# UNIT-2

Presented By:
Dr. Deepak Kumar Sharma
Asst. Professor(SG)
SoCS UPES Dehradun

UNIT-II

- Class Fundamentals, Objects, Constructors, Garbage Collection, this Keyword, Java's Access Modifiers, Method Overloading, static Keyword

- Inheritance, Types of Inheritance, super to Access Superclass Members, Method Overriding

- Abstract Classes, Using final,

- Packages and Interfaces, Build-in Interface, User defined Interfaces.

Inheritance
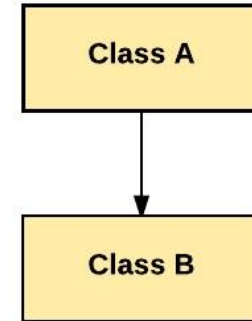
- **Inheritance** is a mechanism in which one class acquires the property of another class.
    - E.g. a child inherits the traits of his/her parents.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

WHY INHERITANCE?

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Key Terms:

- **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.

Extends

- The **extends keyword** indicates that you are making a new class that derives from an existing class.

  class Subclass-name **extends** Superclass-name

  {

     //methods and fields

  }

Example

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

O/P:
Programmer salary is:40000.0
Bonus of programmer is:10000

Example:

```
class Vehicle {
  protected String brand = "TATA";
  public void honk() {
    System.out.println("Horn PLz!");
  }
}
class Car extends Vehicle {
  private String modelName = "NEXON";
  public static void main(String[] args) {
    Car myFastCar = new Car();
    myFastCar.honk();
    System.out.println(myFastCar.brand + " " + myFastCar.modelName);
  }
}
```
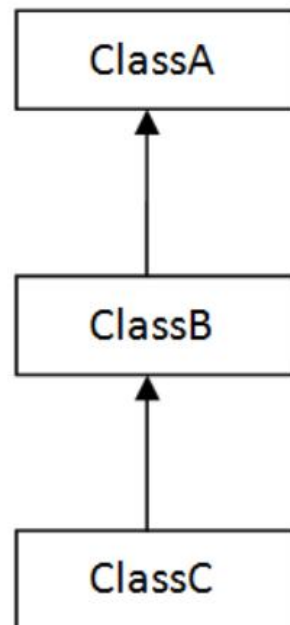
O/P:
Horn PLz!
TATA NEXON

Types of inheritance in java

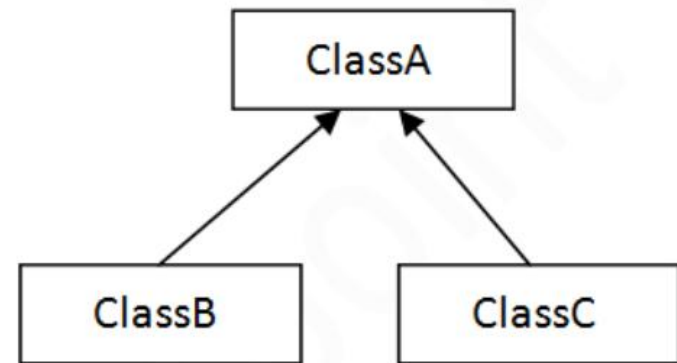- On the basis of class, there can be three types of inheritance in java:
  - Single
  - multilevel
  - hierarchical
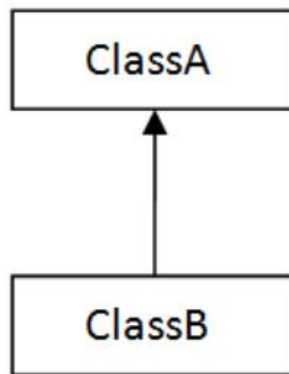
1) Single

2) Multilevel

3) Hierarchical

Dr. Deepak Sharma, UPES
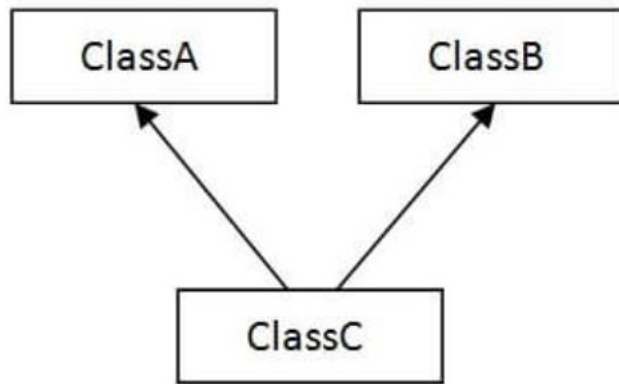
Types of Inheritance

- In java programming, *multiple and hybrid inheritance* is supported through interface only.

4) Multiple
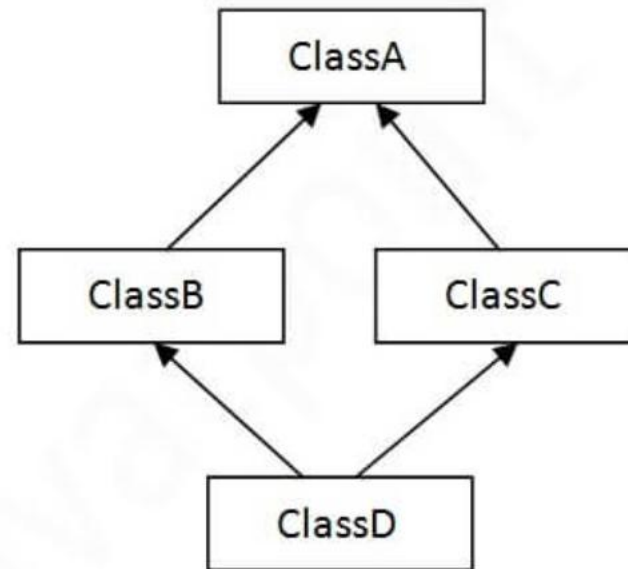
5) Hybrid

Note: Multiple inheritance is not supported in Java through class.

Single Inheritance Example

```java
//Parent Class
class Animal{
void eat(){System.out.println("eating...");}
}

//Child Class
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();
}}
```

O/P:
barking...
eating...

Multilevel Inheritance Example

- When there is a chain of inheritance, it is known as *multilevel inheritance*.

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

O/P:
weeping...
barking...
eating...

Hierarchical Inheritance Example

- When two or more classes inherits a single class.

```java
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 TestInheritance{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//Compile time Error
}}
```

Output:
meowing...
eating…

Why multiple inheritance is not supported in java?

- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

- To avoid ambiguity while calling same method present in both parent classes.

```java
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?
}
}
```

Output: Compile time Error

Aggregation in Java

- If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship (part of relationship).

class Employee{

int id;

String name;

Address address;//Address is a class

…

}

- Employee has an entity reference address, so relationship is Employee HAS-A address.

Dr. Deepak Sharma, UPES

Example of Aggregation

```java
// Address class
class Address {
    String city, state, country;

    public Address(String city, String state, String country) {
        this.city = city;
        this.state = state;
        this.country = country;
    }

    void displayAddress() {
        System.out.println("City: " + city + ", State: " + state + ", Country: " + country);
    }
}


// Student class with Aggregation
class Student {
    int id;
    String name;
    Address address; // Aggregation: Student "has-a" Address

    public Student(int id, String name, Address address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }

    void displayStudent() {
        System.out.println("Student ID: " + id + ", Name: " + name);
        address.displayAddress();
    }
}
```

```java
// Main class
public class AggregationExample {
    public static void main(String[] args) {
        Address addr1 = new Address("Mumbai", "Maharashtra", "India");
        Student student1 = new Student(101, "Rahul", addr1);

        student1.displayStudent();
    }
}
```

Dr. Deepak Sharma, UPES

When to use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.

- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Dr. Deepak Sharma, UPES

Method Overloading and Overriding

Method Overloading

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

- Method overloading *increases the readability of the program*.

- There are two ways to overload the method in java
  - By changing number of arguments
  - By changing the data type

- In Java, Method Overloading is not possible by changing the return type of the method only.

```java
class OverloadDemo {                    //method overloading
    void test() {
        System.out.println("No parameters");
    }
    void test(int a) {
        System.out.println("a: " + a);
    }
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }
    double test(double a) {
        System.out.println("double a: " + a);
        return a*a;
    }
}

class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
    }
}
```

Can we overload java main() method?

- Yes, by method overloading. You can have any number of main methods in a class by method overloading.

- But JVM calls main() method which receives string array as arguments only.

```java
class TestOverloading{
public static void main(String[] args){System.out.println("main with String[]");}
public static void main(String args){System.out.println("main with String");}
public static void main(){System.out.println("main without args");}
}
```

Output: main with String[]

Method Overriding in Java

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.

- Method overriding is used for runtime polymorphism.

- Method overriding applies only to methods, not variables.

- If a subclass declares an instance variable with the same name as the parent class, the subclass variable hides the parent's variable.

Rules for Java Method Overriding
1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Problem without method overriding

```java
class Vehicle{
 void run(){System.out.println("Vehicle is running");}
}

//Creating a child class
class Bike extends Vehicle{
 public static void main(String args[]){
 //creating an instance of child class
 Bike obj = new Bike();
 //calling the method with child class instance
 obj.run();
 }
}                              Output: Vehicle is running
```

Problem: Need to provide a specific implementation of run() method in subclass

Example: method overriding

```java
class Vehicle{
 //defining a method
 void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike extends Vehicle{
 //defining the same method as in the parent class
 void run(){System.out.println("Bike is running safely");}

 public static void main(String args[]){
 Bike obj = new Bike();//creating object
 obj.run();//calling method
 }
}
```

Output: Bike is running safely

Example: Method Overriding

```java
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 4;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 3;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 5;}
}
//Test class to create objects and call the methods
class Test{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}
```
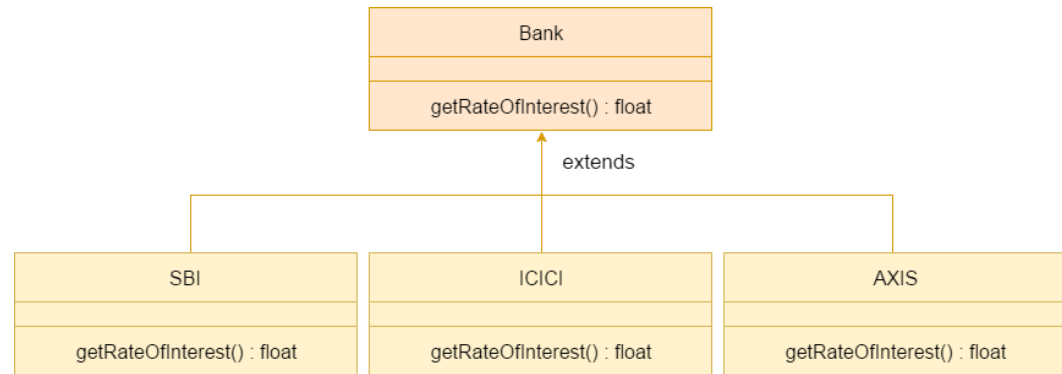

Bank
getRateOfInterest() : float

extends

| SBI | ICICI | AXIS |
| getRateOfInterest() : float | getRateOfInterest() : float | getRateOfInterest() : float |

Output:
SBI Rate of Interest: 4
ICICI Rate of Interest: 3
AXIS Rate of Interest: 5

Dr. Deepak Sharma, UPES

Can we override static method?

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

Can we override java main method?

- No, because the main is a static method.

Super Keyword

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

- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate **parent class instance variable**.
2. super can be used to invoke immediate **parent class method**.
3. super() can be used to invoke immediate **parent class constructor**.

Super- to refer immediate parent class instance variable

- to access the data member or field of parent class.
- if parent class and child class have same fields.

```java
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

Output:
Black
white

Super- to invoke parent class method

- It should be used if subclass contains the same method as parent class (Methods are overridden).

```java
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

Output:
eating…
barking…

Super() - to invoke parent class constructor

- To invoke parent class constructor.

```java
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Output:
animal is created
dog is created

Super() - to invoke parent class constructor

- Note: super() is added in each class constructor automatically as the first statement by compiler if there is no super() or this().

```java
class Animal{
Animal(){System.out.println("animal is created");}
}

class Dog extends Animal{
Dog(){
System.out.println("dog is created");
}
}

class TestSuper{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Output:
animal is created
dog is created

Example- Super()

```
class Person{
int id;
String name;
Person(int id,String name){
this.id=id;
this.name=name;
}
}
```

```
class Emp extends Person{
float salary;
Emp(int id,String name,float salary){
super(id,name);//reusing parent constructor
this.salary=salary;
}
void display(){System.out.println(id+" "+name+" "+salary);}
}
```

```
class TestSuper{
public static void main(String[] args){
Emp e1=new Emp(4,"aman",95000f);
e1.display();
}}
```

Output:
4 aman 95000

Instance initializer block- Example

- **Instance Initializer block** is used to initialize the instance data member.

- It run each time when object of the class is created.

```java
class Bike{
    int speed;

    Bike(){System.out.println("speed is "+speed);}

    {speed=100;}

    public static void main(String args[]){
    Bike b1=new Bike();
    Bike b2=new Bike();
    }
}
```

Output:
speed is 100
speed is 100

Instance initializer block

- It run each time when object of the class is created.

We can directly assign a value in instance data member

```java
class Bike{
    int speed=100;
}
```

Why initializer block?

- To perform some operations while assigning value to instance data member
  - e.g. a for loop to fill a complex array or error handling etc.

There are three places in java where you can perform operations:
- method
- constructor
- block

Instance initializer block

What is invoked first, instance initializer block or constructor?

```java
class Bike{
    int speed;

    Bike(){System.out.println("constructor is invoked");}

    {System.out.println("instance initializer block invoked");}

    public static void main(String args[]){
    Bike b1=new Bike();
    Bike b2=new Bike();
    }
}
```

Output:
instance initializer block invoked
constructor is invoked
instance initializer block invoked
constructor is invoked

- Instance intializer block is invoked at the time of object creation.

Rules for instance initializer block :

Three rules:

- The instance initializer block is created when instance of the class is created.

- The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).

- The instance initializer block comes in the order in which they appear.

Program of instance initializer block that is invoked after super()

```
class A{
A(){
System.out.println("parent class constructor invoked");
}
}
class B2 extends A{
B2(){
super();
System.out.println("child class constructor invoked");
}

{System.out.println("instance initializer block is invoked");}

public static void main(String args[]){
B2 b=new B2();
}
}
```

Note: Sequence of execution when an object is instantiated:
- Memory for the object is allocated.
- Any default values for the instance variables are set.
- The superclass's constructor is invoked.
- The instance initializer block(s) are executed in the order in which they appear in the class.
- The constructor is executed.

Output:
    parent class constructor invoked
    instance initializer block is invoked
    child class constructor invoked

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

- variable
- method
- Class

- The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable.

- It can be initialized in the constructor only.

- The blank final variable can be static which will be initialized in the static block only.

Java final variable

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

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

Output: Compile Time Error

Java final variable

```java
class Bike{
    final int speedlimit; // final variable

    Bike() {
        speedlimit = 400; // Initializing the final variable in the constructor
    }

    void run() {
        System.out.println("Speed Limit: " + speedlimit);
    }

    public static void main(String args[]) {
        Bike obj = new Bike();
        obj.run();
    }
} // end of class
```

Output: Speed Limit: 400

Java final method

- If you make any method as final, you cannot override it.

```java
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();
  }                                         Output: Compile Time Error
}
```

Java final class

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

```java
final class Bike{}

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();
 }
}
```

Output: Compile Time Error

Is final method inherited?

- Yes, final method is inherited but you cannot override it.

```
class Bike{
  final void run(){System.out.println("running...");}
}
class Honda extends Bike{
  public static void main(String args[]){
   new Honda().run();
  }
}
```

Can we declare a constructor final?

- No, because constructor is never inherited.

Blank or uninitialized final variable

- A final variable that is not initialized at the time of declaration.

- If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed. (e.g. PAN Card Number)

- It can be initialized only in constructor.

```java
class Student {
    private String name;
    private final String panCardNumber;

    public Student(String name, String panCardNumber) {
        this.name = name;
        this.panCardNumber = panCardNumber;
    }

    public void displayDetails() {
        System.out.println(name + " has PAN: " + panCardNumber);
    }

    public static void main(String[] args) {
        Student deepak = new Student("Deepak", "XYZDE5678G");
        deepak.displayDetails();
    }
}
```

Output:
Deepak has PAN: XYZDE5678G

static blank final variable

- A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

```java
class A{
  static final int data;//static blank final variable
  static{ data=50;}
  public static void main(String args[]){
    System.out.println(A.data);
  }
}
```

final parameter

- If you declare any parameter as final, you cannot change the value of it.

```java
class Bike{
 int cube(final int n){
  n=n+2;//can't be changed as n is final
  n*n*n;
 }
 public static void main(String args[]){
  Bike b=new Bike();
  b.cube(5);
 }
}
```

Polymorphism in Java

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*.
- Types:
  - compile-time polymorphism
  - runtime polymorphism
- polymorphism in java is done by
  - method overloading
  - method overriding
- Compile time polymorphism
  - Example- Overload a static method
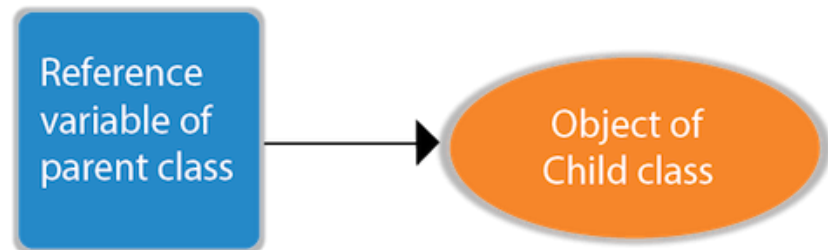
Runtime Polymorphism in Java

- **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

- An overridden method is called through the reference variable of a superclass.

- The determination of the method to be called is based on the object being referred to by the reference variable.

- Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

Runtime Polymorphism

Upcasting

- If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.

```
class A{}
class B extends A{
public static void main(String args[]){
A a=new B(); //upcasting
}
}
```

Example of Java Runtime Polymorphism

```java
class Bike{
 void run(){System.out.println("running");}
}
class Splendor extends Bike{
 void run(){System.out.println("running safely with 60km");}

 public static void main(String args[]){
  Bike b = new Splendor();//upcasting
  b.run();
 }
}
```

Output:
running safely with 60km.

Example

```java
class Bank{
float getRateOfInterest(){return 0;}
}
class SBI extends Bank{
float getRateOfInterest(){return 8.4f;}
}
class ICICI extends Bank{
float getRateOfInterest(){return 7.3f;}
}
class AXIS extends Bank{
float getRateOfInterest(){return 9.7f;}
}
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
b=new ICICI();
System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
b=new AXIS();
System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
}
}
```

Output:

SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7

Java Runtime Polymorphism with Data Member

- A method is overridden, not the data members, so runtime polymorphism can't be achieved by other data members .

```java
class Bike{
 int speedlimit=90;
}
class Honda extends Bike{
 int speedlimit=150;

 public static void main(String args[]){
  Bike obj=new Honda();
  System.out.println(obj.speedlimit);//90
 }
```

Output:

90

Java Runtime Polymorphism with Multilevel Inheritance

```java
class Animal{
void eat(){System.out.println("eating");}
}
class Dog extends Animal{
void eat(){System.out.println("eating fruits");}
}
class BabyDog extends Dog{
void eat(){System.out.println("drinking milk");}
public static void main(String args[]){
Animal a1,a2,a3;
a1=new Animal();
a2=new Dog();
a3=new BabyDog();
a1.eat();
a2.eat();
a3.eat();
}
}
```
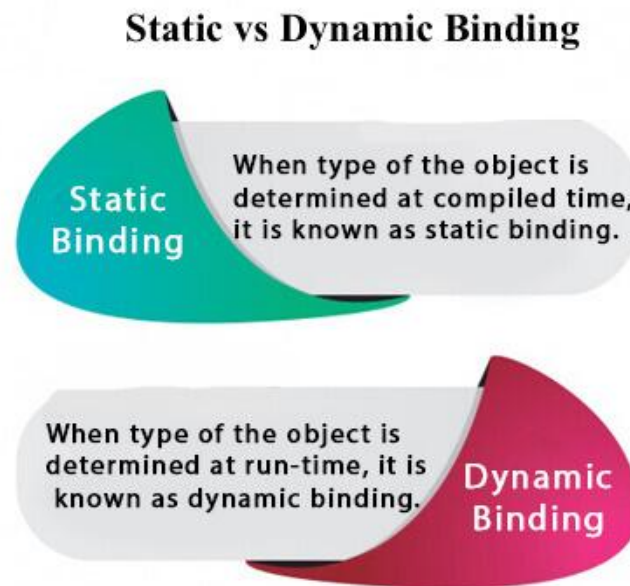
Output:

eating
eating fruits
drinking Milk

Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

- There are two types of binding
  - Static Binding (also known as Early Binding).
  - Dynamic Binding (also known as Late Binding).

Static vs Dynamic Binding

Static Binding — When type of the object is determined at compiled time, it is known as static binding.

When type of the object is determined at run-time, it is known as dynamic binding. — Dynamic Binding

Understanding Type

1) variables have a type

- Each variable has a type, it may be primitive and non-primitive.

  int data=30;

- Here data variable is a type of int.

2) **References have a type**

class Dog{

 public static void main(String args[]){

  Dog d1**;//Here d1 is a type of Dog**

 } }

Understanding Type

3) **Objects have a type**

An object is an instance of particular java class, but it is also an instance of its superclass.

```
class Animal{}
class Dog extends Animal{
 public static void main(String args[]){
  Dog d1=new Dog();
 }  }
```

- Here d1 is an instance of Dog class and is also an instance of Animal.

static binding

- When type of the object is determined at compiled time(by the compiler), it is known as static binding.

- **If there is any private, final or static method in a class, there is static binding.**

Example of static binding

```
class Dog{
 private void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();
 }
}
```

Dynamic binding

- When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding
```
class Animal{                                               Output:dog is eating...
 void eat(){System.out.println("animal is eating...");}
}


class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 }
}
```

- Object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.
- So compiler doesn't know its type, only its base type.

instanceof operator

- The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

- The instanceof in java is also known as type *comparison operator* because it compares the instance with type.

- It returns either true or false.

- If we apply the instanceof operator with any variable that has null value, it returns false.

```java
class Simple{
public static void main(String args[]){
Simple s=new Simple();
System.out.println(s instanceof Simple);//true
}
}
```

Output: true

instanceof operator

- An object of subclass type is also a type of parent class.

```java
class Animal{}
class Dog extends Animal{//Dog inherits Animal


 public static void main(String args[]){
 Dog d=new Dog();
 System.out.println(d instanceof Animal);//true
 }
}
```

Output:true

Note:
Dog extends Animal therefore object of Dog can be referred by either Dog or Animal class.

- If we apply instanceof operator with a variable that have null value, it returns false

```java
class Dog{
 public static void main(String args[]){
  Dog d=null;
  System.out.println(d instanceof Dog);//false
 }
}
```

Output: false

Unit 2 Cont..
More Practice Required!!!