The four pillars of OOP in Java

Encapsulation, Inheritance, Polymorphism, Abstraction

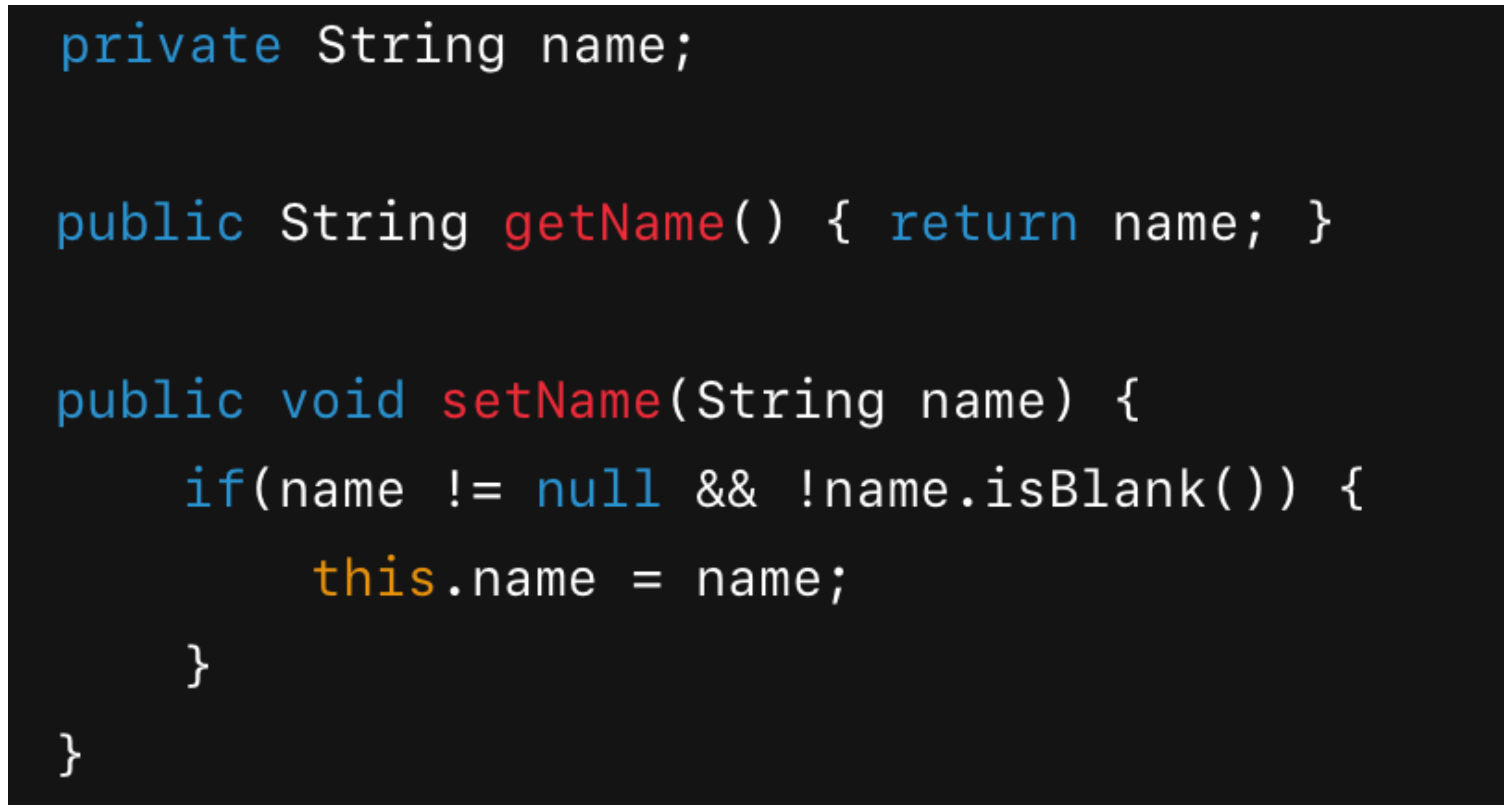
1. What is Encapsulation?

- Encapsulation means bundling data and methods that operate on that data into a single unit – typically a class.
- It also involves restricting direct access to some of an object’s components, which is a form of data hiding.
- Encapsulation helps reduce complexity and increases maintainability and security.

Why Use Encapsulation in Java?

- Provides control over the data (e.g., via getters/setters).
- Helps enforce business rules inside methods (e.g., validation).
- Protects against unexpected modifications.
- Encourages modular code – classes manage their own state.
- Encapsulation is a safeguard for your object's internal state.

How to Achieve Encapsulation in Java



```java
private String name;

public String getName() { return name; }

public void setName(String name) {
    if(name != null && !name.isBlank()) {
        this.name = name;
    }
}
```

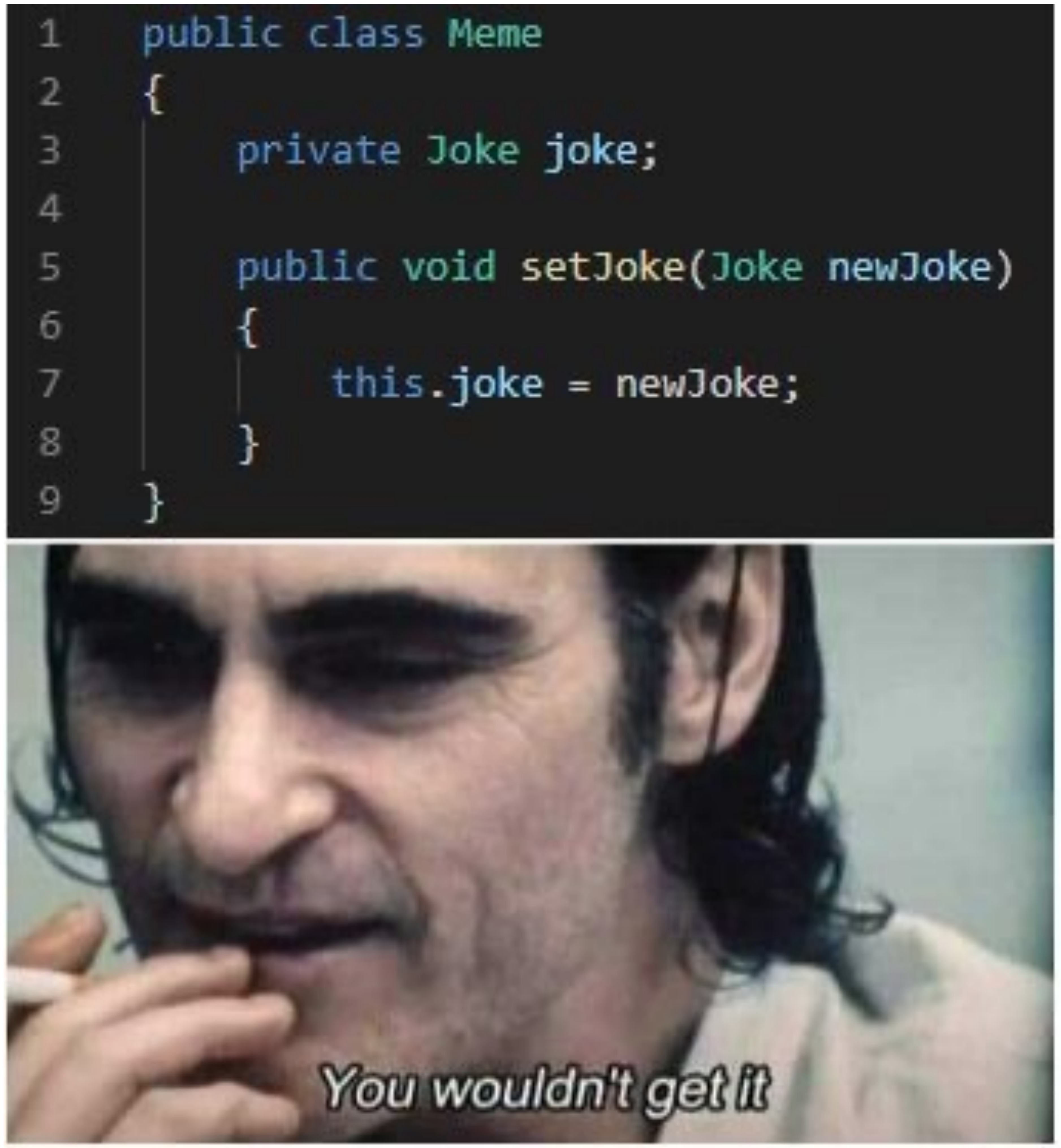
- Make class fields private.
- Provide public getter and setter methods.
- Optionally: add input validation inside setters.

Demo 1 – A Bank Account Class

- We'll now implement a basic BankAccount class that demonstrates Encapsulation:
- ✅ Private fields like balance and ownerName
- ✅ Public methods to access and modify data safely
- ✅ Validation logic inside the setter and transaction methods

Encapsulation

- Are you getting this joke?



```
public class Meme
{
    private Joke joke;

    public void setJoke(Joke newJoke)
    {
        this.joke = newJoke;
    }
}
```

What is Inheritance?

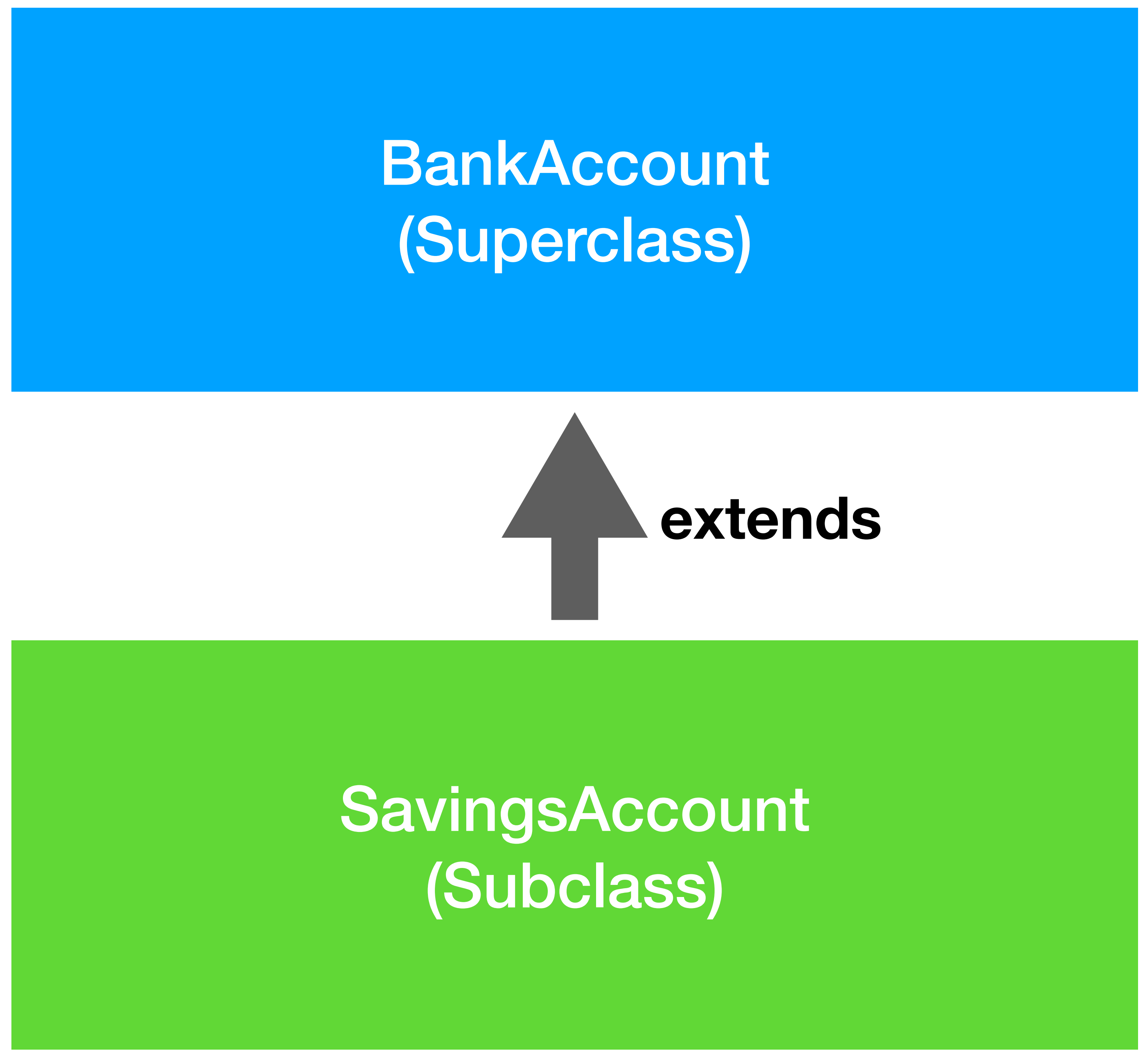
- Inheritance allows a class to inherit properties and behaviors from another class.
- Promotes code reuse by allowing common logic to be defined once in a superclass.
- In Java, inheritance is achieved using the extends keyword.
- Think of it as an "is-a" relationship: A SavingsAccount is a BankAccount.

Inheritance in Java

```java
public class SavingsAccount extends BankAccount {
    // Inherits everything from BankAccount
}
```

- The base class is called the superclass (BankAccount).
- The derived class is the subclass (SavingsAccount).
- Subclasses inherit fields and methods from the superclass.
- Subclasses can add new methods or override inherited ones.

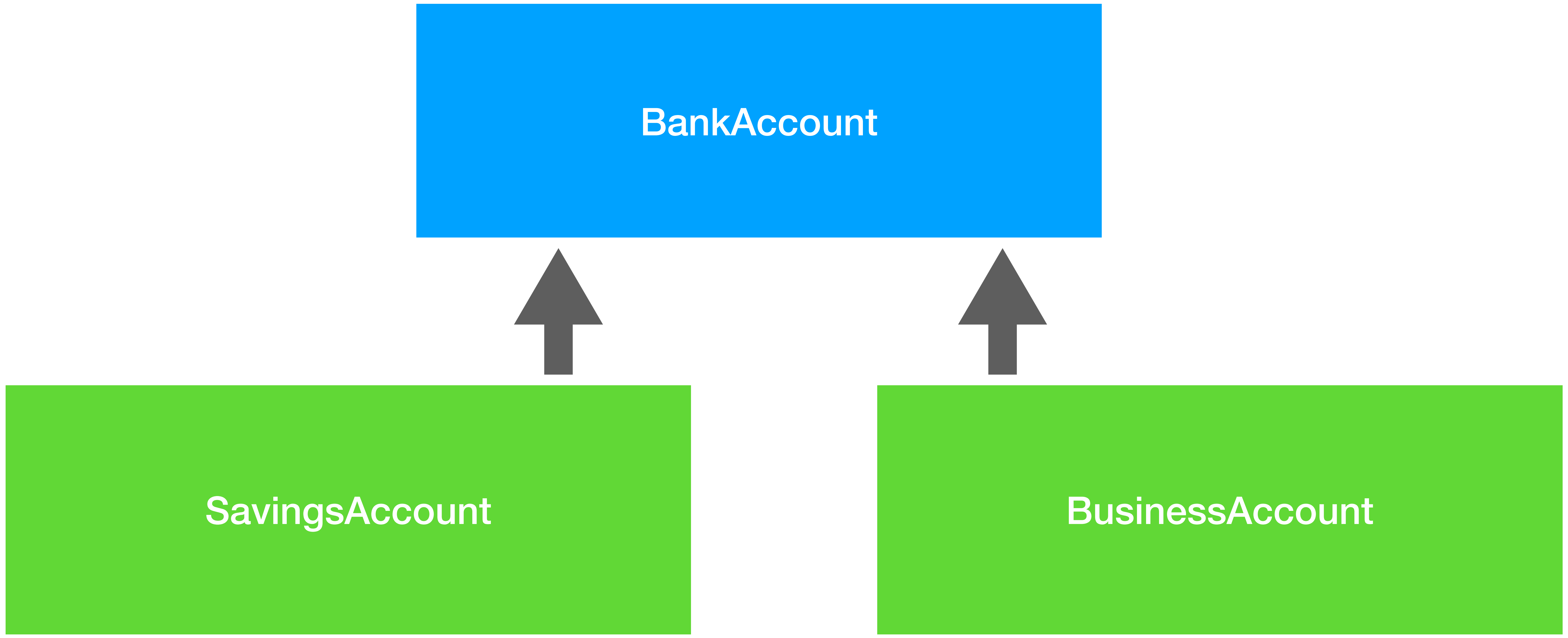
Superclass and subclass



Benefits of Inheritance

- Avoids code duplication by reusing logic.
- Makes code more flexible and extensible.
- Supports polymorphism (covered next).
- Inheritance builds a hierarchy of related classes.

Class hierarchy



Demo 2 – Add a SavingsAccount Class

- We'll now create a subclass SavingsAccount:
- ✅ Inherits all behavior from BankAccount
- ✅ Adds new feature: interest rate
- ✅ Includes method to apply interest

3. What is Polymorphism?

- Polymorphism means “many forms”.
- In Java, it allows objects to be treated as instances of their superclass, while executing the subclass-specific behavior.
- Enables dynamic method dispatch.

Polymorphism in Action

```java
BankAccount account = new SavingsAccount("Anna", 1000, 0.05);
account.deposit(500);
```

- ✅ The reference type is BankAccount
- ✅ The actual object is SavingsAccount
- ✅ Method calls are dynamically resolved at runtime

Benefits of Polymorphism

```java
List<BankAccount> accounts = new ArrayList<>();
accounts.add(new BankAccount("Tom", 1000));
accounts.add(new SavingsAccount("Anna", 2000, 0.03));
```

- ✅ Enables a single interface to represent different underlying forms (data types)
- ✅ Promotes code reuse and scalability

Demo 3 – Show Polymorphism

- ✅ Adding a CheckingAccount subclass with overridden behavior
- ✅ Creating a list of BankAccount references pointing to different subclasses
- ✅ Looping through and invoking overridden methods

4. What is Abstraction?

- Abstraction means hiding implementation details and exposing only the necessary functionality to the user.
- Achieved in Java through abstract classes and interfaces.
- Abstract methods in abstract classes and interfaces define what an object does, not how.

Why Use Abstraction?

- ✅ Simplifies complex systems
- ✅ Makes code more modular and maintainable
- ✅ Encourages separation of concerns
- ✅ Focuses on the "what" not the "how"

Demo 4 — Abstraction in BankAccount Example

- BankAccount is now an abstract class.
- The method endOfMonth() in BankAccount is abstract and does not contain any implementation.
- Subclasses like SavingsAccount and CheckingAccount implement this abstract method (actually they have to).

Interfaces in Java

- An interface in Java defines a contract that implementing classes must fulfill.

- Use interfaces when you want to define a common behavior that multiple, unrelated classes can implement.

Key features of interfaces

- ✅ Only method signatures (until Java 8+ allows default/static methods)
- ✅ No instance fields (only public static final constants)
- ✅ A class can implement multiple interfaces

Abstract Class vs Interface

Feature	Abstract Class	Interface
Inheritance	Single	Multiple allowed
Method Implementation	Yes (concrete methods)	Default methods (Java 8+)
Fields	Instance fields allowed	Constants only
Use Case	”Is-a” relationships	”Can-do” capabilities

- Use an abstract class when classes share state or code.
- Use an interface when classes share behavior, but not implementation.

Demo 5 – Interface-Based Abstraction in Java

- Base Type changed from abstract class BankAccount to interface Account.
- Implementation moved entirely to the implementing class (InvestmentAccount), since interfaces cannot hold shared state.
- Fields like owner and balance are now declared in the implementing class instead of a base class.
- Flexibility: Interfaces support multiple inheritance via implements, unlike abstract classes.
- Emphasizes behavior contracts over shared logic or fields.

Recap of the four pillars of OOP

- ✅ Encapsulation (private fields, public methods)
- ✅ Inheritance (SavingsAccount and CheckingAccount extend BankAccount)
- ✅ Polymorphism (treat all accounts as BankAccount)
- ✅ Abstraction (abstract method endOfMonth() in base class)

Programming Challenge: "Vehicle Rental System"

- Create a simple Vehicle Rental System using Java that applies the four pillars of Object-Oriented Programming:
- ✅ Encapsulation
- ✅ Inheritance
- ✅ Polymorphism
- ✅ Abstraction

Programming Challenge Instructions 1

- ✅ 1. Encapsulation
  - Create a Vehicle class with private fields like brand, model, and rentalPricePerDay.
  - Add constructors and public getters/setters for encapsulation.
- ✅ 2. Inheritance
  - Create two subclasses of Vehicle: Car and Motorcycle, each with an additional specific field (e.g., Car might have numberOfDoors, Motorcycle might have hasSidecar).

Programming Challenge Instructions 2

- ✅ 3. Polymorphism
  - Create a method printRentalInfo(Vehicle vehicle) that takes a Vehicle reference and prints type-specific rental info using overridden toString() or a custom method in each subclass.
- ✅ 4. Abstraction
  - Make Vehicle an abstract class with an abstract method double calculateRentalCost(int days), which is implemented differently in each subclass.

Why this matters when learning Spring

- Spring is built on top of Java – and Java is an object-oriented language
  To understand how Spring works, you need to know:
- ✅ Encapsulation: Keep class details private and control access
- ✅ Inheritance: Reuse code across different parts of your app
- ✅ Polymorphism: Use interfaces and switch out implementations
- ✅ Abstraction: Focus on what a class does, not how
- These ideas help you structure your code clearly and reuse logic, just like Spring does behind the scenes!

Concepts in Action in Spring

- When you build apps with Spring, you will:
- ✅ Create classes with private fields and getters/setters → (Encapsulation)
- ✅ Extend base classes or use shared logic → (Inheritance)
- ✅ Use interfaces like CrudRepository or CommandLineRunner → (Polymorphism)
- ✅ Use annotations like @RestController, @Service without knowing all the internals → (Abstraction)
- Understanding OOP helps you "speak Spring's language" – and makes your learning much smoother!