BACHOLER OF COMPUTER APPLICATIONS

SEMESTER 4

DCA2202

JAVA PROGRAMMING

Unit 4

Arrays and Strings

Table of Contents

1. INTRODUCTION

In the last unit, we discussed operators and control flow statements. In this unit, we will introduce the concept of arrays and strings. The sorting of data can be done using arrays, which represent variables that occupy contiguous spaces in the memory. Each element in the array is distinguished by its index. All the elements in an array must be of the same data type.

1.1 Objectives

After studying this unit, you should be able to:

- *Discuss array representation in Java*
- *Explain various string operations in Java*

2. STRING HANDLING

*An **array** is a contiguous memory location of a similar datatype*. Each element in an array has an index that helps us identify it uniquely. Similar datatype in the definition of the array means all elements in an array have the same datatype, we can't have an array where one element is an integer and the other is of char datatype. All the elements are referenced by a common name. In Java, we first declare an array and then we initialize it.

The syntax for declaring an array:

data- type [ ] variablename;

Example to declare an array:

int [ ] exampleNumbers;

The example shows how to declare a variable that will hold an array of integer type (int) variables. After you declare the variable for array, you have to allocate it in memory. That is achieved by using operator 'new' as mentioned below:

exampleNumbers = new int [10];

The above statement will assign 10 continuous memory spaces to the variable exampleNumbers to store integer type (int) values. Now it can store ten numbers. We use iteration to store values in an array.

Like other programming languages, Java treats ***string*** as a sequence of characters, but there is a difference, here string is an object of ***String*** type unlike other languages where it is declared as array of characters. Treating string as a predefined object, Java provides us a lot of options while operating on strings. There are many built-in methods provided in Java for string. For example, Java has a method to compare two strings, concatenate two strings, etc.

When String objects are created then the characters of the strings cannot be changed. Therefore, ***String*** objects are referred to as immutable. However, this does not act as a restriction, operations create new objects with the required modifications. This can be set to the same variables. String variables are not immutable. There are a few classes that let you create string objects that are mutable. For example, **StringBuffer** whose objects contain string which can be modified.

Both the **String** and **StringBuffer** classes are defined in **java.lang** package. Thus, they are available to all programs automatically. Both are declared **final**, which means that neither of these classes may be sub-classed. This allows certain optimizations that increase performance to take place on common string operations.

The **String** class supports several constructors. To create an empty **String**, you call the default constructor. For example,

```
String s = new String();
```

will create an instance of **String** with no characters in it.

Often you have to create strings that need to be initialized with some value. There are many constructor available in **String** class to handle this. To create a String initialized by an array of characters, use the constructor shown here:

String(char *chars*[ ]);

Here is an example:

```
char s[ ] = { 'a', 'b', 'c', ‘d’ };
String s1 = new String(s);
```

This constructor initializes **s1** with the string "abcd".

You can specify a sub-range of a character array as an initializer using the following constructor:

String(char s[ ], int startIndex, int numChars);

Here, *startIndex* specifies the index at which the sub-range begins, and *numChars* specifies the number of characters to use. Here is an example:

```
char s[ ] = { 'x', 'b', 'c', 'd', 'e', 'f',’g’ };
String s1 = new String(s, 2, 3);
```

This initializes s1 with the string “cde”. Please remember that the array index starts with 0.

We can also construct a String object that contains the same character sequence as another String object using the constructor:

String(String strObj);

Here, *strObj* is a String object. Consider this example:

```
// Construct one String from another.
class StringConstruct {
	public static void main(String args[]) {
		char c[] = { 'S', 'i', 'k', 'k', 'i', 'm' };
		String str1 = new String(c);
		String str2 = new String(str1);
		System.out.println(str1);
		System.out.println(str2);

	}
}
```

Output:

```
Sikkim
Sikkim
```

As you can see, str1 and str2 contain the same string.

Java's char type uses 16 bits to represent the Unicode character set but the format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. As 8-bit ASCII strings are common, the String class provides constructors which can initialize a string when given a byte array. Their forms are shown here:

String(byte asciiChars[ ]);

String(byte asciiChars[ ], int startIndex, int numChars);

The *asciiChars* denoted the array of bytes. The second form lets you specify a sub-range. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform.

The following program illustrates these constructors:

```
// Construct string from subset of char array.
class SubStringCons{
	public static void main(String args[]) {
		byte asc[] = { 65, 66, 67, 68, 69, 70 };
		String str1 = new String(asc);
		System.out.println(str1);
		String str2 = new String(asc, 2, 3);
		System.out.println(str2);
	}
}
```

Output:

```
ABCDEF
CDE
```

The number of characters in the strings is called the length of the string. To get the length of the string you can call the length( ) method, as shown here:

int length( );

The following fragment prints "4", since there are four characters in the string s:

```
char s1[ ] = { 'j', 'a', 'v', 'a' };
String s = new String(s);
System.out.println(s.length());
```

3. SPECIAL STRING OPERATIONS

Strings are an important as well as a common part of programming, thus in Java there are a wide set of methods to support various operations on strings. It includes the creation of new **String** instances, concatenation of two or more strings, other data type conversion into string type, and many more. Explicit methods are available to perform such operations, however, Java does these automatically also to add convenience to programming.

String Literals

Earlier you have seen how you could explicitly create a **String** instance from a character array with help of *new operator*. The same operation could be achieved using string literal. Java automatically creates 'string' object for each literal. Hence string literals can be used to initialize a 'string' object. For example:

```
char charsarry[ ] = { 'r', 'a', 'h' , 'u' , 'l'};
String sthruchar = new String(charsarry);
String sthruliteral = "rahul"; // use string literal
```

A method can also be directly called to a quoted string. For example, we call the method.length() directly to the string "rahul" as shown in the example below and the output will be 5.

```
System.out.println("rahul".length());
```

String Concatenation

Java doesn't allow operators to be directly applied to string objects however in the case of concatenation there is an exception. We can use + operator to directly concatenate string objects as shown in the below example.

```
String age = "19";
String s = "Rahul is” + age + “years old.";
System.out.println(s);
```

This displays the string "Rahul is 9 years old."

One practical use of using + for string concatenation is while we are using it to concatenate a long string. It makes the complete task easier and convenient. Here is an example:

```
// Using concatenation to prevent long lines.
class ConCat {
	public static void main(String args[]) {
		String Str = "Manipal Univeristy " + "is a premier institute " + "in 
the field of education ";
		System.out.println(Str);
	}
}
```

String Concatenation with Other Data Types

Not only string is concatenated with similar data types but it also can be concatenated with other data types as well. As shown in the example:

```
int age = 19;
String str = "Akash is " + age + "years old.";
System.out.println(str);
```

Here age is an integer type and still it is used with string and the output will be:

Akash is 19 years old

The compiler converts any operand to string type when + is used. Let's see another example:

```
String s1 = "five: " + 3 + 2;
System.out.println(s1);
```

The output displayed is:

five: 32

```
String s2 = "five: " + (3 + 2);
System.out.println(s1);
```

To complete the integer addition first, you must use parentheses, like this:

`String s2 = "five: " + (3 + 2);`

Now, s2 contains the string "five: 5".

String Conversion and toString( )

Method tostring() defined by String is called to convert data into string representation during string conversion. *tostring*( ) is overloaded for type Object and for all the simple types. In the case of simple type *tostring* ( ) returns a string that contains the human-readable equivalent of the value with which it is called. In case of objects, *tostring* ( ) calls the method toString( ) on the object. Here toString( ) method is the means by which you can determine the string representation for objects of classes that you create. As it is defined by Object, every class implements toString(). Default implementation of toString( ) is rarely sufficient. In most important classes that we create, we may want to override toString( ) to provide our own string representations. The toString( ) method has this general form:

String toString( )

To implement toString( ), simply return a String object that contains the human-readable string that appropriately describes an object of your class.

Overriding toString( ) for classes that you create allows the resulting strings to be completely integrated into Java's programming environment. For example, they can be used in println( ) and print( ) statements and in concatenation expressions. The following program demonstrates this by overriding toString( ) for the Box class:

```
// Override toString() for Box class.
class Box {
	double wd;
	double ht;
	double dh;

	Box(double w1, double h1, double d1) {
		wd = w1;
		ht = h1;
		dh = d1;
	}

	public String toString() {
		return "Dimensions are " + wd + " by " + dh + " by " + ht + ".";
	}
}

class Hello {
	public static void main(String args[]) {
		Box b1 = new Box(5, 6, 7);
		String s1 = "Box b1: " + b1;

	}
}
```

Output:

```
Dimensions are 5 by 7 by 6.
Box b1: Dimensions are 5 by 7 by 6.
```

As you can see, Box's toString( ) method is automatically invoked when a Box object is used in a concatenation expression or in a call to println( ).

Strings in Switch Statements

With the JDK 7 release, String objects can be directly used in the expression of Switch statements. The below example demonstrates the use of the String object in the expression of Switch:

Example

```
public String DayNumberWithSwitchStatement(String WeekDayArg) {
             String DayType;
             switch (WeekDayArg) {
             case "Saturday":
             case "Sunday":
                    DayType = "Weekend";
                    break;
             case "Monday":
                    DayType = "Start of work week";
                    break;
             case "Tuesday":
             case "Wednesday":
             case "Thursday":
                    DayType = "Midweek";
                    break;
             case "Friday":
                    DayType = "End of work week";
                    break;
             default:
                    throw new IllegalArgumentException(
                            "Invalid day of the week: " + WeekDayArg);
             }
             return DayType;
      }
```

The switch statement uses the String object in the expression of Switch statement for comparison with the expression in each case of the **'switch case'** statement. The comparison is similar to String.equals() statement. Thus the comparisons of strings here are case sensitive. The bytecode generated by the Java compiler from switch statements that uses String objects is more efficient compared to the bytecode generated from if-then-else statements.

Self-assessment questions - 1

1. _________ represents a number of variables which occupy contiguous spaces in the memory.
2. 2.Each element in the array is distinguished by its _______ .
3. 3.In Java, a ________ is a sequence of characters.

4. CHARACTER EXTRACTION

The String class provides a different way to extract characters from a String object. The characters in a string within a String object cannot be indexed like a character array but many of the String methods employ an index (or offset) into the string for their operation.

charAt( )

To extract a single character from a String, you can refer directly to an individual character through charAt( ) method. It has this general form:

char charAt(int where);

Here, *where* is the index of the character that you want to obtain. The value of *where* must be positive and specify a location within the string. **charAt( )** returns the character at the mentioned location.

For example,

```
char ch;
ch = "abc".charAt(1);
```

assigns the value "b" to ch.

getChars( )

If you need to extract more than one character at a time, you can use the getChars( ) method.

It has this general form:

void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied is passed in *targetStart.* Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

The following program demonstrates **getChars( ):**

```
class getCharsDemo {
        public static void main(String args[]) {
                String s = "This is a demo of the getChars method.";
                int start = 10;
                int end = 14;
                char buf[] = new char[end - start];
                s.getChars(start, end, buf, 0);
                System.out.println(buf);
        }
}
```

Here is the output of this program:

```
demo
```

getBytes( )

There is an alternative to getChars( ) that stores the characters in an array of bytes. This method is called getBytes( ), and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

byte[ ] getBytes( );

Other forms of getBytes( ) are also available. getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray( )

If you want to convert all the characters in a String object into a character array, the easiest way is to call toCharArray( ). It returns an array of characters for the entire string.

It has this general form:

char[ ] toCharArray( );

This function is provided as a convenience since it is possible to use getChars( ) to achieve the same result.

5. STRING COMPARISON

The String class includes several methods that compare strings or substrings within strings. Each is examined here.

<u>equals( ) and equalsIgnoreCase( )</u>

To compare two strings for equality, use equals( ). It has this general form:

boolean equals(Object str);

Here, *str* is the String object being compared with the invoking String object. It returns true if the strings contain the same characters in the same order, and false otherwise.

The comparison is case-sensitive. To perform a comparison that ignores case differences, call equalsIgnoreCase( ). When it compares two strings, it considers A-Z to be the same as a-z. It has this general form:

boolean equalsIgnoreCase(String str);

Here, *str* is the String object being compared with the invoking String object. It, too, returns true if the strings contain the same characters in the same order, and false otherwise.

Here is an example that demonstrates equals( ) and equalsIgnoreCase( ):

```java
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
      public static void main(String args[]) {
            String s1 = "Hello";
            String s2 = "Hello";
            String s3 = "Good-bye";
            String s4 = "HELLO";
            System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
            System.out.println(s1 + " equals " + s3 + " -> " + s1.equals(s3));
            System.out.println(s1 + " equals " + s4 + " -> " + s1.equals(s4));
            System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> "
                   + s1.equalsIgnoreCase(s4));
      }
}
```

The output from the program is shown here:

```
Hello equals Hello -> true

Hello equals Good-bye -> false

Hello equals HELLO -> false

Hello equalsIgnoreCase HELLO -> true
```

regionMatches( )

The regionMatches( ) method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:

boolean regionMatches(int startIndex, String str2, int str2StartIndex, int numChars);

boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars);

For both versions, *startIndex* specifies the index at which the region begins within the invoking String object. The String being compared is specified by *str2*. The index at which the comparison will start within *str2* is specified by *str2StartIndex*. The length of the substring being compared is passed in *numChars*. In the second version, if *ignoreCase* is true, the case of the characters is ignored. Otherwise, the case is significant.

startsWith( ) and endsWith( )

String defines two routines that are, more or less, specialized forms of regionMatches( ). The startsWith( ) method determines whether a given String begins with a specified string. Conversely, endsWith( ) determines whether the String in question ends with a specified string. They have the following general forms:

boolean startsWith(String str);

boolean endsWith(String str);

Here, *str* is the String being tested. If the string matches, true is returned. Otherwise, false is returned. For example,

```
"Foobar".endsWith("bar");
"Foobar".startsWith("Foo");
```

are both true.

A second form of startsWith( ), shown here, lets you specify the starting point:

boolean startsWith(String str, int startIndex);

Here, *startIndex* specifies the index into the invoking string at which point the search will begin. For example,

```
"Foobar".startsWith("bar", 3);
```

returns true.

equals( ) Versus ==

It is important to understand that the equals( ) method and the == operator perform two different operations. As just explained, the equals( ) method compares the characters inside a String object. The == operator compares two object references to see whether they refer to the same instance. The following program shows how two different String objects can contain the same characters, but references to these objects will not compare as equal:

```
// equals() vs ==
class equalityCheck {
      public static void main(String args[]) {
            String str1 = "Hello";
            String str2 = new String(str1);
            System.out.println(str1 + " equals " + str2 + " ->" +
str1.equals(str2));
            System.out.println(str1 + " == " + str2 + " -> " + (str1 == str2));
      }
}
```

The variable str1 refers to the String instance created by "Hello". The object referred to by str2 is created with str1 as an initializer. Thus, the contents of the two String objects are identical, but they are distinct objects. This means that str1 and str2 do not refer to the same objects and are, therefore, not ==, as is shown here by the output of the preceding example:

Hello equals Hello -> true

Hello == Hello -> false

compareTo( )

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which *is less than, equal to, or greater than* the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The String method compareTo( ) serves this purpose. It has this general form:

int compareTo(String str);

Here, *str* is the String being compared with the invoking String. The result of the comparison is returned and is interpreted as shown here:

Value	**Meaning**
Less than zero	The invoking string is less than str.
Greater than zero	The invoking string is greater than str.
Zero	The two strings are equal.

Here is a sample program that sorts an array of strings. The program uses compareTo() to determine sort ordering for a bubble sort:

```
// A bubble sort for strings.
class Hello {
      static String arr[] = { "Now", "is", "the", "time", "for", "all", "good",
               "men", "to", "come", "to", "the", "aid", "of", "their", "country" };

      public static void main(String args[]) {
            for (int i = 0; i < arr.length - 1; i++) {
                  for (int j = i+1; j < arr.length; j++) {
                        if (arr[i].compareTo(arr[j]) > 0) {
                              String t = arr[i];
                              arr[i] = arr[j];
                              arr[j] = t;
                        }
                  }
                  System.out.println(arr[i]);
            }
      }
}
```

Output:

```
Now
aid
all
come
country
for
good
is
men
of
the
their
time
to
```

As you can see from the output of this example, compareTo() takes into account uppercase and lowercase letters. The word "Now" comes out before all the others because it begins with an uppercase letter, which means it has a lower value in the ASCII character set.

If you want to ignore case differences when comparing two strings, use compareToIgnoreCase(), shown here:

int compareToIgnoreCase(String str)

This method returns the same results as compareTo(), except that case differences are ignored.

Self-Assessment Questions - 2

4. To extract a single character from a String, you can use _______ method.
5. To extract more than one character at a time, you can use_______ method.
6. To compare two strings for equality, use_______ method.

6. SEARCHING STRINGS

The String class provides two methods that allow you to search a string for a specified character or substring:

- indexOf( ) Searches for the first occurrence of a character or substring.
- lastIndexOf( ) Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or –1 on failure.

To search for the first occurrence of a character, use

int indexOf(int ch);

To search for the last occurrence of a character, use

int lastIndexOf(int ch);

Here, *ch* is the character being sought.
To search for the first or last occurrence of a substring, use

int indexOf(String str);

int lastIndexOf(String str);

Here, *str* specifies the substring.
You can specify a starting point for the search using these forms:

int indexOf(int ch, int startIndex);

int lastIndexOf(int ch, int startIndex); int

int indexOf(String str, int startIndex);

int lastIndexOf(String str, int startIndex);

Here, *startIndex* specifies the index at which point the search begins. For indexOf( ), the search runs from *startIndex* to the end of the string. For lastIndexOf( ), the search runs from *startIndex* to zero.

The following example shows how to use the various index methods to search inside of Strings:

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
	public static void main(String args[]) {
		String s = "Now is the time for all good men "
			+ "to come to the aid of their country.";
		System.out.println(s);
		System.out.println("indexOf(t) = " + s.indexOf('t'));
		System.out.println("lastIndexOf(t) = " + s.lastIndexOf('t'));
		System.out.println("indexOf(the) = " + s.indexOf("the"));
		System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));
		System.out.println("indexOf(t, 10) = " + s.indexOf('t', 10));
		System.out.println("lastIndexOf(t, 60) = " + s.lastIndexOf('t', 60));
		System.out.println("indexOf(the, 10) = " + s.indexOf("the", 10));
		System.out
			.println("lastIndexOf(the, 60) = " + s.lastIndexOf("the",
60));
	}
}
```

Here is the output of this program:

```
Now is the time for all good men to come to the aid of their country. indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

7. STRING MODIFICATION

As string object are immutable we have to copy a string into **StringBuffer** whenever we need to modify it. We can also use some predefined string methods; it will construct a modified copy of the string and returns the same. Methods which we can use for the purpose are stated below:

substring( )

This method is used to extract a substring from the given string. There are two types of substring. The first one is:

String substring(int startIndex);

Where the startindex defines the starting point of the substring that is to be extracted from the given string.

The second type is:

String substring(int startIndex, int endIndex);

Here we mention both the starting index and the end index.

Example to demonstrate the use of substring()

```
// Substring replacement.
class SubStringExample {
      public static void main(String args[]) {
            String orignal = "This is Manipal University of Jaipur, Directorate
of Distance Education";
            String search = "is";
            String sub = "at";
            String result = "";
            int i;

            do { // replace all matching substrings
                  i = orignal.indexOf(search);
                  if (i != -1) {
                        result = orignal.substring(0, i);
                        result = result + sub;
                        result = result + orignal.substring(i +
search.length());
                        orignal = result;
                        System.out.println(result);
                  }

            } while (i != -1);

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

Output:

This is Manipal University of Jaipur, Directorate of Distance Education That is Manipal University of Jaipur, Directorate of Distance Education That at Manipal University of Jaipur, Directorate of Distance Education That at Manipal University of Jaipur, Directorate of Dattance Education

concat( )

To concatenate two strings we use the method concat() as shown below:

String concat(String str);

This method creates a new object that contains the invoking string with the contents of *str* appended to the end. concat( ) performs the same function as +. For example,

```
String str1 = "Sikkim";
String str2 = str1.concat(" Manipal");
```

It works similar to +. In the above example in str2, Sikkim Manipal will be stored. Sikkim comes from str1 and concat() method adds " Manipal" to it.

replace( )

The replace( ) method replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char original, char replacement);

Here, the original specifies the character to be replaced by the character specified by *replacement*. The resulting string is returned. For example,

```
String s = "Hello".replace('l', 'w');
```

puts the string "Hewwo" into s.

trim( )

The trim( ) method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

String trim( );

Here is an example:

```
String s = " Hello World ".trim();
```

This puts the string "Hello World" into **s.**

The trim( ) method is quite useful when you process user commands. For example, the following program prompts the user for the name of a state and then displays that state's capital. It uses trim( ) to remove any leading or trailing whitespace that may have inadvertently been entered by the user.

```
import java.io.*;

// Using trim() to process commands.
class UseTrim{
      public static void main(String args[]) throws IOException {
            BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
            String str;
            System.out.println("Enter 'stop' to quit.");
            System.out.println("Enter State: ");
            do {
                  str = br.readLine();
                  str = str.trim(); // remove whitespace
                  if (str.equals("Illinois"))
                        System.out.println("Capital is Springfield.");
                  else if (str.equals("Missouri"))
                        System.out.println("Capital is Jefferson City.");
                  else if (str.equals("California"))
                        System.out.println("Capital is Sacramento.");
                  else if (str.equals("Washington"))
                        System.out.println("Capital is Olympia.");
                  // ...
            } while (!str.equals("stop"));
      }
}
```

Data Conversion Using valueOf()

The valueOf( ) method converts data from its internal format into a human- readable form. It is a static method that is overloaded within String for all of Java's built-in types, so that each type can be converted properly into a string. valueOf( ) is also overloaded for type Object, so an object of any class type you create can also be used as an argument. (Recall that Object is a superclass for all classes.) Here are a few of its forms:

static String valueOf(double num);

static String valueOf(long num);

static String valueOf(Object ob);

static String valueOf(char chars[ ]);

As we discussed earlier, valueOf( ) is called when a string representation of some other type of data is needed – for example, during concatenation operations. You can call this method directly with any data type and get a reasonable String representation. All the simple types are converted to their common String representation. Any object that you pass to valueOf( ) will return the result of a call to the object's toString( ) method. In fact, you could just call toString( ) directly and get the same result. For most arrays, valueOf( ) returns a rather cryptic string, which indicates that it is an array of some type. For arrays of char, however, a String object is created that contains the characters in the char array. There is a special version of valueOf( ) that allows you to specify a subset of a char array. It has this general form:

static String valueOf(char chars[ ], int startIndex, int numChars);

Here, *chars* is the array that holds the characters*, startIndex* is the index into the array of characters at which the desired substring begins, and *numChars* specifies the length of the substring.

Changing the Case of Characters within a String

The method toLowerCase( ) converts all the characters in a string from uppercase to lowercase. The toUpperCase( ) method converts all the characters in a string from lowercase to uppercase. Non-alphabetical characters, such as digits, are unaffected. Here are the general forms of these methods:

String toLowerCase( );

String toUpperCase( );

Both methods return a String object that contains the uppercase or lowercase equivalent to the invoking String.

Here is an example that uses toLowerCase( ) and toUpperCase( ):

```
// Demonstrate toUpperCase() and toLowerCase().

class ChangeCase {

	public static void main(String args[]) {
		String s = "This is a test.";
		System.out.println("Original: " + s);
		String upper = s.toUpperCase();
		String lower = s.toLowerCase();
		System.out.println("Uppercase: " + upper);
		System.out.println("Lowercase: " + lower);
	}
}
```

Output:

```
Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.
```

Self-Assessment Questions - 3

7. _________method searches for the first occurrence of a character or substring.
8. __________method searches for the last occurrence of a character or substring.
9. You can extract a substring using _______ method.

8. STRINGBUFFER

StringBuffer is a peer class of String that provides much of the functionality of strings. As you know, String represents fixed-length, immutable character sequences. In contrast, StringBuffer represents growable and writeable character sequences.

StringBuffer may have characters and substrings inserted in the middle or appended to the end. StringBuffer will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth. Java uses both classes heavily, but many programmers deal only with String and let Java manipulate StringBuffers behind the scenes by using the overloaded + operator.

StringBuffer defines these three constructors:

StringBuffer( );

StringBuffer(int size);

StringBuffer(String str);

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation. StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, StringBuffer reduces the number of reallocations that take place.

length( ) and capacity( )

The current length of a StringBuffer can be found via the length( ) method, while the total allocated capacity can be found through the capacity( ) method. They have the following general forms:

int length( );

int capacity( );

The following program demonstrates the use of length() and capacity().

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
	public static void main(String args[]) {
		StringBuffer sb = new StringBuffer("Hello");
		System.out.println("buffer = " + sb);
		System.out.println("length = " + sb.length());
		System.out.println("capacity = " + sb.capacity());
	}
}
```

Here is the output of this program, which shows how StringBuffer reserves extra space for additional manipulations.

Output:

```
buffer = Hello
length = 5
capacity = 21
```

Since sb is initialized with the string "Hello" when it is created, its length is 5. Its capacity is 21 because the room for 16 additional characters is automatically added.

ensureCapacity( )

If you want to pre-allocate room for a certain number of characters after a StringBuffer has been constructed, you can use ensureCapacity( ) to set the size of the buffer. This is useful if you know in advance that you will be appending a large number of small strings to a StringBuffer. ensureCapacity( ) has a general form:

void ensureCapacity(int capacity);

Here, *capacity* specifies the size of the buffer.

setLength( )

To set the length of the buffer within a StringBuffer object, use setLength(). Its general form is shown here:

void setLength(int len);

Here, len specifies the length of the buffer. This value must be nonnegative.

When you increase the size of the buffer, null characters are added to the end of the existing buffer. If you call setLength( ) with a value less than the current value returned by length( ), then the characters stored beyond the new length will be lost. The setCharAtDemo sample program in the following section uses setLength( ) to shorten a StringBuffer.

charAt( ) and setCharAt( )

The value of a single character can be obtained from a StringBuffer via the charAt( ) method. You can set the value of a character within a StringBuffer using setCharAt( ).

Their general forms are shown here:

char charAt(int where);

void setCharAt(int where, char ch);

For charAt( ), *'where'* specifies the index of the character being obtained. For setCharAt(), *'where'* specifies the index of the character being set, and ch specifies the new value of that character. For both methods, *'where'* must be non-negative and must not specify a location beyond the end of the buffer.

The following example demonstrates charAt( ) and setCharAt( ):

```
// Demonstrate charAt() and setCharAt().
class setCharAtDemo {
	public static void main(String args[]) {
		StringBuffer sb = new StringBuffer("Hello");
		System.out.println("buffer before = " + sb);
		System.out.println("charAt(1) before = " + sb.charAt(1));
		sb.setCharAt(1, 'i');
		sb.setLength(2);
		System.out.println("buffer after = " + sb);
		System.out.println("charAt(1) after = " + sb.charAt(1));
	}
}
```

Output:

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

getChars( )

To copy a substring of a StringBuffer into an array, use the getChars( ) method. It has this general form:

void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart);

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. This means that the substring contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied is passed in *targetStart*. Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

append( )

The append( ) method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object. It has overloaded versions for all the built-in types and for Object. Here are a few of its forms:

StringBuffer append(String str);

StringBuffer append(int num);

StringBuffer append(Object obj);

String.valueOf() is called for each parameter to obtain its string representation. The result is appended to the current StringBuffer object. The buffer itself is returned by each version of append( ). This allows subsequent calls to be chained together, as shown in the following example:

```
// Demonstrate append().
class appendDemo {
	public static void main(String args[]) {
		String s;
		int a = 42;
		StringBuffer sb = new StringBuffer(40);
		s = sb.append("a = ").append(a).append("!").toString();
		System.out.println(s);
	}
}
```

Output:

a = 42!

The append( ) method is most often called when the + operator is used on String objects. Java automatically changes modifications to a String instance into similar operations on a StringBuffer instance. Thus, a concatenation invokes append( ) on a StringBuffer object. After the concatenation has been performed, the compiler inserts a call to toString( ) to turn the modifiable StringBuffer back into a constant String. All of this may seem unreasonably complicated. Why not just have one string class and have it behave more or less like StringBuffer? The answer is performance. There are many optimizations that the Java run time can make knowing that String objects are immutable. Thankfully, Java hides most of the complexity of conversion between Strings and StringBuffers. Actually, many programmers will never feel the need to use StringBuffer directly and will be able to express most operations in terms of the + operator on String variables.

<u>insert</u>( )

The insert( ) method inserts one string into another. It is overloaded to accept values of all the simple types, plus Strings and Objects. Like append( ), it calls String.valueOf( ) to obtain the string representation of the value it is called with. This string is then inserted into the invoking StringBuffer object. These are a few of its forms:

StringBuffer insert(int index, String str);

StringBuffer insert(int index, char ch);

StringBuffer insert(int index, Object obj);

Here, *index* specifies the index at which point the string will be inserted into the invoking StringBuffer object.

The following sample program inserts "like" between "I" and "Java":

```
// Demonstrate insert().
class insertDemo {
      public static void main(String args[]) {
            StringBuffer sb = new StringBuffer("I Java!");
            sb.insert(2, "like ");
            System.out.println(sb);
      }
}
```

Output:

I like Java!

reverse( )

You can reverse the characters within a StringBuffer object using reverse(), as shown here:

StringBuffer reverse( );

This method returns the reversed object on which it was called. The following program demonstrates reverse( ):

```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
        public static void main(String args[]) {
                StringBuffer s = new StringBuffer("abcdef");
                System.out.println(s);
                s.reverse();

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

Output:

```
abcdef
fedcba
```

delete( ) and deleteCharAt( )

Java adds to StringBuffer the ability to delete characters using the methods delete( ) and deleteCharAt( ). These methods are shown here:

StringBuffer delete(int startIndex, int endIndex);

StringBuffer deleteCharAt(int loc);

The delete( ) method deletes a sequence of characters from the invoking object. Here, *startIndex* specifies the index of the first character to remove, and *endIndex* specifies an index one past the last character to remove. Thus, the substring deleted runs from *startIndex* to *endIndex*–1. The resulting StringBuffer object is returned. The deleteCharAt( ) method deletes the character at the index specified by loc. It returns the resulting StringBuffer object.

Here is a program that demonstrates the delete( ) and deleteCharAt( ) methods:

```
// Demonstrate delete() and deleteCharAt()
class deleteDemo {
	public static void main(String args[]) {
		StringBuffer sb = new StringBuffer("This is a test.");
		sb.delete(4, 7);
		System.out.println("After delete: " + sb);
		sb.deleteCharAt(0);
		System.out.println("After deleteCharAt: " + sb);
	}
}
```

Output:

After delete: This a test.

After deleteCharAt: his a test.

replace( )

Another new method added to StringBuffer by Java is replace( ). It replaces one set of characters with another set inside a StringBuffer object. Its signature is shown here:

StringBuffer replace(int startIndex, int endIndex, String str);

The substring being replaced is specified by the indexes *startIndex* and *endIndex*. Thus, the substring at *startIndex* through *endIndex*–1 is replaced. The replacement string is passed in *str*. The resulting StringBuffer object is returned.

The following program demonstrates replace( ):

```
// Demonstrate replace()
class replaceDemo {
	public static void main(String args[]) {
		StringBuffer sb = new StringBuffer("This is a test.");
		sb.replace(5, 7, "was");
		System.out.println("After replace: " + sb);
	}
}
```

Output:

After replace: This was a test.

substring( )

Java also adds the substring( ) method, which returns a portion of a StringBuffer. It has the following two forms:

String substring(int startIndex);

String substring(int startIndex, int endIndex);

The first form returns the substring that starts at *startIndex* and runs to the end of the invoking StringBuffer object. The second form returns the substring that starts at *startIndex* and runs through *endIndex*–1. These methods work just like those defined for String that was described earlier.

Self-Assessment Questions - 4

10. To set the length of the buffer within a StringBuffer object, use ________ method.
11. The __________ method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object.
12. You can reverse the characters within a StringBuffer object using ________method.

9. ONE DIMENSIONAL ARRAYS

As mentioned in the beginning, An array is a contiguous memory location of a similar datatype. Each element in an array has a unique index. Every element in the array can be accessed by using the name of the array along with the index. The general syntax for declaring an array is

type arr_name[];

The type determines the data type of all the elements in the array.

Example :

int numbers_selected[];

A point to note is that - this is only a declaration of the array and the actual memory will not be allocated by Java at this point of execution.

To allocate the required memory, we must use the keyword 'new'

Also, mention the size required.

Typically, the declaration and allocation is performed in a single step using the below syntax:

type arr_name[] = new type[size];

Example:

int numbers_selected = new int[10];

Initializing an array

If the values that the array needs to store are known, we can use the same to allocate memory and initialize the array

For example:

int numbers_selected[] = { 10, 20, 30, 40, 50 } ;

In this case, an explicit allocation of memory is not required. Java will allocate sufficient memory locations to store the values and initializes the same with the values provided.

The list used in initialization is a set of comma-separated values of the same type enclosed within curly braces. If the values are of different types an error would be thrown during compilation. Arrays can take values of primitive type or of objects of the same class.

To access the element in the array we use the offset values within the square brackets along with the array name. Java provides a strict check on the length of the array and throws an

“ArrayIndexOutOfBoundsException’ if we try to access the values beyond the allocated length.

A loop can also be used to access and modify an array.

Examples:

```
class Hello {
	public static void main (String arg[]) {
		int[] numbersSelected = {10, 20, 30, 40, 50 } ;
		int largest = 0;

		for (int i=0;i<numbersSelected.length;i++) {
			if (largest < numbersSelected[i]) {
				largest = numbersSelected[i];
			}
		}
		System.out.println(largest);
	}
}
```

10. MULTI-DIMENSIONAL ARRAYS

So far, we have seen one-dimensional arrays. In Java, it is possible to create multidimensional arrays. This is done by using an array of arrays. Let us first look at a two-dimensional array. The concepts can be extended to any n-dimensional array.

Declaring a 2D array:

Similar to the one-dimensional array, the two-dimensional array has to be declared to indicate the type of elements that it will store and the number of dimensions.

Syntax:

type arr_name[][]

this indicates that arr_name is a two-dimensional array of data type 'type'.

for example:

int matrix[][];

The allocation of memory will have to be done explicitly using the new command. We could specify both the number of columns and rows at the same time or we could do them separately. Let us understand this with an example

int matrix[][] = new int[3][4];

This will create a matrix of 3 rows and 4 columns.

We can also do this as :

int matrix[][] = new int[3][];

matrix[0]= new int[4];

matrix[1] = new int[4];

matrix[2] = new int[4];

The two-dimensional array can be accessed by providing the offsets across the dimensions within a square bracket

The syntax to access two-dimensional array is :

arr_name[row_id][column_id];

For example:

matrix[2][3]=0;

This sets the value of the 3rd element in 2nd row to the value 0.

11. SUMMARY

In this unit, we have discussed the following:

- **Storing Data in Arrays**

  An array represents a number of variables that occupy contiguous spaces in the memory. Each element in the array is distinguished by its index. All the elements in an array must be of the same data type.

- **Manipulating Strings**

  in Java a *string* is a sequence of characters. Java implements strings as objects of type String. Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient.

12. TERMINAL QUESTIONS

1. What do you mean by an array? Explain with an example?
2. List out with examples various character extraction functions available in Java.
3. List out various String comparison functions in Java.
4. What are the functions available in Java to modify a String? Explain with examples.
5. List out the functions provided by StringBuffer class in Java.

13. ANSWERS

Self-Assessment Questions

1. array
2. index
3. String
4. charAt()
5. getChars()
6. equals()
7. indexOf()
8. lastIndexOf()
9. substring()
10. setLength()
11. append()
12. reverse()

Terminal Questions

1. An array represents a number of variables which occupy contiguous spaces in the memory. (Refer section 2)
2. charAt(), getChars(), getBytes(). (Refer section 4)
3. equals(), equalsIgnoreCase(), regionMatches(). (Refer section 5)
4. concat(), replace(), trim(). (Refer section 7)
5. length(), capacity(), setLength(), charAt(), etc. (Refer section 8)