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

- **Data**
  - `int acctNumber;`
  - `double balance;`
  - `String name;`

- **Constructor**
  - `deposit()`
  - `withdraw()`
  - `getBalance()`
  - `toString()`

- **Methods**

**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

Old example:

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

Invokes the Scanner *constructor*, which is a special method that sets up the object
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!

acct1
acctNumber 72354
balance 102.56
name "Ted Murphy"
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)

Transactions.java
Author: MA Papalaskari
// 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);
        System.out.println(acct2);
        System.out.println(acct3);
    }
}

Account.java
Author: Lewis/Loftus  
// Simplified code by MA Papalaskari
// Represents a bank account with methods deposit and withdraw.
//********************************************************************
import java.text.NumberFormat;
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;
    }
    //-----------------------------------------------------------------
    // Deposit the specified amount into the account.
    //-----------------------------------------------------------------
    public void deposit (double x) {
        balance = balance + x;
    }
    //-----------------------------------------------------------------
    // Get the current balance.
    //-----------------------------------------------------------------
    public double getBalance () {
        return balance;
    }
    //-----------------------------------------------------------------
    // Convert the Account object to a String.
    //-----------------------------------------------------------------
    public String toString () {
        return name + " Account 
        Account Number: " + acctNumber + " Balance: " + NumberFormat.getCurrencyInstance().format(balance) + "
    }
}

Datatype
Client
Datatype
Client
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()
// Withdrawing a specified amount from the account and applies 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 (acct1.toString());
        System.out.println (acct2.toString());
        System.out.println (acct3.toString());
    }
}

Sample Run

<table>
<thead>
<tr>
<th>Account Number</th>
<th>Name</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>72354</td>
<td>Ted Murphy</td>
<td>$102.56</td>
</tr>
<tr>
<td>69713</td>
<td>Jane Smith</td>
<td>$40.00</td>
</tr>
<tr>
<td>93757</td>
<td>Edward Demsey</td>
<td>$759.32</td>
</tr>
<tr>
<td>72354</td>
<td>Ted Murphy</td>
<td>$65.91</td>
</tr>
<tr>
<td>69713</td>
<td>Jane Smith</td>
<td>$40.00</td>
</tr>
<tr>
<td>93757</td>
<td>Edward Demsey</td>
<td>$759.32</td>
</tr>
</tbody>
</table>
Account acct1 = new Account ("Ted Murphy", 72354, 102.56);

Transactions class

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

acct1: acctNumber 72354, balance 102.56, name "Ted Murphy"
Transactions class:

Creating more Account objects

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

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

acct1.deposit (25.85);

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

acct1.deposit (25.85);

// Deposits the specified amount into the account.
//
public void deposit (double x)
{
    balance = balance + x;
}
Another example: \texttt{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:

```java
acct1.withdraw()
```
Another example: `withdraw()`

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

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

acct1

<table>
<thead>
<tr>
<th>acctNumber</th>
<th>72354</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance</td>
<td>65.91</td>
</tr>
<tr>
<td>name</td>
<td>&quot;Ted Murphy&quot;</td>
</tr>
</tbody>
</table>
returning a value: `getBalance()`

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

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

[Diagram showing object `acct1` with properties `acctNumber`, `balance`, and `name`, and a method call to `getBalance` returning a value.]
returning a value: `getBalance()`

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

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

acct1

<table>
<thead>
<tr>
<th>acctNumber</th>
<th>72354</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance</td>
<td>102.56</td>
</tr>
<tr>
<td>name</td>
<td>&quot;Ted Murphy&quot;</td>
</tr>
</tbody>
</table>
returning a value: toString () method

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

"72354  Ted Murphy      $102.56"

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

returning a value: toString () method can be omitted!

acct1

acctNumber  72354
balance  102.56
name

"Ted Murphy"

CSC 1051 M.A. Papalaskari, Villanova University
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);
    }
}
/**
 * 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
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;
}
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
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 ++;
}
```

*effect: acct2.deposit(0.03*balance)*
acct2.addInterest(0.03);
Method Control Flow

• (detail)

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

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

CSC 1051 M.A. Papalaskari, Villanova University
**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.

### datatypes:
- Account
- Die
- Shoe
- Person

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

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

CSC 1051 M.A. Papalaskari, Villanova University
UML Class Diagrams

UML = Unified Modelling Language

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

```java
public class RollingDice {
    public static void main(String[] args) {
        Die die = new Die();
        die.setFaceValue(1);
        System.out.println(die.getFaceValue());
    }
}

class Die {
    private int faceValue;
    // Constructor, getters, setters
    public void roll() {
        // Logic to roll the die
    }
}
```
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&lt;br&gt;Address&lt;br&gt;Major&lt;br&gt;Grade point average</td>
<td>Set address&lt;br&gt;Set major&lt;br&gt;Compute grade point average</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Length&lt;br&gt;Width&lt;br&gt;Color</td>
<td>Set length&lt;br&gt;Set width&lt;br&gt;Set color</td>
</tr>
<tr>
<td>Aquarium</td>
<td>Material&lt;br&gt;Length&lt;br&gt;Width&lt;br&gt;Height</td>
<td>Set material&lt;br&gt;Set length&lt;br&gt;Set width&lt;br&gt;Set height&lt;br&gt;Compute volume&lt;br&gt;Compute filled weight</td>
</tr>
<tr>
<td>Flight</td>
<td>Airline&lt;br&gt;Flight number&lt;br&gt;Origin city&lt;br&gt;Destination city&lt;br&gt;Current status</td>
<td>Set airline&lt;br&gt;Set flight number&lt;br&gt;Determine status</td>
</tr>
<tr>
<td>Employee</td>
<td>Name&lt;br&gt;Department&lt;br&gt;Title&lt;br&gt;Salary</td>
<td>Set department&lt;br&gt;Set title&lt;br&gt;Set salary&lt;br&gt;Compute wages&lt;br&gt;Compute bonus&lt;br&gt;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

```
class Account {
    private String name;
    private String acctNumber;
    private double balance;

    public void deposit() {
        // deposit logic
    }

    public void withdraw() {
        // withdraw logic
    }

    public double getBalance() {
        return balance;
    }
}
```
Use Accessors & Mutators - **RIGHT**

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

**Transactions.java**

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

**Account.java**

```java
void deposit()
void withdraw()
int 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:

Error: ____________________________________________________
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);
```
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><strong>NO</strong></td>
<td><strong>Yes</strong></td>
</tr>
<tr>
<td></td>
<td>(but OK for public</td>
<td></td>
</tr>
<tr>
<td></td>
<td>constants)</td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td><strong>Yes</strong></td>
<td>**Yes, for support</td>
</tr>
<tr>
<td></td>
<td>methods only</td>
<td></td>
</tr>
</tbody>
</table>

See also [ImInUrClassMessingUrInstanceData.java](#)