Handout #11
Nick Parlante

Java 4

Today: Collections, Abstract superclass inheritance

Collections

Built-in classes for storage (like the C++ STL)
Collection type (sequence / set) -- ArrayList is the most useful
Map type (hash table / dictionary)-- HashMap is the most useful
See the Sun docs: http: / / java.sun.com / docs / books / tutorial / collections /

Collection Design

As much as possible, all the various classes implement the same interface so they use the same method names (e.g. add()), so you can substitute one type of collection for another. This also makes it easier to learn, since the method names are all consistent.

Only store pointers to elements

- Can store pointers to objects such as Strings or arrays, but cannot store a primitive like an int. If you need to store an int, use the Integer class (or Float, or Boolean, or Double). This is one irritating area of java, but it's understandable from a language implementation point of view.

CT collection source forgets class -- cast back

- Internally, the collection classes treat all elements as Object pointers.
- The client needs to cast the pointer to the correct class when getting elements out.

At runtime, the objects remember their class.

Collection Details

There are a few key, basic methods...

constructor() -- collection with no elements

- Actually, a Java interface cannot specify a ctor or static method, but all the collection classes implement the default ctor at a minimum.

int size() -- number of elements in the collection

- This could have been called getSize() or getLength(), but they kept the name size() to remain compatible with the old Vector class.

boolean add(Object ptr)

- Add a new pointer / element to the collection. Adds to the "end" for collections that have an ordering. Returns true if the collection is modified (it might not be when adding to a Set type collection).

iterator()

- Return a new iterator set up to iterate through the collection and remove elements. Do not add to the collection while iterating.
- The iterator responds to... hasNext() true if more elements, next() return the next element, and remove() removes the previous elem returned by next()

Utilities -- these are most likely implemented to just call the basic methods above

boolean isEmpty()
boolean contains(Object o) -- iterative search
boolean remove(Object o) -- iterative remove
boolean addAll(Collection c) -- true if receiver changed
... and so on

ArrayList

Replaces the old "Vector" class
add() -- add pointer to end
int size() -- number of elements
Object get(int index) -- retrieve the elem pointer, indexed (0..len-1)
iterator() -- return an iterator object to iterate over the array list

ArrayList Demo Code

```
public static void demoArrayList() {
   ArrayList strings = new ArrayList();

   // add things...
   for (int i= 0; i<10; i++) {
      // Make a String object out of the int
      String numString = Integer.toString(i);
      strings.add(numString); // add an elem to the collection
   }

   // access the length
   System.out.println("size:" + strings.size());


   // ArrayList supports a for-loop access style...
   for (int i=0; i<strings.size(); i++) {
      String string = (String) strings.get(i);
      // Note: cast the get() to its actual class
      System.out.println(string);
   }

   // ArrayList also supports the "iterator" style...
   Iterator it = strings.iterator();
   while (it.hasNext()) {
      String string = (String) it.next(); // get pointer to elem
      System.out.println(string);
   }

   // Call toString()
   System.out.println("to string:" + strings.toString());

   // Iterate through and delete
   it = strings.iterator();   // get a new iterator (at the beginnnig again)
   while (it.hasNext()) {
      it.next();  // get pointer to elem, and discard
      it.remove();   // remove the above elem
   }
}
```

Abstract Superclass

Factor Common Code Up

Several related classes with overlapping code
Factor common code up into a common superclass
Examples
    AbstractCollection class in java libraries
    Account example below

Abstract Method

The "abstract" keyword can be added to a method.
    e.g. public abstract void mustImplement(); / / note: no { .. }
An abstract method defines the method name and arguments, but there's no method code. Subclasses **must** provide an implementation.

Abstract Class

The "abstract" keyword can be applied to a class
    e.g. public abstract class Account { ...
A class that has one or more abstract methods is abstract -- it cannot be instantiated. "New" may not be used to create instances of the Abstract class.
A class is not abstract if all of the abstract methods of its superclasses have definitions.

Abstract Super Class

A common superclass for several subclasses.
Factor up common behavior
Define the methods they all respond to.
Methods that subclasses should implement are declared abstract
Instances of the subclasses are created, but no instances of the superclass

Clever Factoring Style

Common Superclass
    Factor common behavior up into a superclass. The superclass sends itself messages to invoke various parts of its behavior.
Special Subclasses
    Subclasses are as short as possible.
    Rely on the superclass methods for common behavior.
    Use incisive overriding to customize behavior with the minimum of code
    Rely on the "pop-down" behavior -- control pops down to the subclass for overridden behavior, and then returns to the superclass to continue the common code.

Polymorphism

Given an array of Account[] -- pointers with the CT type to the superclass

Send them messages like, withdraw() -- control pops down to the correct subclass depending on its RT type.

Account Example

The Account example demonstrates the clever-factoring technique.

Consider an object-oriented design for the following problem. You need to store information for bank accounts. For purposes of the problem, assume that you only need to store the current balance, and the total number of transactions for each account. The goal for the problem is to avoid duplicating code between the three types of account. An account needs to respond to the following messages:

- constructor(initialBalance)
- deposit(amount)
- withdraw(amount)
- endMonth()
  Apply the end-of-month charge, print out a summary, zero the transaction count.

There are three types of account:

Normal: There is a fixed $5.00 fee at the end of the month.

Nickle 'n Dime: Each withdrawal generates a $0.50 fee — the total fee is charged at the end of the month.

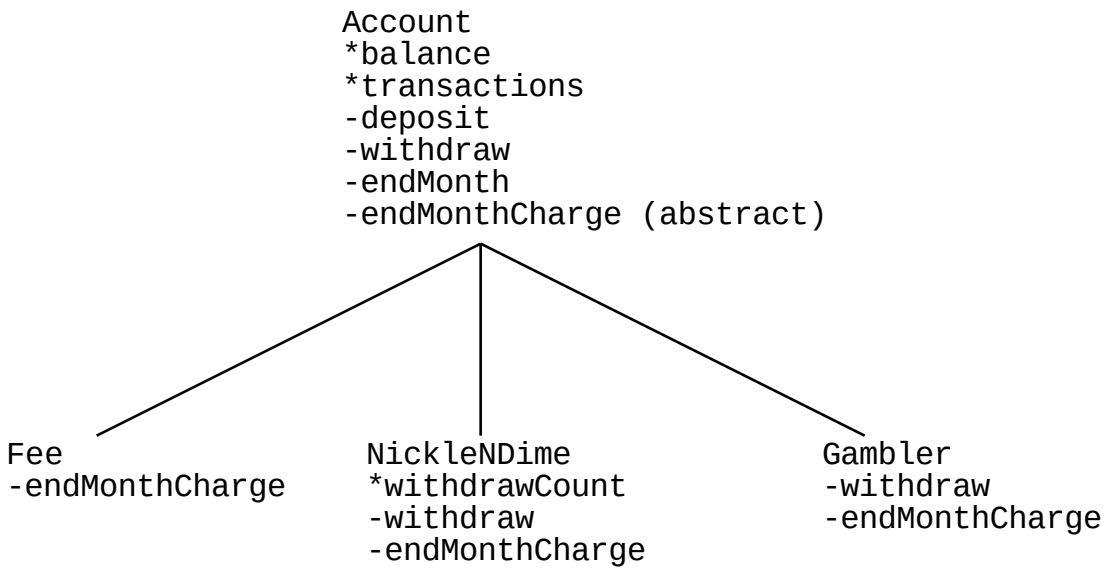
The Gambler: A withdrawal returns the requested amount of money, however the amount deducted from the balance is as follows: there is a 0.49 probability that no money will actually be subtracted from the balance. There is a 0.51 probability that twice the amount actually withdrawn will be subtracted. There is no monthly fee.

1. Factoring

   The point: arrange the three classes in a hierarchy below a common Account class. Factor common behavior up into Account. Mostly the classes use the default behavior. For key behaviors, subclasses override the default behavior e.g. Gambler.withdraw(). This keeps the subclasses very short with most of the code factored up to the superclass.

2. Abstract Methods

   A method declared "abstract" defines no code. It just defines the prototype, and requires subclasses to provide code. In the code below, the endMonthCharge() method is declared abstract in Account, so the subclasses must provide a definition.



Account Code

```
// Account.java
/*
 The Account class is an abstract super class with the default
 characteristics of a bank account. It maintains a balance
 and a current number of transactions.
 There are default implementation for deposit(), withdraw(),
 and endMonth() (prints out the end-month-summary). However
 the endMonthCharge() method is abstract, and so must
 be defined by each subclass.

 This is a classic structure of using clever factoring to pull
 common behavior up to the superclass.
 The resulting sublclasses are very thin.
*/

public abstract class Account  {
   protected double balance;     // protected = avaiable to subclasses
   protected int transactions;

   public Account(double balance) {
      this.balance = balance;
      transactions = 0;
   }

   // Withdraws the given amount and counts a transaction
   public void withdraw(double amt) {
      balance = balance - amt;
      transactions++;
   }

   // Deposits the given amount and counts a transaction
   public void deposit(double amt) {
      balance = balance + amt;
      transactions++;
   }
```

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


/*
 Sent to the account at the end of the month so
 it can settle fees and print a summary.
 Relies on the endMonthCharge() method for
 each class to implement its own charge policy.
 Then does the common account printing and maintenance.
*/
public void endMonth() {
   // 1. Pop down to the subclass for their
   // specific charge policy (abstract method)
   endMonthCharge();


   // 2. now the code common to all classes

   // Get our RT class name -- just showing off
   // some of Java's dynamic "reflection" stuff.
   // (Never use a string like this for switch() logic.)
   String myClassName = (getClass()).getName();

   System.out.println("transactions:" + transactions +
      "\t balance:" + balance + "\t(" + myClassName + ")");

   transactions = 0;
}


/*
 Applies the end-of-month charge to the account.
 This is "abstract" so subclasses must override
 and provide a definition. At run time, this will
 "pop down" to the subclass definition.
*/
protected abstract void endMonthCharge();



//----
// Demo Code
//----

// Returns a random int in the range [0..bound-1]
// NOTE static method -- there is no reason for this
// to execute against a particular Account receiver --
// "balance", "transactions" are not relevant.
// (In Java 2, the Random class has a nextInt(n) method
// that does this, but it's not static.)
static int randomInt(int bound) {
   double x = Math.random();
   return ((int)(x * bound)); // uses double->int truncation
}

// Return a new random account of a random type.
```

```
private static Account randomAccount() {
   int pick = randomInt(3);
   Account result = null;
   switch (pick) {
      case 0: result = new Gambler(randomInt(100)); break;
      case 1: result = new NickleNDime(randomInt(100)); break;
      case 2: result = new Fee(randomInt(100)); break;
   }

   /****
   // Another way to create new instances -- needs a default ctor
   try {
      Class gClass = Class.forName("Gambler");
      result = (Gambler) gClass.newInstance();
   }
   catch (Exception e) {
      e.printStackTrace();
   }
   ****/

   return(result);
}


private static final int NUM_ACCOUNTS = 20;


// Demo polymorphism across Accounts.
public static void main(String args[]) {

   // 1. Build an array of assorted accounts
   Account[] accounts = new Account[NUM_ACCOUNTS];

   // Allocate all the Account objects.
   for (int i = 0; i<accounts.length; i++) {
      accounts[i] = Account.randomAccount();
   }

   // 2. Simulate a bunch of transactions
   for (int day = 1; day<=31; day++) {
      int accountNum = randomInt(accounts.length); // choose an account
      if (randomInt(2) == 0) {   // do something to that account
         accounts[accountNum].withdraw(randomInt(100) + 1);//Polymorphism Yay!
      }
      else {
         accounts[accountNum].deposit(randomInt(100)  + 1);
      }
   }

   // 3. Have each account print its state
   System.out.println("End of month summaries...");
   for (int acct = 0; acct<accounts.length; acct++) {
      accounts[acct].endMonth(); // Polymorphism Yay!
   }
}
```

```
// output
/*
End of month summaries...
transactions:1  balance:-1.0  (Fee)
transactions:5  balance:-84.0 (NickleNDime)
transactions:2  balance:43.5  (NickleNDime)
transactions:1  balance:90.0  (NickleNDime)
transactions:2  balance:89.0  (Fee)
transactions:1  balance:1.0   (Gambler)
transactions:1  balance:88.0  (NickleNDime)
transactions:1  balance:150.0 (Gambler)
transactions:6  balance:-19.5 (NickleNDime)
transactions:2  balance:-29.0 (Fee)
transactions:4  balance:226.0 (Gambler)
transactions:1  balance:86.0  (Gambler)
transactions:2  balance:70.0  (Fee)
transactions:2  balance:131.5 (NickleNDime)
transactions:4  balance:-42.5 (NickleNDime)
transactions:2  balance:-20.5 (NickleNDime)
transactions:3  balance:85.0  (Fee)
transactions:1  balance:-71.0 (Gambler)
transactions:2  balance:-175.0   (Gambler)
transactions:2  balance:-48.0 (Fee)
*/
};


/*
 Things to notice.

 -Because the Account ctor takes an argument, all the subclasses need a ctor
 so they can pass the right value up. This chore can be avoided if the superclass
 has a default ctor.

 -Suppose we want to forbid negative balance -- all the classes
 "bottleneck" through withdraw(), so we just need to implement something
  in that one place. Bottlenecking common code through one place is good.

 -Note the "polymorphism" of the demo in Account.main(). It can
 send Account obects deposit(), endMonth(), etc. messages and rely
 on the receivers to do the right thing.

 -Suppose we have a "Vegas" behavior where a person withdraws 500, and
 slightly later deposits(50). Could implement this up in Account..
 public void Vegas() {
   withdraw(500);
   // go lose 90% of the money
   deposit(50);
 }
 Depending on the class of the receiver, it will do the right thing.
 Exercise: trace the above on a Gambler object -- what is the sequence
 of methods that execute?
*/
```

```
// Fee

// An Account where there's a flat $5 fee per month.
// Implements endMonth() to get the fee effect.

public class Fee extends Account {

   public final double FEE = 5.00;

   public Fee(double balance) {
      super(balance);
   }

   public void endMonthCharge() {
      withdraw(FEE);
   }

};



// NickleNDime

// An Acccount subclass where there's a $0.50 fee per withdrawal.
// Overrides withdraw() to count the withdrawals and
// endMonthCharge() to levy the charge.

public class NickleNDime extends Account {

   public final double WITHDRAW_FEE = 0.50;

   int withdrawCount;

   public NickleNDime(double balance) {
      super(balance);
      withdrawCount = 0;;
   }

   public void withdraw(double amount) {
      super.withdraw(amount);
      withdrawCount++;
   }

   public void endMonthCharge() {
      withdraw(withdrawCount * WITHDRAW_FEE);
      withdrawCount = 0;
   }

};
```

```
// Gambler

// An Account where sometimes withdrawals deduct 0
// and sometimes they deduct twice the amount. No end of month fee.
// Has an empty implementation of endMonthCharge,
// and overrides withdraw() to get the interesting effect.

public class Gambler extends Account {

   public final double PAY_ODDS = 0.51;

   public Gambler(double balance) {
      super(balance);
   }

   public void withdraw(double amt) {

      if (Math.random()<= PAY_ODDS) {
         super.withdraw(2 * amt);   // unlucky
      }
      else {
         super.withdraw(0.0);    // lucky (still count the transaction)
      }
   }

   public void endMonthCharge() {
      // ha ha, we don't get charged anything!
   }

};
```

Account Points...

1. Gambler.withdraw() -- super

Notice we use super.withdraw() to use our superclass code. Do not repeat code that the superclass can do. Be careful if you find yourself copying code from the superclass and pasting it into the subclass.

2. Account.endMonth() -- pop-down

Sends itself the endMonthCharge() message -- this pops-down to the implementation in each subclass.

3. Account.main() -- polymorphism

Constructs and Account[] array
Iterates through, sending the withdraw() message
Pops-down to the right implementation of withdraw() depending on the RT type of the receiver