CMPE 101
Object Oriented Programming

İstanbul
Bilgi University

Dr. Emel Küpçü

Department of Computer Engineering

İstanbul Bilgi University

Week-10: Polymorphism, Abstract Classes and Interfaces

Object-Oriented Programming (OOP) in Java

- Java is built around OOP principles, making it easy to design modular and reusable code.
- The core principles of OOP in Java are:
  - **Encapsulation**: hiding data inside a class and controlling access through methods.
  - **Inheritance**: Allowing a class to inherit properties and behavior from another class.
  - **Polymorphism**: Allowing objects to behave differently based on their context.
  - **Abstraction**: Hiding implementation details and exposing only the functionality.

10.1 Introduction

- Polymorphism
  - Enables you to "program in the *general*" rather than "program in the *specific*."
  - Polymorphism enables you to write programs that process objects that share the same superclass as if they were all objects of the superclass; this can simplify programming.
- **Example**: Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes `Fish`, `Frog` and `Bird` represent the three types of animals under investigation.
  - Each class extends superclass `Animal`, which contains a method **`move`** and maintains an animal's current location as *x*-*y* coordinates. Each subclass implements method `move`.
  - A program maintains an `Animal` array containing references to objects of the various `Animal` subclasses. To simulate the animals' movements, the program sends each object the same message once per second—namely, `move`.

10.1 Introduction (Cont.)

- Each specific type of `Animal` responds to a `move` message in a unique way:
  - a `Fish` might swim three feet
  - a `Frog` might jump five feet
  - a `Bird` might fly ten feet.
- The program issues the same message (i.e., `move`) to each animal object, but each object knows how to modify its x-y coordinates appropriately for its specific type of movement.
- **Relying on each object to know how to "do the right thing" in response to the same method call is the key concept of polymorphism.**
- The same message sent to a variety of objects has "many forms" of results—hence the term polymorphism.

10.1 Introduction (Cont.)

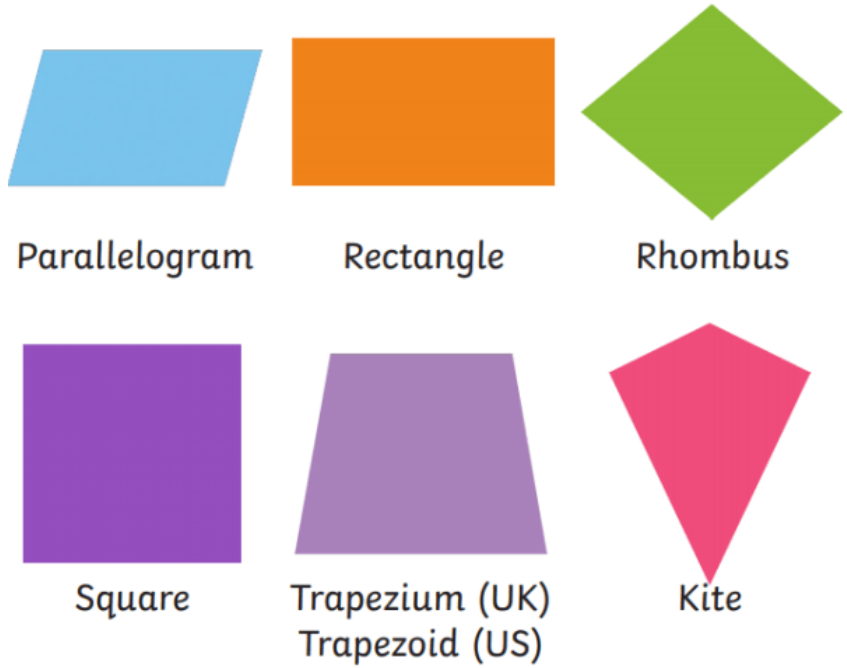
- With polymorphism, we can design and implement systems that are easily *extensible*
  - New classes can be added with little or no changes to the main parts of the program, as long as they belong to the existing inheritance hierarchy.
  - The new classes simply "fit in" without major adjustments.
  - Only the parts of the program that directly interact with the new classes need to be updated.

10.2 Polymorphism Examples

- Example: Quadrilaterals

A quadrilateral is a polygon with four sides, four vertices, and four angles.

- If `Rectangle` is derived from `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral`.
- Any operation that can be performed on a `Quadrilateral` can also be performed on a `Rectangle`.
- These operations can also be performed on other `Quadrilaterals`, such as `Squares`, `Parallelograms` and `Trapezoids`.
- Polymorphism occurs when a program invokes a method through a superclass `Quadrilateral` variable—at execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable.



https://www.twinkl.com.tr/teaching-wiki/quadrilateral-shape

Polymorphism

- It allows objects of different classes to be treated as objects of a common superclass.
- Polymorphism means "many forms" and refers to the **ability of a single function, method, or operator to behave differently depending on the type of object it is interacting with**.
- It enables one interface to be used for a variety of actions, making the code more flexible and easier to extend.
  - Compile-Time Polymorphism
  - Run-Time Polymorphism

Compile-Time Polymorphism- Method Overloading

- **Compile-Time Polymorphism** is achieved through **method overloading.**
- **Method overloading** allows multiple methods in the same class to have the same name but with different parameters (type, number, or order).

```
class Film {
    // Method to display a film with only a name
    void displayFilm(String name) {
        System.out.println("Film Name: " + name);
    }
    // Overloaded method to display a film with a name and release year
    void displayFilm(String name, int releaseYear) {
        System.out.println("Film Name: " + name + ", Release Year: " + releaseYear);
    }
    // Overloaded method to display a film with a name, release year, and genre
    void displayFilm(String name, int releaseYear, String genre) {
        System.out.println("Film Name: " + name + ", Release Year: " + releaseYear + ", Genre: " + genre);
    }
}
public class Main_Film {
    public static void main(String[] args) {
        Film film = new Film();

        // Call different overloaded methods
        film.displayFilm("Inception"); // Calls the method with one parameter
        film.displayFilm("The Godfather", 1972); // Calls the method with two parameters
        film.displayFilm("The Dark Knight", 2008, "Action"); // Calls the method with three parameters
    }
}
```

Output:

```
Film Name: Inception
Film Name: The Godfather, Release Year: 1972
Film Name: The Dark Knight, Release Year: 2008, Genre: Action
```

Run-Time Polymorphism - Method Overriding

- **Run-Time Polymorphism** is resolved at **runtime**. It is achieved through method overriding.
- **Method Overriding** allows a subclass to provide a specific implementation of a method that is already defined in its parent class.
- **The method** in the child class should have **the same name, return type, and parameters** as the **method in the parent class**.
- The **@Override** annotation is optional but recommended.
  - It ensures that the method in the child class correctly overrides a method in the parent class.

Example - Method Overriding

```java
class House {
    void displayInfo() {
        System.out.println("This is a generic house.");
    }
}
```

Base Class

```java
class Villa extends House {
    @Override
    void displayInfo() {
        System.out.println("This is a villa with a garden.");
    }
}
```

Child Class

```java
class Apartment extends House {
    @Override
    void displayInfo() {
        System.out.println("This is an apartment in a building.");
    }
}
```

Child Class

```java
public class Main_House {
    public static void main(String[] args) {
        House villa = new Villa();
        House apartment = new Apartment();

        villa.displayInfo();
        apartment.displayInfo();
    }
}
```

Main Class

Output:

This is a villa with a garden.

This is an apartment in a building.

Example - Method Overriding (cont.)

The statement **House villa = new Villa();** in Java demonstrates the concept of **polymorphism**, where a parent class reference is used to point to a child class object.

- **Villa villa = new Villa();**
  - You can access **all methods** and fields from **both** the House class (parent) and the Villa class (child).
- **House villa = new Villa();**
- You can only access **methods and fields defined in the House class** because the reference type is House. However**, the overridden methods from Villa will still run.**

Example - Method Overriding (cont.)

House myHouse = new Villa();

- you are using the House type for the variable.
- This means your code only knows it's a House, not specifically a Villa.
- Later, if you want to change Villa to, say, Apartment, you only change the creation part:
- **myHouse = new Apartment();**
- but the rest of the code stays the same, because it works with House methods, not Villa or Apartment specific ones!

- Writing **House villa = new Villa();** shows **full power of polymorphism**.
  - You can later **swap** the object with **any other subclass of House** (like Apartment) easily. Your code stays flexible and generic.
- Writing **Villa villa = new Villa(); is still polymorphism**, but **only specific to Villa**.

10.3 Demonstrating Polymorphic Behavior

- In Java, **a reference of a superclass can point to an object of a subclass**. This is known as **polymorphism**.
- When you call a method using that reference, **the actual method that gets executed depends on the type of the object**, **not** the type of the reference variable.
- This means:
  - You can write general code using the superclass type.
  - At runtime, the correct method from the subclass is called.
  - This allows you to, for example, store different subclass objects in a single array of superclass type.
- A superclass object cannot be treated as a subclass object, because a superclass object is not an object of any of its subclasses.
- The is-a relationship applies only up the hierarchy from a subclass to its direct (and indirect) superclasses, and not down the hierarchy.

Example: Polymorphism

```
class AllAnimal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dogs extends AllAnimal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cats extends AllAnimal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }
}
```

```
public class MainDogCatAnimal {
    public static void main(String[] args) {
        // Superclass reference to subclass objects
        AllAnimal a1 = new Dogs();
        AllAnimal a2 = new Cats();

        a1.makeSound();  // Output: Dog barks
        a2.makeSound();  // Output: Cat meows

        // Array of superclass references
        AllAnimal[] animals = {new Dogs(), new Cats()};

        for (AllAnimal animal : animals) {
            animal.makeSound();  // Output: Dog barks, then
Cat meows
        }
    }
}
```

Output:
Dog barks
Cat meows
Dog barks
Cat meows

Dynamic Binding

```
class Animal {
    void sound() {
        System.out.println("Some sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Bark");
    }
}
public class Test {
    public static void main(String[] args) {
        // Animal reference, Animal object
        Animal myAnimal = new Animal();

        // Animal reference, Dog object
        Animal myDog = new Dog();
        myAnimal.sound();  // Output: Some sound
        myDog.sound();     // Output: Bark
    }
}
```

Dynamic binding, also known as **late binding**, is the process where the method that gets invoked is determined at runtime, based on the actual type of the object, not the reference type.

At compile-time, both myAnimal and myDog are of type Animal, so the compiler sees them as Animal references.
At runtime, the **actual object** (whether it's an Animal or Dog) determines which sound() method is called. For myDog, the sound() method in the Dog class is invoked, even though the reference is of type Animal.

This process of determining the method at runtime based on the actual object type (not the reference type) is **dynamic binding**.

Group Discussion Question

In a company, there are employees with two roles: Developer and Intern. Each employee has a name and performs a specific task based on their role. You are required to implement this scenario in Java using **inheritance** and **polymorphism**.

Tasks:

Create a base class Employee that contains:

1. A String attribute name (the employee's name).
2. A constructor that initializes the name attribute.
3. A method work() that prints a general message like "Employee is working". This method will be overridden in the subclasses.

Create two subclasses:

1. Developer: Inherits from Employee. This class should override the work() method and print a message like "Developer is writing code and debugging software."
2. Intern: Inherits from Employee. This class should override the work() method and print a message like "Intern is assisting with tasks and learning from senior employees."

Create the following objects:

1. A Developer object with the name "Bob".
2. An Intern object with the name "Charlie".

Store these objects in an array of Employee references (since they are both employees, even though they are of different types).

Iterate through the array and for each Employee object, call the work() method

Example Output:
Bob is writing code and debugging software. Charlie is assisting with tasks and learning from senior employees.

Abstraction

- **Abstraction** is one of the key concepts in OOP
- It refers to the **process of hiding the implementation details** and **showing only the essential features of an object or system**.
- In **Java**, abstraction is achieved through:
  - **Abstract classes**
    - They cannot be instantiated on their own and may contain both **abstract methods** (methods without implementation) and **concrete methods** (methods with implementation).
  - **Interfaces**
    - These are used to specify what methods a class should implement without defining how they should be implemented.
- It helps in **reducing complexity by only exposing relevant information** and **hiding unnecessary details**.

Advantages of Abstraction in Java

- **Simplifies Code:**
  Focus on essential features without worrying about background details.
- **Improves Maintainability:**
  Easier to update or fix code without affecting other parts of the system.
- **Supports Code Reusability:**
  Abstract classes and interfaces promote building reusable frameworks.
- **Enables Flexible and Scalable Design:**
  Programs can be easily extended with new features or classes.
- **Reduces Complexity:**
  Allows developers to focus on important concepts without worrying about complex code.

10.4 Abstract Classes and Methods

- Abstract classes
  - Sometimes it's useful to declare classes for which you never intend to create objects.
  - Used only as superclasses in inheritance hierarchies, so they are sometimes called abstract superclasses.
  - Cannot be used to instantiate objects—abstract classes are *incomplete*.
  - Subclasses must declare the "missing pieces" to become "concrete" classes, from which you can instantiate objects; otherwise, these subclasses, too, will be abstract.
- An abstract class provides a superclass from which other classes can inherit and thus share a common design.

10.4 Abstract Classes and Methods (Cont.)

- Classes that can be used to instantiate objects are called concrete classes.
- Such classes provide implementations of every method they declare (some of the implementations can be inherited).
- Abstract superclasses are too general to create real objects—they specify only what is common among subclasses.
- Concrete classes provide the specifics that make it reasonable to instantiate objects.
- Not all hierarchies contain abstract classes.

10.4 Abstract Classes and Methods (Cont.)

- You make **a class abstract** by declaring it with keyword **`abstract`**.
- An abstract class normally contains one or more **abstract methods**.
  - **An abstract method** is an instance method with keyword `abstract` in its declaration, as in

```
public abstract void draw(); // abstract method
```

- **Abstract methods do not provide implementations**.
- **A class that contains abstract methods must be an abstract class** even if that class contains some concrete (nonabstract) methods.
- **Each concrete subclass of an abstract superclass also must provide concrete implementations of each of the superclass's abstract methods.**
- Constructors and `static` methods cannot be declared `abstract`.

Example - Abstract Classes

Abstract Class

```
abstract class Computer {

    void display() {
        System.out.println("Displaying content " +
                "on the screen.");
    }

abstract void turnOn();
}
```

Concrete Method

Abstract Method

```
class Laptop extends Computer {
    @Override
    void turnOn() {
        System.out.println("Laptop is " +
                "booting up.");
    }
}
```

Subclass

```
class Desktop extends Computer {
    @Override
    void turnOn() {
        System.out.println("Desktop is " +
                "starting up.");
    }
}
```

Subclass

```
public class Main_Computer {
    public static void main(String[] args) {
        Computer myLaptop = new Laptop();
        myLaptop.display();
        myLaptop.turnOn

        Computer myDesktop = new Desktop();
        myDesktop.display();
        myDesktop.turnOn();
    }
}
```

Output:
Displaying content on the screen.
Laptop is booting up.
Displaying content on the screen.
Desktop is starting up.

Common Programming Error 10.1

Attempting to instantiate an object of an abstract class is a compilation error.

10.9 Creating and Using Interfaces

- Interfaces offer a capability requiring that unrelated classes implement a set of common methods.
- Interfaces define and standardize the ways in which things such as people and systems can interact with one another.
- A Java interface describes a set of methods that can be called on an object.
- An interface declaration begins with the keyword `interface.`

```
interface InterfaceName {// Define the interface
   // Abstract method declarations (no body)
   returnType method1();
   returnType method2(parameterType param);
}
```

10.9 Creating and Using Interfaces (Cont.)

- You don't need to (but can) explicitly write public. All methods are automatically public and abstract.

```
interface Vehicle {
    void move(); // Automatically public and abstract: public abstract void move();
}
```

- **All fields** are public static final — no need to explicitly write it.
  - This means **constant values only** — no regular instance variables!

```
interface Constants {
    int MAX_SPEED = 100; // Automatically public static final :  public static final int MAX_SPEED = 100;
}
```

- **You can't have constructors** or instance fields in interfaces.
  - Interfaces **cannot have constructors** because you cannot create objects directly from interfaces.
  - Interfaces **cannot have normal instance fields** (like private int x) because there's no object to hold them — **only constants are allowed**.

```
interface Example {
    // int x;        // ✗ Error: cannot have instance fields
    // Example() {}  // ✗ Error: interfaces cannot have constructors
}
```

10.9 Creating and Using Interfaces (Cont.)

- Example: In a "basic-driving-capabilities" interface consisting of a steering wheel, an accelerator pedal and a brake pedal would enable a driver to tell the car *what* to do
  - Once you know how to use this interface for turning, accelerating and braking, you can drive many types of cars, even though manufacturers may *implement* these systems *differently*
  - e.g., there are many types of braking systems—disc brakes, drum brakes, antilock brakes, hydraulic brakes, air brakes and more. When you press the brake pedal, your car's actual brake system is irrelevant—all that matters is that the car slows down when you press the brake.

10.9 Creating and Using Interfaces (Cont.)

- To use an interface, a concrete class must specify that it **`implements`** the interface and must declare each method in the interface with specified signature.
  - Add the **`implements`** keyword and the name of the interface to the end of your class declaration's first line.

```
public class MyClass implements MyInterface { ... }
```

- A class that does not implement all the methods of the interface is an abstract class and must be declared `abstract`.
  - If a class *does not* implement all required interface methods, it **must** be marked abstract.

General Format of an Interface

```
interface InterfaceName {// Define the interface
    // Abstract method declarations (no body)
    returnType method1();
    returnType method2(parameterType param);
}

// Define a concrete class that implements the interface
public class ClassName implements InterfaceName {

    // Implement all methods from the interface
    @Override
    public returnType method1() {
        // method body
    }
    @Override
    public returnType method2(parameterType param) {
        // method body
    }
    // You can also add your own methods and fields
}
```

Common Programming Error 10.6

In a concrete class that `implements` an interface, failing to implement any of the interface's `abstract` methods results in a compilation error indicating that the class must be declared `abstract`.

Example - Interfaces

```
interface Job {
    void performDuty();
    int calculateWorkingHours();
}
```

Abstract Methods

Interface

```
class Engineer implements Job {
    @Override
    public void performDuty() {
        System.out.println("Engineer is " +
        "designing projects.");
    }
    @Override
    public int calculateWorkingHours() {
        int dailyHours = 9;
        int workingDays = 25;
        // Total hours worked in a month
        return dailyHours * workingDays;
    }
}
```

Overridden Method

Overridden Method

Output:

```
Engineer is designing projects.
Engineer's total working hours: 225
Teacher is conducting a class.
Teacher's total working hours: 120
```

```
class Teacher implements Job {
    @Override
    public void performDuty() {
        System.out.println("Teacher is conducting a class.");
    }
    @Override
    public int calculateWorkingHours() {
        int dailyHours = 6;
        int workingDays = 20;
        return dailyHours * workingDays
    }
}
```

Overridden Method

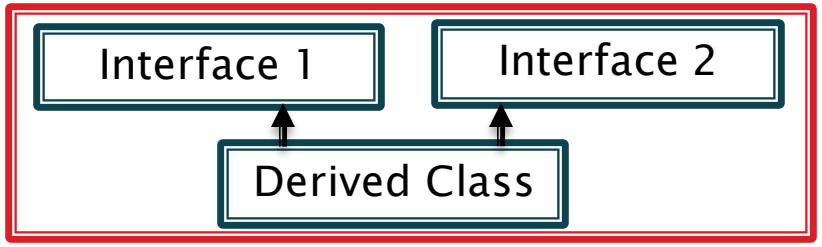
Overridden Method

```
public class Main_Jobs {
    public static void main(String[] args) {
        Job engineer = new Engineer();
        engineer.performDuty();
        System.out.println("Engineer's total working hours: "
                + engineer.calculateWorkingHours());

        Job teacher = new Teacher();
        teacher.performDuty();
        System.out.println("Teacher's total working hours: "
                + teacher.calculateWorkingHours());
    }
}
```

10.9.2 Interface (cont.)

- Interface methods are always `public` and `abstract`, so they do not need to be declared as such.
- Interfaces can have any number of methods.
- Interfaces may also contain `static` and `final` constants.
- Java does not allow subclasses to inherit from more than one superclass, but it allows a class to inherit from one superclass and implement as many interfaces as it needs.



- To implement more than one, use a comma-separated list of interface names after keyword `implements` in the class declaration, as in:

```
public class ClassName extends SuperclassName
   implements FirstInterface, SecondInterface, …
```

Example - Multiple inheritance with Interfaces

```
interface CuttingTool {
    void cut();
}
```

```
interface MeasuringTool {
    void measure(); /
}
```

Interfaces

```
class MultiTool implements CuttingTool, MeasuringTool {
    @Override
    public void cut() {
        System.out.println("MultiTool is cutting materials.");
    }
    @Override
    public void measure() {
        System.out.println("MultiTool is measuring
dimensions.");
    }
}
```

Derived class implementing functions of Interfaces

```
public class Main_Multitool {
    public static void main(String[] args) {
        MultiTool tool = new MultiTool();
        tool.cut(); // Calls the method from
CuttingTool interface
        tool.measure(); // Calls the method from
MeasuringTool interface
    }
}
```

Output:
MultiTool is cutting materials.
MultiTool is measuring dimensions.

Questions?