CSE 142

Classes and Updating Instance Variables

(updated)

Overview

- Review
  - Class declarations
  - Constructors and Methods
  - Variable declaration and assignment (binding)
- Today
  - Defining a class
  - Public and private class members
  - Updating bindings – assignments that change variables
- Reading
  - Dugan notes: part of ch. 6, 7
  - Niño & Hosch: ch. 5

A Scenario

- **Suppose we want to define a class to represent bank account objects**
- **Design issues**
  - **What sort of *behavior* should it provide, i.e., what messages should it understand (what methods and parameters)?**
  - **What sort of instance variables are needed, i.e., what kind of data needs to be stored in a BankAccount object? (This collection of instance variables is often called the object's *state.*)**

    What are appropriate *types* for those variables

Design Your Bank Account Here

Notes

Check: Does the Interface Make Sense

- **Before investing in detailed coding, try the code out from the client's (user's) perspective**
  - **Does the object have the behavior we need? Anything missing?**

```
BankAccount checking =
    new BankAccount("Bill", 0001, 41620000000.0);
checking.deposit(17.42);
double currentBalance =
    checking.getBalance( );
```

Implementation – Instance Variables

- **What sort of data do we need? What are the types?**

```
// instance variables
private String  accountName;      // account holder's name
private int     accountNumber;    // account number
private double balance;           // balance in US dollars
```

  - **Review: These are *declarations* without assignment of initial values**

Visibility: Public & Private

- **Public vs private: all members of a class (instance variables and methods) can be labeled public or private**
  - **public: can be directly accessed anywhere the class or its instances can be used**
  - **private: can only be accessed by code inside the class itself**
- **Design rules**
  - **Constructors and methods that are part of the class *interface* (or *specification*) should be public**
  - **Everything else should be private**
- **Leads to better modularity; limits possibilities for bugs**

Constructors

- **A well-designed class almost always contains one or more constructors**
  - **Executed automatically when a new instance of the class is created**
  - **Allows programmer of the class to guarantee that new instances are properly initialized**

    (Other code in the class can rely on this having been done)

Constructors for BankAccount

```
/** Construct a new BankAccount
 * @param accountName  name of this account
 * @param accountNumber number of this account
 * @param initialBalance  initial balance of this account */
public BankAccount(String accountName, int accountNumber,
                                double initialBalance) {
    this.accountName = accountName;
    this.accountNumber = accountNumber;
    this.balance = initialBalance;
}
```

- **Note use of this.name to refer to instance variables, while name refers to parameter (local name) of the constructor**

Multiple Constructors

- **A class may have many constructors**
  - **Must differ in number or types of parameters (or both)**
  - **Compiler picks correct constructor depending on number and type of arguments when object is created (new)**

```
/** Construct a new BankAccount with a balance of 0.0
 * @param accountName  name of this account
 * @param accountNumber number of this account */
public BankAccount(String accountName, int accountNumber) {
    this.accountName = accountName;
    this.accountNumber = accountNumber;
    this.balance = 0.0;
}
```

Using Private Methods

- Observation: The two constructors contain almost identical (redundant) code
- Design principle: take redundant code and turn it into a (possibly parameterized) method
  - Do things only once in one place – less chance for errors, easier to modify, etc.
- If the new method is not part of the public interface of the class, it should be private, so code outside the class can't access it

Private Method to Initialize BankAccounts

```
/* Initialize this BankAccount
 * @param accountName  name of this account
 * @param accountNumber number of this account
 * @param initialBalance  initial balance of this account */
private void initialize(String accountName, int accountNumber,
                                double initialBalance) {
    this.accountName = accountName;
    this.accountNumber = accountNumber;
    this.balance = initialBalance;
}
```

- **The method name is arbitrary; initialize seems like a good choice here**

Modified Constructors

```
/** Construct a new BankAccount
 * @param accountName  name of this account
 * @param accountNumber number of this account
 * @param initialBalance  initial balance of this account */
public BankAccount(String accountName, int accountNumber,
                                double initialBalance) {
    this.initialize(accountName, accountNumber, initialBalance);
}
/** Construct a new BankAccount with a balance of 0.0
 * @param accountName  name of this account
 * @param accountNumber number of this account */
public BankAccount(String accountName, int accountNumber) {
    this.initialize(accountName, accountNumber, 0.0);
}
```

Accessor Methods

- **Instance variables should be private. If the client (user) code needs access to the values of these variables, supply value-returning methods to provide this access**

```
/** Get the balance of this account
 * @return  current account balance in dollars
 */
public double getBalance( ) {
    return this.balance;
}
```

  - **Naming convention: a method that returns the value of field named, say, xyzzy, is named getXyzzy**

Declaration and Assignment Reviewed (1)

- **We've seen two patterns for creating names and binding them to values**
- **A declaration with an initial value**

  <type> <name> = <expression>;

  both introduces a new name and specifies its value
- **Execution**
  - **(0) create the name**
  - **(1) evaluate the <expression>**
  - **(2) bind the <name> to the value of the <expression>**

Declaration and Assignment Reviewed (2)

- **A declaration may omit the initial value**

  <type> <name>; // typical for class instance variables

- **A variable declared this way may be bound to a value later using an assignment statement (often in a constructor or a method called by a constructor)**

  <object name> . <name> = <expression>;

- **Execution of an assignment statement**

  (1) Evaluate the expression

  (2) Bind the <name> to the value of the <expression>

  - **Any <names> appearing in the <expression> must have been previously initialized**

Using Assignment to Change Bindings

- **An assignment statement may also be used to *change* the value bound to a variable**
  - **Can be used to rebind the value of both instance variables or local variables in methods**
  - **Pattern for assignment to local variables (names) in a method**

    <name> = <expression>;
  - **Pattern for assignment to object's instance variable**

    <object name> . <instance variable name> = <expression>;
  - **Same execution: (0) evaluate <expression>, (1) bind <name>**
  - **The name being assigned may appear in the <expression> (!)**

    No ambiguity: the old value is used to evaluate the expression, then the name is rebound to the new value. ☚

BankAccount setName Method

- **Client code may need to be able to change the name of an account**

```
/** Change the name of this BankAccount
 * @param newName  new name for the account
 */
public void setName(String newName) {
    this.accountName = newName;
}
```

  - **Naming convention: a method that changes the value associated with field xyzzy is normally named setXyzzy**

BankAccount Deposit Method

```
/** Deposit money in this BankAccount
 * @param amount  amount of money to be deposited
 */
public void deposit(double amount) {
    this.balance = this.balance + amount;
}
```

- **Be sure you understand how execution of this method works!**
- **Be sure you understand why the following statement makes sense and what it does:**

```
this.balance = this.balance + 1;
```

BankAccount Withdraw Method

```
/** Withdraw money from this BankAccount
 * @param amount  amount of money to withdraw
 * @return  amount of money withdrawn from account
 */
public double withdraw(double amount) {
    this.balance = this.balance - amount;
    return amount;
}
```

- **What if the amount is greater than the current balance?**
  - **Maybe it would be nice to detect this and react appropriately…**

Transfer Money Between Accounts

- **Idea: given two bank accounts**

```
BankAccount  student = new BankAccount("Huskie", 0154738, 17.42);
BankAccount parents = new BankAccount("Mom & Dad", 148099, 2543.12);
```

would like to have a method to transfer funds from one account to another

```
parents.transferTo(student, 250.00);
```

Method transferTo

```
/** Transfer funds from this account to another
 * @param destination  BankAccount to receive funds from this account
 * @param amount    amount to transfer */
public void transferTo(BankAccount destination, double amount) {
    // deduct balance from this account
    this.balance = this.balance – amount;
    // increase balance of destination account
    destination.balance = destination.balance + amount;
```

- **Method transferTo has access to private instance variables of both accounts since it is a method in class BankAccount**
- **But what if there isn't enough money in the original account?**