**Chapter # 06: Inheritance**

**6.1 Inheritance:**

Inheritance is a fundamental concept in object-oriented programming that enables a new class to be based on an existing class. Inheritance allows the child class to inherit the properties and methods of the parent class, which it can then use, modify, or extend.

The basic idea behind inheritance is that a child class is a more specialized version of its parent class. For example, a car class may have a number of properties and methods, and a sports car class could be a child class of the car class that inherits all of its properties and methods, while also adding new properties and methods that are specific to sports cars.

By using inheritance, programmers can reuse existing code, reduce duplication, and create more organized and modular code structures. It also allows for greater flexibility and extensibility in software development, as new child classes can be created based on existing classes with minimal effort.

**6.2 Purpose of Inheritance:**

The main purpose of inheritance in programming is to promote code reuse and to facilitate the creation of new classes that are similar to existing classes but with some additional or modified functionality. Some of the Main purpose of Inheritance are given below:

1. **Code reuse:**

When a new class is established by deriving from an existing class, all of the parent class's properties and methods are automatically passed down to the child class, allowing the child class to reuse the parent class's functionality without having to write any new code.

1. **Simplification of code**:

Inheritance enables developers to create more simplified and modular code structures, by allowing them to break down complex classes into smaller, more specialized classes that are easier to manage and maintain.

1. **Improved efficiency:**

By minimizing the amount of code that needs to be developed, tested, and reused, inheritance can increase the effectiveness of software development.

1. **Polymorphism**:

Polymorphism, or the ability to treat objects of various classes as though they were of the same class, is one of the core elements of object-oriented programming that inheritance permits. This can simplify coding and increase the maintainability of programs.

Overall, inheritance is a powerful tool that allows developers to build more flexible and efficient software systems by leveraging the strengths of existing code and creating new classes that are better suited to their specific needs.

**6.3 Complex Classes Through Inheritance:**

By establishing new classes that inherit the attributes and methods of existing classes while simultaneously adding new properties and methods that are exclusive to the new class, developers can use inheritance to create more complicated classes. When using inheritance to create more complicated classes, you should do the following steps:

* **Identify the existing class:** First, identify an existing class that contains some of the functionality that you want to include in your new class.
* **Decide on the relationship:** Decide on the relationship between the new class and the existing class. Will the new class be a subtype of the existing class, or will it simply use the existing class as a template?
* **Create the new class:** Create the new class, making sure to include any new properties and methods that are specific to the new class.
* **Inherit from the existing class:** Use the inheritance mechanism to make the new class inherit the properties and methods of the existing class. In most programming languages, this is done using the "extends" keyword.
* **Override and extend methods:** Override any methods in the new class that need to be modified or customized, and extend any methods in the existing class that need to be augmented with additional functionality.
* **Test the new class:** Test the new class thoroughly to ensure that it is working as intended, and that all of the new and inherited properties and methods are functioning correctly.

Developers can utilize inheritance to create more intricate classes that are customized to their own requirements by following these steps while also gaining access to the functionality and effectiveness of pre-existing code.

**6.4 Chaining:**

Programmers can invoke numerous methods on an object in a single line of code by using the chaining approach. When creating a class, this strategy can be especially helpful because it makes the code simpler and more readable. Chaining can assist in creating a class in the following ways:

* Chaining allows developers to write code that is more concise and easier to read, as it eliminates the need to create temporary variables to hold intermediate results.
* By chaining methods together, developers can create code that reads more like natural language, making it easier to understand and maintain.
* Chaining can make it easier to debug code, as it allows developers to pinpoint the exact location of any errors or exceptions that may occur.
* Chaining can also improve the efficiency of code, as it reduces the amount of overhead associated with creating and managing temporary variables.

Here is a Little Example of Chaining in Java

|  |
| --- |
| public class Person {      private String name;      private int age;      private String address;      public Person setName(String name) {          this.name = name;          return this;      }      public Person setAge(int age) {          this.age = age;          return this;      }      public Person setAddress(String address) {          this.address = address;          return this;      }      public String getName() {          return name;      }      public int getAge() {          return age;      }      public String getAddress() {          return address;      }  }  // Usage  Person person = new Person()      .setName("ABC")      .setAge(30)      .setAddress("Street 10, London");  String name = person.getName();  int age = person.getAge();  String address = person.getAddress(); |

**6.5 Dynamic Dispatching:**

The practice of choosing the best method implementation to be executed at runtime based on the actual type of the object the method is being called on is known as dynamic dispatching. Runtime polymorphism or "late binding" are other names for this.

Due to the fact that it enables subclasses to replace the methods of their superclass with alternative implementations, dynamic dispatching in Java is strongly tied to inheritance. The JVM will first search for a method implementation in the subclass when a method is called on an object of a subclass. If it does, it will employ it rather than the superclass implementation. This preserves the functionality of the superclass while enabling subclasses to modify the behavior of their inherited methods. Here is an Example of Dynamic Dispatching

|  |
| --- |
| class Animal {      public void speak() {          System.out.println("I am an animal.");      }  }  class Dog extends Animal {      @Override      public void speak() {          System.out.println("Woof!");      }  }  class Cat extends Animal {      @Override      public void speak() {          System.out.println("Meow!");      }  }  Animal animal1 = new Animal();  animal1.speak(); // prints "I am an animal."  Animal animal2 = new Dog();  animal2.speak(); // prints "Woof!"  Animal animal3 = new Cat();  animal3.speak(); // prints "Meow!" |

**6.6 Abstract and Concrete Classes:**

A class that cannot be instantiated directly but may be used as a foundation class to create other classes is known as an abstract class in object-oriented programming. An abstract class may have one or more abstract methods with no implementation, serving as a template for other classes to derive from. The abstract keyword is used to declare abstract classes.

Here is an Example of Abstract Class in Java

|  |
| --- |
| abstract class Shape {      public abstract double getArea();      public abstract double getPerimeter();      public void print() {          System.out.println("This is a shape.");      }  }  class Rectangle extends Shape {      private double length;      private double width;      public Rectangle(double length, double width) {          this.length = length;          this.width = width;      }      public double getArea() {          return length \* width;      }      public double getPerimeter() {          return 2 \* (length + width);      }  }  Shape shape = new Rectangle(3.0, 4.0);  double area = shape.getArea();  double perimeter = shape.getPerimeter();  shape.print(); // prints "This is a shape." |

The usage of abstract classes to create a common interface for a collection of related classes while enabling each subclass to offer its own implementation of the abstract methods is illustrated by the example given above. Subclasses can inherit from the abstract class's concrete methods, which enables code reuse and reduces duplication.

Contrarily, a concrete class is one that may be created directly and may or may not be descended from an abstract class. Every method in a concrete class, including any inherited methods from its base classes, has an implementation. Here is an Example:

|  |
| --- |
| class Dog {      private String name;      private int age;      public Dog(String name, int age) {          this.name = name;          this.age = age;      }      public void bark() {          System.out.println(name + " is barking.");      }      public void fetch() {          System.out.println(name + " is fetching.");      }      public void sleep() {          System.out.println(name + " is sleeping.");      }  }  Dog dog = new Dog("Fido", 3);  dog.bark(); // prints "Fido is barking."  dog.fetch(); // prints "Fido is fetching."  dog.sleep(); // prints "Fido is sleeping." |

When we have a specific implementation that we wish to use without additional customization, concrete classes come in handy. On the other side, abstract classes come in handy when we wish to specify a common interface for a collection of related classes while yet allowing each subclass to offer its own implementation of the abstract methods.

**6.7 Composition and Inheritance:**

In object-oriented programming, composition and inheritance are two distinct methods for achieving code reuse and creating connections between classes. The primary distinction between them is that while inheritance entails the creation of a subclass that derives properties and behavior from a parent class, composition entails building a class from other objects.

In composition, a class is constructed using other objects, which are typically passed as constructor arguments or instantiated within the class. The composed objects are used to provide specific functionality or behavior to the class. For example, a Car class may be composed of an Engine object, a Transmission object, and a Chassis object, each responsible for a different aspect of the car's behavior. Here is an Example

|  |
| --- |
| public class Car {      private Engine engine;      private Transmission transmission;      private Chassis chassis;      public Car(Engine engine, Transmission transmission, Chassis chassis) {          this.engine = engine;          this.transmission = transmission;          this.chassis = chassis;      }      public void start() {          engine.start();          transmission.shiftToDrive();          chassis.move();      }      public void stop() {          engine.stop();          transmission.shiftToPark();          chassis.stop();      }  }  public class Engine {      public void start() {          System.out.println("Engine started.");      }      public void stop() {          System.out.println("Engine stopped.");      }  }  public class Transmission {      public void shiftToDrive() {          System.out.println("Shifted to drive.");      }      public void shiftToPark() {          System.out.println("Shifted to park.");      }  }  public class Chassis {      public void move() {          System.out.println("Moving.");      }      public void stop() {          System.out.println("Stopped.");      }  }  Engine engine = new Engine();  Transmission transmission = new Transmission();  Chassis chassis = new Chassis();  Car car = new Car(engine, transmission, chassis);  car.start(); // starts the car  car.stop(); // stops the car |

Inheritance, on the other hand, involves creating a subclass that inherits attributes and behavior from a parent class. The subclass can then add or modify the inherited behavior, as well as define its own behavior. For example, a Rectangle class may inherit from a Shape class and add its own behavior to calculate area and perimeter. Here is an Example of it also:

|  |
| --- |
| public class Shape {      private int x;      private int y;      public Shape(int x, int y) {          this.x = x;          this.y = y;      }      public int getX() {          return x;      }      public int getY() {          return y;      }      public void draw() {          System.out.println("Drawing shape at (" + x + ", " + y + ")");      }  }  public class Rectangle extends Shape {      private int width;      private int height;      public Rectangle(int x, int y, int width, int height) {          super(x, y);          this.width = width;          this.height = height;      }      public int getWidth() {          return width;      }      public int getHeight() {          return height;      }      public void draw() {          System.out.println("Drawing rectangle at (" + getX() + ", " + getY() + ") with width " + width + " and height " + height);      }      public int getArea() {          return width \* height;      }      public int getPerimeter() {          return 2 \* (width + height);      }  }  Shape shape = new Shape(10, 20);  shape.draw(); // draws shape at (10, 20)  Rectangle rectangle = new Rectangle(30, 40, 50, 60);  rectangle.draw(); // draws rectangle at (30, 40) with width 50 and height 60  System.out.println("Area: " + rectangle.getArea()); // prints "Area: 3000"  System.out.println("Perimeter: " + rectangle.getPerimeter()); // prints "Perimeter: 220" |

Both composition and inheritance have their own strengths and weaknesses, and the choice between them ultimately depends on the specific needs of the application being developed. Here are some of the Key Differences between them:

* Inheritance entails constructing a subclass that draws characteristics and behavior from a parent class, whereas composition involves using objects to build up a class.
* In contrast to inheritance, which keeps the subclass closely tied to the parent class, composition allows for the easy swapping out or replacement of the composed objects.
* In contrast to inheritance, which can result in a rigid and inflexible class hierarchy, composition permits greater flexibility and modularity.
* While inheritance is better suitable for creating class hierarchies when there is a clear "is-a" relationship between classes, composition tends to be better suited for creating complicated systems with several replaceable elements.