CORE JAVA

Day-5

INHERITANCE

- Inheritance in Java is a fundamental concept in object-oriented programming that allows a class to inherit properties and methods from another class.
- It's one of the core concepts in Java used to achieve code reusability and establish “is-a” relationship between classes.
- There are to important classes involved in inheritance .
  - Superclass (Parent class): The class whose features are inherited.
  - Subclass (Child class): The class that inherits the features from the superclass. It can also add its own features.
- Parent class is a broad category that contains general properties and methods
- Child class takes the form of parent class by deriving the general properties and methods of parent class.

ADVANTAGES OF INHERITANCE

- Reusability: Inheritance supports the reuse of existing code without having to rewrite the code from scratch.
- Extensibility: Enhancing the functionality of existing classes by adding more features in subclasses.
- Maintainability: Changes in the superclass will propagate to subclasses, which can make maintaining and debugging code easier.

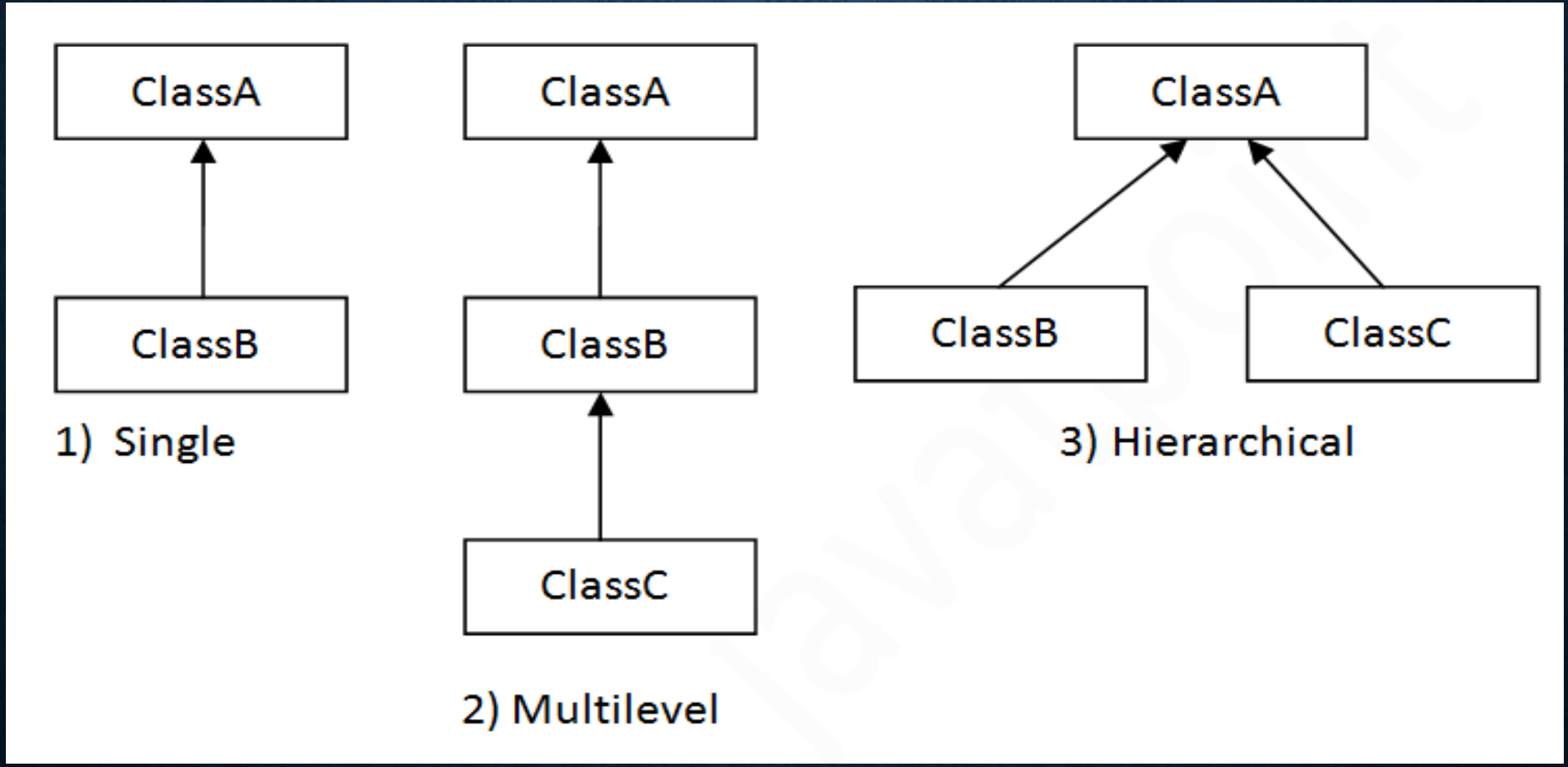
TYPES OF INHERITANCE

Java supports several types of inheritance:

- Single Inheritance: A subclass inherits from one superclass only.
- Multilevel Inheritance: A class can inherit from a subclass making it a superclass for the new class.
- Hierarchical Inheritance: Multiple subclasses inherit from a single superclass.

- Java does not support multiple inheritance directly (a class cannot inherit from more than one class); however, it can be achieved using interfaces.

TYPES OF INHERITANCE



CONSTRUCTOR INVOCATION SEQUENCE

When a subclass object is created, Java ensures that the constructors of all its superclasses are called in a specific order, starting from the top of the hierarchy down to the subclass. Here's how it works:

Constructor Invocation Sequence

- Topmost Superclass First: The constructor of the topmost superclass in the hierarchy (often Object class, since all classes implicitly extend Object if no other superclass is specified) is called first.
- Superclass to Subclass: After the topmost superclass constructor completes, the control returns to the next subclass constructor down the hierarchy, and this continues until the constructor of the actual class being instantiated is called.

The rationale behind this sequence is that a subclass can depend on properly initialized state that is established by its superclass constructors. This ensures that any initialization performed by superclasses is done before the subclass's initialization logic runs.

POLYMORPHISM

- Polymorphism is derived from two Greek words, “poly” and “morph”, which mean “many” and “forms”, respectively.
- Hence, polymorphism meaning in Java refers to the ability of objects to take on many forms. In other words, it allows different objects to respond to the same message or method call in multiple ways.
- Polymorphism allows coders to write code that can work with objects of multiple classes in a generic way without knowing the specific class of each object.

TYPES OF POLYMORPHISM IN JAVA

Java supports two types of polymorphism:

1. Compile-time Polymorphism (Static Binding or Method Overloading):
   - Method Overloading: This occurs when multiple methods in the same class have the same name but different parameters (different type and/or number). The compiler determines which method to invoke based on the method signature (method name and parameter list).
2. Runtime Polymorphism (Dynamic Binding or Method Overriding):
   - Method Overriding: This occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. At runtime, the Java Virtual Machine (JVM) determines the appropriate method to call based on the object's actual class type, not the reference type.

OVERRIDING

- When a method in a subclass has the same name, return type, and parameters as a method in its superclass, the method in the subclass is said to override the method in the superclass. Here are a few key points about method overriding:
  - Same Signature: The method in the subclass must have the same signature as the method in the superclass it overrides (same name, same return type, and same parameter list).
  - Access Level: The access level cannot be more restrictive than the overridden method's access level. For example, if the superclass method is protected, the overriding method in the subclass cannot be private.
  - @Override Annotation: While this annotation is not required, it is good practice to use it because it helps the compiler verify that you are indeed overriding a method from the superclass.

RULES AND CONSIDERATIONS

- Superclass Reference: Method overriding is a runtime concept. If a method of a superclass is overridden in a subclass, the version of the method executed will be determined at runtime based on the object's actual class, not the reference type.
- super Keyword: A subclass can call the superclass's method that it overrides using the super keyword. This is often used when the subclass wants to extend the behavior of the superclass's method rather than completely replacing it.
- Final Methods: Methods that are declared as final cannot be overridden. This is used to prevent alteration in the method's behavior by subclasses.
- Static Methods: Static methods cannot be overridden. If a subclass defines a static method with the same signature as a static method in the superclass, it is known as method hiding, not overriding.
- Constructors: Constructors are not methods, and thus, they cannot be overridden. Each class can define constructors independently of its superclass.
- Private Methods: Private methods are not visible to subclasses and therefore cannot be overridden. If a subclass defines a private method with the same signature as a private method in the superclass, it's simply a new method, not an overridden one.

FINAL VARIABLES

- When the final keyword is used with variables, it means that the variable's value cannot be changed once it has been assigned. This applies to both primitive types and reference variables.
- Primitive type: The value of the primitive cannot be changed.
- Reference type: The reference itself cannot be changed to point to another object, but the object it points to can still be modified unless the object itself is immutable.
- Examples:

```
final int x = 10;  // x cannot be changed after initialization
final String greeting = "Hello";  // greeting cannot point to another String object
```

FINAL METHODS

- A final method cannot be overridden by subclasses. This is useful when you want to lock down the method's behavior to prevent any subclass from changing its functionality.
- Example:

```
public class Car {
    public final void startEngine() {
        System.out.println("Engine started.");
    }
}
public class ElectricCar extends Car {
    // This would cause a compile-time error:
    // @Override
    // public void startEngine() {
    //     System.out.println("Electric engine started.");
    // }
}
```

FINAL CLASSES

- A final class cannot be subclassed. This is often used to prevent modification to the structure or functionality of the class, ensuring security and consistency.
- Example:

```
public final class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
// This would cause a compile-time error:
// public class AdvancedCalculator extends Calculator {
//     ...
// }
```

ABSTRACT CLASSES

- In Java, an abstract class is a special type of class that cannot be instantiated on its own and is designed to be a superclass for other classes.
- Abstract classes are used to provide a base or template for other classes to extend and build upon.
- They can include a mix of methods that are either fully implemented (concrete methods) or declared without an implementation (abstract methods).

CHARACTERISTICS OF ABSTRACT CLASSES

- Cannot Be Instantiated: You cannot create an instance of an abstract class directly. Its purpose is to be subclassed by other classes.
- Abstract Methods: An abstract class can contain abstract methods—methods without a body. These methods must be implemented by subclasses.
- Concrete Methods: Abstract classes can also contain concrete methods (methods with a body). This allows you to define some default behavior for subclasses while requiring other behaviors to be defined explicitly in each subclass.
- Constructors: Abstract classes can have constructors. These constructors are not used to instantiate the class but to initialize attributes of the class when a subclass is instantiated.
- If abstract methods are not overridden in derived class then derived class also becomes abstract class.

USES OF ABSTRACT CLASSES

- Promoting Reusability: Abstract classes allow you to define some common functionality in one place and reuse it in multiple subclasses, while also forcing subclasses to implement specific methods.
- Providing a Template: They act as a template for other classes, ensuring a common interface or behavior across all subclasses.
- Enhancing Code Readability and Maintenance: Abstract classes make it easier to understand the relationships between classes and what they are supposed to do.

RTTI

- RTTI allows you to identify the actual type of an object during runtime, which is fundamental to successfully downcasting from a base class reference to a derived class reference.
- RTTI in Java is primarily facilitated through two mechanisms:
  - 1. instanceOf
  - 2. getClass()

INSTANCEOF

- The instanceof Operator: This operator checks whether an object is an instance of a specific class or an interface. It's used to ensure safe casting between class types. For instance, before performing a downcast, you typically check if the object is an instance of the target type to avoid a ClassCastException.

```
if (animal instanceof Dog) {
    ((Dog) animal).bark();
}
```

GETCLASS()

- The getClass() Method: This method is defined in the Object class, which is the superclass of all classes in Java. It returns the runtime class of an object, encapsulated as an instance of Class. Unlike instanceof, which checks for type compatibility, getClass() tells you the exact runtime class of the object.

```
Class<?> clazz = animal.getClass();
if (clazz.equals(Dog.class)) {
    ((Dog) animal).bark();
}
```

KEY DIFFERENCES AND USAGE

- instanceof checks if an object is an instance of a particular class or any of its subclasses and is also applicable for interface types. It's the preferred method for type checking before downcasting because it accounts for polymorphism and hierarchy.
- getClass() returns the exact runtime class of the object, with no consideration for the class hierarchy (i.e., it does not consider subclasses). It's useful when you need exact type matching, which is less common in typical polymorphic scenarios.