CS112 Lecture: Course Introduction

Last revised 1/17/08

Objectives:

1. Introduce the course.
2. Define programming
3. Introduce fundamental concepts of OO: object, class

Materials:

1. Questionnaire
2. Syllabi
3. BlueJ demonstrations: BankExample
4. On-line copy of this lecture for demonstration

I. Distribute, Pick up Questionnaire

II. What is this course all about?

A. It is about <u>programming.</u>

1. **Definition: programming is the process of spelling out the steps needed to accomplish some task in a form that can be *interpreted* by a *computer system*.**

   a) Like many definitions, this definition contains an additional term that must itself be defined. That term is "**interpret**".

   We say that a computer system interprets a program when it carries out the actions specified by that program.

   (1) Example: a web browser, together with the computer on which it resides, constitutes a computer system that can interpret web pages and display them on your screen. In the process, it must interact with the computer system that is hosting the page, and may have to also contact other systems to get images etc. that are linked to the page it is displaying.

   (2) Example: an electronic spreadsheet program, together with the computer on which it resides, constitutes a computer system that can interpret spreadsheet documents, including the various formulas embedded within them.

b) Programmability is what makes computer systems different from other machines.

   (1) Most machines are built to perform one task, and one task only.

   (2) Notice that some machines are programmable in a limited sense - e.g. they can be programmed to perform some task within a limited family of tasks.

   Examples?

   ASK

   (a) VCR - can be programmed to record specific shows on specific channels at specific times

   (b) Programmable thermostat - can be programmed to change the heat set point in a house at different times on different days.

   (3) A general-purpose computer can be programmed to perform a wide variety of tasks. We concentrate on these. In point of fact, many "programmable" devices actually contain an embedded computer that has been programmed to support the specific tasks the device is designed for.

2. Each given computer system typically requires that programs that it is to interpret be specified in a particular *language* - the language that the particular system is designed to interpret.

   Example: Web browsers are designed to interpret html - the hypertext markup language

3. Note that the programming language used may ultimately determine what tasks can and cannot be programmed.

   a) Example: because html has no ability to do calculations, it would be impossible to write an income tax calculator program in html. (Though you could embed a JavaScript program in the page to do that - but now we're talking a second language.)

   b) The most interesting programming languages are what we call *general purpose*, in the sense that they can be used to write a program to perform any task that can be programmed.

4. In this course, we will be learning a general-purpose programming language known as Java.

   a) However, it is important that you understand that the principles that we learn in the course are more important than the specific details of the Java language - they are applicable to any general-purpose programming language.

      Moreover, many of the principles we will learn are also applicable to special-purpose programming languages, such as the language used for specifying formulas in a spreadsheet.

   b) Indeed, if you do any programming ten years from now, there is a very good chance that the language you will use hasn't even been invented yet!

   c) The principles, then, are more important than the language specifics.

B. The course is about a particular kind of programming: <u>object-oriented programming.</u>

1. If a programming task is small enough, it really doesn't matter a whole lot how one approaches. But larger tasks need to be decomposed into subtasks in order to be accomplished.

   Example: lots of people can build a simple structure like a doghouse. But if someone tried to approach building a skyscraper the same way, the results would be disaster!

   Object-oriented programming is one approach to decomposition that is widely used and often fruitful. This is the approach we are going to take.

2. An object-oriented software system consists of a collection of interacting **objects**.

   a) An object is a software entity that typically models something in the real world and has three critical properties.

      (1) State - the object encapsulates some information about itself, which are sometimes called its **attributes.**

      (2) Behavior - the object can do some things on behalf of other objects

(3) **Identity** - the object is distinguishable from all other objects of the same general type.

(The definition of an object in terms of state, behavior, and identity is a "3 AM phone call fact.")

b) *Example:* yesterday in lab we worked with a simple example of an object-oriented system. One kind of object we worked with was a Circle.

(1) What did the <u>state</u> of a Circle include?

ASK

- its diameter
- its position on the screen
- its color
- its visibility

(2) What did the <u>behavior</u> of a Circle include?

ASK

- drawing itself on the screen
- various kinds of movement
- changing its diameter, color, or visibility

(3) Early in the lab, you saw the notion of <u>identity</u> grapically illustrated. How?

ASK

- you created two circles. moving one left the other unmoved.

c) *Example:* Consider a software system that deals with customer accounts at a bank. One kind of object would certainly be objects representing the individual accounts. These objects would have

(1) A state consisting of attributes like:

(a) Account number
(b) Owner
(c) Current balance
etc.

(2) Behaviors like:

(a) Deposit money
(b) Withdraw money
(c) Calculate interest
(d) Report current balance
etc.

(3) Of course, each account would have its own identity.

3. Each object in an object-oriented system belongs to some **class**.

   a) A class can be thought of as the set of all objects of the same kind - e.g. all the circle objects together constitute the Circle class, and all the bank account objects together constitute the BankAccount class. (Note the name: we follow a convention that classes have names that begin with an uppercase letter, and in which each new word also begins with an upper-case letter. A name having this form will always be the name of a class.)

   b) When we write OO programs, we define classes, and then use them as templates for constructing objects. Each individual object is called an instance of its class, and each object is an instance of exactly one class.

      Example: In lab, you created Circle objects by asking the Circle class to create a new instance.

      Example: For our bank account system, we might define a class known as BankAccount, and then use it to create the individual bank account objects. We might also create a class known as Customer.

4. An example, using BlueJ (software we will use later in lab; freely-available if you want a copy for your own computer: www.bluej.org)

   a) Note that the examples we will be working with today involve a fair amount of sophisticated Java features we won't get to until much later in the course.

   b) Even so, our example will be quite simplistic - we won't attempt to even come close to modeling a complete banking system. For now, we just want to touch on broad concepts concerning objects and classes

   c) Start up, open BankExample

      (1) Note two classes: Customer and BankAccount

      (2) Dashed lines indicate a mutual dependency: you can't have an account without a customer who owns it, and most customers have accounts.

d) Control-click on Customer. Note constructor (new) that can be used to create a new customer object. (We have one customer class, but we can have any number of customer objects.)

(1) Create a new customer. Note that we have to supply a name (enclosed in quotes). (A real system would require lots of other information, but for now this is enough.)

(2) Note that we have two names - the customer's name (why which he/she is known to other people) and an object name (by which the program refers to the object). Note that the object name can be pretty "weird" since it's not meaningful outside the program. The "human" name is enclosed in quotes because it is managed by the program but not actually meaningful to it.

(3) Create another customer. Note that we could create as many as we want. For now, two will be enough.

e) Control-click on each customer in turn.

(1) Each customer has two behaviors - both of which are the same for each customer.

(a) addAccount() - used to add a new BankAccount to the list of accounts owned by the customer

(b) printAccounts() - used to print out information about all of a customer's accounts.

Invoke this method for each object. Note that there are no accounts printed, but obviously each object "remembers" its customer's name.

(c) Note that the behaviors are properties of the individual customer object -not of the class - while the constructor is a property of the class.

(2) We can inspect the state of an object by using the Inspector. There are two components to the state of each customer:

(a) The customer's name - a character string

(b) A List of accounts the customer owns - stored using a structure we won't discuss until much later. (In fact, the form that's used here we don't discuss until CS211!)

f) We can also create bank account objects.

(1) Create a couple for Aardvark. Note that, in each case, the constructor for BankAccount needs a customer object to be

specified as the owner. (We use the "internal" name for the owner.) As part of the process of constructing the account, the constructor will add it to the list of accounts the customer owns.

(2) Demonstrate printing Aardvark's accounts now. Note that the constructor gives each account an initial balance of zero, and assigns each account a unique number, starting at 1.

(3) Demonstrate printing Zebra's accounts - note that each account is associated with a specific owner, so no accounts are printed for Zebra.

(4) Examine behaviors and state of a bank account object. Note how it's possible to go from the owner component of the state to the associated owner object by double-clicking.

(5) Now deposit some money to each account.

(6) Now inspect the state of each.

(7) Now try depositing more money to one of the accounts. Inspect the state.

(8) Now print out Aardvark's accounts.

(9) Now try withdrawing money. Note effect on state and printout.

(10) Now try creating an overdraft by withdrawing too much money.

What do you think will happen?

Do it - note exception.

Examine state - notice no change. (The withdraw operation was refused.)

III. Distribute, go over syllabi

IV. Show course web site.

Show how to access this lecture on line - note that, in general, lectures will not be on line until *after* they are completed in class (maybe several classes).