Java

Lesson 3

CreditCard class example

CreditCard class defines credit card objects that model a simplified version of traditional credit cards.

- They store information
  - about the customer, issuing bank, account identifier, credit limit, and current balance.
- They do not charge interest or late payments, but they do restrict charges that would cause a card's balance to go over its credit limit.

Variables

- We will define five variables, four of which are declared as private and one that is protected.
  - private String customer; // name of the customer (e.g., ”John Bowman”)
  - private String bank; // name of the bank (e.g., ”California Savings”)
  - private String account; // account identifier (e.g., ”5391 0375 9387 5309”)
  - private int limit; // credit limit (measured in dollars)
  - protected double balance; // current balance (measured in dollars)

- The **private** modifier specifies that the member can only be accessed in its own class.
- The **protected** modifier specifies that the member can only be accessed within its own package

Constructors

We will need constructors to initialize the variables

The class defines two different constructor forms.

1. The first requires five parameters, including an explicit initial balance for the account. **CreditCard(String cust, String bk, String acnt, int lim, double initialBal)**
2. The second constructor accepts only four parameters; it relies on use of the special this keyword to invoke the five-parameter version, with an explicit initial balance of zero (a reasonable default for most new accounts). **CreditCard(String cust, String bk, String acnt, int lim)**

Getters & Setters methods

The class defines five basic accessor methods

```
public String getCustomer( ) { return customer; }
public String getBank( ) { return bank; }
public String getAccount( ) { return account; }
public int getLimit( ) { return limit; }
public double getBalance( ) { return balance; }
```

Update methods:

Update methods (**charge** and **makePayment**).

The **charge** method relies on conditional logic to ensure that a charge is rejected if it would have resulted in the balance exceeding the credit limit on the card.

```
public boolean charge(double price){
    if (price + balance > limit)          // if charge would surpass limit
        return false;  // refuse the charge
    // at this point, the charge is successful
    balance += price;                     // update the balance
    return true;                          // announce the good news
    }
public void makePayment(double amount) { // make a payment
        balance −= amount;
}
```

Static Method

- We provide a static utility method, named printSummary

```
public static void printSummary(CreditCard card) {
    System.out.println("Customer = " + card.customer);
    System.out.println("Bank = " + card.bank);
    System.out.println("Account = " + card.account);
    System.out.println("Balance = " + card.balance); // implicit cast
    System.out.println("Limit = " + card.limit); // implicit cast
}
```

Main method

- The main method includes an array, named wallet, storing CreditCard instances.
- The main method also demonstrates a while loop, a traditional for loop, and a for-each loop over the contents of the wallet.
- The main method demonstrates the syntax for calling traditional (nonstatic) methods—charge, getBalance, and makePayment—as well as the syntax for invoking the static printSummary method.

```
public static void main(String[ ] args) {
    CreditCard[ ] wallet = new CreditCard[3];
    wallet[0] = new CreditCard("John Bowman", "California Savings", "5391 0375 9387 5309",
    5000);
     wallet[1] = new CreditCard("John Bowman", "California Federal", "3485 0399 3395 1954",
    3500);
     wallet[2] = new CreditCard("John Bowman", "California Finance", "5391 0375 9387 5309",
    2500, 300);
```

Make some charges

```
for (int val = 1; val <= 16; val++) {
        wallet[0].charge(3*val);
        wallet[1].charge(2*val);
       wallet[2].charge(val);
 }
```

The Java language takes a general and useful approach to the organization of classes into programs. Every stand-alone public class defined in Java must be given in a separate file. The file name is the name of the class with a .java extension. So a class declared as public class Window is defined in a file Window.java. That file may contain definitions for other stand-alone classes, but none of them may be

declared with public visibility. To aid in the organization of large code repository, Java allows a group of related type definitions (such as classes and enums) to be grouped into what is known as a package. For types to belong to a package named packageName, their source code must all be located in a directory named packageName and each file mustbegin with the line:

package packageName;

By convention, most package names are lowercased. For example, we might define an architecture package that defines classes such as Window, Door, and Room. Public definitions within a file that does not have an explicit package declaration are placed into what is known as the default package.

```
for (CreditCard card : wallet) {
        CreditCard.printSummary(card);     // calling static method
        while (card.getBalance( ) > 200.0) {
                card.makePayment(200);
                System.out.println("New balance = " + card.getBalance( ));
         }
}
```

Object-oriented

- Software implementations should achieve ***robustness***, ***adaptability***, and ***reusability***.
  1. For example,if a program is expecting a positive integer (perhaps representing the price of an item) and instead is given a negative integer, then the program should be able to recover gracefully from this error
  2. Related to this concept is ***portability***, which is the ability of software to run with minimal change on different hardware and operating system platforms. An advantage of writing software in Java is the portability provided by the language itself.
  3. Going hand in hand with adaptability is the desire that software be reusable, that is, the same code should be usable as a component of different systems in various applications.

- Abstraction
- Encapsulation
- Modularity

Inheritance

A natural way to organize various structural components of a software package is in a ***hierarchical*** fashion, with similar abstract definitions grouped together in a level-by-level manner that goes from specific to more general as one traverses up the hierarchy. An example of such a hierarchy is shown in Figure 2.3. Using mathematical notations, the set of houses is a ***subset*** of the set of buildings, but a ***superset*** of the set of ranches. The correspondence between levels is often referred to as an ***"is a" relationship***, as a house is a building, and a ranch is a house.

- Java Inheritance example
- Δημιουργήστε ένα project με όνομα MyAnimal.
- Δημιουργήστε μια κλάση Animal με
- Μεταβλητές:
- □ private boolean vegetarian,
- □ private String eats;
- □ private int noOfLegs;
- Μέθοδοι: constructors, οι getters & setters.
- Δημιουργήστε μια κλάση Cat που κληρονομεί την Animal
- Μεταβλητές:
- private String color;
- Μέθοδοι: constructors, οι getters & setters.
- Cat kids() {
- //create a kid that has the same attributes but is not a vegetarian
- Δημιουργήστε μια κλάση AnimalInheritanceTest που θα περιέχει τη main και δημιουργεί
- ένα πίνακα τύπου Cat 10 θέσεων. Αρχικοποιήστε τον πίνακα δίνοντας τα στοιχεία μέσα από
- το πληκτρολόγιο. Στη συνέχεια εκτυπώστε τον πίνακα που φτιάξατε.
- Java