UNIVERSITETI® METROPOLITAN TIRANA

Course: Object Oriented Programming

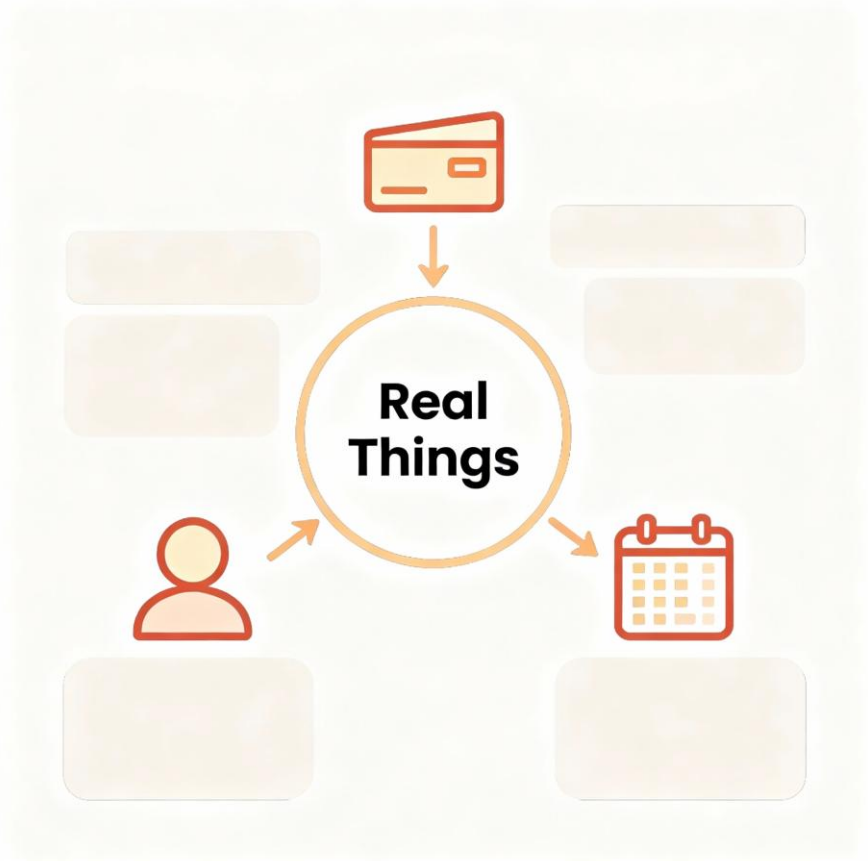
Classes and Objects

The blueprints and instances that build reliable software

Evis Plaku

Model real entities with code that's easier to grow

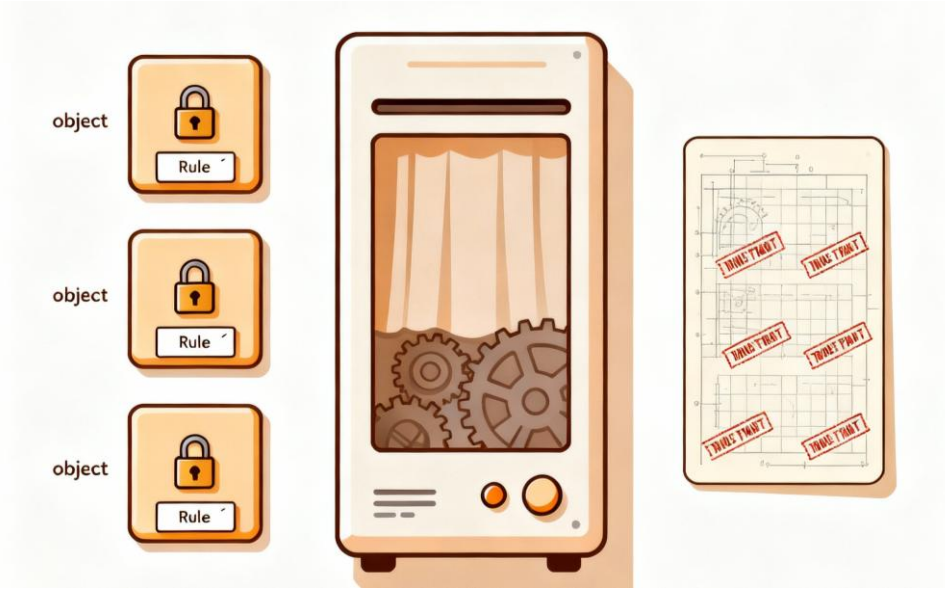
- Code mirrors real things: **objects** with **data** and **actions** acting together
- Classes define shared design; **objects are concrete**, with their own state
- This mapping keeps programs understandable, extendable, and safer over time



UNIVERSITETI METROPOLITAN TIRANA

Clear structure, safer changes, reusable behaviors

- Group data with behavior, so each object guards its own rules
- Hide internals; expose small methods, enabling safe, focused changes later
- Reuse class designs to create many objects, reducing duplication across code



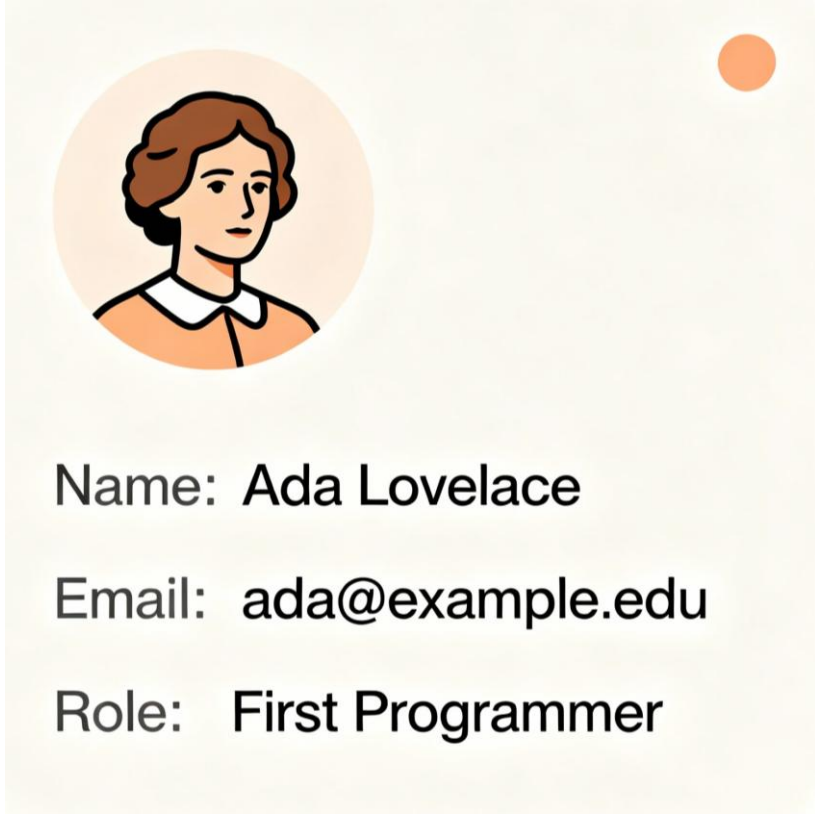
Objects | Internal mechanisms | Class as a blueprint

A class defines state and behavior no concrete data yet

- A class is a **user-defined type** describing fields and methods precisely
- It specifies valid data and actions; it does not hold any actual values

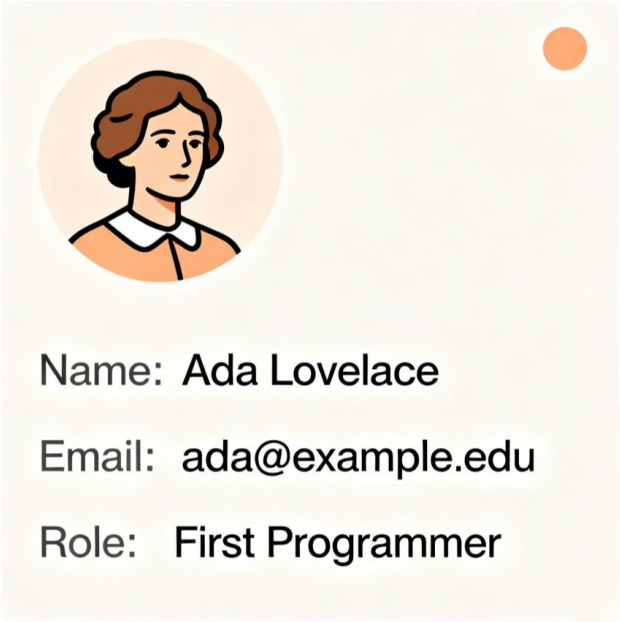
A live entity with actual data and identity

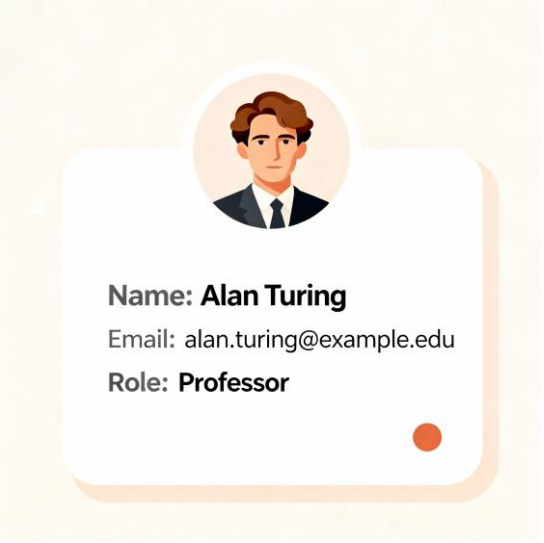
- An object is a created entity with its own data and behavior
- It holds current values in fields and runs methods that act on them

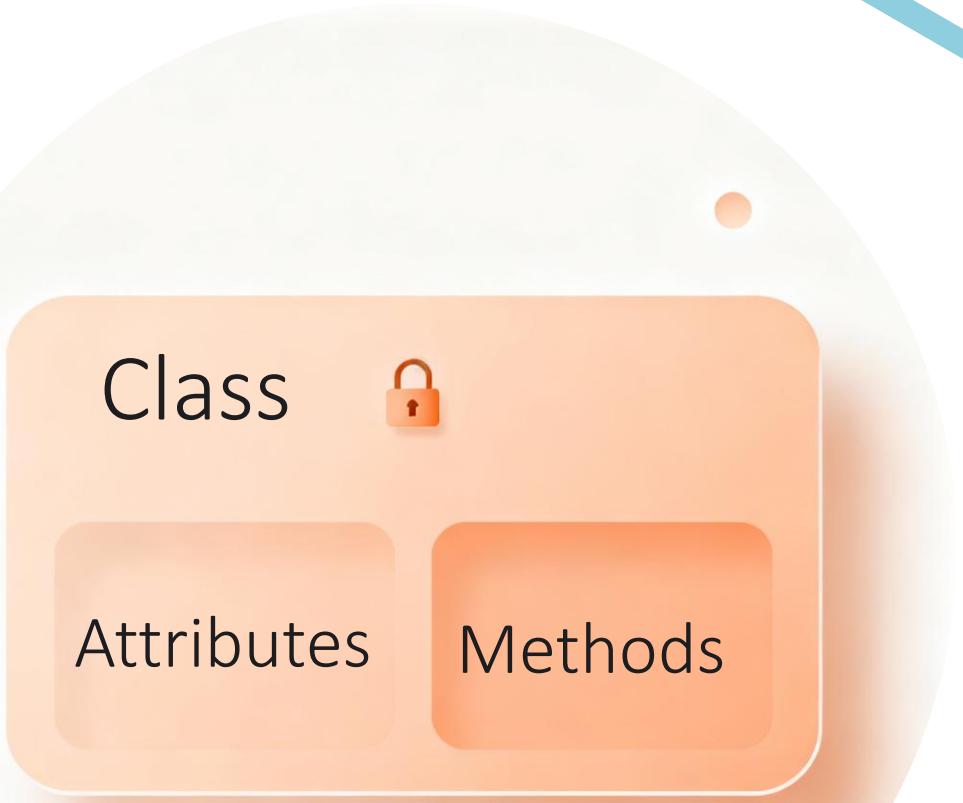


Instances share a design but hold different state

- Fields are named variables inside a class that hold an object's state
- Methods are functions inside a class that operate on that object's state
- Clear rule: fields store data; methods expose safe, well-defined behavior







Class Essentials

State in fields; actions in methods

- One class defines the shared fields and methods all instances will have
- Each object stores its own values for those fields at a given time
- Multiple instances follow one class design, each with unique state

```
public class User {
    // fields
    private String name;
    private String email;
    private String role;


    // method signature (no implementation)
    public void updateProfile(
      String newName, String newEmail, String newRole) {
}
```

Set valid initial state when creating objects

- Special method run at creation that prepares fields to valid values
- Often takes parameters to require needed data and enforce class rules
- Sets things up so the object starts out valid

```
public class User {
    // ... fields

    public User(String name, String email, String role) {
        this.name = name;
        this.email = email;
        this.role = role
    }
    // ... rest of the class
}
```

Control access; guard invariants

- Keep fields private; expose small public methods to read or change values
- Validate in setters; never allow invalid data to enter the object
- Simple rule: get reads safely; set updates safely with checks

```java
public class User {
    // A private field to hold the user's name
    private String name;
    // ... rest of the properties

    // A public "getter" method to read the name
    public String getName() {
        return name;
    }

    // A public "setter" method to update the name
    public void setName(String newName) {
        this.name = newName;
    }
    // ... rest of the methods
}
```

Human-readable summaries for logs and debugging

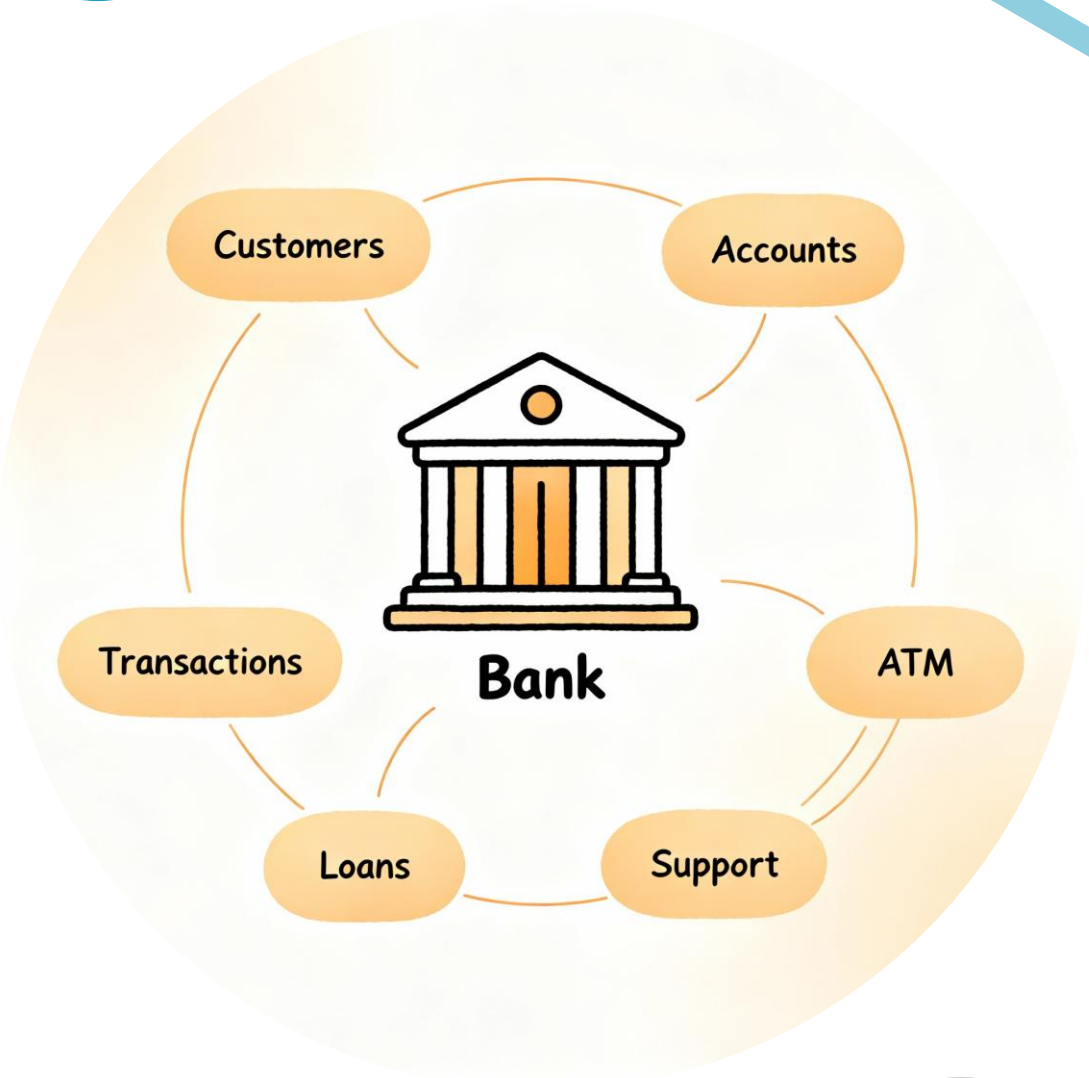
- Returns a clear text view of an object's current important fields
- Override it to show meaningful details instead of class name and hash
- Simple rule: make prints and logs useful at a glance

```java
public class User {
    private String name;
    private String email;
    private String role;

    // ... (constructors, getters, and setters would be here)

    /**
     * Returns a string representation of the User object,
     * including name, email, and role for clear logging.
     */
    @Override
    public String toString() {
        return "User{" +
                "name='" + name + '\'' +
                ", email='" + email + '\'' +
                ", role='" + role + '\'' +
                '}';
    }
}
```

Express domain behaviors clearly and simply

- Put real actions in methods: what the object should do in the domain
- Name methods by intent: transfer, closeAccount, generatePassword, calculateInterest
- Keep methods small, with clear rules and checks for valid outcomes

```java
import java.util.UUID;

public class User {

    /**
     * Generates a simple, random password for demonstration.
     * @return A randomly generated 8-character string.
     */
    public String generatePassword() {
        // Create a random string and return the first 8 characters.
        String randomString = UUID.randomUUID().toString();
        return randomString.substring(0, 8);
    }
}
```

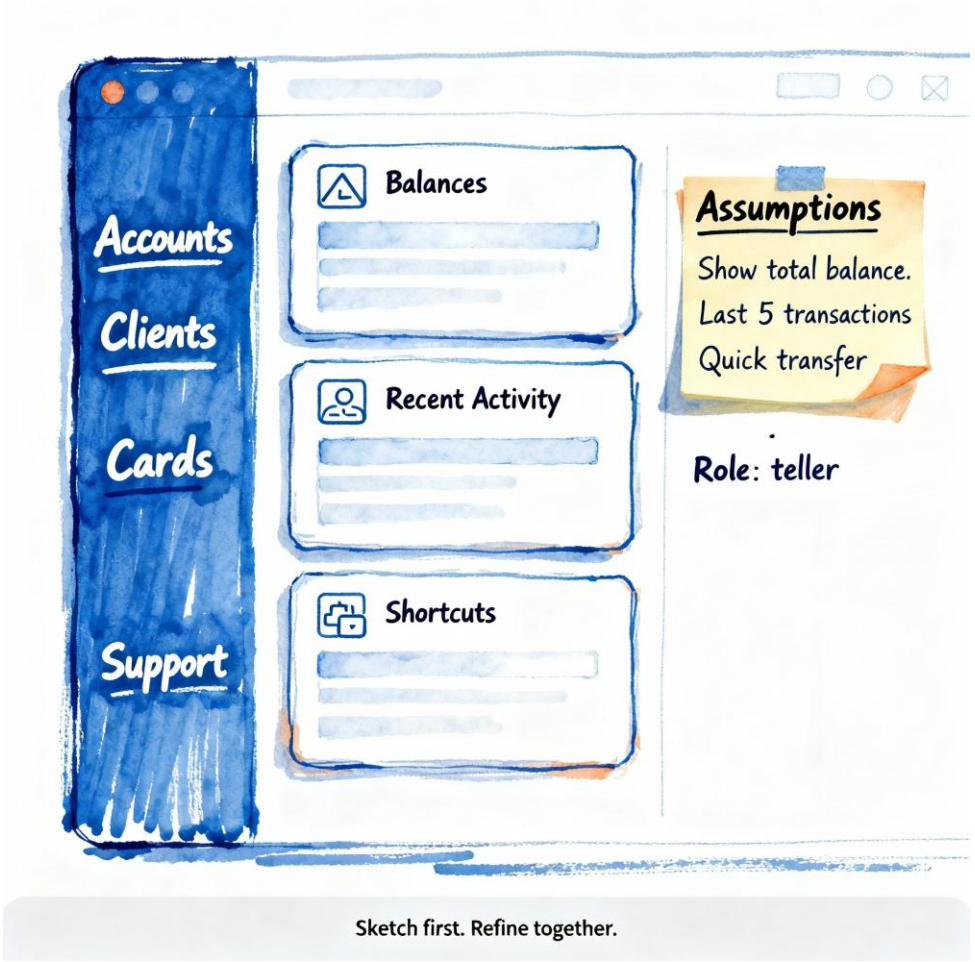


Bank System Example

UNIVERSITETI METROPOLITAN TIRANA

A system to manage customers, their accounts, and core financial activities

- **Purpose:** model key banking functions, focusing on safe customer and financial management
- **Key entities:** Customers (the people), Accounts (the money holders), and Transactions (the actions)
- **Key operations:** open/close accounts, deposit/withdraw funds, check balances, and transfer money



Customers hold accounts; Accounts hold money

Customer

- The central entity, representing the owner with a unique ID and contact info
- All financial activity begins with a customer

Account

- A container for money, identified by an account number and owned by a customer
- It holds the balance and has rules for deposits and withdrawals

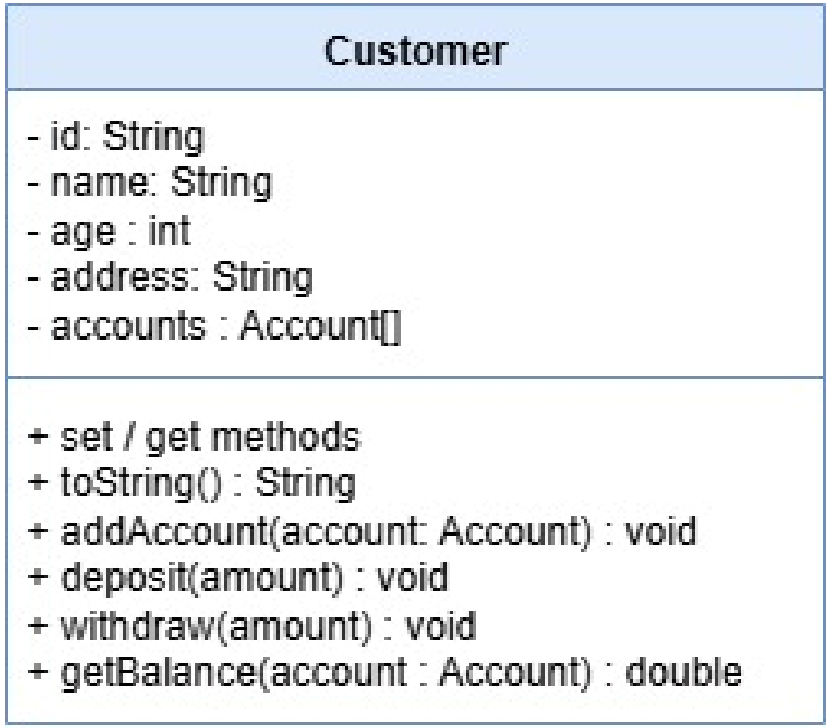
Transaction

- An immutable record of a single financial event, like a deposit or transfer
- It provides the essential audit trail for every account

UNIVERSITETI METROPOLITAN TIRANA

A customer is a key entity in the bank management system

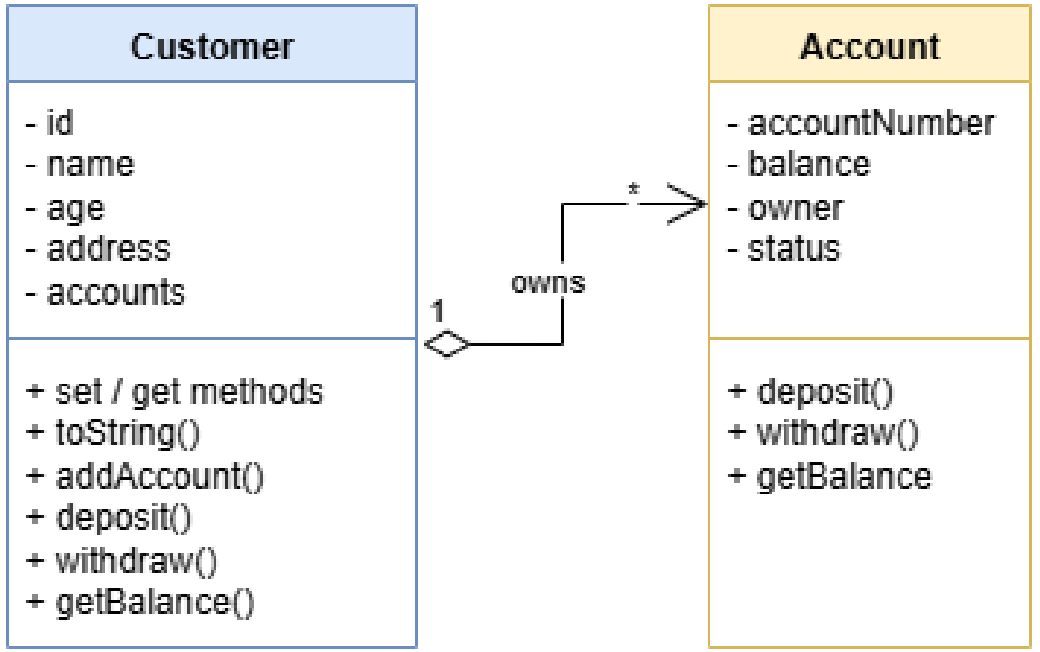
- Identified by a unique ID, it holds essential personal data like name and email
- Maintains a list of associated Account objects, modeling the ownership relationship
- Acts as the central entity for a user, initiating actions like opening new accounts



UNIVERSITETI METROPOLITAN TIRANA

An account is a key entity in the bank management system

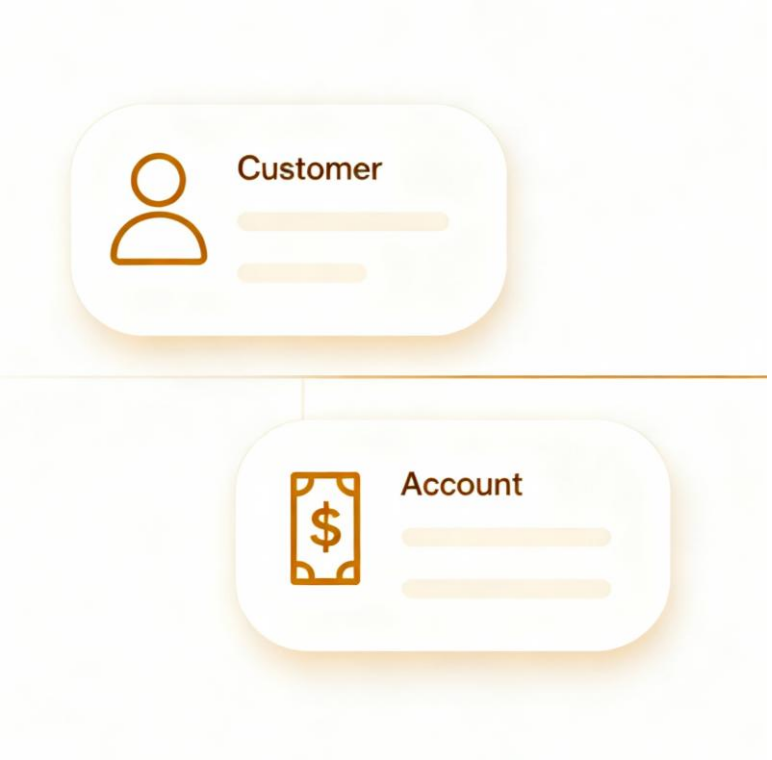
- Uniquely identified by an accountNumber, it's a specific, addressable financial record
- Tracks the current balance and maintains a clear link back to its owner
- Its status (e.g., OPEN, FROZEN) dictates which operations are currently permitted



Design with purpose, keep classes focused, and protect your data

Single Responsibility

- Keep Customer and Account distinct
- The Customer class should only manage customer information (name, contact details)
- The Account class should only manage financial state (balance, status)



Avoid mixing responsibilities

Design with purpose, keep classes focused, and protect your data

Encapsulation is key

- Hide internal data by making fields private
- All changes to state, like updating a balance or changing an email, should happen through public methods (deposit(), updateEmail())
- Add validation and business rules to protect the integrity of your objects

Don’t expose what is not necessary

Design with purpose, keep classes focused, and protect your data

Design for clarity and independence

- A Customer should have a unique, unchangeable ID. Its methods should reflect its role: updateContactInfo(), viewOwnedAccounts()

- An Account must always have an owner and a unique account number. Its methods should manage its financial state: deposit(), withdraw(), getBalance()

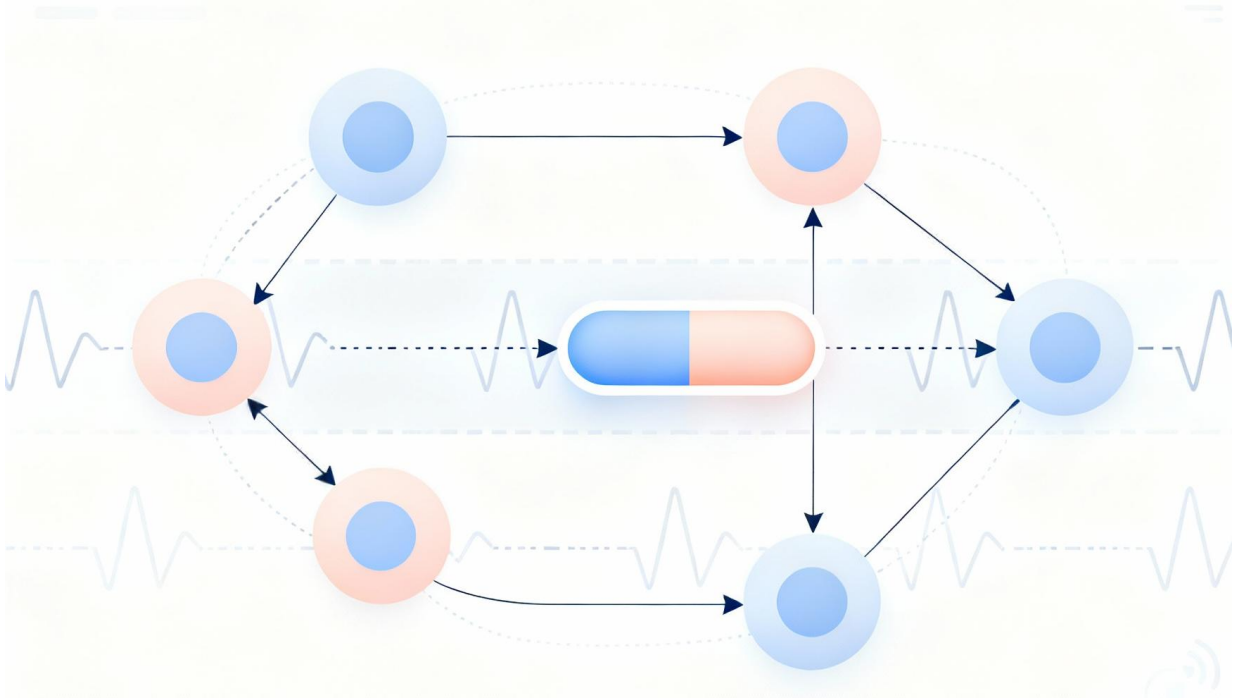
Connect independent objects

UML to Design

UML: a standardized visual language to model systems before and during development

- Defines a common way to visualize, specify, construct, and document software and business systems (we'll discuss it briefly)

- Offers multiple diagram types to capture structure (what exists) and behavior (what happens) across the lifecycle



Class diagram: graphical notation used to construct and visualize object-oriented systems

- Describes the structure of a system by showing
  - classes,
  - their attributes,
  - operations (or methods),
  - and the relationships among objects

A Class is a blueprint for an object

An object is an instance of a class

UNIVERSITETI METROPOLITAN TIRANA

A class represent a concept which encapsulates state (attributes) and behavior (operations)

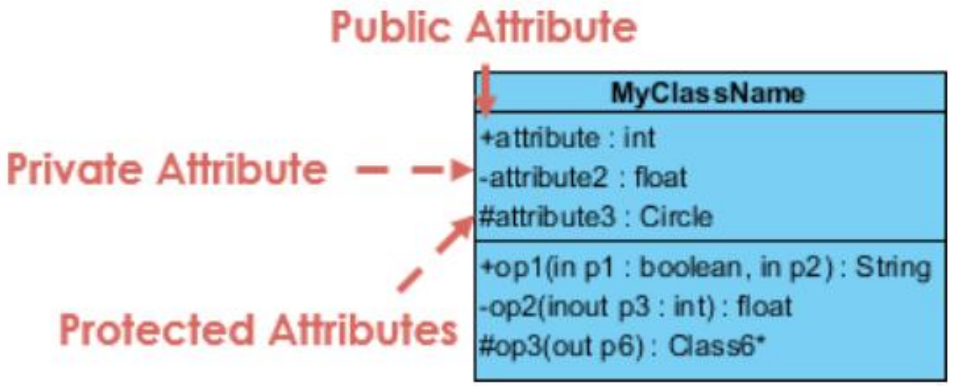
- Each attribute has a type
- Each operation has a signature
- The class name is the only mandatory information

- The name of the class appears in the first partition
- Attributes are shown in the second partition
- Operations are shown in the third partition

UNIVERSITETI METROPOLITAN TIRANA

Attributes and methods can have various visibilities

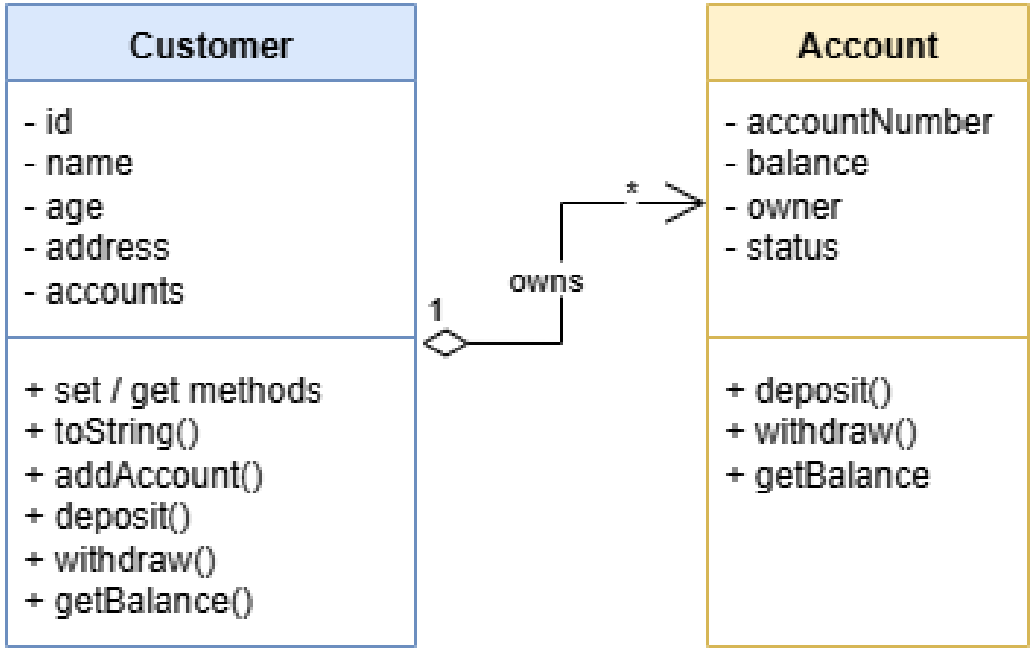
- The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation



- \+ denotes public attributes or operations
- \- denotes private attributes or operations
- # denotes protected attributes or operations

UNIVERSITETI METROPOLITAN TIRANA

Customer: represents the owner of bank accounts

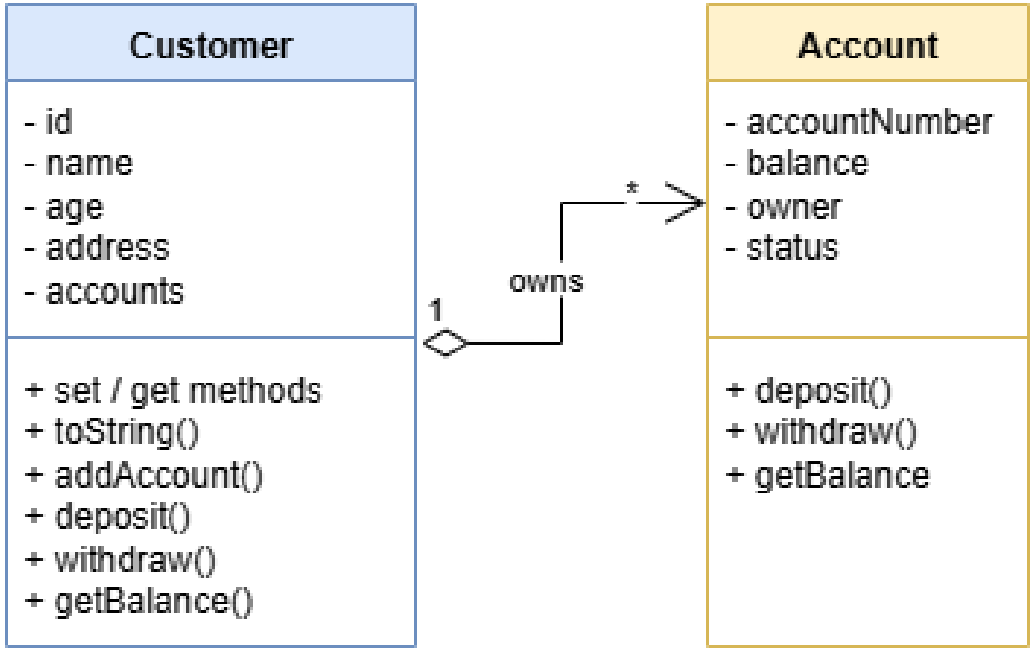



```java
public class Customer {
    private String name;

    public Customer(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Customer{name='" + name + "'}";
    }
}
```

basic implementation; will expand

UNIVERSITETI METROPOLITAN TIRANA

Account: manages money safely via controlled operations




```java
public class Account {
    private double balance;

    public Account(double initialBalance) {
        this.balance = initialBalance;
    }

    public void deposit(double amount) {
        if (amount <= 0) return;
        balance += amount;
    }

    public boolean withdraw(double amount) {
        if (amount <= 0 || amount > balance) return false;
        balance -= amount;
        return true;
    }

    public double getBalance() {
        return balance;
    }

    @Override
    public String toString() {
        return "Account{balance=" + balance + "}";
    }
}
```

Classes define structure; objects are live instances with state and behavior

- A class is a blueprint that specifies attributes and methods
- An object is a concrete instance created from that blueprint

- Clear class design (single responsibility, meaningful names) makes systems easier to understand, extend, and test

UNIVERSITETI METROPOLITAN TIRANA

Keep what matters

Leave what doesn’t