Inheritance and Polymorphism

- Title: Inheritance and Polymorphism
- Goals:
  - Understand how inheritance promotes code reuse
  - Learn how to create and use subclasses
  - Apply polymorphism for flexible code
  - Understand dynamic binding and casting

Introduction to Inheritance

- One of the fundamental principles of Object-Oriented Programming (OOP).
- Allows a new class (subclass) to be based on an existing class (superclass).
- Promotes code reuse and establishes a hierarchical "is-a" relationship.
- Inheritance = acquiring properties from another class
- Promotes code reuse and organization
- Java uses extends keyword

Motivations for Inheritance

- Avoids redundant code.
- Facilitates code maintenance and modification.
- Supports the modeling of real-world hierarchies (e.g., a Student is a Person).
- Enables polymorphism (next major topic).

Superclasses and Subclasses

- Superclass (Parent/Base Class): The existing class from which a new class is derived.
- Subclass (Child/Derived Class): The new class that inherits from a superclass.
- A subclass extends the functionality of its superclass, often adding more specific data fields and methods.

Defining a Subclass (extends keyword)

- In Java, inheritance is achieved using the extends keyword.
- Syntax: class Subclass extends Superclass { ... }
- Example: public class Circle extends GeometricObject { ... }

What is Inherited?

- A subclass inherits all **public** and **protected** data fields and methods from its superclass.
- Private members of the superclass are not directly accessible by the subclass, but they are still part of the superclass object and can be accessed indirectly through public/protected methods of the superclass.
- Constructors are NOT inherited.

The Object Class: The Root of All Classes

- Every class in Java, explicitly or implicitly, extends the java.lang.Object class.
- If a class doesn't explicitly extend another class, it implicitly extends Object.
- Object provides common functionalities (e.g., equals(), toString(), hashCode())

Constructor Chaining

- When a subclass object is created, its constructor is invoked.
- Before the subclass constructor's body executes, a superclass constructor must be called. This forms a "chain" of constructor calls up the hierarchy.
- This ensures that the superclass's portion of the object is properly initialized.

Invoking Superclass Constructors (super() keyword)

- Use super() to explicitly invoke a superclass constructor.
- super() must be the first statement in a subclass constructor.
- If super() is not explicitly called, the Java compiler implicitly inserts a call to the superclass's no-arg constructor as the first statement. If the superclass does not have a no-arg constructor, this will result in a compile-time error.

Example: Constructor Chaining

```
class Person {
    private String name;
    public Person(String name) {
        this.name = name;
        System.out.println("Person constructor: " + name);
    }
}

class Student extends Person {
    private int studentId;
    public Student(String name, int id) {
        super(name); // Must be the first statement
        this.studentId = id;
        System.out.println("Student constructor: " + name + ", " + id);
    }
}
// When new Student("Alice", 123) is called, Person's constructor runs first.
```

Method Overriding

- A subclass can provide a specific implementation for a method that is already defined in its superclass.
- The method signature (name and parameter list) must be identical.
- The return type must be the same or a covariant (sub)type.
- The access modifier cannot be more restrictive than the superclass method.

@Override Annotation

- **Recommended** to use the @Override annotation when overriding methods.

- The compiler checks if the annotated method actually overrides a superclass method. If not, it's a **compile-time error**, helping catch common mistakes (e.g., typos in method names).

Invoking Superclass Methods (super keyword)

- The super keyword can also be used to explicitly call an overridden method from the superclass.
- Useful when you want to extend the superclass's behavior rather than completely replacing it.
- Syntax: super.methodName(arguments);

Overriding vs. Overloading

- Overriding: Same method signature (name + parameter list) in subclass as in superclass. (Runtime polymorphism)

- Overloading: Same method name in the same class (or inherited classes), but different parameter lists. (Compile-time polymorphism)

Introduction to Polymorphism

- Polymorphism: "Many forms." The ability of an object to take on many forms.
- A single reference variable can refer to objects of different (related) types at different times.
- Central to object-oriented design, allowing flexible and extensible code.

Polymorphic References

- A reference variable of a superclass type can refer to an object of any of its subclasses.
- Example:
  GeometricObject obj = new Circle(5);
  GeometricObject obj2 = new Rectangle(4, 6);

- While obj is declared as GeometricObject, it actually holds a Circle object.

Dynamic Binding

- The JVM determines which version of an overridden method to execute at runtime, based on the actual type of the object being referenced, not the type of the reference variable.
- This is also known as runtime polymorphism.
- Example: If obj (referring to a Circle) calls obj.getArea(), the Circle's getArea() method is executed.

Casting Objects

- Upcasting (Implicit): Assigning a subclass object to a superclass reference. Always safe.
- Object obj = new String("Hello");

- Downcasting (Explicit): Assigning a superclass reference to a subclass reference. Requires an explicit cast and is potentially unsafe if the object isn't truly of that subclass type.
- String s = (String) obj;

The instanceof Operator

- Used to check if an object is an instance of a particular class (or an interface it implements).
- Prevents ClassCastException during downcasting.
- Syntax:
  objectRef instanceof ClassOrInterface
- Returns true or false.

instanceof Example

```
Object myObject = new Circle(3);
if (myObject instanceof Circle) {
    Circle c = (Circle) myObject; // Safe downcast
    System.out.println("It's a circle with radius: " + c.getRadius());
} else if (myObject instanceof String) {
    // This block won't execute
}
```

The equals() Method in Object Class

- The default equals() method (from Object) simply compares object references (memory addresses).
- For meaningful content comparison, you almost always need to override equals() in your custom classes.
- Consider instanceof and proper type casting within the method.

The toString() Method

- Returns a string representation of the object.
- Default Object.toString() returns class name and hash code.
- Frequently overridden to provide a concise, human-readable description of the object's state (e.g., a Circle object showing its radius).

Array of Objects and Polymorphism

- An array of a superclass type can hold objects of its subclasses.

- Example:
  GeometricObject[] objects = new GeometricObject[3];

  objects[0] = new Circle(1.0);

  objects[1] = new Rectangle(2.0, 3.0);

  objects[2] = new Circle(4.5);

- Allows processing diverse objects uniformly in a loop.

ArrayList of Objects and Polymorphism

- ArrayList (from java.util) is a dynamic, resizable array.
- Even more flexible for storing polymorphic collections than standard arrays.
- Example:

```
ArrayList<GeometricObject> shapes = new ArrayList<>();
shapes.add(new Circle(5));
shapes.add(new Rectangle(2, 3));
```

protected Modifier Revisited

- Allows members to be accessible within the same package and by subclasses in any package.
- More restrictive than public, less restrictive than private.
- Often used for fields or methods that subclasses might need to access or modify directly.

The final Modifier

- Can be applied to:
  - **Variables**: Makes them constants (value cannot be changed after initialization).
  - **Methods**: Prevents the method from being **overridden** by subclasses.
  - **Classes**: Prevents the class from being **extended** (no subclasses allowed, e.g., String, Math classes).

Designing with Inheritance

- Use inheritance when there is a clear "is-a" relationship.
- Look for common attributes and behaviors to abstract into a superclass.
- Favor composition over inheritance if the relationship is "has-a" rather than "is-a".
- Keep hierarchies shallow and focused.

Key Concepts from Chapter 11

- Inheritance: extends keyword, superclass/subclass, code reuse.
- Constructor Chaining: super() call, automatic invocation of no-arg superclass constructor.
- Method Overriding: Redefining superclass methods in subclasses, @Override.
- Polymorphism: Superclass references to subclass objects, instanceof.
- Dynamic Binding: Runtime determination of method calls.
- Object class methods: equals(), toString() (and their importance for overriding).
- protected and final modifiers.

Conclusion

- Inheritance and polymorphism are cornerstones of robust, flexible, and scalable object-oriented programming.
- They allow for code reuse, extensibility, and the ability to work with objects at different levels of abstraction.
- Mastering these concepts is crucial for advanced Java development.