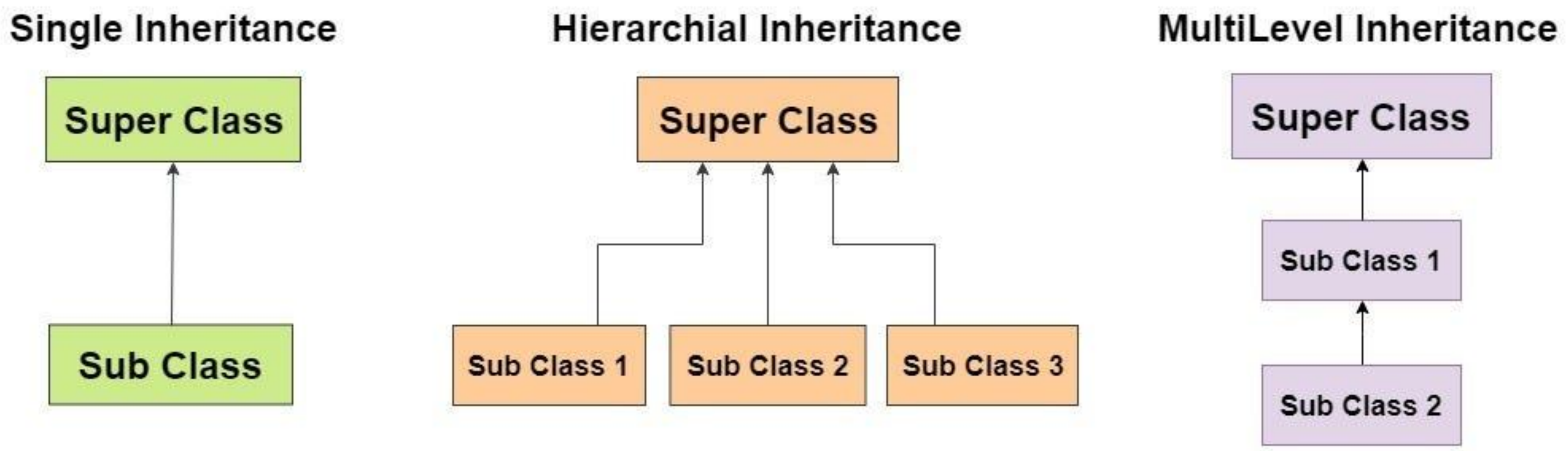
برنامه نویسی پیشرفته

رفع اشکال: جلسه ۵

Inheritance and polymorphism
Type casting (up casting & down casting)
Abstract classes and interfaces

INHERITANCE IN JAVA

Inheritance in Java refers to the mechanism by which one class can inherit the fields (variables) and methods (functions) from another class. This allows for the reuse of code and promotes the creation of a class hierarchy, where subclasses extend the functionality of their parent class.
Java supports single inheritance. This means that a class can inherit from only one superclass.



THE EXTEND KEYWORD

The **extend** keyword in Java is used to implement inheritance.

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

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

class Cat extends Animal {
    void meow() {
        System.out.println("Meowing...");
    }
}
```

THE SUPER KEYWORD

The **super** keyword in Java is used to refer to the immediate parent class object. It is commonly used to access parent class constructors, fields, and methods.
If no super() call is provided as the very first statement, Java will implicitly insert a no-argument super() call. This only works if the parent class has a no-argument constructor.

```
class Animal {
    Animal(String name) {
        System.out.println("Animal name: " + name);
    }
}
class Dog extends Animal {
    Dog() {
        super("Dog");  // Calls the parent class constructor
    }
}
```

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    void sound() {
        super.sound();  // Calls the parent class method
        System.out.println("Dog barks");
    }}
```

THE INSTANCEOF KEYWORD

The instanceof keyword is used to test whether an object is an instance of a particular class or implements a particular interface. It returns true if the object is an instance of the specified class or interface, otherwise it returns false.
This is particularly useful when determining the type of an object at **runtime**.

```
class Animal {}
class Dog extends Animal {}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();
        System.out.println(animal instanceof Dog);  //Returns true
        System.out.println(animal instanceof Animal);  // Returns true
    }
}
```

THE OBJECT CLASS

The Object class is at the top of Java's inheritance chain. Every class you create—whether you explicitly extend another class or not—automatically inherits from Object.

The Object class provides several fundamental methods that are available to all Java objects. Some of these methods include:

- toString(): Returns a string representation of the object.
- equals(Object obj): Determines whether two objects are equal.
- hashCode(): Returns an integer hash code for the object.
- getClass(): Returns the runtime class of the object.

Having a common base class (Object) ensures that all objects share a basic set of behaviors and methods. This uniformity is critical for various operations like collections management, where methods like equals() and hashCode() are used to compare objects.

TYPE CASTING

Type casting is crucial when dealing with object types within an inheritance hierarchy. It allows you to treat an object as a different type, either moving up or down the inheritance chain. There are two primary forms of type casting: upcasting and downcasting.

Upcasting (Widening Conversion)

- Upcasting occurs when you cast a subclass reference to a superclass reference. This conversion is implicit because every instance of a subclass is also an instance of its superclass.
- Facilitates polymorphism by allowing a subclass object to be treated as an object of its superclass.

Downcasting (Narrowing Conversion)

- Downcasting is the reverse process, where a superclass reference is cast back to a subclass reference. This conversion must be done explicitly using a cast operator.
- Always use the instanceof keyword before downcasting to avoid ClassCastException.

TYPE CASTING EXAMPLE

```
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}



class Dog extends Animal {
void sound() {
        System.out.println("Dog barks");
    }
}
```

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

        // casting back explicitly
        if (animal instanceof Dog) {
            Dog dog = (Dog) animal;
            dog.sound();  // Outputs: Dog barks
        } else {
            System.out.println("Not an instance");
        }
    }
}
```

POLYMORPHIC METHOD DISPATCH

Polymorphism is a cornerstone of OOP that enables objects of different types to be treated as objects of a common superclass. Java implements polymorphism through method overriding, where a subclass provides its own implementation of a method defined in its superclass.

Dynamic Method Dispatch is the mechanism by which Java determines which method implementation to execute at runtime based on the actual object's type, not the reference type.

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }}
class Dog extends Animal {
void sound() {
        System.out.println("Dog barks");
    }}
```

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

JAVA ANNOTATIONS: @OVERRIDE

Java **Annotations** are metadata that provide additional information about your code. They don't directly change the program's behavior but can be used by the compiler, development tools, or at runtime to enforce certain behaviors or generate code. It's possible to implement **custom annotations**.

The **@Override** annotation is one of the most commonly used annotations in Java. It tells the compiler that the annotated method is meant to override a method in its superclass or implement an abstract method from an interface.

The **@Override** annotation:

- Compile-Time Checking: When you use @Override, the compiler verifies that a method in the subclass actually overrides a method from its parent. If the method signature is incorrect, the compiler will issue an error.
- Improved Readability
- Bug Prevention

ABSTRACT CLASSES

An **abstract class** in Java cannot be instantiated directly and may contain abstract methods (methods declared without an implementation). Subclasses must implement all abstract methods, making abstract classes useful for defining templates for other classes.

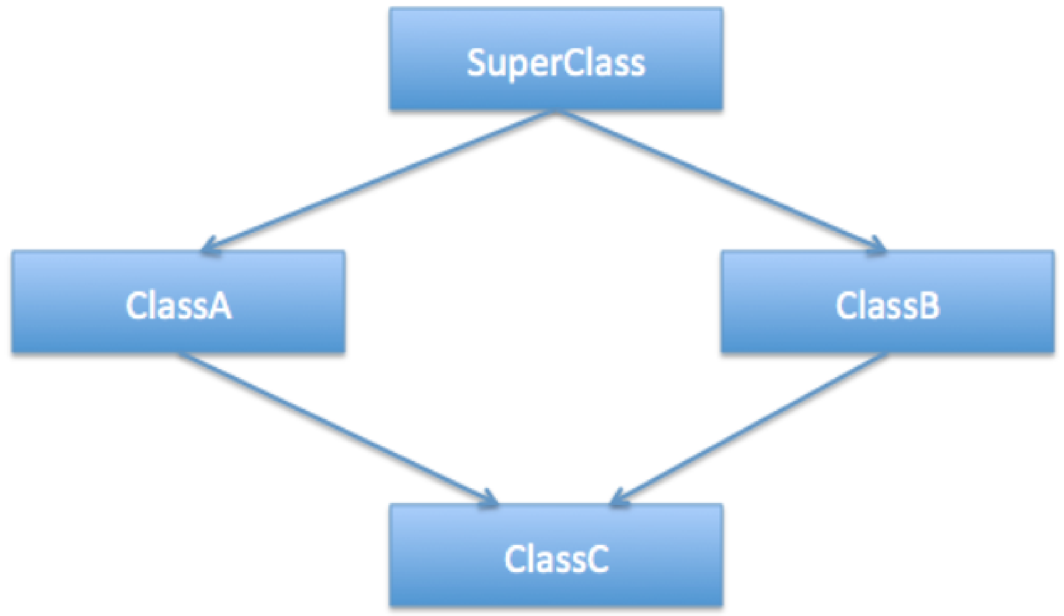
```
abstract class Animal {
    abstract void sound();  // Abstract method

    void eat() {  // Concrete method
        System.out.println("Eating...");
    }
}
```

```
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Barking...");
    }
}
```

THE DIAMOND PROBLEM

The diamond problem in Java refers to the ambiguity that can arise when a class inherits the same method from multiple sources.
Although Java does not support multiple inheritance of classes (which is the classic context for the diamond problem), it does allow a class to implement multiple interfaces.

- Since Java 8 introduced default methods in interfaces, a similar issue can occur if two interfaces provide a default method with the same signature.



INTERFACE

An interface in Java defines a contract with method declarations (and optionally constants) that a class must implement. A class that implements an interface is obligated to provide concrete implementations for all the methods declared in the interface.

Java also allows a class to implement multiple interfaces, offering a workaround for the limitations of single inheritance.

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



interface Pet {
    void play();
}
```

```
class Dog implements Animal, Pet {
    public void eat() {
        System.out.println("Eating...");
    }

    public void play() {
        System.out.println("Playing...");
    }
}
```

THE DIAMOND PROBLEM IN JAVA 8

With Java 8, interfaces can have default methods—methods with a provided implementation. When a class implements two interfaces that each provide a default method with the same signature, Java forces the developer to override the conflicting method in the implementing class, thereby providing a single, unambiguous implementation.

```
interface InterfaceA {
    default void print() {
        System.out.println("Print from InterfaceA");
    }
}
interface InterfaceB {
    default void print() {
        System.out.println("Print from InterfaceB");
    }
}
```

```
public class DiamondProblemExample implements
InterfaceA, InterfaceB {
    @Override
    public void print() {
        // Choose explicitly:
        InterfaceA.super.print();
    }
}
```

QUESTION 1

Given the following class definitions:

```
class A { }
class B extends A { }
class C extends B { }
class D extends A { }
```

Which of the following instantiations will result in a compile-time error?

A. A a = new B();
B. B b = new C();
C. A a = new D();
D. B b = new D();
E. A a = new A();

ANSWER 1

Given the following class definitions:

class A { }
class B extends A { }
class C extends B { }
class D extends A { }

Which of the following instantiations will result in a compile-time error?

A. A a = new B();
B. B b = new C();
C. A a = new D();
D. B b = new D();
E. A a = new A();

QUESTION 2

Given the following code:

```
class Base {
    Base(String s) {
        System.out.print(s);
    }
}
class Derived extends Base {
    Derived() {
        super("X");
        System.out.print("Y");
    }
}
public class Test {
    public static void main(String[] args) {
        new Derived();
    }
}
```

Which of the following statements is **false**?

A. The output of the program is XY.

B. The super call in the Derived constructor must be the first statement.

C. If Base had no matching constructor, this code would fail to compile.

D. If no explicit super call were provided in Derived(), Java would automatically call Base().

ANSWER 2

Given the following code:

```
class Base {
    Base(String s) {
        System.out.print(s);
    }
}
class Derived extends Base {
    Derived() {
        super("X");
        System.out.print("Y");
    }
}
public class Test {
    public static void main(String[] args) {
        new Derived();
    }
}
```

Which of the following statements is false?

A. The output of the program is XY.

B. The super call in the Derived constructor must be the first statement.

C. If Base had no matching constructor, this code would fail to compile.

D. If no explicit super call were provided in Derived(), Java would automatically call Base().

QUESTION 3

Consider the following classes:

```
class Animal {
    void sound() { System.out.print("Animal"); }
}
class Dog extends Animal {
    @Override
    void sound() { System.out.print("Dog"); }
    void sound(String extra)
            { System.out.print("Dog" + extra); }
}
public class TestDispatch {
    public static void main(String[] args) {
        Animal a = new Dog();
        a.sound();
    }
}
```

Which statement is **false**?

A. The output is determined at runtime by the actual type of the object.

B. The overloaded method sound(String) is not invoked because the reference type is Animal.

C. If we add ((Dog)a).sound(" barks"); after a.sound();, the output will be DogDog barks.

D. Overloading is resolved at compile time.

ANSWER 3

Consider the following classes:

```
class Animal {
    void sound() { System.out.print("Animal"); }
}
class Dog extends Animal {
    @Override
    void sound() { System.out.print("Dog"); }
    void sound(String extra)
            { System.out.print("Dog" + extra); }
}
public class TestDispatch {
    public static void main(String[] args) {
        Animal a = new Dog();
        a.sound();
    }
}
```

Which statement is **false**?

A. The output is determined at runtime by the actual type of the object.

B. The overloaded method sound(String) is not invoked because the reference type is Animal.

C. If we add ((Dog)a).sound(" barks"); after a.sound();, the output will be DogDog barks.

D. Overloading is resolved at compile time.

QUESTION 4

Which of the following statements about interfaces in Java are **true**? (Select all that apply)

A. A class can implement multiple interfaces.
B. Interfaces may contain abstract methods, default methods, and static methods.
C. If two interfaces have default methods with the same signature, a class implementing both must override the method.
D. Interfaces can have instance fields.

ANSWER 4

Correct Answers: A, B, C

Explanation:

- **A:** True: this is Java's way to achieve multiple inheritance of type.
- **B:** True: Java 8 and later allow default and static methods.
- **C:** True: to resolve ambiguity, the implementing class must override the default method.
- **D:** False: interfaces cannot have instance (non-final) fields; they may have constants (public static final).

TIME TO CODE

You are tasked with implementing a **Shape and Area Calculation System** in Java that models various types of shapes using object-oriented principles. The system should allow you to create and calculate areas for different shapes like circles, rectangles, and triangles.

1. **ShapeAreaCalculator Interface**: An interface that defines a method calculateArea.
2. **Shape**: An abstract class that represents a generic shape. It should have an abstract method displayInfo.
3. **Circle**: A subclass of Shape that has a radius property and implements the calculateArea() method.
4. **Rectangle**: A subclass of Shape that has width and height properties, and implements the calculateArea.
5. **Triangle**: A subclass of Shape that has a base and height property, and implements the calculateArea
6. **Validation**: The system should validate the provided dimensions.

پایان