Advanced Programming with Java

Prativa Nyaupane

Recap

- Abstraction
- Super class, sub class, inheritance and member access
- Extends and super keyword
- Types of inheritance
- Overriding/Overloading

Today's objectives

- Final classes and methods
- Abstract classes and methods
- Upcasting and Down casting
- Interfaces and Implementation

Final Classes and Methods

1. All methods and variables can be overridden by default in subclasses.
2. To prevent subclasses from overriding the members of the superclass, we can declare them as final using the keyword final as a modifier.
3. In Java, making a class or method “final” means that it cannot be subclassed or extended by other classes.
4. final is a non-access modifier applicable only to a variable, a method or class. Final class can be used by any class, but its value cannot be modified.
    5. **Variable**: If a variable is marked final - its value cannot be changed.
    6. **Method**: If a method is marked final - it cannot be overridden.
    7. **Class**: If a class is marked final - It cannot be subclassed.

```java
public final class MathConstants {
   public static final double PI = 3.14159;
   public final double PI2 = 3.14159;
}

public class Main {
   public static void main(String[] args) {

       double piValue = MathConstants.PI; //static value belongs to class, not an instance of
class
       MathConstants mathConstants = new MathConstants();
       double piValue2 = mathConstants.PI2;//non-static value accessed using instance
   }
}


public class Shape {
   public final double calculateArea() {
       // Common implementation for calculating area
            // it cannot be overridden by any subclasses
   }
}
```

Final variable

- When a variable is declared with the final keyword, its value can’t be modified, essentially, a constant.
- This also means that you must initialize a final variable.
- If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but the internal state of the object pointed by that reference variable can be changed.
- It is good practice to represent final variables in all upperpase, using underscore to separate words.

```java
public class Example {
    // Declare a final reference variable
    final MyClass obj;

    // Constructor to initialize the final reference variable
    public Example() {
        obj = new MyClass(); // Assign a reference to a new object
    }

    // Method to modify the internal state of the object
    public void modifyObjectState() {
        obj.setSomeProperty(42); // Modifying a property of the object
    }
}

class MyClass {
    private int someProperty;

    public void setSomeProperty(int value) {
        this.someProperty = value;
    }
}
```

```
public class GameConfig {
    public final int MAX_PLAYERS;

    public GameConfig(int maxPlayers) {
        MAX_PLAYERS = maxPlayers;
    }
}
```

Imagine you're working on a class representing a configuration for a game, and you have a setting that determines the maximum number of players allowed. You want to make sure this setting doesn't change after it's been set. You can declare it as final:

Initializing a final variable

There are three ways to initialize a final variable when it is declared.

1. **Direct Initialization:**

This is the most common method, where the final variable is initialized at the time of declaration. Once initialized, its value cannot be changed.

Example: `final int MAX_VALUE = 100;`

1. **Initialization in instance initializer block:**

If the final variable's value needs to be calculated based on complex logic or needs to be initialized dynamically, it can be initialized in an instance initializer block.

This block is executed every time an instance of the class is created.

Example:

Initializing a final variable contd..

```
final int MAX_VALUE;
{
// Complex logic to calculate the value
MAX_VALUE = calculateMaxValue();
}
```

3. Initialization in Constructor:

Final variables can also be initialized in a constructor. This allows each instance of the class to have a different initial value for the final variable.

Example:

```
final int MAX_VALUE;

public MyClass() {
   // Initialize MAX_VALUE in the constructor
   MAX_VALUE = 100;
}
```

Final Class

- When a class is declared with final keyword, it is called a final class.
- A final class cannot be extended(inherited).
- An abstract class cannot be final class, because we can only create a final class if it is complete. Abstract class is not considered complete because, it may contain abstract methods which have no implementation.
- There are two uses of a final class:
    a. Usage 1: One is to prevent inheritance, as final classes cannot be extended.
    b. Usage 2: The other use of final with classes is to create an immutable class lik the predefined String class.

Final Methods

- When a method is declared with final keyword, it is called a final method.
- A final method cannot be overridden.
- We must declare methods with the final keyword for which we are required to follow the same implementation throughout all the derived class.

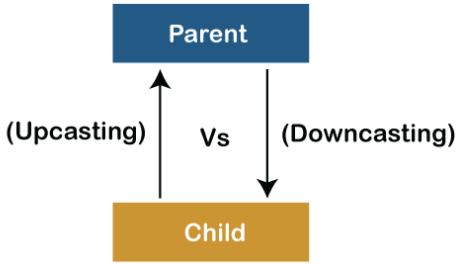
Upcasting and Downcasting

- A process of converting one data type to another is known as Typecasting.
- Upcasting and Downcasting is the type of object typecasting.
- In Java, the object can also be typecasted like the datatypes.
- So, there are two types of typecasting possible for an object, i.e., Parent to Child and Child to Parent or can say Upcasting and Downcasting.
- Typecasting is used to ensure whether variables are correctly processed by a function or not.

Upcasting and Downcasting contd...

In Upcasting and Downcasting, we typecast a child object to a parent object and a parent object to a child object simultaneously.

We can perform Upcasting implicitly or explicitly, but downcasting cannot be done implicitly.



S.No	Upcasting	Downcasting
1.	A child object is typecasted to a parent object.	The reference of the parent class object is passed to the child class.
2.	We can perform Upcasting implicitly or explicitly.	Implicitly Downcasting is not possible.
3.	In the child class, we can access the methods and variables of the parent class.	The methods and variables of both the classes(parent and child) can be accessed.
4.	We can access some specified methods of the child class.	All the methods and variables of both classes can be accessed by performing downcasting.
5.	Parent p = new Parent()	Parent p = new Child() Child c = (Child)p;

```java
class Animal {
  void makeSound() {
    System.out.println(x:"Generic animal sound");
  }
}
```

You, 1 second ago | 1 author (You)

```java
class Dog extends Animal {

  @Override
  void makeSound() {
    System.out.println(x:"Generic dog sound");
  }

  void bark() {
    System.out.println(x:"Woof! Woof!");
  }
}
```

```java
public class CastingExample {
    Run | Debug
    public static void main(String[] args) {
        // Upcasting: Dog to Animal
        Animal animal = new Dog();
        animal.makeSound(); // Calls Dog's overridden method
        // animal.bark(); // Error: bark() is not a method of Animal

        // Downcasting: Animal to Dog
        if (animal instanceof Dog) {
            Dog dog = (Dog) animal;
            dog.makeSound();
            dog.bark();
        } else {
            System.out.println(x:"Cannot downcast to Dog.");
        }
    }
}
```

In this example:

- Dog is a subclass of Animal.
- makeSound() is a method in both Anial and Dog, but bark() is specific to Dog.
- The main method demonstrates upcasting by creating a Dog object and assigning it to an Animal reference.
- While upcasted, the Animal reference can only access methods declared in the Animal class.
- To access bark() (a Dog specific method), downcasting is used. We check if an object is instance of Dog before downcasting to avoid runtime errors.
- If the object is a Dog, we perform the downcast and access both the overridden makeSound() and bark() methods.

Interfaces and Implementation

In Java, an interface is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields or constructors.

```
interface MyInterface {
  public void interfaceFunction();

  public void print();
}
```

```java
class ChildClassInt implements MyInterface {

  public ChildClassInt(int value) {
    this.print();
  }

  @Override
  public void print() {
    System.out.println(x:"Printing the Value in overridden method from interface class");
  }

  @Override
  public void interfaceFunction() {
    System.out.println(x:"Printed on the overridden function of interface class.");
  }
}
```