upGrad

#LifeKoKaroLift

Object-Oriented Programming: Polymorphism

upGrad

Course: Object-Oriented Analysis, Design & Programming

Lecture On: Object-Oriented Programming: Polymorphism

upGrad

Topics covered in the previous class...

1. Generalization Relationship and Inheritance between classes
2. The ‘extends’ keyword to provide inheritance relationship between classes
3. Types of Inheritance: Single, Multilevel, Hierarchical and Multiple
4. The super() keyword (to call parent class constructor)
5. Disadvantages of Inheritance
6. Object class: Parent of all Java classes
7. Abstract class and Abstract methods
8. Interface

Poll 1 (15 sec)

Which of the following is the superclass of all other classes?

1. Super
2. Object
3. Parent
4. Base

Poll 1 (15 sec)

Which of the following is the superclass of all other classes?

1. Super
2. **Object**
3. Parent
4. Base

upGrad

Today's Agenda

- **Making Your Code Flexible Using Polymorphism**
  - Introduction to Polymorphism and its importance
  - Type Casting and Automatic Type Conversion
  - Different Types of Polymorphism: Static and Dynamic
  - Final classes, attributes and methods
  - Anonymous class

upGrad

Polymorphism

Polymorphism

upGrad

- Let's understand polymorphism with the help of the println() method, which we have been using for a long time.
- You can call the println() method in the following ways:
  - Without any argument

```
System.out.println();
```

  - With an int argument

```
System.out.println(100);
```

  - With a String argument

```
System.out.println("upGrad");
```

Polymorphism

upGrad

- So, how is the println() method declared and defined in System.out?
- Is it defined as follows?

```
public void println() {
   ...
}
```

Or

```
public void println(int i) {
   ...
}
```

Or with the String argument?

Polymorphism: Static

upGrad

- The println() method is defined using the principles of polymorphism.
- The term polymorphism literally means “having multiple forms or existing in many forms”.
- Here, the println() method exists in multiple forms. This method can work without any argument (printing an empty line on the console) or with several other arguments (printing an argument on the console).
- So, how it is defined inside the code?

Polymorphism: Static

upGrad

- The println() method is defined as follows:

```
public void println() {}
public void println(boolean x) {}
public void println(char x) {}
public void println(char[] x) {}
public void println(double x) {}
public void println(float x) {}
public void println(int x) {}
public void println(long x) {}
public void println(Object x) {}
public void println(String x) {}
```

Polymorphism: Static

upGrad

- With polymorphism, you can define define several methods with the same name. This is called ***Method Overloading***.
- However, you should keep the following two points in mind:
  - Return type may or may not be same.
  - The parameter list should be different; either the number of parameters should be different or the types of parameters should be different, or the parameters should be in different order.
- These same rules apply to ***Constructor Overloading,*** about which we learnt earlier, as constructor is also a special type of method.

Polymorphism: Static

upGrad

- Method overloading is also known as ***Compile-Time Polymorphism***, as the Compiler knows which method will be called because each method has a different signature.
- As this polymorphism is happening at compile time, Compile-Time Polymorphism is also called ***Static Polymorphism***.

Poll 2 (15 sec)

Overloaded methods must have _____. (Note: More than one option may be correct.)

1. Same method names
2. Different method names
3. Same parameter lists
4. Different parameter lists

Poll 2 (15 sec)

Overloaded methods must have _____. (Note: More than one option may be correct.)

1. **Same method names**
2. Different method names
3. Same parameter lists
4. **Different parameter lists**

upGrad

Poll 3 (15 sec)

Which of the following are the correct ways to overload the given method? (Note: More than one option may be correct.)

```
int getValue(int a, int b) {...}
```

1. `double getValue(int a, int b) {...}`
2. `int getValue(double a, double b) {...}`
3. `int getValue(int a, int b) {...}`
4. `int getValue(double a, double b) {...}`

Poll 3 (15 sec)

Which of the following are the correct ways to overload the given method? (Note: More than one option may be correct.)

```
int getValue(int a, int b) {...}
```

1. `double getValue(int a, int b) {...}`
2. **`int getValue(double a, double b) {...}`**
3. `int getValue(int a, int b) {...}`
4. **`int getValue(double a, double b) {...}`**

Hands-On Experience

upGrad

TODO:

- Create a class and name it MaxFinder.
- Inside this class, provide the following three overloaded methods:
    - int getMax(int a, int b)
    - int getMax(int a, int b, int c)
    - int getMax(int a, int b, int c, int d)
- Use these methods to find the maximum number among two, three or four numbers.

upGrad

Polymorphism: Dynamic

Polymorphism: Dynamic

upGrad

- The println() method works with all primitive data types, String types and Object types.
- But why does it need to work with Object type?
- Let's create an instance of the Object type and print it using the println() method, as shown below.

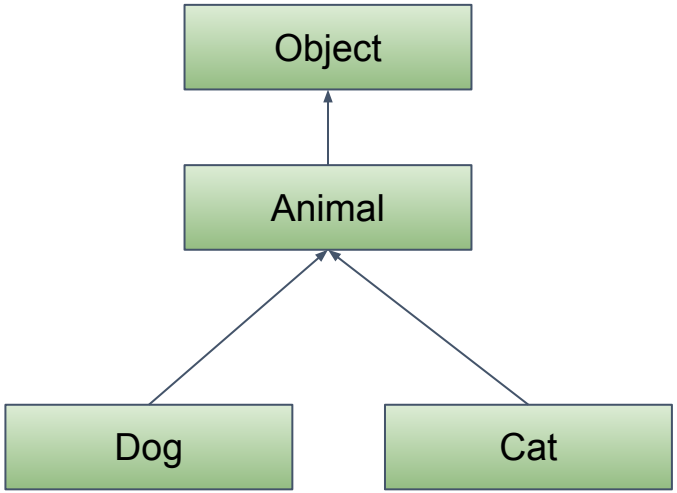
```
Object obj = new Object();
System.out.println(obj);
```

- Output: java.lang.Object@27f674d
- So, this method just printed out the full class name (package + class name) and the memory address in the heap where this object was stored. So, what is use of this method, and why it provided?

Polymorphism: Dynamic

upGrad

- To understand this, we first need to learn about type casting and its two types: upcasting and downcasting.
- Let's consider the following class hierarchy:



Polymorphism: Dynamic

upGrad

- As we learnt earlier, the inheritance relationship between classes is also called the ‘is-a’ relationship.
- Here, we can say ‘Dog is an Animal’ and ‘Cat is an Animal’. We can also say ‘Animal is an Object’.
- The ‘is-a’ relationship not only applies to the direct parent and child but also to the indirect parent and child.
- This means that we can say ‘Cat is an Object’ and ‘Dog is an Object’.
- But how do we represent this in code?

Polymorphism: Dynamic

upGrad

- `Cat cat = new Cat();`

Cat is a Cat.

- `Animal animal = new Cat();`

Cat is an Animal.

- `Object object = new Cat();`

Cat is an Object.

- The process of assigning a child class object to a superclass type is called ***upcasting***.

Polymorphism: Dynamic

upGrad

- Upcasting can be done implicitly, i.e., without informing the Compiler explicitly that we are casting from one type to another type. This is the same as casting int to long.
- Upcasting is allowed because a child class object will have all the attributes and methods that can be accessed using the parent class type.
- Let’s understand this with the help of an example.

Polymorphism: Dynamic

upGrad

- Consider the following classes for Animal and Cat:

```
class Animal {
    public void run() {
        System.out.println("Animal is running.");
    }
    public void sleep() {
        System.out.println("Animal is sleeping.");
    }
}


class Cat extends Animal {
    public void drinkMilk() {
        System.out.println("Cat is drinking milk.");
    }

    public void sayMeow() {
        System.out.println("Meow! Meow!");
    }
}
```

Polymorphism: Dynamic

upGrad

- So, we can call the following methods with a variable of the Animal type:
  run()
  sleep()
- The Cat class has the following four methods:
  run()
  sleep()
  drinkMilk()
  sayMeow()
- Hence, we can pass a subclass object where a superclass object is expected.

Polymorphism: Dynamic

upGrad

- Let's define a method that expects a parent class object as follows:

```
public static void makeAnimalRun(Animal animal) {
    animal.run();
}
```

- Now, let's call this method and pass an object of the type Cat (as Cat is an Animal) as follows:

```
Cat cat = new Cat();
makeAnimalRun(cat);
```

- Output: Animal is running.

Polymorphism: Dynamic

upGrad

- As you can observe, we can assign or pass a child class object where parent class objects are expected.
- This will work because a child class object will have all those methods that can be accessed through the parent class interface (attributes and methods that can be accessed outside of the class, not the Java interface that we implement).
- Thus, upcasting is allowed and works implicitly.

Polymorphism: Dynamic

upGrad

- The process of assigning a parent class object to a child class type is called ***downcasting***.
- Downcasting is similar to assigning long to int, and it can be explicitly, as shown below.

```
Animal animal = new Animal();
Cat cat = (Cat) animal;
```

- This is done explicitly because a child class can have methods that may not be present in the parent class.
- We can access child class objects with a variable of the child class type, but they will not be present in the parent class object, which may break the code.

Polymorphism: Dynamic

upGrad

- Downcasting has to be done explicitly to inform the Compiler that we know what we are doing.
- We can also check the type of an object at runtime using the ***instanceof*** operator.

Polymorphism: Dynamic

upGrad

```
Animal animal1 = new Cat();
System.out.println("animal1 is Cat? " + (animal1 instanceof Cat));
System.out.println("animal1 is Animal? " + (animal1 instanceof
Animal));
System.out.println("animal1 is Object? " + (animal1 instanceof
Object));

Animal animal2 = new Animal();
System.out.println("animal2 is Cat? " + (animal2 instanceof Cat));
System.out.println("animal2 is Animal? " + (animal2 instanceof
Animal));
System.out.println("animal2 is Object? " + (animal2 instanceof
Object));
```

instanceOf operator to check type

Polymorphism: Dynamic

upGrad

- Output:

  animal1 is Cat? true
  animal1 is Animal? true
  animal1 is Object? true
  animal2 is Cat? false
  animal2 is Animal? true
  animal2 is Object? true

- Before downcasting, it is important to always check whether the type of an object is suitable for downcasting or not using the instanceof operator.

Polymorphism: Dynamic

upGrad

- Let's return to the println() method, which was expecting an object of the type Object.
- Now, we know that the println() method is not used for printing objects of the type Object, but objects of any type. This will work as Object is a superclass for all other classes.
- Let's print an object of the Cat class as follows:

```
Cat cat = new Cat();
System.out.println(cat);
```

- Output: com.upgrad.ims.Cat@48140564

Polymorphism: Dynamic

upGrad

- Still same thing as the println() method is printing the class name and memory address. Let's check how the println() method works with the Object type. (A simple representation of the println(Object) method, not the actual implementation, is given below.)

```
public void println(Object obj) {
    System.out.println(obj.toString());
}
```

- So, it takes the object (of any type). Get the String representation of that object using the toString() method (inherited via the Object class) and call the ***println(String str)*** method to print it.

Polymorphism: Dynamic

upGrad

- The issue lies with the toString() method. The toString() method is defined in the Object class in such a way that it returns ‘package name + class name + memory location’.
- As it is inherited by all other classes, the same definition is inherited by all the objects.
- The same issue occurred with the run() method of the Animal class. When we passed an object of the Cat class to the makeAnimalRun() method, the method printed “Animal is running” because this is how it was implemented.
- So, how do you solve this issue? How do you make the run() method print “Cat is running” instead of “Animal is running.”?

Polymorphism: Dynamic

upGrad

- We can make the run() method abstract in the Animal class and provide custom implementation in the Cat class.
- However, to provide custom implementation, it is not necessary to make it abstract; we can provide custom implementation even for a concrete method.
- Let’s redefine the Cat class with custom implementation for the run() and sleep() methods.

Polymorphism: Dynamic

upGrad

```
class Cat extends Animal {
    public void run() {
        System.out.println("Cat is running.");
    }

    public void sleep() {
        System.out.println("Cat is sleeping.");
    }

    public void drinkMilk() {
        System.out.println("Cat is drinking milk.");
    }

    public void sayMeow() {
        System.out.println("Meow! Meow!");
    }
}
```

Customising the inherited methods

Providing new methods

Polymorphism: Dynamic

upGrad

- Now, let's execute the makeAnimalRun() method again as follows:

```
makeAnimalRun(new Animal());
makeAnimalRun(new Cat());
```

- Output:

  Animal is running.

  Cat is running.

- Now, it's working perfectly. The process of re-implementing or overriding the inherited methods in the child class is called ***Method Overriding***. This is another form of polymorphism.

Polymorphism: Dynamic

upGrad

- But shouldn't makeAnimalRun() method call the run() method from the Animal class, instead of the Cat class?
- When we call a method on an object, JVM first searches for that method in the class that was used to instantiate that object (in this case, the Cat class).
- If the method is present in the class, that method will be executed. If the method is not found in the class, then JVM keeps searching for the same method in the superclass, until it finds the method.
- If the method is not found even in the Object class (the topmost class in the hierarchy tree), then JVM will throw an error.

Polymorphism: Dynamic

upGrad

- In this case, the makeAnimalRun() method will not know which run() method to call, whether to call the run() method of the Animal class or that of the Cat class.
- This is decided by JVM at runtime. Thus, this process is called ***Runtime Polymorphism***.
- As things are happening at runtime, it is also called ***Dynamic Polymorphism***.

Polymorphism: Dynamic

upGrad

- Now, let's fix our code so that the println() method can print the details we want for our Cat object.
- All we need to do is override the toString() method in the Cat class as follows:

```
class Cat extends Animal {
    ...
    public String toString() {
        return "I am meow.";
    }
}
```

Polymorphism: Dynamic

upGrad

- Now, let's try to print the Cat object using the println() method as follows:

```
Cat cat = new Cat();
System.out.println(cat);

Animal animal = new Animal();
System.out.println(animal);
```

- Output:

  I am meow.

  com.upgrad.ims.Animal@48140564

Polymorphism: Dynamic

upGrad

- For the Cat object, JVM will start searching for the method from the Cat class and go all the way to the Object class. As the method is found in the Cat class, it will be executed successfully.
- For the Animal object, JVM will start searching for the method from the Animal class and go all the way to the Object class. As the method does not exist in the Animal class, the method present in the Object class will be executed.
- In any case, the makeAnimalRun() method or the Compiler does not know which method is getting called. This is decided dynamically by JVM at runtime.

Polymorphism: Dynamic

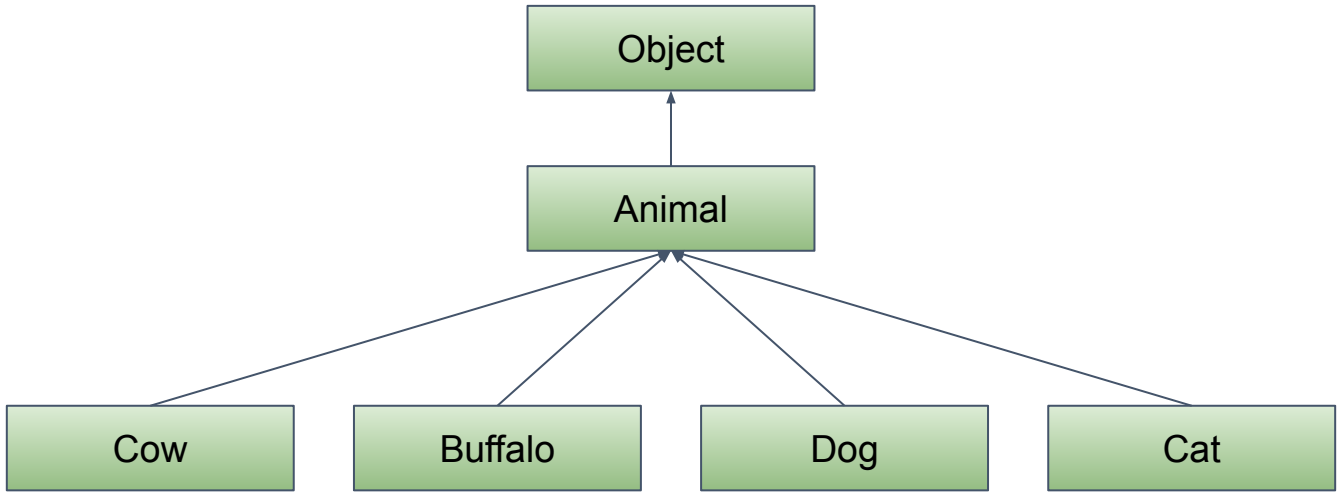
upGrad

- Polymorphism also works with interfaces. A method can accept an object of the type Interface, and you can pass any object that implements the expected interface.
- This works because if you are passing an object which implements a particular interface, then it can be assumed that the object has implemented all the methods provided by the interface.
- Let’s understand this with the help of an example.

Polymorphism: Dynamic

upGrad

- Let’s consider the following hierarchy:



Polymorphism: Dynamic

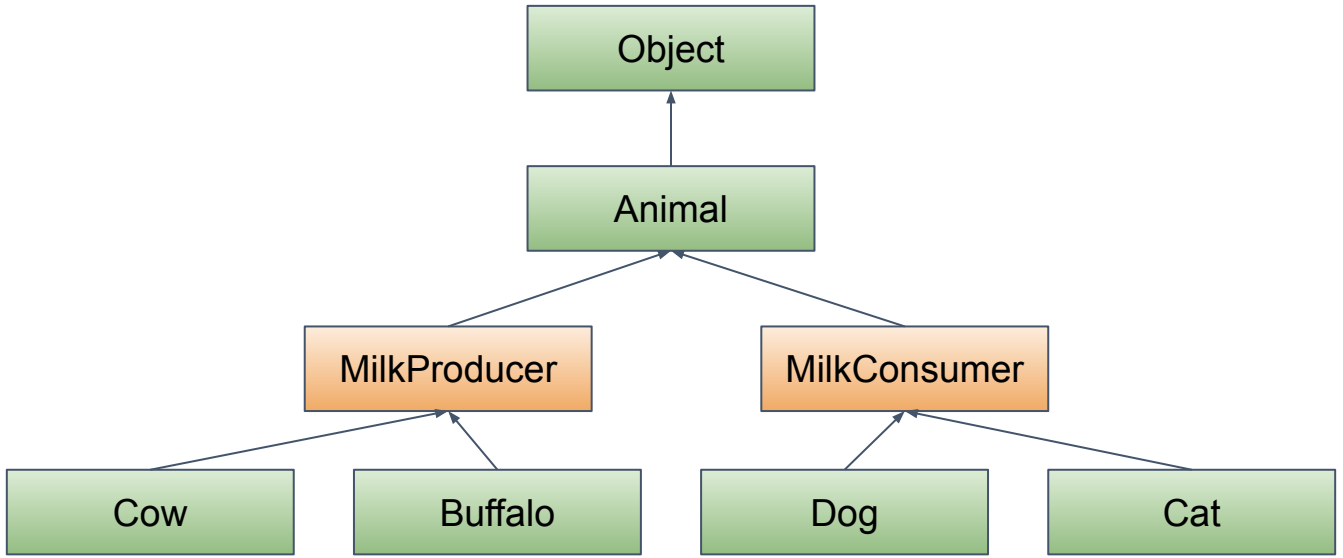
upGrad

- In the given hierarchy, we want to provide the drinkMilk() method in the Dog and Cat classes and the produceMilk() method in the Cow and Buffalo classes.
- We cannot put these methods in the Animal class. If we do this, the Cow and Buffalo classes will have the drinkMilk() method and the Dog and Cat classes will have the produceMilk() method.
- Also, we cannot simply put them in their respective classes; otherwise, we lose the flexibility provided by polymorphism and have to provide separate methods to work with different objects, such as the makeCatDrintMilk() method and the makeDogDrinkMilk() method and so on.

Polymorphism: Dynamic

upGrad

- One solution is that we can introduce two new classes, MilkProducer and MilkConsumer, in the hierarchy as shown below.



Polymorphism: Dynamic

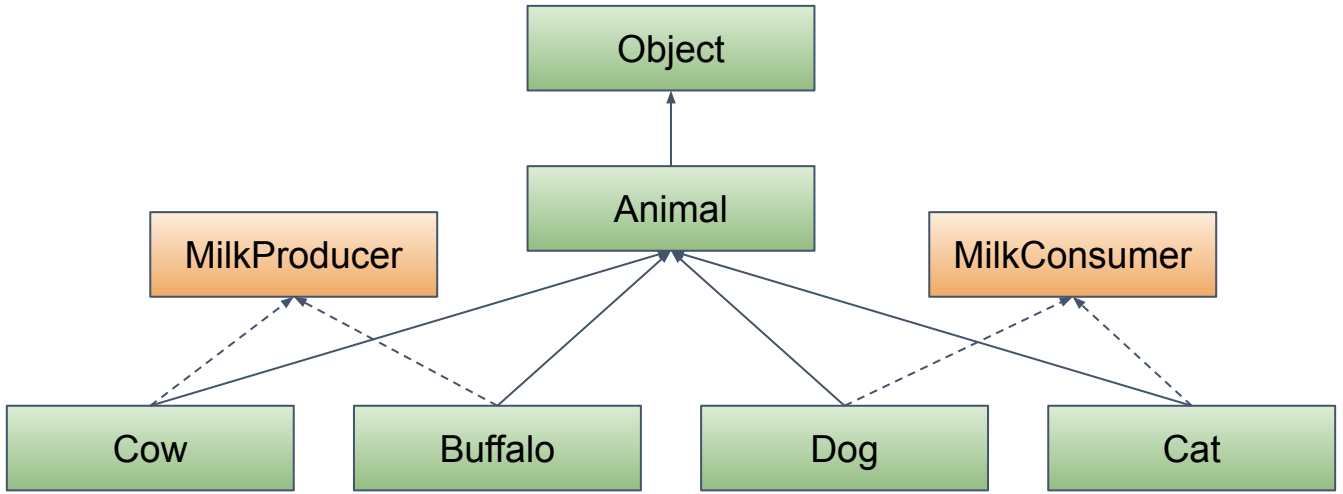
upGrad

- This approach can work in small applications but will not work in big hierarchies.
- Suppose there were 10 levels in the hierarchy, and you wanted to club some of the leaf classes that inherit methods through different paths. In this case, introducing new classes for polymorphism will not be possible.
- This is where Interface comes into the picture.
- There is a fundamental difference between a class and an interface: ***Classes help you group objects based on structure, whereas Interfaces help you group objects based on behaviour.***

Polymorphism: Dynamic

upGrad

- Let’s implement the same functionality through interfaces as follows:



Polymorphism: Dynamic

upGrad

- Let’s implement this approach in the code as follows:

```
interface MilkConsumer {
    void drinkMilk();
}

class Cat extends Animal implements MilkConsumer {
    ...
    public void drinkMilk() {
        System.out.println("Cat is drinking milk.");
    }
    ...
}
```

Polymorphism: Dynamic

upGrad

- Let's implement this approach in the code as follows:

```
public static void makeAnimalDrinkMilk(MilkConsumer milkConsumer) {
    milkConsumer.drinkMilk();
}

Cat cat = new Cat();
makeAnimalDrinkMilk(cat);
```

- As you can see, you can make any animal a milk consumer by simply implementing the interface and providing the implementation for the methods inside the interface.
- Third-party libraries also mostly use polymorphism through interfaces.

Polymorphism: Dynamic

upGrad

- Static methods can be overloaded, but they can never be overridden.
- As static methods are attached to classes, and not to objects, they are not inherited by the subclasses.
- Also, static methods are called using the class name so that the Compiler knows which method you are calling; hence, dynamic polymorphism is not possible with static methods.

Polymorphism: Advantages

upGrad

- One of the biggest advantages of polymorphism is that it helps you write flexible code that can bend in different shapes based on the requirement.
- For example, if method overloading is not allowed, then we have to name the same method differently based on the method arguments such as printlnInt(), printlnLong() and so on.

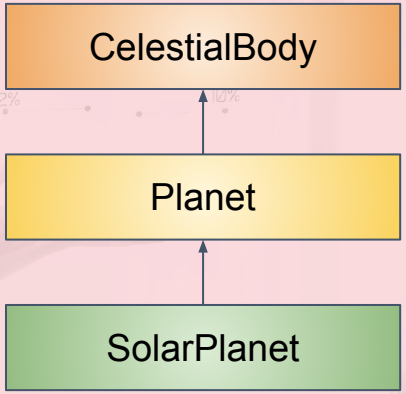
Polymorphism: Advantages

upGrad

- Similarly, if method overriding is not allowed, then we cannot develop a third-party library.
- This is because third-party library methods (such as the println() method) expects objects of the superclass.
- You can simply extend from those superclasses, override the methods that will be invoked by the third-party library method (for example, invoke the toString() method using the println() method) and pass your objects to the third-party library.

Poll 4 (15 sec)

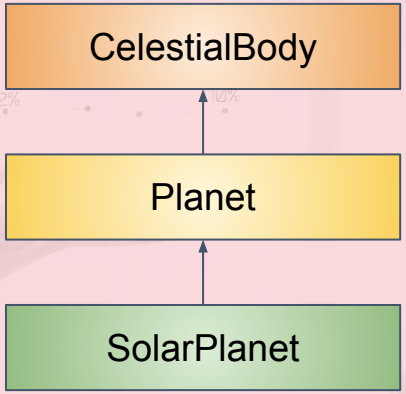
Which of the following is true regarding the class structure given below?



1. CelestialBody is a planet.
2. Planet is a SolarPlanet.
3. SolarPlanet is a CelestialBody.
4. CelestialBody is a SolarPlanet.

Poll 4 (15 sec)

Which of the following is true regarding the class structure given below?



1. CelestialBody is a planet.
2. Planet is a SolarPlanet.
3. **SolarPlanet is a CelestialBody.**
4. CelestialBody is a SolarPlanet.

Poll 5 (15 sec)

Which of the following is true with respect to method overriding?

1. We can only override private methods.
2. It is also called static polymorphism.
3. Overridden methods should have the same method signature.
4. It is also called compile-time polymorphism.

Poll 5 (15 sec)

Which of the following is true with respect to method overriding?

1. We can only override private methods.
2. It is also called static polymorphism.
3. **Overridden methods should have the same method signature.**
4. It is also called compile-time polymorphism.

upGrad

final keyword

final keyword

upGrad

- The function of the ‘final’ keyword is simple; if something is declared final, its value or definition cannot be modified.
- It can be used with attributes, methods and classes.
- If an attribute is declared final, its value cannot be changed once it is initialised.
- If a method is declared final, its definition cannot be changed once it is defined, which means that the method cannot be overridden in the subclasses.
- If a class is declared final, its definition cannot be changed once it is defined, which means that the class cannot be inherited.
- The String class is a final class; hence, you cannot extend the String class.

final keyword

upGrad

```
class User {
    private final int id;
    private final String name;

    public User(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```

Declaring final attributes

final keyword

upGrad

- When you declare something final, you have to initialise or completely define it at the same time.
- For example, if you declare an attribute final, then you have to initialise it in the same line or in the constructor.
- If you make a method or class final, then you have to define it at the same time. This means that an abstract class or method cannot be declared final.

final keyword

upGrad

- Earlier, we got an understanding of the concept of immutability and made attributes private without providing a setter method.
- This approach does not provide complete immutability, as the attributes can be changed in methods other than the setter method.
- To make a class fully immutable, we need to declare those attributes final.

Poll 6 (15 sec)

Which of the following is true with respect to final methods?

1. They cannot be overloaded.
2. They cannot be accessed outside a class.
3. They cannot be overridden.
4. They cannot be public.

Poll 6 (15 sec)

Which of the following is true with respect to final methods?

1. They cannot be overloaded.
2. They cannot be accessed outside a class.
3. **They cannot be overridden.**
4. They cannot be public.

Anonymous Class

Anonymous Class

upGrad

- Earlier, we learnt about three types of nested classes: static nested classes, inner classes and local classes.
- Now, let's learn about the fourth type of nested class, the anonymous class.
- Anonymous classes are used to create classes without any name, which is why they are called anonymous.
- Anonymous classes can be created by extending a superclass or implementing an interface and instantiating it at the same time.
- So, we will be creating a class and instantiating it at the same time. Once it is instantiated, the class cannot be used or accessed to instantiate another object.

Anonymous Class

upGrad

- Let’s understand this concept while developing a simple Calculator.
- We will be developing the calculator using the concepts of the Polymorphism, not the traditional approach.

Anonymous Class

upGrad

```
interface Operator {
    int operate(int a, int b);
}

class Addition implements Operator {
    public int operate(int a, int b) {
        return a + b;
    }
}

class Subtraction implements Operator {
    public int operate(int a, int b) {
        return a - b;
    }
}
```

Addition and Subtraction Classes

Anonymous Class

upGrad

```
public class Calculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int a = scanner.nextInt();
        int b = scanner.nextInt();

        Addition addition = new Addition();
        Subtraction subtraction = new Subtraction();

        calculate(a, b, addition);
        calculate(a, b, subtraction);

        scanner.close();
    }

    public static void calculate(int a, int b, Operator operator) {
        System.out.println(operator.operate(a, b));
    }
}
```

Instantiating Addition and Subtraction Classes

Anonymous Class

upGrad

- In the earlier example, Addition and Subtraction classes were instantiated only once.
- This is similar to writing extra code just to create an extra class.
- We can make this code compact with the help of an Anonymous class.
- The Addition class can be converted into an Anonymous class and instantiated, as shown in the next slide.

Anonymous Class

upGrad

```
Operator addition = new Operator() {
    public int operate(int a, int b) {
        return a + b;
    }
};
```

- Using the same approach, you can also create the subtraction Object.
- Here, we are creating an Anonymous class by implementing an interface, but the same approach can be used to create an Anonymous class by extending parent classes.

Poll 7 (15 sec)

When should we use an Anonymous class?

1. Whenever we want to extend a class
2. Whenever we want to implement an interface
3. Whenever we want to instantiate a subclass only once
4. Whenever we want to instantiate a nested class

Poll 7 (15 sec)

When should we use an Anonymous class?

1. Whenever we want to extend a class
2. Whenever we want to implement an interface
3. **Whenever we want to instantiate a subclass only once**
4. Whenever we want to instantiate a nested class

Poll 8 (15 sec)

Can anonymous classes extend a superclass and implement an interface at the same time?

1. Yes
2. No

Poll 8 (15 sec)

Can anonymous classes extend a superclass and implement an interface at the same time?

1. Yes
2. **No**

upGrad

Important Concepts and Questions

Important Questions

upGrad

1. What is Conversion Constructor?
2. Name a few classes that are made final by Java.
3. How is polymorphism implemented in Java? Why it is used?
4. Differentiate between method overloading and method overriding.
5. Why can’t we override static methods?
6. Can we override the main() method in Java?
7. Differentiate between implicit and explicit type conversion with real-life examples.

upGrad

Doubt Clearance Window

upGrad

Today, we learnt about the following:

1. What is Polymorphism, and why is it important?
2. Different types of Polymorphism: Static and Dynamic
3. What is Type Casting?
4. Upcasting vs Downcasting
5. What is the use of the final keyword?
6. What are final classes, attributes and methods?
7. What is an Anonymous Class, and how do you define one?

TODO:

- Provide the Operators class that contains the following four public static final attributes:
  - ***ADDITION: Operator***
  - ***SUBTRACTION: Operator***
  - ***MULTIPLICATION: Operator***
  - ***DIVISION: Operator***

  The Operator is an interface with one method, which is as follows:
  - ***+ operate(a: int, b: int) : int***

TODO:

- The different attributes of the Operators class should be declared and initialised in the same line using Anonymous class, as shown below.

```
public static final Operator ADDITION = new Operator() {
    public int operate(int a, int b) {
        ...
    }
};
```

- The name of the attribute is corresponding to the operation for which it will be used. For example, ADDITION will used to add two integers.
- These operators (different attributes of the Operators class) will be used in the main() method to calculate the result of different computations.

upGrad

Tasks to Complete After Today's Session

MCQs
Homework
Coding Questions
Project Checkpoint 5

upGrad

#RahoAmbitious

Thank You!