Designing Classes

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/

Example: **Account** datatype

- represents a generic bank account

- acct1
  - acctNumber: 72354
  - balance: 102.56
  - name: "Ted Murphy"

- acct2
  - acctNumber: 69713
  - balance: 40.00
  - name: "Jane Smith"

Where do objects come from?

**Good question!**

**We will learn how to create a class that defines a new datatype, i.e., a new type of objects**

**We need to learn:**

1. What is the structure of a class definition?
2. How to specify what happens when an object is instantiated (i.e., when the `new` operator is used)?
3. How do we define the methods that can be invoked through objects of this class?

1. **Structure of class definition**

   **Account class**
   - int acctNumber;
   - double balance;
   - String name;
   - Constructor
   - deposit()
   - withdraw()
   - getBalance()
   - toString()

   **Account object**
   - acctNumber: 72354
   - balance: 102.56
   - name: "Ted Murphy"

   - The object:
     - is like the house built from the blueprint
     - is an instance of the class
     - has its own data space & shares methods defined for this datatype

   Classes define DATA and METHODS i.e., a datatype
2. Object instantiation

Creating Objects – OLD example:

• We have already seen something like this:

```java
Scanner scan = new Scanner(System.in);
```

This invokes the `Scanner constructor`, which is a special method that sets up the object.

Creating Objects – our newly defined `Account` class:

```java
Account acct1 = new Account("Ted Murphy", 72354, 102.56);
```

Invokes the `Account constructor`, which is a special method that sets up the object.

A new `Account` object is created!

3. Method invocation

• As we have seen, once an object has been created, we can use the `dot operator` to invoke its methods:

```java
ans = scan.nextLine();
numChars = title.length();
amount = acct1.getBalance();
acct1.deposit(25.85);
```

Datatype / Client (also referred to as "servant / driver" classes)
```java
public class Account {
    String name;
    int acctNumber;
    double balance;

    public Account (String x, int y, double z) {
        name = x;
        acctNumber = y;
        balance = z;
    }

    public void deposit (double x) {
        balance = balance + x;
    }

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

    public String toString () {
        return acctNumber + "\t" + name + "\t" + getBalance() + "\t" + toString();
    }

    public double getBalance () {
        return balance;
    }
}
```

```java
public class Transactions {
    public static void main (String[] args) {
        Account acct1 = new Account("Edward Demsey", 69713, 40.00);
        Account acct2 = new Account("Jane Smith", 69713, 40.00);
        Account acct3 = new Account("Edward Demsey", 93757, 759.32);
        System.out.println(acc1);
        System.out.println(acc2);
        System.out.println(acc3);
        acct1.deposit (25.85);
        acct1.withdraw (60, 2.50);
        System.out.println(acc1.toString());
        System.out.println(acc2.toString());
        System.out.println(acc3.toString());
    }
}
```
Account acct1 = new Account ("Ted Murphy", 72354, 102.56);

Account acct2 = new Account ("Jane Smith", 69713, 40.00);

acct1.deposit (25.85);

acct1 = new Account ("Ted Murphy", 72354, 102.56);

acct2 = new Account ("Jane Smith", 69713, 40.00);

acct1.deposit (25.85);
Account class: Another Example

acct1.withdraw (60, 2.50);

acct1
acctNumber 72354
balance 128.41
name "Ted Murphy"

acct2
acctNumber 72354
balance 102.56
name "Ted Murphy"

getBalance () method
double amount = acct1.getBalance();

acct1
acctNumber 72354
balance 128.41
name "Ted Murphy"

toString() method
System.out.println(acct1.toString());

acct1
acctNumber 72354
balance 102.56
name "Ted Murphy"
Bank Account Example

- There are some improvements that can be made to the Account class.
- The design of some methods could also be more robust, such as verifying that the amount parameter to the withdraw() method is positive.
- Some of these improvements are in the book examples:
  - Account.java, Transactions.java (simplified versions)
  - Account.java, Transactions.java (book versions)
NEXT: Focus on **Methods**

- Method control flow
- Method definition
  - Parameters
  - `return` statement
- Different ways of thinking about methods

**Method Control Flow**

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

**Invoking methods within the same class**

- An object’s method may access any of the object’s other methods directly. Eg:

  ```java
  public void addInterest(double rate) {
    deposit(rate * balance);
  }
  ```

  **Client code, eg:** compound the interest for `acct2` over 10 years

  ```java
  int year = 1;
  while (year <= 10) {
    acct2.addInterest(0.03 * balance);
    year ++;
  }
  ```
Method Control Flow: example

```java
acct2.addInterest(0.03);
acct2.deposit();
```

Method definition: Example
- 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;
}
```

More Method Examples:
- Write a method with two `double` parameters `a` and `b` that computes and returns the sum of squares of its two parameters (i.e., \(a^2 + b^2\)).

How do we invoke the method to compute & print: \((14.8)^2 + (37.65)^2\)?

More Method Examples:
- Write a method with one `int` parameter `num`, that returns a String composed of “Happy Birthday” `num` times

- How do we invoke the method to print “happy birthday” 4 times?
Getting to know classes so far

- Using predefined classes from the Java API.
- Defining classes for our own datatypes.

**datatypes:**
- Account
- Die
- Shoe
- Person

**Clients (Driver classes):**
- Transactions, OnePercent
- RollingDice
- YouVeGotShoes (Project)
- PeopleBeingPeople (Lab)

Next: UML class diagrams, visibility modifiers, graphical objects,

UML Class Diagrams

UML = Unified Modelling Language

- Example: A UML class diagram for the RollingDice program:

```
main (args : String[]): void

Die

faceValue : int
roll(): int
getFaceValue(value : Int): void
getToString(): Int
```

Examples of datatypes (Classes)

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>Name, Address, Major, Grade point average</td>
<td>Set address, Set major, Compute grade point average</td>
</tr>
<tr>
<td>Rectangles</td>
<td>length, Width, Color</td>
<td>Set length, Set width, Set color</td>
</tr>
<tr>
<td>Aquarium</td>
<td>Material, Length, Width, Height</td>
<td>Set material, Set length, Set width, Compute volume, Compute total weight</td>
</tr>
<tr>
<td>Flight</td>
<td>Airline, Right number, Origin city, Destination city, Current status</td>
<td>Set airline, Set flight number, Determine status</td>
</tr>
<tr>
<td>Employee</td>
<td>Name, Department, Title, Salary</td>
<td>Set department, Set title, Compute wages, Compute bonus, Compute taxes</td>
</tr>
</tbody>
</table>
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 which in turn manage the instance data.

Violating Encapsulation - **WRONG**

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

Use Accessors & Mutators - **RIGHT**

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

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 experiment

- Revisit the Account example (use our simplified versions from the course website or Lab 8)
- Add some code to the client (Transactions or Onepercent) to modify the value of an instance variable, eg:
  
  ```java
  acct1.name = "Bernie";
  ```

- Run the program to verify that the name on that account has changed.
- Now modify Account.java – insert the modifier `private` in front of the instance variable declaration:
  
  ```java
  private String name;
  ```

- Without changing anything in the client, re-compile the Account class and run your program again. Note the error you get:

  Error: ____________________________________________________

Public Constants… OK

Example: The Account class can have a constant for the interest rate:

```java
public final double RATE = 0.015;
```

A client (eg, OnePercent.java) can access this constant directly:

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

**static** Public Constants… BETTER

It is better to declare constants as **static**

```java
public final static double RATE = 0.015;
```

This way, a client can access the constants without creating an object, using the class name:

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

Visibility Modifiers – the **RULES**

<table>
<thead>
<tr>
<th>Variables</th>
<th>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>(but OK for public constants)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td></td>
<td>Yes, for support methods only</td>
</tr>
</tbody>
</table>

See also [ImInUrClassMessingUrInstanceData.java](#)