

Chapter 5

Arrays

5.1 Creating and Using Arrays

- A collection of data items stored under a single name is known as a ***data structure***.
- An object is one kind of data structure, because it can store multiple data items in its instance variables.

One-Dimensional Arrays

- In an array, all data items (known as ***elements***) must have the same type.
- An array can be visualized as a series of boxes, each capable of holding a single value belonging to this type:



- An array whose elements are arranged in a linear fashion is said to be ***one-dimensional***.

Creating Arrays

- An array declaration contains [and] symbols (“square brackets”):
`int[] a;`
- The elements of an array can be of any type. In particular, array elements can be objects:
`String[] b;`
- It’s also legal to put the square brackets after the array name:
`int a[];`
`String b[];`

Creating Arrays

- Declaring an array variable doesn’t cause Java to allocate any space for the array’s elements.
- One way to allocate this space is to use the new keyword:
`a = new int[10];`
- Be careful not to access the elements of an array before the array has been allocated. Doing so will cause a `NullPointerException` to occur.

Creating Arrays

- It’s legal to allocate space for the elements of an array at the same time the array is declared:
`int[] a = new int[10];`
- The value inside the square brackets can be any expression that evaluates to a single `int` value:
`int n = 10;`
`int[] a = new int[n];`

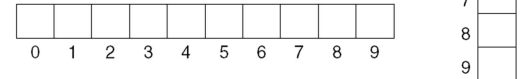
Creating Arrays

- An array can be initialized at the time it's declared:

```
int[] a = {3, 0, 3, 4, 5};
```
- The word `new` isn't used if an initializer is present.
- When an array is created using `new`, the elements of the array are given default values:
 - Numbers are set to zero.
 - `boolean` elements are set to `false`.
 - Array and object elements are set to `null`.

Visualizing Arrays

- Each array element has an **index**, or **subscript**, that specifies its position within the array.
- If an array has n elements, only the numbers between 0 and $n - 1$ are valid indexes.
- In an array with 10 elements, the elements would be numbered from 0 to 9:



Array Subscripting

- **Subscripting** is the act of selecting a particular element by its position in the array.
- If `a` is an array, then `a[i]` represents the i th element in `a`.
- An array subscript can be any expression, provided that it evaluates to an `int` value.
- Examples of subscripting:

```
a[0]
a[i]
a[2*i-1]
```

Array Subscripting

- Attempting to access a nonexistent array element causes an error named `ArrayIndexOutOfBoundsException`.
- An array element behaves just like a variable of the element's type:

```
a[i] = 10;
System.out.println(a[i]);
```

Array Subscripting

- If the elements of an array are objects, they can call instance methods.
- For example, if the array `b` contains `String` objects, the call `b[i].length()` would return the length of the string stored in `b[i]`.

Processing the Elements in an Array

- If it becomes necessary to visit all the elements in an array, a counting loop can be used:
 - The counter will be initialized to 0.
 - Each time through the loop, the counter will be incremented.
 - The loop will terminate when the counter reaches the number of elements in the array.
- The number of elements in an array `a` is given by the expression `a.length`.

Processing the Elements in an Array

- A loop that adds up the elements in the array `a`, leaving the result in the `sum` variable:

```
int sum = 0;
int i = 0;
while (i < a.length) {
    sum += a[i];
    i++;
}
```

Processing the Elements in an Array

- The `while` loop could have been written differently:


```
while (i < 3) {
    sum += a[i];
    i++;
}
```
- The original test (`i < a.length`) is clearly better: if the length of `a` is later changed, the loop will still work correctly.

Program: Computing an Average Score

- The `AverageScores` program computes the average of a series of scores entered by the user:

Enter number of scores: 5

Enter score #1: 68
Enter score #2: 93
Enter score #3: 75
Enter score #4: 86
Enter score #5: 72

Average score: 78

- The first number entered by the user is used to set the length of an array that will store the scores.

AverageScores.java

// Computes the average of a series of scores

`import java.util.Scanner;`

```
public class AverageScores {
    public static void main(String[] args) {
        // Prompt user to enter number of scores
        System.out.print("Enter number of scores: ");
        Scanner input = new Scanner(System.in);
        int numberOfScores = input.nextInt();
        System.out.println();

        // Create array to hold scores
        int[] scores = new int[numberOfScores];
```

```
// Prompt user to enter scores and store them in an array
int i = 0;
while (i < scores.length) {
    System.out.print("Enter score #" + (i + 1) + ": ");
    scores[i] = input.nextInt();
    i++;
}

// Compute sum of scores
int sum = 0;
i = 0;
while (i < scores.length) {
    sum += scores[i];
    i++;
}

// Display average score
System.out.println("\nAverage score: " +
    sum / scores.length);
}
```

5.2 The `for` Statement

- The `AverageScores` program contains two loops with the same general form:

```
i = 0;
while (i < scores.length) {
    ...
    i++;
}
```

- Because this kind of loop is so common, Java provides a better way to write it:

```
for (i = 0; i < scores.length; i++) {
    ...
}
```

- We can also use enhanced `for` loop for arrays

```
for (int aScore : scores) {
    ...
}
```

in which each score in `scores` is assigned to `aScore` exactly once.

General Form of the **for** Statement

- Form of the **for** statement:

```
for ( initialization ; test ; update )
    statement
```
- Initialization** is an initialization step that's performed once, before the loop begins to execute.
- Test** controls loop termination (the loop continues executing as long as *test* is true).
- Update** is an operation to be performed at the end of each loop iteration.

for Statements Versus **while** Statements

- Except in a couple of cases, a **for** loop of the form

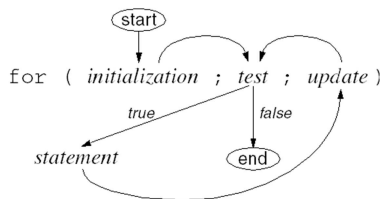
```
for ( initialization ; test ; update )
    statement
```

 is equivalent to the following **while** loop:

```
initialization ;
while ( test ) {
    statement
    update ;
}
```

for Statements Versus **while** Statements

- Flow of control within a **for** statement :



for Statements Versus **while** Statements

- A **for** statement:

```
for ( i = 10; i > 0; i-- )
    System.out.println("T minus " + i +
        " and counting");
```
- An equivalent **while** statement:

```
i = 10;
while ( i > 0 ) {
    System.out.println("T minus " + i +
        " and counting");
    i--;
}
```

for Statements Versus **while** Statements

- Studying the equivalent **while** statement can help clarify the fine points of a **for** statement.
- A **for** loop that uses **--i** instead of **i--**:

```
for ( i = 10; i > 0; --i )
    System.out.println("T minus " + i +
        " and counting");
```
- An equivalent **while** loop:

```
i = 10;
while ( i > 0 ) {
    System.out.println("T minus " + i +
        " and counting");
    --i; // Same as i--;
}
```

for Statements Versus **while** Statements

- Advantages of the **for** statement versus the **while** statement:
 - More concise
 - More readable
- The first **while** loop from the **AverageScores** program:

```
int i = 0;
while ( i < scores.length ) {
    System.out.print("Enter score #" + (i + 1) + ": ");
    scores[i] = input.nextInt();
    i++;
}
```

for Statements Versus while Statements

- The corresponding for loop:

```
for (i = 0; i < scores.length; i++) {
    System.out.print("Enter score #" + (i + 1) + ": ");
    scores[i] = input.nextInt();
}
```

- All the vital information about the loop is collected in one place, making it much easier to understand the loop's behavior.

for Statement Idioms

- for statements are often written in certain characteristic ways, known as *idioms*.
- An idiom is a conventional way of performing a common programming task.
- For any given task, there are usually a variety of different ways to accomplish the task.
- Experienced programmers tend to use the same idiom every time.

for Statement Idioms

- Typical ways to write a for statement that counts up or down a total of n times:

Counting up from 0 to $n - 1$:

```
for (i = 0; i < n; i++) ...
```

Counting up from 1 to n :

```
for (i = 1; i <= n; i++) ...
```

Counting down from $n - 1$ to 0:

```
for (i = n - 1; i >= 0; i--) ...
```

Counting down from n to 1:

```
for (i = n; i >= 1; i--) ...
```

for Statement Idioms

- A for statement's *initialization*, *test*, and *update* parts need not be related.
- A loop that prints a table of the numbers between 1 and n and their squares:

```
i = 1;
odd = 3;
for (square = 1; i <= n; odd += 2) {
    System.out.println(i + " " + square);
    i++;
    square += odd;
}
```

Omitting Expressions in a for Statement

- The three parts that control a for loop are optional—any or all can be omitted.
- If *initialization* is omitted, no initialization is performed before the loop is executed:

```
i = 10;
for (; i > 0; i--)
    System.out.println("T minus " + i +
        " and counting");
```

- If the *update* part of a for statement is missing, the loop body is responsible for ensuring that the value of *test* eventually becomes false:

```
for (i = 10; i > 0;)
    System.out.println("T minus " + i-- +
        " and counting");
```

Omitting Expressions in a for Statement

- When both *initialization* and *update* are omitted, the loop is just a while statement in disguise.
- For example, the loop

```
for (; i > 0;)
    System.out.println("T minus " + i-- +
        " and counting");
```

is the same as

```
while (i > 0)
    System.out.println("T minus " + i-- +
        " and counting");
```

- The while version is preferable.

Omitting Expressions in a `for` Statement

- If the *test* part is missing, it defaults to `true`, so the `for` statement doesn't terminate (unless stopped in some other fashion).
- For example, some programmers use the following `for` statement to establish an infinite loop:

```
for (;;) ...
```
- The `break` statement can be used to cause the loop to terminate.

Declaring Control Variables

- For convenience, the *initialization* part of a `for` statement may declare a variable:

```
for (int i = 0; i < n; i++)
```



```
...
```
- A variable declared in this way can't be accessed outside the loop. (The variable isn't *visible* outside the loop.)
- It's illegal for the enclosing method to declare a variable with the same name. It is legal for two `for` statements to declare the same variable, however.

Declaring Control Variables

- Having a `for` statement declare its own control variable is usually a good idea. It's convenient, and it can make programs easier to understand.
- More than one variable can be declared in initialization, provided that all variables have the same type:

```
for (int i = 0, j = 0; i < n; i++)
```



```
...
```

Commas in `for` Statements

- In a `for` statement, both *initialization* and *update* are allowed to contain commas:

```
for (i = 0, j = 0; i < n; i++, j += i)
```



```
...
```
- Any number of expressions are allowed within *initialization* and *update*, provided that each can stand alone as a statement.
- When expressions are joined using commas, the expressions are evaluated from left to right.
- Using commas in a `for` statement is useful primarily when a loop has two or more counters.

Program: Computing an Average Score (Revisited)

AverageScores2.java

```
// Computes the average of a series of scores

import java.util.Scanner;

public class AverageScores2 {
    public static void main(String[] args) {
        // Prompt user to enter number of scores
        System.out.print("Enter number of scores: ");
        Scanner input = new Scanner(System.in);
        int numberOfScores = input.nextInt();
        System.out.println();

        // Create array to hold scores
        int[] scores = new int[numberOfScores];
```

```
// Prompt user to enter scores and store them in an array
for (int i = 0; i < scores.length; i++) {
    System.out.print("Enter score #" + (i + 1) + ": ");
    scores[i] = input.nextInt();
}

// Compute sum of scores
int sum = 0;
for (int i = 0; i < scores.length; i++)
    sum += scores[i];

// Display average score
System.out.println("\nAverage score: " +
    sum / scores.length);
}
```

5.3 Accessing Array Elements Sequentially

- Many array loops involve *sequential access*, in which elements are accessed in sequence, from first to last.

Searching for a Particular Element

- One common array operation is searching an array to see if it contains a particular value:

```
int i;
for (i = 0; i < scores.length; i++)
    if (scores[i] == 100)
        break;
```

- An `if` statement can be used to determine whether or not the desired value was found:

```
if (i < scores.length)
    System.out.println("Found 100 at position " + i);
else
    System.out.println("Did not find 100");
```

Counting Occurrences

- Counting the number of times that a value occurs in an array is similar to searching an array.
- The difference is that the loop increments a variable each time it encounters the desired value in the array.
- A loop that counts how many times the number 100 appears in the `scores` array:

```
int count = 0;
for (int i = 0; i < scores.length; i++)
    if (scores[i] == 100)
        count++;
```

Finding the Largest and Smallest Elements

- Finding the largest (or smallest element) in an array is another common operation.
- The strategy is to visit each element in the array, using a variable named `largest` to keep track of the largest element seen so far.
- As each element is visited, it's compared with the value stored in `largest`. If the element is larger than `largest`, then it's copied into `largest`.

Finding the Largest and Smallest Elements

- A first attempt at writing a loop to find the largest element in the `scores` array:

```
for (int i = 0; i < scores.length; i++)
    if (scores[i] > largest)
        largest = scores[i];
```

- One question remains: What should be the initial value of `largest`, before the loop is entered?
- It's tempting to choose 0, but that doesn't work for all arrays.

Finding the Largest and Smallest Elements

- The best initial value for `largest` is one of the elements in the array — typically the one at position 0.

- The finished loop:

```
int largest = scores[0];
for (int i = 1; i < scores.length; i++)
    if (scores[i] > largest)
        largest = scores[i];
```

Notice that the loop now initializes `i` to 1 rather than 0.

Finding the Largest and Smallest Elements

- Assume that the `scores` array contains the values 61, 78, 55, 91, and 72.
- A table showing how the variables change during the execution of the loop:

	<i>Initial value</i>	<i>After iteration 1</i>	<i>After iteration 2</i>	<i>After iteration 3</i>	<i>After iteration 4</i>
<code>largest</code>	61	78	78	91	91
<code>i</code>	1	2	3	4	5
<code>scores[i]</code>	78	55	91	72	–

Finding the Largest and Smallest Elements

- The technique for finding the smallest element in an array is similar:

```
int smallest = scores[0];
for (int i = 1; i < scores.length; i++)
    if (scores[i] < smallest)
        smallest = scores[i];
```

5.4 Accessing Array Elements Randomly

- One of the biggest advantages of arrays is that a program can access their elements in any order.
- This capability is often called *random access*.

Program: Finding Repeated Digits in a Number

- The `RepeatedDigits` program will determine which digits in a number appear more than once:

```
Enter a number: 392522459
Repeated digits: 2 5 9
```

The digits will be displayed in increasing order, from smallest to largest.

- If the user enters a number such as 361, the program will display the message No repeated digits.

Storing Digits

- `RepeatedDigits` will use an array of `int` values to keep track of how many times each digit appears in the user's number.
- The array, named `digitCounts`, will be indexed from 0 to 9 to correspond to the 10 possible digits.
- Initially, every element of the array will have the value 0.
- The user's input will be converted into an integer and stored in a variable named `number`.

Design of the `RepeatedDigits` Program

- The program will examine number's digits one at a time, incrementing one of the elements of `digitCounts` each time, using the statement `digitCounts[number%10]++;`
- After each increment, the program will remove the last digit of `number` by dividing it by 10.
- When `number` reaches zero, the `digitCounts` array will contain counts indicating how many times each digit appeared in the original number.

Design of the RepeatedDigits Program

- If number is originally 392522459, the `digitCounts` array will have the following appearance:

0	0	3	1	1	2	0	0	0	2
0	1	2	3	4	5	6	7	8	9

- The program will next use a `for` statement to visit each element of the `digitCounts` array.
- If a particular element of the array has a value greater than 1, the index of that element is added to a string named `repeatedDigits`.

RepeatedDigits.java

```
// Checks a number for repeated digits
import java.util.*;

public class RepeatedDigits {
    public static void main(String[] args) {
        // Prompt user to enter a number and convert to int form
        System.out.print("Enter a number: ");
        Scanner input = new Scanner(System.in);
        int number = input.nextInt();

        // Create an array to store digit counts
        int[] digitCounts = new int[10];

        // Remove digits from the number, one by one, and
        // increment the corresponding array element
        while (number > 0) {
            digitCounts[number%10]++;
            number /= 10;
        }
    }
}
```

```
// Create a string containing all repeated digits
String repeatedDigits = "";
for (int i = 0; i < digitCounts.length; i++)
    if (digitCounts[i] > 1)
        repeatedDigits += i + " ";

// Display repeated digits. If no digits are repeated,
// display "No repeated digits".
if (repeatedDigits.length() > 0)
    System.out.println("Repeated digits: " +
        repeatedDigits);
else
    System.out.println("No repeated digits");
}
```

5.6 Using Arrays as Databases

- [Note: Section 5.5 omitted]
- In many real-world programs, data is stored as a collection of **records**.
- Each record contains several related items of data, known as **fields**.
- Such collections are known as **databases**.
- Examples:
 - Airline flights
 - Customer accounts at a bank

The Phone Directory as a Database

- Even the humble phone directory is a database:

AARON Robin B	4011 Stone Mountain St	384-7110
ABBOTT C Michael	981 Glen Arden Way	776-5188
ABEL A B	343 Lakeshore Ct	871-7406
ABERCROMBIE Bill	5810 Lismoor Tr	844-9400
ABERNATHY C	2120 Martin Rd	779-7559
ABRAHAM Gary	585 Chandler Pond Dr	582-6630
:	:	:

- There are many ways to store databases in Java, but the simplest is the array.

Parallel Arrays

- The first technique for storing a database is to use **parallel** arrays, one for each field.
- For example, the records in a phone directory would be stored in three arrays:

0	AARON Robin B	0	4011 Stone Mountain St	0	384-7110
1	ABBOTT C Michael	1	981 Glen Arden Way	1	776-5188
2	ABEL A B	2	343 Lakeshore Ct	2	871-7406
3	ABERCROMBIE Bill	3	5810 Lismoor Tr	3	844-9400
4	ABERNATHY C	4	2120 Martin Rd	4	779-7559
5	ABRAHAM Gary	5	585 Chandler Pond Dr	5	582-6630
:	:	:	:	:	:

Parallel Arrays

- The code declaring the arrays with size 1000 is


```
String[] name    = new String[1000];
String[] address = new String[1000];
String[] phone   = new String[1000];
```
- `name[i]`, `address[i]` and `phone[i]` corresponds to the information of a person.

Parallel Arrays

- Parallel arrays can be used to store a collection of points, where a point consists of an *x* coordinate and a *y* coordinate:


```
int[] xCoordinates = new int[100];
int[] yCoordinates = new int[100];
```
- The values of `xCoordinates[i]` and `yCoordinates[i]` represent a single point.

Parallel Arrays

- Parallel arrays can be useful. However, they suffer from two problems:
 - It's better to deal with one data structure rather than several.
 - Maintenance is more difficult. Changing the length of one parallel array requires changing the lengths of the others as well.

Arrays of Objects

- An alternative to parallel arrays is to treat each record as an object, then store those objects in an array.
- A `PhoneRecord` object could store a name, address, and phone number.

```
public class PhoneRecord {
    private String name;
    private String address;
    private String phone;
    // ... other code
}

public class PhoneDirectory {
    private PhoneRecord[] phoneRecord =
        new PhoneRecord[1000];

    // ... other code
}
```

Arrays of Objects

- A `Point` object could contain instance variables named *x* and *y*. (The Java API has such a class.)

```
public class Point {
    private int x;
    private int y;
    // ... other code
}
```

- An array of `Point` objects:


```
Point[] points = new Point[100];
```

Creating a Database

- Consider the problem of keeping track of the accounts in a bank, where each account has an account number (a `String` object) and a balance (a `double` value).

- One way to store the database would be to use two parallel arrays:

```
String[] accountNumbers = new String[100];
double[] accountBalances = new double[100];
```

- A third variable would keep track of how many accounts are currently stored in the database:

```
int numAccounts = 0;
```

Creating a Database

- Statements that add a new account to the database:

```
accountNumbers[numAccounts] = newAccountNumber;
accountBalances[numAccounts] = newBalance;
numAccounts++;
```
- `numAccounts` serves two roles. It keeps track of the number of accounts, but it also indicates the next available “empty” position in the two arrays.

Creating a Database

- Writing the code in a class, we have

```
public class BankParallelArrays {
    private int numAccounts = 0;
    private String[] accountNumbers = new String[100];
    private double[] accountBalances = new double[100];

    public void createAccount(String newAccountNumber,
                             double newBalance) {
        accountNumbers[numAccounts] = newAccountNumber;
        accountBalances[numAccounts] = newBalance;
        numAccounts++;
    }
}
```

Creating a Database

- Another way to store the bank database would be to use a single array whose elements are `BankAccount` objects.
- The `BankAccount` class will have two instance variables (the account number and the balance).

```
public class BankAccount {
    private String number;
    private double balance;
}
```

- `BankAccount` constructors and methods:

```
public BankAccount(String accountNumber,
                   double initialBalance)
public void deposit(double amount)
public void withdraw(double amount)
public String getNumber()
public double getBalance()
```

Creating a Database

- `BankAccount` objects will be stored in the `accounts` array:

```
BankAccount[] accounts = new BankAccount[100];
```
- `numAccounts` will track the number of accounts currently stored in the array.
- Fundamental operations on a database:
 - Adding a new record
 - Deleting a record
 - Locating a record
 - Modifying a record

Adding a Record to a Database

- Adding a record to a database is done by creating a new object and storing it in the array at the next available position:

```
accounts[numAccounts] =
    new BankAccount(number, balance);
numAccounts++;
```

- The two statements can be combined:

```
accounts[numAccounts++] =
    new BankAccount(number, balance);
```

- In some cases, the records in a database will need to be stored in a particular order.

Creating a Database

- Writing the code in a class, we have

```
public class BankAccount {
    private String number; // account number
    private double balance;
    public BankAccount(String accountNumber,
                       double initialBalance) {
        number = accountNumber;
        balance = initialBalance;
    }
    public String getNumber() { return number; }
    public void deposit(double amount) { balance += amount; }
}

public class BankObjectArray {
    private int numAccounts = 0;
    private BankAccount[] accounts = new BankAccount[100];
    public void createAccount(String number, double balance) {
        accounts[numAccounts] = new BankAccount(number, balance);
        numAccounts++;
    }
}
```

Removing a Record from a Database

- When a record is removed from a database, it leaves a “hole”—an element that doesn’t contain a record.
- The hole can be filled by moving the last record there and decrementing numAccounts:

```
accounts[i] = accounts[numAccounts-1];
numAccounts--;
```
- A method can be added to the class BankObjectArray with these statements combined:

```
public void deleteAccount(int i) {
    accounts[i] = accounts[--numAccounts];
}
```
- This technique works even when the database contains only one record.

Searching a Database

- Searching a database usually involves looking for a record that matches a certain “key” value.
- A method can be added to the class BankObjectArray to look for the index of an account

```
public int indexOf(String number) {
    int i;
    for (i = 0; i < numAccounts &&
        !accounts[i].getNumber().equals(number); i++);
    return i;
}
```
- The caller of the method need to test whether i is less than numAccounts. If so, the value of i indicates the position of the account in the array.

Modifying a Record in a Database

- A record can be updated by calling a method that changes the object’s state.
- A statement that deposits money into the account located at position i in the accounts array:

```
accounts[i].deposit(amount);
```
- It’s sometimes more convenient to assign an array element to a variable, and then use the variable when performing the update:

```
BankAccount currentAccount = accounts[i];
currentAccount.deposit(amount);
```
- A method can be added to the class BankObjectArray to deposit an amount of money into an account:

```
public void deposit(String number, double amount) {
    int i = indexOf(number);
    if (i < numAccounts)
        accounts[i].deposit(amount);
}
```

BankObjectArray class

```
public class BankObjectArray {
    private int numAccounts = 0;
    private BankAccount[] accounts = new BankAccount[100];

    public void createAccount(String number, double balance) {
        accounts[numAccounts] = new BankAccount(number, balance);
        numAccounts++;
    }

    public void deleteAccount(int i) {
        accounts[i] = accounts[--numAccounts];
    }

    public int indexOf(String number) {
        int i;
        for (i = 0; i < numAccounts &&
            !accounts[i].getNumber().equals(number); i++);
        return i;
    }

    public void deposit(String number, double amount) {
        int i = indexOf(number);
        if (i < numAccounts)
            accounts[i].deposit(amount);
    }
}
```


5.7 Arrays as Objects

- Like objects, arrays are created using the new keyword.
- Arrays really *are* objects, and array variables have the same properties as object variables.
- An object variable doesn’t actually store an object. Instead, it stores a *reference* to an object. Array variables work the same way.

Properties of Object Variables

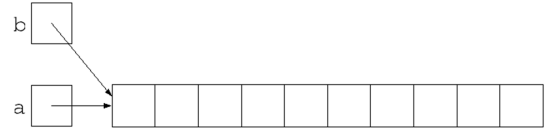
- Object variables have the following properties:
 - When an object variable is declared, it’s not necessary for the variable to refer to an object immediately.
 - The value of an object variable can be changed as often as desired.
 - Several object variables can refer to the same object.
 - When no variable refers to an object, it becomes eligible for garbage collection.
 - Assigning one object variable to another causes only a reference to be copied; no new object is created.
 - Testing whether two object variables are equal or not equal compares the references stored in the variables.

How Arrays are Stored

- An array variable contains a reference to where the array's elements are stored.
 - Storage for an array named `a` containing 10 integers:
- 
- Arrays are “garbage collected” in the same way as other objects. When there are no more references to an array, the space occupied by the array can be reclaimed automatically.

Copying Arrays

- If `a` and `b` are array variables of the same type, it's legal to write
`b = a;`
- The effect is that `b` now contains a reference to the same array as `a`:



Copying Arrays

- The assignment operator doesn't make a true copy of an array. To make a genuine copy, there are two strategies:
 - Create a new array of the same length as the old one and copy the elements from the old array to the new one.
 - Use the `clone` method.
- Testing whether two array variables are equal (or not equal) is legal. However, this only checks whether the two variables refer to the same array.
- Checking whether two arrays contain identical elements requires writing a loop.

Resizing an Array

- Although arrays have fixed sizes, it's possible to resize an array if it becomes full. The trick is to create an entirely new array to replace the old one.
- Resizing an array takes three steps:
 1. Create a new array that's larger than the old one.
 2. Copy the elements of the old array into the new array.
 3. Assign the new array to the old array variable.

Resizing an Array

- Code that doubles the size of the `accounts` array:


```
BankAccount[] tempArray =
    new BankAccount[accounts.length*2];
for (int i = 0; i < accounts.length; i++)
    tempArray[i] = accounts[i];
accounts = tempArray;
```
- Doubling the size of an array provides plenty of space for new elements, yet guarantees that there won't be too much unused space.

Resizing an Array

- When an array is resized, the old array can be reclaimed by the garbage collector.
- Copying elements from the old array to the new usually doesn't take that long. If the elements are objects, only references to the objects are copied, not the objects themselves.
- For additional speed, Java provides a method named `System.arraycopy` that can be used to copy elements from one array to another.

Resizing an Array

- A call of `System.arraycopy` that copies the elements of `accounts` into `tempArray`, starting from position 0 in both arrays:

```
System.arraycopy(accounts, 0, tempArray, 0, accounts.length);
```

The last argument is the number of elements to be copied.
- Instances of Java's `Vector` class behave like arrays that automatically grow when they become full.

5.8 Case Study: A Phone Directory

- The `PhoneDirectory` program will store names and telephone numbers.
- The program will support two operations:
 - Entering new names and numbers
 - Looking up existing names
- `PhoneDirectory` is a menu-driven program. When the program begins to execute, it presents the user with a list of commands. The user can enter as many commands as desired, in any order.

User Interface

- The `PhoneDirectory` program will ask the user to enter one of three commands:
 Phone directory commands:
 a - Add a new phone number
 f - Find a phone number
 q - Quit

 Enter command (a, f, or q): **a**
 Enter new name: **Abernathy, C.**
 Enter new phone number: **779-7559**

User Interface

```
Enter command (a, f, or q): a
Enter new name: Abbott, C. Michael
Enter new phone number: 776-5188

Enter command (a, f, or q): f
Enter name to look up: Abbott
Abbott, C. Michael 776-5188

Enter command (a, f, or q): f
Enter name to look up: ab
Abernathy, C. 779-7559
Abbott, C. Michael 776-5188

Enter command (a, f, or q): q
```

Commands

- **a** command: Prompts the user to enter a name and a number, which are then stored in the program's database.
- **f** command: Prompts the user to enter a name. The program then displays all matching names in the database, along with the corresponding phone numbers.
- The user doesn't need to enter an entire name. The program will display all names that begin with the characters entered by the user. The case of the input doesn't matter.

Design of the `PhoneDirectory` Program

- Overall design for the program:
 1. Display a list of commands.
 2. Read and execute commands.
- A pseudocode version of step 2:


```
while (true) {
    Prompt user to enter command;
    Execute command;
}
```
- When the user enters the "quit" command, the loop will terminate by executing a `break` statement or a `return` statement.

Design of the PhoneDirectory Program

- The body of the `while` loop will need to test whether the command is `a`, `f`, `q`, or something else.
- A good way to implement a series of related tests is to use a cascaded `if` statement.

Design of the PhoneDirectory Program

- A more detailed version of step 2:

```
while (true) {
    Prompt user to enter command;
    if (command is "a") {
        Prompt user for name and number;
        Create a phone record and store it in the database;
    } else if (command is "f") {
        Prompt user for search key;
        Search the database for records whose names begin with the search key;
        Print these names and the corresponding phone numbers;
    } else if (command is "q") {
        Terminate program;
    } else {
        Display error message;
    }
}
```

Design of the PhoneDirectory Program

- Each name and phone number will be stored in a `PhoneRecord` object. The database will be an array of `PhoneRecord` objects.
- The `PhoneRecord` class will need a constructor to initialize the name and number instance variables.
- It will also need a couple of getters, `getName` and `getNumber`, which return the values of the instance variables.

PhoneDirectory.java

```
// Program name: PhoneDirectory
// Author: K. N. King (modified by Kelvin Lee)
// Written: 1999-06-22
//
// Stores names and telephone numbers and allows phone
// numbers to be looked up. The user is given a menu of
// three commands:
//
// a - Add a new phone number
// f - Find a phone number
// q - Quit
//
```

```
// The "a" command prompts the user to enter a name and a
// number, which are then stored in the program's database.
//
// The "f" command prompts the user to enter a name; the
// program then displays all matching names in the database,
// along with the corresponding phone numbers. It is not
// necessary to enter the entire name; all names that begin
// with the specified characters will be displayed. The "f"
// command ignores the case of letters when looking for
// matching names.
//
// The "q" command causes the program to terminate. All names
// and numbers are lost.
```

```
import java.util.Scanner;
// import jpb.*;
public class PhoneDirectory {
    public static void main(String[] args) {
        PhoneRecord[] records = new PhoneRecord[100];
        int numRecords = 0;
        Scanner input = new Scanner(System.in);
        // Display list of commands
        System.out.println("Phone directory commands:\n" +
            " a - Add a new phone number\n" +
            " f - Find a phone number\n" +
            " q - Quit\n");
        // Read and execute commands
        while (true) {
            // Prompt user to enter a command
            System.out.print("Enter command (a, f, or q): ");
            String command = input.nextLine().trim();
            // SimpleIO.prompt("Enter command (a, f, or q): ");
            // String command = SimpleIO.readLine().trim();
```

Chapter 5: Array

```
// Determine whether command is "a", "f", "q", or
// illegal; execute command if legal.
if (command.equalsIgnoreCase("a")) {
    // Command is "a". Prompt user for name and number,
    // then create a phone record and store it in the
    // database.
    if (numRecords < records.length) {
        System.out.print("Enter new name: ");
        String name = input.nextLine().trim();
        System.out.print("Enter new phone number: ");
        String number = input.nextLine().trim();
        // SimpleIO.prompt("Enter new name: ");
        // String name = SimpleIO.readLine().trim();
        // SimpleIO.prompt("Enter new phone number: ");
        // String number = SimpleIO.readLine().trim();
        records[numRecords] =
            new PhoneRecord(name, number);
        numRecords++;
    } else
        System.out.println("Database is full");
}
```

Chapter 5: Array

```
} else if (command.equalsIgnoreCase("f")) {

    // Command is "f". Prompt user for search key.
    // Search the database for records whose names begin
    // with the search key. Print these names and the
    // corresponding phone numbers.
    System.out.print("Enter name to look up: ");
    String key = input.nextLine().trim().toLowerCase();
    // SimpleIO.prompt("Enter name to look up: ");
    // String key = SimpleIO.readLine().trim().toLowerCase();
    for (int i = 0; i < numRecords; i++) {
        String name = records[i].getName().toLowerCase();
        if (name.startsWith(key))
            System.out.println(records[i].getName() + " " +
                                records[i].getNumber());
    }
}
```

Chapter 5: Array

```
} else if (command.equalsIgnoreCase("q")) {
    // Command is "q". Terminate program.
    return;
} else {
    // Command is illegal. Display error message.
    System.out.println("Command was not recognized; " +
                        "please enter only a, f, or q.");
}

System.out.println();
}
}
```

Chapter 5: Array

```
// Represents a record containing a name and a phone number
class PhoneRecord {
    private String name;
    private String number;

    // Constructor
    public PhoneRecord(String personName, String phoneNumber) {
        name = personName;
        number = phoneNumber;
    }

    // Returns the name stored in the record
    public String getName() {
        return name;
    }

    // Returns the phone number stored in the record
    public String getNumber() {
        return number;
    }
}
```

Chapter 5: Array

Remarks

- The `startsWith` method returns `true` if the string that calls the method begins with the characters in another string (supplied as an argument).
- The `PhoneDirectory` program has little practical use because the names and phone numbers are lost when the program terminates. A better version of the program would save the names and numbers in a file.