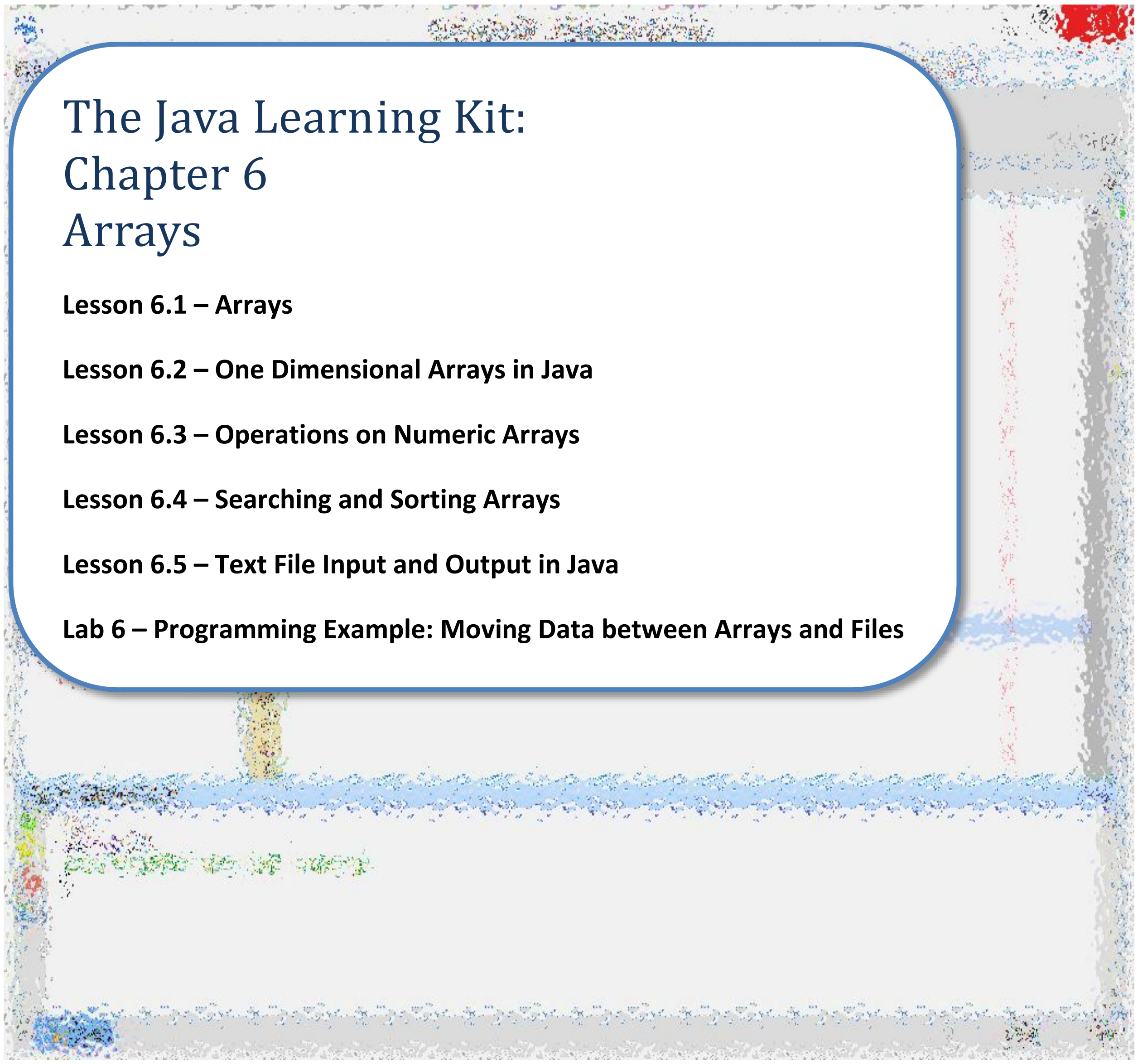


The Java Learning Kit: Chapter 6 Arrays

The Java Learning Kit: Chapter 6 – Arrays

Copyright 2015 by C. Herbert, all rights reserved.

Last edited January, 2015 by C. Herbert

This document is a chapter from a draft of the Java Learning Kit, written by Charles Herbert, with editorial input from Craig Nelson, Christopher Quinones, Matthew Staley, and Daphne Herbert. It is available free of charge for students in Computer Science courses at Community College of Philadelphia during the Spring 2015 semester.

This material is protected by United States and international copyright law and may not be reproduced, distributed, transmitted, displayed, published or broadcast without the prior written permission of the copyright holder. You may not alter or remove any trademark, copyright or other notice from copies of the material.

Contents

The Java Learning Kit: Chapter 6
Arrays

This chapter introduces the concept of an array and working with arrays in Java, along with the use of text files in Java applications for long-term storage of arrays and other data.

Lesson 1 introduces the concept of an array as a set of subscripted variables, including a look at how arrays are stored in memory.

Lesson 2 shows how to implement arrays for Java's built-in data types.

Lesson 3 discusses simple statistical processing of numeric arrays — finding the sum, average minimum, and maximum in an array of numbers.

Lesson 4 looks at simple techniques for searching and sorting arrays.

Lesson 5 introduces simple text file I/O in Java.

Chapter 6 Learning Outcomes

Upon completion of this chapter students should be able to:

- describe the concept of an array, including how arrays are stored in a computer's memory.
- describe the nature of an array in Java; and how to declare an array, initialize an array, and iterate an array in Java.
- describe how to pass a reference to an array as a method parameter in Java; and how to pass an individual array element as a method parameter in Java.
- implement Java code to declare an array, initialize an array, iterate an array; and pass both references to arrays and individual array elements as parameters.
- describe techniques for performing common mathematical operations on numeric arrays, including counting array elements, and finding the sum, average, minimum and maximum of values in an array.
- implement Java code to count array elements, and to find the sum, average, minimum and maximum of values in an array.
- describe an algorithm for a linear to search of an array.
- describe an algorithm to perform a *Bubble Sort* of an array.
- implement Java code to compare Strings, swap the value of two variables, search an array, and sort an array.

Lesson 6.1 Arrays

Until now we have been using single variables in our Java programs. Java, like most other programming languages, Java also supports arrays. An **array** is a set of indexed variables, all of the same data type. Instead of just having individual variables such as *x*, *y,* or *empName*, we can have a set of indexed variables such as x_1, x_2, x_3, … and so on, where the value of the index identifies an element of an array. Each **element** of an array is an individual data item in the array, such as:

$\mathbf{empName_0, empName_1, empName_2, … empName_{n-1}}$

In mathematics, such an array is also referred to as a set of subscripted variables, because the index is written as a subscript. In Java, they may also be called subscripted variables, but instead of using subscripts, Java uses square brackets following the variable name to indicate an array or an element in an array. $empName_0$, $empName_1$, $employname_2$… from mathematics would be written as *empName[0] , empName[1], empName [2] … empName [n-1]* in Java. *empName* is the name of the array; whereas empName[i], where *i* is an integer, is an element in the array empName. The index values in an array start at zero for reasons we will see later in this chapter.

empName[0] , empName[1], empName[2] … empName[n-1]

Arrays are very useful for storing and working with sets of similar data items, such as lists of test scores. We will see how to set up an array to hold a set of numbers, such as test scores, then write code to find the sum, the average, the minimum or the maximum of the numbers. Once data has been established in the computer's memory as an array, we can do many things with the data, including conducting a full statistical analysis of the data, searching the data for specific items, or sorting the data. Later, when we learn to work with objects, we can set up an array of objects, such an array of student records with properties like student number, first name, last name, address, and major, then we can sort or search through the data according to any of the object's properties. But for now, we will only work with arrays of primitive data types and Strings.

Most high-level programming languages support using arrays of data. They generally have two different type of arrays, single-dimensional arrays and mutli-dimensional arrays. A **single dimensional array**, also called a **linear array**, is an array with only one index variable, forming what can be thought of as a line of data items. The above example of *empName[0] , empName[1], empName [2] … empName [n-1] is* a one dimensional linear array.

Multi-dimensional arrays have multiple index variables, such as the two-dimensional array shown here:

$$\begin{array}{lll}
a_{0,0} & a_{0,1} & a_{0,2} \ldots a_{0,n} \\
a_{1,0} & a_{1,1} & a_{1,2} \ldots a_{1,n} \\
a_{2,0} & a_{2,1} & a_{2,2} \ldots a_{2,n} \\
a_{m,0} & a_{m,1} & a_{m,2} \ldots a_{m,n}
\end{array}$$

This is a two dimensional array from the world of mathematics, forming a table with *m* rows and *n* columns. In Java, the array would look like this:

a[0,0] a[0,1] a[0,2]. . . a[0,n]

a[1,0] a[1,1] a[1,2]. . . a[1,n]

a[2,0] a[2,1] a[2,2]. . . a[2,n]

a[m,0] a[m,1] a[m,2] . . . a[m,n]

Two dimensional arrays can be used to represent tables of data with rows and columns of information, similar to an electronic spreadsheet. They are often used for matrices in linear algebra problems.

Multi-dimensional arrays can have more than two dimensions. A three dimensional array could, for example, represent a three-dimensional matrix for solving problems related to three-dimensional Euclidean space with x, y, and z coordinates. However, for the remainder of this chapter we will stick with simple single dimensional arrays, leaving the more complex problems for another day.

Array Storage in Computer Memory

The values for variables in a computer program are stored in a computer's internal memory while the program runs. Hence, each variable name is associated with a specific address in the computer's memory. This matching of variables to memory addresses is usually managed by the computer's operating system with a symbol table. A **symbol table** is a list of symbolic names, such as the names of methods, classes, and variables, mapped to specific addresses in a computer's internal memory.

Almost all modern computer systems have **byte addressable memory**, meaning that bytes in a computer's memory are sequentially numbered, regardless of the word length of a particular processing unit. There are many advantages to this, which are discussed in computer architecture and organization courses such as *CSCI 213– Computer Organization* at Community College of Philadelphia.

The following example shows how an operating system keeps track of the values of variables using a symbol table. The memory map shows us how space is allocated in the computer's memory.

variables

int score;
double velocity;
int colorDepth;
boolean transparent;

symbol table

score	4000
velocity	4004
colorDepth	400C
transparent	4010

memory map

4000	score
4001	
4002	
4003	
4004	velocity
4005	
4006	
4007	
4008	
4009	
400A	
400B	
400C	color depth
400D	
400E	
400F	
4010	transparent

Memory is allocated according to how variables are declared.

Space is allocated based on data types, as described back in chapter 2, for example:
  int 32 bits (4 bytes)
  double 64 bits (8 bytes)
  boolean (1 byte in this system)

The symbol table records where variables are stored in the internal memory. Memory addresses are hexadecimal (base 16).

When an array is declared, the operating system only keeps track of array's **base address**, which is the location where the array starts in memory. Here is an example:

variables

```
priority  = new int [6];
int score;
```

symbol table

priority	4000
score	4018

memory map

4000	priority [0]
4001	
4002	
4003	
4004	priority [1]
4005	
4006	
4007	
4008	priority [2]
4009	
400A	
400B	
400C	priority [3]
400D	
400E	
400F	
4010	priority [4]
4011	
4012	
4013	
4014	priority [5]
4015	
4016	
4017	
4018	score
4019	
401A	
401B	

The symbol table only contains the base address of an array. Since all of the elements of the array are the same data type, they will all use the same number of bytes in memory. So, the computer calculates the address of an element in an array using the formula:

memory address = *base address* + (*index* * *size*)

In this example, the base address for the array priority is 4000. The size of each element is 4 bytes, since it is an array of integers. So, the address of priority[2] is:

address of priority[2] = *4000 + 2 * 4*

which equals 4008.

(Remember, addresses are base 16 – hexadecimal).

This scheme works because the index of the first element in the array is 0, not 1. In fact, this is why array index values start at 0.

Notice that the highest index is one less than the size of the array, because the array index numbering starts at zero. In this example *priority* has 6 elements, *priority[0]* through *priority[5]*.

CheckPoint 6.1

1. What is the relationship among the terms *array*, *indexed variables*, and *subscripted variables*?
2. What is a linear array?
3. What does it mean to say that most computers use byte-addressable memory?
4. If array begins at memory address 5000 and each array element requires 16 bits of storage space, where will the array element with index 3 be stored in memory?
5. How can arrays be used to represent tables of data with rows and columns, similar to an electronic spreadsheet?

Lesson 6.2 One Dimensional Arrays in Java

A **one-dimensional array** in Java is a indexed fixed-length homogeneous data structure. In other words, it is a set of sequentially numbered data elements, in which all of the data must be of the same data type, and with a number of elements that cannot be changed. In Java, an array is an object.

There are two parts to declaring an array in Java, unlike primitive data types, which have a one-step declaration. First, the array must be named; second, the size must be declared.

step 1 – Declare a name for the array. There are two styles for doing this, which both assign a name to an array, but do not create the array in memory: (The first style is preferred.)

```
int[] score;    // declares score to be the name of an int array, it has no elements
int score[];    // declares score to be the name of an int array, it has no elements
```

step 2 – Declare the size of the array.

```
score = new int[10];    // declares the array score to have 10 integer elements
```

The number in brackets is the number of elements or size of the array, also known as the **array length**. The indexes in Java arrays must be non-negative and always start at zero. The highest index will be one less than the size of the array. If the array length is *n,* then the highest index will be *n-1.*

The statement that declares the array length actually creates the array.

The two statements naming an array and declaring an array must both be used to create the array:

```
int[] score;
score = new int[10];
```

The name and size declarations for an array can be put together in one Java statement, which is the easiest and most common way to create an array in Java:

```
int[] score = new int[10];  // score is the name of an integer array with 10 elements
```

The keyword ***new*** is necessary when declaring objects. An array is an object in Java. This statement declares an array named score and sets the size to 10 elements.

What is actually in an array in Java?

Each element in an array is an individual variable of the data type declared for the array. For example, if *empName* is declared as:

```
String[]  empName = new String[10];
```

then each of the array elements, such as empName[3], is a String variable.

If the data type of an array is a primitive data type, then each element of the array contains a value of that data type. If the data type of an array is a class of objects, then each element in the array contains a memory address referring to where an object of the correct data type is stored in memory. A memory address referring to where an object is stored is called a **reference to an object**. Java Strings are objects, so each element in a String array holds a reference to a String, not the String itself.

Example 1 – an integer array

An array of integers contains the actual values of integers, since *int* is a primitive data type.

```
    int[] score = new int[5];
```

This statement creates an array of integers named *score* with 5 elements. The actual values of integers would be stored in the each element in the array, such as:

memory address	variable name for this address	value stored at this address
4000	score[0]	93
4004	score[1]	87
4008	score[2]	91
400C	score[3]	95
4010	score[4]	88

We can think of an array as a set of empty boxes. In the integer array *score*, each box holds an integer:

`Score[]`

93	87	91	95	88

Example 2 – A String array

With any array of objects, including a String array, the elements of the array actually contain references to the objects, not the objects themselves. So, an array of Strings would contain references to where the Strings themselves are stored in memory, like this:

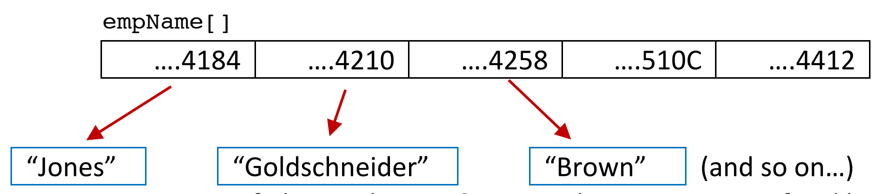
```
String[] empName = new String[5];
```

memory address	variable name for this address	value stored at this address
4000	empName[0]	4184
4004	empName[1]	4210
4008	empName[2]	4258
400C	empName[3]	....510C
4010	empName[4]	4412

This example assumes that each memory address is 32 bits (4 bytes) long, but only the last 4 hexadecimal digits are shown.

The Strings are actually stored in the memory locations referenced by the values in the array.

If we think of an array as a set of empty boxes, then each box in the String array *empName*, holds a reference to where a String value is actually stored in memory:



Why does Java store arrays of objects this way? Remember, an array is a fixed length data structure, but Strings and many other objects are not fixed length, their sizes can change. This method for storing objects in memory allows variable length objects to be store in fixed length arrays. It also has several advantages that are beyond the scope of what we cover in this chapter.

Even though the elements of a String array actually contain references to Strings, if we use a variable from the array in our Java code, we will get the String, not the reference. For the example above:

```
   System.out.println( empName[2] ); //prints the name Brown,  not the reference ….4258
```

Once we learn to work with objects, we will see that we can have an array for any type of object, just as with Strings, using references to the objects stored in memory, such as an array of images:

Example 3 – an Array of Images

```
   JPGimage[] orientation = new JPGimage[5];
```

orientation[]

4100	6404	7820	880C	B204

… and so on. Each element in the array contains a reference to a stored image.

Assigning Values to Array Elements

There are two ways to assign values to array elements:

1. using assignment statements
   For example:

```
score[3] = 98;          // an int value is assigned to the fourth element in score
empName[0] = “Jones”;   // a String value is assigned to the first element in empName
a[4] = x;               // a[4] is set to the value of x (data types must match)
```

2. by listing values in an alternate array declaration

   Arrays may be declared in a statement that lists their values:

```
int[] score =  {93, 87, 91, 95, 88};
```

   This method is sometimes called an **explicit array declaration**. The values listed in braces, separated by commas, are assigned in order to the array elements. The length of the array is determined by the number of values in the list. The keyword *new* is not needed in an explicit array declaration.

An explicit array declaration like the one just shown above works well for small arrays, but can be awkward for large arrays.

One common way to initialize the values in an array is with a count controlled loop. Here are a few examples:

Example 4 – Initializing all of the elements in an Array to Zero

The following code initializes the values of an array of data type double all to the same value, zero:

```
double[] cost = new double[100];     // an array of 100 numbers

for(int i=0; i<100; i++)
    cost[i] = 0.0;
```

Example 5 – Initializing an Array with User Input

The following code initializes the values of an array of data type String with user input. Inside the array the names are numbered starting with 0, but to the user they are numbered starting with 1. This assumes that the code is encapsulated, meaning that the user does not see the code.

```
String[] student = new String[10];   // an array of 10 student names
for( int i= 0; i < 10; i++)
  {
      // Ask the user for a student name, then get the name from the keyboard
      System.out.print("Please enter the name of student number" + (i+1) + “:” );
      student[i] = x.nextLine();
   } // end for
```

Example 6 – Initializing an Array with random numbers

The following code initializes the values of an int array with random numbers between 1 and 100. An array of random numbers is often used in game or simulation programming, or to test other software.

```
   int [] num = new int[100];     // an array of 100 integers
   for( int i= 0; i < 100; i++)
      num[i] = 1+(int)(Math.random()*100);  //  random, 1 <= num[i] <= 100
```

Math.random() yields a value between 0 and 1; the arithmetic converts it to a integer between 1 and 100.

The Array *length* Property; Iterating an Array

`x.length`, where `x` is the name of an array, is a method for instances of array objects that returns the length of the array as an integer. For example, if the array score[] has 10 elements, then `score.length` returns the value 10.

The length property can be used to iterate an array. To **iterate an array** means to go through each of the elements in the array on at a time. The three examples above each iterate an array, and could be written to use the length property. Here is the code from example 4, above, re-written using the length property for the *cost* array.

Example 7 – Initializing an Array using the length method

```
   double[] cost = new double[100];     // an array of 100 numbers
   for(int i = 0; i < cost.length; i++)
      cost[i] = 0.0;
```

There are many different reasons to iterate an array – to initialize the array as above, or to print the elements of the array, for example.

Example 8 – Printing the Elements of an Array

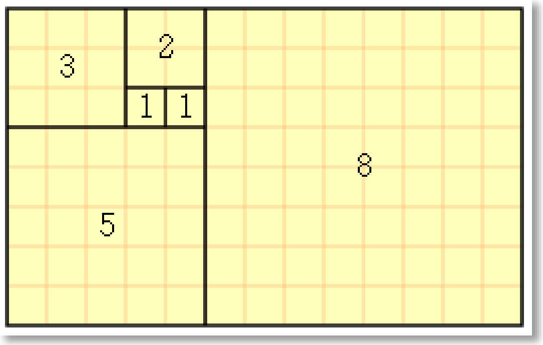
```
   // the following assumes student is a String array that has been initialized
   for( int i= 0; i < student.length; i++)
      System.out.print("Student number " + ( i+1) + “ is ” +  student[i]);
```

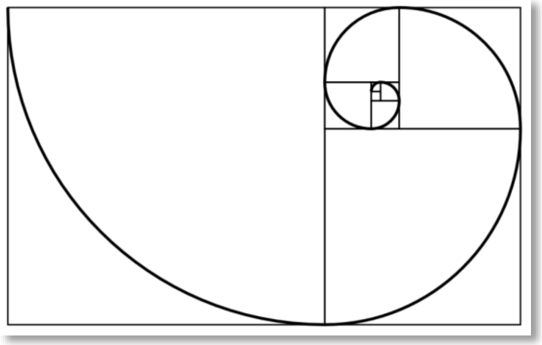
Example 9 – Printing the Fibonacci numbers

The Fibonacci sequence is a sequence of numbers that starts with 0 and 1, then each successive term in the sequence determined by adding the previous two terms.

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Such a pattern is very common in the natural world. The following diagrams help us to understand why that is so. On the left, we see a set of rectangles composed by adding squares whose sides are successively larger Fibonacci numbers. The next square to be added will have a side of length 13, and will be added above the existing rectangle. On the right we see a spiral drawn by connecting the corners of Fibonacci squares. The growth of branches on a tree, the shape of a spiral sea shell, the pattern of seeds on a sunflower, the growth of a population of rabbits, and many other patterns in the real world fit the Fibonacci sequence.





In this example, we will store the first 10 Fibonacci numbers in an array named fib[], then print the array. We will declare the array using a set of values.

```
int[] fib = {0,1,1,2,3,5,8,13,21,34}; // an array initialized with Fibonacci numbers
int i;// used as an array index

System.out.println(“The first ten Fibonacci numbers” );
// this loop prints the values in the array
for( i= 0; i < fib.length; i++)
      System.out.print(fib[i] + “ “);
System.out.println();    // go to a newline after printing the array values
```

The output looks like this:

```
The first ten Fibonacci numbers
0 1 1 2 3 4 5 8 13 21 34
```

The Enhanced *for* Statement

The *for* statement we have been using so far is the *basic for statement*. Java also has a special version of the *for* statement, called the *enhanced for statement*. The **enhanced for statement** iterates an entire array. It may only be used with an array or an object that can be iterated like an array.

The *enhanced for statement* has a loop header with the keyword *for*, followed by parentheses that contain an integer declaration and the name of an array. The statement or block of code following the loop header will be executed once for each element in the array.

Within the statement or block of code, the integer declared in the loop header refers to each element in the array. The syntax within an enhanced for loop can be tricky, so be careful. We do not use the name of the array within the loop, only the integer variable declared in the header. This variable refers to an element of the array, sequentially incremented each time through the loop. Here is the syntax of the enhanced for loop:

```
for ( integer declaration : name of an array)
    block of code
```

Here is the same code from above to print the array with the first 10 Fibonacci numbers, using an enhanced for loop instead of a basic for loop:

Example 10 – the enhanced for loop

```
int[] fib = {0,1,1,2,3,5,8,13,21,34} // an array initialized with Fibonacci numbers

System.out.println("The first ten Fibonacci numbers" );
// this loop prints the values in the array using the enhanced for statement
for(int i : fib)
      System.out.print(i + " ");   // i refers to fib[i] in this enhanced for loop
System.out.println();    // go to a newline after printing the array values
```

The output looks like this:

```
The first ten Fibonacci numbers
0 1 1 2 3 4 5 8 13 21 34
```

Although it is convenient, the enhanced for loop has limited applications. It can only be used to iterate an entire array, not part of an array, and the declared integer variable cannot be used as an integer variable within the loop, only to refer to each element of the array.

Copying an Array

An assignment statement such as *second = first;* will not copy an array; it copies the reference to the array, creating two variables referencing the same array. To copy an array in Java, we must create another array of the same data type and size, then copy each value from the original array into the new array. The actual copying can be done with a *for* loop or with an enhanced *for* loop.

Example 11 – copying an array

```
// an array initialized with ten prime numbers
      int[] first = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
      // To copy an array, we need to create another array of the same data type and size.
      int[] second = new int[10];

      // Then, we copy each element from the first array into the second array.
      for(int i : i < first.length; i++)
              second[i] = first[i];

      // This could also be done with an enhanced for loop
      for(int i : first)
              second[i] = first[i];
```

Arrays as Method Parameters in Java

An entire array can be passed as a method parameter, but remember, the name of the array by itself is a reference to the base address where the array is stored in memory. It is this reference, and not the values in the array, that is passed to another method when an array is passed in Java. We end up with two different reference variables pointing to the same array in memory, not two different arrays, as shown in example 12 on the next page.

Example 12 – Passing an Array as a Parameter

In this example, an entire array is passed from the main method to the method *printJob()*. However, the array is not really passed; it is the value of the reference variable that points to the array's location in memory that is really passed. Java always passes values. In this case the value is the address of the array in memory, so we end up with two different variables, `priority` – the original name for the array – and `category` – the copied name – both referring to the original array.

variables

priority = new int [6];
int score;

int[] category;
int i;

symbol table

priority	4000
score	4018
. . .	
category	4000
i	4054

memory map

4000	priority [0]
4001	
4002	
4003	
4004	priority [1]
4005	
4006	
4007	
4008	priority [2]
4009	
400A	
400B	
400C	priority [3]
400D	
400E	
400F	
4010	priority [4]
4011	
4012	
4013	
4014	priority [5]
4015	
4016	
4017	
4018	score
4019	
401A	
401B	
401C	
401D	
401E	
401F	
4020	

```
   int[] priority = new int[6];
   . . .
   // call method printJob()
   printJob(priority);
   . . .


   public static void printJob(int[] category){

for( int i=0; i < category.length; i++)   {
    System.out.printl("Job" + i + "is in category " + category[i]);
   . . .
  }
```

The value of the actual parameter ***priority*** is passed to the formal parameter ***category***, but that value is a reference to an array.

The method header for the *printJob()* method establishes a reference variable to refer to an array, category, but category has no value until the method is invoked from another method.

When *printJob()* is invoked from the main method, the value of the array reference variable priority is passed to array reference variable category in the formal parameter list. Now both priority and category have the same value, the address of the same array. This means they refer to the same array – priority and category have become two names for the same array – whatever we do to one we do to the other. If we change the value of an element in category we also change the value of that element in priority. *category[3]*, for example, is *priority[3].*

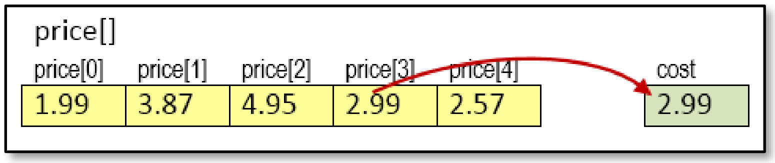
Java always passes a value, not a reference to a value, but that value may be an address of an object, such as the address of the array passed in this case.

Passing an Array Element as a Method Parameter

Each element in an array is an individual variable that is treated like any other any variable with the element's data type. It may be used like any other variable of that data type, including as an actual parameter for a method.

Example 13 – passing an array element as a parameter

In this example, price[] is an array of double values. `price[3]` in the main method is used as an actual parameter. The value of the variable `price[3]` is passed to the formal parameter `cost` when the method is invoked.



```
double[] price = { 1.99, 3.87, 4.95, 2.99, 2.57};
. . . // all of the code is not shown
currentSale(price[3]);
. . . // all of the code is not shown
/**********************************************************************/

public static void currentSale(double cost)  {
System.out.printl(The cost of the item is: “ + cost);
. . . // all of the code is not shown
}
```

The value of `price[3]` is copied to `cost` in the method currentSale(). Any change that is later made to `cost` has no effect on the original value in `price[3]`.

CheckPoint 6.2

1. A one-dimensional array in Java is defined as an *indexed fixed-length homogeneous data structure*. What does this mean?
2. What is actually in an array if the array's data type is a primitive data type and how does this differ from what is in an array if the array's data type is class of objects?
3. What are the two ways that a value can be assigned to an array element?
4. Create an example showing how the enhanced for statement can be used to iterate an array.
5. How is an entire array passed as a method parameter and what happens in the original array if a value is changed in an array within a method that has received the array as a parameter?

Lesson 6.3 Operations on Numeric Arrays

Once we have a set of numbers stored in an array, we can perform numerical analysis of the data. Common operations on numerical arrays include finding the sum, the average, the minimum and the maximum of the values in an array.

The following examples show how to do each of these things for a set of values in a double array.

Sum, Average, and Count of Values in an Array

To find the sum of a set numbers in an array we need a variable to hold the sum. It should have the same data type as the elements of the array, and it should be initialized to zero. Then, we iterate the loop, adding each value in the array to the sum, as in the following code. After the loop to iterate the array is finished, then the variable will hold the sum.

The average is simply the sum divided by the number of elements in the array. The array *length* property is the number of elements in the array.

Example 14 – finding the sum and average of values in an array

```
double[] cost = new double[100];     // an array of 100 numbers
double sum = 0;
// the statements that give values to the elements in the array are not shown

for(int i= 0; i < cost.length ; i++)
       sum = sum + cost[i];
System.out.println("the total cost is " +  sum);
System.out.println( "the average cost is " +  sum/cost.length   );
```

In this example, the average is calculated as part of the print statement. We could also have a variable to hold the average if we need to use it later.

Counting Elements in an Array

Sometimes we wish to count all of the values in an array that meet a certain condition. To do so, we simply initialize a counter to zero, then increment the counter for each element that meets the condition.

Example 15 – counting the elements in an array

In this we will again use the first ten Fibonacci numbers. Instead of printing each of the numbers, we will only print the even Fibonacci numbers and we will count them as well.

The variable `countEven` is first initialized to zero, then, as we iterate the array, we will increment the count for each even Fibonacci number. To test for even numbers, we will use the Java's remaindering operation (%), which returns the remainder of an integer division operation. If the remainder is zero when we divide by two, then the number is an even number.

```
int[] fib = {0,1,1,2,3,5,8,13,21,34}; // an array initialized with Fibonacci numbers
int i;// used as an array index
int countEven = 0;// used to count how many of the first 10 Fibonacci numbers are even

System.out.println("Even Fibonacci numbers" );
// this loop prints the values in the array
for( i= 0; i < fib.length ; i++)
       if (fib[i] % 2 == 0)  {
          countEven++;
          System.out.println( fib[i] + " is even");
          }
System.out.println ( countEven + " of the first 10 Fibonacci numbers are even");
```

The output looks like this:

```
Even Fibonacci numbers
0 is even
2 is even
8 is even
34 is even
4 of the first 10 Fibonacci numbers are even.
```

The Minimum and Maximum Values in an Array

To find the minimum value in an array, we start with a variable to hold the minimum. We initialize the minimum to the first value in the array, then go through the rest of the values in the array, testing each to see if it is less than the minimum. If a value is less than the minimum, it becomes the new minimum. After going through the entire array, the variable will hold the minimum value from the array.

The maximum is calculated in almost the same way, setting the first value to be the maximum, except we then check to see if each value is greater than the maximum.

Example 16 – finding the minimum value in an array

```
double[] cost = new double[100];        // an array of 100 numbers
int i;
double minimum;
// the statements that give values to the elements in the array are not shown

minimum = a[i];
for( i= 1; i < cost.length ; i++)  // iterate starting at the second element
       if ( cost[i] > minimum )
             minimum = cost[i];
System.out.println("the minimum cost is " +  minimum);
```

Example 17 – finding the maximum value in an array

```
double[] cost = new double[100];        // an array of 100 numbers
int i;
double maximum;
// the statements that give values to the elements in the array are not shown

maximum = a[i];
for( i= 1; i < cost.length ; i++)  // iterate starting at the second element
       if ( cost[i] < maximum)
             maximum = cost[i];
System.out.println("the maximum cost is " +  maximum);
```

Lesson 6.4 Searching and Sorting Arrays

Comparing Strings

In order to search or sort an array of Strings we need to be able to compare String values. This is easy for integers and floating point data, but when it comes to Strings, there is a complication.

The Boolean condition (*a* == *b*) will work if *a* and *b* are Strings, but it doesn't work the way most people think it does. Strings are objects, and a comparison of two objects using the == operators compares the reference values stored in the two variables, not the values of the objects themselves. In other words, *(a==b)* will be true only if *a* and *b* refer to the same object, not equal objects. We have a similar problem with the < and > operators.

Java has a number of different ways to compare objects in general and Strings in particular. Here we will look at two different methods from the String class that can be used to compare Strings:

1. the a.*matches(b)* method. The String **matches()** method returns a *true* value if the String variable *a* is equal to the String expression *b,* otherwise, it returns a *false* value. The String expression may be a variable, a String literal, or anything that evaluates to or returns a String.
2. the *a.compareTo(b)* method. The String **compareTo()** method compares two Strings lexicographically. A **lexicographic comparison** compares two text items character-by-character using a predefined collating sequence. In the case of Java Strings, the collating sequence is the UTF-16 version of Unicode. This is similar to the way a person might compare two names to see which one comes first alphabetically. The method returns an integer value,
    a. **If the two Strings are the same**, then the compareTo() method returns the integer 0.
    b. If ***a* is less than *b***, then the method returns an integer less than 0 (a negative integer).
    c. If ***a* is greater than *b*** then the method returns an integer greater than 0 (a positive integer).

The String *matches()* method is an easy way to see if two Strings are the same. The String *compareTo()* method is useful for when we are trying to see which String comes before the other, such as when we are sorting a list of String values.

String handling is covered more in detail later in this course, and in subsequent courses. The *matches()* and *compareTo()* methods are used in the rest of this section.

Linear Search of an Array

Sometimes we wish to search an array to see if it contains a certain value. We can do this many different ways, but the simplest and most straightforward is a linear search. In a **linear search** of an array, we look at each value in the array one at a time to see if the value we are seeking is in the array. If we find the value, then we are done. If we get to the end of the array without finding the value, then the value is not in the array.

There are several different ways to implement a linear search. We will use a loop with a compound condition and a Boolean variable to keep track of whether or not the value we are seeking is in the array. In the following example, we first look at the pseudocode for the array then see the final program built from that pseudocode.

Example 18 – linear search of an array

In this example, an array contains a list of cities that have professional football teams. The user enters the name of a city to see if it is in the list. The program will perform a linear search of the array. First, a boolean variable named *found* is set to *false*. The continuation condition in the *for* loop to iterate the array will contain a compound condition, continuing while *found* is *false* and while the end of the array has not been reached. If the city is found, a message is printed and a boolean variable is set to *true* to terminate the loop. If the found variable is still *false* after the loop, a *not found* message is printed.

Using the *found* variable in the compound condition provides an advantage by making the search more efficient. With it, the loop will stop as soon as the value is found. Without it, the loop always goes through the entire array. Consider, for example what happens if the value being sought is in the first element in an array of 100 elements. With the compound condition, the loop stops after looking at the first element. Without the compound condition the loop iterates all 100 times.

The efficiency of algorithms is important, especially as data sets grow larger. Algorithmic efficiency is studied in detail CSCI 112 and 211.

```
Pseudocode for a linear search of a String array
  /*  in the search, n is the loop counter  -- the continuation condition is:
      ( n less than NFLcities.length )and (target city not found)    */
String[] NFLcities;          //  an array holding the names of cities with NFL teams
n int;                       // loop counter
boolean found = false;       // will be set to true if the city is found in the list
print intro message
get the target name of the city from the user
for (  i=0; not (found) AND n < NFLcities.length  ; n++)
   if ( target  matches NFLcities[n] )         // if the name entered matches this entry in the list.
         print NFLcity[n] has an NFL team
         set found to true
after the loop – if not(found) print  " [target] is not in the list of cities with an NFL team."
```

```
/* CSCI 111 - Fall 2013
 * NFL City Search Program
 * This program performs a linear search of a String array with
 * the names of cities that have NFL teams
 * last editied Sept. 28, 2013 by C. Herbert
 */
package nflcities;
import java.util.Scanner;

public class NFLcities {

public static void main(String[] args) {

  // an array holding the names of cities with NFL teams.
 String[] NFLcities = {"Buffalo", "Miami", "Boston", "New York", "Baltimore",
     "Cincinnati", "Cleveland", "Pittsburgh", "Houston", "Indianapolis",
     "Jacksonville", "Tennessee", "Denver", "Kansas City", "Oakland",
     "San Diego", "Dallas", "New York", "Philadelphia", "Washington", "Chicago",
     "Detroit", "Green Bay", "Minnesota", "Atlanta", "Charlotte", "New Orleans",
     "Tampa", "Arizona", "St. Louis", "San Francisco", "Seattle"};

 String target;         // the city for which we are searching
 int n;                 // loop counter
 boolean found = false; // true if the target city is found in the array

 // set up input stream from the keyboard
 Scanner keyboard = new Scanner(System.in);

 // print intro message
  System.out.print("This program will tell you if a city has an NFL team.\n\n");

 // get the target name of the city from the user
 System.out.print("Please enter the name of a city: " );
 target = keyboard.nextLine();

 // search array of NFL cities for target city
 // the loop coninues to the end of the array if the city is not found
 for (n=0 ; (!found) && (n < NFLcities.length) ; n++)
 {
     if (NFLcities[n].matches(target) )  // uses the String class matches() method
      {
          //print found message and set found to true
          System.out.println(target + " has an NFL team.\n");
          found = true;
      } // end if

  } // for loop

 // after the loop – if not(found) print not found message
 if (!found)
   System.out.println(target + " is not in the list of cities with an NFL team.\n");

    } // end main()
} // end class
```

Sorting an Array

It is useful to be able to sort an array. There are many different ways to do so, which are studied in detail in CSCI 112 and CSCI 211. We will look at a simple sorting method called a *bubble sort.*

In a **bubble sort**, we go through an array of data comparing each data item to the next item. If they are out of order, then we swap them; if they are in the correct order, then we leave them alone. We compare each item to the next item until we reach the end of the array. Each time we go through the entire array is called a *pass* through the array. We continue to make passes until we can make one pass all the way through the array without swapping anything. The bubble sort is based on the **transitive property** *of equality and inequality* (if A≤B and B≤C, then A≤B) If we can make a pass without swapping anything, then we know the array is in the correct order and we're done – the array is sorted.

We can use a boolean variable named *swapped* to keep track of whether or not we needed to swap anything in each pass through the array. Before we go through the array, we set *swapped* to false. If we swap anything, we also set *swapped* to true. When we finish a pass, if *swapped* is true, then something was swapped, and another pass is needed. If *swapped* is still false after making a pass through the array, then nothing needed to be swapped in that pass, and the array is in the correct order; the sort is done.

Here is pseudocode for a bubble sort of an array. Notice that each pass through the array only goes to the second-to-last element in the array, since we are comparing each element to the one that follows it. The last comparison will then be between the second-to-last element in the array and the last element.

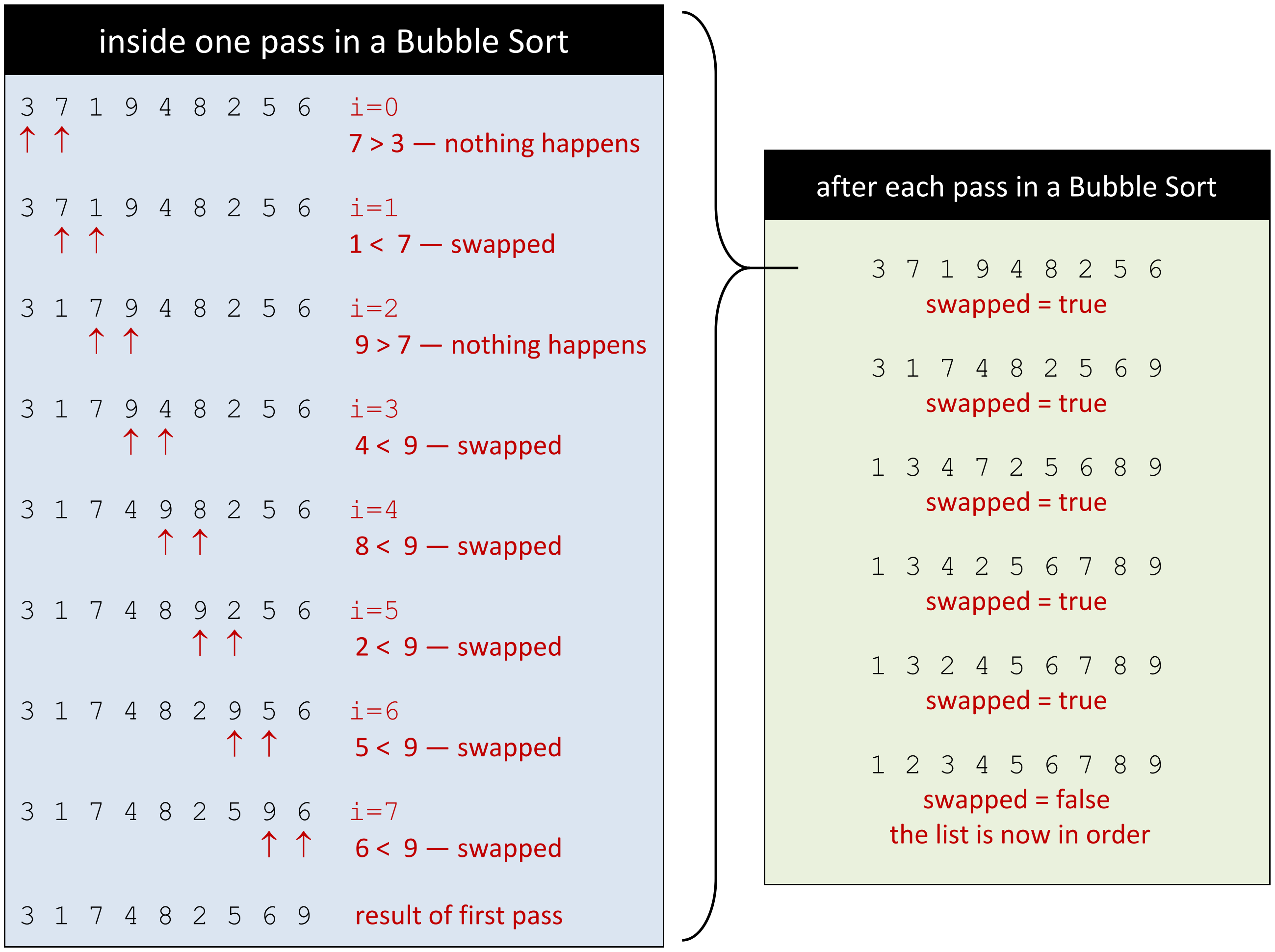
```
Start Bubble Sort – sorts in ascending order  (lowest to highest)
a[n] is an array with n elements
boolean swapped = true;   //keeps track of when array values are swapped; true to start the sort

while (swapped = true);//the outer loop will repeat each pass through the list if swapped is true
{
   swapped = false;  // set swapped to false before each pass
   for ( i=0; i < a.length – 1; i++)    // a pass through the array to the second to last element
   {
         if ( a[i+1] < a[i])      // if the two items are out of order
                  {
                  swap the two items (a[i] and a[i+1])
                  set swapped to true
                  }  // end if
   } // end for
}         // ends while  --   the outer loop will repeat if swapped is true – another pass
Stop Bubble Sort
```

The table on the next page illustrates a bubble sort on the array of integers { `3 7 1 9 4 8 2 5 6).`

On the left, we can see what happens during the first pass through the array. Starting with i=0, A[i+1] is compared to A[i]. If A[i+1] < A[i], they are swapped, otherwise, nothing happens.

On the right, we see what the list looks like after each pass. The sort stops when a pass is made without swapping anything, which means the list is in order.



inside one pass in a Bubble Sort

```
3 7 1 9 4 8 2 5 6   i=0
↑ ↑                 7 > 3 — nothing happens

3 7 1 9 4 8 2 5 6   i=1
  ↑ ↑               1 <  7 — swapped

3 1 7 9 4 8 2 5 6   i=2
    ↑ ↑             9 > 7 — nothing happens

3 1 7 9 4 8 2 5 6   i=3
      ↑ ↑           4 <  9 — swapped

3 1 7 4 9 8 2 5 6   i=4
        ↑ ↑         8 <  9 — swapped

3 1 7 4 8 9 2 5 6   i=5
          ↑ ↑       2 <  9 — swapped

3 1 7 4 8 2 9 5 6   i=6
            ↑ ↑     5 <  9 — swapped

3 1 7 4 8 2 5 9 6   i=7
              ↑ ↑   6 <  9 — swapped

3 1 7 4 8 2 5 6 9   result of first pass
```

after each pass in a Bubble Sort

```
3 7 1 9 4 8 2 5 6
swapped = true

3 1 7 4 8 2 5 6 9
swapped = true

1 3 4 7 2 5 6 8 9
swapped = true

1 3 4 2 5 6 7 8 9
swapped = true

1 3 2 4 5 6 7 8 9
swapped = true

1 2 3 4 5 6 7 8 9
swapped = false
the list is now in order
```

Swapping the Values of Two Variables

In order to implement the bubble sort, we need to learn how to swap the value of two variables. In fact, almost every sorting method requires swapping the values of variables at some point. It is a little more complicated than it seems at first. Consider the following code:

```
int a = 7;     a [ 7]
int b = 3;     a [ 7]    b [ 3]
a = b;         a [ 3]    b [ 3]
b = a:         a [ 3]    b [ 3]
```

The code doesn't swap the two variables. To swap two variables correctly, we need a third variable, called a *catalyst variable*. A **catalyst variable** is like a catalyst in a chemical reaction – we don't care what it is before the operation or after the operation, but it needs to be there to make things work properly. In this case, the catalyst variable preserves the value of one of the variables being swapped.

Consider this code:

```
int a = 7;     a [ 7]
int b = 3;     a [ 7]    b [ 3]
int c;         a [ 7]    b [ 3]    c [  ]
c = a;         a [ 7]    b [ 3]    c [ 7]
a = b;         a [ 3]    b [ 3]    c [ 7]
b = c:         a [ 3]    b [ 7]    c [ 7]
```

In this case *a* and *b* have been successfully swapped. *c* is the catalyst variable we need to do this.

Example 19 – bubble sort

The following code implements a bubble sort in Java. It has one loop inside another loop. The inner loop, the *for* loop, makes a pass through the array comparing each element to the next element. If the two are out of order, then the two values are swapped and the *swapped* variable is set to *true*. The outer loop repeats if anything swapped is true, indicating a swap was made during the pass. When a pass is made through the loop without swapping, *swapped* will remain *false* and the outer loop will end. This means the array is in order. The swapped variable is set to true before the first pass to to get things started.

In this example we will again use the array of cities with NFL teams. We will print the list, sort the list, and then print it again. The program is written in several modules – a main method, a printing method, and a sorting method. The only thing the main method does is to call the other methods. This is how good modular development works.

```
/*  CSCI 111 – Spring 2015
 * Bubble Sort Example
 *
 * This program performs a bubble sort of a String array with
 * the names of cities that have NFL teams
 *
 * it also demonstrates modular development and passing an array as a parameter
 *
 * last edited Feb. 11, 2015 by C. Herbert
 */
package bubblesort;

public class BubbleSort {

    public static void main(String[] args)  {
      //  an array holding the names of cities with NFL teams
      String[] NFLcities = {"Buffalo", "Miami", "Boston", "New York", "Baltimore",
      "Cincinnati", "Cleveland", "Pittsburgh", "Houston", "Indianapolis",
      "Jacksonville", "Tennessee", "Denver", "Kansas City", "Oakland",
      "San Diego", "Dallas", "New York", "Philadelphia", "Washington", "Chicago",
      "Detroit", "Green Bay", "Minnesota", "Atlanta", "Charlotte", "New Orleans",
      "Tampa", "Arizona", "St. Louis", "San Francisco", "Seattle"};

        // call a method to print the list of cities before sorting
        printCities(NFLcities);

        // call a method to sort the array of cities
        sortCities(NFLcities);

        // call a method to print the list of cities after printing
        printCities(NFLcities);

    } // end main()
/*****************************************************************************/
```

(The code is continued on the next page.)

(This code is continued from the last page.)

```
// this method prints the values in a string array
  public static void printCities(String[] a) {
    int i;  // used as a loop counter

    System.out.println();   // print a blank line before starting

    for (i=0; i <a.length; i++)
        System.out.print(a[i] + " ");

  } // end printcities()
/*****************************************************************************/

// this method sorts the values in an array of Strings using the bubble sort algorithm
  public static void sortCities(String[] a)   {
    boolean swapped = true; // keeps track of when array values are swapped, true to start
    int i;                  // used as a loop counter
    String c;               // a catalyst variable for swapping values of variables

    //the loop will repeat when swapped is true (until no swap occurs)
    while (swapped) {
       swapped = false;

       // each iteration of the for loop is a pass through the array to the second-to-last element
       for(i=0; i < (a.length - 1) ; i++)   {

          // if the two items are out of order
          if (a[i+1].compareTo(a[i]) < 0)  {
             // swap the two items and set swapped to true
             c = a[i];
             a[i] = a[i+1];
             a[i+1] = c;
             swapped = true;
          }  // end if

       } // end for

     } // end while

  } // end sortCities()
  /*****************************************************************************/

} // end class
```

CheckPoint 6.4

1. Why is the String class *matches()* method used to tell if two String values are equal, rather than using the comparison operator *(a ==b)*, such as for int values?
2. Describe how the String class *compareTo()* method is used to compare String values
3. Describe what a linear search of an array is and how it works.
4. Describe the transitive property of equality and inequality and why it is important in most techniques for sorting data.
5. Describe how a *bubble sort* works.

Lesson 6.5 Text File Input and Output in Java

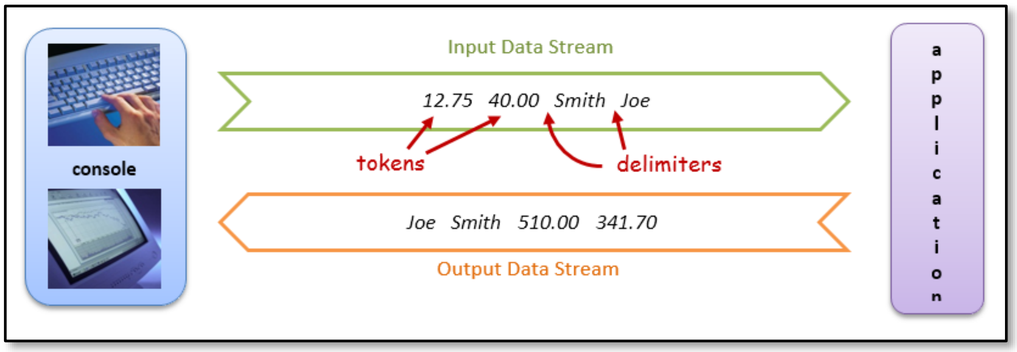
Reading from data files and writing to data files can be quite complicated. Yet, there are a few fairly simple ways to create text files using Java. In this section we will look at a simple techniques for writing text data to a file using the *PrintWriter* class, and for reading text data from a file that uses the Scanner class we have been using for console input.

Accessing a text data file is a two-step process:

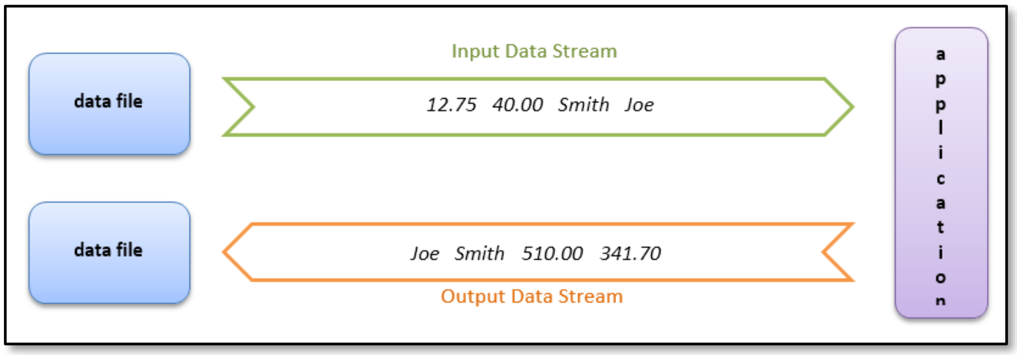
1. use a *File* class objects to establish the file we wish to work with,
2. then connect a text input stream from a file to our application using the *Scanner* class, or connect a text output stream from our application to a file using the *PrintWriter* class.

The data in files on most modern computer systems consists of electronic, magnetic, or optical signals that represent bits of information that could represent almost anything – text, numbers, sound, pictures, video, and so on. The bits need to be formatted following the rules for what they represent, such as the JPEG format for images, an IEEE 754 format for floating point numbers, or the Adobe PDF document format. How many different file formats are there? Many. The Wikipedia page online at: http://en.wikipedia.org/wiki/List_of_file_formats has a list of roughly one thousand of the most commonly used file formats.

In chapter 2 we looked briefly at console file input and output, organized as streams of tokenized text data, as shown in the following diagram.



Input and output to and from data files formatted as simple text can work the same way, except the source and destination are data files, not the console. Within the application, there is little difference, except we need to connect the data streams to files, not to the console.



We will see how to read data into our applications from text files using the features of the Scanner class, which we have already used. Methods in the Scanner class such as the Scanner class's *nextDouble()* method, can turn incoming text data into formats for other data types. (See chapter 2, page22.)

We will use methods from the *Printwriter* Class to send text data to a file. The methods are identical to the *Sytem.out* methods we have already used – *print()*, *println()*, and *printf()*.

The **File class** in the Java IO package includes methods for naming, identifying and accessing data files from within Java applications. It works in conjunction with the host operating system to deal with data files, but not with the formats of the data in the files.

The *Java IO* package with the file class is used so often that it is automatically accessible from any Java program, so we don't need to import it. The *Java Utilities* package, which contains the Scanner class, does require an import statement, such as `import java.util.*` to import the entire package, or `import java.util.Scanner` to import just the Scanner class from the package.

In the rest of this section we look at simple examples of the techniques for reading and writing text data files described above. A **Java text file** is a a **Unicode UTF-16 text file**, organized as characters based on the Unicode UTF-16 specifications. It can be read by almost any modern text editor or word processor. If you send the files to most printers or computer screens, they will display or print the characters in the file so that we can read them. Accessibility tools for the physically disabled, such as *text to speech* text readers, can work with Unicode text files. Almost all modern operating systems have the ability to handle Unicode text files.

Handling Data File Exceptions like Hot Potatoes

Before we start, there is one more thing about data file programming that we need to know. Java requires us to tell it how we are going to handle any exceptions that occur when we are working with data files.

An **exception** is an object that is created whenever a run-time error occurs in a Java program. A **run-time error** is an error that occurs while an application is running, not when the program is being complied. There are different classes of exception objects, such an *ArrayIndexOutOfBoundsException* if an application tries to access *employee[300]* when the array *employee[]* only has 100 elements, or a *FileNotFoundException* if an application tries to read from a data file that does not exist. The type of exception that occurs and properties stored in the exception object contain information that might be useful to software analysts in finding and correcting software errors.

Some data file exceptions will generally cause our programs to crash, or worse, they could mess up other files, perhaps even disabling a hard drive or causing a computer system to crash. Because of this, Java requires us to tell it how we plan to handle file I/O exceptions.

There are generally two ways to handle exceptions in a Java method – the first is to create an **exception handler**, which is a method that handles an exception. What it does depends on the nature of the exception. This is known as catching an exception. To **catch an exception** is to transfer control to an exception handler when an exception occurs. We aren't yet ready to start catching exceptions. Exceptions will be discussed briefly near the end of the semester and in more detail in CSCI 112.

The second way to handle an exception is to throw the exception. To **throw an exception** is to send it back to whatever software invoked the method in which the exception occurred. For the main methods in our Java applications, most of the time this will be the computer's operating system. We can think of this as the "hot potato" method of dealing with an exception. We don't know how to handle it, so we throw it back to someone else to handle. We will throw file exceptions back to the operating system to handle.

This is pretty easy to do. We simply add the "*throws Exception*" clause to the end of a method header. For now, we need to do this whenever we access data files in a method, and in our main method if it calls any methods that handle data files. This will throw the exception like a hot potato back to the main method, which will in turn throw it back to the operating system.

Here is an example of a method header with a "throws Exception" clause: (notice the capital E)

```
        public static void main(String[] args) throws Exception
```

We should add the "*throws Exception*" clause to any method that accesses a data file, or that calls another method which access a data file, or that calls a method that calls a method that accesses a file, or that calls a method that calls a method that calls a method that calls a method that … you get the picture.

The operating system is most likely going to halt our application if we throw it an exception, but at least we won't accidently wipe out a hard drive, crash the entire computer, bring the Internet to a grinding halt, or accidently start a global thermonuclear war because of a run-time error in what should be a relatively simple Java application.

A Technique for Writing Text Data Files using Java

The File class in the Java IO package is used in conjunction with classes that manage the formats of files to read and write data. The File class manages the names and locations of data files from within Java applicaaions, but other classes handle the format of data moving to and from files.

The **PrintWriter class** in the Java IO package includes methods for sending out a stream of data formatted as text. it has *print()*, *println()* and *printf()* methods virtually identical to the System.out *print()*, *println()* and *printf()* methods with which we are already familiar. To quote a famous Saturday Night Live character, "*How convenient!*"

We can create instances of objects from the *File* class and the *PrintWriter* class to write to a data file. They work together – the File class identifies the file and the Printwriter class sends out the data to the file as a stream of formatted text, just as *System.out* sends data to the console.

The easiest way to see how to do this is to look at examples. The following two examples are contained in NetBeans folders in the files for week 6. You should become familiar with how these two programs work.

Example 20 – writing data to a data file

The code below sends the message “Hello World!” to a text data file. The file will be stored in the NetBeans folder for your project. After the program runs, you should be able to open the folder and look at it with NotePad, WorPad, Word, or any application that can read a text file.

```
/*  CSCI 111 - Fall 2013
 * writing "Hello world!" to a data file
 * this program demonstrates writing to a text file
 * last edited Sept. 28, 2013 by C. Herbert
 *
 * warning -- this will overwrite the file "hello.txt"
 */

package writehello;

public class WriteHello {

    public static void main(String[] args) throws Exception {
        // create a File class object and give the file the name hello.txt
        java.io.File x  = new java.io.File("hello.txt");

        // Create a PrintWriter text output stream and link it to the file x
        java.io.PrintWriter y  = new java.io.PrintWriter(x);

        // Write text output to the file using print(), println() or printf()
        y.println("Hello World! ");

        System.out.println("File written...");
        System.out.println("Open the NetBeans folder for this project and then");
        System.out.println("open the file hello.txt to see the result.\n");

        // close the data stream and associated file
        y.close();

    } // end main()

} // end class
```

A Technique for Reading Text Data Files using Java

We can use the File class and Scanner class together to read data from a text file. The File class is used to connect to data files from within Java applications and the Scanner class is used to read the input data stream from the file, just as we did from the console. It’s trickier than writing to a file because we need to know the name of the file, where it is, and what’s in the file to make it work properly.

Example 21 – reading data from a data file

In the example below, we will read a single line of text from a data file named *Hello.txt* that is stored in the NetBeans project folder for the application.

```
/* CSCI 111 - Fall 2013
 * reading "Hello world!" from a data file
 * this program demonstrates reading from a text file
 * last edited Sept. 28, 2013 by C. Herbert
 *
 * the file "hello.txt" must exist in the project folder for this to work
 */

package readhello;
import java.util.Scanner;

public class ReadHello {
  public static void main(String[] args) throws Exception {

    String message;     // holds the line of text coming in from the file

    // Create a File class object x and give it the name of the file to read
    java.io.File x = new java.io.File("hello.txt");

    // Create a Scanner named y to read the input stream from the file x
    Scanner y  = new Scanner(x);

    // Read a line of text from the file
    message = y.nextLine();

    // print the message from the file on the screen
    System.out.println("reading from the data file...\n");
    System.out.println(message);

    // Close the input data stream and associated file
    y.close();

  } // end main()
} // end class
```

CheckPoint 6.5

1. How are Java's File class and Scanner class used together to read text data from a data file?
2. How are Java's *File* class and *PrintWriter* class used together to write text data to a data file?
3. What format does Java to store data in text files?
4. What is an *exception* in Java software?
5. Describe the difference between catching and throwing java exceptions?

Lab 6 – Programming Example: Moving Data between Arrays and Files

In this example we will see how to construct an application to read data from a data file into an array, sort the array, then write the data back out to another data file from the array.

We will use the data files *unsorted.txt*, which contains a list of tutorials for different programming languages. Each line lists a different tutorial. Our task is to read the file into an array of Strings, sort the array alphabetically, then write the list to a new data file named *tutorials.txt*

This example uses most of the things covered in this chapter along with development of a modular program.

First let's look at the contents of the text file *unsorted.txt*:

```
Java - The Java Tutorials - http://docs.oracle.com/javase/tutorial/
Python - Tutorialspoint Java tutorials - http://www.tutorialspoint.com/python/
Perl - Tutorialspoint Perl tutorials - http://www.tutorialspoint.com/perl/
C++ - cplusplus.con C++ tutorials - http://www.cplusplus.com/doc/tutorial/
Objective C - The objective C programming tutorials - http://howtomakeiphoneapps.com/objective-c-tutorial/
C++ - the LearnCPP C++ Tutorial - http://www.learncpp.com/
Objectice C - The Objective C tutorial - http://objectivectutorial.org/
Java - LearnJavaOnline.org Interactive Java Tutorial - http://www.learnjavaonline.org/
Objective C - Objective C beginner's Guide - http://www.otierney.net/objective-c.html
C# - Microsoft's C# Tutorials - http://msdn.microsoft.com/en-us/library/aa288436%28v=vs.71%29.aspx
Python - The Python Tutorial - http://docs.python.org/2/tutorial/
Python - Learn Python interactive Tutorial - http://www.learnpython.org/
COBOL - The ZingCOBOL beginners guide to COBOL Programming - http://cobol.404i.com/
Java - Tutorialspoint Java tutorials - http://www.tutorialspoint.com/java/
C++ - Tutorialspoint Java tutorials - http://www.tutorialspoint.com/cplusplus/
Ruby - Ruby in Twenty Minutes - https://www.ruby-lang.org/en/documentation/quickstart/
PHP - PHP: a Simple Tutorial - http://php.net/manual/en/tutorial.php
JavaScript - W3 Schools Javascript Tutorial - http://www.w3schools.com/js/default.asp
Ruby - The Ruby DOC Introduction to Ruby - http://ruby-doc.org/docs/Tutorial/
JavaScript - JavaScript for the Total Non-Programmer - http://www.webteacher.com/javascript/
```

We can see that each line in the file contains the name of a language, followed by the name of an online tutorial for that language, followed by the URL for the tutorial.

We will first read the data into an array of Strings, with each line as a separate element in the array, then sort the array, then print the array. Each of these tasks will be complete within its own method. We will also print the array on the console after reading it in and after sorting it.

Pseudocode – sorting a text file line by line

```
start main() method

    String[] tutorials          an  array of Strings listing programming language tutorials
    int count                   the number of elements in the that are used

    call a method to read data into tutorials[] and return count
    call a method to print the array
    call a method to sort the array
    call a method to print the array
    call a method to write the data to a file
stop main method
/**************************************************
```

This method reads data from a file into an array. We want our array to work with up to 100 elements. Each line from the file will be an element in the array. The method returns the number of elements used.

```
start method  int readList (String[] lines)

    int count = 0;          // count lines read

    set up file access with File object
    set up input Stream from the file object

    while (the next line in the file exists)   {
        lines[count] = read next line from the file
        count++;

    } // end while
    close file
    return count    // returns the number of items used in the array.
stop readlist
/**************************************************/
```

This method sorts an array of Strings line by line using a simple bubble sort. The first parameter is the array. The second parameter is the number of elements in the array that actually contain data.

```
start method  sortStringArray (String[] lines, int count)

    boolean swapped = true ;    // keeps track of when array values are swapped, true to get started
    int i;                              // a loop counter

    // Each iteration of the outer loop is a pass. If any swap is made in a pass, it repeats another pass.
    while (swapped) {
       set swapped to false before each pass
       for ( i=0; i < count – 1; i++)   {    // a pass through the array to the second to last element
         if ( a[i+1] < a[i])  {                // if the two items are out of order, swap the two items
             swap (a[i] and a[i+1])
             set swapped to true
             }  // end if
       }  // end for
    }  // end while --  the outer loop will repeat if swapped is true – another pass
stop sortStringArray
```

This method prints an array of Strings on the screen. The first parameter is the array. The second parameter is the number of elements in the array that actually contain data.

```
start method printTextArray( String[] lines, int count)

    for ( i=0; i < count – 1; i++)    // a pass through the array
        print an element from the array
stop method writeTextArray
/************************************************
```

This method writes an array of Strings to a text data file. The first parameter refers to the array in the main method. The second parameter is the number of elements in the array that actually contain data.

```
start method writeTextArray( String[] lines)

    set up file access with File object
    set up input Stream from the file object

    for ( i=0; i < count – 1; i++)    // a pass through the array to the second to last element
        write an element from the array to line in the file
    close file
stop method writeTextArray
```

The next step is to convert the pseudocode to actual Java Code. The code is shown on the next page. Here is what the sorted data file should look like:

```
C# - Microsoft C# - http://msdn.microsoft.com/en-us/library/aa288436%28v=vs.71%29.aspx
C++ - Tutorialspoint Java tutorials - http://www.tutorialspoint.com/cplusplus/
C++ - cplusplus.con C++ tutorials - http://www.cplusplus.com/doc/tutorial/
C++ - the LearnCPP C++ Tutorial - http://www.learncpp.com/
COBOL - The ZingCOBOL beginners guide to COBOL Programming - http://cobol.404i.com/
Java - LearnJavaOnline.org Interactive Java Tutorial - http://www.learnjavaonline.org/
Java - The Java Tutorials - http://docs.oracle.com/javase/tutorial/
Java - Tutorialspoint Java tutorials - http://www.tutorialspoint.com/java/
JavaScript - JavaScript for the Total Non-Programmer -
http://www.webteacher.com/javascript/
JavaScript - W3 Schools Javascript Tutorial - http://www.w3schools.com/js/default.asp
Objectice C - The Objective C tutorial - http://objectivectutorial.org/
Objective C - Objective C beginner's Guide - http://www.otierney.net/objective-c.html
Objective C - The objective C programming tutorials -
http://howtomakeiphoneapps.com/objective-c-tutorial/
PHP - PHP: a Simple Tutorial - http://php.net/manual/en/tutorial.php
Perl - Tutorialspoint Perl tutorials - http://www.tutorialspoint.com/perl/
Python - Learn Python interactive Tutorial - http://www.learnpython.org/
Python - The Python Tutorial - http://docs.python.org/2/tutorial/
Python - Tutorialspoint Java tutorials - http://www.tutorialspoint.com/python/
Ruby - Ruby in Twenty Minutes - https://www.ruby-lang.org/en/documentation/quickstart/
Ruby - The Ruby DOC Introduction to Ruby - http://ruby-doc.org/docs/Tutorial/
```

```
/*  CSCI 111 - Fall 2013
 * reading, sorting and writing data in text files
 * this program reads data from a text file, sorts the data,
 * then writes the data back to another text file.
 * last edited Oct 1, 2013 by C. Herbert
 *
 * for this to work, the file "unsorted.txt" must be in the project folder
 * warning -- this will overwrite the file "tutorials.txt"
 *
 * This program has methods to read lines from a text file into an array,
 * display a text array on screen line-by-line, sort a text array, and write
 * a text array to a data file line by line.
 *
 * The program is limited to a file with 100 lines.   To change this, change the
 * size of the array declared in the main method.
 *
 */
package tutorials;
import java.util.Scanner;
import java.io.Scanner;

public class Tutorials  {

    // the main method call methods to perform each part of the program
    public static void main(String[] args) throws Exception  {
        String[] tutorials = new String[100];  // an array to hold a list of tutorials
        int count;                             // the number of elements actually used

        // read data into tutorials[] line by line and return count
        count = readLines(tutorials);

        // print the array on the screen
        System.out.println("The original file:");
        displayLines(tutorials, count);

        // sort the array
        sortStringArray(tutorials, count);

        // print the array on the screen line by line
        System.out.println("\nThe sorted file:");
        displayLines(tutorials, count);

        // write the array to a data file line by line
        writeLines(tutorials, count);

    } // end main()
/**************************************************/

    /* This method reads data from the file into the array.
     * We want our array to work with up to 100 elements
     * Each line from the file will be one element in the array.
     *
     * The parameter refers to the array in the main method.
     *
     * The method returns the number of elements it uses.
     */
     public static int readLines(String[] lines) throws Exception  {
        int count = 0; // number of array elements with data

        // Create a File class object linked to the name of the file to read
        File unsorted = new File("unsorted.txt");

        // Create a Scanner named infile to read the input stream from the file
        Scanner infile = new Scanner(unsorted);

        /* This while loop reads lines of text into an array. it uses a Scanner class
         * boolean function hasNextLine() to see if there another line in the file.
         */
        while ( infile.hasNextLine() ) {
            // read a line and put it in an array element
            lines[count] = infile.nextLine();
            count ++;  // increment the number of array elements with data
        } // end while

        infile.close();
        return count;    // returns the number of items used in the array.

    } // end readList()
/**************************************************/

    /* This method sorts an array of Strings line by line
     * using a simple bubble sort.
     *
     * The first parameter refers to the array in the main method.
     * The second parameter is the number of elements in the array that
     * actually contain data
     */
    public static void sortStringArray(String[] a, int count) {
        boolean swapped = true; // keeps track of when array values are swapped
        int i;                  // a loop counter
        String temp;            // catalyst variable for String swapping

        // Each iteration of the outer do loop is one pass through the loop.
        // If anything was swapped, it makes another pass.
        while (swapped) {
            // set swapped to false before each pass
            swapped = false;

            // the for loop is a pass through the array to the second to last element
            for( i=0 ; (i < count-1) ; i++ ) {

               // if the two items are out of order
                if(a[i+1].compareTo(a[i]) < 0)  {
                    // swap the two items and set swapped to true
                    temp  = a[i];
                    a[i] = a[i+1];
                    a[i+1] = temp;
                    swapped = true;
                }  // end if
            } // end for
        } // end while
    } // end sortStringArray
/*******************************************************************/
```

```
    /* This method prints an array of Strings on the screen.
     * The first parameter refers to the array in the main method.  The second
     * parameter is the number of elements in the array that actually contain data
     */
     public static void displayLines(String[] lines, int count)
     {
       int i;  // loop counter

        // iterate the elements actually used
         for ( i=0; i < count; i++)
                System.out.println(lines[i]);

     } // end displayLines()
/**************************************************/

     /* This method writes an array of Strings to a text data file. The first
      * parameter refers to the array in the main method. The second parameter
      * is the number of elements in the array that actually contain data
      */

     public static void  writeLines(String[] lines, int count) throws Exception  {
       // create a File class object and give the file the name tutorials.txt
       File tut  = new File("tutorials.txt");

       // Create a PrintWriter text output stream and link it to the file x
       PrintWriter outfile  = new PrintWriter(tut);

       // iterate the elements actually used
            for (int i=0; i < count; i++)
                outfile.println(lines[i]);

       outfile.close();

        } // end writeTextArray()
/**************************************************/

} // end class
```

Key Terms

Chapter Questions

1. If the first element in an array is an integer, what do we know about all of the elements in the array? Why are array elements referred to as subscripted variables?
2. What is the highest index in an array of 20 elements? What is the lowest index?
3. What is the difference between a linear array and a multi-dimensional array? How can we represent a table with rows and columns as an array?
4. How are variables matched with memory locations? What are other some items that need to be mapped to specific addresses in a computer's internal memory? How is memory numbered in most computer systems?
5. How does an operating system keep track of where an array is in memory? What is an array's base address? If an array named *points* of data type *int* has a base address of 3000, what is the address of element *points[2]*?
6. What are the two parts to declaring and array? What is a common way of including both parts in the same Java statement?
7. If an array has a primitive data type, what does each element of the array contain? If the data type of an array is an object, what does each element contain? What is actually in an array of *double* values in Java? What is actually in an array of Strings in Java? What will be printed if we use a variable from a String array in a print statement, such as `System.out.println( prices[2] ); ?`
8. What is another name for an alternate array declaration? How are values assigned to array elements in an alternate array declaration? How is the length of the array determined when using an alternate array declaration?
9. How is an array passed as a parameter? If an int array is passed as parameter and we change the value of an element in the array, what happens in the original array? How is an individual element of an array passed as a parameter?
10. What is a good way to initialize all the elements of a large array with random numbers? What is a good way to initialize all the elements of a large array with the same value?
11. What array method can be used to help us determine the size of an array? How can it be used to iterate an array?
12. How can we find the sum of the values in a numeric array? How can we find the average of the values in a numeric array?
13. How can we find the minimum of the values in a numeric array? How can we find the maximum of the values in a numeric array?
14. What does Java's *enhanced for statement* do? What does the integer variable in *enhanced for statement* refer to each time through the loop?
15. What two String class methods can we use to see if two Strings have the same value? Which of these methods can be used to see if one String comes before another lexicographically? What are the meanings of the values that this method returns?

16. How can we implement a linear search of an array in Java? What advantage is provided by using a Boolean *found* variable in a compound condition for a search loop?
17. How does a bubble sort work? What happens during each pass in a bubble sort? When does the algorithm stop making passes in a bubble sort?
18. What happens if we try to swap the values of two variables with statements like *a=b; b=a; ?* What is a catalyst variable? How is a catalyst variable used to swap the values of two variables?
19. What is the *File class* used for in in accessing text data files from a Java application? What can the *Scanner class* be used for in accessing text files? What can the *PrintWriter class* be used for in accessing text files?
20. What is an Exception in a Java program? What is a run-time error? In what two ways can a Java application deal with exceptions?

Chapter Exercises

1. If the memory space for variables starts at memory location 1000, show what the symbol table would look based on the following set of declarations:

```
int count;
double average;
int[] scores = new int[20];
```

2. Create a NetBeans application that asks the user for five test scores, then calculates the average score, the highest score and the lowest score. Your application should read the five test scores into an array of integers, and should output a neatly formatted report to a text file including the scores, the average, the highest score and the lowest score. Your program should exhibit good modular development. Be sure to include the output file with your NetBeans project folder and lab report.

3. Exercises 1 through 9 at the end of Chapter 4 each ask you to write a program to print a table of data or a set of data using console output. Re-write any one of those programs to send the output to a data file. Be sure to include the output file with your NetBeans project folder and lab report.

4. Create a NetBeans application that fills an array with 1,000 random integers, each between 1 and 10. Your program should then perform a frequency count – How many elements are equal to 1, how many are equal to 2, etc. for all 10 possible values. Your program should also calculate the average of the values. Your application should output a neatly formatted summary report to a text file with the summary results, not the entire array. Your program should exhibit good modular development. In your lab report, you should describe how the results compared to what you expected the results to be.

5. The file *enrollments.txt* in the files for Chapter 6 is a text file with each line containing a positive integer that represents the enrollment in a section of CIS 103. The highest value is 36. Write a program to read the data from the file into an array, then calculate and display the number of sections, the average class size, the minimum class size, and the maximum class size.

6. Using the *enrollments.txt* file mentioned in number 5, above, count and display the number of full sections and the percentage of the sections that are full. A section is full if it contains 36 students.

7. The data structure known as a stack reverses the order of things put on the stack, like a stack of dishes – the last item put on the stack will be the first taken off the stack. The data file *months.txt* contains the names of the twelve months in order. Write a program that works like a stack to read the data from a file into an array, then write the data to a new data file in reverse order.

   Parallel arrays –described here – are used in exercises 8, 9 and 10. Parallel arrays are two arrays whose contents are coordinated so that element *i* in one array is somehow related to element *i* in another array, such as a list of names and phone numbers. Parallel arrays are not as common as they previously were, because of the use of objects in modern programming, such as an array of *person* objects with *name* and *phone number* properties in the object. You should use good modular development in creating your application as you should in all applications.

8. In this exercise you should create an old fashioned parallel array program to read data from a file into two parallel arrays – an array of the names of states and an array of state capitals. The data file, named *capitals.txt* contains a list of states and capitals, with a state name on one line followed by a capital name on the next line, such as:

   ```
   Alabama
   Montgomery
   Alaska
   Juneau
   Arizona
   Phoenix
   Arkansas
   Little Rock
   ```

   Your application should read the state names into one array and the capital names into a parallel array. Your code should then ask the user to enter a state and search the state array for the state. If the name entered is in the array, then tell the user the name of the state capital from the parallel array. If the name entered is not in the array, then tell the user this.

9. Using parallel arrays, create a quiz asking people to name state capitals. You should read the data from the *capitals.txt* data file into parallel arrays, then randomly select one of the elements in the states array and ask the user to enter the name of the capital. You should then compare the answer the user entered to the corresponding element in the parallel array of capitals. Tell the user if the answer is correct, or, if it is wrong, then tell the user so, and tell the user the correct answer.

10. The data file *careers.txt* contains a list of job titles and average annual salaries on alternating lines in the file, such as:

    ```
    Computer and Information Research Scientists
    102190
    Computer and Information Analysts
    80460
    Computer Systems Analysts
    79680
    Information Security Analysts
    86170
    ```

    The file has real data from the US Department of Labor's May 2013 survey of occupations and wages. In this exercise you should create two parallel arrays, one for job and one for salary, and read data from the file into the arrays – one line into the job array and then one line into the corresponding element in the salary array. Your program should sort the lists in parallel according to the values in the salary array, with the highest salary first. *Sorting in parallel* means that in your sort, if you swap two elements in the salary array, you must also swap the corresponding elements in the job array. Your program should then display the list of jobs and corresponding salaries on the screen in a neatly formatted table, with commas in the salaries.

— End of Chapter 2 —