Chapter 11 Data Structures

Introduction

Up until now, we have been declaring variables one at a time. But, sometimes you want ten, or a thousand variables. If you were implementing software for a real bank, it might have thousand of accounts. If you wanted to program an army of one hundred snowpeople, it would be hopeless to write one hundred statements for each action you wanted them to perform. A big advantage of computing is that the machine doesn’t mind doing the same thing over and over thousands of times. Data structures allow you to declare and store as many variables as you need with a minimum of effort. This chapter will show you how to use two similar data structures, array and ArrayList<Object>.

Arrays

Arrays are part of most programming languages. An array is a list of variables, all with the same type and name, but distinguished by an index. The declaration:

```java
int [] anArray = new int[100];
```

declares 100 variables of type int, all named anArray[something], where something is an int between 0 and 99. Thus, the first int variable is named anArray[0], the next, anArray[1], and the hundredth, anArray[99]. Each one acts exactly like an int variable, because each one is an int variable. The value in the square brackets is called the index. It may be a constant, but usually it is a variable.

Simplest Examples

Listing 11.1 declares an array of five ints and then displays them to System.out.

Listing 11.1 Declaring and Printing an Array of Five ints: Code and Output

```
1    int [] list = new int[5];
2
3    for (int index=0; index<5; index++) {
4      System.out.println("index=" + index + " list[index]=" + list[index]);
5    }
```

- **Line 1**: Declares an array, called list, of five ints.
- **Line 2**: A for loop to iterate over the five elements of the list

The loop on lines 3-5 is a standard method of accessing the elements of an array one at a time; it is said to “iterate over the elements of the array”. As you can see (by the output), they are auto initialized to zero when the array is declared.

If you want values besides zero in the array element, you must put them there, as shown in Listing 11.2.
Listing 11.2 The Same Array with \texttt{list[0]} Set to 7 and \texttt{list[3]} to 33: Code and Output

1 int [] list = new int[5];
2 list[0] = 7;
3 list[3] = 33;
4
5 for (int index=0; index<5; index++) {
6 System.out.println("index=\" + index + " list[index]=\" + list[index]);
7 }

- **Line 2:** Assign the value 7 to the first element of the array
- **Line 3:** ...and 33 to the fourth.

Listing 11.3 illustrates setting the values of the array in a loop. It uses the current value of the index squared as the value stored in each element.

Listing 11.3 The Same Array with Each Element Set to the Square of Its Index

1 int [] list = new int[5];
2 for (int i=0; i<5; i++) {
3 list[i] = i*i;
4 }
5
6 for (int i=0; i<5; i++) {
7 System.out.println("i=\" + i + " list[i]=\" + list[i]);
8 }

- **Line 3:** Assign each element the value of the square of its index

Notice that “i” stands for “index”.

**Printing a String Backwards**

Arrays may be of any type, primitive, built-in or user defined. An array of \texttt{chars} could be used to print a \texttt{String} backwards as shown in Listing 11.4.

Listing 11.4 Printing a String Forwards and Backwards, One \texttt{char} Per Line

1 char [] letters = new char[100];
2 String s = "pals";
3 for (int i=0; i<s.length(); i++) {
4 letters[i] = s.charAt(i);
5 }
6
7 System.out.println("frontwards, it's: ");
8 for (int i=0; i<s.length(); i++) {
9 System.out.println("i=\" + i + " letters[i]=\" + letters[i]);
10 }
System.out.println("backwards, that's: ");
for (int i=s.length()-1; i>=0; i--) {
    System.out.println("i= " + i + " letters[i]= " + letters[i]);
}

frontwards, it's:
i=0 letters[i]=p
i=1 letters[i]=a
i=2 letters[i]=l
i=3 letters[i]=s
backwards, that's:
i=3 letters[i]=s
i=2 letters[i]=l
i=1 letters[i]=a
i=0 letters[i]=p

Lines 3-5: Assign each char in the String to an element of the array
Lines 7-10: Print them forwards

If length() and charAt() seem unfamiliar you might look back at “A Few String Methods” in Chapter 8.

To print the String frontwards and backwards, all on the same line, just change the printlns to prints and remove some of the text, as in Listing 11.5.

Listing 11.5 Printing a String Forwards and Backwards, All on One Line

char [] letters = new char[100];
String s = "pals";
for (int i=0; i<s.length(); i++) {
    letters[i] = s.charAt(i);
}

System.out.print("the word ");
for (int i=0; i<s.length(); i++) {
    System.out.print(letters[i]);
}

System.out.print(" backwards, is ");
for (int i=s.length()-1; i>=0; i--) {
    System.out.print(letters[i]);
}

the word pals backwards, is slap

Lines 3-5: Assign each char in the String to an element of the array
Lines 7-10: Print them forwards
Lines 12-15: ...and backwards

An Array of Accounts

If you were writing a bank simulation with 1000 Accounts, you could declare an array like this:

Account[] accountList = new Account[1000];

This will give you 1000 Account variables, each initialized to zero, which, when the variable is a reference (as any Object variable is), is interpreted as null. Thus, if you wish to have 1000
Accounts in those 1000 Account variables, you must do the second step of instantiating them all; like this:

```java
    for (int i=0; i<1000; i++)
        accountList[i] = new Account();
```

If you forget to do this (and, everyone does when they start working with arrays of Objects), you will be confronted with NullPointerExceptions the first time you send a message to one of them; and if you’re not paying attention, it could be very confusing. Don’t forget!

Although this could work, it is almost always better to use ArrayList for lists of Objects.

**ArrayList<Object>**

Java provides the ArrayList class to keep track of lists of objects. An ArrayList<Object> stores a list of variables of type Object. Thus it can store any type of object, since every object is an instance of some class and every class extends Object (directly or indirectly). This is very convenient, but has a downside. When you get objects back out of the list, the compiler considers them to be of type Object. So, it will only allow messages that are defined in Object to be sent to them. To fix this you can specify what kind of Object the list contains. If it contains Accounts, you declare it as ArrayList<Account> and then Java knows the things in the list are Accounts.

**The add(Object) Method**

To add an object, any object at all, to a ArrayList, use add(Object); i.e. simply send the list the add() message with that object as a parameter. The object will be added to the end of the list. For example, to make a list containing three Accounts, you could say:

```java
    java.util.ArrayList<Account> theList = new java.util.ArrayList<Account();
    theList.add(new Account("xena", 1234567));
    theList.add(new Account("abe", 100));
    theList.add(new Account("bea", 10000000));
```

Then first Account would be xena’s, the last bea’s.

**The iterator() Method**

To access each item in a ArrayList, from first to last in order, use a for-each loop: like this:

```java
    for (Account nextAccount: theList) {
        System.out.println("\n\nnext account..." + nextAccount);
    }
```

The form of this loop is always the same; it is an idiom. The initialization declares a variable, nextAccount, of type Account and initializes it to all the items in theList, one-by-one from first to last and then executes the body of the loop for each.

In English, this loop iterates over the ArrayList called theList; each time around the loop it stores the next Account from the list in the nextAccount variable, and then (implicitly) sends it toString(); whatever toString() returns is then the parameter to System.out.println().
Simplest Test Program

As usual, to convince yourself that a programming technique works and, more importantly, to become familiar with it before trying to use it for anything difficult, you should write a tiny test program. There are many ways one might do this, but here it is done with an `ArrayList`, as shown in Listing 11.6.

Listing 11.6 Simplest Use of a `ArrayList`

```java
1    import java.util.*;
2
3    public class ArrayListTest {
4
5        ArrayList<Account> theList;
6
7        /** Creates a new instance of ArrayListTest */
8        public ArrayListTest() {
9            theList = new ArrayList<Account>();
10            theList.add(new Account("xena", 12345));
11            theList.add(new Account("abe", 100));
12            theList.add(new Account("bea", 10000000));
13
14                for (Account nextAccount: theList) {
15                    System.out.println("\n\nnext account..." + nextAccount);
16                }
17            }
18        }
19
20        /**
21         * @param args the command line arguments
22         */
23        public static void main(String[] args) {
24            new ArrayListTest();
25        }
26    }

Lines 10-17: Add three `Accounts` to a `ArrayList` and print them.

Line 24: Create a `ArrayListTest` that will invoke the default constructor and so run the test code in lines 8-18.

Adjourn to the keyboard, input and run this program. It will require that there is an `Account` class in that directory (you could use the `Classmaker` to generate one, or look around and find the one you used before).

That’s all you need to know to use a `ArrayList`. The next example will use a `ArrayList` as the database for a bank simulation.

A Simple Bank Database

According to `dictionary.com`, database means, “A collection of data arranged for ease and speed of search and retrieval”. Thus a database management system, or DBMS, is software that manages a collection of data easily and quickly. Somehow that’s not as impressive sounding as “database management system”. So it goes.
The Database

In the simple Bank program in Chapter 3, the database consisted of three Account variables. There were always exactly three Accounts and the only thing a user could do was select the current account and withdraw money from it. It was hardly a database at all. The bank database here will have a variable number of Accounts. The bank administrator will be able to add, delete, or edit accounts, and then save the changes to disk. The data structure used will be a list of all the Accounts in the Bank. A ArrayList is suitable to implement this list.

Inputting the Database: Load

Assuming there were hundreds or thousands of accounts in a bank, they should not have to be input by hand each time you start the program. Even for a small test bank DBMS, you would not want to type in all the data every time you run the program. Instead, account information should be stored on disk, in files. Program initialization would include inputting the database. An obvious place to input the database would be in the Bank constructor.

File Format

The code to read the data from the file will expect it in a particular order and format. There are many ways to write files, but if they are human readable, then they are easy to maintain (since you can simply edit them!). Thus, MyReader and MyWriter from Chapter 10, are well suited for this job.

It doesn’t really matter what format the data is stored in, but you must decide what that format will be. For simplicity, let’s store the data for each Account on one line; first the name of the person, then their balance, with spaces in between. So, if there were four Accounts, the file might look like:

Amy 17
Zoe 9898
Joe 98
Bea 1000000

If there were more data fields, like an account number, or social security number, address, phone number, ATM number and password; they could be appended. Two fields are enough for illustrative purposes.

Encapsulation! Input in the Account Constructor

The code to open the file and build the database belongs in the Bank class (since that Bank will be working with the database). Conceptually, it will look something like Listing 11.7.

Listing 11.7 Pseudocode for Reading and Building the Database

```
1    while (more data in the input file) {
2        read the data for the next account
3        create and store the new account
```

Your first idea might be to write the inside of the loop as shown in Listing 11.8.
Listing 11.8 First Idea for Inputting and Creating the Accounts in that Loop

// read the data for the next account
StringTokenizer st = new StringTokenizer(mr.giveMeTheNextLine());
String name = st.nextToken();
int balance = Integer.parseInt(st.nextToken());

// create and store the new account
theList.add(new Account(name, balance);

Listing 11.8 is the inside of the loop from Listing 11.7 (the details of lines 2 and 3) with the pseudocode made into comments and the actual code written beneath it. The use of pseudocode as comments for the actual code is good form; you can type the pseudocode right into the class and then comment it out as you implement it. That way the compiler will remind you if you haven’t implemented everything (since the pseudocode will generate compiler errors) and the comments will remind you what you were thinking when you wrote it.

Logically, Listing 11.8 is impeccable (assuming mr is a MyReader and has been properly initialized). But, it violates the principal of encapsulation. What if later, more fields are added to the Account class? Then it would be necessary to edit the Bank class as well. It would decouple Bank and Account better, and be simpler for the programmer adding the fields to Account, if all the changes could be made in Account.

The standard technique to accomplish this is to write a constructor that is passed the input stream, and that reads the data it needs from that, as shown in Listing 11.9.

Listing 11.9 Account(MyReader) Constructor

```
1    Account(MyReader mr){   //empty default constructor
2        StringTokenizer st = new StringTokenizer(mr.giveMeTheNextLine());
3        name = st.nextToken();
4        balance = Integer.parseInt(st.nextToken());
5    }
```

This reads and stores information for one Account from the parameter. By doing the I/O for Account in Account, encapsulation is increased and the programmer’s job is simplified.

To test this code (stepwise implementation!) use the Bank class in Listing 11.10.

Listing 11.10 Testing the Account Constructor that Inputs

```
1    import java.util.*;
2
3    public class Bank {
4        private ArrayList<Account> accountList;
5
6        /** Creates a new instance of Bank */
7        public Bank() {
8            accountList = new ArrayList<Account>();
9            inputAccounts();
10        }
11
12        private void inputAccounts() {
13            MyReader mr = new MyReader();
14            while (mr.hasMoreData())   // read and store database
15                accountList.add(new Account(mr));
16            displayAccounts();
```
private void displayAccounts() {
    for (Account nextAccount: theList)
        System.out.println(nextAccount));
}

Line 15: Reads in the entire file and stores it in the database!
This code reads, stores and displays a database from a user-selected file. It is written simply to
test input, storage and retrieval of a database of Accounts. The displayAccounts() method
was written (instead of just a loop) so it can be reused. Line 15 would also be good to focus on;
this is the line where the database is constructed -- oddly enough. It adds one new Account to the
list, by invoking the Account(MyReader) constructor (which reads the information for this
Account from the file associated with the MyReader, mr). Since it is in a while loop, it will read
all the account information, one Account at a time and store them all in the list, in the same order
as they were in the file. That’s a lot of functionality for one line, and elegantly done (although,
likely aesthetics are personal).

Outputting the Database: Save
To test the save() method (once we have written it!), one line can be added, as shown in Listing
11.11.

Listing 11.11 Testing the Bank: save() Method
1    public BankDBMS() {
2        initComponents();
3        setBounds(100,100,500,500);
4        theBank = new Bank();
5        theBank.save();
6    }

Line 5: Will output the database to a file of the user’s choice.
Can you tell which class this method goes in? If you don’t know, answer this question; “What
type does the BankDBMS() method return?” It is not specified. Ordinary methods must specify a
return type, or void if there is none (see the paragraph titled “Return types”, in Chapter 5). Since
there is no return type, this must be a constructor, and the name of a constructor is the name of
the class it appears in.

Bank: save()
Save seems a good name for a method that saves the database to a file. The Bank must iterate
over the ArrayList that is the internal database and write each Account to the disk file. A
MyWriter will do the job perfectly; see Listing 11.12.

Listing 11.12 Bank:save()
1    public void save() {
2        MyWriter mw = new MyWriter();
3        for (Account nextAccount: theList) {
4            nextAccount.save(mw);
5        }
6    }
mw.close();

Line 2: Creates and saves the MyWriter.

Lines 4-7: This iterates over the account list, sending each Account the save(MyWriter) message.

Line 9: Don’t forget to close the file when done with it!

File Format

The file format is entirely arbitrary, but it must be compatible with the input method. The input method expects first the name and then the balance on one line separated by at least one space.

A common pitfall is to write into the same file you are reading from before the save() code is completely debugged. Then the next time you try to read the data, your program crashes. You can ameliorate this by writing to a different file, or making a backup of the input file (before trashing it!).

Encapsulation: Output in the Account Class

Just as input is best done within the class, so is output. Listing 11.13 is the save() method -- tough work, eh?

Listing 11.13 Account: save(MyWriter)

```java
public void save(MyWriter mw) {
    mw.println(name + " " + balance);
}
```

Not much to it. Simply print the name and balance with a space between them.

This looks (and is) simple, but if a careless programmer wrote it as,

```java
mw.println(name + balance);
```

what would go wrong? Attention to detail while programming will save hours of frustration.

Enhancing the DBMS

We now have a DBMS that loads and saves a list of Accounts. Its only usefulness is to demonstrate that we can do that. There are many things you might do to enhance such a database. The most obvious are to allow the user to select an account and withdraw or deposit funds. Other functions include adding and deleting Accounts, changing information in an Account, and transferring funds between Accounts.

Adding a java.awt.Choice

Back in Chapter 3, you selected the current Account by pushing one of three buttons. If there are dozens or hundreds of Accounts, that would make a very cluttered GUI. A Choice (or, in JFrames, a ComboBox) would be a better choice. As the name implies, it is a Component for allowing the user to make a choice between a number of options. Follow the instructions in the Netbeans Appendix “Adding and using a Choice” to add one to your Application.
Following those instructions will give you a Choice with “this”, “that” and “the other thing” in it. What you want in it for the Bank database is the names from all the Accounts. The easiest place to add them is when you read in the Accounts, in Bank:input(); that means you will need access to the Choice there. The easiest way to have a reference to it there is to pass it as a parameter with the Bank constructor (see Listing 11.14). Remember, the Bank class will not know what Choice is unless import java.awt.*; is added. Notice that the reference to the Choice is passed along as a parameter to inputAccounts().

Selecting an Account Given a Name

Like in the Chapter 3, example, Bank will have a withdraw(int) method that withdraws the amount passed in the parameter from the current account. This will require that there is a variable containing a reference to the current Account; so it must be declared and initialized. It cannot be initialized until after the database is read in, so the logical spot to do that is right after input; see Listing 11.14.

Listing 11.14 Initializing theChoice and currentAccount

```java
import java.util.*;
import java.awt.*;

public class Bank {
    private ArrayList<Account> accountList;
    private Account currentAccount;

    /** Creates a new instance of Bank */
    public Bank(Choice theChoice) {
        accountList = new ArrayList<Account>();
        inputAccounts(theChoice);
        currentAccount = accountList.get(0);
    }

    private void inputAccounts(Choice theChoice) {
        MyReader mr = new MyReader();
        while (mr.hasMoreData()) {
            Account newAccount = new Account(mr);
            accountList.add(newAccount);
            theChoice.addItem(newAccount.getName());
        }
        mr.close();
    }

    public void withdraw(int withdrawalAmt) {
        currentAccount.withdraw(withdrawalAmt);
    }
}
```

Line 11: Fills the list with Accounts and initializes theChoice (see line 20).

Line 12: Initializes currentAccount to the first thing in the ArrayList.

The get(int) method returns the Object at the ith position in the ArrayList. where i is the value of the parameter. The first position in the ArrayList is 0, just like an array.

When the user selects a new name from the Choice, presumably the code looks something like Listing 11.15.
Listing 11.15 itemSelected Handler for theChoice
1   private void newChoice(java.awt.event.ItemEvent evt) {
2       String item = theChoice.getSelectedItem();
3       System.out.println("new choice from the choice: "+ item);
4       theBank.setCurrentAccount(item);
5       displayCurrentBalance();
6   }

Line 2: Get the selected item from theChoice.
Line 3: Diagnostic; delete when the code works.
Line 4: The point of this example; set the currentAccount variable in theBank.
Line 5: So the user can always see the balance of the current Account.

Notice that the type of item is String, so on the Bank side the setCurrentAccount() method will have a String parameter and will have to search through all the Accounts looking for one with that name. As always, to iterate through a ArrayList, copy and paste the for-each loop; see Listing 11.16.

Listing 11.16 Bank:setCurrentAccount(String)
1   public void setCurrentAccount(String name) {
2       for (Account nextAccount: accountList {  
3           if (nextAccount.getName().equals(name)) {
4               currentAccount = nextAccount;   
5               return;                      
6           } // if
7       } // for
8   }

Lines 2-6: Iterate over the Accounts, and if one is found with the parameter as its name, set currentAccount to it.
Line 4: Note that you cannot compare Strings with ==. You must use equals().

This code will work, but could be improved in two ways. First, the loop will continue through the entire list even if it finds the name in the first Account; so it will waste fewer cycles if it exits the loop as soon as it finds the name (although, since searching the entire list takes less than a millisecond, this hardly matters). Second, if it never finds the name, that’s a bug, and it would be good to report the problem, instead of blithely ignoring it. Listing 11.17 fixes both of these by returning when the name is found and complaining if control falls out the bottom of the loop without having found the name. This loop with an internal return is an idiom that is worthwhile remembering; you will see it again (assuming you continue on in computing!).

Listing 11.17 Bank:setCurrentAccount(String) Improved
1   public void setCurrentAccount(String name) {
2       for (Account nextAccount: accountList {
3           if (nextAccount.getName().equals(name)) {
4               currentAccount = nextAccount; // found it
5               return; // exit the method NOW!
6           } // if
7       } // for
8       System.out.println("Error! Name not found! name="+ name);
9   }

Line 4-7: If the name is found, setCurrentAccount and exit the method.
Line 10: Complain about not finding it; realize it will only get here if line 6 never executes.
**Editing an Account**

Part of database management is the ability to modify the data, both to correct errors, and simply to update changing information. This may be done by the software (like updating the balance when money is withdrawn) or by hand (like changing a misspelled name). Conceptually here’s what to do to allow editing.

1. Add an Edit button (it’s action is to tell the Bank to edit).
2. Open a new window to edit the current account.
3. Reflect changes the user makes in the edit window in the database.

Step 1 is easy, you know how to add and connect a Button. The other two steps need some explaining.

*Opening a New Window*

Opening a new window is very useful when you wish to present or input information under certain circumstances, but don’t want to clutter up the GUI. Displaying a new Frame is described in the Netbeans Appendix “Adding a pop-up Frame”.

*Editing from the New Window*

Once you have created a GUI Frame Form, you can use the FormEditor to add TextFields for each field in the Account to be edited; as of now that would just be name and balance. Rename those TextFields and connect them so that you get control when the user hits Enter in either.

NetBeans will write the shell of the event handling code, but you must write the code to do the editing. When the user enters a change in the name TextField, you would like to simply set the name of the current account to what they have entered, as shown in Listing 11.18.

**Listing 11.8 Editing the name Field**

```java
private void nameTFActionPerformed(java.awt.event.ActionEvent evt) {
    theAccount.setName(nameTF.getText());
}
```

**Line 2:** Change the value of the name in theAccount.

But, how to access the currentAccount back in the Bank from the EditFrame? If you simply type this code, the compiler will inform you that it cannot resolve the symbol “theAccount”, and for good reason; it is not declared in this class! You could declare it as an instance variable (Account theAccount;), and then it would compile; unfortunately at runtime this, alone, would generate a NullPointerException -- again for good reason, since you have never changed the default null. Some people find themselves stuck at this point.

The solution is obvious if you think carefully. Or maybe draw a picture? Figure 11.1 depicts the data structure after the program has read in those four Accounts from before and the user has selected “Zoe” with the Choice and pressed the Edit Button.
The Application has a variable named theBank, which contains a database of Accounts, a current account, and an EditFrame. When the user pushes the Edit Button, an EditFrame is opened; but, how can the EditFrame get access to the current Account?

The problem is that theAccount in theEditFrame is null. It should have been set to point to Zoe’s Account, which is the currentAccount back in theBank. How could this information be passed from theBank to theEditFrame? You already know, right? Either with an accessor, or as a parameter to the constructor. That easy.
Once we have access to the current account from the edit window, Listing 11.18 will work perfectly. The code to update the balance is similar, except that the String from the TextField must be converted to an int (see the paragraph titled “String to int” in Chapter 5, if it has slipped your mind).

Error Checking

The code above assumes that the user will type a legal int into the balanceTF in the EditFrame. What if they accidentally type some letters? Try it out. It throws an Exception. It would be good to catch that Exception. It would also be nice to read the values from both textFields whenever the user hits Enter (imagine how annoying it would be to type a new name an balance, hit Enter, and only update the balance). But, that is deferred to the exercises; it’s just details.

Molecules in Box

In Chapter 9, a single ball bouncing under the influence of gravity was simulated. The task for this section is to simulate a number of molecules in a box. Most of the code from the BallFrame can be leveraged for this new task.

The Molecule Class

The Ball class checked for bouncing on the floor, but not on the ceiling or the two side walls. Thus, the Molecule class will need to check for those other three cases as well. Perhaps you are ready to write that code now; in that case, do it, and when you’re done go on to “Changes to the Controller”, later in this chapter. Otherwise read on.

Designing the Bounce Code

The code to check whether a Molecule will collide with any of the four walls on this time step is a little bit complicated. Whenever you are writing code that is not simple, it is important to think clearly before starting, otherwise you can waste many frustrating hours debugging, sometimes with nothing to show for it when you are done.

First, you need a clear conception of the problem. Second, you need to formulate a simple way to solve the problem. Start with a simple approach that you understand, or writing and debugging the code is likely to be a disaster.

Two problem solving techniques spring to mind here: Draw a Picture, and Analysis By Cases; if those don’t seem familiar, you might want to review them before continuing (see “Draw a Picture” in Chapter 4, “Graphics and Inheritance”, or “Analysis By Cases (ABC)” in Chapter 7, “Conditional Statements”). Then, using the picture for the bottom as a model (see Figure 9.1), draw the pictures for the top, right and left.

Applying the Analysis By Cases Technique

Recall the steps in the ABC technique; first you must decide what the cases are, then how to distinguish between them, and finally, what to do in each case.


**Step 1 -- Distinguishing the Cases**

Use the pictures you have drawn for the first step. Once you have decided how to determine which of the five cases is applicable, then you can decide what to do in each case. Only then should you start writing code.

Are there really five cases? At least. The molecule may bounce on any of the four walls this time step, or none. That makes five cases. Perhaps there are four more cases: bouncing off the left and top, the left and bottom, right and top, and right and bottom. It is possible to check the four bounce cases so as to handle all eight cases; nevertheless it is something to keep in mind.

**Step 2 -- Deciding on Actions for Each Case**

The action required in each case is similar. If the molecule would hit the wall this time step, reverse its velocity (x or y, depending) and calculate its position after the bounce.

**Writing the Code**

You could write the code from scratch, but it would be much faster and easier to adapt the code you already have.

**Adapting the Ball Bounce Code**

You already have code for the floor bounce (see Listing 11.19).

**Listing 11.19 The Ball:step() Improved?** (Copied from: Listing 9.12)

```java
1    public void step() {
2        x += vx;
3        int bottomY = BallFrame.HT -(radius+y);
4        if (vy >= bottomY) {
5            //System.out.println("vy=" + vy + " y=" + y);
6            int bounceHt = vy - bottomY;    // how high it bounces this step
7            y = BallFrame.HT - (radius + bounceHt); // y after this step
8            vy = -(vy-1)*9/10;
9        }
10        else {
11            y += vy;
12            vy += GRAVITY;
13        }
14    }
```

**Line 7:** Calculate how far it will travel up after bouncing.

**Line 8:** From that calculate the new y-coordinate of the center.

**Line 9:** Reverse vy and subtract 1 to avoid endless small bounces.

Odds are you can adapt this code for the other three cases. Take a minute to review this code and get it in your head so you’ll know how to modify it... Okay. Give it one more try... Huh. Maybe you can’t make head nor tail of that code. Not good. Too much tricky code. Sorry about that. Perhaps the way to make it make sense of it is by using stepwise refinement (see “Stepwise refinement” in Chapter 3).
Stepwise Refining the Bounce Code

Let’s start with the `step()` method for `Molecule` without worrying about whether the `Molecule` will hit the wall. It simply updates the `x` and `y`-coordinates and the `y` velocity of the `Molecule`, as shown in Listing 11.20.

Listing 11.20 Molecule: `step()` Without Considering Bounces

```java
1    public void step() {
2        x += vx;
3        y += vy;
4        vy += g;
5    }
```

**Line 2:** Add the `x`-velocity, `vx`, to the `x`-position, `x`; this moves the simulated ball the distance in one time step that it would move in the world.

**Line 3:** The same for the `y`-position.

**Line 4:** Adjust the `y`-velocity by the acceleration, `g`.

One of the reasons `step()` in the previous example is complicated is that the code for `step()` and `bounce()` is intertwined. These two parts can be separated as shown in Listing 11.21.

Listing 11.21 Molecule: `step()` Considering Bounces

```java
1    public void step() {
2        if (!willHitWall()) {
3            x += vx;
4            y += vy;
5            vy += GRAVITY;
6        }
7        else bounce();
8    }
```

Here the use of methods with descriptive names makes the logic clear. The downside is that now the `willHitWall()` and `bounce()` methods must be written. Listing 11.22 shows those two methods.

Listing 11.22 `willHitWall()` and `bounce()`

```java
1    private boolean willHitWall() {
2        return willHitSide() || willHitTopOrBottom();
3    }
4
5    private void bounce() {
6        if (willHitTopOrBottom())
7            handleYBounce();
8        if (willHitSide())
9            handleXBounce();
10    }
```

**Lines 1-3:** `willHitWall()` returns `true` just if `willHitSide()` or `willHitTopOrBottom` is true.

**Lines 5-10:** Handles a `y`-bounce, or an `x`-bounce, or both, in that order.

Notice that both of them depend on `willHitSide()` and `willHitTopOrBottom()`. The `Molecule` will hit a wall just if it hits a side wall or the top or bottom; thus `willHitWall()` returns that. The `bounce()` method may have to handle a bounce in the `y`-direction or the `x`-direction, or both. If it were written as an `if-else`, it would fail in the cases where the `Molecule` was going to hit two walls on the same time step.
There are now four more methods that need to be written (stepwise refinement tends to generate
a number of small methods). They are shown in the next two listings. Listing 11.23 has the two
methods that detect upcoming bounces.

Listing 11.23 Checking for Upcoming Bounces

```java
1    private boolean willHitTopOrBottom() {
2        if (vy > 0) {
3            return vy > distanceToBottom();
4        }
5        else {
6            return -vy > distanceToTop();
7        }
8    }
9
10   private boolean willHitSide() {
11        if (vx > 0) {
12            return vx > distanceToRightWall();
13        }
14        else {
15            return -vx > distanceToLeftWall();
16        }
17    }
```

The logic is the same for both, so only the first of the four will be explained. If the $y$-component
of velocity is downwards (i.e. if $vy > 0$, see line 2) then it would hit the bottom in the next time
step just if the distance it will move in the $y$ direction ($vy$) is more than the distance to the bottom
(line 3). The distances are calculated by the methods in Listing 11.24.

Listing 11.24 Calculating the Distance to the Wall the Molecule is Headed Towards

```java
1    private int distanceToBottom() {
2        return HT-(radius+y);
3    }
4
5    private int distanceToTop() {
6        return y-radius;
7    }
8
9    private int distanceToRightWall() {
10       return WIDTH - (x+radius);
11    }
12
13    private int distanceToLeftWall() {
14       return x-radius;
15    }
```

These four methods implement the four cases of distance to walls.

Refer to your pictures to make sure these make sense. You will notice the use of WIDTH and HT in
these methods. You no doubt recall that identifiers in all caps are constants (see the paragraph
titled “Case conventions”, in Chapter 5, “Towards Consistent Classes”). It is better to define
variables for values like this to avoid embedding numbers in the. The reason is, if you change the
size of the display, it will be done automatically (instead of having to search for the numbers that
represent the size of the display in various classes. This is accomplished as shown in Listing
11.25.

Page 17 of 23
Listing 11.25 Setting and Accessing the Display Dimensions
1 // from MoleculeFrame.java
2 public class MoleculeFrame extends Frame {
3        Controller theController;
4        public static final int WIDTH=900;
5        public static final int HT=900;
6    }
7
8 // from Molecule.java...
9        public class Molecule extends FilledCircle {
10        private int HT=MoleculeFrame.HT;
11        private int WIDTH=MoleculeFrame.WIDTH;
12        private double ELASTICITY=1.0;
13    }

Lines 10-11: WIDTH and HT are declared static and public in MoleculeFrame, so they are accessible from everywhere, including the Molecule class. These variables are not strictly necessary; the width and height of the frame could be accessed directly using MoleculeFrame.WIDTH, and MoleculeFrame.HT, but that tends to clutter up the code since it they are accessed repeatedly.

The only remaining methods are handleXBounce() and handleYBounce(). The logic of these two methods is a little tricky. The author got it wrong twice and spent more hours than he wants to admit to to get it right. There are still four cases to consider (two in each method). The logic for calculating the new values for x and y is shown in Listing 11.26.

Listing 11.26 Pseudocode for Handling Bounces
1    handleYBounce() {
2        case 1: bottom -- y = HT - reboundDistance() - radius;
3        case 2: top    -- y = reboundDistance() + radius;
4    }
5    handleYBounce() {
6        case 3: right -- x = WIDTH - reboundDistance() - radius;
7        case 4: left  -- x = reboundDistance() + radius;
8    }

Logically, if it hits the top or the top or the left the new position is just the distance it rebounds plus the radius. In the other two cases the rebound distance and the radius must be subtracted from the wall position.

The only remaining question is how to calculate the distance the Molecule rebounds on this time step. If you care about that detail, it is included in Listing 11.27.

Listing 11.27 handleYBounce() and handleXBounce()
1    private void handleYBounce() {
2        if (vy >= distanceToBottom()) {
3            y = HT - (vy - distanceToBottom()) - radius;
4            vy = - (vy-1)*9/10;
5        }
6        else { // top
7            y = -vy -(distanceToTop() - radius) + radius;
8            vy = -vy * 9/10;
9        }
10    }
11
12    private void handleXBounce() {
if (vx > 0) // right wall
    x = WIDTH - (vx - (WIDTH - x - radius)) - radius;
else x = -vx - (x-radius) + radius;

vx = -vx * 9/10;

This is the detailed code to handle bounces; finally the stepwise refinement bottoms out!. To understand this code, draw a careful picture. Or ignore it, it’s hardly pivotal to understanding computing.

The 9/10s should be ELASTICITY, but if it is declared as:

    int ELASTICITY = 9/10;
something terrible happens. Do you know what? See Code Example 5.16 for a clue. And if it is declared as

    double ELASTICITY = 0.9;
then it will not compile unless vx * ELASTICITY is cast as an int; like this:

    vx = (int) (-vx * ELASTICITY);
Which clutters up the code (although it would be okay to do).

Finally we are done writing Molecule and can turn to testing it.

Changes to the Controller

There are several changes to make to the Controller.

A Single Molecule

To make the Controller from the BallFrame simulate and animate a single Molecule instead of a Ball is very simple. The changes needed are shown in Listing 11.28.

Listing 11.28 Modified Controller from the BallFrame

```java
import java.awt.*;
import java.applet.*;

public class Controller extends Thread {
    private Frame theFrame;
    private Molecule theMolecule;
    private boolean running=true;

    /** Creates a new instance of Controller */
    public Controller(Frame theFrame) {
        this.theFrame = theFrame;
        theMolecule = new Molecule(10,40,20,Color.RED, 1, -5);
    }

    public void paint(Graphics g) {
        theMolecule.paint(g);
    }

    private void step() {
        theMolecule.step();
        theFrame.repaint();
    }
}
This **Controller** is very similar to the one in Listing 9.7; the differences are that every **Ball** has been replaced by a **Molecule**.

All that has been done is to replace **Ball** with **Molecule** everywhere. The methods that are unaffected are omitted.

**Many Molecules**

To make the **Controller** simulate and animate many **Molecules** is slightly more complicated. Instead of a **Molecule** variable, it needs a **ArrayList** (which will contain all the **Molecules**). Then, everywhere the original code did something with a **Molecule**, the new code must iterate over all the **Molecules**. There are three methods that need to be changed, the constructor, **paint()** and **step()**.

**Changes to the Constructor**

Listing 11.29 shows the changes needed to create and store **NUM_MOLECULES** **Molecules** in a **ArrayList**.

**Listing 11.29 Changes to the Constructor**

```java
public Controller(Frame theFrame) {
    this.theFrame = theFrame;
    for (int i=0; i<NUM_MOLECULES; i++) {
        addOneMolecule();
    }
}

private void addOneMolecule() {
    theList.add(new Molecule(100,100,20,Color.RED, rand(7), rand(7)));
}
```

**Line 3:** Loop **NUM_MOLECULES** times (if you have questions about this, see Listing 8.6).

**Line 4:** Each time around the loop, uh, add one **Molecule**.

**Lines 8-10:** The **addOneMolecule()** method.

**Line 9:** Create a red **Molecule** at 100, 100, radius 20, with random **vx** and **vy** (between 0 and 6).

Notice that they all start at the same place, are the same color and size, but have random velocities. The use of **rand(int)** requires that it be defined (see Code Example 5.19 for a reminder). You might want to experiment with random sizes and colors once the code is working.

This code assumes that **NUM_MOLECULES** and **theList** are defined, and that **java.util.*** is imported: see Listing 11.30 for the changes needed at the beginning of **Controller**.

**Listing 11.30 Changes to the Beginning of Controller**

```java
import java.awt.*;
import java.applet.*;
import java.util.*;

public class Controller extends Thread {
    private final int NUM_MOLECULES=20;
```
Changes to **paint() and step()**

The `paint()` method for the bouncing ball had just one line:

```
    theMolecule.paint(g);
```

To make it instead send `paint(g)` to every `Molecule` in the list requires an `Iterator` in a loop, as shown in Listing 11.31.

**Listing 11.31 Modified `paint()` to Paint Every `Molecule` in the List**

```
1    public void paint(Graphics g) {
2        for (Iterator it=theList.iterator(); it.hasNext();) {
3            Molecule nextMolecule = (Molecule) it.next();
4            nextMolecule.paint(g);
5        }
6    }
```

This uses the idiom for iterating over all elements of a list (see “The `iterator()` method”, earlier in this chapter).

The changes required for `step()` are very similar; instead of sending `step()` to the `Molecule`, it must be sent to every `Molecule` in the list; see Listing 11.32.

**Listing 11.32 Modified `step()` to Step Every `Molecule` in the List**

```
1    private void step() {
2        for (Iterator it=theList.iterator(); it.hasNext();) {
3            Molecule nextMolecule = (Molecule) it.next();
4            nextMolecule.step();
5        }
6        theFrame.repaint();
7    }
```

Like the previous Listing, this again uses the idiom for iterating over the elements of a list.

**Experimenting with the Program**

Having made those changes, the program is ready to run. Experiment with different elasticities, turning off gravity, different delay times, different ranges of velocities, or different numbers of molecules. How many molecules can there be before it stops looking like animation? Note: if you wish, you may access the code at:

```
http://www.willamette.edu/~levenick/SimplyJava/code/molecules/
```

It would be more educational to write it yourself, but if you don’t have the time or inclination, experimenting with the code provided is much better than nothing.
Conclusion

This chapter introduced `ArrayList` from `java.util`, `Choice` and pop-up Frames from `java.awt`, and techniques to read/write a database from/to disk files. It included a lot of new material and combined most of the material from the previous chapters to make two substantial programs. If you understand both of those examples well, congratulations! If not, condolences; you might consider rereading and working through the chapter again. Or, perhaps, doing the exercises will help solidify your understanding.

Review Questions

11.1 How do you add an `Object` to the end of the list in a `ArrayList`?
11.2 Write the idiom to access every `Object` in a `ArrayList` and send it to `System.out`.
11.3 How were the 9 ways a `Molecule` could bounce collapsed into 5? What are those 5?

Programming Exercises

The next exercises refer to the `BankDBMS`:

11.4 Add a Save button. Save to a file the user specifies. Then (after that works) save to the same file that the database was input from. Hint: save the file name and path when it is input; one `String` variable will do it.
11.5 Modify the `EditFrame` code so that both the name and balance fields are input and both the name and balance are updated when the user hits Enter in either. Hint: write an `update()` method that is invoked from both.
11.6 Add error checking code so that if the entered balance is not an `int` the user is notified in a reasonable manner; maybe pop up a `Panel`?
11.7 Add an `addAccount` button. You could reuse the `EditFrame` class to get the info for the new `Account`.
11.8 Add a `deleteAccount` button. You will find the `ArrayList:remove(Object)` method very useful here; as in `theAccountList.remove(currentAccount)`; Don’t forget to also remove the deleted account’s name from the `Choice`. Reading the Sun documentation will help with both of these.

The next exercises refer to the `Molecules` program:

11.9 Take out the -1 in `(vy-1)` in `handleYBounce()` in Listing 11.27. What happens? Explain why. Hint: add debugging `printlns` to display `vy` and `y` at each step.
11.10 Add code to catch mouse clicks and when the user clicks on a `Molecule`, make it bigger. See the Netbeans Appendix, “Catching mouse clicks”.
11.11 Start with just two large molecules and color them solid red when they are overlapping. Hint: two circles overlap when the distance between their centers is less than the sum of their radii.
11.12 Modify your code to detect overlap in a simulation with `n` `Molecules`. Hint: you will need a loop that checks every pair of `Molecules` for overlap.
11.13 Difficult! Modify your code to keep the Molecules from overlapping (it is the same kind of logic as anticipating bounces off the wall. If two Molecules are going to overlap, reverse the signs on their $vx$ and $vy$ variables.

11.14 More difficult! Make the collisions realistic, as if the molecules were billiard balls. If two Molecules hit head on, they should rebound the way they came, but if they hit a glancing blow, their directions change in a more complicated way. Don’t forget to conserve momentum! Note: the mass of the Molecules matters -- do large Molecules have more mass?

The next exercises refer to a completely new different project

11.15 Fun! Create a crowd of SnowPersons. Make them all melt each day. The control structure should be just like the Molecules program.

11.16 More fun! Create a crowd of SnowPersons. Then, when the user clicks on a particular SnowPerson, make all the rest of the army move to surround that one.