Java Abstraction

Table of contents

What is abstraction? 1
Interfaces 1
Abstract classes 2
Interfaces vs abstract classes 4

Introduced in 2026-03-05-object-oriented-programming-i[1] (Lecture, Week 2).

What is abstraction?

The ability to hide complex implementation details and expose only the essential behaviours. In Java, abstraction is often implemented using **interfaces** and **abstract classes** — both define methods that must be implemented by subclasses, providing a contract that specifies *what* operations can be performed on an object without specifying *how* those operations are implemented.

Interfaces

- An interface is a group of related methods **with empty bodies**.
- An interface defines a **contract** that implementing classes must fulfil.
- To access the interface methods, the interface must be **implemented** — and a class can implement multiple interfaces if required.

  Interfaces provide abstraction in Java by defining behaviour without specifying implementation.

[1] courses/csse2002/2026-03-05-object-oriented-programming-i.pdf

```java
public interface PaymentProcessor {
    void processPayment(double amount);
}
```

Any payment processor must have a ***`processPayment()`*** *method.* Two classes take the interface up, each with its own body for that one method:

```java
public class PayPalProcessor implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment: " + amount);
    }
}

public class StripeProcessor implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing Stripe payment: " + amount);
    }
}
```

and the caller is written against the interface, never against either class:

```java
public class Main {
    public static void main(String[] args) {
        PaymentProcessor processor = new PayPalProcessor();
        processor.processPayment(100);
    }
}
```

The declared type of `processor` is `PaymentProcessor`, so `Main` can only call what the interface promises: the program only interacts with the interface. Swapping `new PayPalProcessor()` for `new StripeProcessor()` is therefore a one-line change: it changes what runs without touching anything else in `Main`.

> An interface in Java represents an abstraction. It defines a set of behaviours that classes must implement, but it does not define how those behaviours are implemented. The implementation details are hidden inside the classes that implement the interface.

Abstract classes

- An abstract class is a class that **cannot be instantiated**.

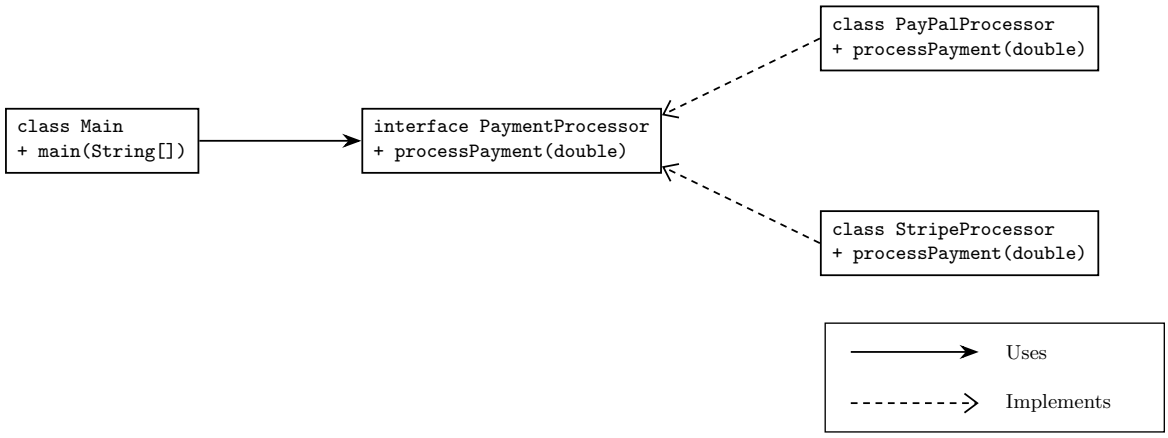

Figure 1: Main depends on the PaymentProcessor interface, not on either implementation - which is what makes swapping PayPal for Stripe a one-line change. Each rectangle is labelled with its Java keyword and lists its members as + lines. A solid arrow with a solid head means uses. A dashed arrow with an open head means implements and runs from the implementer to the interface. The client is on the left and the abstraction in the centre.

- It can contain **both** abstract methods and fully implemented methods.
- The `abstract` keyword is used to declare abstract classes and methods.
- Subclasses must implement all abstract methods.

One qualification to the first bullet: an instance of an abstract class can be created using an anonymous class. The two statements are consistent. The ban is on `new Payment()`, while `new Payment() { ... }` instantiates an anonymous subclass rather than `Payment` itself.

```
public abstract class Payment {
    public void validateTransaction() {
        System.out.println("Validating transaction...");
    }

    abstract void processPayment(double amount);
}
```

That class **defines common behaviour for every payment** (`validateTransaction`), and does **NOT** define how a payment is processed. A subclass supplies the missing half:

```
public class PayPalPayment extends Payment {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment: $" + amount);
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Payment payment = new PayPalPayment();

        payment.validateTransaction();
        payment.processPayment(150.0);
    }
}
```

The abstract class defines the common behaviour of all payments while leaving the specific logic to subclasses. The two calls in `main` do different things structurally: `validateTransaction()` runs the body inherited from `Payment`, while `processPayment()` runs the body `PayPalPayment` had to supply.

Interfaces vs abstract classes

A **homework / self-study** item in Week 2, taken up in 2026-03-12-object-oriented-programming-ii[2] (Lecture, Week 3).

- **Use an interface** to define a capability or contract, e.g. `Flyable`, `Runnable`, `Payable`. A class can implement multiple interfaces.
- **Use an abstract class** to define a shared base implementation (common behaviour), e.g. an `Animal` abstract class shared by `Dog` and `Cat`. A class can only extend one (super)class.

```
abstract class Animal {
    public void eat() {
        System.out.println("Animal is eating");
    }
    public abstract void makeSound();
}

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

class Cat extends Animal {
```

[2] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf

```
    @Override
    public void makeSound() {
        System.out.println("Cat Meows");
    }
}
```

A class can combine both: extend at most one abstract/concrete class, while implementing as many interfaces as needed:

```
class Dog extends Animal implements Runnable, Swimmable
class Duck extends Animal implements Flyable, Swimmable
```

Both mechanisms lean on the same machinery as java-inheritance[3] — the `@Override` annotation and identical method signatures — and both pay off in Week 3 as java-polymorphism[4].

[3] courses/csse2002/java-inheritance.pdf
[4] courses/csse2002/java-polymorphism.pdf