

# Chapter 3 Class Design and Implementation

## Introduction

This chapter illustrates the process of class design, coding and testing. It starts with a description of a simple ATM simulator and concludes with a working applet that implements it. The program is grown in stages, starting with a very simple prototype and adding features only after each prototype works. The resulting `Account` class will be incorporated into a larger bank database system in Chapter 11.

## A Description of the Task

Imagine you are given this description for a programming assignment: Write a minimal ATM program that will manage three bank accounts. Each account will have a name and a balance. Allow users to display their current balances and withdraw as much (simulated) money as they want with a graphical user interface (*GUI*).

## Before Beginning to Program: Design!

There is a strong impulse to start to program too soon. When given a problem description, some beginning programmers start typing before they have a clear idea of what it is they are doing. In a way this is unavoidable, since a novice programmer knows practically nothing about the programming language. Nevertheless, it is possible to have a clear idea of *what* one is attempting to accomplish even if the *how* is a bit unclear.

It is human nature to experience confusion as pain. Experienced programmers have learned not to start typing before they have a clear understanding of what it is they are attempting. Painful, frustrating experiences have taught them to think through problems before committing to code. Therefore, between the time they read or formulate a description of a programming task, and when they sit down to type code, they do what is called *design*. Design can take many forms, but it always features clear, simple thinking. A coherent design supports the programming process, and can make the difference between success and failure.

One design technique is to start with the user interface; decide what the user will see and what actions the user can take; then design classes that support those actions.

## The User Interface

Start by drawing a picture of what the user will see when the program runs. For simplicity, start with a single bank account. To design a good user interface you must consider what information will flow into and out of your program, and what actions the user will take to interact with it. What are those actions and information in this case?

If you reread the description (do that!) you will see that the user can do just two things: ask for the current balance, and withdraw funds. Actions: For a withdrawal the user must specify how much money they want (this must provide that information). To display their balance they only

need indicate that they wish to see it, so a button press will be sufficient. Information: For a withdrawal the amount the user requested must be input to the program, and when they ask to see their balance, that information must be output.

Knowing that, what sort of user interface does your ATM need? At minimum, it will need: 1) a button to request the current balance; and, 2) somewhere to type the amount to withdraw. The Java components for these two screen objects are `Button` and `TextField`. You are probably familiar with both of these components (even if you don't know their names), they appear on many Web pages. When you fill out a form on the Web, the boxes you type in are typically `TextFields`, and the buttons, `Buttons`. Grab a piece of paper and make a quick sketch of what your user will see when your simulated ATM runs. Note that you can do this without any knowledge of how the program will work.

### **What Classes Will We Need? What Will They Do?**

A principle of object design is to create classes that correspond to the things that the program is modeling, and to store information in the objects corresponding to where it resides in the world. A real ATM machine communicates with a bank; user information is kept in accounts. So, it is reasonable to think of having an `Account` class, and to store information about each person's account in a separate `Account` object. Note the use of the `Courier` typeface and the capital letters at the beginning of class names; this usage is to both remind you that class names begin with capitals and also help make clear when words refer to classes (as opposed to objects in the world).

So, to simulate an ATM associated with a bank that has three customers, we will need three `Accounts` (one to keep track of each customer's name and balance), a `Bank` (to contain the three `Accounts`), and an `Applet` (to handle the GUI). That would be a lot of code to write all at once, so those three tasks will be attempted one at a time and then combined. This is an example of the following:

### **Problem Solving Technique: *Stepwise refinement***

*First, understand the problem, then break it into 5+/- 2 subproblems. For each: if it is trivial, just do it; if it is not trivial, solve it by using stepwise refinement*

The first step is the most important -- until you clearly understand a problem, any attempt to solve it is very unlikely to succeed. Sometimes people attempt to solve problems before understanding them; then mostly they fail and are frustrated.

Notice that this is a *recursive definition*, the thing being defined is used as part of its own definition. Circular definitions are bad. Recursive definitions can be very good, so long as they do not recurse forever. Since the subproblems are smaller than the original, as this recurses, eventually they are all so small as to be trivial and the recursion stops.

## Building and Testing the Prototype GUI

Figure 3.1 is a rough sketch of a possible GUI for an ATM that keeps track of the balance of a single bank account. There is a `Button` with the label “Display” and a `TextField` with the text “Amount to withdraw” above it, and “Current balance” below it. The `TextField` will be used to both enter the amount to withdraw and display the current balance.

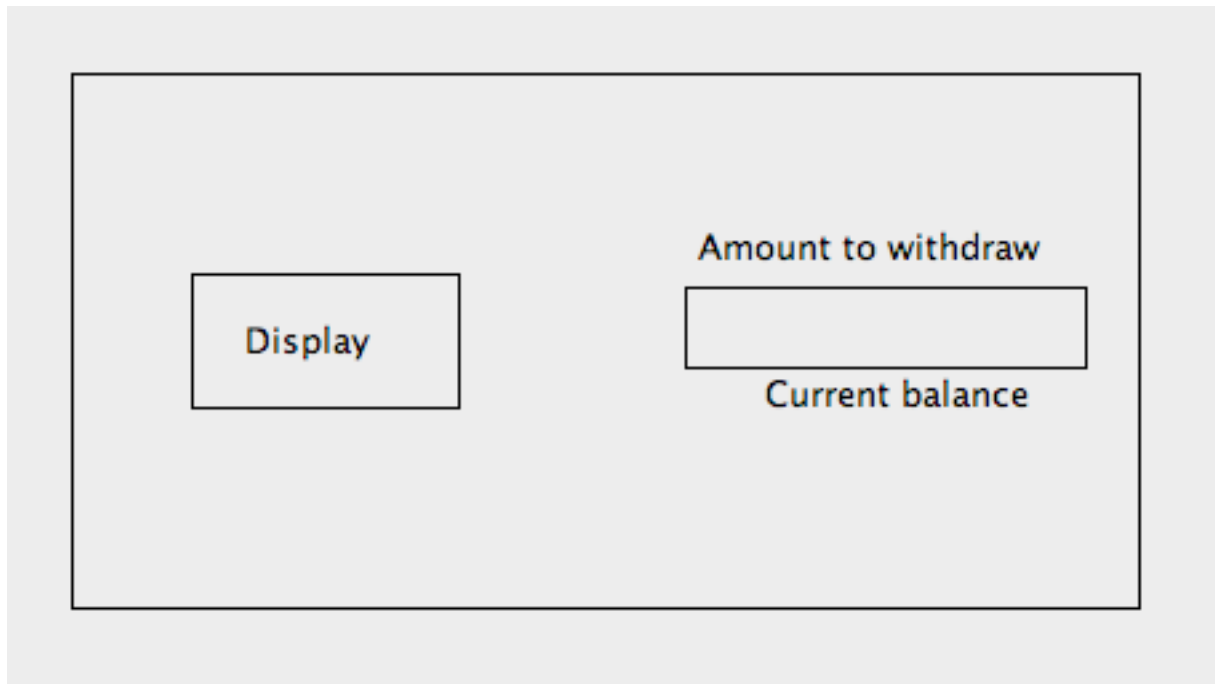


Figure 3.1 Rough sketch of GUI for the ATM.

The next sections will explain how to build and test a GUI with those two components using Netbeans.

### Getting Started

1. Start Netbeans and create a new project with a GUI `Applet` (see the Netbeans Appendix, “Creating a GUI `Applet`” for details).
2. Add a `Button` with an action and test it (see the Netbeans Appendix, “Adding, connecting and testing a `Button`”).
3. Add a `TextField` to type the withdrawal amount in and to display the balance (see the Netbeans Appendix, “Adding, connecting and testing a `TextField`”).

### Using the `Button` to Alter the `TextField`

When the user presses the `Button`, its `actionPerformed()` method will be executed. Netbeans writes the shell of the method; you, the programmer, must insert code to make it do what you want when the button is pushed. Add this line of code after line 50 (in the body of the `button1ActionPerformed()` method, replacing the comment `//TODO add your handling code here:`) in the Netbeans Appendix, “Adding, Connecting, and Testing a `Button`”, Listing A.6.

```
textField1.setText("Greetings!");
```

Execute the modified program, it should make the `TextField` say “Greetings” when you press the button.

## Simulating One Bank Account By Hand (Without the Account Class)

You know how to get the text from a `TextField` (by using `getText()`), how to set the text in a `TextField` (by using `setText(String)`), and how to get control when a `Button` is pushed. Before going on to writing classes, let’s experiment with a simple ATM with just one bank account balance stored in the `Applet`. You must do the following three things:

1. Create a variable to contain the current balance
2. When the user types an amount to withdraw and hits enter, get that withdrawal amount into the program as a number
3. Subtract it from the balance and display the new balance

These will be explained next; after some practice the explanations will make more sense.

### Create a Variable to Contain the Current Balance

Assume the bank account starts with 1000 dollars. The program must store the value 1000, and then decrease it when money is withdrawn. To store information (like 1000) a Java program uses what is called a *variable*. A variable must be declared, with a name and a type and a value, like this:

```
int balance = 1000;
```

This declares a variable, named `balance`, whose type is `int` (i.e. like an integer, it stores one whole number) and sets its initial value to 1000.

Add that line of code to your test `Applet`, outside of any method, but inside the class. You may be asking, “Where is inside the class?”. The class definition is everything between the `{}`’s following the name of the class. Now, you might ask, “Where is outside any method?”; this will be explained soon, for now use line 14 of the Netbeans Appendix, “Adding, Connecting, and Testing a `Button`”, Listing A.6 (i.e. after line 13, that reads):

```
public class TestGUIApplet extends java.applet.Applet {
```

### When the User Hits Enter, Get the Withdrawal Amount

Change the body of `textField1ActionPerformed()` by adding these lines:

```
int withdrawal = Integer.parseInt(textField1.getText());  
System.out.println("withdraw=" + withdrawal);
```

The first gets the text from the `TextField` (which has type `String`), converts it to a number (`Integer.parseInt()` does that) and stores it in a variable called `withdrawal`. The second

prints that value, so you can see if it worked. Try it out now (don't forget to type a number in the `TextField`! What happens if you type anything else?).

You will soon get tired of typing `System.out.println("whatever");` every time you want to output debugging information. NetBeans provides a shortcut; simply type "sout" -- the letters s-o-u-t and then a tab. This will be automatically expanded into `System.out.println("");`

### **Subtract It From the Balance and Display the New Balance**

Add two more lines (after the two you just added):

```
balance = balance - withdrawal;
System.out.println("new balance is: " + balance);
```

The first subtracts the value of `withdrawal` from the `balance` and stores the remainder back in `balance`. The second, um, prints the new balance. Execute this code. Try hitting enter more than once.

Congratulations! There are three things that do all the work in computing: input, assignment and output; you have just used all three! Now, on to creating classes.

## **A Generic Problem Solving Technique**

Let us take a slight detour for a problem solving technique that will be useful in thinking about the `Account` class. Try to answer this word problem (even if you don't like algebra).

*Mary is twice as old as John. Two years ago, she was 3 years older than he will be next year. How old is John?*

Maybe you can read that, see through it, and just know the answer. Maybe you have learned a technique called "guess and check" where you try out ages and adjust them in the right direction until you stumble on the answer. Maybe you know how to write equations to compute the answer. The first technique is a great one, if it works. Guess and check finds the answer, but does not give you any insight into the problem. Writing equations and solving them is a more general and useful method. But not everyone can do this sort of algebra problem. Here's a generic problem solving technique that may help you learn to solve such problems using that more general method (and which has direct relevance to designing object programs).

**Problem Solving Technique:** *What are the things? What are their relationships?*

*By answering these two questions you concretize your conception of the problem. Just doing that may make solving it easier because it will cause you to think carefully about it. In the context of object design, the things are potential classes, the relationships, potential messages.*

In the problem at hand, at first glance there are several things, Mary, John, and their ages. But, if you think about it, you will soon realize that Mary and John don't matter, only their ages. For

brevity, call their ages  $M$  and  $J$ . And not only are their current ages of interest, but also Mary's age two years ago ( $M-2$ ) and John's next year ( $J+1$ ). Those are the things of interest.

The two statements before the question state two relationships. They can be rewritten as: *Mary is twice as old as John*. In symbols, that's  $M = 2*J$ ; and, *Two years ago, she was 3 years older than he will be next year*. In symbols, that's  $M-2 = (J+1)+3$ . The latter simplifies to  $M = J+6$ , and you can then substitute  $2*J$  for  $M$  to find the answer.

## **Account Class: Design, Implementation and Testing**

Your program must manage three bank accounts. It would be natural to create three bank account objects, where each would correspond to and hold the information for one account. To be able to create objects (like bank account objects) you must first define a class. When you define a class you create a user-defined type. A class can then manufacture objects of that type whenever you tell it to.

### **Account Class Design**

Let's apply the "things and relationships" technique to the ATM problem. The things in this problem are bank accounts, plus the names and balances of each account; the relationship between those things is that each account has a name and balance associated with it. Each account will need a way to store those two pieces of information.

Along with the information to be stored in each account, we must also consider what actions will be performed on, or using, that information. There are only two: display, and withdraw. In the former the balance (and perhaps the name) must be displayed, in the latter the balance must be reduced when money is withdrawn.

The details of how to store the information in each account and how to access and change it will be covered in the next section.

### **Converting the Design to Java Code**

A class definition includes declarations of variables (where information is stored) and methods (which operate on that information).

#### **Variables (state)**

Variables encode state. This usage of "state" is very similar to a common English usage. If someone says to you, "What is the state of your bank account?", they mean, how much money is in it. Here are some facts about variables that you will need to know later:

1. Variables are containers for information. The purpose of variables is to hold information. That information may be of various types; numbers, letters, words, or even objects. Java variables are similar to, but different from, variables in algebra. Like in mathematics, Java variables might contain various values. The particular values determine the state of the computation (see the section, "State and Its Representation"; in Chapter 1.)

2. There is only one way to change the information in a variable, which is to execute an *assignment statement*. In algebra, you might encounter a formula like  $x = y * 2$ , and be asked, if  $y=2$ , then what is  $x$ ? In Java, by contrast,  $x = y * 2$ ; means take the value of  $y$ , multiply it by 2, and store that value in the variable  $x$  (or, assign that value to  $x$ ). So, if the value of  $y$  were 17, and that assignment statement were executed, afterwards the value of  $x$  would be 34.

3. Variables hold exactly one value at a time. Whatever value had been in  $x$  before that assignment statement was executed is irretrievably lost. If you wanted to keep the old value, you would need another variable to hold it.

4. Every variable has three attributes: name, type and value. Java is what is called a typed language. In addition to a name and a value, every variable in Java has a type.

For now we will only consider variables of two built-in types. The first you have already seen; `String` is used to store a series of literal characters. The second, `int`, is used to store whole numbers, like 12, 1000000, or -37.

### Adding Variables to the Account Class

To start with, create an `Account` class in Netbeans that includes a `main()` method. Follow the instructions in the Netbeans Appendix “Creating a Class with a Test Driver”. Add the variables in Listing 3.1 after line 12 in Listing A.7 in that section (i.e. after the `{` in the line `public class Account`).

Listing 3.1 Adding the Variables to the `Account` Class

```
1 public class Account {
2     String name;           // a String variable called name
3     int balance;          // an int variable called balance
```

**Line 3:** `int balance;` declares a variable of type `int` with the name `balance`.

```
    // an int variable called balance
```

is a comment meant for a person to read; Java ignores it.

So far, this class is not very useful because although each `Account` could store a name and a balance, there isn't any way to do anything with them. That's what methods are for.

### Methods (Action) -- Accessors and Withdrawal

Methods allow classes to do things, to perform actions. You can write methods to do whatever you like. The body of a method has a list of instructions to follow to accomplish whatever that method is supposed to do.

Perhaps the most common methods are called accessors. Almost every class has accessors. Their form and usage is simple once you get used to them, and it is the same in every class. At first, until you are familiar with the Java constructs involved, they may seem a bit mysterious. For

now, simply accept them as idioms without a complete understanding of every detail; in a few chapters they will seem simple and obvious.

### *Accessing/Changing Information in an Object*

There are two things that our `Accounts` are supposed to do besides retaining the name and balance associated with the `Account`: they must allow some other method, outside of `Account` to discover the value of the balance variable, and also allow the balance to be changed when money is withdrawn. So the questions that must be answered are: 1) “How to get the value of a variable that’s inside an object?”, and, 2) “How to set the value of a variable that’s inside an object?”.

The answer is *accessors*, methods that allow you to access the variables inside objects. These common, simple, useful methods are not easily understood at first. The problem is that even though accessors all follow the same pattern, there are many details involved in the mechanisms that implement them. Once you have programmed for a few weeks (or a half a dozen, depending on how often and how carefully you do it) accessors will seem easy and natural. For now, accept them as an idiom. Add the two methods, in Listing 3.2, right after the two variables you already added.

#### Listing 3.2 Accessors For Balance

```
1  public void setBalance(int nuBalance) {
2      balance = nuBalance; // set the balance
3  } // setBalance
4
5  public int getBalance() {
6      return balance; // return the balance
7  } // getBalance
```

**Lines 1-3:** The `setBalance()` method sets the balance to the value that is sent along with the message.

**Line 4:** This blank line is inserted to set off the methods visually. It is a simple, but important, element of style.

**Lines 5-7:** The `getBalance()` method returns (i.e. sends back as its value) the current value of the `balance` variable.

This class, though very small, can now be tested. To reiterate, it is important to test your code as you develop it; that way, when something goes wrong, it is easier to isolate the error -- there are simply fewer places to look.

### *Testing the Accessors*

How can you test this class? You write what is called a *driver program*, the sole purpose of which is to test your class. This may seem like a waste of time, but it is not! To convince yourself that a class works you must test all its methods. Fortunately, there are only two: `getBalance()`

and `setBalance(int)`. Go to the `Account` class. Modify the `main()` method so that it looks like the code in Listing 3.3.

### Listing 3.3 Testing the Accessors For Balance

```
1 public static void main(String[] args) {
2     Account myAccount = new Account();
3     System.out.println("Before balance=" + myAccount.getBalance());
4     myAccount.setBalance(1234);
5     System.out.println("After balance=" + myAccount.getBalance());
6 }
```

**Line 2:** Create and store an `Account` object, named `myAccount`.

**Line 3:** Check the initial balance, it should be 0. This tests `getBalance()`.

**Line 4:** Set the balance to 1234.

**Line 5:** Find out if the balance is now 1234. This tests `setBalance()` and `getBalance()`

Now execute your program. Assuming it displays 0 and then 1234, it worked and you can move on to the `withdraw(int)` method. If you made any typing mistakes, fix them. Then, forward!

### *Withdrawals*

Along with accessing the `balance` variable, the other action our `Account` class must take is handling withdrawals. Therefore it will need a method, which might as well be named `withdraw`. It is important to give methods (as well as classes and variables) descriptive names. That makes less things to remember.

When a user types an amount to withdraw and then hits the `Enter` key we should send our `Account` object a `withdraw()` message. Will the `withdraw()` method need any information to carry out its task? Yes, it needs to know how much to withdraw! So it will need a parameter to pass that information to the method. The amount to withdraw is a number, in whole dollars, so the type of the parameter is `int`. The name of the parameter is up to us; it can be any legal identifier -- in the example, the name `amountToWithdraw` was chosen (see Listing 3.4).

How should the `withdraw()` method change the state of the `Account` object which receives it? This is the same as asking how the state of a bank account should change when a person takes money from it using an ATM. The answer is that the balance should be reduced by the amount of money withdrawn. So, first our method must calculate the new balance by

```
balance - amountToWithdraw,
```

and then store that amount back in the `balance` variable. To change the value of a variable you use an assignment statement; like this:

```
balance = balance - amountToWithdraw;
```

This assignment statement is the only line in the body of the `withdraw()` method.

#### Listing 3.4 The `withdraw()` Method For the `Account` Class

```
1 public void withdraw(int amountToWithdraw) {
2     balance = balance - amountToWithdraw;
3 }
```

**Line 1:** The method heading. There is one parameter of type `int` whose name is `amountToWithdraw`.

**Line 2:** An assignment statement. This will subtract whatever value is in the parameter `amountToWithdraw` from the value in the variable `balance` (inside whatever `Account` the `withdraw()` message was sent to), and then store the result of the subtraction back in that same variable (inside the `Account` object that got the `withdraw()` message).

Add this method to your `Account` class and test it. How to test? Add two more lines to your main method, possibly those in Listing 3.5.

#### Listing 3.5 Testing the `withdraw()` Method as Well

```
1 public static void main(String[] args) {
2     Account myAccount = new Account();
3     System.out.println("Before balance=" + myAccount.getBalance());
4     myAccount.setBalance(1234);
5     System.out.println("After set balance=" + myAccount.getBalance());
6     myAccount.withdraw(235);
7     System.out.println("Withdrew, balance=$" + myAccount.getBalance());
8 }
```

Assuming your program says the final balance is \$999, this means all three methods work and you are ready to use your `Account` class along with your GUI to build the complete system. Well, almost ready. There's the small matter of keeping track of three accounts instead of one.

## Objects and Classes

By defining a class (like the `Account` class defined in Listings 3.1, 3.2, and 3.4) you create a user-defined type. Then you can create objects of that type (see Line 2 of Listing 3.5). Novice programmers sometimes struggle with the distinction between classes and objects. It is rather like the relationship between the part of speech, noun, and individual nouns. Words like “dog”, “house”, and “money”, are nouns. I.e., “dog”, “house”, and “money”, are instances of the category noun. Similarly, an object is an instance of a class. Every object is an instance of some class.

If you are just learning to program, until very recently you had never heard the terms class and object used the way they are used in the context of programming. This unfamiliarity makes making sense of the difference between them rather difficult. It gets easier with practice.

To review: classes are patterns for creating objects of that type; they define the type of information those objects will contain (variables) and the actions they can take (methods). Once the class is defined, you can create (instantiate) as many objects as you want of that type. Once you have created an object, you can send it messages to accomplish whatever task you are working on.

### **Cookies and Cookie Cutters Metaphor**

A useful metaphor for classes and objects is cookie cutters and cookies. If you have a cookie cutter in the shape of a star, you can use it over and over to make many star-shaped cookies. After the various star cookies are cut, they can be decorated in different ways. Classes are patterns (like cookie cutters) for making objects (cookies). All objects (also called instances) of a particular class have the same form (like all the cookies from the same cutter). Every `Account` you create (by saying `new Account()`) will have its own `balance` variable and its own `name` variable. These variables may contain different values in different objects.

It's important to remember the distinction between objects and classes. For while cookies can be delicious, if you bite a cookie cutter you could hurt your mouth; plus it wouldn't taste good.

### **`new Account()`; Instantiation! Alakazam!**

Before you can send a message to an `Account` you must create it, and store it in a variable. This is demonstrated in Listing 3.6. Instantiation may safely be considered as magic for the present. In Chapter 5, the details will be explained, and at that point it will be essential to understand, but for now you might just think of `new Account()` as the incantation to make a new `Account` object appear.

Listing 3.6 Instantiation. A line of Code That Creates an `Account` Object

```
1
2  Account myAccount = new Account();
```

**Line 2:** There are two parts here: 1) `Account myAccount` -- declares a variable of type `Account` (that's the class name) whose name is `myAccount`. 2) `new Account()` -- instantiates an `Account` (i.e. creates an instance of an `Account` object). The equals sign between them makes these two parts into an assignment statement. The action of this assignment is to store the newly created `Account` in the `myAccount` variable.

Notice that the first character of `myAccount`, 'm', is lowercase; the convention is that classes have uppercase first letters, objects, lowercase -- if you follow this convention, you can tell right away from the name of a thing whether it is a class or an object. Thus, `Account` is the name of the class, and `myAccount` is the name of the object.

### **Instances and Instance Variables**

The `Account` class has two instance variables, `name` and `balance`. Therefore, every object of type `Account` (or instance of `Account`) has its own instance variables named `name` and `balance`; just as real every bank account has a name and a balance associated with it. You will see an example of what this means in the next section.

*Exercise: Use your `Account` class in conjunction with your GUI to keep track of the balance of one account. You already have all the necessary code; all you need is to understand it well enough to rearrange it to do the job.*

## Creating and Testing the Finished GUI

As always, there are two phases to creating a GUI: design and implementation.

### GUI Design

We have built and tested a prototype GUI and completed the `Account` class. We are now ready to solve the original problem. We will again follow the technique of starting with the user interface and then writing the classes to support it. Designing the GUI both puts our focus on the user and what actions they can take (which is a good policy) and helps us form an image of the task ahead of us.

We already have a `display` `Button` and a `TextField` to handle withdrawals; all we need is a means to choose between the three accounts. How should the user select the account to withdraw from or display the balance of? In a real ATM machine, the user inserts their card and then enters their PIN. But, we don't have a card reader, and don't want to keep track of PINs just yet. A simple solution is to add three buttons for the three accounts. When the user pushes the `Account2` button, the program will act as if the owner of the `Account2` has successfully logged in; the program will display the balance from `Account2`. Subsequent withdrawals will come from `Account2`, until another button is pushed.

### GUI Implementation

Add three more `Buttons` to your GUI (see the Netbeans Appendix, "Adding, connecting and testing a `Button`", if you've forgotten how). The `Buttons` appear named `Button2`, `Button3`, and `Button4` and they are labeled the same way. These names do not remind you what they mean, and the user will object if to select account 3, they must push the `Button` labeled `Button4`. Just as it is important to give classes, variables and methods descriptive names so that you can remember what they do without thinking, it is important to label `Buttons` well or the user will become confused. When there is only one, the name is not so important, but the more there are, the more it matters. Change the labels on the `Buttons` to "Account 1", "Account 2" and, "Account 3" (see the Netbeans Appendix, "Adding and using a Choice"). Change their names to "`selectAccount1Button`", "`selectAccount2Button`", and "`selectAccount3Button`" (see the Netbeans Appendix, "Renaming Components"). Make sure the `Button` labeled "Account1" is named "`selectAccount1`"! It is very confusing and hard to figure out when the `Button` with the label "Account1" gives access to `Account3`.

Now, add actions for the three new `Buttons` (double-click them, remember?). They are supposed to select the current account, so have each one send the `Bank` a message: `selectAccount1()`, `selectAccount2()`, or `selectAccount3()`. Naturally, the `selectAccount1Button` should send the bank the `selectAccount1()` message.

Listing 3.7 shows how the `actionPerformed()` code Netbeans wrote for the `account1Button` looks.

Listing 3.7 The `actionPerformed()` Method for the `account1Button`

```
1 private void account1ButtonActionPerformed(java.awt.event.ActionEvent e){
2     // Add your handling code here:
3 }
```

Notice that the name of the `Button` appears as part of the method name.

All you need to do is add code to tell the bank to select `Account1()`, as shown in Listing 3.8.

Listing 3.8 The Code to Add to Make the `account1Button` Select `account1`

```
1 private void account1ButtonActionPerformed(java.awt.event.ActionEvent e){
2     theBank.selectAccount1();
3 }
```

Do the same for the other two `Buttons`. You will notice that Netbeans indicates that there is an error on each of those lines. This is because it does not know what `theBank` means. You will need to add the line

```
Bank theBank = new Bank();
```

outside of any method (details below) and create the `Bank` class. After you do these two things, the errors will go away and the program will be ready to run.

## The Bank Class: Design, Implementation and Testing

Just as a GUI must be designed, implemented, and then tested, so must every class.

### Bank Class Design

The task of the `Bank` class is to simulate a tiny bank with three accounts. Thus, it will need three `Accounts`. It must also keep track of which `Account` is currently in use, and make withdrawals from that `Account`.

There are two tasks we must accomplish; 1) create the three accounts, and, 2) conceptualize and implement a technique to remember which of the accounts the user is currently working with. It turns out that declaring one more `Account` variable and setting it equal to whichever `Account` is currently in use will be sufficient; see below.

### Converting the Design to Java Code

The first thing to do is to create a `Bank` class with a `main()` method; see the Netbeans Appendix, “Creating a class with a test driver”, for instructions. Netbeans will write the class shown in Listing 3.9.

### Listing 3.9 The Initial Bank Class

```
1  /*
2   * Bank.java
3   *
4   * Created on April 21, 2004, 2:40 PM
5   */
6  public class Bank {
7
8      /** Creates a new instance of Bank */
9      public Bank() {
10     }
11
12     /**
13      * @param args the command line arguments
14      */
15     public static void main(String[] args) {
16     }
17 }
```

### Variables

To create the three accounts, plus the current account variable, add these four lines after line 6 in Listing 3.9. That's all it takes.

```
Account account1 = new Account();
Account account2 = new Account();
Account account3 = new Account();
Account currentAccount = account1; // to start with
```

The first three lines declare variables called `account1`, `account2` and `account3`, instantiate three Accounts, and store one Account in each variable. The fourth declares another Account variable called `currentAccount`, and sets it equal, initially, to `account1`. Thus, if you do a withdrawal before pushing any of the three buttons it will come from `account1`. The modified code is shown in Listing 3.10.

### Listing 3.10 The Account Declarations Inserted Into the Bank Class

```
1  public class Bank {
2
3      Account account1 = new Account();
4      Account account2 = new Account();
5      Account account3 = new Account();
6      Account currentAccount = account1; // to start with
7
8      /** Creates a new instance of Bank */
9      public Bank() {
```

**Line 6:** The `// to start with` is a comment. Anything on a line after `//` is a comment. Comments are ignored by Java, they are to help people understand code.

**Line 8:** This is also a comment; anything between `/*` and `*/` is a comment; these block comments can span multiple lines.

## Methods

The bank needs to withdraw money from and display the balance of the current account; it also must be able to set the current account. Listing 3.11 shows the methods that do these things. These may be inserted anywhere in the class block of the `Bank` class, as long as they are outside of other methods. Right after the variables would be fine.

Listing 3.11 The Method Declarations For the `Bank` Class

```
1 public void selectAccount1() {currentAccount = account1;}
2 public void selectAccount2() {currentAccount = account2;}
3 public void selectAccount3() {currentAccount = account3;}
4
5 public int getBalance() {
6     return currentAccount.getBalance();
7 }
8
9 public void withdraw(int withdrawalAmt) {
10     currentAccount.withdraw(withdrawalAmt);
11 }
```

Both `getBalance()` and `withdraw()` just pass the buck to the `Account` class by sending the same message to the `currentAccount` object. Notice that `getBalance()` returns the value that comes back from `Account`'s `getBalance`.

None of these methods has `static` in their headings. That is because they are instance methods. All methods are either instance methods or class methods. When a method uses or changes the values of variables within an instance (like `withdraw()` changes the value of an `Account`'s balance variable), it must be an instance method. Otherwise it cannot access the information in the instance. If it does not use or change any information inside an instance (like the `PersonalizedGreeter sayHi()` method in the previous chapter), it *may* be a class method. All methods for the next six chapters will be instance methods; you can safely forget about `static` for some time.

## Testing

Modify the `main()` method in `Bank` as shown in Listing 3.12. Build and run the program again to make sure the `Bank` is working correctly. Make sure to compile all the classes by right-clicking on the project node (at the top of the Projects tab pane) and selecting Build Project, before you run it; otherwise the other classes will not be recompiled and changes you have made there will not occur.

Listing 3.12 Test Code for the `Bank` Class

```
1 public static void main(String[] args) {
2     Bank theBank = new Bank();
3     System.out.println("Initial acct1 balance=" +
theBank.getBalance());
4     theBank.withdraw(100);
5     System.out.println("-100 acct1 balance=" + theBank.getBalance());
6     theBank.selectAccount3();
7     System.out.println(" acct3 balance=" + theBank.getBalance());
```

```

8         theBank.selectAccount1();
9         System.out.println(" acct1 balance=" + theBank.getBalance());
10    }

```

- Line 2:** Create a new `Bank` object and store it in a `Bank` variable named `theBank`.
- Line 3:** Display the balance of `account1` (remember `currentAccount` starts as `account1`).
- Line 4:** Withdraw 100 dollars from `account1`
- Line 5:** See if the balance is really -100.
- Line 6:** Tell `theBank` to change the current account to `account3`.
- Line 7:** Verify that `getBalance()` now returns 0 (instead of -100)
- Line 8:** Go back to `account1`.
- Line 9:** Verify that its balance is still -100.

Sometimes it is helpful to draw a picture of the state of the program at various points during its execution. For instance, after line 2 in Listing 3.12 executes, the state of the program looks roughly like Figure 3.3.

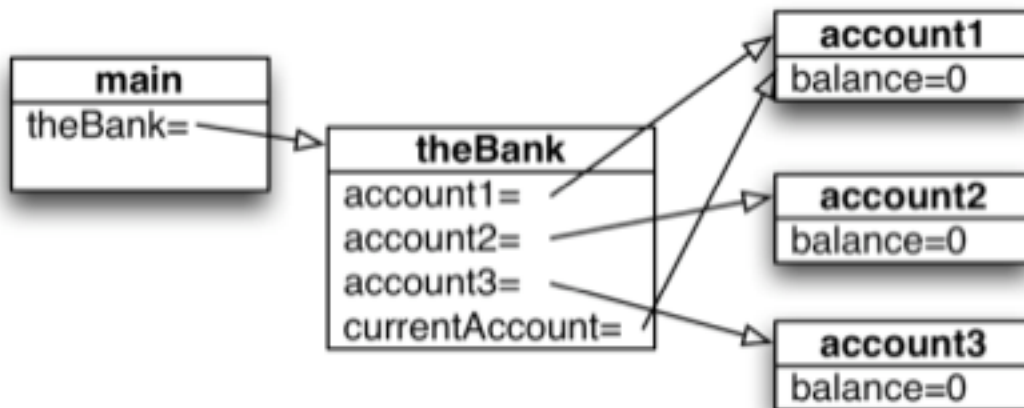


Figure 3.3 The state of the program after line 2 in Listing 3.12 executes. The `main()` method has a variable named `theBank`. The object called `theBank` has four variables. The first three point to the three `Accounts`. The fourth, currently points to `account1` as well.

After line 6 executes, the state of the program looks like Figure 3.4.

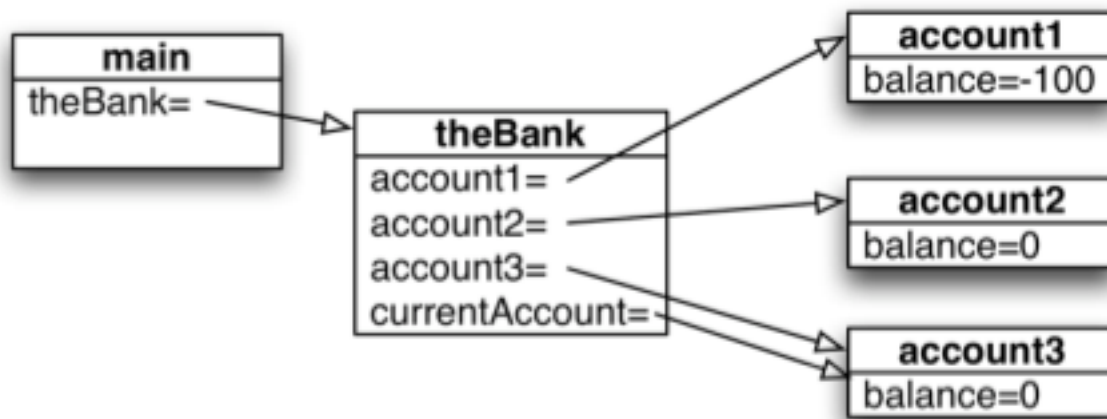


Figure 3.4 The only differences are the value of the `balance` variable in `account1` and the value of `currentAccount` in `theBank`, which now points to `account3` instead of `account1`.

## Putting It All Together -- Finally!

All you need to do to finish the project is to add one line to the `Applet`, as seen in Listing 3.13.

Listing 3.13 Alteration to `Applet` Code

```

1  public class ATM_Applet extends java.applet.Applet {
2      Bank theBank = new Bank ();
3  }
  
```

To declare the `Bank` variable, `theBank`, add this line at the beginning of the `ATM_Applet` class.

That's it! You are done! Now you can compile and run the `Applet`. Do that now (don't forget to change the main class back to `ATM_Applet -- Project/Set Project Main Class...`). Make sure it really does keep track of the three balances correctly. You will notice that everyone starts out with a balance of 0, but it still allows them to withdraw money.

*Could you arrange for them to start with some other balance besides zero?*

This turns out to be easy, but not at all obvious. The initial *default value* for every instance variable is 0, but you can set it to something else as shown in Listing 3.14. Do that now and rerun your program.

Listing 3.14 Alteration to Listing 3.1

```

1  public class Account {
2      String name;
3      int balance=1000000; // Every Account starts with $1,000,000!
4  }
  
```

To set the initial balance for every account to \$1,000,000. Change is in **bold**.

Assuming you've made that program run, good work! You have done some of the most difficult programming you will ever do; working in a new language in a new IDE is incredibly difficult. There are so many details and everything is unfamiliar so even the simplest problem can seem overwhelming. It only gets easier from here. With practice everything you did today will become easy and effortless, and in a few weeks you'll be amazed that this ever seemed difficult. That's how expertise works; you learn something and it seems simple. Then it's hard to understand why everyone else can't do it. Welcome to Java programming.

## Conclusion

Programming is always an iterative process. No one writes finished programs from scratch in one go. Rather it is a process of successive approximation. There are two compelling reasons for starting with a prototype, which does almost nothing, and then adding functionality after each simpler program works. First, it allows the programmer to focus on one part of the program at a time, and thus reduces cognitive overhead during the design and implementation phase. This is also an advantage of writing and testing classes one at a time. Second, it makes the task of debugging much easier. Debugging is the hardest part of programming for all levels of programmers; it can be baffling, frustrating and exhausting. Stepwise implementation makes it much easier to find bugs when they occur, simply because a smaller part of the program is new at any time.

This chapter introduced classes; their design, implementation, testing, and incorporation into a larger program. It included various language constructs and components, including: variables, the assignment statement, declarations, instantiation, accessors, parameters, and return values along with the AWT `Component`, `Button`, and `TextField`. These various elements were presented rather telegraphically, with most of the detail omitted. A more complete and careful discussion of these topics may be found later on. Here the emphasis is on constructing a working program to experiment with to get a feel for both the Java language and the process of programming.

If you are new to programming and/or Java you may be feeling a bit confused. You probably have many questions about what you've just done, and a number of concepts you are very unsure about. If so, good! There were too many concepts and details to describe or understand all at once. But, at least you have seen the process of constructing a working `Applet` with several classes and a GUI. That's a lot (and it's most of what programming is about). It usually takes about 3 or 4 weeks for undergraduates in an introductory class to be able to do this sort of program. Oh, and then there was all the detail of using Netbeans to build the GUI and compile/execute the program. The next chapters will fill in the detail, and if it answers questions you have in the back of your mind, the details will be easier to understand and remember.

The next chapter will introduce `Graphics`, `Color` and software reuse techniques through the development of another example. After that comes a detailed explanation of many of the concepts

and programming features glossed over here. Then some more elaborate (and interesting) examples will be undertaken.

## What Could Go Wrong?

Earlier, in the paragraph titled, “When the user hits enter, get the withdrawal amount”, if the contents of the `TextField` is not an `int`, an `Exception` will be thrown.

In the paragraph titled, “GUI implementation”, it is very easy to get confused over which `Button` is which, and which `ActionPerformed()` should send what message. If something goes wrong with the account selection that would be the first place to look.

## Review Questions

- 3.1 How do you create objects?
- 3.2 What is the difference between a class and an object?
- 3.3 When you declare a variable, you must specify its type and name (in that order). What are you allowed to call variables? I.e. what rules must variable name (indeed all Java identifiers) adhere to?
- 3.4 Here are two legal names: `something`, and `Something`. These are different, because Java is case sensitive. If you are following the convention for naming classes and objects which of these is the name of a class and which the name of an object?
- 3.5 Where is information stored in Java program?
- 3.6 How do you change the value of a variable?
- 3.7 What is the difference between the types `int` and `String`?
- 3.8 What are the three attributes of every variable?
- 3.9 What are parameters used for?
- 3.10 Where do parameters go?
- 3.11 Is `doSomething()` a variable name or a method? How can you tell?
- 3.12 In `anything.everything(something)`, you can tell by the context what `anything`, `something` and `everything` are. What are these three things? The answers are message, object, and parameter, but which is which?
- 3.13 Recall that computing is always information processing; in the ATM problem description, what information is processed, input, or output, for each possible user action?

## Programming Exercises

3.14 The instructions omitted the labels above and below the `TextField`. Add them.

3.15 Your finished `Applet` was not very user friendly. When a customer withdrew cash, and wanted to see the new balance, they had to push the display button again. It is simple (i.e. one line) to automatically display the new balance every time a withdrawal is made. Do so. Hint: write a `public void display()` method in your `Applet` (that does exactly what the display button does) and use it in `actionPerformed()` for the `TextField` right after you do the withdrawal (i.e. say `display()`).

3.16 After you do the previous exercise, you will discover a small inconvenience. If you want to withdraw the same amount repeatedly from the same account, you have to keep removing the balance and reentering the amount before you can withdraw again. Improve the usability of your interface as follows. After a withdrawal is made, move the cursor to the `TextField` and select the text so the user can simply copy the amount to be withdrawn (ctrl-c) and then paste and hit enter over and over. Add the code to `actionPerformed` for the `TextField` (right after the `display()` from the previous exercise). If the name of your `TextField` was `theTF`, the code would be:

```
theTF.requestFocus();
theTF.selectAll();
```