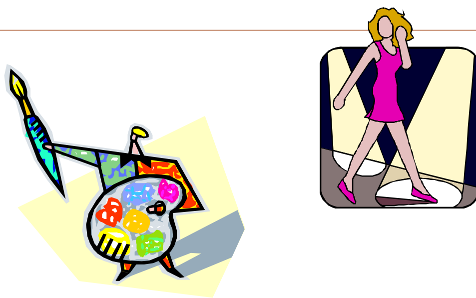
CSE 143 Java

Programming as Modeling

Reading: Ch. 1-6



Building Virtual Worlds

- Much of programming can be viewed as building a **model** of a real or imaginary world in the computer
  - a banking program models real banks
  - a checkers program models a real game
  - a fantasy game program models an imaginary world
  - a word processor models an intelligent typewriter
- Running the program (the model) simulates what would happen in the modeled world
- Often it's a lot easier or safer to build models than the real thing
  - Example: a tornado simulator

Java Tools for Modeling

- **Classes** in Java model *things* in the (real or imaginary) world
  - Accounts
  - Checkerboard, pieces, players
  - Characters, monsters, obstacles, weapons, treasure, scores
  - Documents, paragraphs, words, symbols, smart paper-clip
- A class describes a *template* for things;
  an **instance** is a *particular* thing
- **Constructors** model ways to create new instances
- **Methods** model *actions* that these things can perform
- **Messages** (method calls) model requests from one thing to another
- **Instance variables** model the state or properties of things

What Makes a Good Model?

- Often, closer the model matches the (real or imaginary) world, the better
  - More likely it's an accurate model
  - Easier for human readers of the program to understand what's going on in the program
- Sometimes, a too detailed model of reality is not a good thing. Why?

What Else Makes a Good Model?

- The easier the model is to extend & evolve, the better
  - May want to extend the model...
  - May need to change the model...
- Sad fact of life: "A Program is Never Finished"
- Why??

A Java Tool for Good Modeling

- One way to aid evolution is to define good **interfaces** separate from the implementation
- An interface specifies to clients (users of the class) what are the operations (methods) that can be invoked; anything else in the class is hidden
  - Clients get a simpler interface to learn
  - Implementors protect their ability to change the implementation over time without affecting clients
- In Java: `public` vs. `private`
  - Instance variables should usually be private

Behavior vs. State

- An interface prescribes only behavior (methods, operations, queries)
- The state (properties) are best left hidden
  - hidden, or accessible only through methods
- Example: Bank accounts have balances
  - Does this mean they must have a "balance" instance variable??
- Keeping behavior and state separate is an important aspect of design
  - important, and often difficult

Which is More Fundamental?

- Behavior or State?
- What do you think, and why?

The High vs. The Low

- Some aspects of system design are very high-level
- Yet… programming requires attention to low-level details
- This spectrum is one thing that makes our job hard
  - hard, and interesting

A Review Example

```
/** A Bank Account */

public class BankAccount {
   private double balance;      // the current balance of the account
   private String ownerName;   // the name of the person who owns this account
   private int accountNumber;  // the account number of this account

   /** Create a new bank account with a zero balance and a unique account number
         @param ownerName   the name of the person who owns this account */
   public BankAccount(String ownerName) {
      this.ownerName = ownerName;
      this.balance = 0.0;
      this.assignNewAccountNumber( );
   }

   …
```

Bank Example (2)

```
…

/** Assign this account a new unique account number */
private void assignNewAccountNumber() {
  this.accountNumber = …;
}

/** Return the current balance.
    @return the current balance  */
public double getBalance() {
    return this.balance;
}

…
```


Bank Example (3)

```
…
/** Deposit into account.
    @param amount the amount to deposit
    @return whether or not the transaction was successful */
public boolean deposit(double amount) {
    return this.updateBalance(amount);
}

/** Withdraw from account.
    @param amount the amount to withdraw
    @return whether or not the transaction was successful  */
public boolean withdraw(double amount) {
    return this.updateBalance(- amount);
}
…
```


Bank Example (4)

```
…
/** A helper method that adds its argument to the balance, if it doesn't cause overdraft.
    @param amount the amount to add to the balance (negative to withdraw)
    @return whether or not the transaction was successful   */
private boolean updateBalance(double amount) {
  if (this.balance + amount < 0) {
    // don't change the balance, if this would overdraw it.  print an error message instead.
    System.out.println("Sorry, you don't have that much money to withdraw.");
    return false;
  } else {
    // update the balance
    this.balance = this.balance + amount;
    return true;
  }
}
…
```


A Recommended Practice for All Classes

A method with this signature:

Public String toString( );

```
/** Compute a string representation of the account, e.g. for printing out */
public String toString( ) {
  return "BankAccount#" + this.accountNumber +
    " (owned by " + this.ownerName + "): current balance: " + this.balance;
}
```


toString

- Good while debugging

  System.out.println(myObject.toString());
- Java treats toString in a special way
  - In many cases, will automatically call toString when a String value is needed:

  System.out.println(myObject);
- Secret Java lore:
  - All Objects in Java have a built-in, default toString method
  - So why define your own??

Another Good Practice

- A static method in each class, just for testing it. No special name.

```
/** A method to test out some of the BankAccount operations */
public static void test() {
  BankAccount myAccount = new BankAccount("Joe Bob");
  myAccount.deposit(100.00);
  myAccount.deposit(250.00);
  myAccount.withdraw(50.00);
  System.out.println(myAccount);  // automatically calls myAccount.toString()
}

} // end of BankAccount
```