UNIT-4

INHERITANCE, PACKAGES AND INTERFACES

4.1 BASIC OF INHERITANCE

inheritance is a fundamental feature of object-oriented programming that allows you to create new classes based on existing classes. The class that is being extended is called the superclass or parent class, and the class that inherits from the superclass is called the subclass or child class. Inheritance promotes code reuse and supports the concept of "is-a" relationship between classes.

To define a subclass and establish inheritance, you use the **extends** keyword in the class declaration. The subclass inherits all the fields and methods (except for private members) of the superclass.

Here's an example to illustrate the basics of inheritance in Java:

```
// Superclass
class Animal {
  protected String name;

public Animal(String name) {
 this.name = name;
 }

   public void eat() {
System.out.println(name + " is eating.");
   }
}

// Subclass inheriting from Animal
class Dog extends Animal {
public Dog(String name) {
 super(name);  // Calling the superclass constructor
  }

   public void bark() {
   System.out.println(name + " is barking.");
 }
}
```

In this example, the **Animal** class serves as the superclass, and the **Dog** class is the subclass that extends the **Animal** class.

The **Dog** class inherits the name field and the **eat**() method from the **Animal** class. It also defines its own method, **bark(),** which is specific to dogs.

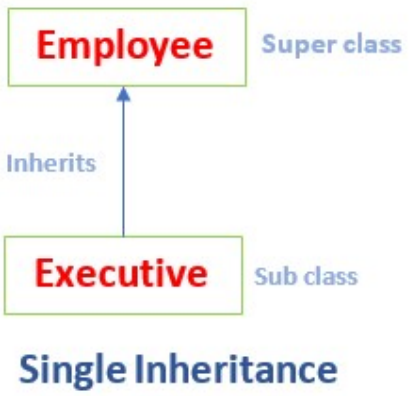
Key points to note about inheritance:

- Subclasses can access non-private members (fields and methods) of the superclass.
- Subclasses can add their own fields and methods or override the behavior of the superclass methods.
- Constructors are not inherited but can be called using the super() keyword to invoke the superclass constructor.
- Java supports single inheritance, which means a class can only extend one superclass. However, it allows for multiple levels of inheritance, where a subclass can itself become a superclass for another subclass.

4.2 TYPES OF INHERITANCE

There are several types of inheritance that you can utilize based on your program's requirements. The different types of inheritance are as follows:

1.Single Inheritance: In single inheritance, a subclass extends a single superclass. It is the most common type of inheritance.

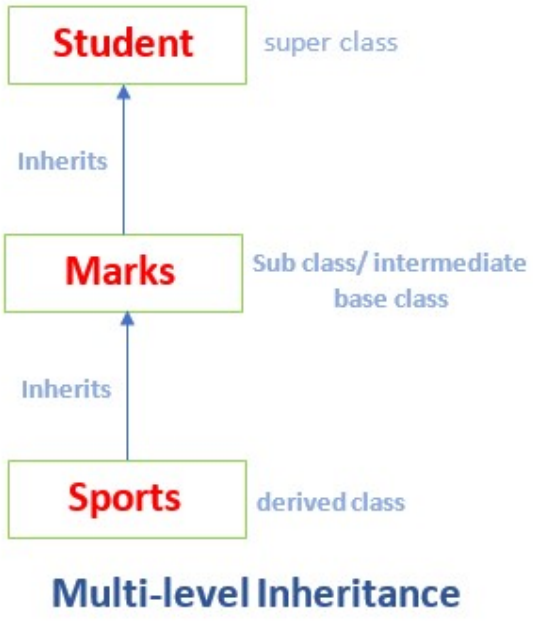


Single Inheritance

```
class Superclass {
 // Superclass members
}

class Subclass extends Superclass {
  // Subclass members
}
```

2.Multilevel Inheritance: In multilevel inheritance, a subclass becomes the superclass for another subclass, creating a chain of inheritance.




```
class Superclass {
  // Superclass members
}

class IntermediateSubclass extends Superclass {
        // Intermediate subclass members
}

class Subclass extends IntermediateSubclass {
  // Subclass members
}
```

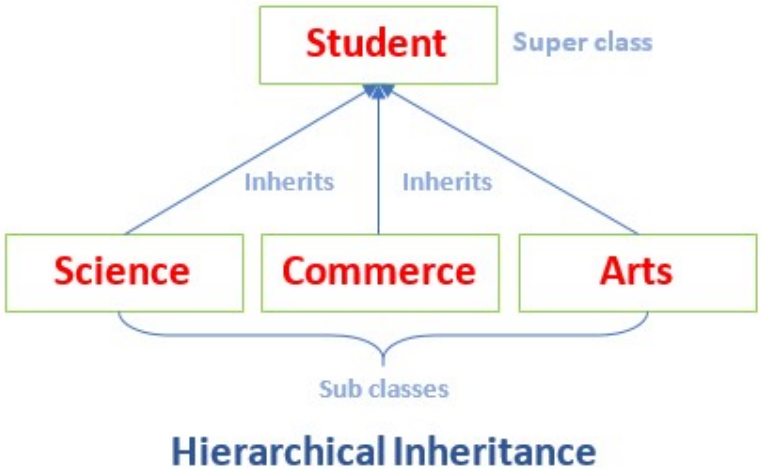
3.Hierarchical Inheritance: In hierarchical inheritance, multiple subclasses extend a single superclass.

```
class Superclass {
// Superclass members
}

class Subclass1 extends Superclass {
        // Subclass1 members
}

class Subclass2 extends Superclass {
```

```
// Subclass2 members
}
```



4.Multiple Inheritance (through Interfaces): Java does not support multiple inheritance of classes, but it allows multiple inheritance of interfaces. A class can implement multiple interfaces, inheriting their method signatures.

```
interface Interface1 {
// Interface1 members
}

interface Interface2 {
  // Interface2 members
}

class MyClass implements Interface1, Interface2 {
 // MyClass members
}
```

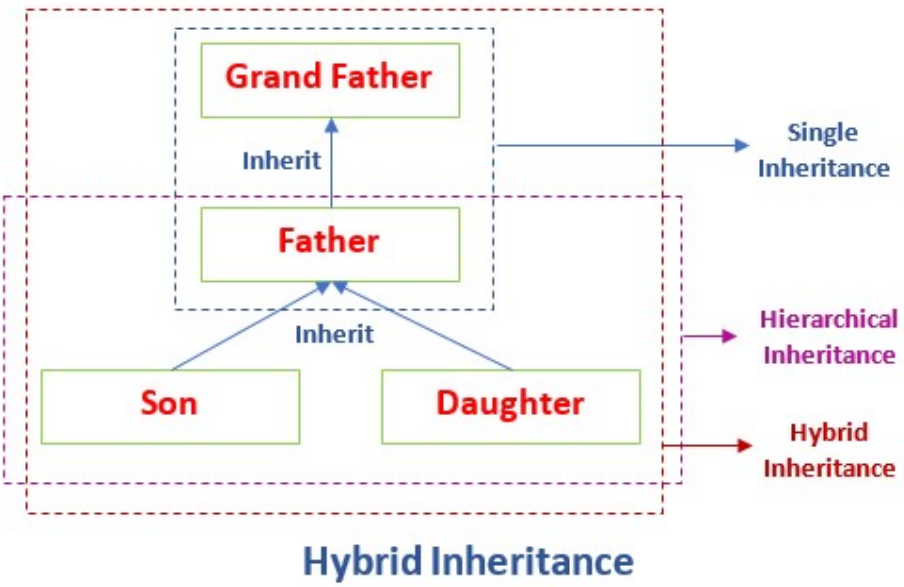
5.Hybrid Inheritance: Hybrid inheritance combines multiple types of inheritance, such as a combination of single inheritance, multilevel inheritance, and multiple inheritance.

```
class Superclass {
  // Superclass members
}

class IntermediateSubclass extends Superclass {
 // Intermediate subclass members
```

```
}

class Subclass extends IntermediateSubclass implements Interface1, Interface2 {
  // Subclass members
}
```



4.3 METHOD OVERRIDING

Method overriding in Java allows a subclass to provide a different implementation of a method that is already defined in its superclass. It is a feature of inheritance that promotes polymorphism, allowing objects of different subclasses to be treated uniformly through their common superclass.

To override a method in a subclass, the following conditions must be met:

- The method in the subclass must have the same name and parameters (method signature) as the method in the superclass.
- The method in the superclass must be marked as public, protected, or have default (package-private) access, to be accessible to the subclass.
- The return type of the method in the subclass must be the same or a subtype (covariant return type) of the return type in the superclass. In Java 5 and later versions, you can override a method with a covariant return type.

Example:

```
class Animal {
  public void makeSound() {
  System.out.println("Animal is making a sound.");
 }
}
class Dog extends Animal {
```

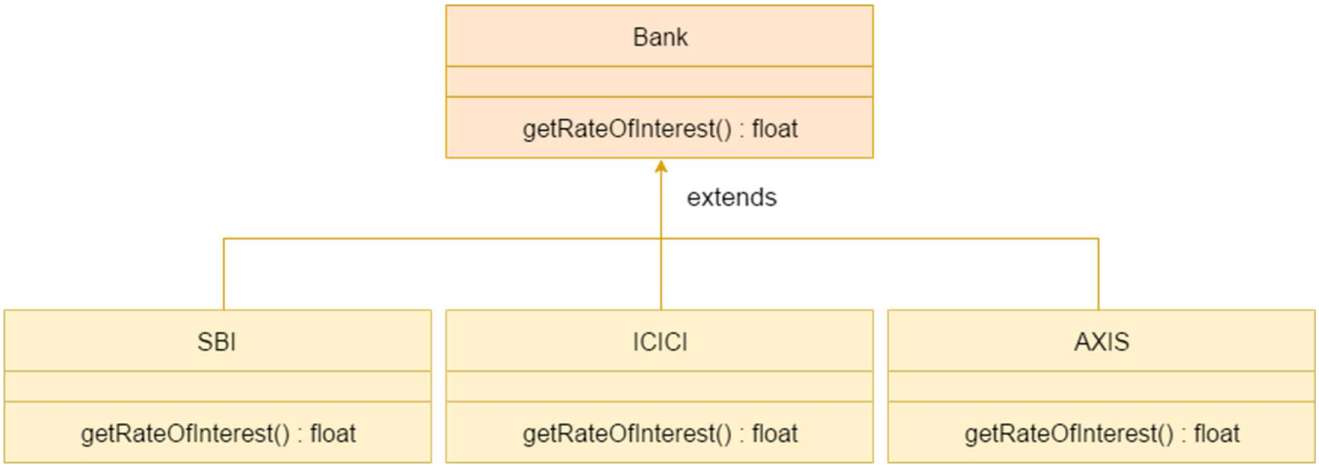
```
@Override
 public void makeSound() {
    System.out.println("Dog is barking.");
 }
}
```

In this example, the **Animal** class has a method called **makeSound**(). The **Dog** class extends the **Animal** class and overrides the **makeSound()** method to provide its own implementation.

When an overridden method is called on an object, the decision on which implementation to invoke is made at runtime based on the actual type of the object (dynamic method dispatch). This allows for polymorphism, where different subclasses can have their own specific implementation of the same method.

Method overriding allows you to provide specialized behavior in subclasses while still adhering to the common interface defined by the superclass. It is a powerful mechanism for achieving code reuse and flexibility in object-oriented programming.

Real Life Example Of Method Overriding:



4.4 SUPER KEYWORD

The **super keyword** is used to refer to the superclass or parent class of a subclass. It is commonly used in two scenarios:

Usage of Java super Keyword

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor

1) Super Is Used To Refer Immediate Parent Class Instance Variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
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

2) Super Can Be Used To Invoke Parent Class Method:

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
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...

3) Super Is Used ToInvoke Parent Class Constructor:

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
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

4.5 DYNAMIC METHOD DISPATCH

Dynamic method dispatch in Java is a mechanism that allows the selection of the appropriate method implementation at runtime, based on the actual type of the object rather than the reference type. It enables polymorphism, where objects of different subclasses can be treated uniformly through their common superclass.

Here's how dynamic method dispatch works in Java:

1.Inheritance Hierarchy: Create a class hierarchy with a common superclass and multiple subclasses that override a method.

```
class Animal {
 public void makeSound() {
```

```
  System.out.println("Animal is making a sound.");
  }
}

class Dog extends Animal {
  @Override
  public void makeSound() {
 System.out.println("Dog is barking.");
 }
}

class Cat extends Animal {
 @Override
 public void makeSound() {
 System.out.println("Cat is meowing.");
}
}
```

2.Object Creation: Create objects of the subclasses and assign them to variables of the superclass type.

```
Animal animal = new Animal();
Animal dog = new Dog();
Animal cat = new Cat();
```

3.Method Invocation: Invoke the overridden method on the objects.

```
animal.makeSound();  // Output: "Animal is making a sound."
dog.makeSound();     // Output: "Dog is barking."
cat.makeSound();     // Output: "Cat is meowing."
```

In this example, we have an **Animal** superclass and two subclasses, **Dog** and **Cat**, that override the **makeSound**() method. We create objects of both subclasses and assign them to variables of the superclass type. When we invoke the **makeSound**() method on these objects, the appropriate overridden method implementation is executed based on the actual type of the object. This is dynamic method dispatch.

At runtime, the JVM determines the actual type of the object being referred to (known as the runtime type), and then the corresponding method implementation is executed. This mechanism allows for treating objects of different subclasses uniformly through their common superclass, promoting code flexibility and polymorphic behavior.

Dynamic method dispatch is a fundamental concept in object-oriented programming and enables the principles of polymorphism and code reuse. It allows for writing more generic code that can be extended and specialized by subclasses while maintaining a consistent interface defined by the superclass.

4.6 FINAL KEYWORD

The final keyword is used to restrict the modification or inheritance of classes, methods, and variables. When applied to different elements, it provides different behaviors:

1.Final Classes: When a class is declared as **final**, it cannot be subclassed. It ensures that the class's implementation cannot be changed by any other class.

```
final class FinalClass {
 // Class members and implementation
}


// Error: Cannot inherit from final class
class Subclass extends FinalClass {
  // Subclass implementation
}
```

2.Final Methods: When a method is declared as **final**, it cannot be overridden by any subclass. This is useful when you want to enforce a specific implementation of a method in a class hierarchy.

```
        class Superclass {
  // Final method
          public final void finalMethod() {
 // Method implementation
  }
}


class Subclass extends Superclass {
  // Error: Cannot override final method
  public void finalMethod() {
    // Method implementation
  }}
```

3.Final Variables: When a variable is declared as **final**, its value cannot be modified once it is assigned. It becomes a constant, and the variable name is written in uppercase by convention.

```
class MyClass {
  // Final variable
public static final int MY_CONSTANT = 10;

  public void myMethod() {
    // Error: Cannot assign a value to final variable
   MY_CONSTANT = 20;
  }
}
```

4.Final Parameters: When a parameter is declared as **final** in a method, its value cannot be changed within the method. It is useful when you want to ensure that the parameter is not modified accidentally.

```
class MyClass {
public void myMethod(final int value) {
   // Error: Cannot assign a value to final parameter
  value = 10;
}
}
```

The **final** keyword provides immutability, security, and performance benefits. By marking a class, method, or variable as **final**, you can ensure that its behavior cannot be modified or overridden, protecting critical functionality or values.

It's important to note that using the **final** keyword should be done judiciously. Overusing it may limit the flexibility and extensibility of your code, so it's recommended to use it when necessary, such as for constants, non-overridable methods, or classes that should not be subclassed.

4.7 ABSTRACT CLASS AND METHOD

An abstract class is a class that cannot be instantiated and is meant to serve as a blueprint for subclasses. It is declared using the abstract keyword. Abstract classes can contain abstract methods, which are declared without an implementation and must be overridden by concrete subclasses.

Here are the key points about abstract classes and methods in Java:

1.Abstract Classes:

- An abstract class is declared using the abstract keyword.
- It may contain abstract methods, concrete methods, instance variables, and constructors.
- Abstract classes cannot be instantiated directly using the new keyword.
- Subclasses of an abstract class must either implement all the abstract methods or be declared as abstract themselves.

- Abstract classes can provide common functionality and define a common interface for their subclasses.

Example:

```
abstract class Shape {
  protected String color;

public Shape(String color) {
this.color = color;
 }

  public abstract double getArea(); // Abstract method

 public void display() {
   System.out.println("Color: " + color);
 }
}
```

2.Abstract Methods:

- An abstract method is declared using the abstract keyword and does not have an implementation.
- Abstract methods are meant to be overridden by concrete subclasses.
- Subclasses must provide an implementation for all the abstract methods inherited from the abstract class.
- Abstract methods do not have a body and end with a semicolon.

Example:

```
abstract class Shape {
 // ...

 public abstract double getArea(); // Abstract method

  // ...
}
```

To Use The Abstract Class And Override The Abstract Method:

```
class Rectangle extends Shape {
 private double length;
private double width;
```

```
  public Rectangle(String color, double length, double width) {
   super(color);
this.length = length;
this.width = width;
 }

    @Override
   public double getArea() {
 return length * width;
 }
}

class Circle extends Shape {
private double radius;

  public Circle(String color, double radius) {
    super(color);
this.radius = radius;
}

 @Override
  public double getArea() {
      return Math.PI * radius * radius;
  }
}
```

In this example, the Shape class is an abstract class that defines an abstract method **getArea**(). The Rectangle and Circle classes extend the Shape class and provide their own implementations for the **getArea**() method.

Abstract classes and methods are useful when you want to provide a common interface and some default behavior for related classes, while still allowing each class to have its specific implementation. They help enforce structure and contracts in your code and facilitate polymorphism.

4.8 INTERFACE

An interface is a reference type that defines a contract of methods that a class implementing the interface must adhere to. It provides a way to achieve abstraction and multiple inheritance of behavior.

1.Declaring Interfaces:

- An interface is declared using the interface keyword.
- It can contain method declarations (without implementations), constant fields, and default methods (with implementations) since Java 8.
- By convention, interface names should be nouns or noun phrases and are typically named using uppercase camel case**.**

```
interface Shape {
    void draw(); // Method declaration

    double getArea(); // Method declaration

    String COLOR = "Red"; // Constant field (implicitly public, static, and final)
}
```

2.Implementing Interfaces:

- A class implements an interface using the **implements** keyword.
- The class must provide implementations for all the methods declared in the interface.
- A class can implement multiple interfaces by separating them with commas.

```
class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a circle.");
    }

    @Override
    public double getArea() {
        return Math.PI * radius * radius;
```

```
    }
}
```

3.Interface Inheritance:

- An interface can extend one or more other interfaces using the extends keyword.
- When an interface extends another interface, it inherits the methods and constant fields of the parent interface.
- A class implementing a child interface must provide implementations for all the methods in the child interface as well as the parent interfaces.

```
interface Drawable {
    void draw();
}

interface Shape extends Drawable {
    double getArea();
}

class Circle implements Shape {
    // ...
}
```

Interfaces are useful for achieving abstraction, defining contracts, and enabling polymorphism. They provide a way to define behavior without specifying the implementation details, allowing for flexibility and code reuse. They are widely used in Java to define APIs and ensure a consistent interface across different classes.

4.9 PACKAGES

Packages are used to organize and group related classes, interfaces, and other resources. They provide a way to create a hierarchical structure and prevent naming conflicts between classes with the same name.

1.Package Declaration:

- A package is declared at the top of a Java source file using the package keyword followed by the package name.
- The package declaration must be the first non-comment line in the source file.
- By convention, package names are written in lowercase and use the reverse domain name convention, such as **com.example.myapp.**

package com.example.myapp;

2.Package Structure:

- Packages can be organized into a hierarchical structure, separated by periods (.).
- The package structure reflects the directory structure in which the corresponding Java files are stored.

- For example, the package **com.example.myapp** would typically correspond to the directory structure **com/example/myapp** on the file system.

3.Package Visibility:

- Classes and members within a package have package-level visibility by default.
- Package-level visibility means that they are accessible only within the same package.
- Classes and members marked as **public** can be accessed from other packages.

4.Importing Packages and Classes:

- To use a class from another package, it must be imported using the **import** statement.
- The **import** statement is placed at the top of the Java source file, below the package declaration and above the class declaration.
- Importing a package allows you to use any class within that package without specifying the package name each time.

import com.example.myapp.MyClass;

5.Standard Java Packages:

- Java provides a set of standard packages that contain classes and APIs for various functionalities, such as **java.util**, java.io, **java.lang**, etc.
- These packages are automatically available for use without requiring an import statement.

Packages play a crucial role in organizing and managing larger Java projects. They help avoid naming conflicts, provide encapsulation, and promote code modularity. By grouping related classes together, packages make it easier to understand and maintain code. Additionally, they facilitate code reuse by enabling the use of classes from other packages through import statements.