Software Construction
Introduction to Object Oriented Programming

Dhammika Elkaduwe

Department of Computer Engineering
Faculty of Engineering

University of Peradeniya

ILOs

- Object Oriented Programming
- Object vs class
- Constructor (what is it, how to use, rules)
- *this* keyword
- *new* keyword and creating objects
- Garbage collection
- None-static functions
- Using classes as a data-type
- objects are called by reference

Basic idea of object oriented programming

- Not only in Java
- OOP is a way of modelling/solving a problem
  - as objects and their interactions

OOP concepts

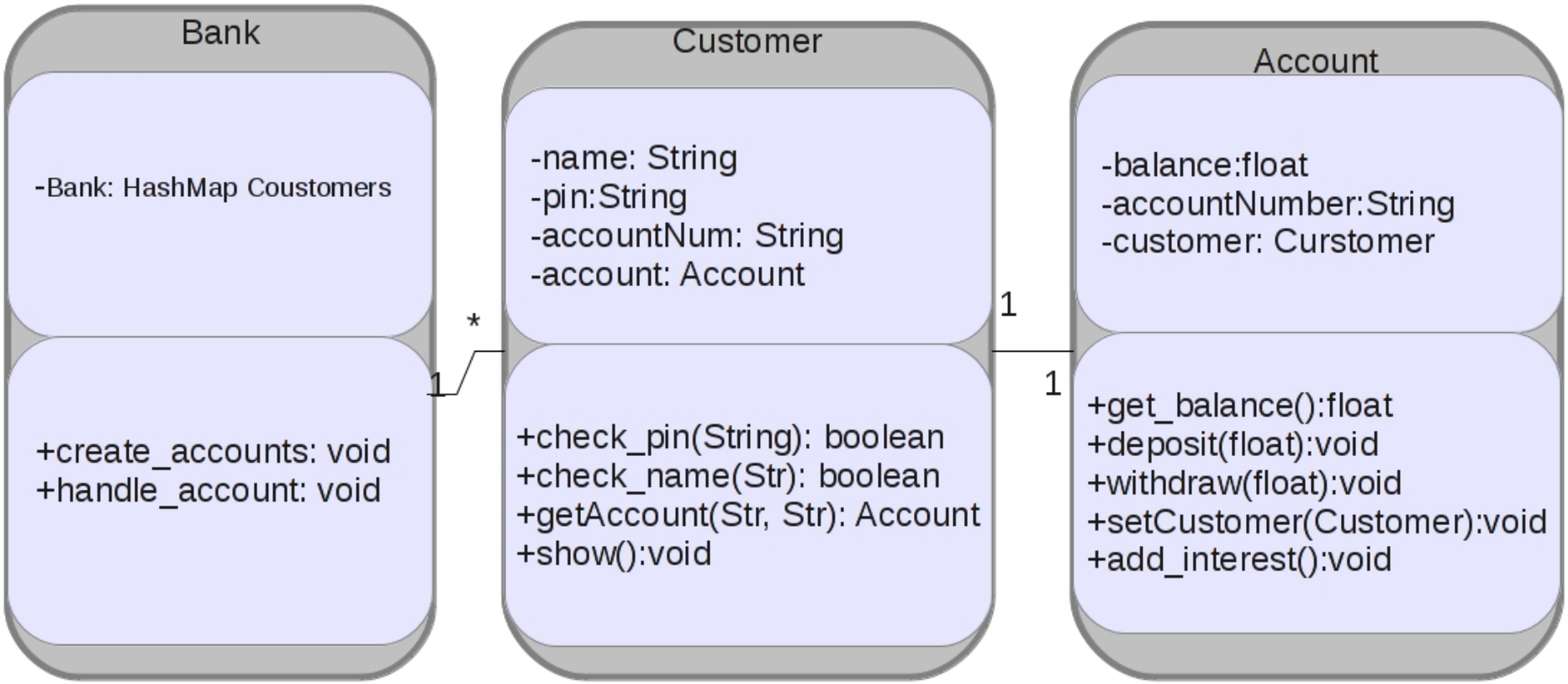
There are three main concepts:

- **Encapsulation** (we will look at this today)
- Polymorphism
- Inheritance

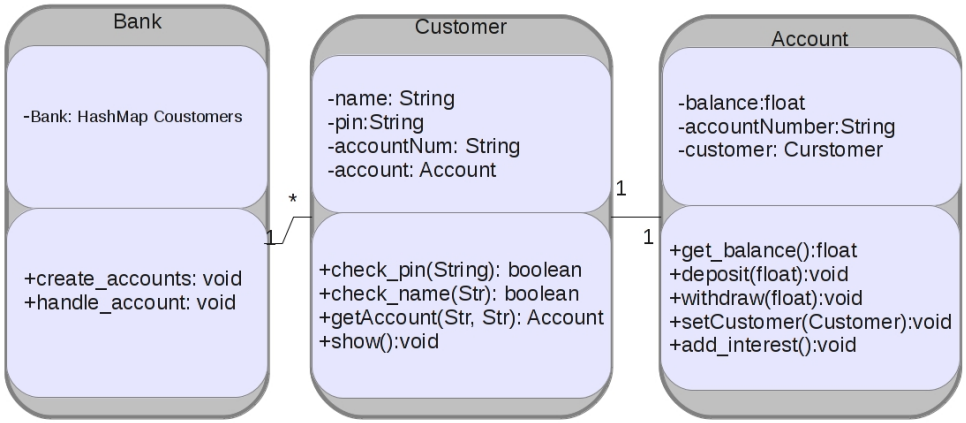
Problem: *A Bank*

- Bank has Customers;
  - *Customer* has a *name*, *pin* and *account number*,
  - Each Customer has an account associated. (later we can modify a customer to have more than one account).
- Account has;
  - *Account* has an *account number* and a *customer* associated with it,
  - Each account has a *balance*.
- The Bank is then collection of Customers (who has accounts).

Class Diagram

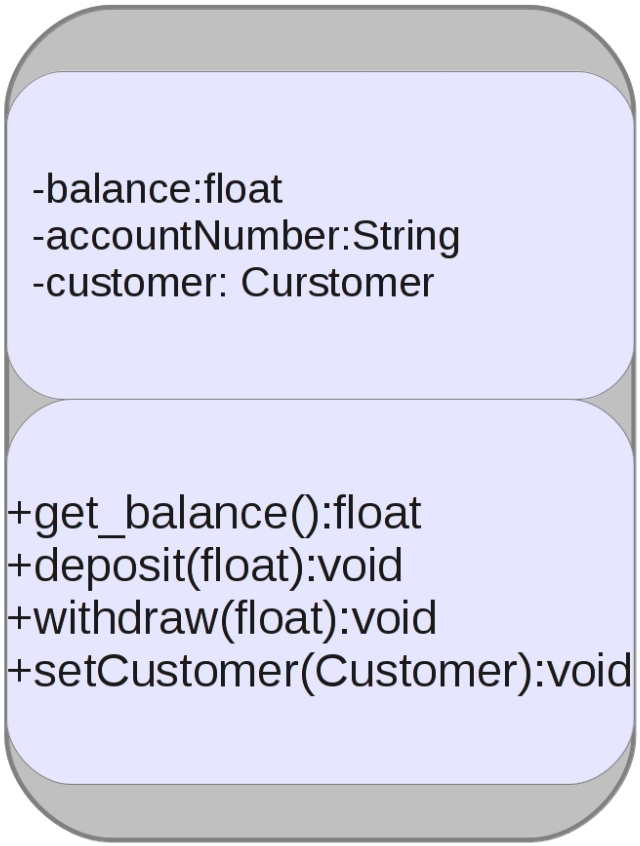


Notes



- The problem is modelled as interaction between Bank, Costumer and Account objects
- Each object has attributes and methods associated with it
- Objects are created out of classes
- Classes are created/tested separately

OOP Concepts 1: Encapsulation



- Binds attributes and methods to manipulate them into a single unit (*in Java into a class*).
- (*in C: structures to keep attributes and functions to manipulate them are separate*)

OOP Concepts 1: Encapsulation

```
public class Account {
    /* attributes of an account */
    private float balance;
    private String accountNumber;
    private Customer customer;


    /* methods that can be invoked on an account object */
    public float getBalance() {
      return this.balance;
    }
    public void deposit(float amount) {
      balance += amount;
    }
    ....
}
```

see Account.java

Private vs Public

```
public class Account {
    /* attributes of an account */
    private float balance;
    ...

    /* methods that can be invoked on an account object */
    public float getBalance() {
      return this.balance;
    }
    ....
}
```

- if a method/attribute is Public, then it can be called/accessed by all methods
- if a method/attribute is Private, then it can be called/accessed **only** by methods in the same class

Information Hiding

```
public class Account {
    /* attributes of an account */
    private float balance;
    ...

    /* methods that can be invoked on an account object */
    public float getBalance() {
      return this.balance;
    }
    ....
}
```

- **Information hiding**: hide the internals of the class from outside world (ex: balance is private so methods outside the class cannot access it).
- There is a public method (interface) to access it
- *Motivation*: can change the implementation later

The *Customer* class

- each *Account* has a *Customer*.
- an object containing attributes of a customer

```
public class Customer {

  private String name;
  private String pin;
  private String accountNum;
  private Account account;

  public boolean check_pin(String pin) {
    return this.pin.equals(pin);
  }
  public boolean check_name(String name) {
    return this.name.equals(name);
  }
```

The **constructor**

Need to create a new *Customer*.
Start with *C*:

```
// allocate memory
struct customer * c = malloc(sizeof(struct customer));
// initialise with values
c -> name = customer_name;
c -> pin = pin;
c -> accountNum = accountNumber;
c -> account = account;
```

- allocate memory for the structure
- initialise attributes

The **constructor**

- In Java this done, via a function called the **constructor**.
- Rules about the constructor:
  - should be public
  - should have the same name as the *class*
  - no return type
  - no return statement at the end of the function
  - can be overloaded

```
public class Customer {
  private String name;
  private String pin;
  private String accountNum;
  private Account account;
  // Constructor
  public Customer(String name, String pin,
  String accountNum, Account account) {
    this.name = name; this.pin = pin;
    this.accountNum = accountNum; this.account = account;
  }
```

this keyword

```
public class Customer {
  private String name;
  private String pin;
  private String accountNum;
  private Account account;
  // Constructor
  public Customer(String name, String pin,
  String accountNum, Account account) {
    this.name = name; this.pin = pin;
    this.accountNum = accountNum; this.account = account;
  }
```

- this $\Longrightarrow$ *of this object*
  - this.name *implies* the name attribute of the object
  - this.pin *implies* the pin attribute of the object

this keyword

```
public class Customer {
  private String name;
  private String pin;
  private String accountNum;
  private Account account;
  // Constructor
  public Customer(String name, String pin,
  String accountNum, Account account) {
    name = name; pin = pin;
    accountNum = accountNum; account = account;
  }
```

- What happens if we remove the *this* keyword?
  - *name* will refer to argument *name*
  - remember local variable hiding global variables
- *this* keyword can be dropped, if there is no ambiguity
- Good practice to keep it

Constructor in *Account* class

```
class Account {
   /* attributes of an account */
   private float balance;
   private String accountNumber;
   private Customer customer;
   public Account(String accountNumber) {
     this.balance = 0;
     this.accountNumber = accountNumber;
     this.customer = null;
   }
   /* methods that can be invoked on an account object */
   public float getBalance() {
     return this.balance;
   }
```

- this $\Longrightarrow$ *of this object*
- Note that *this* keyword can be used inside all the (non-static) functions

Object vs Class

```
class Account {
   /* attributes of an account */
   private float balance;
   ....


   /* methods that can be invoked on an account object */
   public float getBalance() {
     return this.balance;
   }
   ...
```

- **Class**: a blueprint to make an object (says what it should have and how it should behave)
- **Object**: is an instance of a class (specific values for (non-static) attributes and has defined behaviour)

Creating an object

steps:

- find memory for the new object (via the *new* keyword)
- call the constructor to initialise the attributes

see TestAccount.java

```
class TestAccount {
  public static void main(String [] args) {
    Account acc = new Account("057-12122-1223-12");
    ...
  }
  ...
```

- acc is an *Account* object
- it has its own attributes and methods

Objects as arguments

Points to remember:

- Objects can be passed as arguments to functions
- *Objects can be returned as results from functions*
- Objects **encapsulate** attributes and methods together

see TestAccount.java

```
class TestAccount {
  public static void main(String [] args) {
    Account acc = new Account("057-12122-1223-12");
    addMoney(acc, 1000f);
  }
  public static void addMoney(Account acc, float value) {
    acc.deposit(value); // deposit method of acc object
    acc.show(); // show method of acc object
  }
  ...
```

None-static functions

Points to remember about none-static functions:

- *static* keyword is not there in the definition
- cannot be invoked without an object

see Account.java

```
public class Account {
    /* attributes of an account */
    private float balance;
    private String accountNumber;
    private Customer customer;
    ...
    /* methods that can be invoked on an account object */
    public float getBalance() { return this.balance; }
    public void deposit(float amount) { balance += amount; }
    public void withdraw(float amount) {
      if(getBalance() > amount) balance -= amount;
      else System.out.println("This cannot happen");
    }
```

None-static and *this* keyword

meaning of *this* keyword in the code:

- `this.balance` $\Longrightarrow$ balance of the object on which the method was called
- `non-static` $\Longrightarrow$ cannot be called without an object

```
public class Account {
    /* attributes of an account */
    private float balance;
    ...

    public void deposit(float amount) {
      balance += amount;
    }
    ...
}
/* inside the TestAccount class */
public static void addMoney(Account acc, float value) {
  acc.deposit(value);
  ..
```

Classes as new data-types

see Account.java

```
public class Account {
    /* attributes of an account */
    private float balance;
    private String accountNumber;
    private Customer customer; // Account has a Customer object
    // (that would track details about the owner)
    ...
```

Somewhat complete example

see TestAccountAndCustomer.java

```
class TestAccountAndCustomer {
  public static void main(String [] args) {

    String accNumber = "057-121-3234-4431";

    Account acc = new Account(accNumber);
    // create a new customer and associate this account with
        him/her
    Customer cus = new
    Customer("Dhammika Elkaduwe", "1234", accNumber, acc);

    acc.setCustomer(cus); // set the customer of account
    acc.show();
  ...
  }
```

Passed by reference

see TestAccountAndCustomer.java

```
class TestAccountAndCustomer {

  public static void main(String [] args) {
    acc.setCustomer(cus); // set the customer of account
    acc.show();
    depositMoney(acc, 3000f); // pass by reference

    System.out.printf("\nAfter depositing..");
    acc.show();
  }

  public static void depositMoney(Account acc, float val){
    acc.deposit(val);
  }
```

Objects are passed/stored by reference

ILOs: Revisited

- Object Oriented Programming
- Object vs class
- Constructor (what is it, how to use, rules)
- *this* keyword
- *new* keyword and creating objects
- Garbage collection
- None-static functions
- Using classes as a data-type
- objects are called by reference