Unit-03

Inheritance and Interfaces

3.1 Inheritance Basics

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a new class (subclass or derived class) to inherit the properties and behaviors (fields and methods) of an existing class (superclass or base class). Inheritance promotes code reuse, scalability, and the creation of hierarchical class structures.

Inheritance Basics:

1. **Superclass (Parent Class)**: The class whose properties and methods are inherited by another class.
2. **Subclass (Child Class)**: The class that inherits properties and methods from another class.
3. **extends Keyword**: Used to create a subclass that inherits from a superclass.
4. **Method Overriding**: Allows a subclass to provide a specific implementation of a method that is already defined in its superclass.
5. **Access Control**: Controls which members (fields and methods) are accessible to subclasses.

3.2 Inheritance and Constructors

In Java, constructors are special methods used to initialize objects. When dealing with inheritance, understanding how constructors work and how they interact between superclasses and subclasses is essential.

Example of Inheritance and Constructors:

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

    // Superclass constructor
    public Animal(String name) {
        this.name = name;
        System.out.println("Animal constructor called");
```

```
    }
}
// Subclass
class Dog extends Animal {
    String breed;
    public Dog(String name, String breed) {
        super(name); // Call to superclass constructor
        this.breed = breed;
        System.out.println("Dog constructor called");
    }
}

public class InheritanceConstructorExample {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy", "Golden Retriever");
        System.out.println("Name: " + dog.name);
        System.out.println("Breed: " + dog.breed);
    }
}
```

3.3 super keyword

The super keyword in Java is used to refer to the superclass (parent class) of the current object. It can be used in three main contexts: to access superclass methods, to access superclass fields, and to invoke a superclass constructor.

The super keyword can be used to call a constructor of the superclass from within a subclass constructor. This is particularly important for ensuring that the superclass is properly initialized when a subclass object is created.

Example:

```
class Animal {
```

```
    String name;
     public Animal(String name) {
        this.name = name;
        System.out.println("Animal constructor called");
    }
}
class Dog extends Animal {
    String breed;
     public Dog(String name, String breed) {
        super(name); // Call to superclass constructor
        this.breed = breed;
        System.out.println("Dog constructor called");
    }
}
public class SuperKeywordExample {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy", "Golden Retriever");
        System.out.println("Name: " + dog.name);
        System.out.println("Breed: " + dog.breed);
    }
}
```

3.4 Method Overriding

Method overriding is a feature in Java that allows a subclass to provide a specific implementation for a method that is already defined in its superclass. This is a crucial aspect of polymorphism in object-oriented programming, enabling dynamic method dispatch at runtime.

Example of Method Overriding:

```
class Animal {
```

```
    public void makeSound() {
        System.out.println("Some generic animal sound");
    }
}
class Dog extends Animal {
    public void makeSound() {
        System.out.println("Bark");
    }
}
public class MethodOverridingExample {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();

        myAnimal.makeSound();
        myDog.makeSound();
    }
}
```

3.5 Polymorphism

Polymorphism is one of the core concepts of object-oriented programming (OOP) in Java. It allows objects of different classes to be treated as objects of a common superclass. Polymorphism enables a single interface to represent different underlying forms (data types).

- **Compile-Time Polymorphism (Method Overloading)**: Achieved through method overloading. This is when multiple methods in the same class have the same name but different parameters.
- **Runtime Polymorphism (Method Overriding)**: Achieved through method overriding. This allows a subclass to provide a specific implementation of a method that is already defined in its superclass.

3.6 Dynamic Binding

Dynamic binding, also known as late binding or runtime binding, is a concept in Java where the method to be called is determined at runtime rather than compile-time. This allows for more flexible and dynamic code, enabling polymorphism in object-oriented programming.

Example of Dynamic Binding:

```
class Animal {
    public void makeSound() {
        System.out.println("Some generic animal sound");
    }
}
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}
class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
}
public class DynamicBindingExample {
    public static void main(String[] args) {
        Animal myAnimal; // Reference variable of type Animal


        myAnimal = new Dog();
        myAnimal.makeSound();
```

```
        myAnimal = new Cat();
        myAnimal.makeSound();
    }
}
```

3.7 final Keyword

The final keyword in Java is used to restrict the user. It can be applied to variables, methods, and classes, each with a different purpose and effect. Understanding the final keyword helps in creating more predictable and stable code.

When a variable is declared as final, its value cannot be changed once it is initialized. This effectively makes it a constant.

Example:

```
public class FinalVariableExample {
    public static void main(String[] args) {
        final int MAX_VALUE = 100;
        System.out.println("The maximum value is: " + MAX_VALUE);
    }
}
```

3.8 Abstract Classes

Abstract classes in Java are classes that cannot be instantiated directly and are designed to be extended by other classes. They serve as a blueprint for other classes, providing a common interface and sharing code across multiple related classes. To declare an abstract class, use the abstract keyword in the class definition. Abstract classes cannot be instantiated directly. They are meant to be subclassed. Abstract classes can have abstract methods, which are methods declared without an implementation. Subclasses must provide implementations for these methods.

3.9 Access Specifiers

Access specifiers, also known as access modifiers, define the visibility and accessibility of classes, methods, and variables. They control where and how different parts of your code can be accessed and modified. Java has four main access specifiers:

1. **Public: Visibility**: Everywhere, **Usage**: Classes, methods, and variables.
2. **Private: Visibility**: Within the same class only, **Usage**: Methods and variables (cannot be used for classes).
3. **Protected: Visibility**: Within the same package and subclasses, **Usage**: Methods and variables (cannot be used for top-level classes).
4. **Default** (no modifier): **Visibility**: Within the same package, **Usage**: Classes, methods, and variables (when no other specifier is used).

3.10 Interfaces

Interfaces in Java provide a way to achieve abstraction by defining a contract that implementing classes must follow. They allow you to specify methods that a class must implement, without specifying how they should be implemented. Interfaces are fundamental in achieving loose coupling and enabling polymorphism.

Characteristics

1. **Abstract Methods**: Interfaces can contain method signatures without implementations. These methods are implicitly abstract and must be implemented by any class that implements the interface.
2. **Constants**: Interfaces can contain constants, which are public, static, and final by default.
3. **Multiple Inheritance**: Java allows a class to implement multiple interfaces, enabling a form of multiple inheritance of type.

Declaring an Interface

To declare an interface in Java, use the interface keyword followed by the interface name. Here's a simple example:

```
// Interface declaration

interface Animal {

   void makeSound(); // Abstract method (implicitly public and abstract)

   // Constant declaration
```

```
    int MAX_AGE = 100; // public, static, final
}
```

Implementing an Interface

A class implements an interface using the implements keyword, and it must provide implementations for all abstract methods defined in the interface.

```
// Class implementing the Animal interface
class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}
```