Abstract Classes and Interfaces

Josh Wilcox (jw14g24@soton.ac.uk)

February 17, 2025

Contents

1 Abstract Classes in Java

1.1 Definition

An abstract class in Java is a class that cannot be instantiated directly. It serves as a blueprint for other classes, allowing you to define both abstract methods (without implementation) and concrete methods (with implementation).

1.2 Purpose and Features

- **Template for Subclasses:** Abstract classes provide a framework for other classes to build upon.
- **Mix of Methods:** They can contain both abstract methods and methods with full implementations.
- **Code Reuse:** Common functionality shared by multiple subclasses can be implemented once in the abstract class.

1.3 Syntax

To declare an abstract class, use the `abstract` keyword:

```
public abstract class Animal {
    // Abstract method: no body provided
    public abstract void makeSound();

    // Concrete method: has an implementation
    public void sleep() {
        System.out.println("Sleeping...");
    }
}
```

1.4 Implementing Abstract Methods

Any concrete subclass of an abstract class must implement all its abstract methods.

```
public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }
}
```

1.5 Additional Characteristics

- Abstract classes can have constructors, fields, and static methods.
- They can also define non-abstract methods that provide default functionality to subclasses.
- While you cannot instantiate an abstract class directly, constructors of abstract classes are called when a subclass is instantiated.

1.6 Requirements

- At least one abstract method → the class has to be an abstract class
- All the subclasses inherited from an abstract class, have to provide implementations for all the abstract methods. Otherwise, that subclass has to be abstract

- Abstract methods can't be declared as private. (i.e. either protected or public).

1.7 Comparing Abstract Classes and Interfaces

- **Abstract Classes:** Use when several related classes share common behavior or state. They allow both method declarations and definitions.
- **Interfaces:** Use when you require a contract that a class must fulfill, with Java allowing default and static methods in interfaces (from Java 8 onwards) but focusing on abstract method declarations.

1.8 When to Use Abstract Classes

- When creating a base class that should not be instantiated on its own.
- When defining a common template for a group of subclasses with shared code.
- When you want to ensure certain methods are implemented by all subclasses, along with providing some default behavior.

2 The point of using Abstract Classes

- Acts as a **guarantee** that the subclasses will provide them
- This allows you to code for polymorphism without having to write methods that contain no code or will never be called

3 Interfaces

- Interfaces allow a solution to **multiple inheritance**
  - An interface behaves like a 100% abstract class that **only contains abstract methods**
  - A class can extend **one** Class but can implement **many** classes
  - Think of an interface like a **contract**, any class that implements will guarantee to provide code for all of its methods

4 Interfaces in Practice: The Pet Example

Interfaces define a contract that any implementing class must follow. In this example, we create an interface named Pet and then implement it in several classes. Each class provides its own behavior for the play() method, demonstrating polymorphism.

```
1  public interface Pet {
2      void play();
3  }
4
5  public class Dog extends Canine implements Pet {
6      // Code omitted: Additional methods and fields for Dog
7
8      @Override
9      public void play() {
10         System.out.println("Dog plays with a ball");
11     }
12 }
13
14 public class Wolf extends Canine {
```

```
15      // Code omitted: Additional methods and fields for Dog
16      public void play() {
17          System.out.println("Wolfs are too sigma to play");
18      }
19  }
20  public class Cat extends Feline implements Pet {
21      // Code omitted: Additional methods and fields for Cat
22
23      @Override
24      public void play() {
25          System.out.println("Cat plays with some string");
26      }
27  }
28
29  public class Hamster extends Rodent implements Pet {
30      // Code omitted: Additional methods and fields for Hamster
31
32      @Override
33      public void play() {
34          System.out.println("Hamster plays on its wheel");
35      }
36  }
```

This example highlights how interfaces allow multiple unrelated classes to share a common behavior while still providing their own specific implementations.

5 "Extends" vs "Implements" in Java

- A class can use `implements` for as **many** interfaces as it wants
- A class can use `extends` for **at most one** class
-