Inheritance, Packages & Interfaces

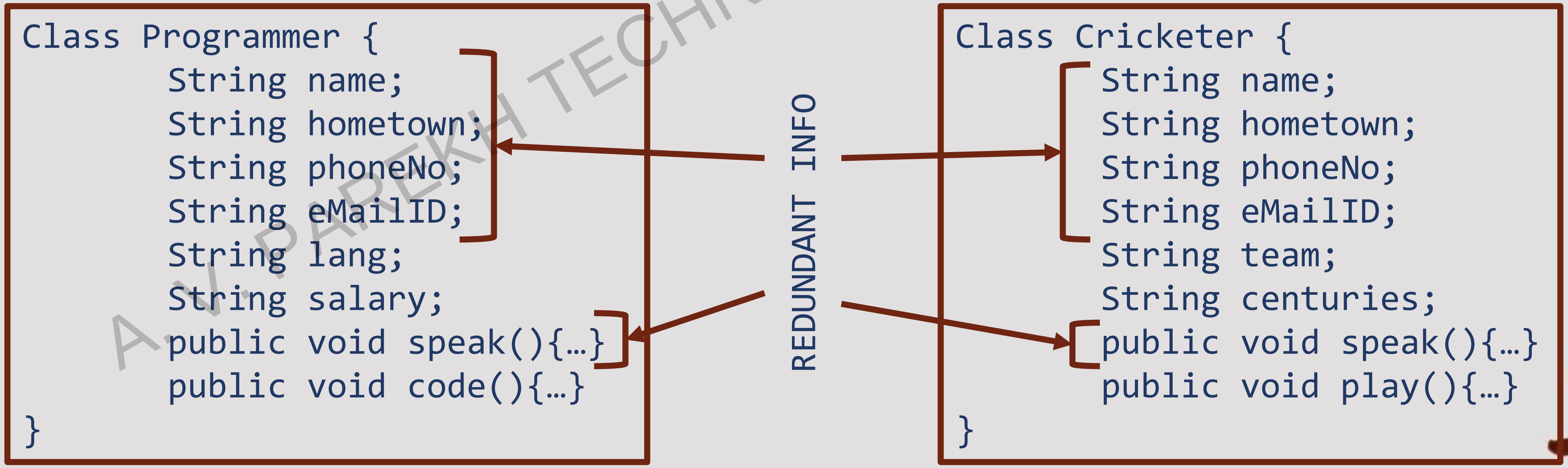
Unit- III

Topics to be Covered…

- Basics of Inheritance and Types of Inheritance
- Super Keyword
- Method Overriding
- Understanding Objects with Inheritance
- Dynamic Method Dispatch
- Abstract Class
- Final Class
- Basics of Interface
- Implementing interface
- Multiple Inheritance using Interface
- Basics of Package
- Creating and Importing Package
- Access Rules for PackagesObject Class and Overriding its Methods: equals(), toString(), finalize(), hashCode()

Basics of Inheritance

- Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in java by which one class is allowed to inherit the features(fields/states and methods/behaviors) of another class.
- Meaning of **Code Reusability**:
  - Code reusability refers to the concept where you can easily reuse what you have already developed to the newer class and add more features to it.

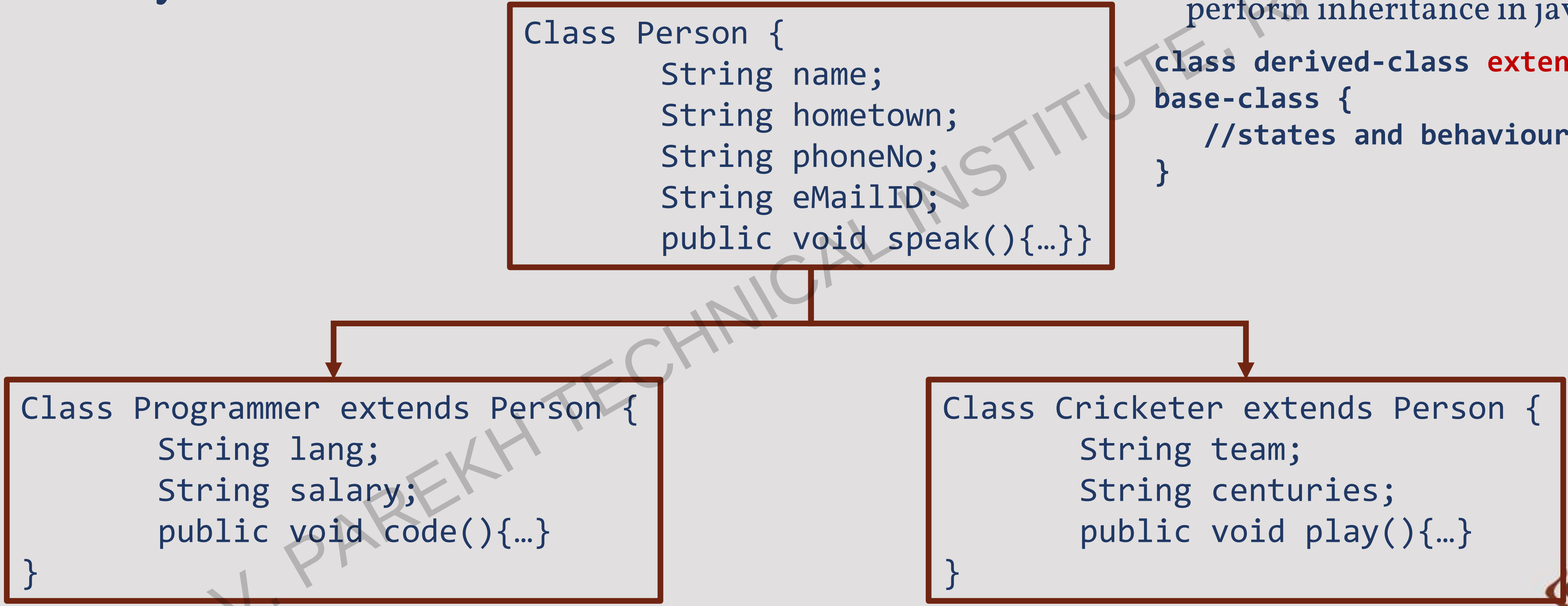


```
Class Programmer {
        String name;
        String hometown;
        String phoneNo;
        String eMailID;
        String lang;
        String salary;
        public void speak(){…}
        public void code(){…}
}
```

```
Class Cricketer {
        String name;
        String hometown;
        String phoneNo;
        String eMailID;
        String team;
        String centuries;
        public void speak(){…}
        public void play(){…}
}
```

Basics of Inheritance

- Instead you can do:



```
Class Person {
        String name;
        String hometown;
        String phoneNo;
        String eMailID;
        public void speak(){…}}
```

```
Class Programmer extends Person {
        String lang;
        String salary;
        public void code(){…}
}
```

```
Class Cricketer extends Person {
        String team;
        String centuries;
        public void play(){…}
}
```

- **extends** keyword is used to perform inheritance in java.

```
class derived-class extends
base-class {
   //states and behaviours
}
```

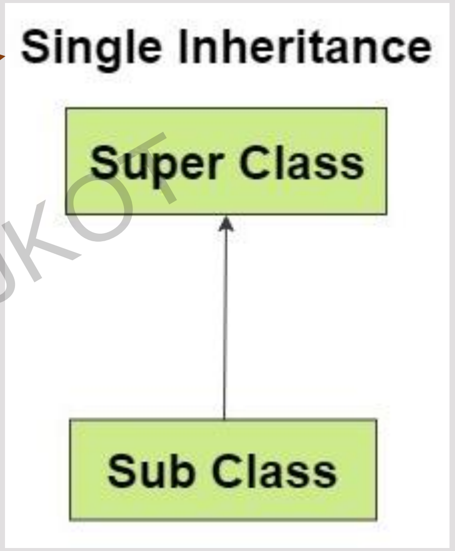
This methods reduces redundancy and improves Code Reusability.

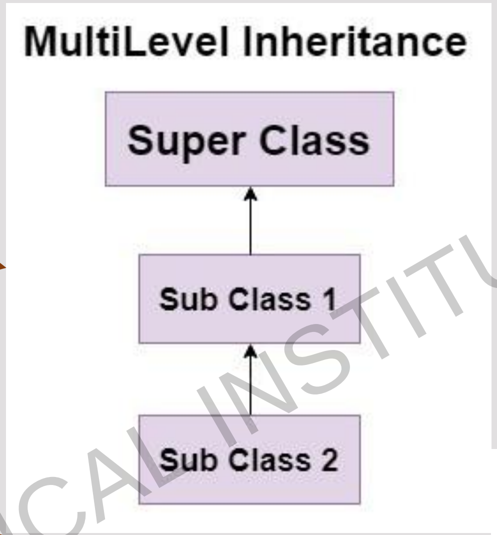
Inheritance: Concept of Super class, Sub class

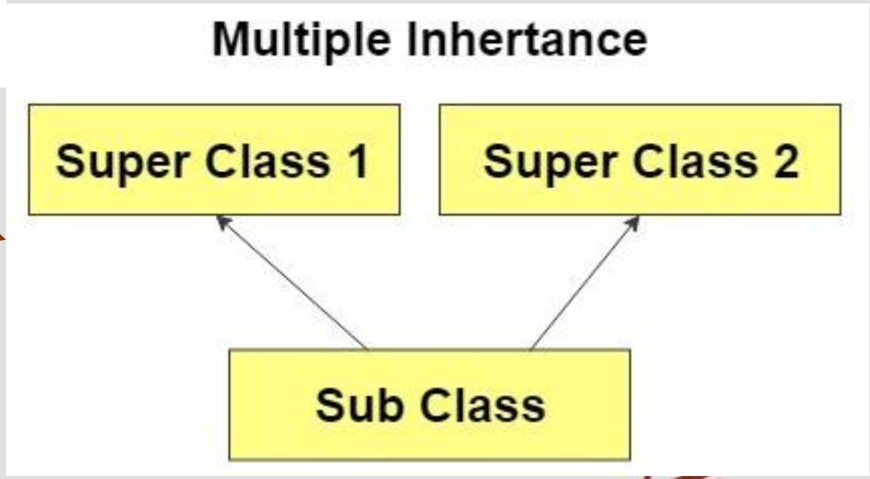
- **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can have its own states and behaviors in addition to the states and behaviors inherited from the parent class.
- We cannot access private members of a class through inheritance.
- If any class declared as a Final then it can not be inherited in sub class.
- Method Overriding only possible through inheritance.

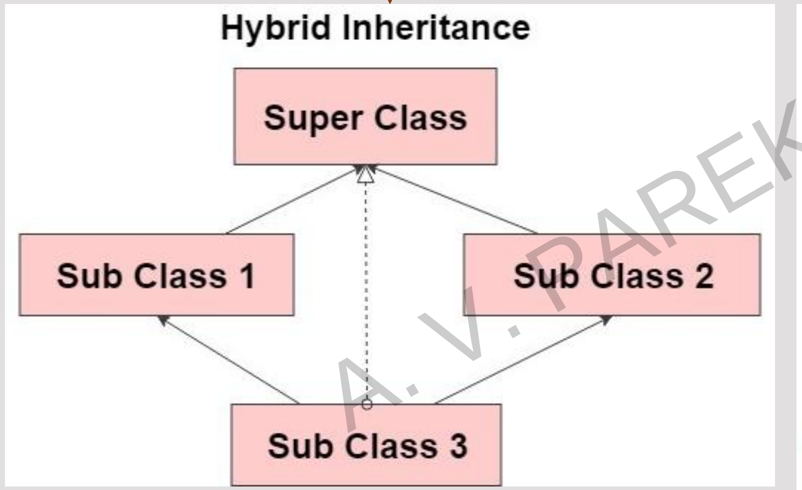
Types of Inheritance

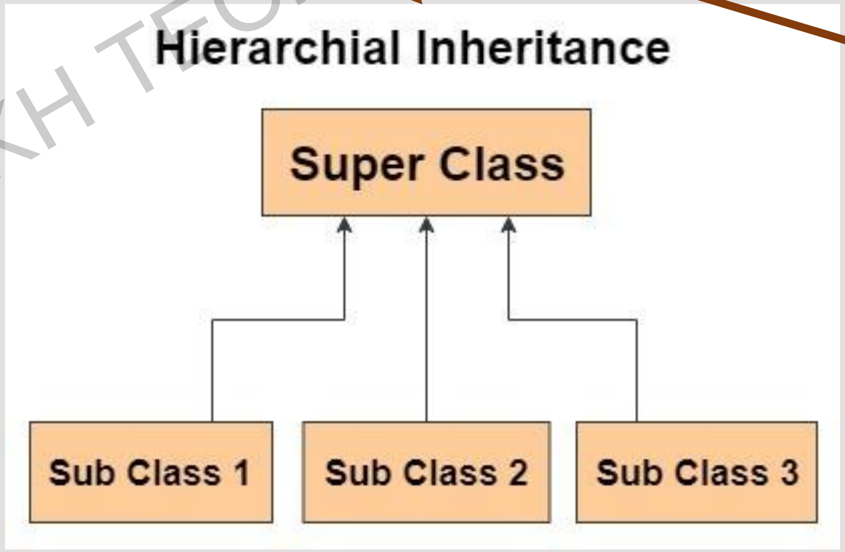
- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance





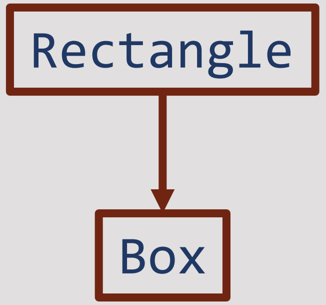






Single Inheritance

```
class Rectangle{
      float width = 5, length = 3;
}
class Box extends Rectangle {
      float height = 2;

      public static void main(String args[]){
            Box b = new Box();
            float area = b.width * b.length;
            float volume = area * b.height;
            System.out.println("Area of Rectangle:" + area);
            System.out.println("Volume of Box:" + volume);
      }
}
```

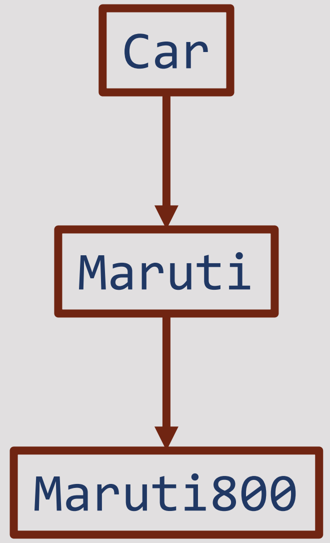


Output

```
Area of Rectangle: 15
Volume of Box: 30
```

Multilevel Inheritance

```
class Car{
   public Car()
   { System.out.println("Class Car");}
   public void vehicleType()
   { System.out.println("Vehicle Type: Car"); }
}
class Maruti extends Car{
   public Maruti()
   { System.out.println("Class Maruti"); }
   public void brand()
   { System.out.println("Brand: Maruti"); }
}
```



Multilevel Inheritance

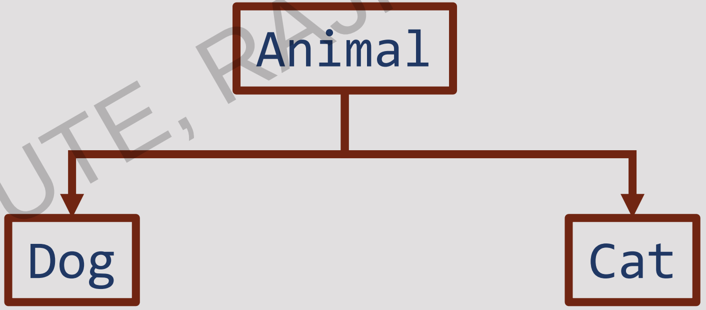
```
public class Maruti800 extends Maruti{
   public Maruti800()
   { System.out.println("Maruti Model: 800"); }
   public void speed()
   { System.out.println("Max: 80Kmph"); }

   public static void main(String args[])
   {
       Maruti800 obj = new Maruti800();
       obj.vehicleType();
       obj.brand(); obj.speed();
   }
}
```

Output

```
Class Car
Class Maruti
Maruti Model: 800
Vehicle Type: Car
Brand: Maruti
Max: 80Kmph
```

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(); //Error
        }
}
```

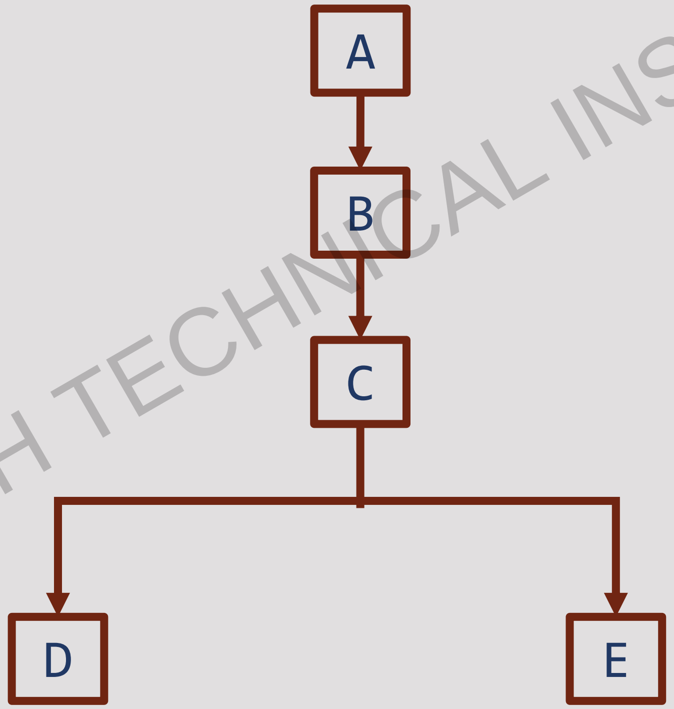


Output

```
Meowing
Eating
```

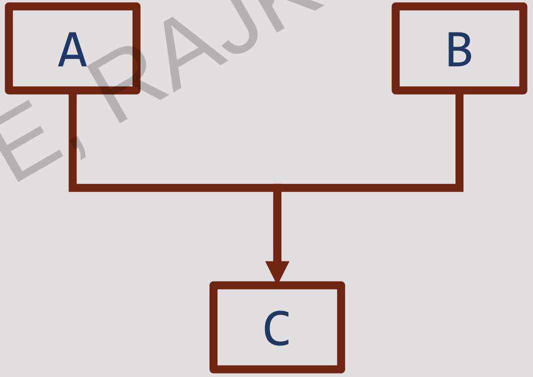
Hybrid Inheritance

- Inheritance type in java where classes are arranged in a way where more than one type of inheritances are combined to get the final structure.



Multiple Inheritance

- To reduce ambiguity and Complexity this inheritance is **"NOT"** supported in Java, HOWEVER it can be done using Interfaces. We will learn that later.



- Example of Ambiguity:

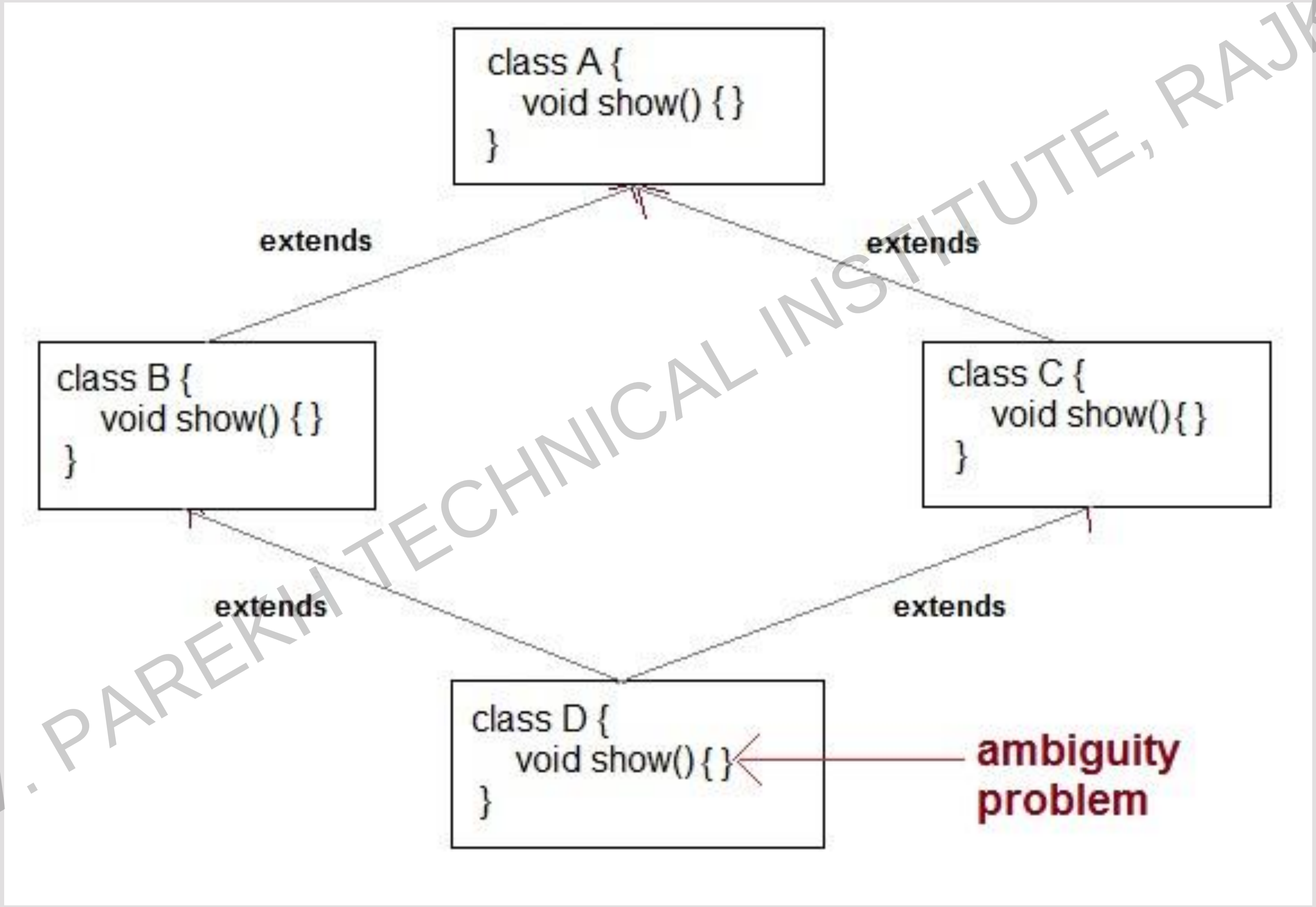
```
class A{
	void msg(){System.out.println("Hello");}
}
class B{
	void msg(){System.out.println("Welcome");}
}
class C extends A, B { //suppose if it were  }
```

```
public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

Output

```
Compile Time Error
```

Diamond Problem



super Keyword

- The super keyword is similar to this keyword.
- It is used to differentiate the members of superclass from the members of subclass, if they have same names.
- It is used to invoke the superclass constructor from subclass.
- **Differentiating the Members of super class and sub class if they have the same name. (Only used if Inheritance is done.)**
- If a class is inheriting the properties of another class and if the members of the superclass have the names same as the sub class, to differentiate these members we use super keyword as shown below.
    - `super.variable // to refer to the super class variable`
    - `super.method(); // to invoke super class method.`
    - `super(args if any); // to invoke super class constructor`
- These are the 3 ways in which super keyword can be used.

super keyword: Variable

```
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 TestSuper1{
        public static void main (String args[]){
                Dog d=new Dog();
                d.printColor();
        }
}
```

Output

```
black
white
```

super keyword: Method

```
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 TestSuper2{
      public static void main(String args[]){
            Dog d=new Dog();
            d.work();
}  }
```

Output

```
eating...
barking...
```

super keyword: Constructor

```
class Animal{
      Animal() {System.out.println("animal is created");}
}
class Dog extends Animal{
      Dog(){
            super();
            System.out.println("dog is created");
      }
}
class TestSuper3{
      public static void main(String args[]){
            Dog d=new Dog();
      }
}
```

Output

```
animal is created
dog is created
```

- super() is added in each class constructor automatically by compiler if there is no super() or this().
- It must be added as 1st non comment line of the constructor.

super keyword

- You **cannot use this() and super() both** in a constructor as the 1st non comment line of the constructor.
- We cannot make use this() in all the constructors, we need at least one constructor with super().

Method Overriding

- Concept where the method with same name and signature (return type & arguments: count and data types) are written in 2 or more inherited classes.
- Actually we are providing a new implementation for that particular method in the sub class which then hides the older implementation of that method.
- Method overriding is used for runtime polymorphism.
- Rules for Method Overriding
  - The method must have the same name as in the parent class.
  - The method must have the same parameter as in the parent class.
  - There must be an IS-A relationship (inheritance).

Method Overriding

```
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 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[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output

```
Bike is running safely
```

If we want to access both method using a single object then?????

ANS: DYNAMIC METHOD DISPATCH

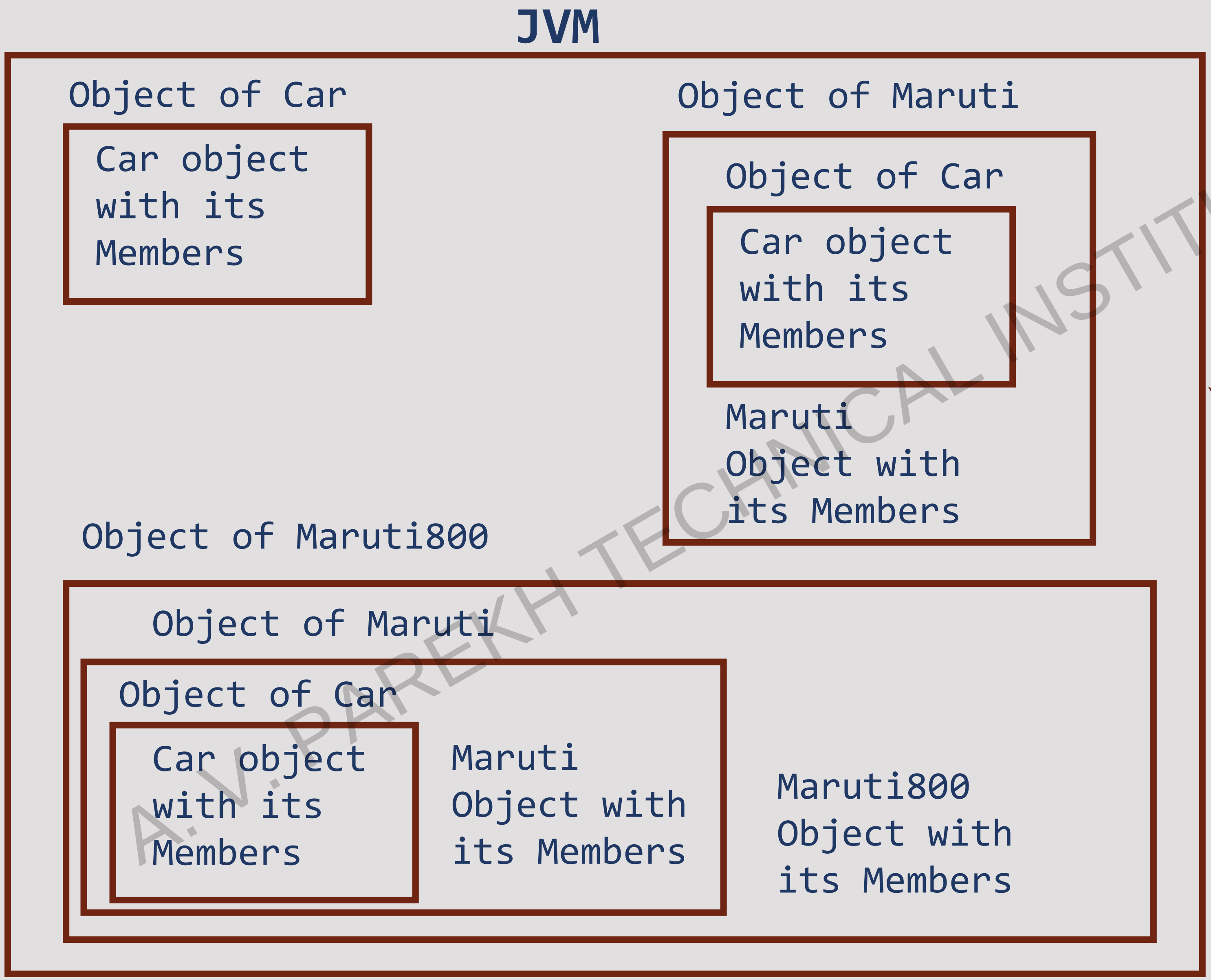
Method Overriding

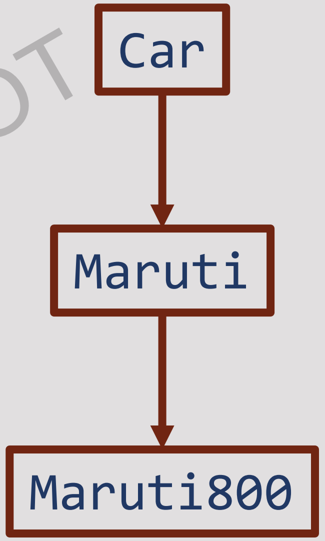
- We cannot override static methods because they are bound with the class not objects/instances. So runtime binding is not possible because they are not allocated in the runtime heap area.
- Hence we cannot override the main() method in java.

Method Overloading v/s Method Overriding

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.
Method overloading occurs when two or more methods in the same class have the same name but different parameters.	Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.
Private and final method can be overloaded.	Private and final method can't be overridden.

Understanding Objects with Inheritance


JVM
Object of Car
Car object with its Members
Object of Maruti
Object of Car
Car object with its Members
Maruti Object with its Members
Object of Maruti800
Object of Maruti
Object of Car
Car object with its Members
Maruti Object with its Members
Maruti800 Object with its Members



Here you can see that each object instantiates not only their members but also all of its superclass's members

Dynamic Method Dispatch

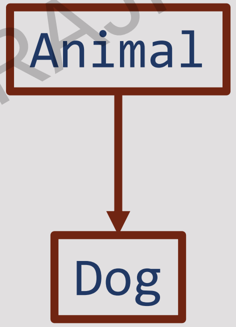
- Dynamic method dispatch is a mechanism in Java where the behavior of the method to be executed is determined at runtime rather than at compile time, This allows for polymorphic behavior, where different subclasses of a class can provide their own implementation of a method, It is also known as runtime polymorphism and it is achieved through method overriding.
- When an overridden method is called, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
  - Compile Time: This is the stage where the Java compiler translates your Java source code into bytecode.
  - Runtime: This is the stage where the bytecode generated by the compiler is executed by the JVM.

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.
- Dynamic method dispatch is a mechanism in Java where the behavior of the method to be executed is determined at runtime rather than at compile time, This allows for polymorphic behavior, where different subclasses of a class can provide their own implementation of a method, It is also known as runtime polymorphism and it is achieved through method overriding.
- Late Binding (which version of that method to execute based upon the type of the object being referred to at the time the call occurs) can be done using this concept.
- Upcasting is needed to be done for this to occur.
  - Upcasting is a scenario where a super class object is assigned the reference of the sub class.

Dynamic Method Dispatch

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

class Dog extends Animal {
   public void move() {
      System.out.println("Dogs can walk and run");
   }
   public void move1() { System.out.println("Another Method");}
}
```



Dynamic Method Dispatch

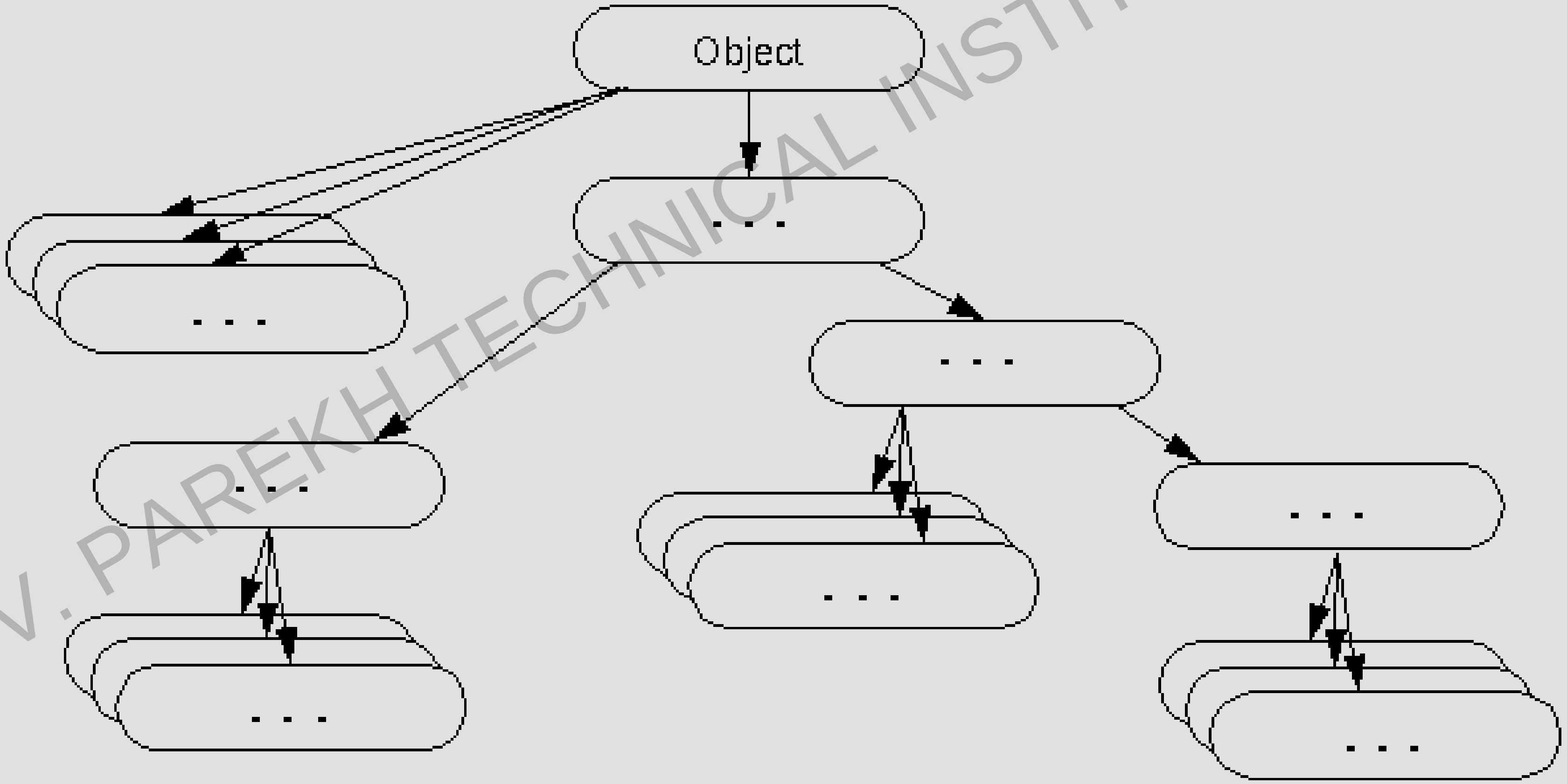
```
public class TestDog {
   public static void main(String args[]) {
      Animal a = new Animal(); // Animal reference and object
      Animal b = new Dog(); // Animal reference but Dog object (DMD)
      Dog c = new Dog(); // Dog reference and object
      // Dog d = new Animal(); // Dog reference but animal object
      // The above line generates an error
      // Sub class reference cannot hold a super class object
      a.move(); // runs the method in Animal class
      b.move(); // runs the method in Dog class
      c.move(); // runs the method in Dog class
     //b.move1(); //Cannot Access new defined methods
}   }
```

Output

```
Animals can move
Dogs can walk and run
Dogs can walk and run
```

Object Class

- The Object class is the parent class of all the classes in java by default.
- i.e. Topmost class in the class hierarchy.



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 notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout, int nanos) throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds.
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	**is invoked by the garbage collector before object is being garbage collected.**

Abstract Class

- As per dictionary, abstraction is the quality of dealing with ideas rather than events.
- Similarly Abstract class in a java is a class which might contain one or more method which are not well defined. i.e. only declaration.

Abstract classes may or may not contain *abstract methods*, i.e., methods without body

(ex.: public void get();)

But, if a class has at least one abstract method, then the class **must** be declared abstract. For that abstract keyword is used

If a class is declared abstract, **it cannot be instantiated.**

To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it. i.e. Abstract class is never at the end (leaf) of inheritance hierarchy.

If you inherit an abstract class, you have to provide implementations to all the abstract methods in it or declare that class also as an abstract. Otherwise that class won't compile

Abstract Class

```
public abstract class Animal {
      public int color;
      public int getColor() { return color; }
      public abstract void speak();
}
public class Cat extends Animal {
      public void speak() { System.out.println("Meow"); }
}
public void Dog extends Animal {
       public void speak() { System.out.println("Bow Wow");
}
```

Must Override this method in this class or make these class abstract.

Interfaces

- A fully abstract class is known as an interface in Java. i.e. if all the methods are needed to be undefined in a class then we can make that class as an interface.
- However Interface can have some default or static methods which can be implemented in the interface itself. **Why???**
- All the data members declared inside an interface are implicitly given as public, static and final, writing explicit private will generate an error. **Why???**
- For that **interface keyword is used instead of class**.
- Same as abstract class there are methods which are undefined so the interfaces are never at the leaf of inheritance hierarchy. Also Interfaces cannot be instantiated.
- Inherited class must override all the methods declared in the interface or declare that class as an abstract, otherwise compilation error will be generated.

Interfaces

You cannot instantiate an interface.

An interface does not contain any constructors.

All of the methods in an interface are abstract.

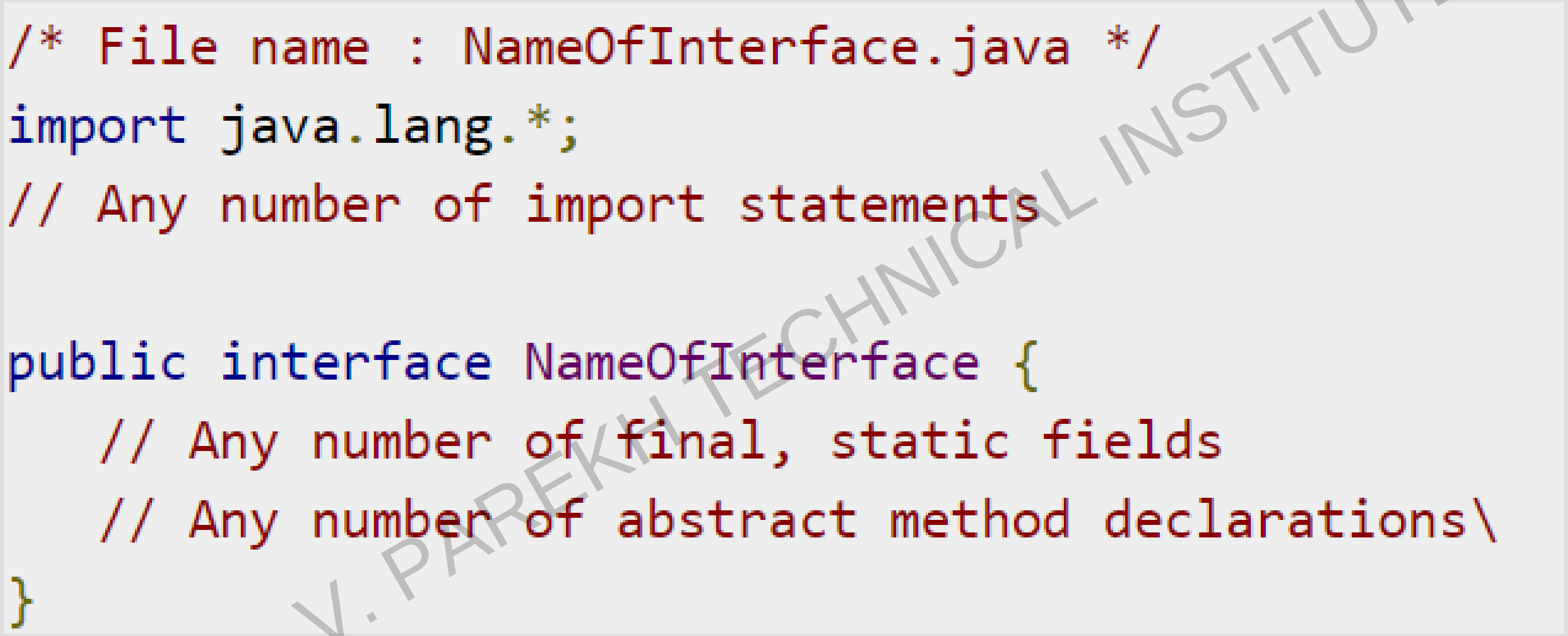
An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.

An interface is not extended by a class; it is **implemented** by a class.

An interface can **extend** multiple interfaces.

Interface

- The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface



```
/* File name : NameOfInterface.java */
import java.lang.*;
// Any number of import statements

public interface NameOfInterface {
   // Any number of final, static fields
   // Any number of abstract method declarations\
}
```

Interface

- An interface is implicitly abstract, so, the abstract keyword is not needed.
- Methods in an interface are implicitly public.

```
//File Name: Animal.java
public interface Animal {
    public void eat();
    public void travel();
}
```

Interface

- A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

```
public class Mammal implements Animal {
    public void eat() { System.out.println("Mammal eats"); }
    public void travel() { System.out.println("Mammal travels"); }
    public int noOfLegs() {
       return 0;
    }

   public static void main(String args[]) {
      Mammal m = new Mammal();
      m.eat();
      m.travel();
   }  }
```

```
OUTPUT
Mammal eats
Mammal travels
```

Interface

- Rules for **Overriding Methods**:
  - New Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method
  - The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
  - An implementation class itself can be abstract and if so, interface methods need not be implemented.

Interface

- A class can implement more than one interfaces at a time. i.e. Multiple inheritance can be done using Interfaces.
- Either a class and one or more interfaces can be inherited or only one or more interfaces can be inherited from a class.
- But in any case only ONE class can be inherited that's the thumb rule to remember. That is not allowed to avoid diamond problem.
- An interface can inherit one or more interfaces using **extends** keyword.
- **Special Case:** An interface can have static methods that are IMPLEMENTED inside an INTERFACE . Also from Java 8 a new feature is added that java can provide DEFAULT IMPLEMENTATION of some methods inside an interface using **default** keyword that are not necessary to be overridden in implementing class. **Why???**

Interface: Example

```
File: ShapeDrawable.java
public interface ShapeDrawable {
     public void draw();
}
File: Rectangle.java
public class Rectangle implements ShapeDrawable {
     public void draw() { System.out.println("Drawing Rectangle");
}
File: Circle.java
public class Circle implements ShapeDrawable {
      public void draw() { System.out.println("Drawing Circle");
}
```

Interface: Example

```
File: ShapeMain.java
public class ShapeMain {
     public static void main() {
          Rectangle r = new Rectangle();
          Circle c = new Circle();
          r.draw();
          c.draw();
     //Applying Dynamic Method Dispatch (Runtime Binding)
          ShapeDrawable s = new Rectangle();
          s.draw();
          s = new Circle();
          s.draw();
     }
}
```

OUTPUT

```
Drawing Rectangle
Drawing Circle
Drawing Rectangle
Drawing Circle
```

Note: In Overriding, Only behaviors are overridden, not states.

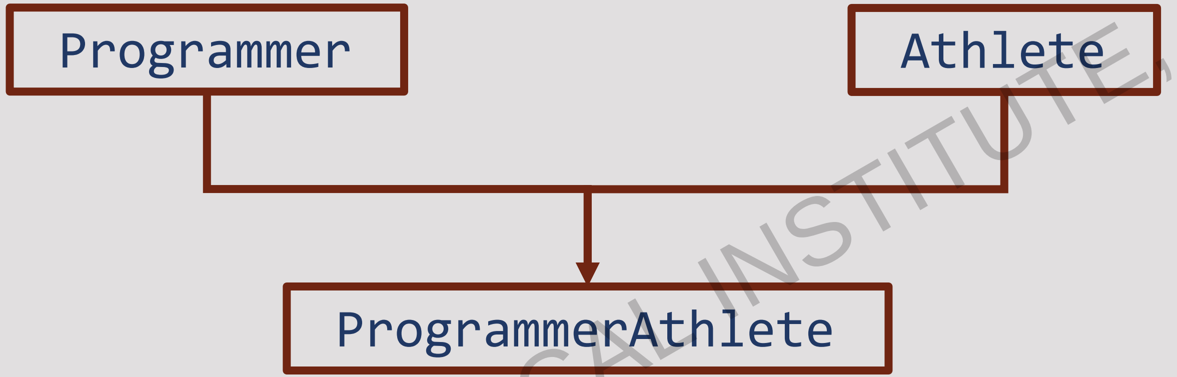
Interface: Multiple Inheritance

```
interface Programmer {
        public void code();
}
```

```
interface Athlete {
        public void play();
}
```

```
public class ProgrammerAthlete implements Programmer, Athlete {
        public void code() { System.out.println("Coding in Java..."); }
        public void play() { System.out.println("Playing Tennis..."); }
        public static void main(String[] args) {
                ProgrammerAthlete rohit = new ProgrammerAthlete();
                rohit.code();
                rohit.play();
} }
```

Interface: Multiple Inheritance

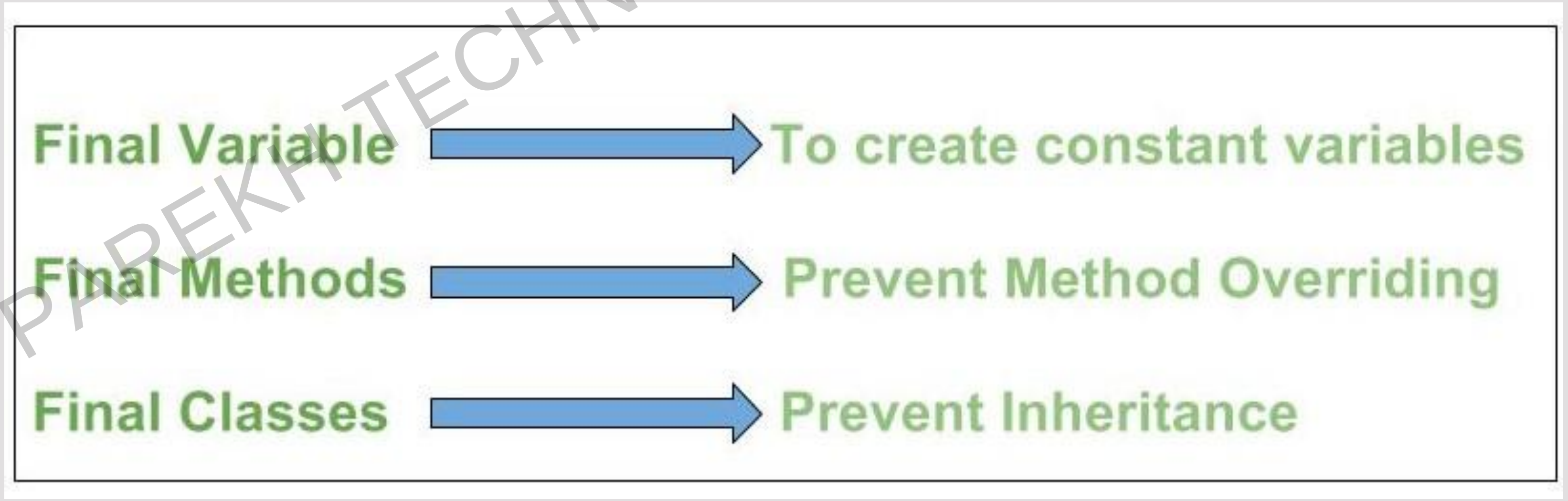


OUTPUT

```
Coding in Java...
Playing Tennis...
```

final keyword revisited

- final keyword can be used with
  - A Variable/Object: makes it a constant.
  - A Method: Makes the current declaration of a method as final and it cannot be overridden.
  - A Class: Makes the current declaration of the class as final and cannot be extended/inherited further.



final keyword revisited

```
class Super {
      public void methodA() {
            System.out.println("Method A from Class Super");
      }
      public final void methodB() {
            System.out.println("Method B from Class Super");
      }
}
class Sub extends Super {
      public void methodA() {
            System.out.println("Method A from Class Sub");
      }
      public void methodB() {} //Cannot be done, It will generate a Compiler Error
}
```

final class

- If a class is declared as final then we cannot inherit that class in any other class. i.e. That class will have the final implementation that cannot be changed further.
- final class always stays at the bottom of the inheritance class hierarchy.

```
final class Bike{}
class Honda1 extends Bike{
  void run(){
        System.out.println("running safely with 100kmph");
  }
  public static void main(String args[]){
        Honda1 honda= new Honda1();
        honda.run();
  }
}
```

This will generate a Compile time error because you cannot inherit a final class

Packages

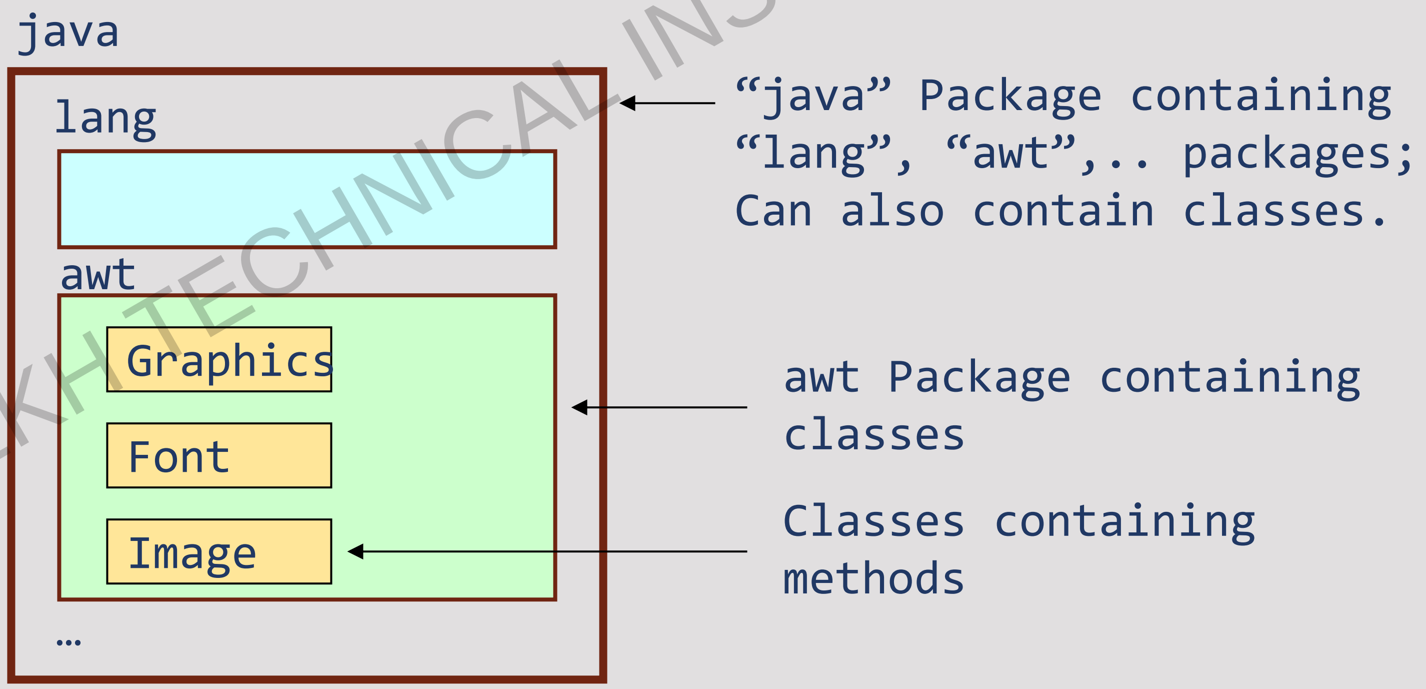
- If we have created a library of classes that supports some particular kind of tasks. Those classes and/or interfaces can be grouped under a single name known as a package in java.
- Packages are Java’s way of grouping a number of related classes and/or interfaces together into a single unit. That means, packages act as “containers” for classes.
- There are many built-in packages available in java library containing some built-in classes.
  - Examples:
    - java.lang package: Contains classes for primitive types, strings, math functions, threads, and exception.
    - java.util package: Contains classes such as vectors, hash tables, date etc.
    - java.awt package: Classes for implementing GUI – windows, buttons, menus etc.
    - java.io package: Stream classes for I/O
    - etc.
- We can also create our own package and add classes into them.

Packages

- The benefits of organising classes into packages are:
  - The classes contained in the packages of other programs/applications can be reused.
  - In packages, classes can be unique compared with classes in other packages. That two classes in two different packages can have the same name. If there is a naming clash, then classes can be accessed with their fully qualified name.
  - Classes in packages can be hidden if we don’t want other packages to access them.
  - Categorization of classes
  - Packages also provide a way for separating “design” from coding.

Packages

- The packages are organized in a hierarchical structure. For example, a package named “java” contains the package “awt”, which in turn contains various classes required for implementing GUI (graphical user interface).


java
lang
awt
Graphics
Font
Image
…
“java” Package containing “lang”, “awt”,.. packages; Can also contain classes.
awt Package containing classes
Classes containing methods

Packages: Import

- There are two ways of accessing the classes stored in packages:
  - Using fully qualified class name
    - `java.lang.Math.sqrt(x);`
  - Import package and use class name directly.
    - `import java.lang.Math`
    - `Math.sqrt(x);`
- Selected or all classes in packages can be imported:

```
import package.class;
import package.*;
```

- Implicit in all programs: import java.lang.*;
- ‘*’ sign will only import the classes in the current package, subpackages will not be imported in it.

Packages: Creation

- Java supports a keyword called "package" for creating user-defined packages. The package statement must be the first non-comment statement in a Java source followed by one or more classes.

```
package myPackage;
public class ClassA {
    // class body
}
class ClassB {
  // class body
}
```

- Here package name is "myPackage" and classes are considred as part of this package; The code is saved in a file called "ClassA.java" and it must be located in a directory called "myPackage".
- Compilatin can be done simply like any class
- To Execute this class (if void main is present in ClassA) then we need to go in the directory containing myPackage and the execute:
  - `java ../myPackage.ClassA`

Packages: Sub-packages

- Classes in one or more source files can be part of the same packages.
- As packages in Java are organised hierarchically, sub-packages can be created as follows:

```
package myPackage.Math
package myPackage.secondPakage.thirdPackage
```

- Store "thirdPackage" in a subdirectory named "myPackage\secondPackage". Store "secondPackage" and "Math" class in a subdirectory "myPackage".

Packages: Example

- Let us store the code listing below in a file named “ClassA.java” within subdirectory named “myPackage” within the current directory (say “abc”).

```
package myPackage;
public class ClassA {
 // class body
 public void display()
 {
     System.out.println("Hello, I am ClassA");
 }
}
class ClassB {
 // class body
}
```

Packages: Example

- Within the current directory "abc" store the following code in a file named "ClassX.java"

```
import myPackage.ClassA;

public class ClassX
{
        public static void main(String args[])
        {
                ClassA objA = new ClassA();
                objA.display();
        }
}
```

Packages: Example

- When ClassX.java is compiled, the compiler compiles it and places .class file in current directory. If .class of ClassA in subdirectory “myPackage” is not found, it compiles ClassA also.
- *Note: It does not include code of ClassA into ClassX*
- When the program ClassX is run, java loader looks for ClassA.class file in a package called “myPackage” and loads it.

Packages: Access Rules Revisited

Access Modifier	Within Class	Within Package	Outside package by Subclass Only	Outside Package
private	Y	N	N	N
default	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y