CS112 Lecture: Defining Classes

Last revised 2/12/08

Objectives:

1. To describe the process of defining an instantiable class

Materials:

1. BlueJ SavingsAccount example project
2. Handout of code for a `SavingsAccount` class showing various members
3. Projectable version of `SavingsAccount` with overloaded constructor.
4. Projectable version of UML Diagrams
5. Online documentation for 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 3, we were led to define a class "CarpeterRobot" 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. Example:

   • The various robot objects we created early in the course
   • We haven't yet seen any clear examples of this in the Bruce text
   • In the BankAccount example we have referred to from time to time in lecture, the various `BankAccount` and `Customer` objects fulfill this role.

   PROJECT BankAccount example

2. Controller objects are responsible for directing the activities of other objects - Examples:

   • Our robot programs
   • The `WindowController` extension object in lab and in the examples we have done in class.
   • There is no example of this in the BankAccount example - it is not really a complete program!

3. Boundary objects represent the interface between the system and the outside world. Examples:

   • The class World which we used in our robot programs provided access to an object that visibly displayed the world our robots operated within.
   • The various drawn shapes (e.g. `FramedRects`) we have used in lab and in the examples we have done in class.
   • Again, there is no clear example of this in the BankAccount example

4. For simple problems, we may find that one class serves more than one role

   a) In `objectdraw`, a `WindowController` extension is both a controller and a boundary object, since it also represents the displayed window.

   b) It could be argued that the various geometric object classes in `objectdraw` (`FramedRect`, etc.) play the roles both of entity objects and boundary objects.

   c) In general, though, we will need to use/develop classes of all three kinds to solve a complete problem.

II. Defining a 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.

a) Whereever possible, we should try to extend an existing class, rather than creating a class from scratch

However, in order to do this the classes should obey the "isa" rule: if some new class "Foo" is going to extend some existing class "Bar", then we have to be able to meaningfully say "A Foo is a Bar") - which, in turn, means that wherever a "Bar" is needed a "Foo" can be used.

Examples:

It is proper to have a class called Student extend a class called Person, because it is meaningful to say "A Student is a Person" - i.e. anywhere a Person object is needed, a Student object can be used.

It is not proper to have a class called Arm extend a class called Person, because it is <u>not</u> meaningful to say "An Arm is a Person" (though it may be a <u>part</u> of a Person) - i.e. it is not true that anywhere a Person object is needed, an Arm object can be used. (Try asking an Arm object for its Social Security Number, for example - a meaningful behavior for a Person, but not for an Arm!)

b) Whether we are extending an existing class or creating a new class from scratch, we follow essentially the same process; but in the case of extending an existing class, we <u>inherit</u> features belonging to the base class.

Example: If we extend a class Person to create a class Student, we do <u>not</u> need to define features like "name", "Social Security Number", etc. - those features are inherited from Person. We do need to define features that are unique to Students - e.g. major, gpa.

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* and *whom* do objects belonging to this class need to *know*? (The answer to this question specifies what needs to be part of the state of an object that belongs to the class.)

   a) What do objects belonging to this class need to *know* (about themselves?)

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

   b) 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.

3. *What* do objects belonging to this class need to be able to *do*? (The answer to this question specifies the behaviors of an object that belongs to the class.)

   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.

4. 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: Suppose we were interested in defining a class for interest-bearing savings accounts. In this case, we would also need an "interest rate" property. However, this is likely a property shared by all objects of the class - i.e. if we change the interest rate, we change it for every account.

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

HANDOUT Annoted `SavingsAccount` class example

1. Each file should begin with a *file/class comment* - a comment (in English) that describes the purpose of the class and specifies its author and date.

   NOTE in handout

   Generally each class is in its own file. The name of the file will be the same as the name of the class - e.g. the example would appear in a file called `SavingsAccount.java`. (This is a requirement enforced by the java compiler).

   There is the possibility of defining a class inside another class - but we won't discuss that now..

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

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

4. 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.

   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.

      Insofar as possible, the name of an instance variable should make its purpose crystal clear. In some cases, a supplementary comment may be desirable to add information not obvious from the name. (e.g. the comment on `currentBalance` that notes how it 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) Instance variables always have a visibility specifier, which will be one of `public, protected, private`.

(1) We will discuss these later in the course - but for now you should simply use `private`.

(2) This relates to a key concept in OO called encapsulation - each object encapsulates within itself certain information that can only be accessed through its methods, and hence is not subject to surprise changes.

e) Conceptually, we decide on the instance variables for a class early in the process. In this example, 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!)

5. 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 more detailed coverage of this task later in this lecture.

6. 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 the interest rate

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. (Constants use a special naming convention as here)

   (One might argue that this should also be a variable, so that the bank can change its rules. I made it a constant here so that I could illustrate both possibilities!)

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: `setInterestRate()` method.

d) 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`.

f) The `main()` method of a program is always a class method ("`public static void main ...`"). Why?

   ASK

   No objects exist at the time the `main()` method of a program is called, so there would be no way to call an instance method. You can think of this method as being sort of a "special case", since it really doesn't fit the normal meaning of a class method, either.

D. Note the use of comments throughout the class - a file/class prologue, method prologues, and prologues for <u>public</u> variables/constants.

1. In Java, the preferred way to do this is with a comment that goes immediately before the entity, begins with `/**`, and includes <u>tags</u> that begin with "`@`". (The permitted tags depend on what type of comment you are writing - e.g. for a class comment tags like "`@author`" occur; for a method comment tags like "`@param`" occur.)

2. This style of comment is called a "javadoc" comment - and gets its name from a program (called javadoc) which automatically creates a set of documentation from code containing this style of comment.

   DEMO - "Project documentation" in "Tools" menu. Do for Bank Account example and show result.

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 prologue 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`) - to be discussed later in the course. These control who may make use of this method. It is also possible to omit this, in which case you get what is called "default" visibility that may create problems!

      For now, the methods you create should generally be `public`.

      (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.

(a) An instance method is called by *objectname.methodname*

Example: `karel.turnLeft()`

(b) A class method is called by *classname.methodname*

Example: `SavingsAccount.setInterestRate(0.05)`

(3) The world `final`, to indicate that the method may not be overridden in any subclass. (We won't see examples of this until much later in the course.)

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. Accessors always have non-void return types; mutators are generally void.

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 `SavingsAccount` needs a numeric parameter to specify the amount of the deposit

(3) There is 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: The `SavingsAccount` method named `reportBalance()` returns a neatly formatted String representation of the balance.

(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.)

(4) In any class, it is possible to have more than one method having the same name, provided that their formal parameters differ.

Example: Suppose we wanted to define a move() method for a robot that allowed us to specify how far to move - e.g.

```
karel.move(3);
```

would move Karel 3 blocks.

This would require an additional definition for `move()` in the Robot class - which could be distinguished from the original definition because it requires one parameter of type int.

(a) When two methods have the same name, we say that the name is <u>overloaded.</u>

(b) Any type of method (constructor, instance mutator, instance accessor, class mutator, class accessor) can be overridden

(c) If a class has overloaded methods, the methods must have different signatures.

i) The signature of a method includes its name and parameter types.

Example: The signature of deposit is `deposit(int)`

ii) The signature of a method does not include its visibility, return type, or the name of its parameters.

(d) If a class has overloaded methods, it is good practice to make one of them the primary method, and define the others in terms of it.

i) Example: overloaded `move()` method in various objectdraw classes, as discussed in the book.

ii) Why?

ASK

Doing it this way guarantees that any changes are made consistently. With duplicated code, there is always the possibility of changing one variant and forgetting to change another.

(e) Overloading is different from overriding.

i) An overloaded method has a different signature from another method of the same name in the same class.

ii) An overridden method has the same signature as a method in a base class.

Example: in the `SavingsAccount` example, we might have a subclass called `CertificateOfDeposit`.

- If `SavingsAccount` had a method with signature `deposit(double)`, that would overload the existing `deposit(int)` method.

- If `CertificateOfDeposit` had a method with signature `deposit(int)`, that would override the `deposit(int)` method of `.SavingsAccount.`

2. A method is called by invoking it from elsewhere on an object of the class (e.g. `karel.move()`) If a method has formal parameters, then when it is used actual parameters must be specified that agree with the formals in order and type.

   Example: since `deposit(int)` requires a single integer parameter, it is legal to say something like

   ```
   someAccount.deposit(500);
   ```

   but not any of the following:

   ```
   someAccount.deposit();          // No parameter
   someAccount.deposit(500, 3);    // Too many parameters
   someAccount.deposit("five");    // Wrong type parameter
   ```

3. 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: Local variable `cents` in the `reportBalance()` method

   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.

   c) What happens if a local variable (or parameter(has the same name as an instance variable?

      (1) By default, the name is taken as referring to the local variable

      (2) To refer to the instance variable, the name must be qualified by `this`. Inside any constructor or instance method `this`, refers to the object to which the method was applied .

      Example: the constructor for `SavingsAccount`.

      It is necessary to say `this.owner = owner;` because we want the left hand use of to be refer to the instance variable named `owner` and the right hand use to refer to the parameter `owner`.

The instance variable `currentBalance` could have been written `this.currentBalance`; but this was not necessary because there was no possibility of confusion of meaning. (Legal but not needed)

d) 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.

4. A method prologue comment should go before each method. This comment should include:

a) A brief, clear description of the what the method does. Note the distinction between saying what and saying how. The comment should focus on the "what" - the "how" is learned by reading the code of the method.

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:

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

(2) The name of the method

(3) 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. We have already noted that instance methods fall into one of several broad categories:

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

   a) 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.

   b) 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.

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

      EXAMPLE: The `SavingsAccount()` constructor.

   d) 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.

   e) It is possible to have overloaded constructors. In this case, one constructor can invoke another by using `this()`.

      Example: We might imagine an overloaded constructor for `SavingsAccount` that allows one to specify both the owner and an initial balance. In that case, we might have:

      PROJECT

```
/**
 * Constructor for objects of class SavingsAccount
 *
 * @param owner the owner of this account
 * @param initialBalance the initial balance
 * The account number will be set to the first available
 * unused number
 */
public SavingsAccount(Customer owner, int initialBalance)
{
    this.owner = owner;
    owner.addAccount(this);
    currentBalance = initialBalance;
    accountNumber = nextAccountNumber ++;
}

/**
 * Constructor for objects of class SavingsAccount
 *
 * @param owner the owner of this account
 * The balance will be set to zero
 * The account number will be set to the first available
 * unused number
 */
public SavingsAccount(Customer owner)
{
    this(owner, 0);
}
```

f) 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)

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

   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.)

D. As we have already noted, it is also possible to have class mutators and accessors (but not constructors!)

IV. Variables and Constants

A. We have now seen four different kinds of variables/constants.

1. What are they?
   ASK

   a) Instance variables

   b) Class variables

   c) Parameters

   d) Local variables

2. How do they differ?
   ASK

   a) Each object belonging to a class has its own copy of each instance variable, that exists as long as the object exists.

      Example: any one of you can answer a question like "what is your name?" (Each of you has an "instance variable" called name)

   b) All objects belonging to a class share a single copy of each class variable.

      Example: all of you would give the same answer to the question "how many credits does it take to graduate?"

   c) Parameters and local variables exist only while a particular method is being executed.

      Example: If I were to ask you "what are you eating?" while you are sitting in Lane, you could give me an answer. But it would be silly for me to ask you that question while you are asleep, and probably offensive for me to ask you that question now!

3. Where are they declared?
   ASK

   a) Instance and class variables are declared outside any method - often at the start or end of a class.

   b) Parameters are declared in a method heading.

   c) Local variables are declared inside the body of a method.

B. Any variable can be given a value by an initializer when it is declared.

1. Example: `nextAccountNumber` in the SavingsAccount example

2. If an instance variable does not have an initializer, then it is initialized to a default value (e.g. 0 for numbers).

3. If a local variable does not have an initializer, it cannot be used until it is given a value by an assignment statement.

C. A constant is a special kind of variable that is given a value that cannot be changed.

1. Constants are declared using the word `final`.

2. Constants, of necessity, must be given a value by an initializer.

3. Constants are declared as class constants (`static`), since it makes little sense to think of different instances of a class as each having their own value of some constant!.

4. Constants typically are given names that consist of all uppercase letters, with underscores used to separate words (since mixed case is not used.)

   Example: `MINIMUM_AMOUNT_FOR_INTEREST`.

D. It is also possible to have instance variables that are given an initial value, and then cannot be changed after that.

1. They are also declared using the word `final`.

2. Though they can be given a value by an initializer, they are typically given a value by a constructor.

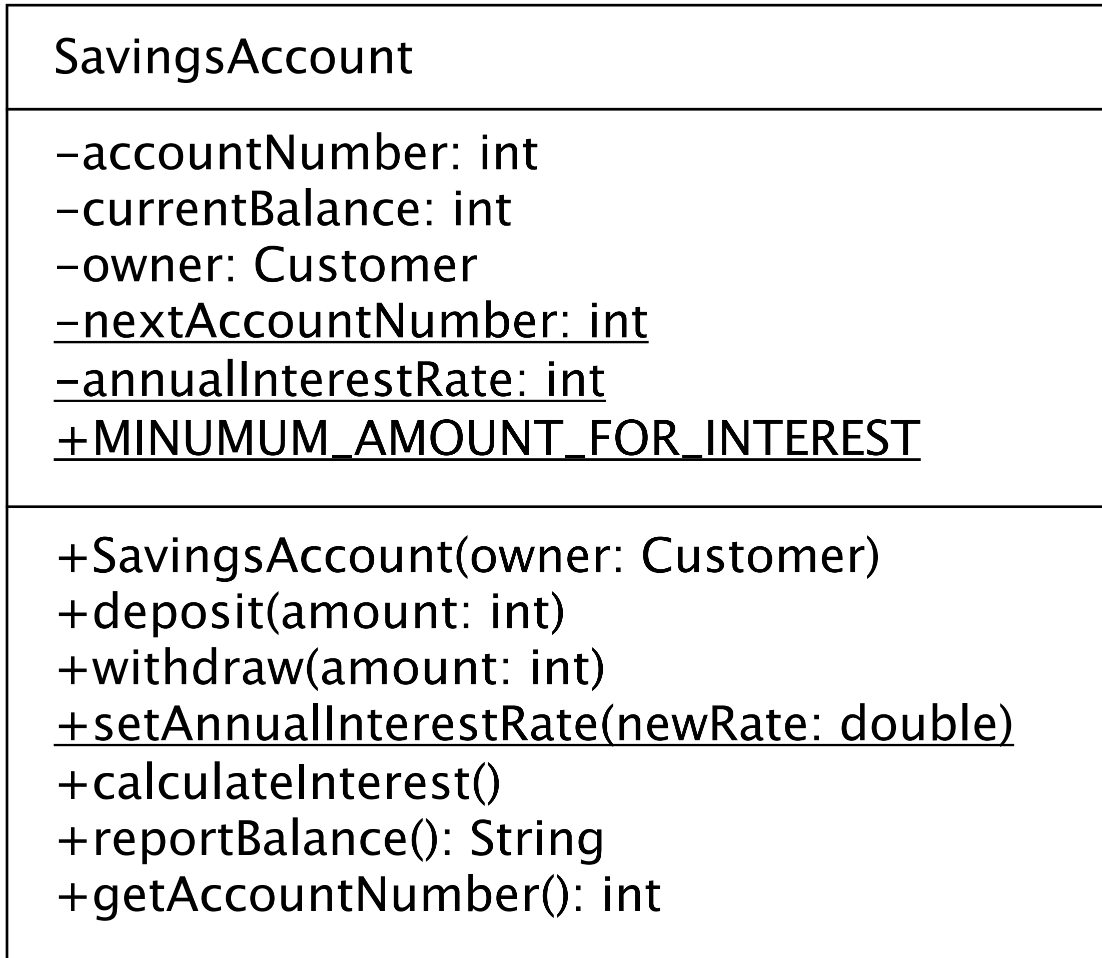
3. They have a name that looks like any other variable name.

   Example: `accountNumber` - once given a value by the constructor, it cannot be changed.

V. UML Diagrams for Classes

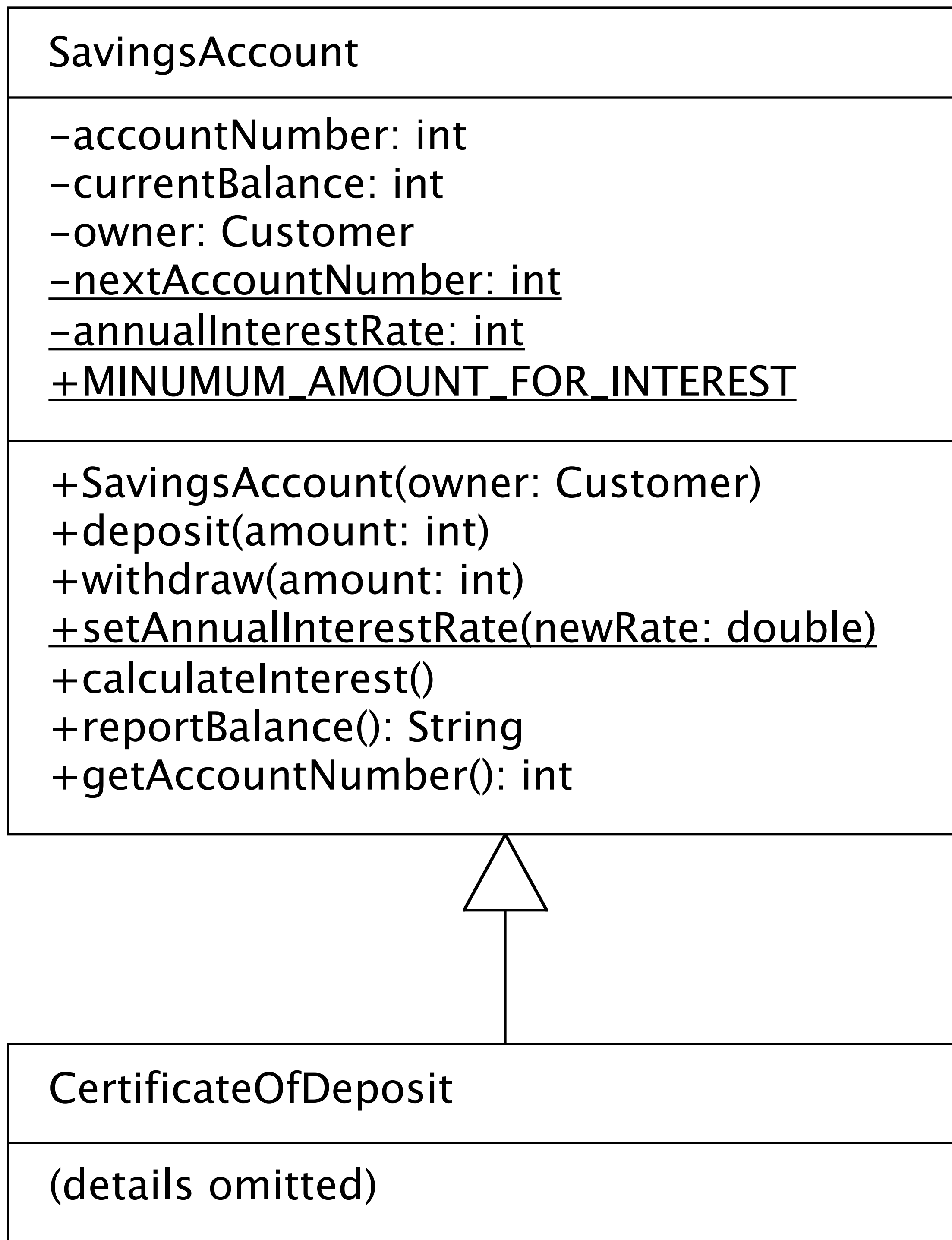
A. The design of a class can be depicted by a UML class diagram. For example, the following diagram shows our `SavingsAccount` class:

PROJECT



B. UML also allows one to depict the "extends" relationship between classes. For example, suppose we extended `SavingsAccount` to create a `CertificateOfDeposit class`. That could be depicted as follows (details of internal structure of new class omitted)

PROJECT



VI. A Final Example

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