CS12 Lecture: Defining Instantiable Classes

Last revised 1/28/03

Objectives:

1. To describe the process of defining an instantiable class
2. To discuss public and private visibility modifiers.

Materials:

1. Handout of code for BankAccount class showing various sections
2. Transparency of Wu figure 4.2
3. Transparency of Wu figure 4.3
4. Transparency of Wu figure 4.4
5. Online documentation for class java.awt.Color

I. Introduction

A. One of the most important questions we must ask when designing an OO system is “what classes do we need for this system?”

1. This is a question we will keep coming back to as our understanding of OO grows.

2. It turns out that identifying the right set of classes is a key step in OO design. Finding the right set of classes for a given problem is non-trivial, and requires considerable thought.

3. In a later course (CS211), we’ll talk a lot about the process of determining which classes will be needed. I did want to say just a little bit about this now, though.

B. In determining what classes are needed for a given program, we can begin by thinking about what objects must exist when the program is running. Each object will, of course, have to belong to some class - so if we can identify the objects, we are well on our way to identifying the classes.

C. One helpful approach we can take is to make use of the notion of responsibility driven design - every thing that the program does must be the responsibility of one or more objects belonging to some class in the program.

Example: In lab 2, we were led to define a class “CarpenterRobot” because the problem’s requirements included needing an object responsible for laying carpet.
D. It turns out that objects in a program often fall into one of three broad categories:

1. Entity objects represent the basic entities the program deals with - e.g. robots.

2. Controller objects are responsible for directing the activities of other objects - e.g. our robot programs.

3. Boundary objects represent the interface between the system and the outside world. The class World which we used in our robot programs provided access to an object that visibly displayed the world our robots operated within.

In general, we will need to use/develop classes of all three kinds to solve a particular problem.

II. Defining an Instantiable Class

A. As we discover the responsibilities that need to be fulfilled by a given system and assign them to objects, we will need to determine what class each object must belong to.

1. When possible, we will want to make use of existing classes that already have methods for carrying out the necessary responsibilities.

2. But, in any non-trivial system, there will be a need to define one or more new classes of objects to fulfill responsibilities that are distinctive to that system - either from scratch, or by extending an existing class.

B. Once we have determined that a new class needs to be defined to fulfill a given responsibility or set of responsibilities, we need to design the class. To do this, we can consider a series of basic questions:

1. What is the purpose of this class? Can we summarize the responsibilities the class and/or its objects must fulfill in a single, short sentence?

   EXAMPLE: We earlier discussed an example using classes that represent bank accounts. We might say something like the following:

   “The purpose of the BankAccount class is to model individual bank accounts”.


2. What do objects belonging to this class need to know (about themselves?)

   EXAMPLE: BankAccount objects need to know their number and their current balance.

3. Whom do objects belonging to this class need to know (what other objects are they associated with - what objects to they need to use to help them do their job?)

   EXAMPLE: BankAccount objects need to know their owner.

4. What do objects belonging to this class need to be able to do?

   EXAMPLE: BankAccount objects need to be able to deposit money, withdraw money, etc.

Instance methods are of two general kinds:

   a) Mutators that change the state of the object (e.g. deposit, withdraw). Typically - but not always - mutators return void.

   b) Accessors that furnish information about the state of the object without changing it (e.g. reportBalance, getAccountNumber). Accessors always return some non-void result.

5. Also to be considered is whether there is any knowledge or behavior that belongs to the class as a whole, rather than to individual objects.

   EXAMPLE: We might require our BankAccount class to maintain a master list of all open accounts. We might have a behavior payInterest() that calculates interest for each open account. These are properties/behaviors of the class, not of each individual account.

C. Having considered these questions, we are ready to define the class.

HANDOUT Annotated BankAccount class example

1. Each file should begin with a file comment - a comment that describes the file and specifies its author and date.

   NOTE in handout

2. We begin each class with a class comment - a comment (in English) that describes the purpose of the class. In Java, the preferred way to do this is
with a comment that goes immediately before the class, and begins with
/** - e.g.

NOTE in handout

3. Give the class a name. The name should be clearly related to the purpose of the class.

4. Determine whether the class should stand alone or be an extension of an existing class.

5. Identify the instance variables of the class

   a) Recall that each object that is an instance of a given class has its own copy of each instance variable. These variables exist as long as the object exists.

   TRANSPARENCY - Wu figure 4.2

   b) In general, each piece of information that an object needs to know will be represented by an an instance variable.

   NOTE instance variables accountNumber and currentBalance in handout. Note use of a comment to explain how currentBalance is represented

   c) Further, each object will also need an instance variable to refer to each kind of other object that it knows. (I.e. the instance variables hold answers to two questions: what does this object know? and who does this object know?)

   There are two possibilities for these instance variables

   (1) The instance variable may simply be a reference to some other object.

       NOTE in handout: Instance variable owner

   (2) Or, the instance variable may be a special kind of object called a collection which holds references to several other objects of the same kind.

       EXAMPLE: Since bank accounts can have multiple owners (joint accounts), instead of an instance variable called owner, we might have one called the owners that is a collection. (Collections are a
topic we will discuss later this semester - so far now we don’t know how to do this in Java, so we’ll stick with a single owner for this example.)

d) Conceptually, we decide on the instance variables for a class early in the process. In this example, and in the examples in Wu, they are placed at the beginning of the class. However, a case can also be made for listing the instance variables near the end of the class declaration. Either place is acceptable (but not strung out everywhere!)

6. Identify the instance methods (mutators and accessors)

a) In general, each thing an object needs to be able to do will be represented by an instance method.

   NOTE in handout

b) Because defining methods is so important, we’ll come back to a much more detailed coverage of this task later in this lecture.

7. Determine whether any class variables, constants, or methods are needed.

a) A variable that holds a value that is shared by all the objects in the class can be declared as a class variable.

   NOTE in handout: two class variables.

   One is used to ensure that each account gets a unique number. The constructor uses this to assign a number to a newly-created account, and then adds one to it. Obviously, this must be shared by all instances of the class.

   Another is used to hold a master list of open accounts.

b) A constant value that is the same for all objects in the class, and that either methods or clients of the class need to make use of can be declared as a class constant.

   NOTE in handout: many banks have a minimum balance allowed in account for paying interest. (An account with a balance below this amount is not eligible for interest.) Since this rule applies to all accounts, a class constant MINIMUM_BALANCE_FOR_INTEREST is used. (Note naming convention)
c) A method that represents a responsibility of the entire class, not associated with any particular object in the class, can be declared as a class method.

NOTE in handout: payInterest() method.

d) Note that instance methods can freely make use of class constants, variables, and methods; but that class methods cannot use instance variables or methods since they are not associated with any particular instance of the class.

e) Class variables, constants, and methods are distinguished from instance variables and methods by being declared static.

III. Defining Methods Within A Class

A. When we are defining a class, most of our time will be spent defining its methods.

B. In general, a method definition consists of a method header and a method body. In addition, a method should have a method header comment that describes what it does and how to use it.

1. Recall that a method is invoked when some object sends a message to our object. The method header specifies what form that message must take. It consists of optional modifiers, a return type, the method name, and a (possibly empty) parenthesized list of formal parameters.

   a) The possible modifiers are

      (1) The visibility modifiers (public, protected, private) - discussed below. These control who may make use of this method.

      (2) The word static, to indicate that the method is a class method. (Otherwise, it is taken to be an instance method). If the method is an instance method, the message is sent to a particular object; if it is a class method, it is sent to the class as a whole.

      (3) The world final, to indicate that the method may not be overridden in any subclass.

   b) The return type is either the name of a primitive type, or the name of an object type, or the word “void”. It specifies what information, if any, is returned to the sender of the message when the method has completed its work.
c) The method name should clearly and simply state what the method does. Good method names have the property that the method does exactly what its name says it does - no more and no less. Good method names generally consist of an imperative verb, perhaps with an object - e.g.

    deposit()
    reportBalance()

d) The formal parameters specify the information the sender of the message that invokes the method must supply.

   (1) In the simplest case, there are no parameters, and the parameter list consists of an empty pair of parentheses ( ).

   (2) In many cases, the parameter list will consist of one or more items of information. Each element in the parameter list has two components: a type and a name.

     (a) The type specifies what kind of information will be supplied - e.g. an integer value (int) or a reference to an object or what have you. Any type that is legal for a variable is legal for a parameter - in fact, within the method the parameters behave like variables.

     (b) The name indicates what the parameter is used for - i.e. what it means. Again, the name should be chosen carefully so that anyone using the method can know clearly what information is expected.

        EXAMPLE: The deposit method of class BankAccount needs a numeric parameter to specify the amount of the deposit

   (3) Note an important distinction in function between the parameters and the return type:

     (a) Parameters represented information flowing in to the method from the caller.

     (b) The return type represents information flowing out of the method back to the caller.

        EXAMPLE: A BankAccount method named reportBalance() would return a number
(c) In Java, it is possible to have any number of items flowing in to a method, but only one item coming back. While other programming languages provide a way for some of the parameters to represent information flowing back out of the method, this is not the case in Java. (However, if one of the parameters is a reference to an object, the method can potentially alter the object that is referred to.)

2. The method body contains the code that is executed when the method is called. It may contain a mixture of variable declarations and executable statements.

   a) When a variable is declared in a method, it is called a local variable. A local variable comes into existence when its declaration is encountered during the execution of the method, and ceases to exist when the method is finished. This is distinguished from the instance variables of an object, which exist the entire time that the object exists.

   Example: TRANSPARENCY - Wu figure 4.3

   b) Actually, a formal parameter can be thought of as a special kind of local variable - one that is declared in the method header and initialized with a value when the method is called.

   TRANSPARENCY - Wu figure 4.4

   c) We have seen a number of different kinds of executable statements that can occur within a method body - e.g.

   (1) Assignment statements

   (2) if statements

   (3) loop (for, while) statements

   (4) return statements. A return statement serves two purposes:

      (a) It always immediately ends the execution of the method - even if it occurs in the middle.

      (b) If the method is defined as returning a value, the return statement specifies what value is to be returned.

3. A method comment should go before each method. This comment should include:
a) A brief, clear description of the what the method does

b) An explanation of each parameter (what its role is)

c) An explanation of the return value - if any

The handout shows the standard form for method comments

NOTE on handout - method comments for deposit(), withdraw(), reportBalance() - @param tags, @exception tag, @return tag

d) When a method is used, the caller must specify:

4. The object or class to which it is being applied (depending on whether it is an instance or class method)

5. The name of the method

6. A list of actual parameters. These must correspond to the formal parameters in the method header in *number, order, and type*.

   EXAMPLE: The constructors we used for our robots required four integer parameters: the starting street, the starting avenue, its initial direction, and its initial number of beepers. When we used new to construct a new robot, we had to specify these values in exactly that order.

C. Instance methods fall into one of several broad categories:

1. A *constructor* method is used when creating an object.

   a) Wu calls the constructor a class method, because it is used when the “new” message is sent to the class. However, it partakes of some of the nature of an instance method because, at the time it is called, the new object has already been brought into existence, and hence its instance variables and methods are available to the constructor. (True class methods do not have access to any instance variables and methods of any particular object).

   b) The name of a constructor is always exactly the same as the name of the class - that is how the compiler recognizes a method as a constructor.
c) The purpose of the constructor is to put the object into a consistent initial state. When a new object is created by sending a “new” message to a class, the class always calls the constructor of the newly created object.

d) The parameters of a constructor specify things which the object must know from the very outset of its existence.

EXAMPLE: When we defined a new class of robot, we always defined a constructor method whose parameters specified its initial location, orientation, and number of beepers.

e) Note that while a constructor may have parameters, it never has a return value. Also, a constructor is never called directly - it is always called as a result of sending the new message to the class, requesting that a new object be created.

f) It is possible to define more than one constructor for a class, taking a different number or types of parameters. The names of the constructors are always the same, but the compiler can distinguish between them based on the types of their parameters.

EXAMPLE: If we were frequently in the practice of starting robots off at 1st st and 1st ave facing East, we could define a constructor for our robot subclasses that took only one parameter - the number of beepers - in addition to the constructors we defined.

NOTE: In distinguishing between two or more methods having the same name, the compiler considers only the number and types of the parameters - not their names.

g) If no constructor is defined for a given class, the compiler automatically creates a default constructor that takes no parameters. However, if any constructor is defined for the class, no default constructor is automatically created - if one is needed, you must write it yourself.

2. An accessor method is used to obtain information from the object, without altering the object itself.

a) Accessors may or may not have parameters going in, but always have a return value.

b) Accessors never alter any instance variables of the object they are applied to. (In some programming languages, there is a way to enforce this - but not in Java)
EXAMPLE: What were some of the accessor methods for our robots?

ASK

boolean nextToABeeper()
boolean nextToARobot()
boolean frontIsClear()
boolean facingNorth()

etc.

We could also imagine a more sophisticated robot class that might have accessors like

int numberOfBeepersInBeeperBag()
int currentStreet()

etc.

c) As was the case with constructors, it is possible to have two accessors with the same name, but different parameter types. This, however, is not common.

3. A mutator method is used to tell the object to alter itself in some way.

a) Mutators frequently have parameters going in, but usually do not have return values. (If a mutator does have a return value, it is usually some boolean that indicates whether or not the operation was successful.)

EXAMPLE: What were some mutators of our robot classes?

ASK

turnLeft()
move()
pickBeeper()

etc.

As was the case with constructors, it is possible to have two mutators with the same name, but different parameter types. For example, we might have had two move() methods in one of our robot classes - one with no parameters that moves the robot one block, and one with an int parameter that specifies the number of blocks to move, so that we might be able to say something like:

karel.move(3);
IV. Visibility of Class Members

A. Each class, and each member in a class (instance or class variable, instance or class method) has associated with it a visibility that controls whether it can be accessed by the methods of other classes.

B. Java provides four levels of visibility - listed in order from greatest access allowed to least access allowed

1. public - any method of any class can access this item
2. protected - we will discuss this later in the course
3. default - any method of any other class in the same package can access this item
4. private - only methods of this class can access this item

All four of these may be applied to class members. Ordinary classes themselves can only have public or default visibility - the other two would not be meaningful for ordinary classes and therefore are not allowed. (There is a case in which private or protected can be applied to certain kinds of classes, but that’s an advanced feature of the language we are not likely to get to in this course.)

C. The visibility of a class or member is specified by preceding its declaration with a keyword:

1. The keywords public, protected, and private specify the corresponding level of visibility.
2. The omission of a visibility keyword produces default visibility.

D. In general, the following guidelines are helpful in deciding what visibility to assign to each class member:

1. Instance variables should almost always be private. (Maybe the correct word is just always!).

   a) If it is necessary for other classes to have access to an instance variable, this can be provided by defining appropriate accessor and or mutator methods. Such methods typically have names like getWhatever() or setWhatever(newValue) - and for this reason are sometimes called “getter” and “setter” methods.
b) This represents a key notion called *encapsulation* or *information hiding* - a class should encapsulate (or hide) the details of its implementation to help make a system modular and more easily modifiable.

*EXAMPLE:* If we had a class representing bank accounts that had a publicly-accessible balance field, we could do deposits and withdrawals by simply modifying this field directly. By making it private - and thus forcing other classes to use mutator methods like deposit() and withdraw() - we make it possible to enforce rules like the rule that a balance cannot go below 0 - or to implement features like overdraft protection if it does.

2. Instance methods are frequently public, since they provide the behaviors that the class is responsible for providing. The exception would be instance methods that serve as “helpers” to other methods and are not used directly.

3. Class variables and methods follow similar rules to those given for instance variables and methods.

4. Class constants are often made public. No encapsulation issue arises, since they cannot be modified. However, if the constant is used only in the implementation of the methods of the class, it should be private.

5. The other two levels of protection for class members will be discussed later.

E. The following guidelines are helpful in deciding what visibility to apply to a class.

1. The main class in an application, or the Applet subclass in an applet, should be public. (It is, after all, the entry point into the whole system.)

2. Classes that provide service to lots of other classes should be public.

3. A class that serves to provide service only to one or a few other classes might have default protection if it resides in the same package as the class(es) it helps.

One important note: Java requires that a public class be defined in a file whose name is the same as that of the class. That, of course, implies that only one public class can be defined in any one file.
V. Summary

Go over online documentation for class java.awt.Color, noting class constants; identifying methods as constructors, accessors, and mutators and discussing parameters.

Note that this documentation was generated automatically (by a program called javadoc) from the source code for the class. The special class and method comments we have discussed (beginning with /**)), plus the special tags @param and @return, are recognized by javadoc - hence, this style of comment is called a “javadoc comment”.