Object Oriented Programming II

Table of contents

Recap: the four (five) pillars . . . 1
Interface vs abstract class . . . 2
Revising encapsulation . . . 3
Mutable vs immutable classes . . . 4
Polymorphism . . . 4
Casting . . . 4
Worked: what is the parent of `Student`? . . . 4
Specification . . . 5
Key takeaways . . . 5

See csse2002[1] for course logistics. Continues 2026-03-05-object-oriented-programming-i[2].

The week has two parts:

1. **Object Oriented Programming II**: interface vs abstract class, encapsulation revision, mutability, polymorphism, casting.
2. **Specification**: Javadoc, restrictiveness, generality, clarity, formality, contracts, defensive programming.

Three items are homework (self-study): primitive casting, defensive copying, and how a subclass can introduce mutability.

Recap: the four (five) pillars

Week 2 covered objects/classes/OOP, encapsulation, inheritance and abstraction. This week adds **polymorphism** as the fifth, and goes back over encapsulation and the interface/abstract-class comparison in more depth.

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

Interface vs abstract class

Week 2 left this comparison as homework. The full treatment is in java-abstraction[3]. In short:

- **An interface defines a capability or contract** — `Flyable`, `Runnable`, `Payable`. Unrelated classes can share it: `Bird implements Flyable` and `Plane implements Flyable`.
- **An abstract class defines a shared base implementation** — common behaviour that subclasses inherit rather than restate.

In code, the interface carries nothing but a signature. The abstract class carries a *finished* method alongside an unfinished one.

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

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

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

The course's running `Animal` example has the same shape. `eat()` is inherited as-is, and `makeSound()` must be supplied by each subclass:

```
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 {
    @Override
```

[3] courses/csse2002/java-abstraction.pdf

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

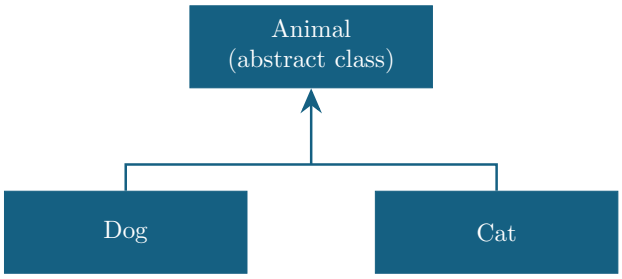

Figure 1: The abstract-class case: shared behaviour (eat()) lives in Animal, while each subclass supplies its own makeSound(). One connector joins both subclasses to the parent, with a single arrowhead into Animal.

The two combine. A class has **at most one superclass, as many interfaces as you like**.

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

Revising encapsulation

`Person` from java-encapsulation[4] again: `private` fields with public getters. One field matters for the next section:

```
public class Person {
    private String name;
    private int age;
    private String email;
    private String[] phoneNumber;   // <- a mutable field

    public Person(String name, int age, String email, String[] phoneNumber) {
        this.name = name;
        this.age = age;
        this.email = email;
        this.phoneNumber = phoneNumber;
    }

    public String getName()          { return name; }
    public int getAge()              { return age; }
```

[4] courses/csse2002/java-encapsulation.pdf

```
    public String getEmail()         { return email; }
    public String[] getPhoneNumber() { return phoneNumber; }
}
```

The array field is the case where `private` and `final` together are still not enough.

Mutable vs immutable classes

See java-mutability[5] — mutable and immutable class design, the arrays/lists gotcha where `final` protects only the reference, and defensive copying.

Polymorphism

See java-polymorphism[6] — compile-time (overloading), runtime (overriding / dynamic method dispatch), and subtype polymorphism.

Casting

See java-casting[7] — primitive widening/narrowing, and reference upcasting/downcasting. Practised in week4-tutorial-method-dispatch-and-casting[8].

Worked: what is the parent of `Student`?

What is the parent of class `Student`?

```
public class Student {
    public void study() {
        System.out.println("CSSE2002");
    }
}
```

`Object`. A class with no `extends` clause still has a superclass. The class hierarchy has `Object` at the root of everything.

[5] courses/csse2002/java-mutability.pdf
[6] courses/csse2002/java-polymorphism.pdf
[7] courses/csse2002/java-casting.pdf
[8] courses/csse2002/week4-tutorial-method-dispatch-and-casting.pdf

Specification

See java-specification[9] — Javadoc, restrictiveness, generality, clarity, formality, contracts, and defensive programming.

Key takeaways

Object oriented programming:

- Encapsulation protects implementation details and prevents direct access to internal state.
- Inheritance lets a class reuse behaviour from another class — but **can expose the internal implementation details of a superclass**.
- Abstraction hides complex implementation detail: the user knows *what* it does, not *how* it works internally.
- Polymorphism lets objects be treated as their common type, and lets different objects respond differently to the same method call.

OOP helps build flexible, reusable, and maintainable software by organizing code around objects and behaviour.

Specification:

- A specification describes the **contract between a method and its client**.
- It explains what the method does, not how it is implemented.
- A good specification clearly states **preconditions** (true before the call), **postconditions** (true after a successful call) and **side effects** (any other change the method makes to the system).
- Clear specifications make code easier to understand, use and maintain.

[9] courses/csse2002/java-specification.pdf