Week 7

1. **FUNDAMENTALS OF INHERITANCE**

Inheritance in Java is a concept that acquires the properties from one class to other classes; for example, the relationship between father and son. Inheritance in Java is a process of acquiring all the behaviours of a parent object. The concept of inheritance in Java is that new classes can be constructed on top of older ones. You can use the parent class’s methods and properties when you inherit from an existing class. You can also add additional fields and methods to your existing class. The parent-child relationship, also known as the IS-A relationship, is represented by inheritance.

One object can acquire all of a parent object’s properties and actions through the technique of inheritance in Java Programming. It is a crucial component of OOPs (Object Oriented programming system).

In Java, the idea of inheritance means that new classes can be built on top of existing ones. When you derive from an existing class, you can use its methods and properties. To your current class, you may also add new fields and methods.

**What is inheritance and example?**

A new item can inherit the traits of an older object through the process of inheritance. As an illustration, think of the class “human.” You might want to add other human characteristics in your class, such as height, weight, and so on. Therefore, one approach is to redefine each of those attributes in your class. Though not a good practise, it might be a useful approach to learn object-oriented programming. Inheriting all of those properties from one particular class is the best way to go about it. All of the attributes of class “human” (or “parent”) may be inherited by class “child.” The term “inheritance” in object-oriented programming refers to this.

|  |
| --- |
| Class PetAnimal {  // field and method of the parent class  String name;  public void eat() {  System.out.println("I can eat");  }  }  // inherit from PetAnimal  class Dog extends PetAnimal {  // new method in subclass  public void display() {  System.out.println("My name is " + name);  }  }  class Main {  public static void main(String[] args) {  // create an object of the subclass  Dog labrador = new Dog();  // access field of superclass  labrador.name = "Rohu";  labrador.display();  // call method of superclass  // using object of subclass  labrador.eat();  }  } |

In Java, a class can inherit attributes and methods from another class. The class that inherits the properties is known as the sub-class or the child class. The class from which the properties are inherited is known as the superclass or the parent class.

In Inheritance, the properties of the base class are acquired by the derived classes.

**Syntax:**

|  |
| --- |
| class​ superclass  {  // superclass data variables  // superclass member functions  }  class​ subclass ​extends​ superclass  {  // subclass data variables  // subclass member functions  } |

Inheritance uses the “extends” keyword to create a derived class by reusing the base class code.

**Extends keyword in Java**

The extended keyword extends a class and is an indicator that a class is being inherited by another class. When you say class B extends a class A, it means that class B is inheriting the properties(methods, attributes) from class A. Here, class A is the superclass or parent class and class B is the subclass or child class.

|  |
| --- |
| class Base  {  public void M1()  {  System.out.println(“ Base Class Method ”);  } }  class Derived extends Base  {  public void M2()  {  System.out.printIn(“ Derived Class Methods “);  }  }  class Test  {  public static void main(String[] args)  {  Derived d = new Derived(); // creating object  d.M1(); // print Base Class Method  d.M2(); // print Derived Class Method  } } |

The main advantage of inheritance is code reusability and also method overriding (runtime polymorphism). Inheritance is also known as the IS-A relationship.

Terms:

* Class: ​A class is a collection of objects which have common properties.
* Derived Class/Sub-class: ​Derived class is a class that inherits from a base class. It is also known as subclass or child class.
* Base Class/Superclass: ​The base class is the main class where derived classes inherit the features. It is also known as the superclass or parent class.
* Reusability: ​The name itself says reuse the repeated code in the programs. It is a mechanism to reuse existing code when you are creating new classes.

A diagram of a computer

Description automatically generated

**Fig 1: Inheritance**

1. **TYPES OF INHERITANCE**

The 5 types of inheritance related to Java are given below:

* Single-level inheritance
* Multi-level Inheritance
* Hierarchical Inheritance
* Multiple Inheritance
* Hybrid Inheritance

**Single-level inheritance**

Single-level inheritance in Java is a fundamental concept where a subclass inherits from only one superclass. This means that the subclass can acquire the properties and methods of the superclass, enabling code reuse and the extension of existing functionality. In Inheritance, we can access superclass methods and variables. We can also access subclass methods and variables through subclass objects only. We have to take care of superclass and subclass methods, and variable names shouldn’t conflict.

A diagram of a class

Description automatically generated

**Fig 2: Single inheritance**

**Example:**

|  |
| --- |
| class A  {  int a, b;  void display()  {  System.out.println(“Inside class A values =”+a+” ”+b);  }  }  class B extends A  {  int c;  void show()  {  System.out.println(“Inside Class B values=”+a+” “+b+” “+c); }  }  class SingleInheritance  {  public static void main(String args[])  {  B obj = new B(); //derived class object  obj.a=10;  obj.b=20;  obj.c=30;  obj.display();  obj.show();  }  } |

**Multi-level Inheritance**

Multilevel inheritance in Java is a feature that allows a class to inherit properties and behaviours from another class, which in turn inherits from another class, forming a "chain" of inheritance. This mechanism enables a class to inherit methods and fields from multiple ancestors but in a direct line, where each class in the chain inherits from one class directly above it. As the name implies, numerous base classes are involved in multi-level inheritance. As the newly derived class from the parent class becomes the base class for another newly derived class, the inherited features in multilevel inheritance in Java likewise come from several base classes.

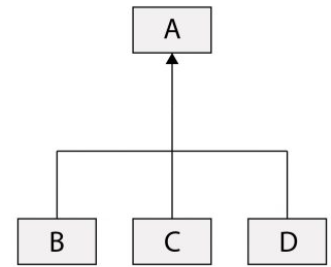
A diagram of a class b

Description automatically generated

**Fig 3: Multi level inheritance**

|  |
| --- |
| class Electronics {  public Electronics(){  System.out.println("Class Electronics");  }  public void deviceType() {  System.out.println("Device Type: Electronics");  }  }  class Grinder extends Electronics {  public Grinder() {  System.out.println("Class Grinder");  }  public void category() {  System.out.println("Category - Grinder");  }  }  class WetGrinder extends Grinder {  public WetGrinder() {  System.out.println("Class WetGrinder");  }  public void grinding\_tech() {  System.out.println("Grinding Technology- WetGrinder");  }  }  public class Tester {  public static void main(String[] arguments) {  WetGrinder wt= new WetGrinder();  wt.deviceType();  wt.category();  wt.grinding\_tech();  } }  **Output:**  Class Electronics  Class Grinder  Class WetGrinder  Device Type: Electronics  Category: Grinder  Grinding Technology: WetGrinder |

**Hierarchical inheritance**

****

**Fig 4: Hierarchical inheritance**

The type of inheritance where many subclasses inherit from one single class is known as Hierarchical Inheritance. Hierarchical Inheritance a combination of more than one type of inheritance. It is different from the multilevel inheritance, as the multiple classes are being derived from one superclass. These newly derived classes inherit the features, methods, etc, from this one superclass. This process facilitates the reusability of a code and dynamic polymorphism (method overriding).

|  |
| --- |
| public class ClassH1  {  public void dispH1()  {  System.out.println("disp() method of ClassH1");  }  }  public class ClassH2 extends ClassH1  {  public void dispH2()  {  System.out.println("disp() method of ClassH2");  }  }  public class ClassH3 extends ClassH1  {  public void dispH3()  {  System.out.println("disp() method of ClassH3");  }  }  public class ClassH4 extends ClassH1  {  public void dispH4()  {  System.out.println("disp() method of ClassH4");  }  }  public class HierarchicalInheritanceTest  {  public static void main(String args[])  {  //Assigning ClassH2 object to ClassH2 reference  ClassH2 h2 = new ClassH2();  //call dispH2() method of ClassH2  h2.dispH2();  //call dispH1() method of ClassH1  h2.dispH1();      //Assigning ClassH3 object to ClassH3 reference  ClassH3 h3 = new ClassH3();  //call dispH3() method of ClassH3  h3.dispH3();  //call dispH1() method of ClassH1  h3.dispH1();    //Assigning ClassH4 object to ClassH4 reference  ClassH4 h4 = new ClassH4();  //call dispH4() method of ClassH4  h4.dispH4();  //call dispH1() method of ClassH1  h4.dispH1();  }  }  **Output:**  disp() method of ClassH2  disp() method of ClassH1  disp() method of ClassH3  disp() method of ClassH1  disp() method of ClassH4  disp() method of ClassH1 |

**Hybrid Inheritance:**

A diagram of a class

Description automatically generated

**Fig 5: Hybrid Inheritance**

Hybrid inheritance is a combination of more than two types of inheritances single and multiple. It can be achieved through interfaces only as multiple inheritance is not supported by Java. It is basically the combination of simple, multiple, hierarchical inheritances.

|  |
| --- |
| class C  {  public void disp()  {  System.out.println("C");  }  }  class A extends C  {  public void disp()  {  System.out.println("A");  }  }  class B extends C  {  public void disp()  {  System.out.println("B");  }    }  class D extends A  {  public void disp()  {  System.out.println("D");  }  public static void main(String args[]){  D obj = new D();  obj.disp();  }  }  **Output:**  **D** |

**Multiple Inheritance**

To reduce the complexity and simplify the language, multiple inheritance is not supported in java. Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class. Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or diferent, there will be compile time error.

1. **SUPER KEYWORD**

The 'super' keyword allows referencing the parent class or superclass of a subclass in Java. It is often employed to access members (fields or methods) of the superclass that have been overridden in the subclass. You can call the superclass's method from within the subclass using super.methodName(). Additionally, super() is used to call the constructor of the superclass from the subclass constructor, which is essential for initializing inherited members.

**Characteristics of Super Keywords in Java**

**Accessing Superclass Members:** One of the main uses of super is to access members (variables or methods) of the parent class from a subclass. This is helpful when a subclass has its version of a method or variable with the same name as the superclass. This is also helpful if you want to use the superclass's version specifically.

**Avoiding Name Conflicts:** Imagine you have a class hierarchy, and both the superclass and subclass have a method with the same name. To avoid confusion and specify which method to call, you can use super followed by the method name to explicitly refer to the superclass's method.

**Calling Superclass Constructors:** When a user creates an object of a subclass, Java automatically calls the constructor of the superclass before the constructor of the subclass is called. In such cases, sometimes the superclass constructor requires parameters, and there, we can use super to pass those parameters and ensure proper initialization.

**Maintaining Inheritance:** Java's super is essential for maintaining the "is-a" relationship between classes in inheritance. It helps to ensure that the subclass inherits the characteristics and behaviors of the superclass. Als helps to promote code reuse and extend functionality.

**Chaining Constructors:** In more complex class hierarchies, constructors can be chained using super(). This means that a subclass constructor can call a constructor in its immediate superclass, which, in turn, can call the constructor of its superclass, and so on, ensuring that all necessary initialization is performed.

**Initialization:** Using super() in a constructor is the first thing that should be done. It initializes the inherited members from the superclass before any subclass-specific initialization.

**Helps Prevent Errors:** Super can help avoid errors that might arise if you inadvertently override a superclass member without realizing it. By explicitly using super, you clarify your intention to use the superclass's version.

**Use of Super Keywords in Java:**

**Use of Super With Variables:** In Java, sometimes the subclass can have a variable with the same name. The super keyword can be employed to access superclass variables. For example, if both the superclass and subclass have a variable called count, you can differentiate between them using super. Count to refer to the superclass's count variable and. Count refers to the subclass's count variable. This ensures you access the correct variable in cases of naming conflicts.

**Use of Super With Methods:** When a subclass overrides a method from its superclass, you can use the super keyword to call the superclass's version of the method. This is helpful when you want to add functionality to the inherited method rather than completely replacing it. For example, if a superclass has a calculated method and a subclass wants to enhance it, you can use super.calculate() within the subclass's method to ensure that the superclass's logic is executed alongside the subclass's additions

|  |
| --- |
| class Animal {  // overridden method  public void display(){  **Use of Super With Constructors:** In Java, constructors in a subclass automatically call the no-argument constructor of the superclass if not explicitly specified. However, you can use super() in a subclass constructor to call a specific constructor in the superclass. This allows you to pass arguments and initialize the superclass's state. For instance, if the superclass has a parameterized constructor, you can use super(argument) in the subclass constructor to ensure proper initialization of the superclass before initializing the subclass-specific attributes. System.out.println("I am an animal");  }  }  class Dog extends Animal {  // overriding method  @Override  public void display(){  System.out.println("I am a dog");  }  public void printMessage(){  // this calls overriding method  display();  // this calls overridden method  super.display();  }  }  class Main {  public static void main(String[] args) {  Dog dog1 = new Dog();  dog1.printMessage();  }  }  **Output:**  I am a dog  I am an animal |

**Access Attributes of the Superclass**

|  |
| --- |
| class Animal {  protected String type="animal";  }  class Dog extends Animal {  public String type="mammal";  public void printType() {  System.out.println("I am a " + type);  System.out.println("I am an " + super.type);  }  }  class Main {  public static void main(String[] args) {  Dog dog1 = new Dog();  dog1.printType();  }  **}**  **Output:**  I am a mammal  I am an animal |

1. **METHOD OVERRIDING**

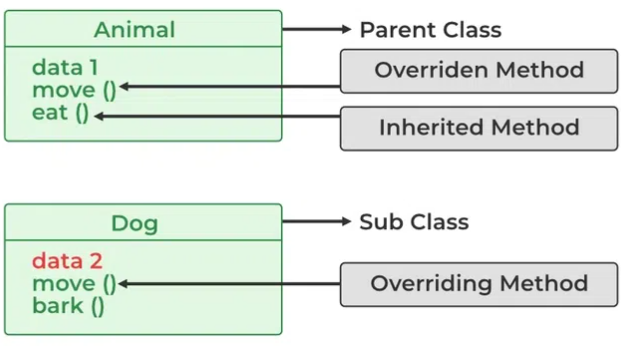
Overriding is when a child class has its method implementation for the method already present in the parent class. Technically, overriding is a function that requires a subclass or child class to provide a variety of method implementations, that are already provided by one of its superclasses or parent classes, in any object-oriented programming language. When a method in a subclass has the same name and signature as in its super-class, the subclass is originated from the super-class. One of the ways that Java manages Run Time Polymorphism is by method overriding.

The object that is used to trigger a method specifies the variant of the process that is executed. If it implements a method with an object from a parent class, the parent class's version will be used. But if the method is triggered with an object from a subclass, the child class's version will be used.

|  |
| --- |
| class Animal {  public void displayInfo() {  System.out.println("I am an animal.");  }  }  class Dog extends Animal {  @Override  public void displayInfo() {  System.out.println("I am a dog.");  }  }  class Main {  public static void main(String[] args) {  Dog d1 = new Dog();  d1.displayInfo();  }  }  **Output:**  I am a dog. |

**Java Overriding Rules**

* Both the superclass and the subclass must have the same method name, the same return type and the same parameter list.
* We cannot override the method declared as final and static.
* We should always override abstract methods of the superclass.

****

**Fig 7: Method overriding**

**Access Specifiers in Method Overriding**

The same method declared in the superclass and its subclasses can have different access specifiers. However, there is a restriction. We can only use those access specifiers in subclasses that provide larger access than the access specifier of the superclass. For example, Suppose, a method myClass() in the superclass is declared protected. Then, the same method myClass() in the subclass can be either public or protected, but not private.

|  |
| --- |
| class Animal {  protected void displayInfo() {  System.out.println("I am an animal.");  }  }  class Dog extends Animal {  public void displayInfo() {  System.out.println("I am a dog.");  }  }  class Main {  public static void main(String[] args) {  Dog d1 = new Dog();  d1.displayInfo();  }  **}**  **Output:**  I am a dog. |

In the above example, the subclass Dog overrides the method displayInfo() of the superclass Animal. Whenever we call displayInfo() using the d1 (object of the subclass), the method inside the subclass is called. Notice that, the displayInfo() is declared protected in the Animal superclass. The same method has the public access specifier in the Dog subclass. This is possible because the public provides larger access than the protected.

1. **DYNAMIC METHOD DISPATCH**

Dynamic method dispatch is the mechanism in which a call to an overridden method is resolved at run time instead of compile time. This is an important concept because of how Java implements run-time polymorphism. Java uses the principle of ‘a superclass reference variable can refer to a subclass object’ to resolve calls to overridden methods at run time. When a superclass reference is used to call an overridden method, Java determines which version of the method to execute based on the type of the object being referred to at the time call. In other words, it is the type of object being referred to that determines which version of an overridden method will be executed.

**Advantages of dynamic method dispatch**

* It allows Java to support overriding of methods, which are important for run-time polymorphism.
* It allows a class to define methods that will be shared by all its derived classes, while also allowing these sub-classes to define their specific implementation of a few or all of those methods.
* It allows subclasses to incorporate their own methods and define their implementation.

|  |
| --- |
| // Implementing Dynamic Method Dispatch  class Apple  {  void display()  {  System.out.println("Inside Apple's display method");  }  }  class Banana extends Apple  {  void display() // overriding display()  {  System.out.println("Inside Banana's display method");  }  }    class Cherry extends Apple  {  void display() // overriding display()  {  System.out.println("Inside Cherry's display method");  }  }  class Fruits\_Dispatch  {  public static void main(String args[])  {  Apple a = new Apple(); // object of Apple  Banana b = new Banana(); // object of Banana  Cherry c = new Cherry(); // object of Cherry    Apple ref; // taking a reference of Apple    ref = a; // r refers to a object in Apple  ref.display(); // calling Apple's version of display()    ref = b; // r refers to a object in Banana  ref.display(); // calling Banana's version of display()    ref = c; // r refers to a object in Cherry  ref.display(); // calling Cherry's version of display()  }  } |

This program creates one superclass (i.e., class Apple) and two subclasses of it (i.e., Banana class and Cherry class). Subclasses Banana and Cherry override the display() method declared in Apple. Inside the main() method in class Fruits\_Dispatch, objects of type Apple, Banana, and Cherry are declared. A reference of type Apple, called ref, is declared. The program then assigns a reference to each type of object to ref and uses the reference to invoke display(). The version of display() executed is determined by the type of the object being referred to at the time of the call.