CPS122 Lecture: Course Intro; Introduction to Object-Orientation

last revised January 4, 2017

Objectives:

1. To introduce the course requirements and procedures.
2. To introduce fundamental concepts of OO: object, class

Materials:

1. "SMOP" projectable
2. Syllabus
3. Contrasting Python and Java versions of "Hello, world"
4. "Languages used in the CS curriculum" projectable
5. BlueJ demonstrations: SavingsAccount
6. On-line copy of this lecture for demonstration

I. Preliminaries - About this Course

A. This course is a continuation of CPS121, but it will differ from CPS121 in some important ways.

1. In CPS121, a great deal of your mental energy was invested in learning about *programming,* and more specifically in learning how to program in Python. Programming can be defined as the process of spelling out the steps needed to accomplish some task in a form that can be *interpreted* by a *computer system*.

2. While programming is part of the content of this course too, we want to step back from programming per se to set it in the broader context of software development, which is a form of *problem solving*.

   a) It turns out that much the same issues arise in any sort of problem solving - whether it be the creation of a piece of software, or the creation of some other engineered artifact, or

the design of a curriculum in education, or the development of a business plan, or ...

b) In this course, we are specifically concerned with *software development* - an entire process of which programming is just one part. Software development is the process of producing a software product that will fulfill some real need for someone.

c) However, since the basic process is similar whenever one is involved in designing a quality solution to a problem someone has - software or otherwise, the basic approach we will take is broadly applicable to any sort of problem solving.

3. Some of you may wind up developing software professionally, while others will not. However, an understanding of the issues involved in software development is not just important for practitioners. It is perhaps useful at this point to think a bit about the spectrum of things people with a CS background actually do.

   a) Some, of course, do develop software in a variety of settings - whether embedded systems (e.g. the software that controls many of the systems in a modern car) or moderate sized systems (like web sites) or huge systems (like those that control the telephone network etc.)

   b) Others are involved in research. Often, this research finds practical application in software systems.

   c) There is a growing recognition of the need to develop people with cross-disciplinary understanding - i.e. people who understand not only computing, but also some field where computing is important. In fact, whole new cross-disciplinary areas like this are emerging - e.g. bioinformatics.

d) Even a person who simply uses software - or helps others do so - can benefit from some understanding of what goes into producing software.

4. The guiding principle in the content of this course comes from the title of a talk I heard at the OOPSLA Educator's Symposium in some time ago: "Teaching Design: the rest is SMOP"

   PROJECT: SMOP and explanation

B. Distribute, go over syllabus.

C. As you may have noticed, the programming language used in this course is Java, rather than Python. So it may be appropriate to say a bit about why we are moving to a different programming language.

1. One thing that is <u>not</u> the reason is that some faculty like Python better and others like Java better! In fact the decision to use two different programming languages in these two courses was made by the whole department.

2. Rather, the shift in languages reflects a difference in size between the sort of problems CSP121 addressed and those we will be thinking about here.

   a) For smaller problems, a programming language like Python is very suitable. As we shall see, Java has a great deal more overhead which can be burdensome when one is trying to solve a small problem.

      Example: Project Python and Java versions of "Hello, world".

   b) On the other hand, for larger problems, a language like Java has three characteristics that make it easier to produce software that is correct, which more than makes up for the additional overhead.

Recall what we said about the various programming languages we teach at Gordon in CPS121.

PROJECT: Languages used in the CS curriculum

(1) Java is designed for an approach to software development known as object orientation. (Python uses object-orientation internally, and has support for object-orientation, but not as fully - and we did not make direct use of the OO features in CPS121.)

(2) Java is a compiled language. (Python is an interpreted language.)

(3) Java is a statically-typed language. (Python is a dynamically typed language.)

We'll say a more about the last two next class. For now, though, we want to introduce the basic idea of object orientation.

II. **Introduction to Object-Orientation.**

A. If a problem is small enough, it really doesn't matter a whole lot how one approaches. But larger problems need to be decomposed into subproblems in order to be accomplished reliably and in reasonable time.

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!

B. Object-orientation is one approach to decomposition that is widely used and often fruitful.

1. Object-orientation was first developed in the late 1970s.

2. It began to become very common in the 1990's, and by the 2000's had become the dominant paradigm in software development.

3. It is the approach we are going to take in this course, and, in fact, throughout Gordon's CS curriculum.

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

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

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

   b) **Behavior** - the object can do some things on behalf of other objects

   c) **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.")

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

   a) A state consisting of attributes like:

   (1) Account number

   (2) Owner

   (3) Current balance

   etc.

b) Behaviors like:

(1) Deposit money

(2) Withdraw money

(3)Calculate interest

(4)Report current balance

etc.

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

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

1. A class can be thought of as the set of all objects of the same kind - e.g. all the savings account objects together constitute the SavingsAccount 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.)

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

E. An example

1. For this example, we will use BlueJ (a very simple IDE for Java; freely-available if you want a copy for your own computer: www.bluej.org). (We will also use this in Lab 1)

2. 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.

3. 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

4. Start up, open SavingsExample

    a) Note two classes: Customer and SavingsAccount

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

5. 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.)

    a) 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.)

    b) 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.

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

6. Control-click on each customer in turn.

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

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

      (2) 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.

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

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

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

      (2) A List of accounts the customer owns - stored using a structure we won't discuss until much later.

7. We can also create savings account objects.

   a) Create a couple for Aardvark. Note that, in each case, the constructor for SavingsAccount 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.

b) 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.

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

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

e) Now deposit some money to each account.

f) Now inspect the state of each.

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

h) Now print out Aardvark's accounts.

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

j) Now try creating an overdraft by withdrawing too much money.
- Set up the parameter, but don't actually execute the method.

What do you think will happen?

Do it - note exception.

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

III. **Suggest students download a copy of Dr. Java for next class**

`http://drjava.org`

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).