9. Arrays and Strings

In object-oriented systems, the architectural view is that of a number of objects interacting via function calls in order to handle the program's inputs.

The question is, where do you put all these objects? They can be held/referenced by other objects as member fields. However, where are those holders held? Static member attributes can hold addresses to some. But often, we need more sophisticated containers in which to hold bunches of them.

In this section, we study two of the basic collections used most frequently by Java programmers: arrays and strings.

Arrays are fixed size containers that can contain a number of elements of a specific type (sort of like an ice cube tray: it has a lot of compartments all designed to hold exactly the same kind of thing).

Though you can put a sequence of characters like "Russ Tront" into an array of 'char', Java provides both String and StringBuffer classes that are already written for you and which provide a wonderful variety of powerful and sophisticated features.

Required Readings: Savitch[2001] Chapters 6 and 2.1

Section Table Of Contents

9.1 Introduction

Java provides a number of very convenient simple collection mechanisms. Only two are syntactically special:

- arrays – similar to those in other languages, but they seem more object-like in Java. In addition, they are specifically checked on every access so that a program cannot access passed the end of an array.
- String class – a normal class except that the "+" operator is also used to provide string concatenation. E.g. "Russ " + "Tront". If this operator has a String as one operand, it will trigger the toString( ) method of its other operand if the other operand is not a string. This is what makes Strings different from other normal classes.

These are syntactically special because they are so useful and commonly used that the Java designers decided to make them even more special by providing useful additional features that require special syntax.

Other collection types that we may or may not cover are Vectors, HashTables, Sets, Lists, Queues, Dictionaries, etc. For more information on the Collections Framework which is part of the Java Foundation Classes, see:

http://java.sun.com/j2se/1.4/docs/guide/collections/index.html

9.2 Arrays

An array is a complete row of similar size compartments that you can treat as one. An ice cube container comes to mind. Though most arrays that you will use in computer languages have only one row, most languages are capable of 2 dimensional, 3 dimensional, and even 7 dimensions!

Like most programming languages, Java provides syntax for defining and using fixed-size arrays. In most computer languages, an array's size is fixed at compile time, when you write the program. Others languages allow you to set size at run time, or either time.

However, the sizes of all Java arrays are only set at run, rather than compile time. Like pointers to dynamically allocated arrays in C/C++, they can be created with the particular size needed that day, or for that iteration of the program. This is nice and saves memory on some days. However, you can***not*** change the size of an array *once it has been created*!

Though Java arrays are not really a class, they seem somewhat like a class. For example, you create an array very much like calling a constructor, with the keyword 'new'. You also must use square brackets indicating the size of the array you want allocated:

```
int[] myArray = new int[3];

//or the older:

int myArray[] = new int[3];
```

Most Java programmers use the newer int[ ] style.

The above definitions create a reference to an array of 3 integers, and also create the array itself in the heap area of RAM memory.

Each array element does not get its own name. Instead, you use an 'index' number in square brackets to refer to particular compartments in the array.

```
myArray[0]
myArray[1]
myArray[2]
```

Each compartment contains what is called an array 'element' of specifically the type defined (e.g. int).

You can assign a new value to, and get the present value of, an array like this:

```
myArray[1] = 2;

int i = myArray[1];
```

Note that the array index does not have to be a literal or named constant.

```
myArray[i] = 99;

int j = myArray[i];
```

This is some of what gives arrays their power as you see in the next subsection.

Like C/C++, Java array indices start at 0. There is no element at myArray[3]. If you try to read or write an array element outside of the defined index range, unlike less safe languages, Java will throw an ArrayIndexOutOfBoundsException. Though we have not studied Exceptions yet, if uncaught an exception will cause Java to halt the program and print out a short error message. This adds considerable safety to the Java language. This safety is not just due to reading and writing portions of memory outside of the array and having that mess up your program, but also makes network downloaded applets safer from being able to access various places in the RAM memory of your computer.

9.2.1 Arrays Are A Special Kind of Object

Arrays seem like a special subclass of the Object class. However, it is just as appropriate to think of them as a collection of variables all of the same type.

Though you cannot control what some of their member fields contain, you can access some of the member fields of an array. One that is always available right in every instance is the length of the array.

```
System.out.println(myArray.length);

for(int i=0; i<myArray.length; i++){

  myArray[i] = 10*i;

  System.out.println(i + " " + myarray[i]);
}
```

Here you see the use of the special member attribute called 'length'. You can also see the use of a variable as an array index, so the program itself can select which array element it wants at any particular point.

Note that the end of the loop should be when

(i < myArray.length), and

not

(i <= myArray.length) (why?).

Also, like an object instance, a whole array is like an object reference. Assigning one array variable to another does

NOT make a copy of the array. It is just that the two references both refer to the same array.

If you pass an array as an argument to a method, the method's formal parameter is assigned a reference to the array instance, including its accessible length field. Partly because of this, a method definition does NOT have to specify the size of the array it is expecting as an extra formal parameter.

Arrays can be returned by functions; like other Java objects they are returned by reference.

If you do want to make a copy of an array, you must call a particular static member function of the System class:

```
System.arraycopy(fromArray,
                 fromStartIndex,
                 destinationArray,
                 destStartIndex,
                 numberOfElementsToCopy);
```

Of course, just like any object, checking for equality will just check whether the two references point to the same array. If you want to check whether references to two different arrays have the exact same contents, you need to do this:

```
import java.util.Arrays;
boolean same;
same = Arrays.equals(oneArray, anotherArray);
```

The above function checks whether the arrays are the same type, length and contents.

There are other static methods in the Arrays class: sort, binarySearch, and fill. However, all of these are designed to work on arrays of primitives, NOT arrays of objects!

Nonetheless, you can create arrays of any kind of object class. In essence, you end up with a reference to references.

Note: When passing an array to as a function parameter, essentially it is passed by reference. In other words, arrays behave similar to objects. Only the address of where the array is physically located in heap RAM is passed in a parameter. This allows a function to change the values in the array, and when the function ends, the caller finds different values in the array than when the call was started.

9.2.2 Initializing Arrays

By default, arrays elements are initialized to zero or null.

Like C/C++ there is a shorthand form method of initializing an array to specific values.

```
int[] myArray = {8, 17, 53, 100};
String[] s = {“Bill”, “Doug”, “Harry”};
```

The nice thing about this notation is that it even counts the number of initializers and makes the array the exact correct size.

Note: If you create an array of objects, it is really a reference to an array of references. Thus to initialize such an array properly, you need to do something like the following:

```
Airplane[ ] myArray = new Airplane[3];

myArray[0] = new Airplane( );

myArray[1] = new Airplane( );

myArray[2] = new Airplane( );
```

9.2.3 Multi-Dimensional Arrays

Like most good programming languages, Java supports multi-dimensional arrays. A 2 dimensional array provides a grid or 2 dimensional matrix of elements into which you can put values/references. e.g.

```
int [][] twoD = new int[3][5];
                //3 rows by 5 columns.

twoD[0][4] = 57;                //set element.
System.out.println(twoD[0][4]); //get element.
```

Both the row and the column index are zero based.

Note: A two dimensional array is really just one dimensional array of one dimensional arrays. So,

twoD.length is the number of rows;

twoD[0].length is the number of columns in the first row.

9.3 Strings

Java has wonderful facilities for working with strings, even strings written in other character sets. As you learned earlier, type 'char' is a 16-bit Unicode encoded character.

The string support in Java is in three parts.

- First, there is the class String that encapsulates most of the functionality. Instances of the String class are immutable, meaning you can NOT change the contained string value once it has been initialized.
- Secondly, the '+' operator is overloaded so that if one operand is a string, the other will be converted into a string if possible using the member function toString inherited or overridden from root class Object.
- Third, there is a StringBuffer class that provides a mutable string that you can append to, insert within, etc.

9.3.1 The String Class

The class that provides most of the string support in Java is called String.

```
String s1 = new String( );
String s2 = new String("Joan Smith");
String s3 = "Joan Smith";
```

The first statement above sets the string s1 to the empty string "". Note that the empty string is not the same as the reference just being null.

The latter two statements are equivalent. The literal "Joan Smith" is just a literal whose natural type is String, so you are just copying the reference to that string to the string s3.

Note that the following:

```
s1 = s3;
```

does NOT make a new copy of the string. Really, both s1 and s3 simply point to the same string.

If you want to check that two string reference variables refer to (i.e. contain the address of) the same string object instance, just use:

```
if(s1 == s3 )
  System.out.println("references equal");
```

On the other hand, if you want to see if two different String instances have the same *contents*, then use:

```
if( s2.equals(s3) )
  System.out.println("contents equal");
```

Note: equals( ) is a widely used instance member function provided by many classes.

9.3.2 String Concatenation and Conversion

A string can be appended onto the end of another string using the '+' operator. This is called 'concatenation'. e.g.

```
String s = "Bill";
String t = s + " " + "Smith";
```

Note that when concatenating strings, you often have to remember to add a space if you want a space.

If Java finds a string operand concatenated to a non-string operand, it will try to convert the non-string operand (be it a primitive, or some other class) to a string. It will then concatenate them together. e.g.

```
int i = 3;
Exception e = new Exception("buggy");
System.out.println("integer i is " + i);
System.out.println("e.toString() is " + e);
```

will print:

```
integer i is 3
e.toString is java.lang.Exception: buggy
```

This is quite unusual. In most programming languages you cannot add a string and an integer (which is internally stored in unintelligible 2's complement form).

For primitives, Java will do the conversion with, for example, the static function Integer.toString(theIntegerVariable) or String.valueOf(theIntegerVariable).

For objects, the conversion is done using the toString( ) member function that all object's inherit from the root Object class.

Most classes in the existing Java class libraries have overridden the root toString( ) member function, so it is available for use. As shown above, the Exception class is a good.

Unfortunately, arrays do not convert to strings well at all. In addition, classes that you write may not convert to strings very well (unless you write a toString( ) instance member function for your class). If a class does not have a proper custom toString( ) member function, I think Java just arranges to print out the contents of the reference variable (which contains the address of the object?).

If you want to print out the contents of an array, or your own custom class instance, you have to write a function to print out the contents one field at a time.

Note, to convert a primitive variable, like an integer (which contains the strange 2s complement representation of the number), you can use the static member function

```
Integer.toString(99);
```

```
Integer.toString(myIntegerVariable);
```

Note that all the primitive types have a corresponding class with such static helper functions in them.

The PrintStream class's print( ) and println( ) methods are overloaded. That means there are many different versions of them, each with a different parameter list. e.g.

```
print(String)
print(int)
print(float)
print(char)
print(boolean)
```

So usually you don't have to convert a primitive to a string to get them to print.

If you want to convert them to a string to display on a GUI window panel, or you have to write a toString( ) method for a complex class containing several primitives, you may want to know about these static conversion functions:

```
String.valueOf(boolean)
String.valueOf(char)
String.valueOf(char[])
String.valueOf(int) //handles byte, short too!
String.valueOf(long)
String.valueOf(float)
String.valueOf(double)
```

Here is a list of conversions functions for converting the other way, from string to a primitive:

```
new Boolean(String).booleanValue( )
Byte.parseByte(String)
Short.parseShort(String)
Integer.parseInt(String)
Long.parseLong(String)
Float.parseFloat(String)
Double.parseDouble(String)
```

Note that in Java 1.1 and earlier, there was no parseFloat( ) and parseDouble( ). Instead you had to use a round-about method similar to that shown for booleans above.

Also note that there are overloaded versions of parseByte( ), parseShort( ), parseInt( ), and parseLong( ), which allow reading in strings written in binary, octal, hexadecimal bases, etc.

These conversions from strings to primitives are very important, as Java has no facilities for formatted input such as C's scanf( ) function, or C++'s overloaded input extraction operator. If you have to read several numbers from a single line of input that has been read into a string, you must break the string into tokens using the StringTokenizer or StreamTokenizer classes.

9.3.3 String Editing

There is a large set of member functions in the String class devoted to string editing. Here are just a few:

- length( ) – returns the length of a string.
- substring( ) – returns a new string which is a copy of a portion of another string.
- toLowerCase( ) – returns a new string which is similar to this string, except converted to lower case.
- endsWith(suffix) – checks if a string ends in a particular suffix.
- indexOf(str) – returns location of a sub-string within the string.
- regionMatches( ) – searches only a portion of a string for a substring.
- compareTo(String other) – check which string is alphabetically 'before' the other.
- charAt(n) - returns the n'th character in the string (note that the numbering starts at zero).

LARGE NOTE: You can use these functions to create new strings from parts of existing strings. You CANNOT change an existing string. The character array field inside a string instance is labeled final. This is an advantage as you can pass a reference to a string around, even in a multi-threaded program, and bad things don't happen.

But, because of this, even when passing a string to a function as a parameter by reference, the function cannot change the referred to string for later use back in the calling code. If you really want to edit a string in place, use class StringBuffer.

9.3.4 The StringBuffer Class

A mutable string class does exist; it is called StringBuffer. It allows you to insert characters into the middle of a string, change characters in a string, append characters to a string, and delete characters from a string, all ***without*** having to create a new string. It is somewhat like a string Vector in that it will grow automatically and silently if you append too many characters onto the existing buffer inside a StringBuffer instance. (Java Vectors are an advanced topic and might not be discussed in this course).

The StringBuffer class is also multithread safe, in that its member functions are 'synchronized'. This makes them a little slower, but safe if multiple threads of a program are trying to modify/access the string at the same time!

You should have a glance at the list of member functions of the StringBuffer and StringTokenizer classes, in the Java online API Specification documentation. While you are there, look at the other String member functions also available (the list above was only a small sample).

9.4 Arrays and Algorithms - Searching and Sorting

As you have seen, in order for computers to do much, they have to have places to store data. They need variables into which to read input data, they need variables to store calculated values, and they need arrays to store stuff that can be scanned, sorted, taken the average of, etc. Computing scientists use the term 'algorithm' to mean a set of instructions (i.e. a recipe), that is often repetitive in nature, which accomplishes some worthwhile task. Two of the most interesting topics are searching and sorting. There are many different ways to do searching. And there are many different ways to do sorting, some more efficient in certain circumstances than others. e.g.

- Which algorithm should you use if the data is somewhat sorted already and you need to sort it fully?
- Is there a better algorithm for searching than the so-called 'linear search algorithm' if the data that you are searching is pre-sorted?

We will study some of these issues later, and you will study them extensively in Cmpt 201 and 307. However, I want to take this opportunity to expose you to some simple versions of these algorithms so you can perhaps even deal with them on the midterm.

```
/********************************************
*File: SearchSort.java
*This class provides static search array and
*sort array algorithms.
*/
public class SearchSort{

  /** main function unit tests the other
  *   function in this class.
  */
  public static void main(String[] args){

    int[] myArray = {10,5,11,3,6};

    int location;

    //Find the index of the value 11 in
    //myArray.
    location = linearSearch(myArray, 11);

    if(location == -1)
      System.out.println("value not found");
    else{
      System.out.println(location);
      System.out.println();
    }

    ExchangeSort(myArray);
    //main() will find that myArray is
    //now sorted even though it was not
    //'returned'.
```

```
    //Print the sorted array.
    for(int i = 0 ; i < myArray.length; i++){
      System.out.println(myArray[i] + " ");
    }
  }
```

```
/******************************************
*LinearSearch searches for value in array.
*Will return -1 if not found.
*Though I don't really like middle return
*functions, this one is very compact and
*anything else is messier requiring extra
*boolean variables and/or tests.
*/
public static int linearSearch(int[] array,
                         int value){

  for(int index = 0; index < array.length;
                            index++){
    if (array[index] == value)
return index;
  }
return -1;
}

/******************************************
*Sort the array parameter in place
*using the selection sort algorithm.
*/
public static void ExchangeSort(
                         int[] array){

  //Index of smallest element found   so far
 //in inner loop:
  int indexOfSmallest;

  int temp;  // used for swap only.
```

```
  for(int i = 0; i < array.length-1; i++){
    //Search 'the rest' of the array
    //assuming 'the rest' starting at 0,
    //then 1, up to one short of end of the
    //array (so on last iteration there are
    //only two left to compare).

    //Assume (without reason) that the first
    //in the rest [i..last] is the smallest,
    //until we encounter a reason to believe
    //another in the rest is smaller.

    indexOfSmallest = i;

    //Inner Loop:
    //Starting one past the beginning of
    //'the rest' of the array, start looking
    //for an element that is smaller.

    for(int j=i+1; j < array.length; j++){

      if(array[j] < array[indexOfSmallest])
        indexOfSmallest = j;
        //makes note of where you found a
        //smaller one, and perhaps later an
        //even smaller one.
    } //end inner for loop.

    //Now we know where the smallest in
    //range [i..last] is.  Swap the smallest
    //element of the rest found by the inner
    //loop, with the first of the rest
    //(which was not likely the smallest).
```

```
      temp = array[indexOfSmallest];

      array[indexOfSmallest] = array[i];

      array[i]= temp;

    } //end outer for.
  } //end function selectionSort.
} //end class SearchSort.

//Note: There are slightly more efficient
//versions of exchange sort (can you
//see ways to make the above more
//efficient?).   In fact, there are much
//better algorithms that exchange sort.
```

Note: The previous code uses ‘special’ javadoc style comments before each function. When a program called javadoc is run on this .java file, beautiful documentation for the .java file is produced in .html format that your instructor will show you.