An Introduction to Computer Science with Java

Last edited January 6, 2014 by C. Herbert

Chapter 1 Introduction

Chapter 2 Reading, Writing, and Arithmetic

Chapter 3 Programming Logic

Chapter 4 Repetition, Repetition, Repetition

Chapter 5 Methods and Modularity

Chapter 6 Using Arrays

Chapter 7 Graphical User Interfaces

Chapter 9 Object-Oriented Software Concepts

Chapter 9 Classes and Objects in Java

Chapter 10 Event-Driven Software

Chapter 11 Java Graphics

Chapter 12 Java Exceptions

Chapter 13 Software Testing

Chapter 14 Java Software Deployment

Chapter 15 Java Strings

Contents Chapter 15 – Java Strings

Chapter 15 – Java Strings 2
Chapter Learning Outcomes 2
15.1 The String Class 2
15.2 Conversion among Strings, Primitive Data Types, and Wrapper Classes 4
Converting Primitive Data to Strings 4
Java's Wrapper Classes 5
Converting Strings to Primitive Data 5
15.3 Retrieving Characters and SubStrings from within a String 7
charAt() and *substring()* Methods 7
Contains and *IndexOf* methods 9
15.4 Manipulating Strings, SubStrings, and Characters within a String 12
Replacing SubStrings within a String 12
Example Program – Converting MS-DOS Style Path Names to POSIX Style Path Names 13
Other Useful String and Character Manipulation Methods 16
Example: Data Standardization 16
15.5 Comparing Strings and Substrings 17
Example: putting two Strings in order 18
15.6 Immutability and the The StringBuilder and StringBuffer Classes 19
The StringBuffer Class 21

Introduction to Computer Science with Java

Chapter 15 – Java Strings

This chapter discusses the storage and processing of character string data in Java. It includes a look at Java's String class and conversion between Strings and primitive data types using features of the String class and Java's wrapper classes. Methods from the String and Character class for processing character strings and for retrieving and manipulating substrings of a character string are discussed. The chapter also explains how the StringBuilder and StringBuffer classes relate to the String class.

Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

- describe the nature of a character string and String class objects, and list and describe some of the more commonly used methods in the String class
- describe what a wrapper class is, list Java's wrapper classes, and describe how methods from the String class and from wrapper classes can be used to convert between String data and primitive data.
- describe how to retrieve characters and substrings from within a string.
- describe how to manipulate String data.
- create Java code that converts data between Strings and primitive data types, that manipulates Strings, and that finds and manipulates substrings within a string.
- describe how to compare two Strings and how to test Strings and substrings for specified characteristics. Write code to do so.
- describe the concept of an immutable object and how this relates to the String class, the StringBuilder class, and the StringBuffer class.

15.1 The String Class

A **character string** in the Java programming language is an instance of the *String* class. A String is an array of character data. According to the Java 7 documentation,

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

`String(data)` in the code above is a constructor that instantiates a String from a *char* array. There are more than a dozen different constructors in the String class.

The Java 7 implementation of the String class bases strings on the default character set of the native system. Most modern computers use UTF-16, the 16-bit version of Unicode. (See the Unicode home page, on the Web at: http://www.unicode.org/standard/WhatIsUnicode.html) This means the characters in String objects on most systems are encoded as UTF-16 characters. This includes non-printing and control characters, such as the tab (Unicode /u0009), the form feed (/u000C), and the carriage return (/u000E).

Simplified access to the String class, unlike most other classes, is built into the Java language, so String variables and constants are declared and used in a manner very similar to primitive data types:

```
String name;
String message = "Hello World!";
final String EOP = "— end of page —" ;
```

All character string literals in Java, such as "*Go Eagles!*" in the statement

```
System.out.println("Go Eagles!");
```

are implemented as instances of the String class.

We have been using Strings since the beginning of this course, but we really have only used them as placeholders to store text data. In this chapter we will learn about more about the methods of the String class and some techniques for processing String data. The complete definition of the String class in Java 7 is included in the Java 7 documentation, online at:
http://docs.oracle.com/javase/7/docs/api/java/lang/String.html

The String class includes methods for examining individual characters of a String, for comparing strings, for searching strings, for extracting substrings, and for converting a String to all uppercase or all lowercase characters.

Classes of other objects in Java often have a *toString()* method that will create a String representing an instance of the object. The conversion of primitive data types to Strings is built into the Java programming language's type casting, but there are also methods to do this explicitly.

The plus sign in Java *(+)* is defined for String class objects as the **concatenation operator**, which will combine two Strings into a single String by appending the right hand String to the end of the left hand String, as in the following:

```
String fname;    // first name
String lname;    // last name
String fullName;      //
fname = "Joe";
lname = "Smith";
fullName= fname + lname;
System.out.println(fullName);  // prints "JoeSmith";
fullName= fname + " " + lname;
System.out.println(fullName);  // prints "Joe Smith";
```

In addition to the the String class, there are related *StringBuilder* and *StringBuffer* classes, which will be discussed later in this chapter.

15.2 Conversion among Strings, Primitive Data Types, and Wrapper Classes

User I/O and data stored in files is often in the form of text that corresponds to Strings. For this reason, and for other reasons, methods exist to convert primitive data into String objects and String objects into primitive data.

Converting Primitive Data to Strings

The String class has a set of polymorphic *valeuOf()* methods that will convert primitive data values other than those of the *byte* data type into a String object. Each is a static method (class level) that returns a String object. The following table of these methods is based on the Java String class documentation:

Method	Description
String.valueOf(boolean b)	Returns the string representation of the boolean argument.
String.valueOf(char c)	Returns the string representation of the char argument.
String.valueOf(double d)	Returns the string representation of the double argument.
String.valueOf(float f)	Returns the string representation of the float argument.
String.valueOf(int i)	Returns the string representation of the int argument.
String.valueOf(long l)	Returns the string representation of the long argument.
String.valueOf(**Object** obj)	Returns the string representation of the Object argument.
Notes: Notice that there is no String.valueOf() method for the byte primitive data type.	

Of course, Java's type casting feature will also convert primitive data to a String object, such as in the following:

```
double gross;    // gross pay for one employee
. . .  // not all of the code is shown here
System.out.print("Your gross is " + gross);
```

The value of gross is converted to a String and concatenated with the String *"Your gross is "* by the plus sign operator because one of its operands is a String. The resulting String is then sent to the system's standard output device. In cases such as this, there is no need to use a *valeuOf()* method.

While the String class *valueOf()* methods can be used to convert primitive data to Strings, the String class does not have methods to go in the other direction – from a String to a primitive data type. Those methods are in Java's wrapper classes.

Java's wrapper classes have *valueOf* methods that each convert a String value to a value of the wrapper class, and *parse* methods that each convert a String to a value of the primitive data type corresponding to the wrapper class. To understand more about this, we need to discuss the wrapper classes.

Java's Wrapper Classes

A **wrapper class** is a class that wraps a primitive data type as an object. The value of the primitive data type becomes a property of the object. There are wrapper classes for each of the primitive data types. The *Integer* class, for example, corresponds to the *int* data type.

The wrapper classes each have methods that can be used to manipulate data in ways that primitive data types cannot be directly manipulated. Wrapper class objects can also be used when it is necessary to pass primitive data as an object (pass-by-reference) as opposed to passing as primitive data (pass-by-value).

The wrapper classes corresponding to the numeric primitive data types – *byte, short, int, long, float* and *double* – are often called the **number classes**, even though there is no parent class for them named *Number*. This is just a convenient way to refer to them.

Wrapper classes also exist for the remaining two primitive data types: *Character* is a wrapper class for the *char* data type; *Boolean* is a wrapper class for the *boolean* data type.

The following table lists the primitive data types and their corresponding classes.

Primitive Data Type	Wrapper Class	Class documentation on the Web at:
byte	Byte	http://docs.oracle.com/javase/7/docs/api/java/lang/Byte.html
short	Short	http://docs.oracle.com/javase/7/docs/api/java/lang/Short.html
int	Integer	http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html
long	Long	http://docs.oracle.com/javase/7/docs/api/java/lang/Long.html
float	Float	http://docs.oracle.com/javase/7/docs/api/java/lang/Float.html
double	Double	http://docs.oracle.com/javase/7/docs/api/java/lang/Double.html
Boolean	Boolean	http://docs.oracle.com/javase/7/docs/api/java/lang/Boolean.html
char	Character	http://docs.oracle.com/javase/7/docs/api/java/lang/Character.html

Converting Strings to Primitive Data

Each wrapper class has a *valueOf()* method that returns an object of that particular wrapper class. The wrapper classes also have **parse methods**, which return a value of the corresponding primitive data type. The actual name of each parse method includes the name of the primitive data type using camelCase, such as *Integer.parseInt()* in the Integer class and *Double.parseDouble()* in the Double class.

```
Integer.valueOf("42")   //returns an Integer object
Integer.ParseInt("42") // returns a primitive int value
```

The *valueOf* methods and the *parse* methods in the wrapper classes are static methods, which means they are class-level methods, which is why we use the class name when we invoke them, as in `Integer.value("123")`, similar to the way the name of the Math class is used with the static method to find a square root -- `Math.sqrt(x)`.

Java will automatically type cast from a wrapper class to its corresponding primitive data type, and from a primitive data type to its corresponding wrapper class. These two processes are known as *autoboxing* and unboxing (or, *autounboxing*). **Autoboxing** is the conversion of a primitive data type to an instance of

the corresponding wrapper class. The primitive data is captured as a property of the object. **Unboxing** is the opposite. The following code works, but it is inefficient because of the unboxing that occurs:

```
int AdamsNumber;
AdamsNumber = Integer.valueOf("42");
```

`Integer.valueOf("42")` returns an *Integer* class object which is type cast to an *int* value by being assigned to a variable of type *int*. It would be better to go directly from a String to an *int* value by using a parse method:

```
int AdamsNumber;
AdamsNumber = Integer.ParseInt("42");
```

Each wrapper class method also has a *ValueOf* method that will explicitly perform autoboxing on values of its corresponding primitive data type, such as *Integer.valueOf(int x)*

Each Number class's *valueOf* and *parse* methods only work with data of the corresponding primitive data type. *Integer.ParseInt(),* for example, will not work on a double value. *Double.parseDouble()* will not work on an int value. If the data being converted is not in the correct format for its data type, then a *NumberFormatException* will be generated. Because of this we should use exception handling try and catch blocks when we use these methods.

The following program converts data to Strings with exception handling for data type errors. It is included with the files for this chapter in the zipped NetBeans project *ValueOfTest.zip* . You can try running the program several times with different input values to see how this works.

```
/*
 * ValueOfTest.java
 * program to demonstrate converting String data to numbers
 * CSCI 112 Fall 2014
 * last edited Jan. 24, 2013 by C. Herbert
 */
package valueoftest;
import java.util.Scanner;

public class ValueOfTest {

    /* The main method gets keyboard input and attempts to convert it to
     *  an integer and a double value, then displays the result.
     */
    public static void main(String[] args)
    {
           //declare variables

         String inString;   // entered by the user
         int newInt;        // holds input converted to an integer
         double newDouble;  // holds input converted to a double

         // set up instance of Scanner for input
         Scanner kb = new Scanner(System.in);

         // get the input as a String
         System.out.println("This program will attempt to convert String input to
         a numeric data.");
```

```
         System.out.print("\nPlease enter a String to be converted to a number: ");
         inString = kb.nextLine();

         System.out.println("\nYou entered: " + inString);

         // convert to an int value
         try {
            newInt = Integer.ParseInt(inString);
            System.out.println("As an int value it is: " + newInt);
         } // END TRY
         catch (NumberFormatException e) {
            // bad as an int error message
            System.out.println("The input cannot be converted to an int");
         }   // end catch

         // convert to a double value
         try {
            newDouble = Double.parseDouble(inString);
            System.out.println("As a double value it is: " + newDouble);
         } // end try
         catch (NumberFormatException e) {
            // bad as adouble error message
            System.out.println("the input cannot be converted to a double");
         } // end catch

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

15.3 Retrieving Characters and Substrings from within a String

The String class has methods for finding and for manipulating characters within a String, and for finding substrings within a string. A **substring** is a String that is part of another String, such as the substring "*box*" within the string "*autoboxing*".

The characters within a String objectmay be addressed according to the index of each character, as if the String is an array of characters. The first (leftmost) character in a String has index 0, while the last (rightmost) character in a String has the index *length()-1*. *length()* is a String method that returns as an int value the number of characters in the String. The index of the last character is one less than the length of the String, just as with any array. The index values for the characters in the String "Pennsylvania", with length 11, are shown below:

P	e	n	n	s	y	v	a	n	i	a
0	1	2	3	4	5	6	7	8	9	10

charAt() and *substring()* Methods

The String method `charAt(int i)` returns a *char* with the character at index i.

The String class has two methods named *substring*, which each return part of a String as a new String.

- `String substring(int beginIndex)` returns a String starting at `char[beginIndex]` and continuing through to the end of the String.
- `String substring(int beginIndex, int endIndex)` returns a String from `char[beginIndex]` up to but not including `char[endIndex]`.

The following program illustrates the use of the ***charAt*** method and the two **substring** methods. It is with the files for this chapter as the zipped NetBeans project *SubStringDemo.zip.*

```
/*
 * SubStringDemo.java
 * program to demonstrate substring operations
 * CSCI 112 Fall 2014
 * last edited Jan. 24, 2013 by C. Herbert
 */
package substringdemo;

public class SubStringDemo {

    /* The main method demonstrates the use of several String functions
     * such as charAt(), substring()
     */
    public static void main(String[] args) {

   String colony = "Pennsylvania";

   System.out.println("the String is: " + colony);

   for ( int i =0; i < colony.length(); i++)
      System.out.println("the character at index " + i + " is\t" + colony.charAt(i) );

   System.out.println();
   System.out.print("substring(4) is ");
   System.out.println(colony.substring(4));

   System.out.print("substring(0,4) is ");
   System.out.println(colony.substring(0,4));

   System.out.print("substring(4,10) is ");
   System.out.println(colony.substring(4,10));

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

The output for this code looks like this:

```
the String is: Pennsylvania
the character at index 0 is   P
the character at index 1 is   e
the character at index 2 is   n
the character at index 3 is   n
the character at index 4 is   s
the character at index 5 is   y
the character at index 6 is   l
the character at index 7 is   v
the character at index 8 is   a
the character at index 9 is   n
the character at index 10 is  i
the character at index 11 is  a

substring(4) is sylvania
substring(0,4) is Penn
substring(4,10) is sylvan
```

Contains and *IndexOf* methods

In addition to methods that can return data from a specified location in a String, the String class has search methods that can be used to find and return specified *char* or String data from within a String. These are the *contains()* method and four pairs of *indexOf* and *lastIndexOf* methods described in the following table:

Method	Description
`boolean contains(String s)`	returns **`true`** if the String contains the specified character sequence; returns **`false`** otherwise
`int indexOf(int ch)`	returns the index of the first occurrence of the specified character
`int indexOf(String str)`	returns the index of the first occurrence of the specified substring
`int indexOf(int ch, int fromIndex)`	returns the index of the first occurrence of the specified character, searching forward (to the right) from the specified *fromIndex*
`int indexOf(String str, int fromIndex)`	returns the index of the first occurrence of the specified substring, searching forward (to the right) from the specified *fromIndex*
`int lastIndexOf(int ch)`	returns the index of the last occurrence of the specified character
`int lastIndexOf(String str)`	returns the index of the last occurrence of the specified substring
`int lastIndexOf(int ch, int fromIndex)`	returns the index of the last occurrence of the specified character searching backward (to the left) from the specified from index
`int lastIndexOf(String str, int fromIndex)`	returns the index of the last occurrence of the specified substring, searching backward (to the left) from the specified index
Note: The index of methods return -1 if the String does not contain the target character or substring.	

The *contains()* method can be used to determine if one String is contained in another String . The following example shows how this can be used. It is in the files for this chapter as the zipped Netbeans project *ContainsDemo.zip.* You should read through the program and try it to see how it works.

```
/*
 * ContainsDemo.java
 * This program demonstrates the String class contains() method
 * CSCI 112 Fall 2014
 * last edited Jan. 24, 2013 by C. Herbert
 */
package containsdemo;
import java.util.Scanner;

public class ContainsDemo {

    /* The main method gets the name of a school as String input,
     * then checks to see if it contains specific substrings --
     * such as "College", "University", or "High School"
     */
```

```
    public static void main(String[] args) {

        String inString;   // entered by the user

        // set up instance of Scanner for input
         Scanner kb = new Scanner(System.in);

         System.out.print("Please enter The name of your school: ");
         inString = kb.nextLine();

         if ( inString.contains("College") )
              System.out.println("Your school name contains the word \"College\".");

         if ( inString.contains("University") )
              System.out.println("Your school name contains the word \"University\".");

         if ( inString.contains("High") )
             if ( inString.contains("High School") )
             System.out.println("Your school name contains the term \"High School\".");
             else
             {
                 System.out.print("Your school name contains the word \"High\" ");
                 System.out.println("but not \"School\".");
                 System.out.println("I suspect it is really " + inString + " School.");
             }

         if ( inString.contains("Academy") )
              System.out.println("Your school name contains the word \"Academy\".");

         if ( inString.contains("Institute") )
              System.out.println("Your school name contains the word \"Institute\".");

         System.out.println("\nThank You");

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

The following program demonstrates the use of *indexOf* and *lastIndexOf* methods. It is in the files for this chapter as the zipped Netbeans project *StringSearchDemo.zip.*

```
/*
 * StringSearchDemo.java
 * program to demonstrate indexOf() and lastIndexOf() methods
 * CSCI 112 Fall 2014
 * last edited Jan. 24, 2013 by C. Herbert
 */
package stringsearchdemo;

public class StringSearchDemo {

     /* The main method demonstrates the use of several IndexOf()
      *  and lastIndexOf() methods
      */
    public static void main(String[] args) {

        String colony = "Pennsylvania";
        int position; // position within the String
```

```
        System.out.println("The String is: " + colony);
        System.out.println();

        // find the first position of 'y' in the String
        findChar(colony, 'y');

        // find the first position of the first letter 'n'
        findChar(colony, 'n');

        // find the first position of the last letter 'n'
        findLastChar(colony, 'n');

        // find the position of the substring  "sylvan"
        findSubString(colony, "sylvan");

        // find the position of the substring  "woods"
        findSubString(colony, "woods");

    }// end main()

/**********************************************************************
 * The findChar method searchers String s  for the first occurrence of character c
 * If found, it tells us the position of the character in the String.
 * If not found, it tells us the Character is not in the String.
 */
static void findChar(String s, char c) {

    int position;
    position = s.indexOf(c);    // returns -1 if not found
    if (position < 0)
        System.out.println(c + " is not in the String");
        else
        System.out.println("The first \'" + c + "\' is at position " + position);

} //end findChar()

/**********************************************************************
 * The findLastChar method searchers String s  for the last occurrence of character c
 * If found, it tells us the last position of the character in the String.
 * If not found, it tells us the Character is not in the String.
 */
static void findLastChar(String s, char c) {

    int position;
    position = s.lastIndexOf(c);    // returns -1 if not found
    if (position < 0)
        System.out.println("\'" + c + "\'" + " is not in the String");
        else
        System.out.println("The last \'" + c + "\' is at position " + position);

} //end findChar()


/**********************************************************************
 * The findSubString method searchers String s  for the first occurrence of
 * the substring ss
 * If found, it tells us the position of the character in the String.
```

```
 * If not found, it tells us the Character is not in the String.
 */
static void findSubString(String s, String ss) {

    int position;
    position = s.indexOf(ss);   // returns -1 if not found
    if (position < 0)
        System.out.println("\"" + ss + "\"" + " is not in the String");
        else
        System.out.println( "\"" + ss + "\"" + " begins at  position " + position);

} //end findChar()
} // end class SubStringDemo
```

Here is the output from the *StringSearchDemo* program:

```
The String is: Pennsylvania


The first 'y' is at position 5

The first 'n' is at position 2

The last 'n' is at position 9

"sylvan" begins at  position 4

"woods" is not in the String
```

15.4 Manipulating Strings, Substrings, and Characters within a String

In addition to methods that can find substrings and characters within a String, the String class has methods that will manipulate Strings, substrings, and characters within a String. These include methods to:

- replace substrings and characters with a different String or character
- trim a String by removing part of the String
- convert a String to all uppercase or all lowercase characters

Replacing SubStrings within a String

The String class has four different replace methods, as described in the following table:

Method	Description
`String replace(char oldChar, char newChar)`	Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
`String replace(String target, String replacement)`	Replaces each substring of this string that matches the literal target String with the specified literal replacement String.
`String replaceAll(String regex, String replacement)`	Replaces each substring of this string that matches the given regular expression with the given replacement.
`String replaceFirst(String regex, String replacement)`	Replaces the first substring of this string that matches the given regular expression with the given replacement.

The above table shows us that the *replaceAll()* and *replaceFirst()* methods use *regular expressions*. A **regular expression** is a formally defined use of symbols to select specific character strings from a set of character strings. Three different types of regular expressions are defined in the IEEE POSIX standards for computer operating systems. We will not work with regular expressions in this chapter, so we will not use the *replaceAll()* and *replaceFirst()* methods. Anyone who is familiar with the *replace()* methods and needs to create Java software to work with regular expressions should find the *replaceAll()* and *replaceFirst()* methods easy to use, since they work in a manner very similar to the *replace()* method.

For more about POSIX regular expressions see The Open Group / IEEE Joint Standard for Regular Expressions, on the Web at: http://www.regular-expressions.info/posix.html. The POSIX standard was enacted as *Federal Information Processing Standards 151-2*, requiring almost all information processing systems used by the US Federal Government to be POSIX compliant. (The Federal standard was published in 1993. see: http://www.itl.nist.gov/fipspubs/fip151-2.htm) For more information about using regular expressions in Java, see Oracle's *Java Tutorial* lesson for the *Java Regular Expressions API*, online at: http://docs.oracle.com/javase/tutorial/essential/regex.

The following example shows a use of the replace method with character string data.

Example Program – Converting MS-DOS Style Path Names to POSIX Style Path Names

There are two symbols on modern computer keyboards that are similar to one another: the slash / and the backslash \. (*There is no "forward slash", just the slash and backslash. Only the British say "Forward slash".*) The slash is easy to find – it has been in the same place for more than 135 years: on the bottom right hand corner of the typewriter portion of the keyboard. It appeared on the first keyboard to have upper and lower case characters, the 1878 *Remington Standard No.2* Typewriter, shown below.

The backslash is newer. It is not in the same place on every keyboard. It was used on some teletypes just before World War II, but did not appear on any standard computers or typewriters until the late 1950's. (See http://www.bobbemer.com/BACSLASH.HTM). It is not on the same place on every keyboard.

The original Microsoft operating system, MS-DOS, used a backslash as a delimiter in path names for directories and files. The UNIX operating system and those derived from UNIX, including Linux and Android, use the slash. The *Open Group/ IEEE* POSIX operating system standards call for the use of the slash as the delimiter in path names. To be in compliance with the standard, and to be compatible with

older Microsoft operating systems, Microsoft Windows will allow either the slash or backslash to be used. The backslash is used in MS-DOS style path names and the slash is used in POSIX compliant path names, also called UNIX style path names.

MS-DOS Style Path Name	POSIX Compliant Path Name
C:\Users\Public\Documents\Java textbook\Chapter 15	C:/Users/Public/Documents/Java textbook/Chapter 15

The following Java program uses the String class *replace()* method to convert MS-DOS style path names to POSIX compliant path names. The MS-DOS path names are read in from a text file named *oldPaths.txt*. The POSIX compliant path names are written to a file named *newPaths.txt*.

When you strip away all the file access, a very simple single instruction, in the *convertMStoPOSIX()* method, does the conversion.

```
/*
 * PathNamesDemo.Java
 * program to convert MS-DOS style pathnames to POSIX compliant pathnames
 * using the String replace() command
 * CSCI 111 Fall 2103
 * last edited Sept 24, 2013 by C. Herbert
 */
package pathnamesdemo;

import java.util.Scanner;

public class PathNamesDemo {

    /* The main method reads MS-DOS style pathnames from a file one at a time.
     * It calls a method to convert each pathname to a POSIX compliant name,
     * then writes it to a newe data file.
     *
     * This happens in a loop that processes a file with a set of MS-DOS pathnames,
     * such as a list of where to find installed software.
     */
    public static void main(String[] args) throws Exception {

        String oldPathName;  // MS-DOS style path name
        String newPathName;  // POSIX compliant path name

        // Create a File class object linked to the name of the input file
        java.io.File oldFile = new java.io.File("oldPaths.txt");
        // Create a Scanner named infile to read the input stream from the old file
        Scanner infile = new Scanner(oldFile);

        // Create a File class object linked to the name of the output file
        java.io.File newFile = new java.io.File("newPaths.txt");
        // Create a PrintWriter to write the output stream to the new file
        java.io.PrintWriter outfile = new java.io.PrintWriter(newFile);

        /* This while loop uses a Scanner class boolean function hasNextLine()
         * to see if there another line in the file.
```

```
         *
         * It eads each line, calls the convert method, then writes it to the new file
         */

        while (infile.hasNextLine()) {

            // read a line into oldPathName from the input file stream
            oldPathName = infile.nextLine();

            // convert it to a POSIX name and store in newPathName
            newPathName = convertMStoPOSIX(oldPathName);

            // show the user what's happening
            System.out.print("The MS-DOS style pathname is:    ");
            System.out.println(oldPathName);
            System.out.print("As a POSIX style pathname it is: ");
            System.out.println(newPathName);
            System.out.println(" ");

            // write the converted apth name to the output file stream
            outfile.println(newPathName);

        } // end while

        // close files
        infile.close();
        outfile.close();

    } // end main()

    /**
     * ********************************************************************
     * The convertMStoPOSIX method replaces all slashes in a path name with
     * backslashes If converts MS style path name to POSIX style path names
     */
    static String convertMStoPOSIX(String oldPath) {

        /* note - the first character below is the backslash character.
         * in Java the backslash in a String is used to indicate a special character,
         * such as \t for the tab charaacter. so '\\' is the backslash by itself.
         * see: http://docs.oracle.com/javase/tutorial/java/data/characters.html
         */
        return oldPath.replace('\\', '/');

    } //end convertMStoPOSIX()
} // end class PathNamesDemo
```

Other Useful String and Character Manipulation Methods

The tables below show several other String class and Character class methods that can be useful in processing text data.

String Methods	Description
`String trim()`	Returns a copy of this string with leading and trailing white space removed.
`String toLower()`	Returns a copy of this string converted to all lowercase.
`String toUpper()`	Returns a copy of this string converted to all uppercase.

Character Methods	Description
`char toLowerCase(char ch)`	Returns the lowercase form of the specified char value.
`char toUpperCase(char ch))`	Returns the uppercase form of the specified char value.
`boolean isLowerCase(char ch)`	Returns true if the specified char is lowercase.
`boolean isUpperCase(char ch)`	Returns true if the specified char is uppercase.
`boolean isLetter(char ch)`	Returns true if the specified char is a letter.
`boolean isDigit(char ch)`	Returns true if the specified char is a numeric digit.
`boolean isWhitespace(char ch)`	Returns true if the specified char is a white space.
NOTE: The Character methods listed in this table are all static methods, so they are used with the name of the class, such as Character.toUpper(c); where *c* is a *char* variable or expression that returns a *char* value.	

The String *toLower()* and *toUpper()* methods convert an entire String to all lowercase or all uppercase, while the Character *toLowerCase()* and *toUpperCase()* methods do the same thing, but to a single character. The String methods could also be used to do this to a single character.

We can combine Character methods with String methods to process text data. Here are some short examples.

Example: Data Standardization

Sometimes we need to work with data that is not all in the same format, so we need to standardize the data. In this short example, data should be in the following format:

- the first character should be uppercase
- the other characters should be lowercase

We need a method that will make sure data is in the specified format. There are several ways to do this. Code for one of these ways follows. We will call our method *standardizeString()*. It is used in the Netbeans project *UpperLowerDemo* included as a zipped folder with the files for this chapter.

```
/*
 * UpperLowerDemo.java
 * program to demonstrate the toLowercase() and toUppercase() methods
 * CSCI 112 Fall 2014
 * last edited Jan. 24, 2013 by C. Herbert
 */
package upperlowerdemo;

public class UpperLowerDemo {

    public static void main(String[] args) {

       String testString = "wedNESday";
       String result;

       // call method to convert the string to the right format
       result = standardize(testString);

       System.out.println("The original string is: " + testString);
       System.out.println("The standardized string is: " + result);

    }// end main()

    /**************************************************************************
     * The standardize method returns the input String with
     * the first character uppercase and the rest of the String lowercase
     */
    static String standardize(String inString) {

        // get the first character of inString as a one character String
        String first = inString.substring(0, 1);

        // get the rest of inString
        String rest = inString.substring(1, inString.length());

        // return a String with first uppercase and rest lowercase
           return first.toUpperCase() + rest.toLowerCase();

       // Note that this whole method could all be done in one instruction
       // return inString.substring(0, 1).toUpperCase()
       //    + inString.substring(1,inString.length()).toLowerCase();

    } // end standardize()
} // end class UpperLowerDemo
```

15.5 Comparing and Testing Strings and Substrings

The String class implements the Comparable interface, which means it has a *compareTo()* method. The *compareTo()* method compares the current String to a String used as an argument of the method, and returns and integer value that indicates which string comes first.

```
int result = stringA.compareTo(stringB);
```

Strings are compared using the lexicographical order of the string's underlying character set. This means Strings are compared to one another by comparing each successive character of the two Strings in order, from left to right. On most systems, the characters are compared using their Unicode values.

If the character in position 0 of each String is the same, then the comparison moves on to position 1, and so on, until it finds two characters in the same respective positions that are different, or until it reaches the end one of the Strings. Once the computer finds a difference between the Strings, the comparison ends. If two Strings are the same except that one ends and one continues, then the shorter comes before the longer. If two strings are different, then the one with the lowest Unicode value at the first position where they differ is comes before the other. Two Strings are the same only if they are the same at each successive position and have the same number of characters.

`stringA.compareTo(stringB)` will return the integer 0 if the Strings are the same. If the current String (stringA in the example) comes before stringB then the method returns a negative integer.

If the stringA comes after the stringB then the method returns a positive integer.

Example: putting two Strings in order

The following code demonstrates the use of the String *compareTo* method to put two Strings in the correct lexicographical order. It is included within the zipped Netbeans project *SortStringsDemo* included with the files for this chapter.

```
        // swap the Strings if B should come before A
        if ( StringA.compareTo(StringB) > 0)  {
            StringC = StringA;   // use the catalyst to save a copy of StringA
            StringA = StringB;
            StringB = StringC;
        } // end if
```

Java also has a method to compare strings ignoring their case, the `compareToIgnoreCase()` method.

In addition to the two *compareTo* methods that return integer results, the String class uses two boolean comparison methods that return true if two Strings are the same:

```
boolean equals(Object anObject)

boolean equalsIgnoreCase(String anotherString)
```

Notice that the equals() method works with any Object. It is inherited from the Object class by the String class. The `equalsIgnoreCase()` method is specific to String objects.

Java has several additional Boolean methods that can be used to examine the characters within a string, shown in the following table.

Method	Description
`boolean startsWith(String prefix)`	Returns true if this string begins with the substring argument
`boolean endsWith(String suffix)`	Returns true if this string ends with the substring argument
`boolean startsWith(String prefix, int offset)`	Considers the string beginning at the index offset, and returns true if that part of the String begins with the substring specified as an argument.
`boolean regionMatches(int toffset, String other, int ooffset, int len)`	Tests whether the specified region of this string matches the specified region of the String argument. Region is of length len and begins at the index toffset for this string and ooffset for the other string.
`boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)`	Tests whether the specified region of this string matches the specified region of the String argument. Region is of length len and begins at the index toffset for this string and ooffset for the other string. The boolean argument indicates whether case should be ignored; if true, case is ignored when comparing characters.

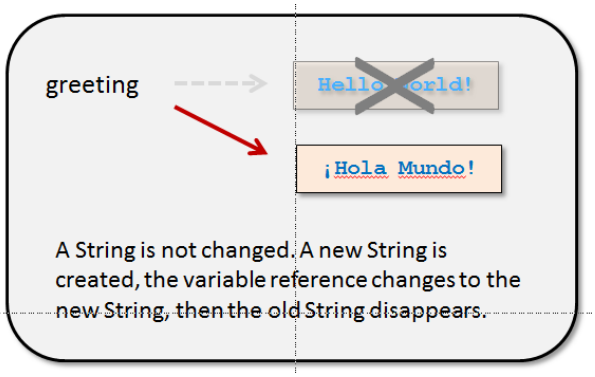
15.6 Immutability and the StringBuilder and StringBuffer Classes

In the Java programming language, a String is **immutable**, which means once it is created in memory, it cannot be modified. It can be deleted or destroyed, but not modified. However, String variables are not immutable. Variables are not immutable; but they can be fixed as constants. Consider the following code:

```
String greeting = "Hello World!";
greeting = "¡Hola Mundo!";
```

It looks as if we just changed the String *"Hello World!"* to *"¡Hola Mundo!"*, but we did not. The String variable *greeting* originally referred to a String object in memory with the value *"Hello World!"* The instruction `greeting = "¡Ola Mundial!"` actually created a new String object with the value *"¡Hola Mundo!"* in a new memory location, then changed the variable *greeting* to point to the new String.

The following diagram shows the situation after the two instructions above are executed.



The **StringBuilder** class in Java is similar to the String class, except that its objects are not immutable, they can be changed. A *String* object in Java is an *immutable* sequence of characters. A *StringBuilder* object is a *mutable* sequence of characters.

A *StringBuilder* object's mutability allows for several methods in the *StringBuilder* class that do not exist in the String class. Among these are several *append*, *delete* and *insert*, methods, along with a *reverse* method and, of course, a *toString* method. The class also has a *replace* method that works a little differently from the String class *replace* methods.

The *StringBuilder* class has several constructors, including one that will construct a *StringBuilder* object from an existing *String* object.

Here is a summary of some useful *StringBuilder* methods that do not exist for *String* objects:

Method	Description
`StringBuilder( String s)`	A constructor that instantiates a StringBuilder object from an existing String.

`StringBuilder append(boolean b)`
`StringBuilder append(char c)`
`StringBuilder append(char[] str)`
`StringBuilder append(char[] str, int offset, int len)`
`StringBuilder append(double d)`
`StringBuilder append(float f)`
`StringBuilder append(int i)`
`StringBuilder append(long lng)`
`StringBuilder append(Object obj)`
`StringBuilder append(String s)`
	Each of the append methods appends the argument to this string builder. Data is cast to a string before the append operation takes place.

`StringBuilder delete(int start, int end)` `StringBuilder deleteCharAt(int index)`	The first method deletes the substring from *start* up to but not including *end* in the StringBuilder's character sequence. The second method deletes the character located at *index*.

`StringBuilder insert(int offset, boolean b)`
`StringBuilder insert(int offset, char c)`
`StringBuilder insert(int offset, char[] str)`
`StringBuilder insert(int index, char[] str, int offset, int len)`
`StringBuilder insert(int offset, double d)`
`StringBuilder insert(int offset, float f)`
`StringBuilder insert(int offset, int i)`
`StringBuilder insert(int offset, long lng)`
`StringBuilder insert(int offset, Object obj)`
`StringBuilder insert(int offset, String s)`
	Each of the insert methods inserts the second argument into the StringBuilder at the location specified by *offset*. Inserted data is cast to a string before the insert operation takes place.

`StringBuilder replace(int start, int end, String s)`	Replaces the characters starting at *start* up to but not including *end* in this StringBuulder object with the specified String. The String does not need to be the same length as the number of characters being replaced.
`void setCharAt(int index, char c)`	Replaces the character at *index* with the specified character.

`StringBuilder reverse()`	Reverses the sequence of characters in this StringBuilder object.
`String toString()`	Returns a string that contains the same character sequence as thisStringBuilder object.

Complete specifications for the *Stringuilder* class are in the Java 7 documentation, on the Web at:
http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html

Oracle's java tutorials include a lesson about the *StirngBuilder* class, on the Web at:
http://docs.oracle.com/javase/tutorial/java/data/buffers.html

Almost all of the things that can be done with the *StringBuilder* class can be done with a combination of methods from the *String* and *Character* class, so why use the *StringBuilder* class? There are two reasons: ease of use and efficiency.

The methods in the *StringBuilder* class make it easier for a programmer to write code to manipulate strings than by using just *String* and *Character* methods. Consider exercise 7 at the end of this chapter. It is often easier to create an instance of *StringBuilder* from a String, work with the character string, then convert the resulting *StringBuilder* object back to a *String*.

It is also more efficient because *Stringbuilder* objects are mutable. The *String* class must create a new *String* object in memory and change a variable reference each time we change a *String* value. The *StringBuilder* class simply changes the existing object, which is both spatially and temporally more efficient. In other words, it uses less memory and takes less time to complete.

The StringBuffer Class

The java language also has a *StringBuffer* class that is very similar to the *StringBuilder* class, but it has a synchronization feature that makes it safe to use in multithreaded programming, when more than one method may be running at the same time manipulating the same data. The use of the *StringBuffer* class for multithreaded programming is beyond the scope of this chapter, but the *StringBuffer* has many of the same methods as the *StringBuilder* class, including all of those described in the table above.

Chapter Review

Section 1 of this chapter discussed the String class in Java. Key terms included *character string* and *concatenation operator*.

Section 2 described converting Strings to primitive data types and primitive data types to Strings. It included a discussion of Java's wrapper classes and the concept of autoboxing and unboxing. Key terms included *wrapper class,number classes, parse methods, autoboxing,* and *unboxing.*

Section 3 talked about retrieving characters and substrings from a String. The *charAt, IndexOf* and *contains* methods were described, with examples. The key term substring was defined.

Section 4 discussed manipulating strings, substrings, and characters within a string. An example in the chapter mentioned *regular expressions* and the *POSIX* standards and talked about the slash and backslash characters. The key term *regular expression* was defined.

Section 5 was about comparing Strings and substrings and testing the characteristics of strings using the compareTo and equals methods and several Boolean methods that test strings and substrings for certain characteristics.

Section 6 discussed the *StringBuilder* class and how a StringBulder object differs from a String object. The notion of immutable objects was discussed. The StringBuffer class was briefly disucssed. Key terms included *immutable object*, *StringBuilder*, and *StringBuffer*.

.

Chapter Questions

1. How are Strings related to character data? What is the default character set of a String? What version of Unicode code do most modern computers use?
2. How is the String class different from most other classes? How are String variables and constants created? How are String literals implemented in Java?
3. What method do other classes often have to create a String representing an instance of an object? How can primitive data be turned into a String?
4. What does the plus sign mean when used with String data? What is it called?
5. Which methods in the String class convert primitive data types to Strings? The textbook says this set of methods is polymorphic; what does that mean? What primitive data type is not covered by these methods?
6. What methods does the String class have to convert Strings to primitive data types? What classes have methods to convert Strings to primitive data types? What happens to the value of a primitive data item when it is autoboxed into an object?
7. What advantages are in using wrapper classes? What are the number classes in java?
8. What methods are more efficient than the ValueOf methods for converting Strings to primitive data types? What must we use when we invoke these methods and why?
9. How can individual characters within a String object be addressed? How are they numbered?
10. What method returns a character from a String? What method returns a set of characters from a String? How does this method's numbering work in specifying the beginning and ending characters to return?
11. What boolean method can be used to tell if a string contains a specified substring? What methods can be used to find the index of a character or a substring in a string? What value do these methods return instead of the index if the specified character or substring is not in the String?
12. What is a regular expression? How are regular expressions related to the POSIX standards? Why are the POSIX standards important for US federal government information processing systems?
13. What delimiter should be used in file pathnames to be POSIX compliant? What other delimiter does Microsoft Windows accept and why?
14. What method can be used to eliminate leading and trailing white spaces from a String?

15. What is the difference between the *toUpper()* method and the *toUpperCase()* methods? What values do these methods return?
16. What method can be used to tell if a character is a letter of the alphabet? What method can be used to tell if a character id a numeric digit?
17. What do we know must be true if a class implements the Comparable interface? What is the meaning of the values returned by the String class *compareTo()* method?
18. According to the chapter, what two boolean methods does the String class have to tell if two Strings are the same? How are they different from one another?
19. What does it mean to say that an object is immutable? What happens when we change the name of a variable referencing an immutable object, such as a String variable?
20. How do StringBuilder objects differ from String objects? Why should we consider using the StringBuilder class? How is the StringBuffer class different from the StringBuilder class?

Chapter Exercises

1. Password Checking

   Software to reset passwords often requires the user to enter the password twice, checking to make sure it was entered the same way both times.

   Write a method in Java that can do this. Your method should ask the user to enter a password twice, then either tell the user that the two entries were not the same and start the process over again, or, if they are the same, tell the user that the new password was accepted.

2. Coded Messages

   A substitution cypher substitutes one letter for another according to some pattern. For example, each character could be replaced by a character 10 letters higher in the alphabet – A is preplaced by K, B is replaced by L, and so on. at the end of the alphabet, the code wraps around, so W is replaced by G. Uncoded messages are plain text. Coded messages are cypher text.

   Write a program with one method to convert from plain text to cypher text and another method to convert from cypher text to plain text. Your methods should only encode characters that are letters, and should leave numbers, spaces, punctuation marks, or other characters as is.

   You should create a program that asks the user to choose coding or decoding, then asks the user to enter a message. The program should then code or decode the message and display the result.

3. Finding File Types by Extension

   A data file contains a list of names of files on a computer system, with one file name on each line. It has simple file names, not full path names. The file names contain no periods, except a period before the file extension, which identifies what kind of file it is.

   Your task is to write a program to read the names from the data file and create a list of files that need to be scanned for viruses. Files that have the extension .BAT, .COM, .DOC, .DOCX, .EXE, .HTM, .HTML. XLS and .XLSX need to be scanned.

4. Creating User Names from Email Addresses

   Email addresses are of the form *name@domain.xxx* where *xxx* identifies the domain type. A new online software subscription service is free to anyone who is a student or who works for a school or college. Users are considered students if they have educational email address ending in "*.edu*" The person's user name for the subscription service must be the name part of their email address.

   Your task is to create a method that will ask for the user's email address, verify that it is an educational email address, then tell the user his or her user name, then ask the user to enter a new password.

   If the email address is not an educational email address, the program should tell the user this and allow the user to start over or end the program.

5. VIP Phone List

   A listing of US phone numbers for very important people should include a person's last name, first name, and phone number, with phone numbers in the form AAA-XXX-XXXX where AAA is the area code and both A and X are numeric digits.

   The current list of phone numbers for very important people, stored on a disk drive, has each one set of (last name, first name, and phone number) on each line, with the fields separated by slashes. The phone numbers are in different formats. Some use parenthesis, some have no dashes, and some use white spaces or other symbols to separate parts of the phone number. Here are some examples:

   Obama/Barack/(202) 456-1111
   Nelson/Craig/215 751-8000
   Christie/Chris/609-292-6000
   Corbett/Tom/717-787-2500
   Hickenlooper/John/(303) 866-2003
   Herbert/Gary/8015381000
   Cuomo/Andrew/518 474 8390
   Martinez/Susana/ (505) 476-2200

   Your task is to write a program that will read the data from a file, standardize the data with phone numbers in the format specified, then write the data back to a new disk file.

   You should submit a zipped file with your program as NetBeans project and a programming report.

6. Proper Passwords

This assignment is an extension of assignment 1, above. Proper passwords for a specific system are required to follow the these rules:

- The password must be at least 8 characters long.
- The password must contain at least:
  - one alpha character [a-zA-Z];
  - one numeric character [0-9];
  - one character that is not alpha or numeric, such as
    ! @ $ % ^ & * ( ) - _ = + [ ] ; : ' " , < . > / ?
- The password must not:
  - contain spaces;
  - begin with an exclamation [!] or a question mark [?];
- The password cannot contain repeating character strings of 3 or more identical characters, such as “1111” or “aaa”.

Your task is to write a method to verify whether or not a prospective proper password meets these requirements.

7. Palindromes

A palindrome is word or phrase that it the same forward or backward. For example, the name “*Otto*”is a palindrome – but only if we ignore case. The phrase “Able was I ere I saw Elba” is another example. “*Yo, banana boy!*” is a palindrome if we ignore capitalization, punctuation and spacing. The actor *Robert Trebor* (born in Philadelphia) has a name that is a palindrome. Palindromic sequences can be found in DNA and RNA. Numbers can be palindromic: The continued fraction equal to $\sqrt{n} + \lfloor \sqrt{n} \rfloor$ is a repeating palindrome when *n* is an integer. The year 2002 was the last palindromic year.

The Website http://www.palindromelist.net claims to have “The biggest list of palindromes online”.

Your task is to write software that has the following:

A. a method that uses only *String* and *Character* methods to tell us if a String is a palindrome just as it is, without changing case or ignoring non-letter characters.

B. a method that uses the *StringBuilder reverse()* method to tell us if a String is a palindrome just as it is, without changing case or ignoring non-letter characters. This is a first class palindrome.

C. a method that will tell us if the letters in a String form a palindrome if we ignore case, white spaces, and punctuation. This is a second class palindrome.

D. a complete Java program that asks the user to enter a String, then tells the user if the String is a first class palindrome, a second class palindrome, or not a palindrome. First class palindromes do not need to be tested to see if they are second class palindromes.

You should submit in a single zipped folder, a copy of the method for A, above, and a copy of the NetBeans project with the complete program from D that uses the methods described in B and C, along with a report on this assignment.

8. Phone Numbers Local Exchange Names

Back in the middle of the Twentieth Century, it was a common practice to use words and letters to help people remember phone numbers. For example 235-3610 would be BE5-3610, which would be remembered as Bedford 5 – 3295. The first two letters of the word would be used to match the numbers on the phone. A standard telephone keypad (a dial in the old days) shows the correspondence between letters and numbers.

The names actually corresponded to names of local telephone branch exchanges. Here are some examples:

Allegheny 4 1978
Mayfair 4 3425
Germantown 8 4387
Victor 7 3248

(See: http://phone.net46.net/philadelphia-city/latealphnumer.html for a list of a telephone exchange names in Philadelphia in 1946.)

Your task is to write a program to read a data file with a list of old style phone numbers that use exchange names and rewrite the list using all numeric phone numbers. You should also add the 215 area code to the front of each number and output the numbers in standard format, such as (215) 438-4387. The numbers are currently in the format: [name][white space][digit][white space][four digits].

You should submit a single zipped folder with the Netbeans project and a programming report.

9. Checking Credit Card Numbers

Credit card numbers follow certain patterns. A credit card number must have between 13 and 16 digits. It must start with:

- 4 for Visa cards
- 5 for MasterCard cards
- 37 for American Express cards
- 6 for Discover cards

Hans Luhn of IBM proposed an algorithm for validating credit card numbers. The algorithm is useful to determine if a card number is entered correctly or if a credit card is scanned correctly by a scanner. Most major credit card numbers are generated following this validity check, commonly known as the Luhn Check or the Mod 10 Check, which can be described as follows (for illustration, consider the card number 4388576018402626):

1. Double every second digit from right to left. If doubling of a digit results in a two-digit number, add up the two digits to get a single-digit number.

2 * 2 = 4
2 * 2 = 4
4 * 2 = 8
1 * 2 = 2
6 * 2 = 12 (1 + 2 = 3)
5 * 2 = 10 (1 + 0 = 1)
8 * 2 = 16 (1 + 6 = 7)
4 * 2 = 8

2. Now add all single-digit numbers from Step 1.

4 + 4 + 8 + 2 + 3 + 1 + 7 + 8 = 37

3. Add all digits in the odd places from right to left in the card number.

6 + 6 + 0 + 8 + 0 + 7 + 8 + 3 = 38

4. Sum the results from Step 2 and Step 3.

37 + 38 = 75

5. If the result from Step 4 is divisible by 10, the card number is valid; otherwise, it is invalid. For example, the number 4388576018402626 is invalid, but the number 4388576018410707 is a valid Visa Card.

Write a program to validate Visa and MasterCard credit cards. The program should prompts the user to enter a credit card number as a String. It should display whether the card is a valid MasterCard a valid Visa card, or invalid. Your program should be composed of several smaller methods to perform each part of the verification process, with a main method that calls each of the smaller methods.