Designing Classes

CSC 2014 – Java Bootcamp

Dr. Mary-Angela Papalaskari
Department of Computing Sciences
Villanova University
Where do objects come from?
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"
```

Java Bootcamp Dr. Papalaskari, Villanova University
1. Structure of class definition

**Account** class

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

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

The class is the **blueprint**

Classes define DATA and METHODS i.e., a **datatype**

**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
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.
2. Object instantiation

Creating Objects – our newly defined `Account` class:

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

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();
```
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);
```
Writing Classes

• So far, 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)
Datatype / Client (also referred to as "slave / driver" classes)

---

**Transactions.java**

Author: MA Papalaskari

Demonstrates the creation and use of multiple Account objects.

```java
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.

```java
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 () {
        balance = balance - 2.50;
    }
    public double getBalance () {
        return balance;
    }
    public String toString () {
        return name + " account number: " + acctNumber + " balance: " + balance;
    }
}
```

---

Java Bootcamp Dr. 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()
// 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();
    String acctNumber = "\t" + name + "\t" + fmt.format(balance);
    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);
      System.out.println (acct2);
      System.out.println (acct3);
   }
}
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);
    }
}
Transactions class:

Creating Account objects

Transactions class

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

Transactions class

Account class

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

Constructor method
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 = new Account();
acct1.deposit (25.85);

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

acct1.deposit (25.85);

// Deposits the specified amount into the account.
//
public void deposit (double x) {
    balance = balance + x;
}
acct1.deposit (25.85);

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

acct1.deposit (25.85);

// 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);
```

 acct1

 acctNumber 72354

 balance 128.41

 name "Ted Murphy"
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;
}
Account class: Another Example

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

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

acct1

- **acctNumber**: 72354
- **balance**: 65.91
- **name**: "Ted Murphy"
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
void deposit()
void withdraw()
int getBalance()
String toString()
```
**toString() method**

```java
toString() method
```

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

```java
acct1.toString();
```

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

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

Java Bootcamp Dr. Papalaskari, Villanova University
**toString() method**

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

```
acct1

acctNumber 72354
balance 102.56
name "Ted Murphy"
```

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

**can be omitted!**
getBalance () method

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>
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>
</table>
| Student    | Name
            | Address
            | Major
            | Grade point average | Set address
            | Set major
            | Compute grade point average |
| Rectangle  | Length
            | Width
            | Color                  | Set length
            | Set width
            | Set color                                    |
| Aquarium   | Material
            | Length
            | Width
            | Height                      | Set material
            | Set length
            | Set width
            | Set height
            | Compute volume
            | Compute filled weight                       |
| Flight     | Airline
            | Flight number
            | Origin city
            | Destination city
            | Current status | Set airline
            | Set flight number
            | Determine status                            |
| Employee   | Name
            | Department
            | Title
            | Salary                      | Set department
            | Set title
            | Set salary
            | Compute wages
            | Compute bonus
            | Compute taxes                               |
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);
    }
}
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 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;
}
Next: Focus on **Method definition**

- 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;
}
```

Java Bootcamp Dr. Papalaskari, Villanova University
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

```java
obj.doIt();
```

• 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 its other methods directly. Eg:

```java
public void addInterest(double rate) {
    deposit (rate*balance);
}
```
Invoking methods within the same class

- The main method can do this too!

```java
public static void main (String[] args)
{
    printManyGreetings(5);
}
```

- **assumes** `printManyGreetings()` is defined in the same class

  ➔ A convenient way to test methods
Invoking methods within the same class

- The main method can do this too!

```java
public static void main (String[] args) {
    printManyGreetings(5);
}
```

- assumes `printManyGreetings()` is defined in the same class

**NOTE:** Since `main()` is `static`, `printManyGreetings()` must also be `static`
Implementing `printManyGreetings()` method:

Invoking the method:

```java
printManyGreetings(5);
```
Method Control Flow revisited

• Recall, if the called method is in the same class, only the method name is needed
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:** Focus on method definition/invocation
UML Class Diagrams

UML = Unified Modelling Language

- Example: A UML class diagram for the RollingDice program:
UML class diagram for Transactions program?
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!*
- See `Account.java`
- See `ImInUrClassMessingUrInstanceData.java`

---

**Account.java**

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

---

`ImInUrClassMessingUrInstanceData.java`
Use Accessors & Mutators - **RIGHT**

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

**Transactions.java**

Example

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

**Account.java**

Data

Methods

```java
    deposit()
    withdraw()
    addInterest()
```

```java
    name
    acctNumber
    balance
```
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 your solution for the Account Class Exercise
• Add some code to the OnePercent.java class to modify the value of an instance variable, eg:
  ```java
  acct1.name = "Joe";
  ```
• This should work as expected
• Now modify Account.java – insert the modifier `private` in front of that variable declaration:
  ```java
  private String name;
  ```
• Re-compile the Account class and run your program again. Note the error you get.
public constants are 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 are ok

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

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

Usually, constants are declared `static`

A driver class (eg, OnePercent.java) can access this constant directly without creating an object:

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

```
System.out.print("Interest rate = "+Account.RATE);
```
### Visibility Modifiers – the Rules

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