Abstract Classes and Interfaces

- Abstraction: Focusing on the "what" rather than the "how." Hiding implementation details and showing only essential features.

- In OOP, abstraction is achieved through:
  - Abstract Classes: Partially implemented classes.
  - Interfaces: Pure contracts defining behavior.

- Both are crucial for designing flexible, extensible, and maintainable systems.

What are Abstract Classes?

- A class that cannot be instantiated directly (you cannot create an object of an abstract class).
- Designed to be a superclass from which other classes will inherit.
- Can contain a mix of abstract methods (no implementation) and concrete methods (with implementation).
- Declared with the abstract keyword.

Why Use Abstract Classes?

- Provide a common base: Define a common interface and some shared implementation for a family of related classes.
- Enforce implementation: Ensure that subclasses provide concrete implementations for certain methods defined in the abstract superclass (abstract methods).
- Model partial concepts: Represent concepts that are not fully concrete on their own (e.g., a Shape is an idea, but a Circle or Rectangle is a concrete shape).

Abstract Class Syntax

```
public abstract class ClassName { // Declared as abstract
    // Data fields (can be concrete or final)
    private String name;
    // Constructors (can have them, but not inherited)
    public ClassName(String name) {
        this.name = name;
    }
    // Concrete method (with implementation)
    public String getName() {
        return name;
    }
    // Abstract method (no implementation, ends with semicolon)
    public abstract double getArea(); // Must be implemented by concrete subclasses

    // Another abstract method
    public abstract void printInfo();
}
```

Abstract Methods

- A method declared without an implementation (no method body, ends with a semicolon).
- Must be declared in an abstract class.
- Any concrete (non-abstract) subclass of an abstract class must provide an implementation for all inherited abstract methods, or it too must be declared abstract.

Example: GeometricObject Abstract Class

```
// GeometricObject.java
public abstract class GeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;

    protected GeometricObject() { // Constructor
        dateCreated = new java.util.Date();
    }

    protected GeometricObject(String color, boolean filled) { // Overloaded constructor
        this(); // Call no-arg constructor
        this.color = color;
        this.filled = filled;
    }

    public String getColor() { return color; }
    public void setColor(String color) { this.color = color; }
    public boolean isFilled() { return filled; }
    public void setFilled(boolean filled) { this.filled = filled; }
    public java.util.Date getDateCreated() { return dateCreated; }

    @Override
    public String toString() {
        return "created on " + dateCreated + "\ncolor: " + color + " and filled: " + filled;
    }

    // Abstract methods: must be implemented by concrete subclasses
    public abstract double getArea();
    public abstract double getPerimeter();
}
```

Concrete Subclass of an Abstract Class

- A non-abstract class that extends an abstract class.
- Must provide concrete implementations for all abstract methods inherited from its abstract superclass.
- Can define its own specific data fields, constructors, and methods.

Example: Circle Subclass of GeometricObject

```
// Circle.java
public class Circle extends GeometricObject {
    private double radius;
    public Circle() { } // No-arg constructor
    public Circle(double radius) {
        this.radius = radius;
    }
    public Circle(double radius, String color, boolean filled) {
        super(color, filled); // Call superclass constructor
        this.radius = radius;
    }
    public double getRadius() { return radius; }
    public void setRadius(double radius) { this.radius = radius; }

    // Implement ALL abstract methods from GeometricObject
    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }

    @Override
    public double getPerimeter() {
        return 2 * Math.PI * radius;
    }
}
```

Abstract Class and Polymorphism

- You cannot instantiate an abstract class directly: GeometricObject obj = new GeometricObject(); (Compile error).
- However, you can declare a reference variable of an abstract class type.
- This variable can then refer to an object of any concrete subclass. This is polymorphism in action!

Example: Polymorphism with Abstract Classes

```
public class TestGeometricObject {
    public static void main(String[] args) {
        // GeometricObject obj = new GeometricObject(); // COMPILE ERROR: Cannot instantiate abstract class
        GeometricObject circle = new Circle(5.0); // Polymorphic assignment
        GeometricObject rectangle = new Rectangle(4.0, 6.0, "blue", true); // Assuming Rectangle class exists
        System.out.println("Circle Area: " + circle.getArea());
        System.out.println("Circle Color: " + circle.getColor());
        System.out.println("Rectangle Perimeter: " + rectangle.getPerimeter());
        System.out.println("Rectangle is filled? " + rectangle.isFilled());
        // You can pass abstract class types to methods
        displayObject(circle);
        displayObject(rectangle);
    }
```

Example: Polymorphism with Abstract Classes

```
public static void displayObject(GeometricObject obj) {
        System.out.println("\n--- Object Info ---");
        System.out.println("Area: " + obj.getArea());
        System.out.println("Perimeter: " + obj.getPerimeter());
        System.out.println(obj.toString()); // Calls overridden toString() if available, or GeometricObject's
    }
}
```

What are Interfaces?

- A contract that defines a set of behaviors (methods) that a class can agree to implement.
- Represents a "can-do" or "has-a-capability" relationship (e.g., a Car is a Vehicle, but it can Drive and can Brake).
- Interfaces provide pure abstraction (before Java 8, only abstract methods and public static final constants were allowed).
- Declared with the interface keyword.

Why Use Interfaces?

- Achieve multiple inheritance of type: A class can implement multiple interfaces, allowing it to have multiple "types" or capabilities. (Java does not support multiple inheritance of implementation from classes).
- Define common behavior for unrelated classes: Classes that don't share a common superclass can still implement the same interface.
- Decouple design: Separate the definition of an API from its implementation.
- Support callbacks and plugging new modules.

Interface Syntax (Pre-Java 8)

```
public interface InterfaceName {
    // Constants (implicitly public static final)
    int DEFAULT_VALUE = 10;
    String DEFAULT_MESSAGE = "Hello";

    // Abstract methods (implicitly public abstract)
    void doSomething();
    int calculate(int x, int y);
}
```

- Before Java 8, all interface methods were implicitly public abstract.
- All data fields were implicitly public static final.

Interfaces: Abstract Methods (Pre-Java 8 Syntax)

- Methods in an interface without a body.
- They are by default public abstract. You can omit these keywords.
- Any class that implements an interface must provide concrete implementations for all of its abstract methods.
- Example:

```
public interface Edible {
   /** Describe how to eat */
   String howToEat();  // Implicitly public abstract
}
```

Interfaces: Constants

- Variables declared in an interface are implicitly public static final.
- They are constants accessible using the interface name.
- Example :

```
public interface Measurable {

    double PI = 3.14159; // Implicitly public static final double PI

    int MAX_VALUE = 100; // Implicitly public static final int MAX_VALUE


    double getMeasure();

}
```

Interfaces: Default Methods (Java 8+)

- Allow you to add new methods to an interface without breaking existing classes that implement that interface.
- Have a default implementation, which can be overridden by implementing classes.
- Declared with the default keyword.
- Example :

```
public interface Flyable {
    void fly(); // Abstract method
    default void takeOff() { // Default method
        System.out.println("Taking off default way.");      }
    default void land() { // Another default method
        System.out.println("Landing default way.");     }
}
```

Interfaces: Static Methods (Java 8+) and Private Methods (Java 9+)

- Static Methods (Java 8+):
  - Can be defined in interfaces.
  - Belong to the interface itself, not to implementing objects.
  - Called directly on the interface: InterfaceName.staticMethod().
  - Cannot be overridden by implementing classes.
- Private Methods (Java 9+):
  - Can be used to break down complex default or static methods into smaller, reusable parts within the interface.
  - Not accessible outside the interface.

Example :

```
public interface Calculator {
    int add(int a, int b);
    static int multiply(int a, int b) { // Static method
        return a * b;
    }
default void printSum(int a, int b) { // Default method using private helper
        int sum = add(a, b);
        log("Sum calculated: " + sum); // Calls private method
    }
private void log(String message) { // Private method (Java 9+)
        System.out.println("[Calculator Log]: " + message);
    }
}
```

Implementing an Interface (implements keyword)

- A class uses the implements keyword to indicate that it provides concrete implementations for the methods defined in an interface.
- Syntax: class ClassName implements InterfaceName1, InterfaceName2 { ... }
- A class can implement multiple interfaces.

Example: Chicken implementing Edible

```
// Chicken.java
public class Chicken extends Animal implements Edible { // Assuming Animal class
    @Override
    public String howToEat() {          return "Fry it";       }     // Other Chicken specific methods and properties
}
// And an example of another class using it:
class TestEdible {
    public static void main(String[] args) {
        Object[] objects = {new Chicken(), new Tiger(), new Apple()}; // Tiger, Apple also assuming exists
        for (int i = 0; i < objects.length; i++) {
            if (objects[i] instanceof Edible) {  System.out.println(((Edible)objects[i]).howToEat());          }
            if (objects[i] instanceof Animal) { System.out.println(((Animal)objects[i]).sound());
              // Assuming Animal has sound()
            }       }    }}
```

Interface and Polymorphism

- You cannot instantiate an interface: Edible e = new Edible(); (Compile error).
- However, you can declare an interface reference variable.
- This variable can refer to any object of a class that implements that interface.
- This allows you to treat diverse objects uniformly based on their shared capabilities.

Example: Polymorphism with Interfaces

```
// TestPolymorphicInterface.java
public class TestPolymorphicInterface {
    public static void main(String[] args) {
        // Assume Circle implements Comparable (as it does in Liang's book)
        Comparable<Circle> c1 = new Circle(5.0);
        Comparable<Circle> c2 = new Circle(3.0);

        // You can compare objects through the Comparable interface
        int comparison = c1.compareTo((Circle) c2); // Casting c2 to Circle for compareTo

        if (comparison > 0) {
            System.out.println("Circle 1 is larger than Circle 2.");
        } else if (comparison < 0) {
            System.out.println("Circle 1 is smaller than Circle 2.");
        } else {
            System.out.println("Circle 1 is equal to Circle 2.");
        }
```

Example: Polymorphism with Interfaces

```
// Another example with the Flyable interface
        Flyable bird = new Bird(); // Assuming Bird implements Flyable
        Flyable plane = new Airplane(); // Assuming Airplane implements Flyable

        bird.fly();
        bird.takeOff(); // Calls default method
        plane.fly();
        plane.takeOff(); // Calls default method

        // Calculator static method
        System.out.println("2 * 3 = " + Calculator.multiply(2, 3));
    }
}
```

Abstract Classes vs. Interfaces: Key Differences (Part 1)

Feature	Abstract Class	Interface
Type of Inheritance	"Is-a" relationship (strong hierarchy)	"Can-do" relationship (capability)
Multiple Inheritance	No (Java classes can only extend one class)	Yes (a class can implement multiple interfaces)
Constructors	Can have constructors (used by subclasses via super())	Cannot have constructors (no state to initialize)
Data Fields	Can have any type of data field (private, protected, public, static, final)	Only public static final fields (constants)
Methods	Can have abstract and concrete methods	Can have abstract, default, static, and private methods
Access Modifiers	Can have public, protected, private, default (package-private) members	All abstract, default, static methods are implicitly public (private methods are private)
Implementation	Provides partial implementation	Provides no implementation (pre-Java 8), or default/static implementations (Java 8+)

When to Use Which? (Design Principles)

Use an Abstract Class when:

- You have a strong "is-a" hierarchy and want to share common implementation details among related subclasses.
- You need to provide a base for subclasses but the base class itself isn't a complete concept.
- You need to define abstract methods that subclasses must implement, but also want to provide common concrete methods or state (instance variables).
- You want to control access modifiers for inherited members (private, protected).

When to Use Which? (Continued)

Use an Interface when:

- You want to define a contract for behavior that multiple, potentially unrelated classes can adhere to. (e.g., Comparable, Runnable, Serializable).
- You need to achieve polymorphic behavior across classes that don't share a common class hierarchy.
- You want to separate what a class does from how it does it.
- You are designing a system where new capabilities might be "plugged in" easily.

Example: The Comparable Interface

- A built-in Java interface (java.lang.Comparable<T>).
- Defines the compareTo(T o) method.
- Classes that implement Comparable can be sorted naturally (e.g., using Collections.sort() or Arrays.sort()).

Example :

```
// Assuming Circle implements Comparable<Circle>
public class Circle extends GeometricObject implements Comparable<Circle> {
    // ... (previous Circle class code) ...
    @Override
    public int compareTo(Circle other) {
        if (this.radius > other.radius) {
            return 1;
        } else if (this.radius < other.radius) {
            return -1;
        } else {
            return 0;
        }    } }
```

Example: Arrays.sort() with Comparable

```
import java.util.Arrays;
public class TestComparableCircle {
    public static void main(String[] args) {
        Circle[] circles = { new Circle(3.5),new Circle(1.2),new Circle(4.0),new Circle(2.8) };
        System.out.println("Circles before sorting:");
        for (Circle c : circles) {
              System.out.printf("Radius: %.1f%n", c.getRadius());        }
        Arrays.sort(circles); // Requires Circle to implement Comparable
        System.out.println("\nCircles after sorting by radius:");
        for (Circle c : circles) {
            System.out.printf("Radius: %.1f%n", c.getRadius());
 }
    }
}
```

```
/* Expected Output:
Circles before sorting:
Radius: 3.5
Radius: 1.2
Radius: 4.0
Radius: 2.8

Circles after sorting by radius:
Radius: 1.2
Radius: 2.8
Radius: 3.5
Radius: 4.0
*/
```

Design Considerations

- Flexibility: Interfaces often lead to more flexible designs, as classes can implement multiple behaviors.
- Extensibility: Easier to add new implementations without modifying existing code.
- Loose Coupling: Code depends on an abstract type (interface or abstract class) rather than specific concrete implementations.
- Liskov Substitution Principle: Objects of a superclass should be replaceable with objects of its subclasses without breaking the application. Abstract classes and interfaces support this.

Conclusion & Key Takeaways

- Abstraction is key: Hide complexity, expose essentials.
- Abstract Classes: Provide a base for related "is-a" objects, allowing shared implementation and enforced abstract methods.
- Interfaces: Define contracts for "can-do" capabilities, enabling multiple type inheritance and loose coupling.
- Polymorphism: Both abstract classes and interfaces are fundamental for achieving polymorphism, allowing uniform handling of diverse objects.
- Mastering Chapter 13 is crucial for designing robust, adaptable, and maintainable object-oriented systems in Java.