UNIVERSITETI® METROPOLITAN TIRANA

Course: Object Oriented Programming

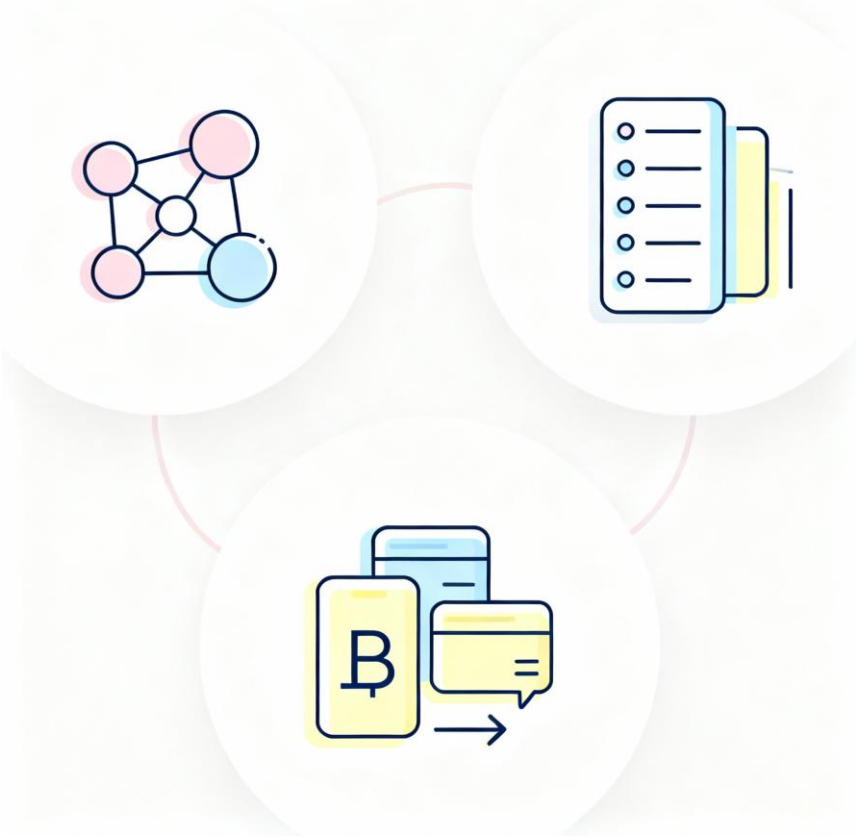
Collections and Relationships

Connecting objects with arrays, lists, and class-level design

Evis Plaku

Build solid software that grows using collections and relations

- Focus: connect objects, manage collections, and model real relationships
- Three parts: static/relationships, arrays and lists, bank application walkthrough
- Goal: move from isolated classes to collaborating objects with clean design



Static belongs to the class
instance belongs to each individual object

- Static fields are **shared by all objects**; one copy for the whole class
- Instance fields are **unique per object**; each object stores its own state

```java
public class Account {

    /** Static counter for generating unique account IDs */
    private static int accountCounter = 1000;

    /** Instance identifier for this account */
    private int accountId;
}
```

- `Account counter` is class-wide; `accountId` is per account

Use static for shared utilities and counters, not per-object state

- Static methods cannot read instance fields unless given an object reference

```
public Account() {
    this.accountId = ++accountCounter;
    this.balance   = 0.0;
    this.isFrozen  = false;
}
```

```
Account a = new Account();
// Wrong style: a.accountCounter    // static belongs to the class
int next = Account.accountCounter;  // Access via the class name
```

Real programs connect objects: who owns what, and how many

- Real systems link entities: customers own accounts; accounts contain transactions
- Three common patterns: one-to-one links, one-to-many collections of objects, and many-to-many relationships

Implement with **references** for single links, **collections** for grouped relationships

Link one object to exactly one partner using a direct reference

```
class Account {
    private int id;
    private Customer owner;          // reference to single counterpart
}

class Customer {
    private int id;
    private Account primaryAccount;  // mirrored single reference
}
```

- Each object connects to a single counterpart, forming a unique pairing
- Implement with a reference field; optionally mirror it on both classes

One object manages a group of related objects using collections

```
class Customer {
    /** List of accounts owned by this customer */
    private ArrayList<Account> accounts;
}
```

- One object links to several others; each child links back to one
- Implement with a collection field on the “one” side, like a list

Both sides have many; connect them with a dedicated linking class

- Each object relates to many others, and the reverse is also true
- Implement with a join class holding references to both linked objects
- Customers access many Services; Services are used by many Customers

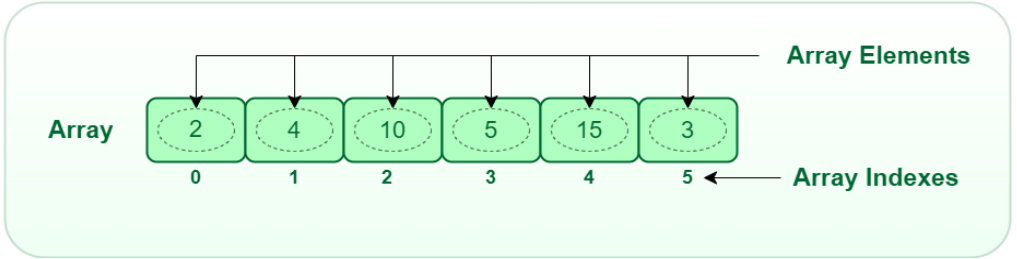
```java
class Customer {
    private int id;
    // other fields ...
}

class Service {
    private int id;
    // other fields ...
}

// Join class linking both sides (many-to-many)
class Subscription {
    private final Customer customer;
    private final Service service;
    public Subscription(Customer c, Service s) {
      this.customer = c;
      this.service = s;
    }
}
```

Collections of Objects

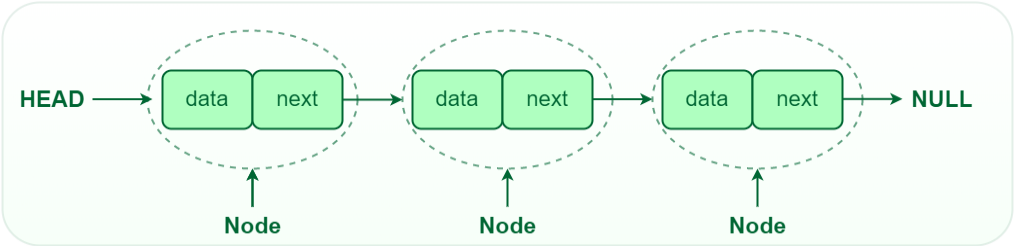
Arrays and Lists

Manage groups of objects to model real tasks and workflows

- Real applications track many objects together, not just single instances



- Collections enable add, remove, search, and filter over related objects



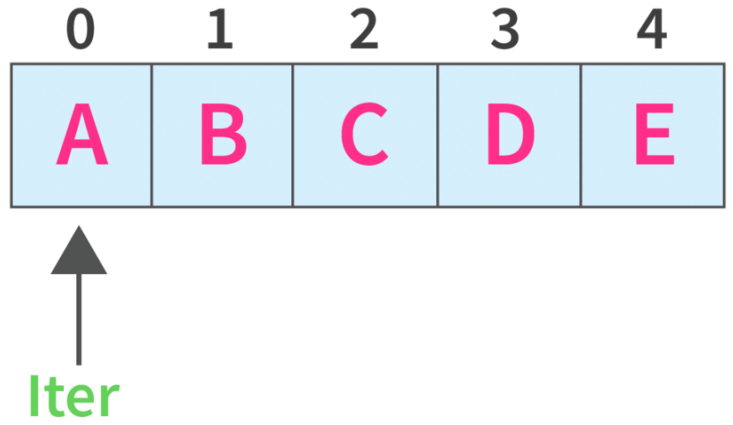
- Arrays and lists are core tools for storing and organizing objects

UNIVERSITETI METROPOLITAN TIRANA

Arrays store a fixed number of elements with fast indexed access

- Declare with type and size
- Index access is constant-time;
  size cannot change after creation
- Best when maximum capacity is known and predictable upfront

```java
// primitive types
int[] scores = new int[5];

// objects
Account[] accounts = new Account[3];
```



Initialize each slot, then iterate to use your object references

```java
// 1) Create array, then instantiate each element
Account[] accounts = new Account[3];
for (int i = 0; i < accounts.length; i++) {
    accounts[i] = new Account(); // each slot gets a new Account
}

// 2) Iterate with enhanced for-each
for (Account acc : accounts) {
    System.out.println(acc.getAccountId());
}
```

- Create the array, then instantiate each object / element in a loop
- Iterate using **classic “for to” index**, or **enhanced for-each** for clarity

Arrays are fixed-size containers without rich operations built-in

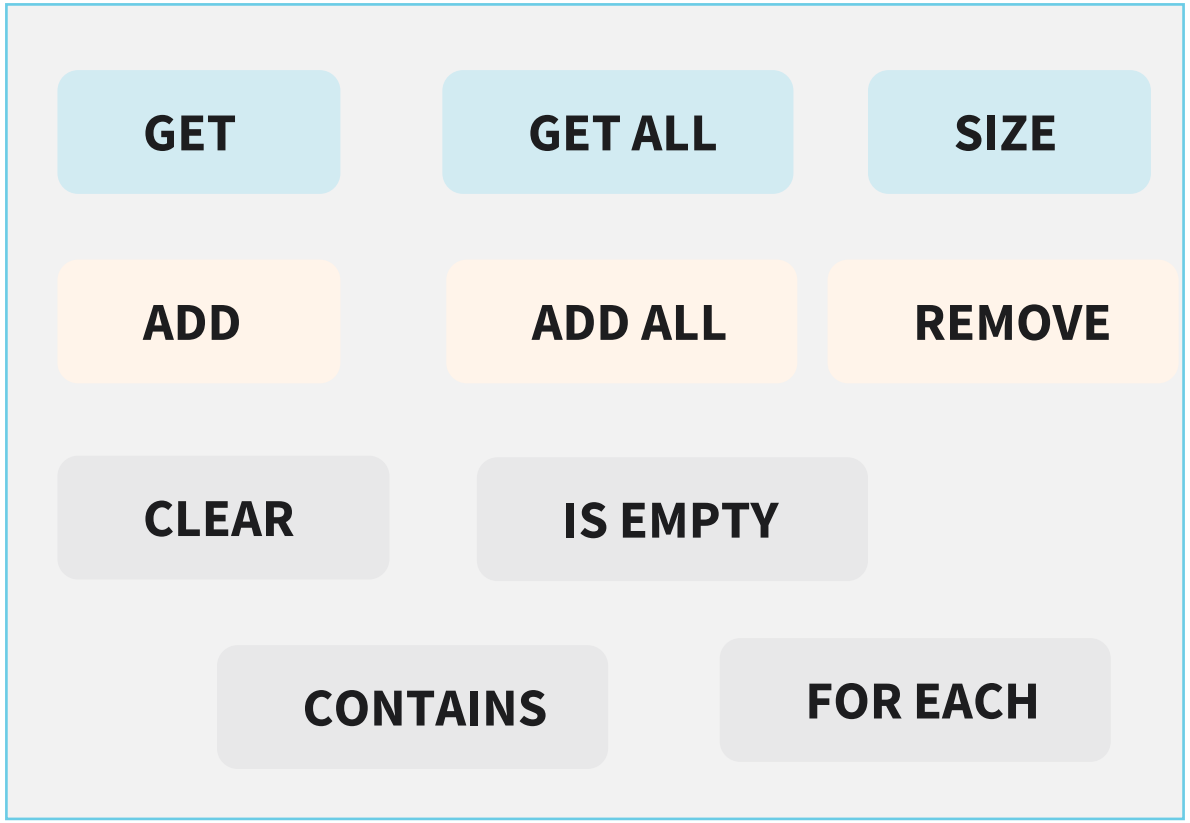
- **Fixed size**: you must know capacity in advance or waste memory space
- No built-in search/insert/delete; you write manual indexing logic yourself

```
// Need capacity upfront (fixed at 100)
// If we only use 3 slots,
// the other 97 remain unused (wasted space).
Account[] accounts = new Account[100];
accounts[0] = new Account();
accounts[1] = new Account();
accounts[2] = new Account();
```

UNIVERSITETI METROPOLITAN TIRANA

A resizable list of objects with convenient methods and indexing

- Prefer `ArrayList` for dynamic sizing and convenient helper methods when growing
- Grows and shrinks automatically as elements are added or removed



UNIVERSITETI METROPOLITAN TIRANA

Declare, add, and access elements with simple, readable list syntax

- Declare with proper import
- Add or remove elements
- Access elements or iterate through enhanced for each loops

```
import java.util.ArrayList;

ArrayList<Account> accounts = new ArrayList<>();

accounts.add(new Account());       // add
accounts.remove(0);                // remove by index
// accounts.remove(acc);           // or remove by object reference

for (Account acc : accounts) {     // enhanced for-each
    System.out.println(acc.getAccountId());
}
```

Find, iterate, and update list elements with simple, readable calls

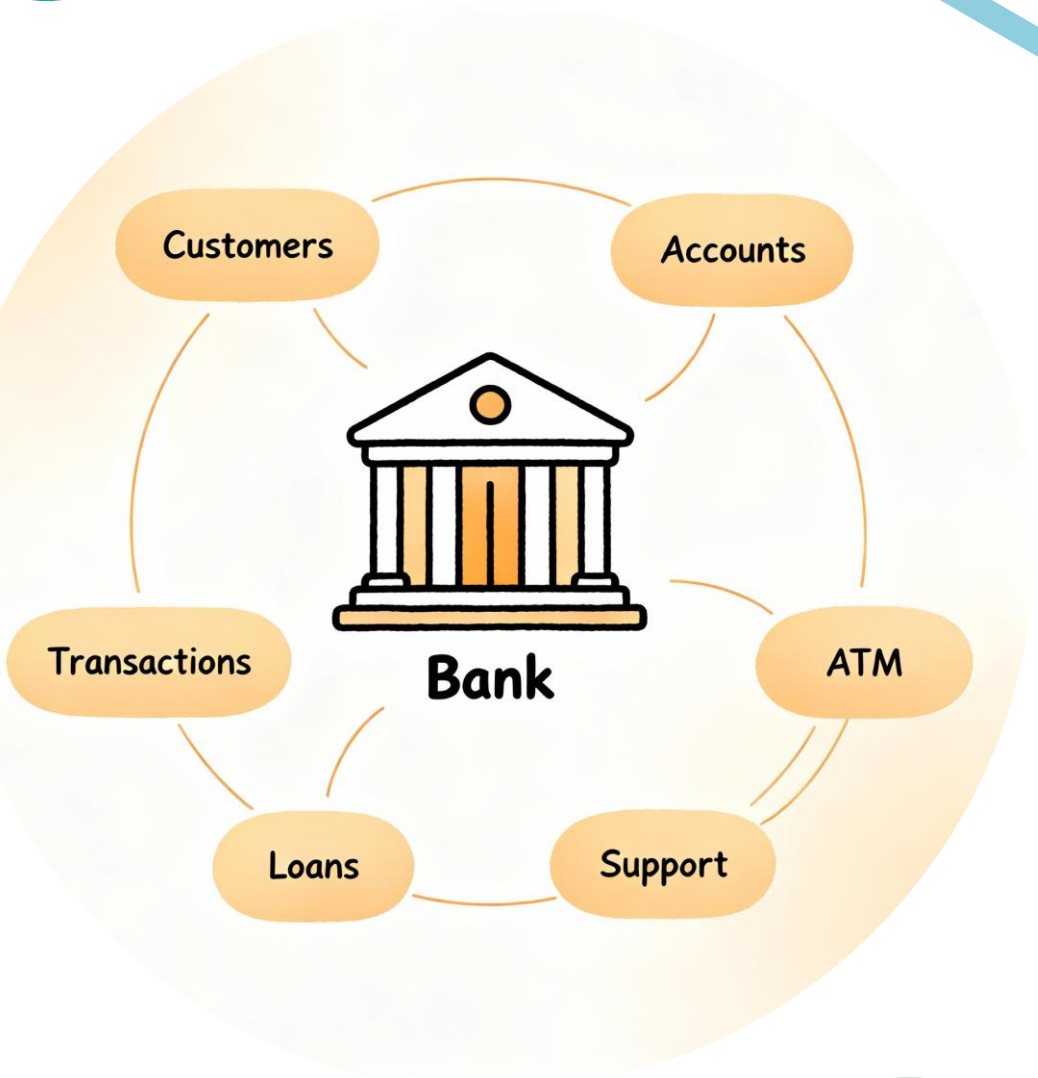
- Use `contains()`, `indexOf()`, `remove(object)`, and `clear()` for common operations

- Enhanced **for-each** to read items cleanly

```
// contains: checks by equals() identity unless overridden
boolean hasA1 = accounts.contains(a1);

// indexOf: first position of the element (or -1 if absent)
int idx = accounts.indexOf(a2);

// remove(object): removes first matching element, returns boolean
boolean removed = accounts.remove(a1);

// clear: removes all elements
accounts.clear();
```

Fixed size vs dynamic growth pick based on stability and flexibility

Aspect	Arrays	ArrayList
Size behavior	Fixed at creation; cannot change later	Dynamic; grows and shrinks automatically
Ease of use	Manual add/remove and resizing logic required	Built-in add(), remove(), clear(), and more
Access syntax	Fast indexed access with a[i]	Indexed access with get(i), still fast
Data types	Supports primitives and objects directly	Objects only; primitives via autoboxing
Typical usage	Known capacity, performance-critical segments	Managing domain objects in application code

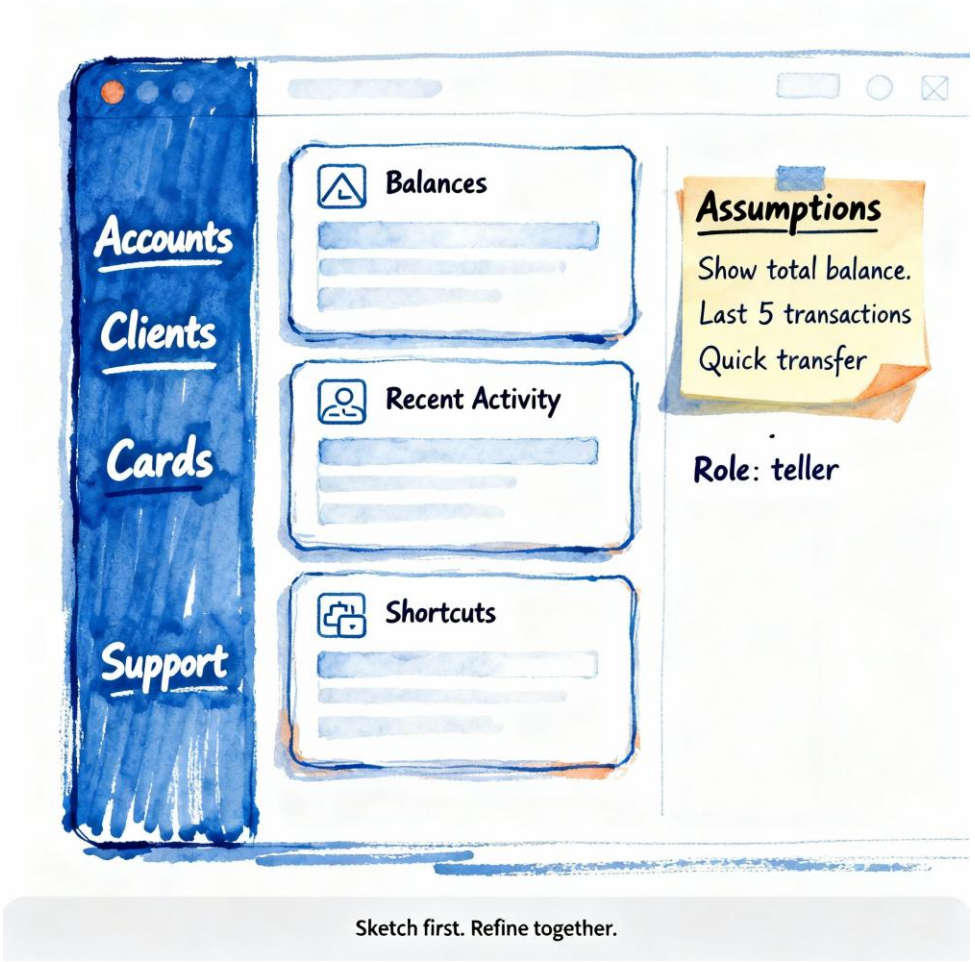


Bank System Example

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

Use a static counter to assign unique IDs to each account

- Keep a private static counter that increments on every `Account` creation
- Set `accountId` in the constructor using the incremented counter value

```
public class Account {
  private static int accountCounter = 1000;
  private int accountId;
}

public Account() {
  this.accountId = ++accountCounter;
  this.balance   = 0.0;
  this.isFrozen  = false;
}
```

A customer manages many accounts through a private list

- Store accounts in `private ArrayList<Account>` to protect internal state
- Initialize the list in the constructor to ensure it’s always usable

```
public class Customer {

  private ArrayList<Account> accounts;

  public Customer() {
        this.accounts = new ArrayList<>();
    }
}
```

UNIVERSITETI METROPOLITAN TIRANA

Provide simple, clear methods to manage a customer's accounts

- `addAccount(Account a)`: validate non-null and avoid duplicates before adding
- `removeAccount(Account a or id)`: `return boolean`; log not-found cases
- `findAccount(int id)`: iterate list, compare `getAccountId()`, return the match or null

```java
public boolean addAccount(Account account) {
  accounts.add(account);
  return true;
}

public boolean removeAccount(Account account) {
  return accounts.remove(account);
}
```

UNIVERSITETI METROPOLITAN TIRANA

Compute totals and route operations to the right account

- `getTotalBalance()`: sum `account.getBalance()` across the accounts list
- `deposit(id, amount)` and `withdraw(id, amount)`: locate account, then delegate

```java
public double getTotalBalance() {
    double total = 0.0;
    for (Account account : accounts) {
        total += account.getBalance();
    }
    return total;
}
```

Use static wisely; model relationships manage objects with collections

- Static is for class-level data like counters; avoid global mutable state overuse
- ArrayList is the default for dynamic groups of objects in application code
- Relationships + collections let objects collaborate to mirror real systems

Design cleaner systems by separating responsibilities and reducing coupling

UNIVERSITETI METROPOLITAN TIRANA

Keep good company

Count what counts. Prune your list