Designing Classes – part 2

CSC 1051 – Data Structures and Algorithms I
Dr. Mary-Angela Papalaskari
Department of Computing Sciences
Villanova University

Course website:
www.csc.villanova.edu/~map/1051/

Next:
• Review what we learned so far
• Focus on method definition
• Encapsulation

Getting to know classes so far
• Predefined classes from the Java API.
• Defining classes of our own:

  Driver classes:
  – Account
  – Transactions
  – Die
  – RollingDice
  – Shoe
  – YouVeGotShoes (Project 5)
  – Person
  – PeopleBeingPeople (Lab 8)

Review

• class declaration

Data declarations
long acctNumber;
double balance;
String name;

Constructor
deposit()
withdraw()
toString()

Method declarations
Review

**driver classes**

```java
public class Transactions {
    public static void main(String[] args) {
        Account acct1 = new Account("Ted Murphy", 72354, 102.56);
        Account acct2 = new Account("Jane Smith", 69713, 40.00);
        System.out.println(acct1);
        System.out.println(acct2);
        acct1.deposit(25.85);
        System.out.println();
        System.out.println(acct1);
        System.out.println(acct2);
    }
}
```

**Account class**

```java
public class Account {
    String name;
    long acctNumber;
    double balance;

    public Account(String owner, long account, double initial) {
        acctNumber = account;
        balance = initial;
        name = owner;
    }

    public void withdraw(double amount, double fee) {
        balance = balance - amount - fee;
    }

    public String toString() {
        NumberFormat fmt = NumberFormat.getCurrencyInstance();
        return (acctNumber + "  " + name + "  " + fmt.format(balance));
    }
}
```

**toString() method**

```java
System.out.println(acct1);
```

**Object creation: Constructors**

```java
public Account(String owner, long account, double initial) {
    acctNumber = account;
    balance = initial;
    name = owner;
}
```
**Method definition**

- parameters
- return type
- return statement

```java
char calc (int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt (sum);
    return result;
}
```

```java
ch = obj.calc (25, count, "Hello");
```

**Method Control Flow**

- If the called method is in the same class, only the method name is needed

**More Method Examples:**

- Write a method that has one int parameter num, and prints “Happy Birthday” num times
More Method Examples:

• Write a method with two double parameters a and b that computes and returns the sum of squares: \(a^2 + b^2\) of its two int parameters

More Method Examples:

• Write a method that has one int parameter num, and returns the String “Happy Birthday” num times

Encapsulation

• An encapsulated object can be thought of as a black box -- its inner workings are hidden from the client
• The client invokes the interface methods and they manage the instance data

Violating Encapsulation experiment

• Revisit your solution for the Account Class Exercise
• Add some code to the OnePercent.java class to modify the value of an instance variable, eg:
  ```java
  acct1.name = "Joe";
  ```
• Now modify Account.java -- insert the word private in front of that variable declaration:
  ```java
  private String name;
  ```
• Re-compile the Account class and run your program again. Note the error you get.
Violating Encapsulation

- It is possible for a class to access the instance data of another class directly.

Visibility Modifiers

- In Java, we enforce encapsulation through the appropriate use of visibility modifiers:
  - `public` – can be referenced from other classes
  - `private` – can be referenced only within that class:
  - `protected` – involves inheritance (discussed later)

- Data declared without a visibility modifier have default visibility and can be referenced by any class in the same package

- An overview of all Java modifiers is presented in Appendix E

Violating Encapsulation - Example

- It is possible for a class to access the instance data of another class directly – but it's not a good idea!

- See `Account.java` (modified)
- See `ImInUrClassMessingUrInstanceData.java`

- `acct1.name = "Joe";`

public constants are ok - Example

Add some code to `OnePercent.java` to print out the interest rate used for the accounts:

```
System.out.println("Interest rate = " + acct1.RATE);
```
public constants are ok - Example

Add some code to OnePercent.java to print out the interest rate used for the accounts:

```java
System.out.println("Interest rate = " + acct1.RATE);
```

Normally, constants are declared as static.
- If RATE had been declared as follows:
  ```java
  final static double RATE = 0.03;
  ```
- Then the last statement above would have been:
  ```java
  System.out.println("Interest rate = " + Account.RATE);
  ```

Visibility Modifiers – the **RULES**

<table>
<thead>
<tr>
<th>Visibility Modifier</th>
<th>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>NO</td>
<td>Yes</td>
</tr>
<tr>
<td>(but OK for public constants)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td>Yes</td>
<td>Yes, for support methods only</td>
</tr>
</tbody>
</table>

Encapsulation – Accessing the data

- Indirect access through methods
- accessors and mutators (“getters” and “setters”)
- Usually named getX() and setX()

Encapsulation – Mutators (setters) can restrict access to the data, as appropriate

- Example: Say a class has an instance variable:
  ```java
  private int quantity;
  ```
- The mutator may also work to ensure that `quantity` does not become negative:
  ```java
  public void setQuantity(int num)
  {
      if (num<0)
      {
          System.out.println("*Error in setQuantity()";
          System.out.println("negative quantity.");
          System.out.println("quantity not changed.");
      }
      else
          quantity = num;
  }
  ```