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

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();
2. Object instantiation

Old example:
```java
Scanner scan = new Scanner (System.in);
```
**Invokes the Scanner constructor, which is a special method that sets up the object**

using 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)
class Account

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

    Account (String x, int y, double z)
    {  // Constructor
        name = x;  // Data, aka, Instance Variables
        acctNumber = y;
        balance = z;

        acctNumber = acctNumber + x;  // method definition
        balance = balance + x;
        acctNumber = acctNumber + x;
        balance = balance + x;
    }

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

    public void withdraw (double x, double fee)
    {  // withdraw()
        // the fee
        balance = balance - x - fee;
    }

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

    public String toString ()
    {  // toString()
        return acctNumber + \t"Name": "Ted Murphy"
            acctNumber + \t"Balance": $759.32;
    }
}
```

### Transactions class:

Creating Account objects

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

Sample Run

```
72354   Edward Demsey      $102.56
69713   Jane Smith        $40.00
93757   Edward Demsey      $759.32
```

### Transactions class:

```java
class Transactions
{
    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 ("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 (acct1);
    }
```

CSC 1051 M.A. Papalaskari, Villanova University
Designing Classes

### Transactions class:

Creating more Account objects

```java
Account acct1 = new Account("Ted Murphy", 72354, 102.56);
Account acct2 = new Account("Jane Smith", 69713, 40.00);
```

#### Using methods: `deposit()`

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

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

#### Using methods: `withdraw()`

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

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

```java
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;
}
```

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

**returning a value: getBalance()**

```java
double amount = acct1.getBalance();  
System.out.println (acct1.toString());
```

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

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

**returning a value: toString() method**

```java
double amount = acct1.getBalance();  
System.out.println (acct1.toString());
```

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

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

**returning a value: toString() method**

```java
double amount = acct1.getBalance();  
System.out.println (acct1.toString());
```

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

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

//********************************************************************
// 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 ('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

Another Example: RollingDice.java

//********************************************************************
// 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
  public Die() {
    faceValue = 1;
  }
  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**

- Common methods in Java
- Method control flow
- Method definition
  - Parameters
  - `return` statement
- UML class diagrams
- Encapsulation

**Common methods in Java classes**

- **Constructor** – always the same name as class, *e.g.*:
  - `public Account(String x, int y)`
  - `public Die()`
  - Always the same name as class
  - No return value
- `toString()` – returns a String corresponding to the object.
  - `public String toString()`
  - Always the exact same heading
- **getters (or accessors)** – return instance variable’s value.
  - `public int getFaceValue()`
  - No parameters
  - Return type is the same as the instance variable’s
- **setters (or mutators)** – to set or change an instance variable’s value
  - `public void setFaceValue(int value)`
  - One parameter, same type as instance variable.
  - Return type *void*

**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);
}
```

**Method Control Flow**

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

```
compute
myMethod();
```
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)

Method Control Flow: example

- main
  - acct2.addInterest(0.03)
  - deposit()
  - acct2.addInterest(0.03)

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 ++;
  }
  ```

Method Control Flow

- (detail)
**Method definition: Example**
- parameters
- return type
- return statement

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

```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., \(a^2 + b^2\)).

```java
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.

```java
• 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:

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

Die
faceValue : int
roll() : int
setFaceValue (value : int) : void
getFaceValue() : int
toString() : String
```

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, Get, Set average</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Length, Width, Color</td>
<td>Set, Get, Set color</td>
</tr>
<tr>
<td>Aquarium</td>
<td>Material, Length, Weight</td>
<td>Set, Get, Set weight, Volume</td>
</tr>
<tr>
<td>Fight</td>
<td>Airline, Flight number, Origin, Destination city</td>
<td>Set, Get, Set status</td>
</tr>
<tr>
<td>Employee</td>
<td>Name, Department, Title, Salary</td>
<td>Set, Get, Calculate bonus</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
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!

**Account.java**

```
public class Account {
    private String name; // Private data
    private int acctNumber;
    private double balance;

    public void deposit() {
        // Method to deposit money
    }

    public void withdraw() {
        // Method to withdraw money
    }

    public double getBalance() {
        // Method to get balance
    }
}
```

**Transactions.java**

```
public class Transactions {
    public static void main(String[] args) {
        Account acct1 = new Account();
        acct1.name = "Joe";
        acct1.deposit();
        acct1.withdraw();
        acct1.getBalance();
    }
}
```

**Use Accessors & Mutators - **RIGHT**
- Indirect access through methods
  - accessors and mutators ("getters" and "setters")
  - Usually named getX() and setX()

**Account.java**

```
private class Account {
    private String name; // Private data
    private int acctNumber;
    private double balance;

    public void deposit() {
        // Method to deposit money
    }

    public void withdraw() {
        // Method to withdraw money
    }

    public double getBalance() {
        // Method to get balance
    }
}
```

**Transactions.java**

```
public class Transactions {
    public static void main(String[] args) {
        Account acct1 = new Account();
        acct1.deposit();
        acct1.withdraw();
        acct1.getBalance();
    }
}
```

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:
  ```java
  Error: ____________________________________________________
  ```
Public Constants… OK

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

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

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

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

**static** Public Constants… BETTER

It is better to declare constants as static

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

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

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

See also ImInUrClassMessingUrInstanceData.java