the class is used.
- The creator of a class describes the functions of the class and lets the user know how the class can be used.
- The collection of methods and fields that are accessible from outside the class, together with the description of how these members are expected to behave, serves as the class's contract.

Class abstraction and Encapsulation

- A class is also known as an abstract data type (ADT).

- As example, consider getting a loan.
- A specific loan can be viewed as an object of a Loan class.
- The interest rate, loan amount, and loan period are its data
- properties, and computing the monthly payment and total payment are its methods.
- As a user of the Loan class, you don’t need to know how these methods are implemented.

Abstract Class

Hiding the internal implementation of the feature and only showing the functionality to the users. i.e. what it works (showing), how it works (hiding). Both abstract class and interface are used for abstraction.

Rules:

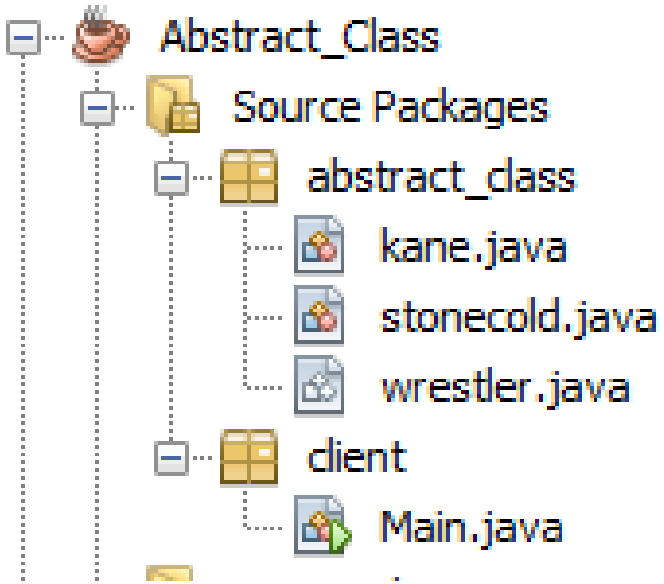
1. Abstract method must present in abstract class only.
2. Cannot be instantiated i.e not allowed to create object.
3. Method must be overridden.
4. It can have constructors & final and static methods

Abstract Class

Example for abstract class and method:

```
abstract class Animal                    //abstract parent class
{
   public abstract void sound();//abstract method
// public abstract void disp();//if this method is not overridden then class error
occurs
}
public class Dog extends Animal{
            public void sound()
            {
                        System.out.println("Woof");
            }
            public static void main(String args[])
            {
                        Animal obj = new Dog();
                        obj.sound();
            }}
```

Abstract Class – Case Study



Abstract Class – Case Study

Marwadi education foundation

wrestler.java

```
1  package abstract_class;
2  public abstract class wrestler {
3      public void paymentForWork(int hours)
4      {
5          System.out.println("Amount" + hours*250);
6      }
7      public abstract void themeMusic();
8      public abstract void finisher();
9  }
```

kane.java

```
1  package abstract_class;
2  public class kane extends wrestler {
3      public void themeMusic(){
4        System.out.println("Kane intro")  ;
5      }
6      public void finisher(){
7        System.out.println("Tombstone")  ;
8      }
9  }
```

stonecold.java

```
1  package abstract_class;
2  public class stonecold extends wrestler{
3      public void themeMusic(){
4        System.out.println("Stonecold intro")  ;
5      }
6      public void finisher(){
7        System.out.println("stonecold stunner")  ;
8      }
9  }
```

Abstract Class – Case Study

Main.java

Source | History

```
1   package client;
2   import abstract_class.*;
3   public class Main {
4       public static void main(String args[])
5       {
6           wrestler w1 = new kane();
7           w1.themeMusic();
8           w1.finisher();
9           w1.paymentForWork(2);
10
11          wrestler w2 = new stonecold();
12          w2.themeMusic();
13          w2.finisher();
14          w2.paymentForWork(3);
15      }
16  }
```

Encapsulation

- Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.

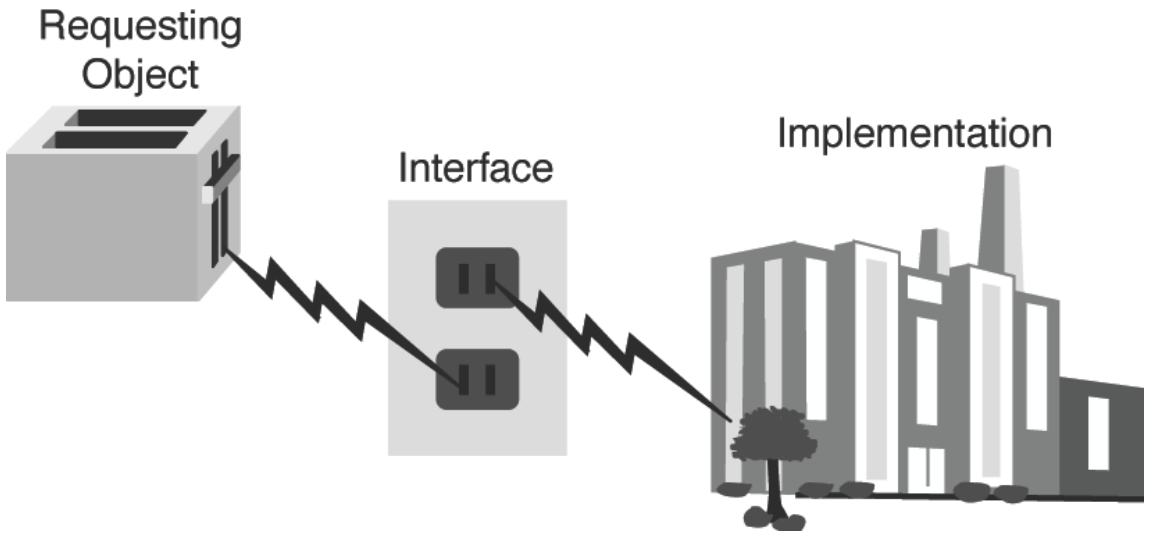
- In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.

Encapsulation


```
class Encapsulate {
private String Name;
public String getName()
{ return Name; }
public void setName(String newName)
   {
          Name = newName;
   }

public class Main{
   public static void main(String[] args)
   {
      Encapsulate obj = new Encapsulate();
          obj.setName("Harsh");
          System.out.println("name: " + obj.getName());
          //System.out.println("name: " + obj.Name);
}}
```

Thinking in Objects

- The procedural paradigm focuses on designing methods. The object-oriented paradigm couples data and methods together into objects.
- Software design using the object-oriented paradigm focuses on objects and operations on objects.
- Classes provide more flexibility and modularity for building
- reusable software



Power plant example

Class Relationships

- To design classes, you need to explore the relationships among classes.
- The common relationships among classes are
  - Association
  - Aggregation
  - Composition
  - Inheritance

Class Relationship in Java

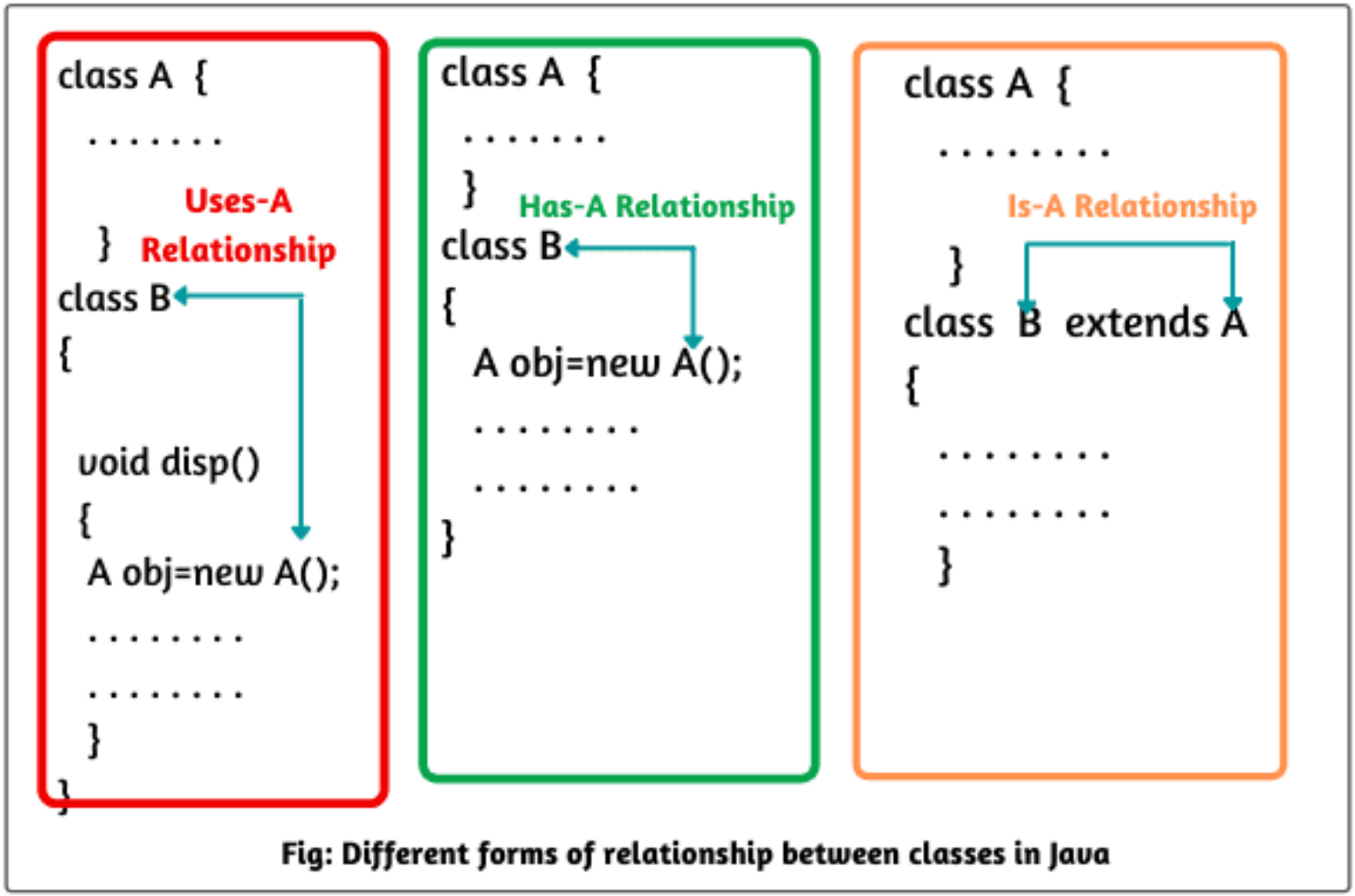


Fig: Different forms of relationship between classes in Java

Class Relationship in Java

Association is a general binary relationship that describes an activity between two classes.

For example, a student taking a course is an association between the Student class and the Course class, and a faculty member teaching a course is an association between the Faculty class and the Course class.

```
public class Student {
  private Course[]
    courseList;

  public void addCourse(
    Course s) { ... }
}
```

```
public class Course {
  private Student[]
    classList;
  private Faculty faculty;

  public void addStudent(
    Student s) { ... }

  public void setFaculty(
    Faculty faculty) { ... }
}
```

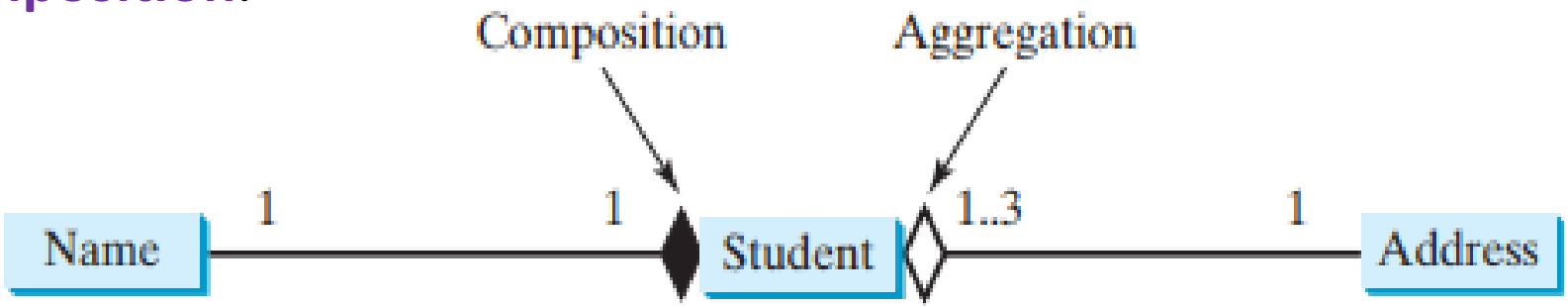
```
public class Faculty {
  private Course[]
    courseList;

  public void addCourse(
    Course c) { ... }
}
```

Class Relationship in Java

Marwadi education foundation

Aggregation and Composition

- **Aggregation** is a special form of association that represents an ownership relationship between two objects. **Aggregation models has-a relationships.**
- The owner object is called an aggregating object, and its class is called an aggregating class. The subject object is called an aggregated object, and its class is called an aggregated class.
- An object can be owned by several other aggregating objects. If an object is exclusively owned by an aggregating object, the relationship between the object and its aggregating object is referred to as a **composition**.



Class Relationship in Java

Aggregation and Composition

```
public class Name {
   ...
}
```

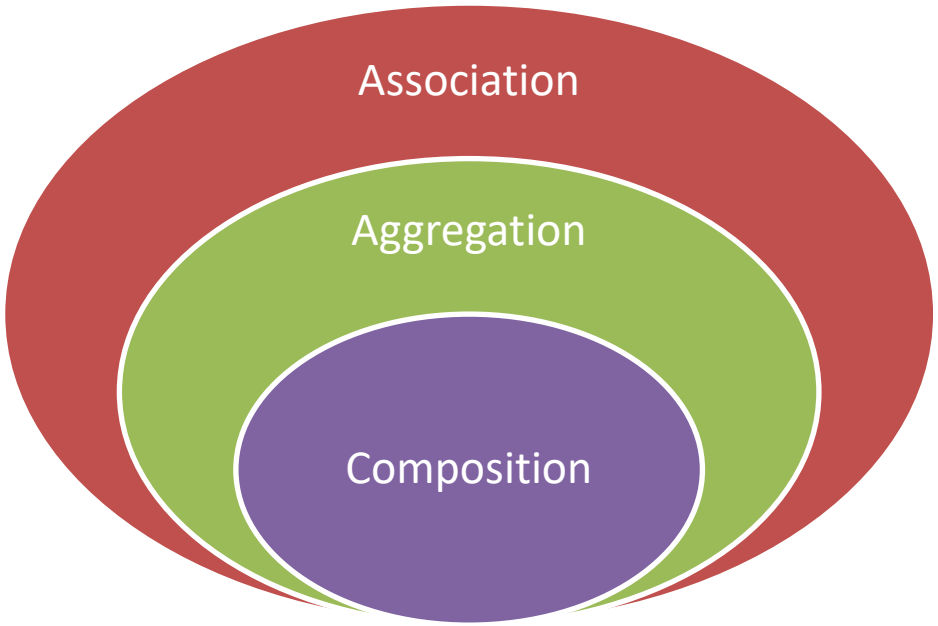
Aggregated class

```
public class Student {
   private Name name;
   private Address address;
   ...
}
```

Aggregating class

```
public class Address {
   ...
}
```

Aggregated class



Primitive Data Type and Wrapper Class Types

- A primitive type value is not an object, but it can be wrapped in an object using a wrapper class in the Java API.

- Java offers a convenient way to incorporate, or wrap, a primitive data type into an object.
- e.g. wrapping int into the Integer class, wrapping double into the Double class, and wrapping char into the Character class etc.
- Java provides Boolean, Character, Double, Float, Byte, Short, Integer, and Long wrapper classes in the java.lang package for primitive data types.

Wrapper Classes

- The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

- Since J2SE 5.0, autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

- Change the value in Method: Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- Serialization: We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- Synchronization: Java synchronization works with objects in Multithreading.
- java.util package: The java.util package provides the utility classes to deal with objects.
- Collection Framework: Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Use of Wrapper classes in Java

The eight classes of the java.lang package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	**Wrapper class**
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Use of Wrapper classes in Java


```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer

int a=20;
Integer i=Integer.valueOf(a);    //converting int into Integer explicitly
Integer j=a;   //autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);
}}
```

Use of Wrapper classes in Java


```
//Java program to convert object into primitives
//Unboxing example of Integer to int

public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly

int j=a;   //unboxing, now compiler will write a.intValue() internally

System.out.println(a+" "+i+" "+j);
}}
```

Big integer and Big decimal class

- The BigInteger and BigDecimal classes can be used to represent integers or decimal numbers of any size and precision.

- If you need to compute with very large integers or high-precision floating-point values, you can use the BigInteger and BigDecimal classes in the java.math package.
- Both are immutable.

Big Integer


```
import java.math.*;
public class Main{
public static void main(String[] args) {
int aa = 92233720368547758O7;
int bb = 2 ;
int cc = aa*bb;
System.out.println(cc);

BigInteger a = new BigInteger("92233720368547758O7");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 92233720368547758O7 * 2
System.out.println(c);
}}
```

Big Decimal


```
import java.math.*;
public class Main{
public static void main(String[] args)    {

BigDecimal bd1 = new BigDecimal("124567890.0987654321");
BigDecimal bd2 = new BigDecimal("987654321.123456789");

bd1 = bd1.add(bd2);
System.out.println("BigDecimal1 = " + bd1);
bd1 = bd1.multiply(bd2);
System.out.println("BigDecimal1 = " + bd1);
}}
```

String class, String Builder and String Buffer

- Covered in previous chapter.

Super Class and Subclass


```
Class A            //Base class //Super class // Parent class
{
…
}
Class B extends A        //Derived class //Sub class //Child class
{

..//Use of class A properties

}
```

Super Class and Subclass


```
Class A            //Base class //Super class // Parent class
{
…
}
Class B extends A        //Derived class //Sub class //Child class
{

..//Use of class A properties

}
```

Inheritance

- It is defined as the process where derived class can borrow the properties of base class.
- Inheritance can be achieved by using the “extends” keyword.

Advantages:
i. Code reusability
ii. Extensibility
iii. Data hiding
iv. Overriding

Inheritance

Types of inheritance

1. Single inheritance - It has one parent per derived class. | Whenever a class inherits another class, it is called single inheritance.

2. Multilevel inheritance – when a derived class is derived from base class which itself is derived again is called multilevel inheritance. | Multilevel inheritance is a part of single inheritance where more than two classes are in the sequence.

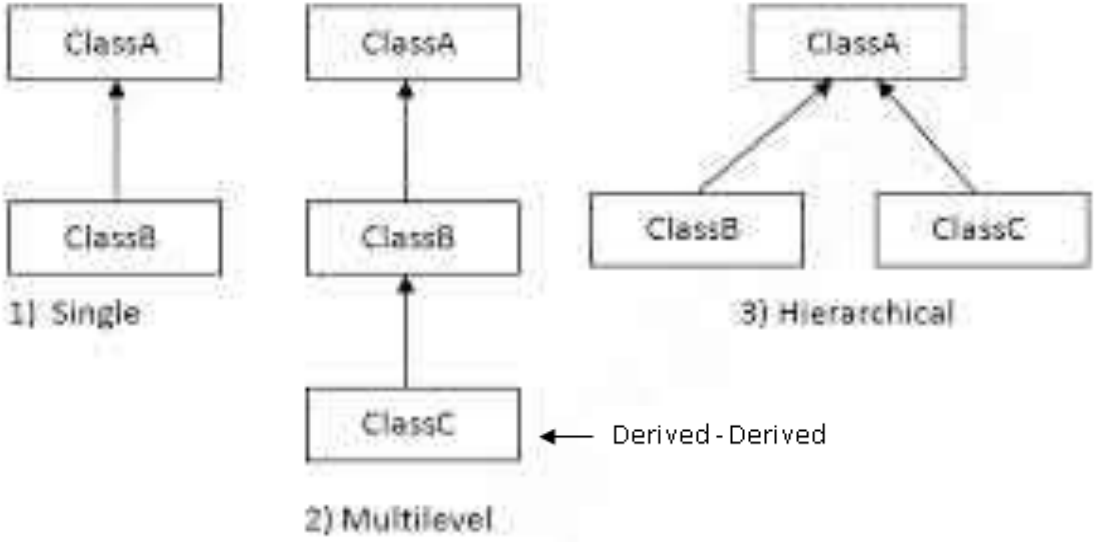
3. Hierarchical inheritance - If a class has more than one derived classes This is known as hierarchal inheritance.

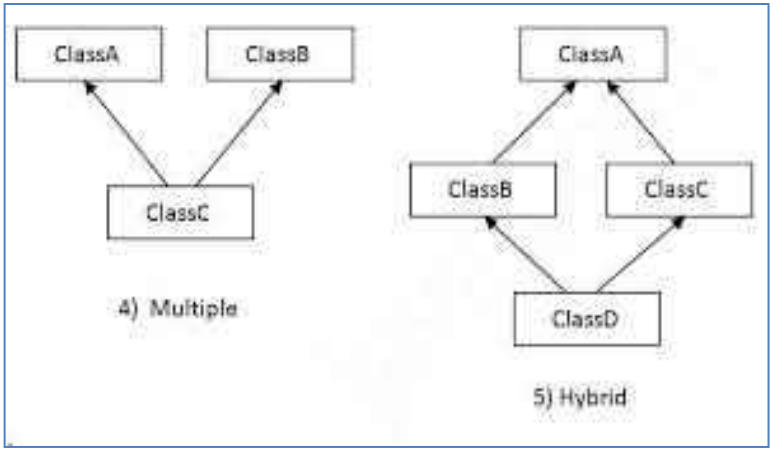
4. Multiple inheritance - If a class inherits the data member and member function from more than one base class, then this type of inheritance is called multiple inheritance. It is not supported in java so "Interface" concept is used.

5. Hybrid inheritance – when two or more types of inheritances(Single and multiple inheritance) are combined together then it is called hybrid inheritance

Inheritance

Types of inheritance





Inheritance

```
//Single inheritance – Example
class a                 //base class
{           void disp()         //method 1
            {
                        System.out.println("Hello");
            }}
class b extends a  //derived class
{           void disp1()        //method 2
            {
            System.out.println(" Good morning");
            }}
class singleinh     //main class
{           public static void main(String args[])          //main function
            {
            b B1 = new b();   // reference variable A1, Creating object to allocate memory space
            B1.disp1();                     //calling method
            B1.disp();
            }}
```

Inheritance

```
//Multilevel inheritance – Example
class a
{           void disp()
            {  System.out.println("Helllo");       }}
class b extends a
{           void disp1()
            {  System.out.println("Good morning");        } }
class c extends b
{           void disp2()
            {  System.out.println("Welcome to BEC");  }
}
class multilevelinh{
            public static void main(String args[])          {
            c C1 = new c();
            C1.disp2();
            C1.disp1();
            C1.disp();
            }}
```

Inheritance

```
//Hierarchical inheritance – Example
class a
{           void disp()
            {  System.out.println("Helllo");       }}
class b extends a
{           void disp1()
            {  System.out.println("Good morning");        } }
class c extends a
{           void disp2()
            {  System.out.println("Welcome to BEC");  }
}
class multilevelinh{
            public static void main(String args[])          {
            c C1 = new c();
            C1.disp2();
            C1.disp1();
            C1.disp();
            }}
```

Inheritance

Why Interface should be used in multiple inheritance? Consider the example

```
class A {
void msg()
{
System.out.println("Hello");
}  }
class B {
void msg()
{
System.out.println("Welcome");
}  }
class C extends A,B //suppose if it were
{    public static void main(String args[])     {
            C obj=new C();
            obj.msg();//Now which msg() method would be invoked?
    }  }
```

Compile time error occurs

Inheritance

Marwadi education foundation

```
//Multiple inheritance example
interface vehicleone {
        int  speed=90;
        public void distance();
}
interface vehicletwo {
        int distance=200;
        public void speed();
}
class Vehicle  implements vehicleone, vehicletwo //two interfaces
{       public void distance() {
                int  distance=speed*200;
                System.out.println("distance travelled is "+distance);
        }
        public void speed() {
                int speed=distance/90;
                System.out.println("Speed is "+speed);
        }
}
```

```
class MultipleInterface{
        public static void main(String args[]){
                Vehicle obj = new Vehicle();
                obj.distance();
                obj.speed();}  }
```

Inheritance

```
class A                 //class
{           int a=10; //variable with value
assigned
}

interface B         //Interface
{int b=20;
}
//Multiple inh.
class C extends A implements B      {
int c;
int mul()
            //method1
{           c=a*b;
            return c;
}
}
```

```
class D extends C                   //Single inh.
{void sum()                                 //method2
            { System.out.println("Adding all 3
variables");
            int d=a+b+mul();
            //Calc
            System.out.println(d);
            }}
class hybridinh       //main class
{
public static void main(String[] args)     //main fun.
            { C obj1 = new C();
            //Obj. creation
            D obj2 = new D();
            System.out.println("Multiplying two
variables");
            System.out.println(obj1.mul());
            //Print stmt
            obj2.sum();
            //Calling method
            }}
```

Abstract Class Vs Interface

Marwadi education foundation

S.No	Abstract class (Cannot be instantiated)	Interface (Cannot be instantiated)
1.	Partial abstraction	100% data abstraction
2.	Programmer knows 50% of implementation	Programmer doesn't aware of implementation.
3.	A class can inherit only one abstract class.	A class can implement more than one interface
4.	Methods may or may not have implementation.	Methods have no implementation.
5.	Efficient performance	Slow performance
6.	Extends keyword used	Implements keyword
7.	Members of abstract class can have public, private, protected.	Members of interface are public by default

Using Super Keyword

The super keyword in Java is a reference variable which is used to refer immediate parent class object.

Conditions where super key word used:

1. Only used in subclass constructor and methods.
2. Call to super must first statement in subclass constructor.
3. Parameters must be in same order.

Using Super Keyword


```
//Use of super with variables:
class Vehicle
{    int maxSpeed = 120; }

class Car extends Vehicle
{    int maxSpeed = 180;
     void display()     {
        System.out.println("Maximum Speed: " + super.maxSpeed);
    }}

class Test
{    public static void main(String[] args) {
        Car small = new Car();
        small.display();
    } }
```

Using Super Keyword


```
//Use of super with methods
class Person
{   void message()   {
      System.out.println("This is person class");
   }
}
 class Student extends Person
{   void message()
   {      System.out.println("This is student class");
   }
    void display()
   {       super.message();
       message();
   }
}
 class Test
{   public static void main(String args[])   {
      Student s = new Student();
       s.display();
   }}
```

Using Super Keyword


```
//Use of super with constructors
class Person {
   Person()   {
      System.out.println("Person class Constructor");
   }
   Person(int i)    {
      System.out.println("Person class Const. with param");
   }}
 class Student extends Person {
   Student()    {
            System.out.println("Student class Constructor");
   }
   Student(int i)    {
       super(i);
       System.out.println("Student class Const. with param");
   }}
 class Main {
   public static void main(String[] args)    {
      Student s = new Student(5);
   } }
```

Overriding and Overloading Methods

Method Overriding

- Declaring a method in sub class which is already present in parent class 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.
- Occurs during runtime
- Performed between two classes.

Rules:

- Private method cannot be overridden. (so don't use private access specifier)
- Static method can be inherited but cannot be overridden
- Method overriding occurs only when name of two methods are same.
- Arguments must be same and in same order from both child and parent class.

Overriding and Overloading Methods

```
//Method Overriding
class parent
{           //Overridden method
            public void property(){
             System.out.println("Land+Property+Cash");
            }
            public void marriage()
            {
                System.out.println(“abc");
            }
}
class son extends parent
{           public void marriage() //Overriding method
            {
                        System.out.println(“xyz");
            }
 public static void main( String args[])
  {     son obj = new son();
     //This will call the child class version of marriage()
   obj.property();   obj.marriage();
   }}
```

Overriding and Overloading Methods

Method Overloading

- It means many methods can have same name but can pass different number of parameters
- Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different.

Rules:

- Performed with in class
- Functions may have different return types.

Overriding and Overloading Methods


```
//Method overloading
class Main
{
    private static void display(int a)
            {
       System.out.println("Arguments: " + a);
            }
    private static void display(int a, int b)
            {
       System.out.println("Arguments: " + a + " and " + b);
            }
    public static void main(String[] args)
            {
       display(1);
       display(1, 4);}}
```

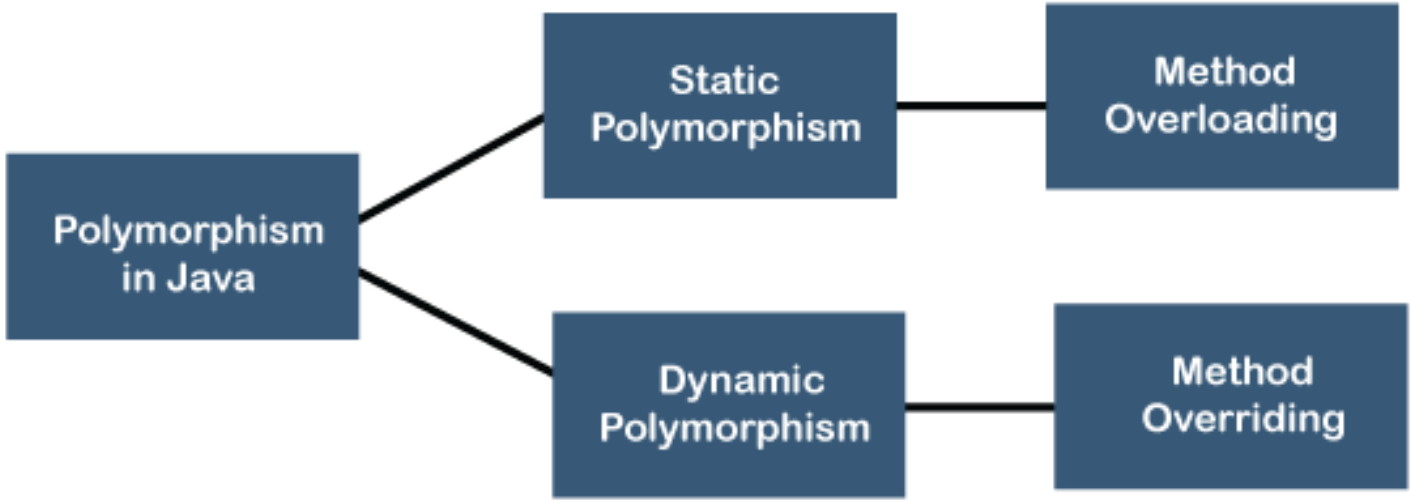
Polymorphism and Dynamic Binding

The word polymorphism is a combination of two words i.e. ploy and morphs. The word poly means many and morphs means different forms. In short, a mechanism by which we can perform a single action in different ways.

Types of Polymorphism

There are two types of polymorphism in Java:

- Static Polymorphism (Compile Time Polymorphism)
- Dynamic Polymorphism (Run Time Polymorphism)



Data Type Casting

Type casting is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

There are two types of type casting:

- Implicit or Widening Type Casting (automatic)
  - **byte -> short -> char -> int -> long -> float -> double**

- Explicit or Narrowing Type Casting (programmer)
  - **double -> float -> long -> int -> char -> short -> byte**

Data Type Casting


```
public class Implicit_WideningCasting
{
public static void main(String[] args)
{
int x = 7;
//automatically converts the integer type into long type
long y = x;
//automatically converts the long type into float type
float z = y;
System.out.println("Before conversion, int value "+x);
System.out.println("After conversion, long value "+y);
System.out.println("After conversion, float value "+z);
}
}
```

Data Type Casting


```
public class Explicit_NarrowingCasting
{
public static void main(String args[])  {
double d = 166.66;
//converting double data type into long data type
long l = (long)d;
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
}
}
```

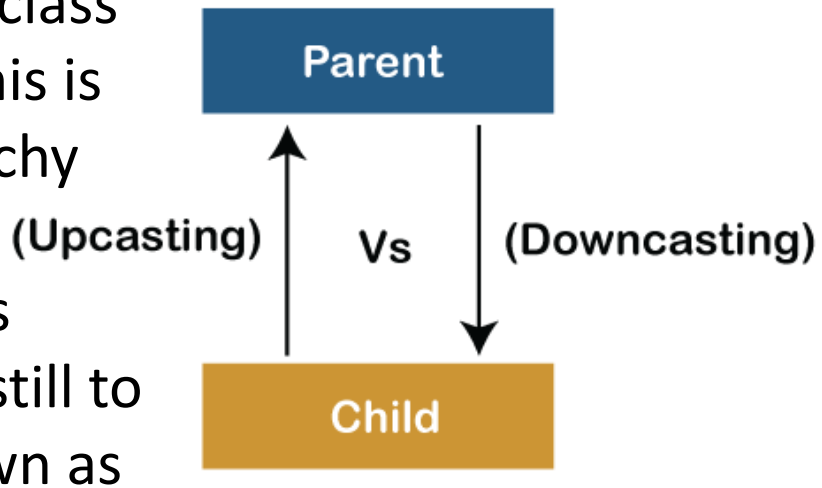
Casting Objects

Marwadi education foundation

Assigning one data type to another or one object to another is known as casting. Java supports two types of casting – data type casting and object casting.

Conditions of Object Casting

- Same class objects can be assigned one to another.
- Subclass object can be assigned to a super class object and this casting is done implicitly. This is known as Upcasting (upwards in the hierarchy from subclass to super class).
- Java does not permit to assign a super class object to a subclass object (implicitly) and still to do so, we need explicit casting. This is known as downcasting (super class to subclass). Downcasting requires explicit conversion.



Casting Objects


```
class Flower{
public void smell()  {
   System.out.println("All flowers
give smell,
 if you can smell");
  }
}
public class Rose extends Flower{
public void smell()  {
   System.out.println("Rose gives
rosy smell");
  }
  public static void main(String
args[])
  {
Flower f = new Flower();
 Rose r = new Rose();
 f.smell();
 r.smell();
 f = r;          // subclass to super class, it is valid
 f.smell();
 // r  = f;     // super class to subclass, not valid
 r = (Rose) f;   // explicit casting
 f.smell();
  }
}
```

instanceof Keyword

- instanceof is a keyword that is used for checking if a reference variable is containing a given type of object reference or not.

```
// Java Program to Illustrate instanceof Keyword
// Importing required I/O classes
import java.io.*;
// Main class
class Main {
        public static void main(String[] args)
        {
                // Creating object of class inside main()
                Main object = new Main();
                // Returning instanceof
                System.out.println(object instanceof Main);
        }
}
```

Object Class

The Object class is the parent class of all the classes in java by default.
In other words, it is the topmost class of java.

```
class A extends Object
{ }
Class B extends A
{ }
class objClassDemo
{
public static void main(String args[])
{
        A obj = new A();
        System.out.println("obj:" +obj); //or +obj.toString()
}}
Output: A@3e25a5  //ClassName@HexaDecimalCode
```

Object Class

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait()throws InterruptedException	
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

The ArrayList class and Its Methods

- The ArrayList class implements the List interface. It is used to implement dynamic array.
- The size of array extended automatically. When objects removed the size will be reduced.

- The collection framework provides a well designed set of interface and classes for storing and manipulating group of data as a single unit. (Collection classes are AbstractList, LinkedList, ArrayList, HashMap, TreeMap, TreeSet etc.,)

The ArrayList class and Its Methods

Advantages:

1. ArrayList can contain duplicate elements
2. ArrayList maintains insertion order
3. It is non-synchronized
4. Allows random access
5. Type safe

Disadvantages:

1. Manipulation (Remove, Replace) is slow because lot of shifting needed.

The ArrayList class and Its Methods


```
import java.util.*;
 public class Main{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();

      list.add("Fruit");//Adding object in arraylist
      list.add("Apple");
      list.add("Banana");
      list.add("Grapes");

      System.out.println(list);
 }
}
```

The ArrayList class and Its Methods


```
import java.util.*;
public class Main{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
  list.add("Mango");//Adding object in arraylist
  list.add("Apple");
  list.add("Banana");
  list.add("Grapes");

  //Traversing list through for-each loop
  for(String fruit:list)
    System.out.println(fruit);

 }
}
```

The ArrayList class and Its Methods

› ArrayList() // Creates an empty list.
› add(o: E) // Appends a new element o at the end of this list.
› add(index: int, o: E) // Adds a new element o at the specified index in this list.
› clear() // Removes all the elements from this list.

› contains(o: Object) // Returns true if this list contains the element o.
› get(index: int) // Returns the element from this list at the specified index.
› indexOf(o: Object) // Returns the index of the first matching element in this list.

The ArrayList class and Its Methods

› isEmpty() // Returns true if this list contains no elements.
› lastIndexOf(o: Object) // Returns the index of the last matching element in this list.

› remove(o: Object) // Removes the first element o from this list. Returns true if an element is removed.
› size() // Returns the number of elements in this list.

› remove(index: int) // Removes the element at the specified index. Returns true if an element is removed.
› set(index: int, o: E) // Sets the element at the specified index.

The ArrayList class and Its Methods

Operation	Array	ArrayList
Creating an array/ArrayList	String[] a = new String[10]	ArrayList<String> list = new ArrayList<>();
Accessing an element	a[index]	list.get(index);
Updating an element	a[index] = "London";	list.set(index, "London");
Returning size	a.length	list.size();
Adding a new element		list.add("London");
Inserting a new element		list.add(index, "London");
Removing an element		list.remove(index);
Removing an element		list.remove(Object);
Removing all elements		list.clear();

The Protected Data and Methods

- Often it is desirable to allow subclasses to access data fields or methods defined in the superclass, but not to allow nonsubclasses to access these data fields and methods.
- A protected member of a class can be accessed from a subclass.

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass in a different package	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	-
default (no modifier)	✓	✓	-	-
private	✓	-	-	-

Marwadi education foundation

END OF UNIT - 5