

An Introduction to Object-Oriented Programming (OOP)

Concepts, Modeling, Design Principles, and Trade-offs

Diogo Ribeiro

January 14, 2026

Learning Objectives

By the end of this session, you will be able to:

- Explain core OOP concepts: objects, classes, identity, state, behavior.
- Describe encapsulation, abstraction, inheritance, and polymorphism.
- Model simple domains with responsibilities and relationships.
- Read and sketch basic class and interaction diagrams.
- Apply key design principles (e.g., SOLID) pragmatically.
- Recognize trade-offs and common pitfalls in OOP design.

Agenda (*5 min*)

- ① Why OOP? (*10 min*)
- ② Core Concepts & Terminology (*15 min*)
- ③ The Four Pillars (*25 min*)
- ④ Modeling & Diagrams (*15 min*)
- ⑤ Design Principles (SOLID) (*15 min*)
- ⑥ Case Study & Trade-offs (*10 min*)
- ⑦ Wrap-up, Quiz, Resources (*10 min*)

Motivation (*5 min*)

- Real-world entities have identity, state, and behavior.
- OOP models systems as **objects** that interact.
- Goals: modularity, reuse, maintainability, and clarity.
- Suitable for complex domains with rich interactions.

When OOP Helps (and When It Doesn't) (5 min)

Helps:

- Complex domains, evolving requirements.
- Teams needing clear ownership and interfaces.
- Systems benefiting from composition and extension.

Caution:

- Over-engineering small scripts.
- Deep inheritance hierarchies.
- Excessive indirection harming performance/readability.

Objects and Classes (*6 min*)

Object: an entity with identity, state, and behavior.

Class: a blueprint defining properties (state) and methods (behavior).

- **Identity:** distinguishes one object from another.
- **State:** data held by the object (fields/properties).
- **Behavior:** operations the object can perform (methods).

Instances, Messages, and Interfaces (*4 min*)

- **Instance:** a concrete occurrence of a class.
- **Message/Call:** a request sent to an object to invoke behavior.
- **Interface:** a contract describing behavior without revealing implementation.

Pseudocode Sketch (5 min)

```
// Blueprint
class Counter:
    private value: Integer

    constructor(start: Integer):
        this.value = start

    method increment():
        this.value = this.value + 1

    method read() -> Integer:
        return this.value

// Use
let c = new Counter(0)
c.increment()
print(c.read()) // 1
```

Encapsulation (*6 min*)

- Bundle state and behavior together.
- Hide implementation details; expose a stable interface.
- Benefits: invariants, safer evolution, reduced coupling.

Encapsulation: Example (4 min)

```
class BankAccount:  
    private balance: Money  
  
    constructor(initial: Money):  
        this.balance = initial  
  
    method deposit(amount: Money):  
        assert amount > 0  
        this.balance = this.balance + amount  
  
    method withdraw(amount: Money):  
        assert 0 < amount <= this.balance  
        this.balance = this.balance - amount  
  
    method getBalance() -> Money:  
        return this.balance
```

Abstraction (*5 min*)

- Focus on *what* an object does, not *how*.
- Specify behavior via contracts (preconditions, postconditions).
- Replaceable implementations behind a stable interface.

Inheritance (*5 min*)

- Mechanism for reusing and specializing behavior.
- **Subtype** extends or refines a **supertype**.
- Use with care: prefer composition when reuse is ornamental.

Polymorphism (*5 min*)

- Same message, different implementations.
- Enables substitutability and open-ended extension.
- Often paired with interfaces/abstract types.

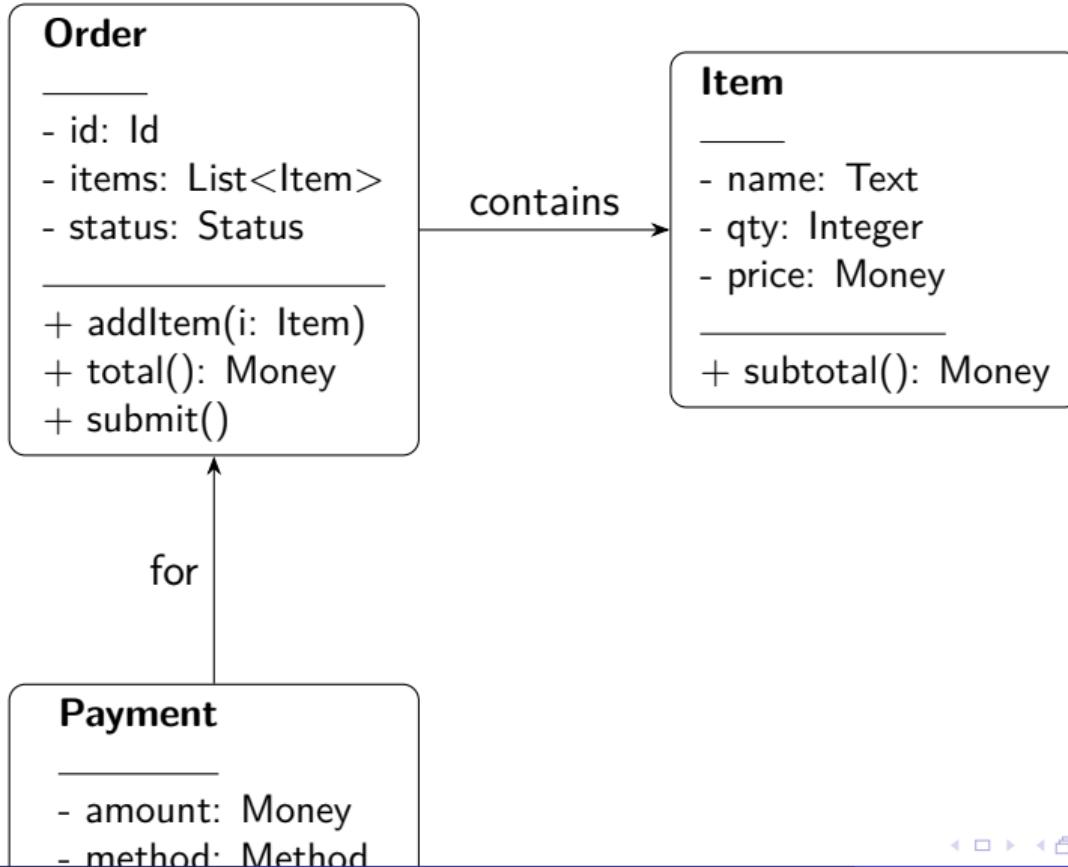
Polymorphism: Example (4 min)

```
interface Shape:  
    method area() -> Number  
  
class Circle implements Shape:  
    private r: Number  
    method area() -> Number: return 3.14159 * r * r  
  
class Rectangle implements Shape:  
    private w: Number; private h: Number  
    method area() -> Number: return w * h  
  
// Client code (works with any Shape)  
let shapes: List<Shape> = [Circle(2), Rectangle(3,4)]  
print(sum(s.area() for s in shapes))
```

From Domain to Design (*3 min*)

- ① Identify core domain concepts (nouns) and responsibilities (verbs).
- ② Clarify relationships (association, aggregation, composition).
- ③ Define interfaces and collaboration scenarios.

Basic Class Diagram (6 min)



Interaction Sketch (Sequence) (5 min)

- **Scenario:** Customer submits an order and pays.
- Steps:
 - ① `Order.submit()` validates lines, computes total.
 - ② `Payment.authorize()` checks funds.
 - ③ `Payment.capture()` finalizes transaction.

SOLID in Practice (*10 min*)

- **S:** Single Responsibility — one reason to change.
- **O:** Open/Closed — open for extension, closed for modification.
- **L:** Substitution — subtypes must honor supertype contracts.
- **I:** Interface Segregation — many small contracts over one large.
- **D:** Dependency Inversion — depend on abstractions, not details.

Composition over Inheritance (*5 min*)

- Prefer assembling behavior from parts to rigid hierarchies.
- Reduces coupling and diamond problems.
- Inheritance still useful for true “is-a” relationships.

Law of Demeter (*3 min*)

- Talk only to your immediate collaborators.
- Avoid “train wreck” calls (chained navigation across objects).

Activity 1: Identify Objects & Responsibilities (7 min)

Prompt: Choose a simple domain (library loans, food delivery, or ticket booking).

Task:

- ① List candidate objects, their state, and responsibilities.
- ② Propose 2–3 interactions among them.

Deliverable: A quick sketch on paper or whiteboard.

Case Study: Payment Methods (8 min)

```
interface PaymentMethod:  
    method authorize(amount: Money) -> Boolean  
    method capture(amount: Money) -> Boolean  
  
class CardPayment implements PaymentMethod:  
    method authorize(amount): /* talks to card network */ return true  
    method capture(amount): /* settle */ return true  
  
class WalletPayment implements PaymentMethod:  
    method authorize(amount): /* wallet API */ return true  
    method capture(amount): /* settle */ return true  
  
class Checkout:  
    method pay(orderTotal: Money, pm: PaymentMethod):  
        require orderTotal > 0  
        if pm.authorize(orderTotal):  
            return pm.capture(orderTotal)
```

Trade-offs (*6 min*)

- **Flexibility vs. Simplicity:** Abstractions add indirection.
- **Reuse vs. Performance:** Extra dispatch or object hops may cost.
- **Inheritance vs. Composition:** Maintainable extension paths.
- **Granularity:** Over-segmentation leads to “class explosion”.

Activity 2: Refactor Toward SOLID (*7 min*)

Prompt: A ReportGenerator both fetches data, formats it, and writes files.

Task: Split responsibilities and introduce abstractions where helpful.

- Identify at least 2 interfaces.
- Show how a new output format would be added without modifying existing code.

Design by Contract (*5 min*)

- **Preconditions:** what must be true before a call.
- **Postconditions:** what is guaranteed after a call.
- **Invariants:** what remains true for a class's state.

Substitutability (LSP) Checks (*5 min*)

- Subtypes must not strengthen preconditions nor weaken postconditions.
- Clients should not need to know the concrete type.
- Violations often surface as surprising runtime behavior.

Why Patterns? (*5 min*)

- Shared vocabulary for recurring design problems.
- Examples: Strategy, Adapter, Composite, Observer, Factory.
- Use judiciously; avoid pattern-driven overdesign.

Example: Strategy (5 min)

```
interface PricingRule:  
    method price(order: Order) -> Money  
  
class RegularPricing implements PricingRule:  
    method price(order): return order.total()  
  
class HolidayDiscount implements PricingRule:  
    method price(order): return order.total() * 0.9  
  
class Checkout:  
    private rule: PricingRule  
    constructor(rule): this.rule = rule  
    method finalPrice(order): return this.rule.price(order)
```

Common Pitfalls (*6 min*)

- God objects and feature envy.
- Overuse of inheritance; fragile base classes.
- Anemic domain models (data holders with no behavior).
- Leaky abstractions and tight coupling.

Heuristics (4 min)

- Name classes by responsibility, not by data container role.
- One clear reason to change.
- Prefer small, cohesive interfaces.
- Make illegal states unrepresentable.

Quick Quiz (6 min)

- ① What are the four pillars of OOP?
- ② Why is composition often preferred over inheritance?
- ③ Give a brief example of polymorphism.
- ④ What makes a good interface?

Key Takeaways (3 min)

- Think in terms of responsibilities and collaborations.
- Encapsulation and abstraction reduce coupling.
- Use inheritance sparingly; embrace polymorphism via interfaces.
- Apply principles pragmatically; measure outcomes.

Further Exploration (*2 min*)

- Practice modeling small domains weekly.
- Rewrite a procedural solution into object collaborations.
- Study a few design patterns; implement tiny examples.

Appendix: Glossary

Term	Meaning
Object	Entity with identity, state, behavior
Class	Blueprint for objects
Interface	Contract describing behavior
Encapsulation	Hiding representation behind methods
Abstraction	Focus on intent, not implementation
Polymorphism	One interface, many forms
Inheritance	Reuse/extension via subtype

Appendix: Relationship Types

- **Association:** “uses/knows about”.
- **Aggregation:** whole–part, parts may exist independently.
- **Composition:** strong ownership; parts’ lifecycle bound to whole.
- **Dependency:** relies on behavior of another type.

Appendix: Timing Plan (90–120 min)

Segment	Minutes
Intro, Objectives, Agenda	5
Why OOP?	10
Core Concepts	15
Four Pillars	25
Modeling & Diagrams	15
Activity 1	7
Design Principles	15
Case Study & Trade-offs	10
Activity 2	7
Quiz, Wrap-up	6
Buffer / Q&A	5–15

Thank you!