Writing New Java Classes

15-110 Summer 2010
Margaret Reid-Miller
What Are Objects?

- What are objects and what are not objects?
  - **Objects**: bicycle, book, lamp, song, meeting
  - **Non-objects**: green, 30% of all pencils, large
- If you can touch it, name it, or talk about it, it is likely to be an object.
- Objects can be physical things or conceptual.
- Humans seem to want to think in terms of objects and their relationships with each other.
Object Features

- When we try to describe an object, we tend to
  - name or label it (e.g., Homer’s car)
  - say what it can do (e.g., turn, drive, brake)
  - list its attributes or properties (e.g., red, 4-door)
- Sometimes we clarify the way that it does something (e.g., how fast to drive or what direction to turn).
- Sometimes the properties are observable from the outside (visible); sometimes we can only inferred them from the way the object behaves (hidden).

Example:
- Visible: V-8 engine, has a sun-roof
- Hidden: The amount of fuel in the tank is indirectly determined by the fuel gauge.
Models and Programs

• Often computer programs model some process or system.

• A model is a simplified representation:
  • It includes features that are important for the aim of the system.
  • Excludes features that are not relevant to the situation.

• When we model objects we define what is common for all objects of a particular type and then state what is special about each individual object. (e.g., All cars have a color, have an engine, drive forward, and turn. But a particular car may be silver with a V8 engine).
Software Objects

• In object-oriented programming we describe types of objects by defining classes.

• A Java class definition contains
  • fields - what properties an object has
    • The values assigned to the fields define the state of the object. (e.g., the car is painted silver, has a half a tank of gas, and is stopped.)
  • methods - what behaviors (actions) an object can perform
    • Typically these actions supply or modify its state.
Software Objects

• A Java class is a “blue print” for creating objects of that type.
• We then can create multiple objects from that class and “fill in” the properties with values specific to each object, and ask the object to perform their behaviors.
• Every object belongs to one class and is an instance of the class.
• Types of objects that we have used are String, Scanner, File, PrintStream, Die, Spinner, TrainCar
Fields

- The **fields** (*instance variables*) of an object are the variables that define an object’s properties.
  Example: An object from an Elevator class might have the following fields:
  - the current floor,
  - the top floor,
  - the number of riders, and
  - the capacity.
- Once an object is created, each field has some value.
- These values define the **state** of the object and describe the current condition of the object.
public class Elevator {

    // Fields: The object state
    private int topFloor;       // maximum floor number
    private int currentFloor;
    private int capacity;       // max number of riders
    private int numRiders;

    // Methods: The object behaviors

}

Fields should be defined as **private** (visible to methods of the same class and hidden to methods of other classes).
Creating Objects

- A class provides a blueprint for objects of the type of the class.
- Use the `new` operator to *instantiate* (create) the object, followed by a call to the class constructor, which initializes the object's fields. The `new` operator returns a *reference* to the new object.

For example in the `main` method we might write:

```
Elevator weanLeft = new Elevator(8, 10);
```
Constructors

Constructors initializes all the fields of the object.

```java
public Elevator(int numberOfFloors,
                int maxRiders) {
    topFloor = numberOfFloors;
    currentFloor = 1;       // Starting floor
    capacity = maxRiders;
    numRiders = 0;          // Initially empty
}
```

NOTE: For each parameter, use a name different from the field names.
Constructors

- Constructors are like methods with two differences:
  - There is no return type. (The object reference returned always has the type of the class.)
  - The name of the constructor is always the name of the class.

- The constructor should initializes all the fields of the object.

- Any Java statement can be in the body of the constructor. For example it might check that a parameter is in the correct range of values.
Overloading

• We can have more than one constructor.
• Additional constructors can supply default values for fields that have no corresponding parameter.

```java
public Elevator(int numberOfFloors) {
    topFloor = numberOfFloors;
    currentFloor = 1;    // Starting floor
    capacity = 12;       // Standard capacity
    numRiders = 0;       // Initially empty
}
```
Sample *Client* Program

```java
public class ElevatorController {
    public static void main(String[] args) {
        Elevator weanLeft = new Elevator(8, 10);
        Elevator weanRight = new Elevator(8);
    }
}
```

Each object has its own copy of the fields.
(Instance) Methods

- The *behaviors* of an object are defined by the methods we write in the object’s class.
- These (instance) methods report or act upon the data of an object (instance of the class).
- One of the biggest benefits of object-oriented programming is that we put both the data and methods together.
- The program code that creates and uses these objects, called the *client* code, is now more expressive and concise. It simply asks the objects to perform their behaviors (i.e., to provide a *service*).
Accessors

• **Accessors** are methods that access an object’s state without changing the state.

Examples:

```java
public int getNumRiders() {
    return numRiders;
}
```

```java
public int getCurrentFloor() {
    return currentFloor;
}
```

Accessors should be defined as **public** (visible by everyone).
public boolean isFull() {
    return numRiders == capacity;
}

This accessor compares the number of riders with the capacity and returns true if the elevator is at its maximum capacity and false otherwise. It does not change the state of the object.
Using an Accessor

- How can we invoke the getNumRiders method in the main method?

num = Elevator.getNumRiders();              NO!
leftNum = weanLeft.getNumRiders(3);         NO!
weanLeft.getNumRiders();                    NO!
int weanLeftNum = weanLeft.getNumRiders(); YES!
System.out.println(
    weanLeft.getNumRiders());               YES!
if (weanLeft.getNumRiders() < 10){
    System.out.println(“not full”);        YES!
}
Mutators

- **Mutators** are methods that can change an object’s state.

```java
public void addRiders(int numEntering) {
    if (numRiders + numEntering <= capacity) {
        numRiders = numRiders + numEntering;
    } else {
        numRiders = capacity;
    }
}
```

Mutators should be defined as `public` (visible to methods of every class).
Mutators

- *Mutators* should ensure that the object’s state stays consistent, e.g., that the number of riders is never greater than the elevator capacity or negative, and that the elevator never goes to a nonexistent floor of the building.

```java
public void goUpOneFloor() {
    if (currentFloor < topFloor)
        currentFloor++;
}
```

In another class, to call this method:

```java
weanLeft.goUpOneFloor();
```
The toString Method

• Every class should have a toString() method that returns a string that represents the current state of the object.

```java
public String toString() {
    return "current floor = " + currentFloor
        + " top floor = " + topFloor
        + "\nnumber of riders = " + numRiders
        + " capacity = " + capacity;
}
```
Invoking toString()

Typically, toString is used for debugging.

```java
public class ElevatorController {
    public static void main(String[] args) {
        Elevator weanLeft = new Elevator(8, 10);
        Elevator weanRight = new Elevator(8);
        weanLeft.addRiders(5);
        weanLeft.goUpOneFloor();
        System.out.println("Left: " + weanLeft);
    }
}
```

Java invokes toString on object references in print statements and string concatenation expressions automatically.
The equals Method

• Every class should have an `equals` method that compares two objects and returns `true` if they have the same `state` and `false` otherwise.

```java
public boolean equals(Elevator other) {
    return topFloor == other.topFloor &&
    currentFloor == other.currentFloor &&
    capacity == other.capacity &&
    numRiders == other.numRiders;
}
```

In another class, to call this method:

```java
if (weanLeft.equals(weanRight))
```
The this Reference

- The reserved word `this` allows an object to refer to itself within its class. (It is sometimes called the *implicit parameter*.)

```java
public Elevator(int topFloor, int capacity) {
  this.topFloor = topFloor;
  
  ...
}
```

```java
public boolean equals(Elevator other) {
  return this.topFloor == other.topFloor &&
  ...
}
```

**Usage:** if (weanLeft.equals(weanRight)) ...
Remarks

- The fields (properties) of a class define what space in memory is needed to hold the current state of the object. They should be private.
- The public methods of a class define the behaviors of the objects. These methods define the interface to the object; The interface defines what client code in other classes can ask the objects to do. (Private methods are “helper” methods that help the public methods to their job.)
- Fields and methods for the instances of the class (objects) do not include the static keyword.