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/

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?

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"
1. Structure of class definition

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

- 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

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:
- A new **Account** object is created!

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

Invokes the **Account constructor**, which is a special method that sets up the object
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();
```

Chapter 4: Writing Classes

- We’ve been using predefined classes from the Java API. Now we will learn to write our own classes.
  - class definitions
  - constructors
  - instance data
  - method declaration and parameter passing
  - encapsulation and Java modifiers
  - more about creating graphical objects (next week)
```java
public class Account
{
    int acctNumber;
    double balance;
    String name;
    // Sets up the account by defining its owner's name, account number, and initial balance.
    public Account (String x, int y, double z)
    {
        name = x;
        acctNumber = y;
        balance = z;
    }
    // Deposits the specified amount x into the account.
    public void deposit (double x)
    {
        balance = balance + x;
    }
    // Withdraws the specified amount from the account and applies the fee.
    public void withdraw (double x, double fee)
    {
        balance = balance - x - fee;
    }
    // Returns the current balance of the account.
    public double getBalance ()
    {
        return balance;
    }
    // Returns a one-line description of the account as a string.
    public String toString ()
    {
        return acctNumber + "\t" + name + "\t" + fmt.format(balance);
    }
    // Demonstrates the creation and use of multiple Account objects.
    public static void main (String[] args)
    {
        Account acct1 = new Account ("Edward Demsey", 93757, 759.32);
        Account acct2 = new Account ("Jane Smith", 69713, 40.00);
        Account acct3 = new Account ("Ted Murphy", 72354, 102.56);
        System.out.println (acct1);
        System.out.println (acct2);
        System.out.println (acct3);
        acct1.deposit (25.85);
        acct1.withdraw (60, 2.50);
        System.out.println ();
        System.out.println (acct1);
        System.out.println (acct2);
        System.out.println (acct3);
    }
}
```
Transactions class:
Creating Account objects

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

Transactions class:
Creating more Account objects

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

Account acct2 = new Account ("Jane Smith", 69713, 40.00);
Account class: Using methods

acct1.deposit (25.85);

acct1
acctNumber 72354
balance 102.56
name "Ted Murphy"

ACCOUNT CLASS: USING METHODS

acct1.deposit (25.85);

acct1
acctNumber 72354
balance 102.56
name "Ted Murphy"

ACCOUNT CLASS: USING METHODS

acct1.deposit (25.85);

acct1
acctNumber 72354
balance 102.56
name "Ted Murphy"

ACCOUNT CLASS: USING METHODS

acct1.deposit (25.85);

acct1
acctNumber 72354
balance 102.56
name "Ted Murphy"
Designing Classes

Account class: Using methods

```java
acct1.deposit (25.85);
```

```java
ACCOUNT CLASS: Using methods
// Deposits the specified amount into the account.
public void deposit (double x) {
    balance = balance + x;
}
```

Account class: Another Example

```java
acct1.withdraw (60, 2.50);
```

```java
ACCOUNT CLASS: Another Example
// Withdraws the specified amount from the account
// and applies the fee.
public void withdraw (double x, double fee) {
    balance = balance - x - fee;
}
```

```
```
Class definitions

- A class can contain data declarations and method declarations

**Data declarations** (also called *fields*)

```java
int acctNumber;
double balance;
String name;
```

**Constructor**

```java
deposit()
withdraw()
getBalance()
toString()
```

**Method declarations** (note: the constructor is also a method)

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

**toString() method**

```java
acct1 = new Account(72354, 102.56, "Ted Murphy");
System.out.println(acct1.toString());
```

**getBalance() method**

```java
double amount = acct1.getBalance();
// Note: this code is not used in Transactions.java
```
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)

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, Height</td>
<td>Set material, Set length, Set height, Compute volume, Compute weight</td>
</tr>
<tr>
<td>Flight</td>
<td>Airline, Flight number, Origin city, Destination city, Current status</td>
<td>Set price, Set flight number, Determine status</td>
</tr>
<tr>
<td>Employee</td>
<td>Name, Title, Salary</td>
<td>Set department, Set title, Compute wages, Compute bonus, Compute taxes</td>
</tr>
</tbody>
</table>

CSC 1051 M.A. Papalaskari, Villanova University

---

```
//********************************************************************
// RollingDice.java Author: Lewis/Loftus
// Demonstrates the creation and use of a user-defined class.
//********************************************************************
public class RollingDice {
    public static void main (String[] args) {
        Die die1, die2;
        int sum;
        die1 = new Die();
        die2 = new Die();
        die1.roll();
        die2.roll();
        System.out.println("Die One: " + die1 +", Die Two: " + die2);
        die1.roll();
        die2.setFaceValue(4);
        System.out.println("Die One: " + die1 +", Die Two: " + die2);
        sum = die1.getFaceValue() + die2.getFaceValue();
        System.out.println("Sum: " + sum);
        sum = die1.roll() + die2.roll();
        System.out.println("New sum: " + sum);
    }
}
```

---

```
//********************************************************************
// Die.java Author: Lewis/Loftus
// Represents one die (singular of dice) with faces showing values
// between 1 and 6.
//********************************************************************
public class Die {
    private final int MAX = 6; // maximum face value
    private int faceValue; // current value showing on the die
    // Constructor: Sets the initial face value.
    public Die() {
        faceValue = 1;
    }
    // Rolls the die and returns the result.
    public int roll() {
        faceValue = (int)(Math.random() * MAX) + 1;
        return faceValue;
    }
}
```
continue

// Face value mutator.
public void setFaceValue(int value)
{
    faceValue = value;
}

// Face value accessor.
public int getFaceValue()
{
    return faceValue;
}

// Returns a string representation of this die.
public String toString()
{
    String result = Integer.toString(faceValue);
    return result;
}

UML Class Diagrams

UML = Unified Modelling Language
• Example: A UML class diagram for the RollingDice program:

UML class diagram for Transactions program?

Next: Focus on Method definition
– parameters
– return type
– return statement

char ch = obj.calc(start, 2, "ABCDE");

char calc (int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt(sum);
    return result;
}
## 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 its other methods directly. Eg:

  ```java
def 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);
    year ++;
  }
  ```

- Thus the dot operator is an addressing mechanism. Note that it can also be used to access an object’s or class’s data directly, for example:
  - acct1.name
  - Color.black
- more on this later (encapsulation)
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: visibility modifiers, graphical objects,

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

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

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!*

```
acctl.name = "Joe";
```

```
Account.java

private String name;
```

**Account.java**

```
acct1.name = "Joe";
```

```
transactions.java

int x1 = acct1.getBalance();
```

**Transactions.java**

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 & Transactions example from last week (be sure to use our simplified versions from the course website)
- Add some code to the Transactions.java class to modify the value of an instance variable, eg:
  ```
  acct1.name = "Joe";
  ```
  (This should work as expected)
- Now modify Account.java – insert the modifier **private** in front of that variable declaration:
  ```
  private String name;
  ```
- 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.print("Interest rate = " + acct1.RATE);
```

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.print("Interest rate = " + Account.RATE);
```

Visibility Modifiers – the RULES

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

See also [ImInUrClassMessingUrInstanceData.java](#)