***Chapter 16***

## *Strings and Characters in Java*

**String** is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.

The **1st** thing to understand about strings is that every string you create is actually an object of type **String**. Even string constants are actually String objects. For example, in the statement:

 System.out.println("Coach Dependahl");

"Coach Dependahl" is a **String** constant.

The 2nd thing to understand about strings is that objects of type String are immutable; once a String object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:

* If you need to change a string, you can always create a new one that contains the modifications
* Java defines a peer class of **String**, called **StringBuffer**, which allows strings to be altered, so all of the normal string manipulations are still available in Java.

This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, there is a companion class to **String** called **StringBuffer**. **StringBuffer** objects contain strings that can be modified after they are created. Both the **String** and **StringBuffer** classes are defined in **java.lang**. Thus, they are available to all programs automatically. Both are declared **final**, which means that neither of these classes may be subclassed.

Strings can be constructed in a variety of ways. The easiest is to use a statement like this:

 String myString = "Coach Dependahl"

Once you have created a **String** object, you can use it anywhere that a string is allowed. For example, this statement displays myString:

 System.out.println(myString);

Java defines one operator for String objects: +. It is used to concatenate two strings. For example:

 String myString = "Coach" + " " + "Dependahl"

Results in myString containing "Coach Dependahl"

The **String** class contains several methods that you can use. Here are a few. You can test two strings for equality by using **equals( )**. You can obtain the length of a string by calling the **length( )** method. You can obtain the character at a specified index within a string by calling **charAt( )**. Note, like arrays, the first element of a **String** is considered to be at position 0.

String myString = "Coach Dependahl";

 String myString2 = "Coach";

 System.out.println("Length of myString=" + myString.length( ) );

 System.out.println("Char 3 in myString: " + myString.charAt(2));

 If (myString.equals(myString2))

 System.out.println("myString equals myString2");

 else System.out.println("myString does not equal myString2");

 **Output Produced:**

 **Length of myString=15**

 **Char 3 in myString: a**

 **myString does not equal myString2**

 // Demonstrate String Arrays

 class StringDemo {

 public static void main(String args[ ]) {

 String str[ ] = {"one" , "two" , "three"};

 for (int x=0; x<str.length; x++)

 System.out.println("str[" + x + "]: " + str[x]);

 }

 }

 **Output Produced:**

 **str[0]: one**

 **str[1]: two**

 **str[2]: three**

*Command-Line Arguments*

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing *command-line arguments* to **main( )**. A command-line argument is the information that directly follows the program's name on the command line when it is executed. Accessing command-line arguments inside a Java program is quite easy - they are stored as strings in the String array passed to **main( )**.

 // Display all command-line arguments.

 class CommandLine {

 public static void main(String args[ ]) {

 for(int x=0; x<args.length; x++)

 System.out.println("args[" + x + "]: " + args[x]);

 }

 }

To execute this program: jview CommandLine this is a test 100 -1

**Output:**

 **args[0]: this**

 **args[1]: is**

 **args[2]: a**

 **args[3]: test**

 **args[4]: 100**

 **args[5]: -1**

*String Constructors*

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

**Examples**

 String s = new String( );

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

 char chars[ ] = {'a', 'b', 'c'};

 String s = new String(chars);

This constructor initializes **s** with the string "abc".

 char chars[ ] = {'a', 'b', 'c', 'd', 'e', 'f'};

 String s = new String(chars, 2, 3);

This initializes **s** with the characters **cde**.

 char c[ ] = {'J', 'a', 'v', 'a'};

 String s1 = new String(c);

 String s2 = new String(s1);

 String s3 = "Java";

This code defines and initializes strings **s1**, **s2**, and **s3** to "Java".

*String Concatenation*

In general, Java does not allow operators to be applied to String objects. The one exception to this rule is the **+** operator. The **+** concatenates two strings, producing a String object as the result. This allows you to chain together a series of **+** operations.

 String firstname = "Coach";

 String lastname = "Dependahl";

 String fullname = firstname + " " + lastname;

When concatenating strings, Java will automatically convert items of a different type (i.e., int) into a string before performing the concatenation. But when doing this, be careful:

 String s = "four: " + 2 + 2;

 System.out.println(s);

Will display: **four: 22**

 String s = "four: " + (2 + 2);

Now, **s** contains the string: **four: 4**

*toString ( )*

Every class implements **toString( )** because it is defined by Object. However, the default implementation is seldom sufficient. For most classes that you create, you will want to override **toString( )** and provide your own string representations:

// Override toString( ) for Box class

class Box {

 int width;

 int height;

 int depth;

 Box ( int w, int h, int d ) {

 width = w;

 height = h;

 depth = d;

 }

public String toString( ) {

 return "Dimensions are " + width + " by " +

 depth + " by " + height + ".";

 }

}

class toStringDemo {

}

public static void main(String args[ ]) {

 Box b new Box(10, 12, 14);

 String s = "Box b: " + b; // concatenate Box object

 System.out.println(b); // convert Box to string

 System.out.println(s);

}

**Output: Dimensions are 10 by 14 by 12.**

 **Box b: Dimensions are 10 by 14 by 12.**

*Character Extraction*

The **String** class provides a number of ways that characters can be extracted from a **String** object:

**charAt( )**: extracts a single character from a string.

 char ch;

 ch = "abc".charAt(1); // assigns the value "**b**" to **ch**.

**getChars( )**: extracts more than one character at a time.

 String s = "This is a demo of getChars";

 char HoldingArea[ ] = new char[4];

 s.getChars(10, 14, HoldingArea, 0); // HoldingArea gets "demo"

**getBytes( )** and **toCharArray( )** are two additional methods for character extraction. However, use of getChars can be used to accomplish the same tasks.

*String Comparison*

The String class includes several methods that compare strings or substrings within strings:

**Equals( )** and **equalsIgnoreCase( )** : returns true if the strings contain the same characters in the same order and false otherwise. With equals( ), the comparison is case-sensitive.

 String s1 = "Coach";

 String s2 = "Coach";

 String s3 = "COACH";

 System.out.println(s1 + " equals " + s2 + s1.equals(s2)); //?\_\_\_\_\_\_\_

 System.out.println(s1 + " equals " + s3 + s1.equals(s3)); //?\_\_\_\_\_\_\_

 System.out.println(s2 + " equals " + s3 + s2.equals(s3)); //?\_\_\_\_\_\_\_

 System.out.println(s1 + " equals " + s2 +

 s1.equalsIgnoreCase(s2)); //?\_\_\_\_\_\_\_

 System.out.println(s1 + " equals " + s3 +

 s1.equalsIgnoreCase(s3)); //?\_\_\_\_\_\_\_

 System.out.println(s2 + " equals " + s3 +

 s2.equalsIgnoreCase(s3)); //?\_\_\_\_\_\_\_

**regionMatches( )** : compares a specific region inside a string with another specific region in another string.

 String s4 = "Hi to Coach Dependahl";

 If s1.regionMatches(0, s4, 6, 5) System.out.println("Found a match");

A boolean value **true** (preceding the start index) may be passed to compare ignoring case.

**startsWith( )** and **endsWith( )** : are specialized forms of **regionMatches( )**. **startsWith( )** determines if a given **String** begins with a specified string. Conversely, **endsWith( )** determines if the **String** in question ends with a specified string. -- **"Coach Dependahl".startsWith("Coach")** is true.

 **"Coach Dependahl".startsWith("Dependahl",6)** is true.

**Equals( )** versus **= =** : It is important to understand that the **equals( )** method and the **= =** operator perform two different operations. **equals( )** compares the characters inside a **String** object. The **= =** operator compares two object references to see if they refer to the same instance.

class EqualsNotEqualTo {

}

public static void main(String args[ ]) {

 String s1 = "Coach";

 String s2 = new String(s1);

 String s3 = "Coach";

 System.out.println(s1 + " equals " + s2 + s1.equals(s2));

 System.out.println(s1 + " = = " + s2 + (s1 = = s2);

 System.out.println(s1 + " = = " + s3 + (s1 = = s3);

}

**Output: Coach equals Coach true**

 **Coach = = Coach false**

 **Coach = = Coach true**

Note: for the 2nd case, s1 and s2 are uniquely different string objects (since s2 was reconstructed using “new String(s1)”. In the 3rd case, s1 and s3 end up pointing to the same object, so when comparing with = = , their addresses will be the same, therefore we’ll get a true returned.

**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. **compareTo( )** has the following form:

int compareTo(String *str*)

It returns an integer value according to the following:

Value Meaning

Less than 0 The invoking string is less than *str*

Greater than 0 The invoking string is greater than *str*

Zero The two strings are equal

// Bubble sort of strings

class SortString {

 static String args[ ] = {"Coach" , "Dependahl", "absolutely", "loves", "Java"};

}

public static void main(String args[ ]) {

 for (int y = 0; y < arr.length; y++) {

 for (int x = y+1; x < arr.length; x++) {

 if (arr[x].compareTo(arr[y]) < 0) {

 String z = arr[y];

 arr[y] = arr[x];

 arr[x] = z;

 }

 }

 System.out.println(arr[y]);

 }

}

**Output:**

**Coach**

**Dependahl**

**Java**

**absolutely**

**loves**

*Searching Strings*

**indexOf( )**: searches for the first occurrence of a character or substring. These two methods are overloaded several different ways.

**lastIndexOf( )**: searches for the last occurrence of a character or substring.

In all cases, the methods return the index at which the character or substring was found or -1 on failure.

 int indexOf(int *ch*): searches for the first occurrence of a character

 int lastIndexOf(int *ch*): searches for the last occurrence of a character

 int indexOf(String *str*): searches for the first occurrence of a string

 int lastIndexOf(String *str*): searches for the last occurrence of a string

In each of these methods, a 2nd parameter can be passed (of type int) which specifies where to begin searching.

*Modifying a String*

Since **String** objects are immutable, whenever you want to modify a **String**, you must either copy it into a **StringBuffer** or use one of the following **String** methods which constructs a new copy of the string with your modifications complete.

**Substring( )**: to extract a substring

 String substring(int *startIndex*)

*StartIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string. The second form of **substring( )** allows you to specify both the beginning and ending index of the substring:

 String substring(int *startIndex*, int *endIndex*)

The following program uses substring( ) to replace all instances of one substring with another within a string:

// Substring replacement

class StringReplace{

 public static void main(String args[ ]){

 String org = "This is a test. This is, too.";

 String search = "is";

 String sub = "was";

 String result = " ";

 int i;

 do { // replace all matching substrings

 System.out.println(org);

 i = org.indexOf(search);

 if (i != -1) {

 result = org.substring(0, i);

 result = result + sub;

 result = result + org.substring(i + search.length( ));

 org = result;

 }

 } while (i != -1);

 }

}

***Output:***

This is a test. This is, too.

Thwas is a test. This is, too.

Thwas was a test. This is, too.

Thwas was a test. Thwas is, too.

Thwas was a test. Thwas was, too.

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

String replace (char *original*, char *replacement*)

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

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

Puts the string "Hewwo" into s.

**trim( )**: a method that returns a copy of the invoking string from which any leading and trailing white-space 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.

*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. **Object** is a superclass for all classes. All **Object**s can be converted to **String**s with the **toString** method. 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*[ ])

You can call this method directly with any data type and get a reasonable **String** representation. All of the simple types are converted to their common **String** representation. Any object that you pass 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.

*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.

***Assignment for Chapter 16***

Write an applet that uses random number generation to create sentences. Use four arrays of strings called **article**, **noun**, **verb** and **preposition**. Create a sentence by selecting a word at random from each array in the following order: **article**, **noun**, **verb**, **preposition**, **article** and **noun**. As each word is picked, concatenate it to the previous words in the sentence. The words should be separated by spaces. When the final sentence is output, it should start with a capital letter and end with a period. The program should generate 20 sentences and display them on an applet window.

The arrays should be filled as follows: The article array should contain the articles “**the**”, “**a**”, “**one**”, “**some**” and “**any**”; the noun array should contain the articles “**boy**”, “**girl**”, “**dog**”, “**town**” and “**car**”; the verb array should contain the verbs “drove”, “**jumped**”, “**ran**”, “**walked**”, and “**skipped**”; the preposition array should contain the prepositions “**to**”, “**from**”, “**over**”, “**under**” and “**on**”.