Course: Object Oriented Programming

Abstract Classes and Interfaces

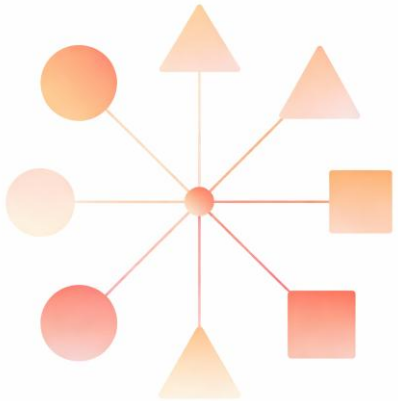
Defining What Must Be, Allowing What Can Be

Evis Plaku

Two powerful mechanisms for designing robust extensible object-oriented systems

- Moving beyond concrete inheritance to contracts and architectural constraints
- Preventing common design mistakes through compiler-enforced rules
- Enabling code reuse while guaranteeing implementation of critical methods

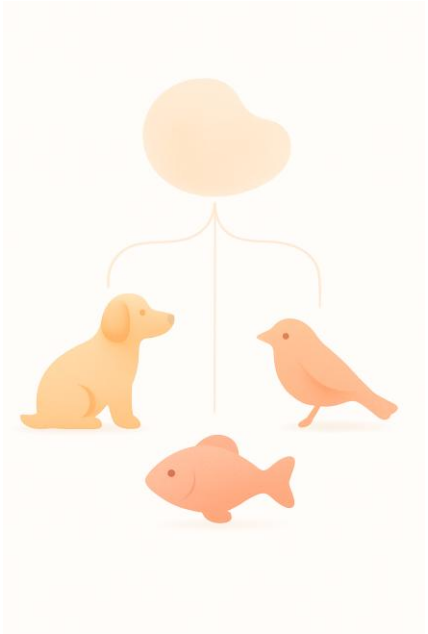
Abstract



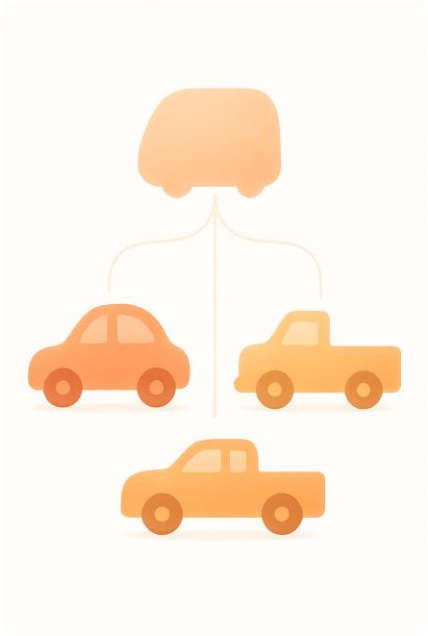
Common contract

Some classes represent concepts too abstract to exist as standalone objects

- Example: Animal is a concept but "generic animal" isn't real



- Example: Shape defines common properties but has no specific area



- Example: Vehicle shares features but lacks concrete drive mechanism

Allowing instantiation of conceptual parent classes creates meaningless incomplete objects

- `Animal animal = new Animal()`
  makes no sense without species
- `Shape shape = new Shape()`
  cannot calculate area without type

Need mechanism preventing instantiation while preserving hierarchy benefits

Inheritance alone cannot force subclasses to implement required methods

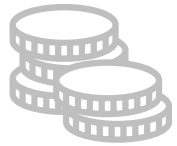
- Developer might forget to override critical methods
- Compiler accepts incomplete subclasses *catching errors only at runtime*

```
public class User {
    void audit() { } // meant to log actions, but empty default
}

public class AdminUser extends User { } // compiles, developer forgot audit()
// Later: admin.audit(); // silently does nothing → security gap
```

- Documentation insufficient to enforce architectural requirements across team

Unrelated classes often need to share behaviors without common parent

- Example: CreditCard, PayPal, BankTransfer all need a Payable interface(`processPayment()`, `validate()`, `refund()`)

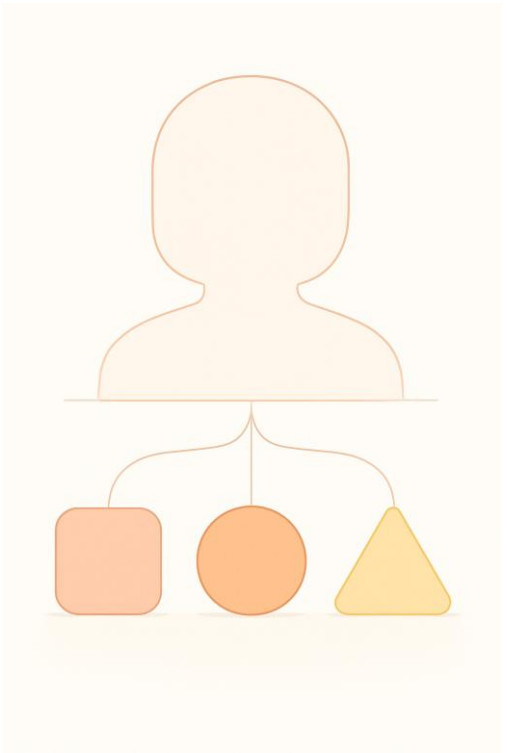


- Example: Authentication providers `GoogleAuth`, `LocalAuth`, `TokenAuth` all implement `Authenticatable`

- Cannot use inheritance; these are fundamentally different hierarchies

We need mechanisms enforcing contracts enabling reuse across different hierarchies

- Prevent instantiation of conceptually incomplete parent classes
- Force subclasses to implement required methods at compile time
- Allow unrelated classes to guarantee shared capabilities



Java provides abstract classes for hierarchies and interfaces for capabilities

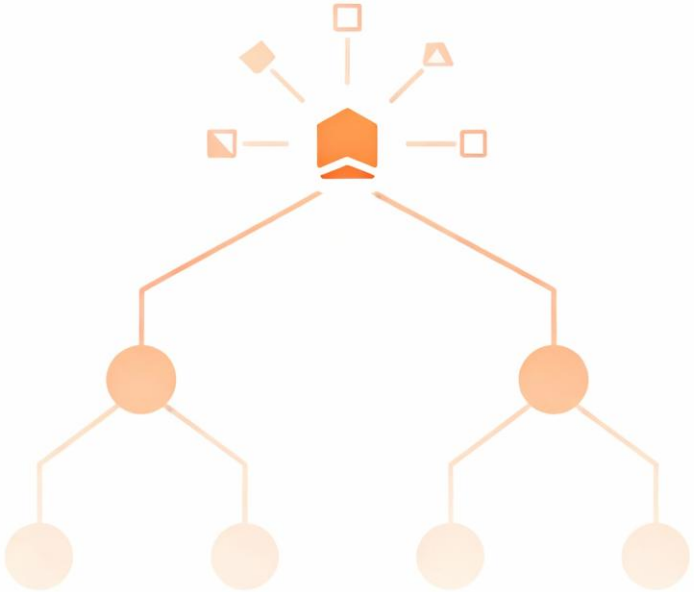
Abstract classes

Partial implementation with enforced customization points

Interfaces

Pure contracts defining capabilities across hierarchies

Code reuse with flexible contracts



Core Concepts

Abstract classes cannot be instantiated but provide foundation for subclasses

- Declared with abstract keyword
- Cannot create instances, it causes compilation error
- Must be extended by concrete subclasses to be useful

```
abstract class User {              // abstract class
    abstract void login();         // abstract method
}

class AdminUser extends User {  // concrete subclass
    void login() {
        System.out.println("Admin logged in");
    }
}

// User u = new User();
// compilation error: cannot instantiate abstract class
AdminUser admin = new AdminUser();
admin.login();                     // works
```

Abstract methods declare signatures without implementation forcing subclasses to provide logic

- Example: `public abstract double calculateSalary();`
  no method body
- Subclasses must implement or become abstract themselves
- Compiler enforces implementation preventing forgotten methods

Abstract classes mix implemented methods with abstract ones enabling code reuse

- Example: User has concrete login() all users authenticate similarly
- Example: User has abstract getPermissions() each role differs
- Subclasses inherit working code and implement only what's unique

```
abstract class User {
    void login() { // shared concrete method
        System.out.println("Logging in...");
    }
    // abstract: each role differs
    abstract String getPermissions();
}

class AdminUser extends User {
    String getPermissions() {
        return "READ, WRITE, DELETE";
    }
}

class RegularUser extends User {
    String getPermissions() {
        return "READ";
    }
}
```

User hierarchy demonstrates abstract parent with concrete specialized subclasses

- Abstract: `public abstract class User {abstract String getRole();}`

- Concrete: Student tracks courses;
  Instructor manages classes;
  Admin controls system

User

Student

Instructor

Admin

- Prevents meaningless generic users guarantees role identification exists

Interfaces define pure contracts with method signatures but zero implementation

- Declared with interface keyword
- Contains only method signatures
- Implementing classes must provide all method bodies

```java
// Declare interface
public interface Notifiable {
    void sendNotification(String msg); // only method signature
}

// Implementing class must provide method body
class EmailNotification implements Notifiable {
    public void sendNotification(String msg) {
        System.out.println("Email: " + msg);
    }
}

class SMSNotification implements Notifiable {
    public void sendNotification(String msg) {
        System.out.println("SMS: " + msg);
    }
}
```

Classes implement interfaces using implements keyword creating contractual obligations

- Syntax: `public class MyClass implements MyInterface`
- Must implement every interface method
  or become abstract
- Multiple interfaces allowed:
  `class MyClass implements FirstInterface, SecondInterface`

Abstract classes share code within hierarchies
Interfaces define contracts across hierarchies

- Abstract class: can have fields constructors, **concrete methods**, single inheritance
- Interface: **only method signatures**, no state, multiple implementation

Choose based on relationship:

is-a uses abstract;

can-do uses interface

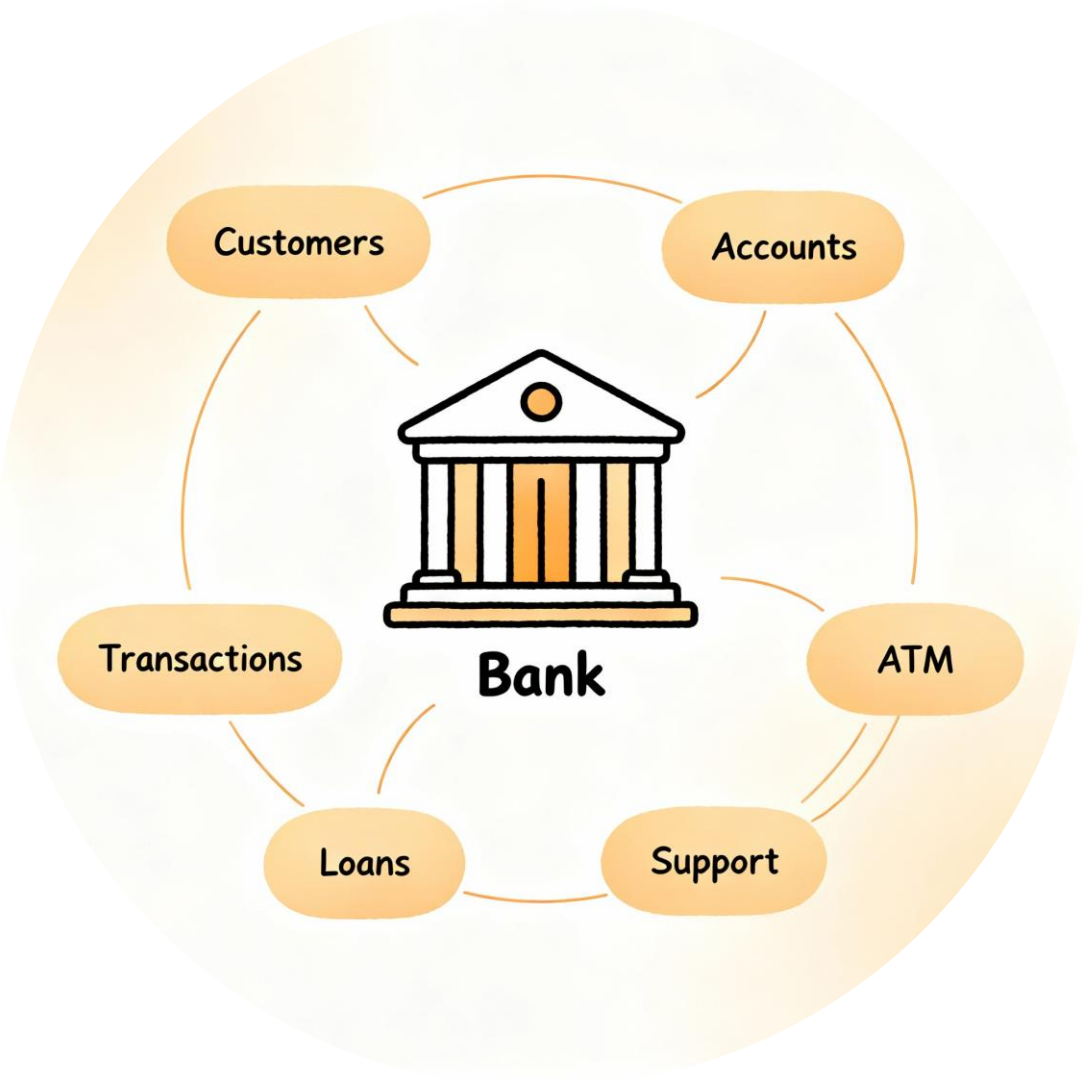
UNIVERSITETI METROPOLITAN TIRANA

Abstract classes

- When subclasses share significant code but need customization
- Common implementation exists
- Closely related classes form natural hierarchy

Interfaces

- Capabilities shared across unrelated classes
- Support multiple inheritance
- Objects of different types need to share a behavior



Bank System Example

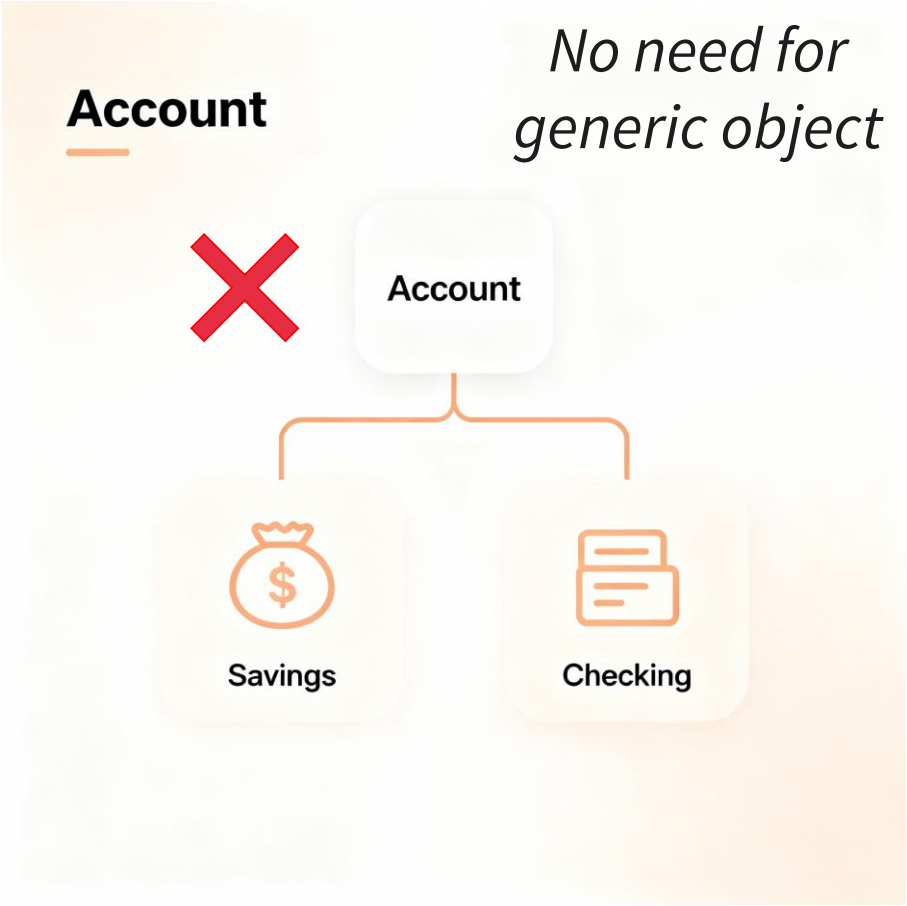
Current system uses concrete Account class as parent for specialized types

- Account is concrete: can instantiate generic `new Account(1000)`
- SavingsAccount and CheckingAccount extend Account
- Problem: generic accounts semantically meaningless in real banking

```
// Concrete base class
public class Account {
    // properties and methods
}

// Subclasses inherit and can add unique behavior
class SavingsAccount extends Account {

}

class CheckingAccount extends Account {

}

// Usage: Account can instantiate
Account generic = new Account(1000);
```

Banks offer specific account types never generic unspecialized accounts

- Real scenario: Customer must choose savings, checking or other type
- Generic Account has no business rules for interest or overdraft
- Making Account abstract prevents creating meaningless generic accounts



Adding abstract keyword prevents instantiation while preserving inheritance benefits

- Before: we allow generic class
- After: we force accounts to be checking, saving, or any other type
- All shared code remains; subclasses inherit working implementations

```
public abstract class Account {

  public abstract boolean withdraw(double amount);

  public abstract double calculateMonthlyFee();

  public abstract String getAccountTypeName();

}
```

Multiple classes need audit capabilities *without being in same hierarchy*

- Account needs transaction history auditing
- Bank needs operation logging
  Customer needs change tracking
- Different hierarchies same capability: perfect for interface

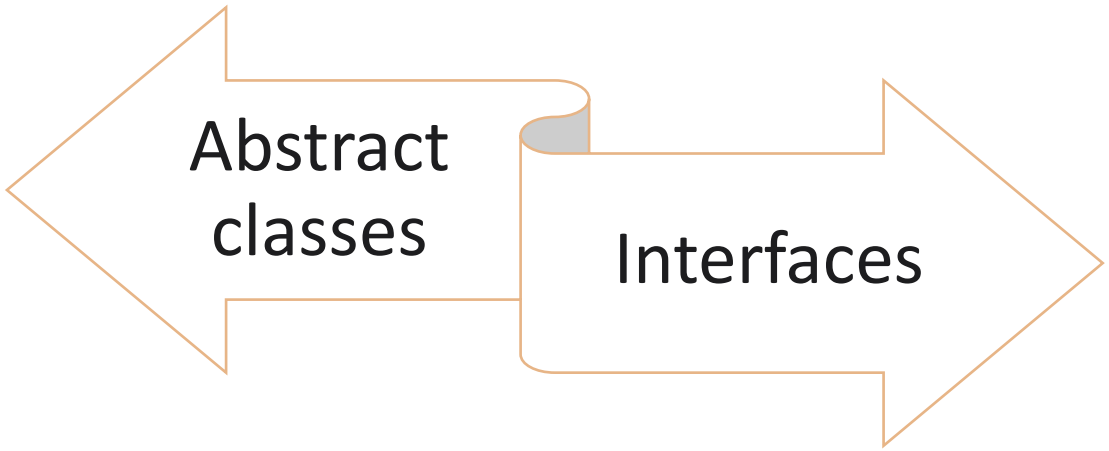
```
public interface Auditable {

  void generateAuditReport();

  List<Transaction> getAuditTrail();

  String getLastModifiedTime();

  int getTransactionCount();

}
```

Both abstractions enable powerful polymorphic code working with contracts

- Abstract class: `Account a = new SavingsAccount()` works perfectly
- Interface: `Auditable item = account; item.generateAuditReport();`
- Code depends on abstractions not concrete implementations

Abstract classes and interfaces transform good designs into bulletproof architectures

- Abstract classes: prevent incomplete instantiation, share code, and enforce customization
- Interfaces: define cross-hierarchy capabilities and support multiple implementation



UNIVERSITETI METROPOLITAN TIRANA

Stay concrete in your values

Stay flexible in your implementations