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"
```

CSC 1051 M.A. Papalaskari, Villanova University
1. Structure of class definition

**Account class**

**Data**
- 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

• The class is the *blueprint*
  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.
2. Object instantiation

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!
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)

public class Transactions {
   // Demonstrates the creation and use of multiple Account objects.
   public static void main(String[] args) {
      Account acct1 = new Account("Ted Murphy", 72354, 102.56);
      Account acct2 = new Account("Jane Smith", 69713, 40.00);
      Account acct3 = new Account("Edward Demsey", 93757, 759.32);
      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);
   }
}

Datatype / Client

import java.text.NumberFormat;
public class Account {
   int acctNumber;
   double balance;
   String name;

   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() {
      // Implementation of withdraw method
   }

   public String toString() {
      return "Account: " + name + " Account Number: " + acctNumber + " Balance: " + balance;
   }

   public int getBalance() {
      return (int) balance;
   }
}

CSC 1051 M.A. Papalaskari, Villanova University
import java.text.NumberFormat;

public class Account {
    int acctNumber;
    double balance;
    String name;

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

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

    Constructor:
    deposit()
    withdraw()
    getBalance()
    toString()

    Instance Variables:
    int acctNumber;
    double balance;
    String name;
}

// Sets up the account by defining its owner's name, account number, and initial balance.
// Deposits the specified amount x into the account.
continue

// 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 ()
{
    NumberFormat fmt = NumberFormat.getCurrencyInstance();
    return (acctNumber + "\t" + name + "\t" + fmt.format(balance));
}
public class Transactions {
    // --------------------------------------------------------------------
    //  Creates some bank accounts and requests various services.
    // --------------------------------------------------------------------
    public static void main (String[] args) {
        Account acct1 = new Account("Ted Murphy", 72354, 102.56);
        Account acct2 = new Account("Jane Smith", 69713, 40.00);
        Account acct3 = new Account("Edward Demsey", 93757, 759.32);

        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.toString());
        System.out.println(acct2.toString());
        System.out.println(acct3.toString());
    }
}
/**
 * Transactions.java   Author: MA Papalaskari
 * (based on Lewis/Loftus example)
 * Demonstrates the creation and use of multiple Account objects.
 **/

public class Transactions
{
    //------------------------------------------------------------------
    // Creates some bank accounts and requests various services.
    //------------------------------------------------------------------
    public static void main (String[] args)
    {
        Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
        Account acct2 = new Account ("Jane Smith", 69713, 40.00);
        Account acct3 = new Account ("Edward Demsey", 93757, 759.32);

        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.toString());
        System.out.println (acct2.toString());
        System.out.println (acct3.toString());
    }
}
Transactions class:

Creating Account objects

Transactions class

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

Account class

Constructor method

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

cacct1

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

Constructor

deposit() withdraw() getBalance() toString()
Transactions class:

Creating more Account objects

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

Account acct2 = new Account ("Jane Smith", 69713, 40.00);
acct1.deposit (25.85);

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

CSC 1051 M.A. Papalaskari, Villanova University
acct1.deposit (25.85);

```java
// Deposits the specified amount into the account.
public void deposit (double x) {
    balance = balance + x;
}
```
Account class: Another Example

acct1.withdraw (60, 2.50);

//------------------------------------------------
// Withdraws the specified amount from the
// account and applies the fee.
//------------------------------------------------
public void withdraw (double x, double fee)
{
    balance = balance - x - fee;
}
acct1.withdraw (60, 2.50);

```java
public void withdraw (double x, double fee) {
    balance = balance - x - fee;
}
```

 acctNumber 72354
 balance 65.91
 name "Ted Murphy"

ACCOUNT CLASS: Another Example

CSC 1051 M.A. Papalaskari, Villanova University
getBalance () method

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

public double getBalance ()
{
    return balance;
}

acct1

acctNumber  72354
balance  102.56
name  "Ted Murphy"
toString() method

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

System.out.println(acct1.toString());
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)
public class RollingDice
{

    // Creates two Die objects and rolls them several times.
    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("Die One: " + die1 + ", Die Two: " + die2);
        System.out.println("New sum: " + sum);
    }
}

Sample Run

Die One: 5, Die Two: 2
Die One: 1, Die Two: 4
Sum: 5
Die One: 4, Die Two: 2
New sum: 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;
    }
}
public void setFaceValue (int value)
{
    faceValue = value;
}

public int getFaceValue()
{
    return faceValue;
}

public String toString()
{
    String result = Integer.toString(faceValue);
    return result;
}
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
Method Control Flow

• The called method is often part of another class or object

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)
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);
    year ++;
}
```
acct2.addInterest(0.03);
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;
}
```

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

<table>
<thead>
<tr>
<th>datatypes:</th>
<th>Clients (Driver classes):</th>
</tr>
</thead>
<tbody>
<tr>
<td>– Account</td>
<td>– Transactions, OnePercent</td>
</tr>
<tr>
<td>– Die</td>
<td>– RollingDice</td>
</tr>
<tr>
<td>– Shoe</td>
<td>– YouVeGotShoes (Project)</td>
</tr>
<tr>
<td>– Person</td>
<td>– PeopleBeingPeople (Lab)</td>
</tr>
</tbody>
</table>

**Next:** UML class diagrams, visibility modifiers, graphical objects,
UML Class Diagrams

UML = Unified Modelling Language

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

```
RollingDice

main (args : String[]) : void

Die

faceValue : int
roll() : int
setFaceValue (value : int) : void
getFaceValue() : int
toString() : String
```
UML class diagram for Transactions program?
### 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>Rectangle</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, Set height, Compute volume, Compute filled weight</td>
</tr>
<tr>
<td>Flight</td>
<td>Airline, Flight 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, Set salary, 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
It is possible for a class to access the instance data of another class directly
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!**

```java
acct1.name = “Joe”;
```

**Account.java**

- `deposit()`
- `withdraw()`
- `getBalance()`

```
name
acctNumber
balance
```
Use Accessors & Mutators - **RIGHT**

- Indirect access through methods
- accessors and mutators ("getters" and "setters")
- Usually named \( \text{getX}() \) and \( \text{setX}() \)

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

---

**Transactions.java**

**Account.java**
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:

  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:

  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.print("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></td>
<td>NO</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>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes</td>
<td>Yes, for support methods only</td>
</tr>
</tbody>
</table>

See also [ImInUrClassMessingUrInstanceData.java](#)