

Inheritance and Polymorphism

extends, super, overriding, overloading, static vs dynamic binding, instanceof

Lecture + Lab 2

Branch: week3-inheritance-polymorphism

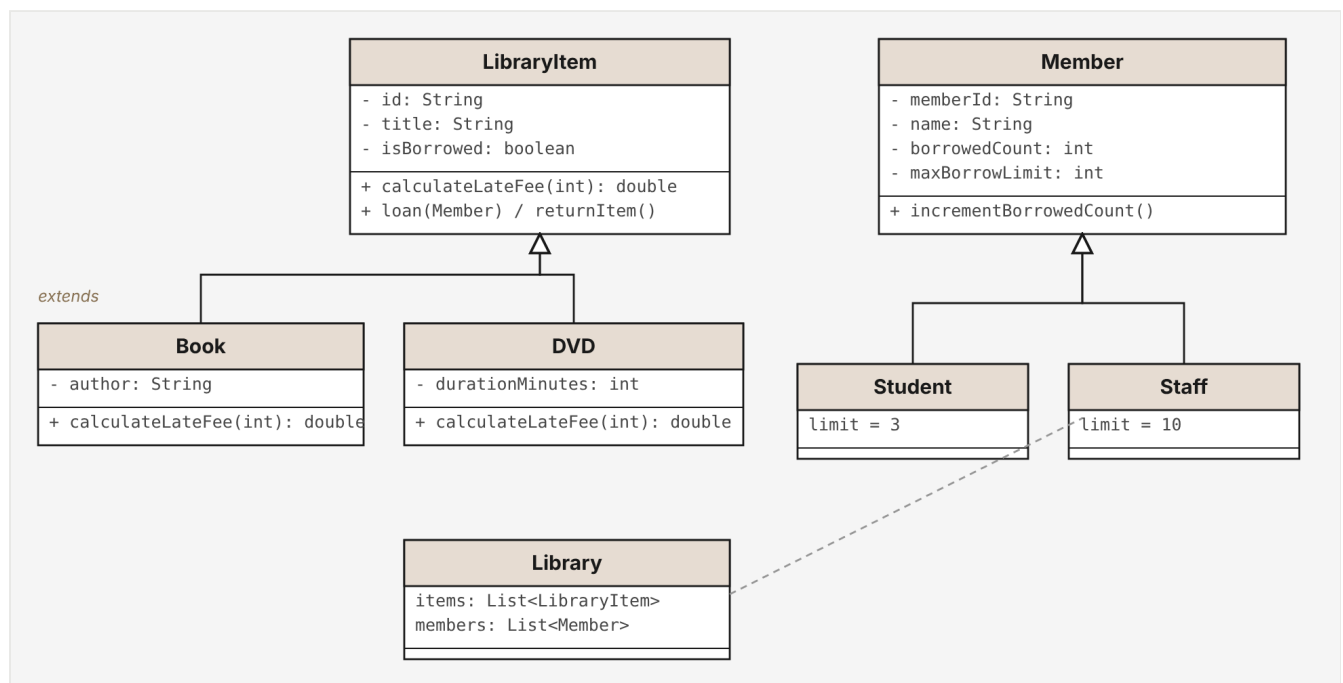
Learning outcomes 1 and 3

What you will be able to do after today

- Use `extends` and `super` to build a class hierarchy and reuse code.
- **Override** a method and explain **dynamic binding**.
- **Overload** a method and explain **static binding**.
- Use `instanceof` safely, and know when polymorphism is the better answer.

1. Why inheritance?

In Week 2 we had `Book`. A library also lends **DVDs**. Copy-pasting `Book` into `DVD` duplicates the id, title and borrowing logic, so one bug becomes two. Inheritance says: *put what they share in a superclass; put what is special in each subclass.*



```

public class LibraryItem {                                // superclass: what ALL items share
    private String id;
    private String title;
    private boolean isBorrowed;

    public LibraryItem(String id, String title) {
        setId(id);
        setTitle(title);
        this.isBorrowed = false;
    }
    // getters, validated setters, borrowing logic ...
}

public class Book extends LibraryItem {                   // "a Book IS-A LibraryItem"
    private String author;

    public Book(String id, String title, String author) {
        super(id, title);                                // run the parent's constructor FIRST
        setAuthor(author);
    }
}

```

Tip Test for inheritance with the sentence "**X is a Y**". A Book *is a* LibraryItem. A Student *is a* Member. But a Library is **not** a Book: it *has* books, so that is composition.

`super(...)` must be the first line of a subclass constructor. If you omit it, Java quietly calls the parent's no-argument constructor, and if there is none, the code will not compile.

2. Polymorphism: "many forms"

A `LibraryItem` variable can hold a `Book`, a `DVD`, or any future subclass.

```

LibraryItem item = new DVD("D001", "Introduction to Algorithms", 120); // fine: a DVD is a
LibraryItem

```

The late fee rules differ, so each subclass **overrides** the method:

```

public class Book extends LibraryItem {
    @Override
    public double calculateLateFee(int daysLate) { return daysLate * 10.0; }
}
public class DVD extends LibraryItem {
    @Override
    public double calculateLateFee(int daysLate) { return daysLate * 25.0; }
}

```

Now one loop works for every type, now and in the future:

```

for (LibraryItem item : items) {
    System.out.println(item.getTitle() + ": " + item.calculateLateFee(3));
}

```

```

Clean Code: 30.0
Effective Java: 30.0
Introduction to Algorithms: 75.0

```

Check Which `calculateLateFee` ran for the DVD? Java decided **while the program was running**, by looking at the real object, not the variable's type. That is **dynamic binding**.

Always write `@Override`. If you misspell the method name or get a parameter wrong, the compiler tells you instead of silently creating a second method.

3. Overriding vs overloading

	Overriding	Overloading
Where	Subclass redefines a parent method	Same class, same name, different parameters
Signature	Identical	Different
Decided	At run time by the object's real class	At compile time by the variables' declared types
Name	Dynamic binding	Static binding

Overloading in the LMS:

```
public void addItem(LibraryItem item) { ... }
public void addItem(Book book)      { ... }
public void addItem(DVD dvd)        { ... }
```

```
Book b1 = new Book("B001", "Clean Code", "Robert C. Martin");
library.addItem(b1);           // declared type Book -> addItem(Book)
LibraryItem same = b1;
library.addItem(same);          // declared type LibraryItem -> addItem(LibraryItem) (same
                                object!)
```

The *same object* picks a different method because overloads look only at the **declared type**. This surprises everyone once. Run the experiment in the lab.

4. `instanceof` and casting

Sometimes only one subclass has a feature, such as streaming a DVD:

```
public void streamIfPossible(LibraryItem item) {
    if (item instanceof DVD dvd) {           // test AND cast in one step (pattern matching)
        System.out.println("Streaming \"" + dvd.getTitle() + "\" (" + dvd.getDurationMinutes() + "
min)...");
    } else {
        System.out.println(item.getTitle() + " cannot be streamed - it isn't a DVD.");
    }
}
```

Without the test, `item.getDurationMinutes()` does not compile: the variable's type, `LibraryItem`, has no such method. That is **cannot find symbol**, the most common error of this week.

Warning Many `instanceof` chains are a sign that the method belongs *in the subclass* and should be overridden instead. Use `instanceof` for genuine special cases only.

5. Members too: `Student` and `Staff`

```
public class Student extends Member {
    private static final int STUDENT_BORROW_LIMIT = 3;
    public Student(String memberId, String name) { super(memberId, name, STUDENT_BORROW_LIMIT); }
}
```

`Staff` is the same with limit 10. A `Library` can hold both in one `List<Member>`.

6. Lab 2 preview (Week 3)

Create `LibraryItem` (move shared code up), make `Book` and `DVD` extend it, add `Student / Staff`, add `calculateLateFee` overrides, the `addItem` overloads and `instanceof`. Include one deliberate compile error and its message in your PR description. Branch `week3-inheritance-polymorphism`.

7. Self-check

1. Complete the sentence: "a `Student` ___ a `Member`; a `Library` ___ `Members`."
2. What happens if a subclass constructor has no `super(...)` and the parent has no no-argument constructor?
3. Overloading is bound at ___ time; overriding is bound at ___ time.
4. Why does `item.getDurationMinutes()` fail to compile when `item` is a `LibraryItem` that really is a `DVD`?