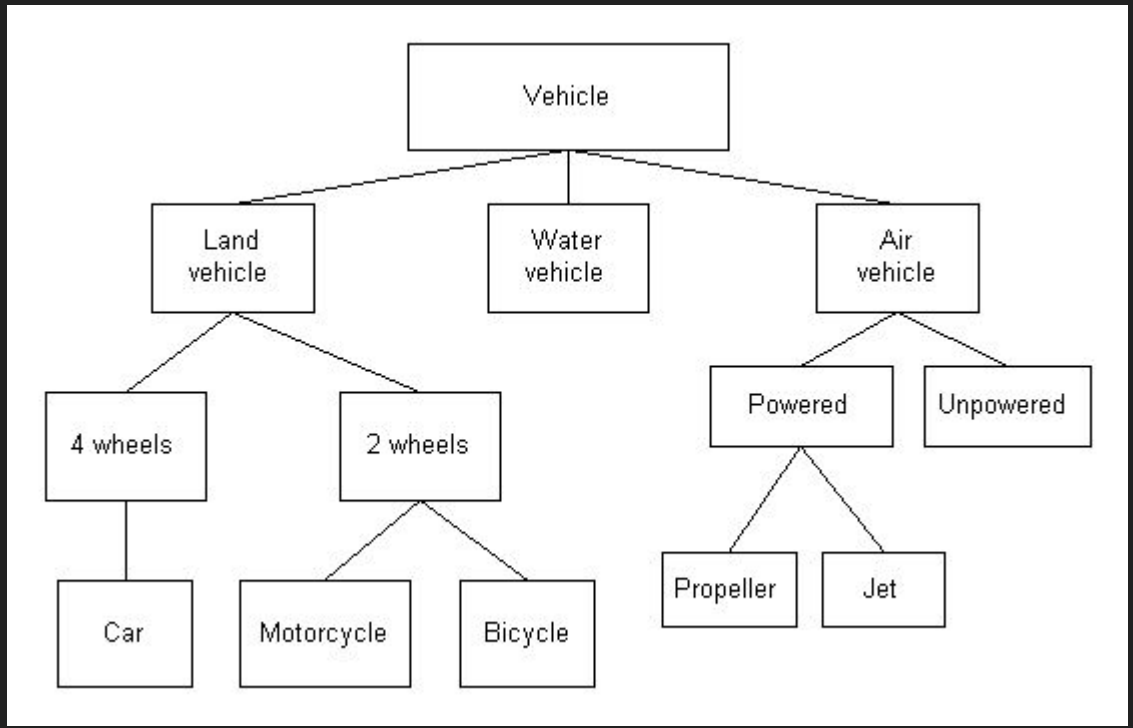
دستورکار ۶

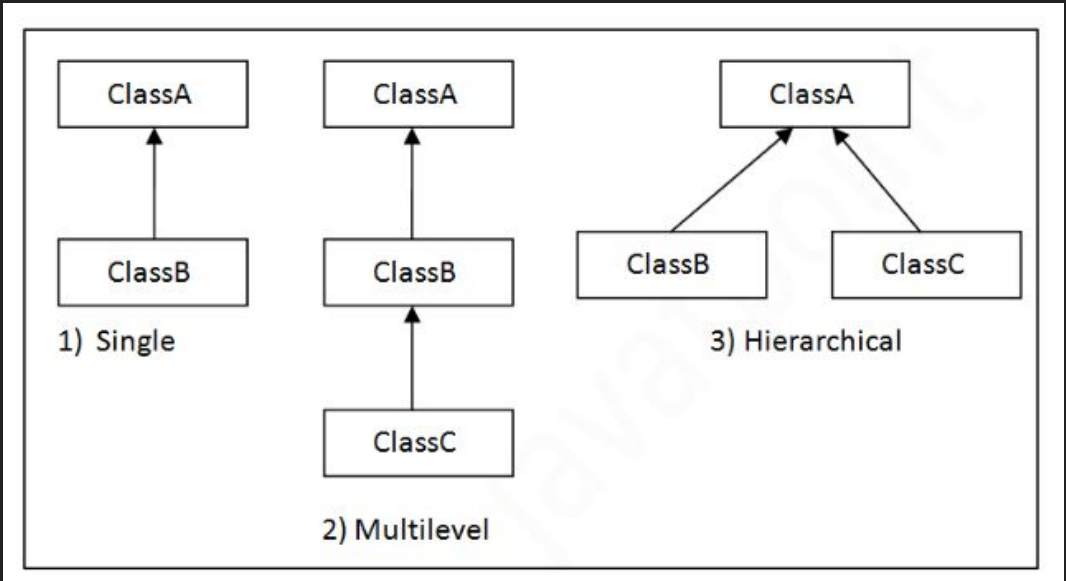
Inheritance

Inheritance is a fundamental concept in Java (and object-oriented programming in general) that allows classes to inherit properties and behaviors from other classes. It enables code reuse, promotes modularity


Vehicle
Land vehicle
Water vehicle
Air vehicle
4 wheels
2 wheels
Powered
Unpowered
Car
Motorcycle
Bicycle
Propeller
Jet

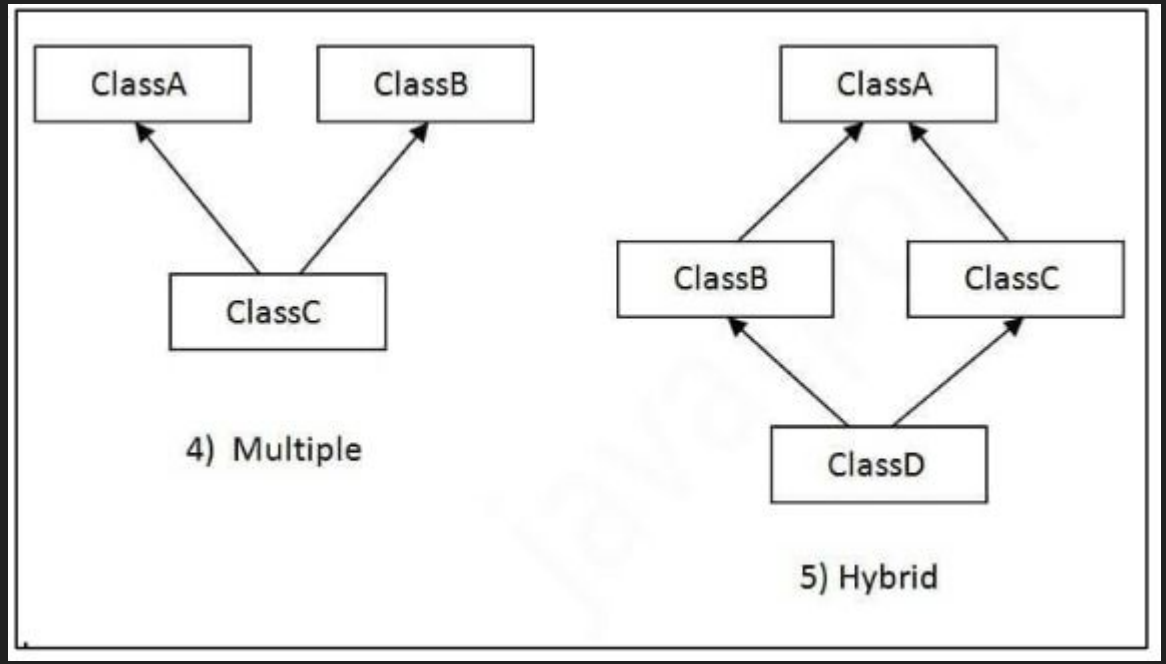
inheritance

the following inheritance models are supported in java



inheritance

the following inheritance models are not supported in java



Why?

- suppose object c inherits both from objects A and B have implemented a method with the same name but with different implementation. then which one should the compiler call?
- to solve this problem and to avoid any future problems that might occur, the java compiler does not permit a class to inherit from 2 classes even if they don’t share any mutual methods or fields. thus minimizing the errors that could occur in runtime

the super keyword

How to access the parent object’s fields?

- if a method is defined as private in the parent class, no child could access that method.
- in order to change this define the method must be either `public` or `protected`

```
public class Computer{
	private Memory memory;
	private CPU cpu;
	private ArrayList<Peripheral> peripherals;
	public Computer(Memory memory, CPU cpu, ArrayList<Peripheral> peripherals){
		this.cpu = cpu;
		this.memory = memory;
		this.peripherals = peripherals;
	}

	// suppose this method is only called inside the Computer method
	private void compute(){
		// some computation is done here
	}
}
public class Laptop extends Computer {
	private double batterySize;

}
```

```java
public class Main{
      public static void main(String[] args){
            Laptop laptop = new Laptop(); // will this work?
      }
}
```

```java
public class Computer{
	private Memory memory;
	private CPU cpu;
	private ArrayList<Peripheral> peripherals;
	public Computer(Memory memory, CPU cpu, ArrayList<Peripheral> peripherals){
		this.cpu = cpu;
		this.memory = memory;
		this.peripherals = peripherals;
	}

	// suppose this method is only called inside the Computer method
	private void compute(){
		// some computation is done here
	}
}
public class Laptop extends Computer {
	private double batterySize;

	public Laptop(double batterySize, Memory memory, CPU cpu, ArrayList<Peripheral> peripherals){
		super(memory, cpu, peripherals); // calls the constructor of the parent class
		this.batterySize = batterySize;

}
```

notes

- `super()` must be called inside the constructor, if the constructor of the parent class takes no arguments, then the compiler will insert it automatically
- `super()` must be the first command in the child constructor

```
Computer computer1 = new Computer(new Memory(), new CPU(), new ArrayList<Peripheral>());
Laptop laptop1 = new Laptop(75000, new Memory(), new CPU(), new ArrayList<Peripheral>());
Computer computer2 = laptop1; // this is allowed
Laptop laptop2 = computer1; // this is not allowed
ArrayList<Computer> computers = new ArrayList();
computers.add(laptop1);
computers.add(computer1);
```

instanceof keyword

The instanceof keyword in Java is used to check whether an object is an instance of a particular class or implements a specific interface. It allows you to determine the type of an object at runtime.

```java
class Animal {
    // Animal class implementation
}

class Dog extends Animal {
    // Dog class implementation
}

class Cat extends Animal {
    // Cat class implementation
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();

        if (animal instanceof Dog) {
            System.out.println("The animal is a dog.");
        } else if (animal instanceof Cat) {
            System.out.println("The animal is a cat.");
        } else {
            System.out.println("The animal is not a dog or a cat.");
        }
      // but how can we convert animal back to Dog type again?

    }
}
```

Casting

- **Downcasting** in Java refers to the process of casting an object reference from a more general type (superclass or interface) to a more specific type (subclass). It allows you to access the specific members or behaviors of the subclass that are not present in the superclass.
- **Upcasting** in Java refers to the process of casting an object reference from a more specific type (subclass) to a more general type (superclass) in the class hierarchy. It allows you to treat an object of a subclass as an object of its superclass. Unlike downcasting, upcasting does not require an explicit cast because it is implicitly done by the Java compiler. Since the subclass inherits all the properties and behaviors of the superclass, an object of the subclass can be assigned to a variable of the superclass type without any explicit casting.

upcasting and downcasting example

```
class Animal {
    // Animal class implementation
}

class Dog extends Animal {
    public void bark() {
        System.out.println("Woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();

        // Upcasting from Dog to Animal
        Animal animal = dog;

        // Accessing superclass members
        // (bark() method is not accessible)
        if(animal instanceof Dog){
               // upcasting example
               dog = (Dog) animal;
        }

    }
}
```

method overriding

- Method overriding in Java is the ability of a subclass to provide a different implementation of a method that is already defined in its superclass.
- It allows a subclass to replace or modify the behavior of the inherited method.
- The method in the subclass must have the same name, return type, and parameters as the method in the superclass.
- Method overriding is achieved by using the @Override annotation to indicate that the method is intended to override a superclass method.
- During runtime, the JVM determines the actual type of the object and invokes the overridden method accordingly.

method overriding example

```
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("The dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.makeSound(); // Output: The dog barks
    }
}
```

abstract classes

An abstract class in Java is a class that cannot be instantiated directly but serves as a blueprint for other classes. It is designed to be extended by subclasses, which provide implementations for the abstract methods defined in the abstract class.

Abstract methods: An abstract class can contain one or more abstract methods. These methods are declared without an implementation and are meant to be overridden by the subclasses. Subclasses must provide concrete implementations for all the abstract methods inherited from the abstract class.

Abstract classes are useful when you want to define a common set of methods and behavior that should be shared among multiple related classes, while still allowing each subclass to have its own specific implementation. They promote code reusability, enforce method implementation contracts

abstract classes

```
abstract class Animal {
        private String name;
        private int age;

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

        public abstract void makeSound();

        public void move() {
        System.out.println(name + " is moving.");
        }

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

        public String getName() {
        return name;
        }

        public int getAge() {
        return age;
        }
}
```

abstract classes

```
class Dog extends Animal {
      public Dog(String name, int age) {
      super(name, age);
      }

      public void makeSound() {
      System.out.println("Woof!");
      }
}
```

interfaces

- In Java, an interface is a programming construct that defines a contract of methods that a class must implement. It serves as a blueprint for classes, specifying a set of methods that must be implemented without specifying the implementation details.
- a class can implement multiple interfaces
- the class implementing the interface must implement the methods inside the interface.

interfaces

```
interface Animal {
        void makeSound();
}

interface Movable {
        void move();
}

class Dog implements Animal, Movable {
        public void makeSound() {
        System.out.println("Woof!");
        }

        public void move() {
        System.out.println("Dog is running.");
        }
}

class Bird implements Animal, Movable {
        public void makeSound() {
        System.out.println("Chirp!");
        }

        public void move() {
        System.out.println("Bird is flying.");
        }
}
```