UNIVERSITETI® METROPOLITAN TIRANA

Course: Object Oriented Programming

Inheritance & Polymorphism

Reuse the common, specialize the rest

Evis Plaku

Build flexible software by reusing code and extending behavior intelligently

- Inheritance lets you define shared features once, then specialize
- Polymorphism allows different types to work together through common interfaces

Essential for building scalable, maintainable systems that grow with requirements

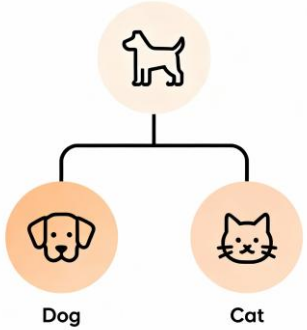
Copying similar code across classes creates maintenance nightmares and bugs

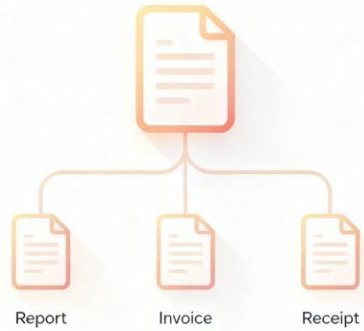
- Similar entities share common features but differ in specific details
- Copy-paste approach means fixing bugs in multiple places simultaneously
- Adding new features requires updating every duplicated version individually

UNIVERSITETI METROPOLITAN TIRANA

Natural hierarchies move from general concepts to specific instances everywhere

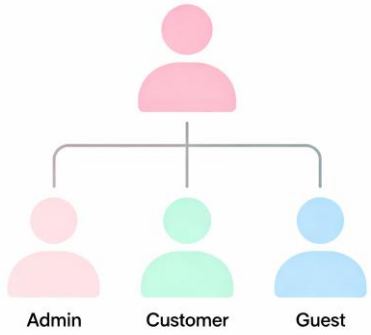
- Animals: Mammal → Dog, Cat; each adds specific traits





- Documents: Document → Report, Invoice, Receipt; shared formatting, unique content rules

- Users: User → Admin, Customer, Guest; common authentication, different permissions



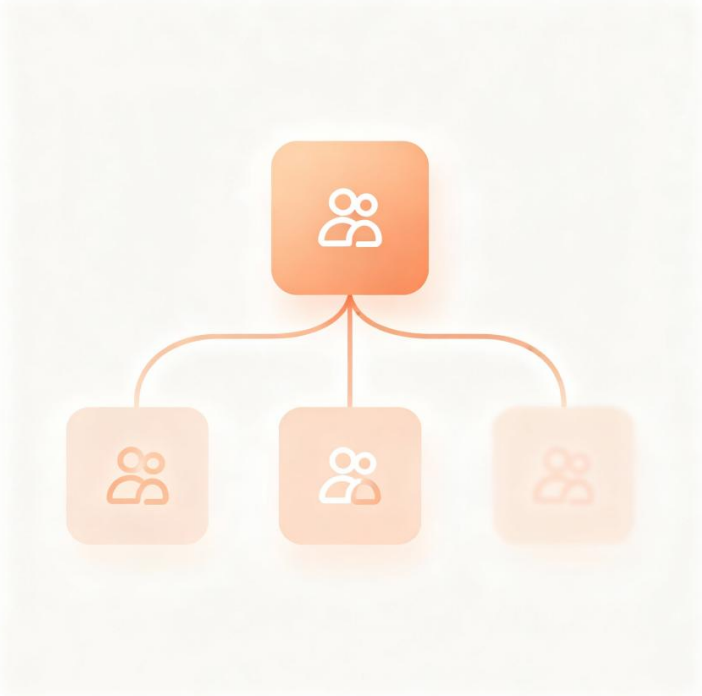
Don't Repeat Yourself

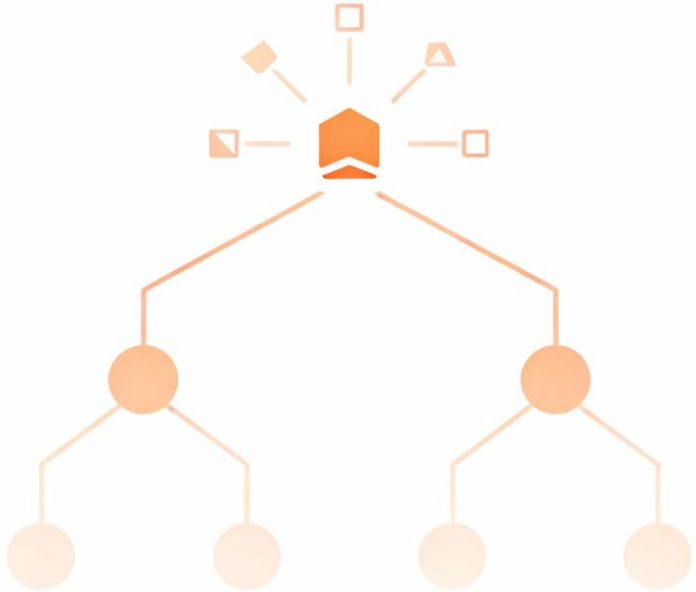
define shared behavior once in the parent class

- Write common logic in one place;
  all children inherit automatically
- Bug fixes and improvements propagate to all
  specialized types instantly
- Reduces codebase size, increases consistency,
  simplifies maintenance significantly

Learn inheritance for code reuse and polymorphism for flexible design

- Create parent classes that capture shared structure and behavior
- Build specialized child classes that extend and customize parent functionality
- Use polymorphism to write code that works with current and future types

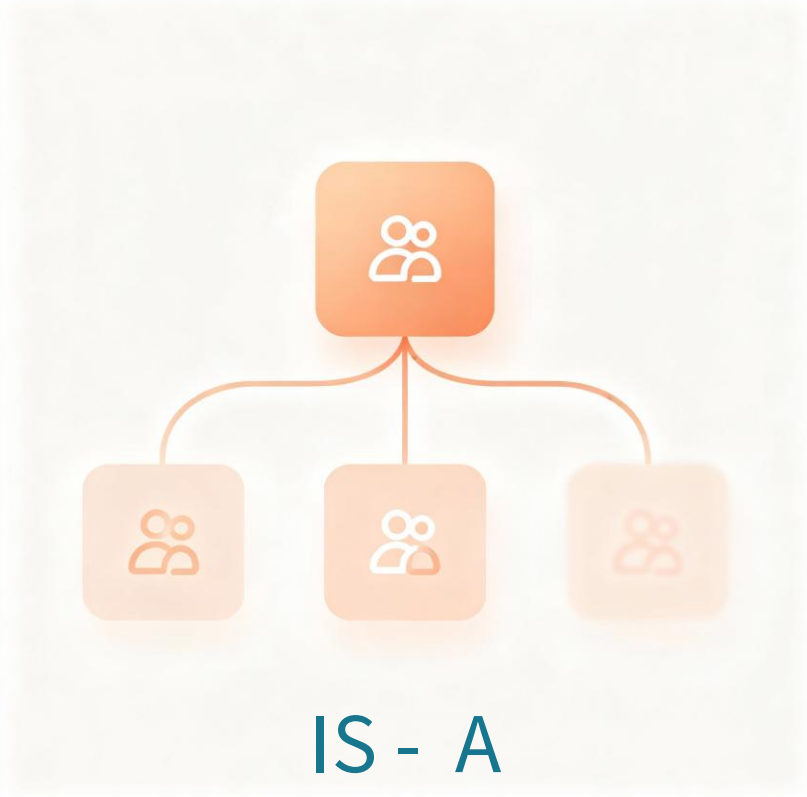




Core Concepts

Parent class defines common features; children inherit and add specifics

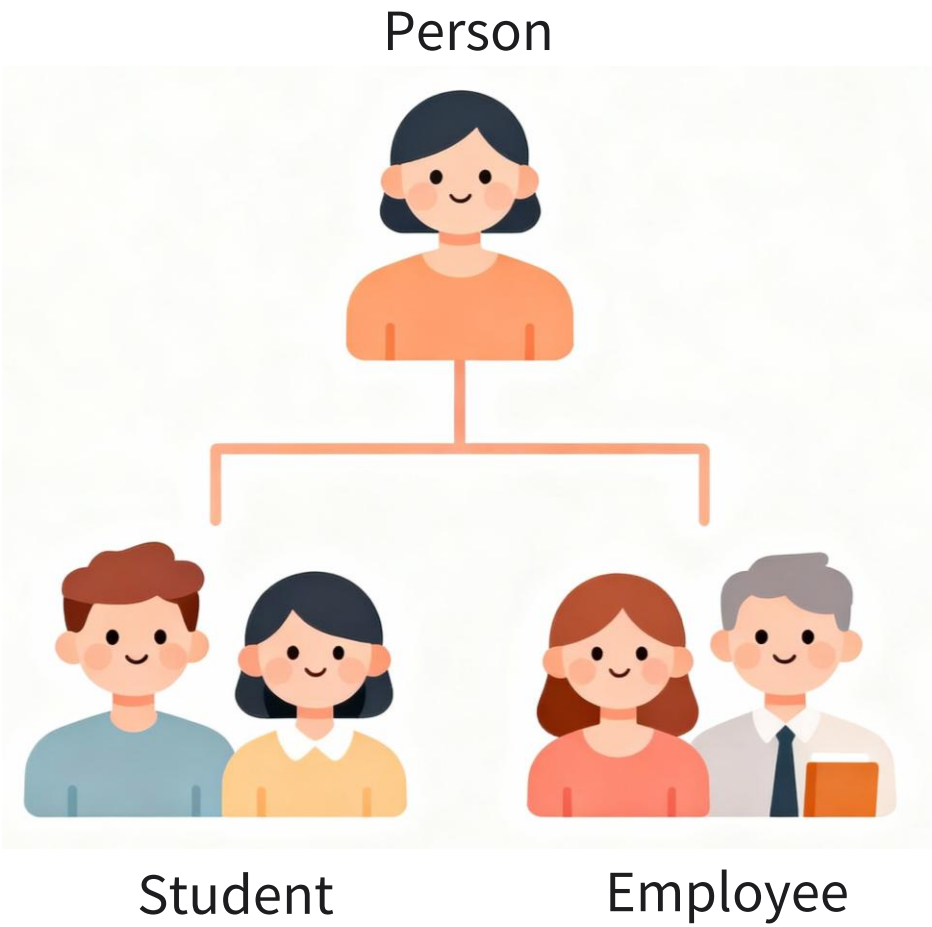
- Superclass (parent) contains fields and methods shared by all children
- Subclass (child) uses **`extends`** keyword to inherit everything from parent
- Creates "is-a" relationship
  Student IS-A Person, Employee IS-A Person



UNIVERSITETI METROPOLITAN TIRANA

Person holds name and age
Student and Employee add specialized fields

- Person: id, name, age, address, displayInfo() method
- Student adds: major, GPA tracking
- Employee adds: salary, department, employee benefits



UNIVERSITETI METROPOLITAN TIRANA

Base class defines common state and behavior all people share

- `protected` allows subclasses to access these fields directly
- Constructor initializes common data every person needs
- `displayInfo()` provides default behavior children can override

```java
public class Person {
    protected String name;
    protected int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void displayInfo() {
        System.out.println(name + ", " + age + " years old");
    }
}
```

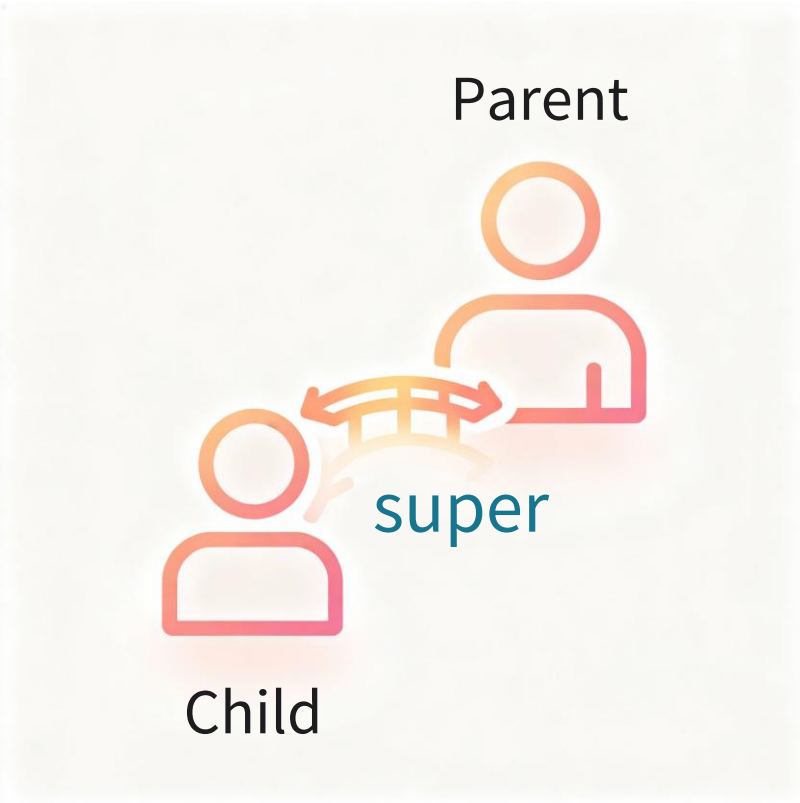
Student inherits Person's fields and adds student-specific data and behavior

- `extends` Person establishes inheritance relationship
- `super(name, age)` calls parent constructor to initialize inherited fields
- Override `displayInfo()` to add student-specific details after parent information

```java
public class Student extends Person {
    private int studentId;
    private String major;

    public Student(String name,
            int age, int id, String major) {
        super(name, age);
        this.studentId = id;
        this.major = major;
    }

    @Override
    public void displayInfo() {
        super.displayInfo();  // Call parent version
        System.out.println("Student ID: " + studentId
             + ", Major: " + major);
    }
}
```

Access parent constructors and methods from within child classes using super

- `super()` calls parent constructor; must be first line in child constructor
- `super().methodName` calls parent's version of an overridden method
- Ensures parent initialization happens before child adds its own setup



Children replace parent method implementations to provide specialized behavior

```java
public class Employee extends Person {
    private double salary;

    @Override
    public void displayInfo() {
        System.out.println(name + ", Salary: €" + salary);
    }
}
```

- `@Override` annotation tells compiler you're replacing parent method
- Method signature must match parent exactly (name, parameters, return)
- Child's version executes when called on child objects

Protected fields are accessible within class, subclasses, and same package

- `private:` only this class sees it
- `public:` everyone everywhere can access it
- `protected:` this class, all subclasses, and package classes

Parent references can hold child objects; behavior resolves at runtime

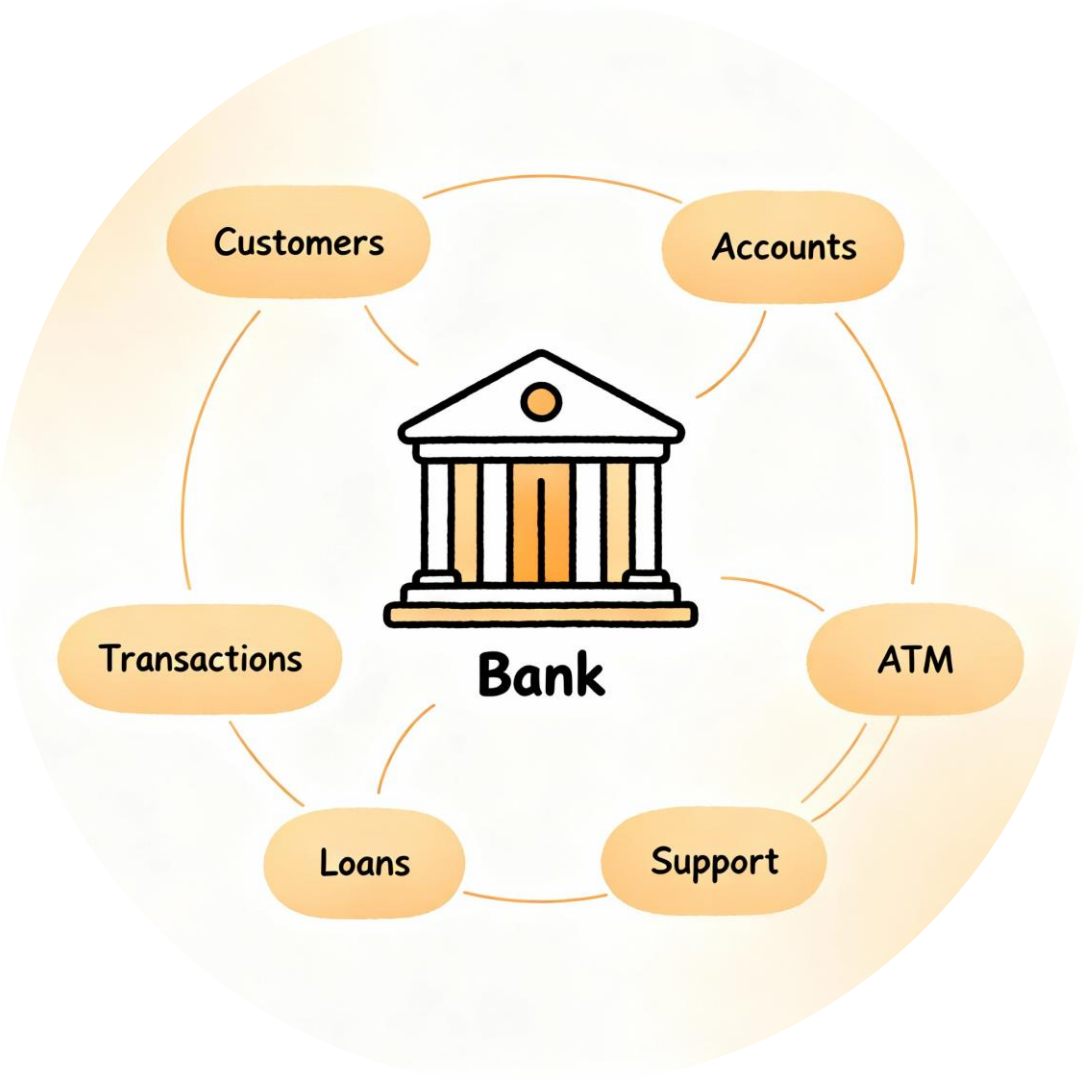
- Parent type Person holds any child type object
- Method calls execute child's version automatically
- Decided at runtime based on actual object type

```
Person p1 = new Student("Ada", 20, 101, "CS");
Person p2 = new Employee("Alan", 30, 50000);
Person p3 = new Person("Grace", 40);

p1.displayInfo();  // Calls Student version
p2.displayInfo();  // Calls Employee version
p3.displayInfo();  // Calls Person version
```

Single collection holds mixed types; each behaves according to actual class

- `ArrayList<Person>` accepts Student, Employee, and Person objects
- Loop treats all uniformly as Person references
- Each object executes its own `displayInfo()` implementation automatically

```
ArrayList<Person> people = new ArrayList<>();
people.add(new Student("Ada", 20, 101, "CS"));
people.add(new Employee("Alan", 30, 50000));

for (Person p : people) {
    p.displayInfo();  // Correct version for each type
}
```



Bank System Example

Multiple account types share core banking operations but differ in rules

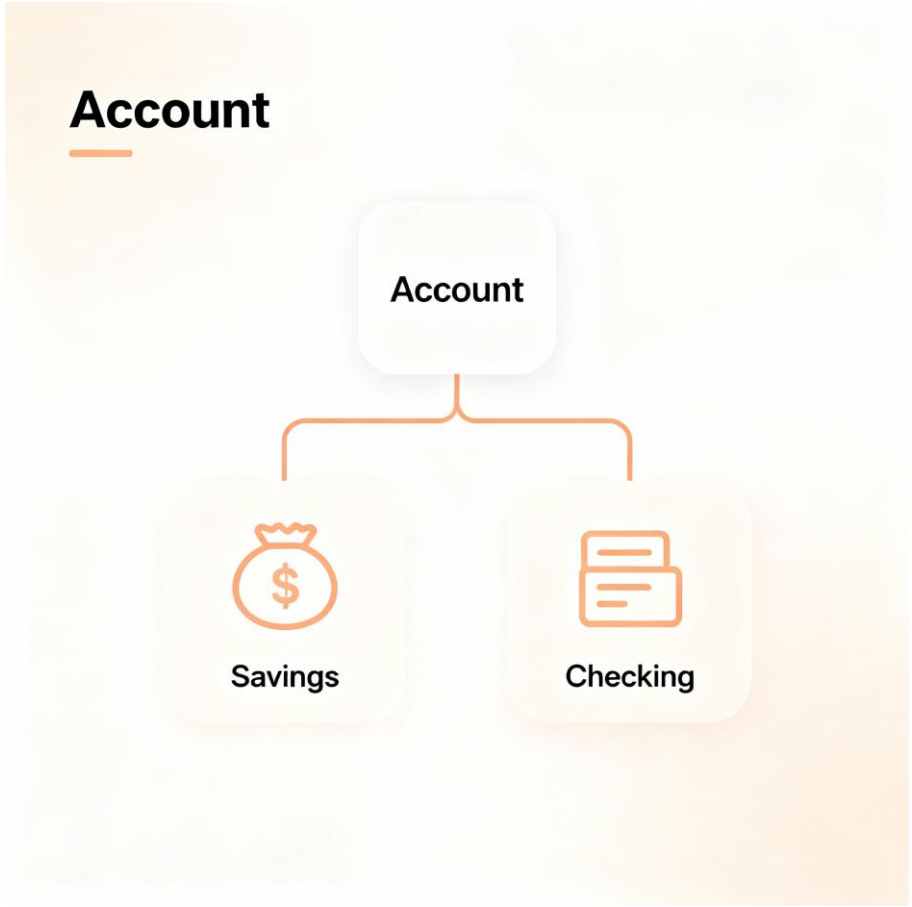
- CheckingAccount, SavingsAccount, InvestmentAccount all need balance tracking and transactions
- Each type enforces different withdrawal rules and business constraints

Inheritance captures commonality

Overriding customizes behavior per account type

Base Account holds shared fields children add type-specific features and rules

- Account base class: accountId, balance, isFrozen, deposit(), withdraw()
- CheckingAccount: adds overdraft limit for negative balances
- SavingsAccount: adds interest rate and minimum balance enforcement



Common fields and operations all account types share in one place

- Static counter generates unique IDs automatically
- Protected fields allow subclasses direct access when needed
- Base withdraw() prevents overdrafts; children can override

```
public class Account {
    private static int accountCounter = 1000;
    protected int accountId;
    protected double balance;
    protected boolean isFrozen;

    public boolean deposit(double amount);
    public boolean withdraw(double amount);
    public double getBalance();
}
```

Checking accounts allow overdrafts up to a specified limit for flexibility

- Adds overdraftLimit field not present in base Account
- Overrides withdraw() to allow negative balance up to limit
- Calls super() to initialize parent Account fields first

```java
public class CheckingAccount extends Account {
    private double overdraftLimit;

    @Override
    public boolean withdraw(double amount) {
        if (isFrozen || amount <= 0) return false;
        if (balance - amount < -overdraftLimit) {
            System.out.println("Exceeds overdraft limit");
            return false;
        }
        balance -= amount;
        return true;
}
```

Savings accounts enforce minimum balance and calculate interest on deposits

- Adds interestRate and minimumBalance constraints
- Override withdraw() to enforce minimum balance rule
- New method applyInterest() specific to savings accounts only

```java
public class SavingsAccount extends Account {
    private double interestRate;
    private double minimumBalance;


    @Override
    public boolean withdraw(double amount) {
        if (isFrozen || amount <= 0) return false;
        if (balance - amount < minimumBalance) {
            System.out.println("Cannot go below minimum balance");
            return false;
        }
        balance -= amount;
        return true;
    }

    public void applyInterest() {
        balance += balance * interestRate;
    }
}
```

Customer's account list holds all account types; single collection manages diversity

- List holds all types of Account objects
- addAccount() works with any current or future subclass
- getTotalBalance() treats all types uniformly

```java
public class Customer {
    private ArrayList<Account> accounts;

    public boolean addAccount(Account account) {
        if (account == null) return false;
        accounts.add(account);  // Accepts any Account subclass
        return true;
    }

    public double getTotalBalance() {
        double total = 0.0;
        for (Account acc : accounts) {
            total += acc.getBalance();  // Works for all types
        }
        return total;
    }
}
```

Each account type executes its own validation rules when withdraw() is called

- Same withdraw() call behaves differently for each account type

```
Customer customer = new Customer(1, "Ada Lovelace", 30, "London");

customer.addAccount(new CheckingAccount(500, 200));         // ID 1001
customer.addAccount(new SavingsAccount(5000, 0.03, 1000)); // ID 1002

// Try to withdraw from each account
customer.withdraw(1001, 600);  // Works: overdraft allowed
customer.withdraw(1002, 4500); // Fails: below minimum balance
```

- CheckingAccount checks overdraft limit
- SavingsAccount checks minimum balance

Add new account types without changing existing Customer or main code

- Customer code works with all current and future Account types
- getTotalBalance(), displayAllAccounts() handle any account automatically
- New type? Extend Account, override what's needed, add to system

Inheritance eliminates duplication polymorphism enables flexibility and extensibility

- Write shared code once in parent; children inherit and specialize
- Treat different types uniformly while preserving unique behavior
- Banking system now handles diverse account types elegantly and extensibly

UNIVERSITETI METROPOLITAN TIRANA

Grow from strong roots

Branch with purpose