STUDENT LESSON

A10 – The String Class

STUDENT LESSON

A10 – The String Class

INTRODUCTION: Strings are needed in many programming tasks. Much of the information that identifies a person must be stored as a string: name, address, city, social security number, etc. This lesson covers the specifications of the `String` class and how to use it to solve string-processing problems.

The key topics for this lesson are:

A. The `String` Class
B. `String` Constructors
C. Object References
D. The `null` Value
E. `String` Query Methods
F. `String` Translation Methods
G. Immutability of `Strings`
H. Comparing `Strings`
I. `Strings` and Characters
J. The `toString` Method
K. `String` I/O

VOCABULARY:

charAt
CONCATENATION
GARBAGE
IMMUTABLE
lengthOf
nextLine
STRING CLASS
substring
toString
trim
compareTo
equals
GARBAGE COLLECTION
length
next
null
STRING LITERAL
toLowerCase
toUpperCase

DISCUSSION:

A. The `String` Class

1. Groups of characters in Java are not represented by primitive types as are **`int`** or **`char`** types. Strings are objects of the `String` class. The `String` class is defined in `java.lang.String`, which is automatically imported for use in every program you write. We've used String literals, such as `"Enter a value"` with `System.out.print` statements in earlier examples. Now we can begin to explore the `String` class and the capabilities that it offers.

2. So far, our experience with Strings has been with String literals, consisting of any sequence of characters enclosed within double quotation marks. For example:

```
"This is a string"
"Hello World!"
"\tHello World!\n"
```

The characters that a `String` object contains can include escape sequences. This example contains a tab (`\t`) and a linefeed (`\n`) character.

3. A second unique characteristic of the `String` class is that it supports the `"+"` operator to concatenate two `String` expressions. For example:

```
sentence = "I " + "want " + "to be a " + "Java programmer.";
```

The `"+"` operator can be used to combine a `String` expression with any other expression of primitive type. When this occurs, the primitive expression is converted to a `String` representation and concatenated with the string. For example, consider the following instruction sequence:

```
PI = 3.14159;
System.out.println("The value of PI is " + PI);
```

Run Output:

```
The value of PI is 3.14159
```

To invoke the concatenation, at least one of the items must be a String.

B. `String` Constructors

1. Because Strings are objects, you can create a `String` object by using the keyword **new** and a `String` constructor method, just as you would create any other object.

```
String name = new String();
String name2 = new String("Nancy");
```

2. Though they are not primitive types, strings are so important and frequently used that Java provides additional syntax for declaration:

```
String aGreeting = "Hello world";
```

A `String` created in this short-cut way is called a *String literal*. Only `Strings` have a shortcut like this. All other objects are constructed by using the **new** operator.

Many new Java programmers get confused because of this shortcut and believe that `Strings` are primitive data types. However, `Strings` are objects and therefore have behaviors and attributes.

C. Object References

1. Recall from Lesson A2 that an object is constructed as an instance of a particular class. The object is most often referenced using an identifier. The identifier is a variable that stores the reference to the object. This identifier is called an object reference. Now that we are working with a simple class, `String`, it is a good time to discuss object references.

2. Whenever the **new** operator is used, a new object is created. Each time an object is created, there is a reference to where it is stored in memory. The reference can be saved in a variable. The reference is used to find the object.

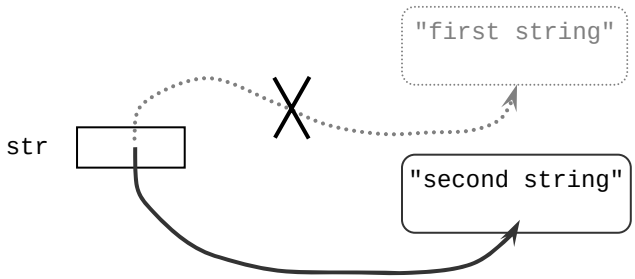
3. It is possible to store a new object reference in a variable. For example:

```
String str;

str = new String("first string");
System.out.println(str);

str = new String("second string");
System.out.println(str);
```

Run Output:

```
first string
second string
```

In the example above, the variable `str` is used to store a reference to the `String`, “first string”. In the second part of the example a reference to the `String`, “second string” is stored in the variable `str`. If another reference is saved in the variable, it *replaces* the previous reference (see diagram below).



4. If a reference to an object is no longer being used then there is no way to find it, and it becomes "*garbage*." The word "garbage" is the correct term from computer science to use for objects that have no references. This is a common situation when new objects are created and old ones become unneeded during the execution of a program. While a program is running, a part of the Java system called the "garbage collector" reclaims each lost object (the "garbage") so that the memory is available again. In the above example, the `String` object “first string” becomes garbage.

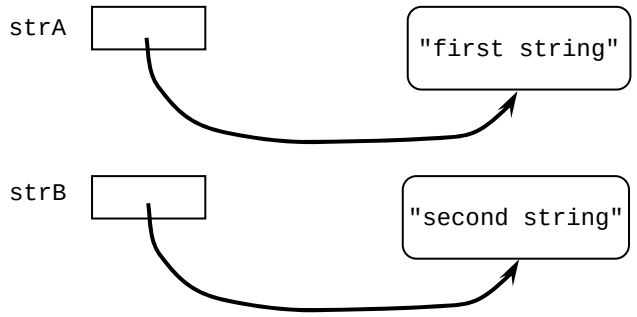
5. Multiple objects of the same class can be maintained by creating unique reference variables for each object.

```
String strA;  // reference to the first object
String strB;  // reference to the second object

// create the first object and save its reference
strA = new String("first string");

// print data referenced by the first object.
System.out.println(strA);

// create the second object and save its reference
strB = new String("second string");

// print data referenced by the second object.
System.out.println(strB);

// print data referenced by the first object.
System.out.println(strA);
```

Run Output:

```
first string
second string
first string
```

This program has two reference variables, `strA` and `strB`. It creates two objects and places each reference in one of the variables. Since each object has its own reference variable, no reference is lost, and no object becomes garbage (until the program has finished running).

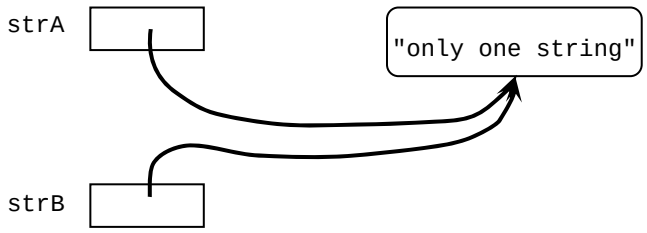


6. Different reference variables that refer to the same object are called *aliases*. In effect, there are two names for the same object. For example:

```
String strA;  // reference to the object
String strB;  // another reference to the object

// Create the only object and save its
// reference in strA
strA = new String("only one string");
System.out.println(strA);

strB = strA;  // copy the reference to strB.
System.out.println(strB);
```

Run Output:

```
only one string
only one string
```



When this program runs, only one object is created (by **new**). Information about how to find the object is put into `strA`. The assignment operator in the statement

```
strB = strA; // copy the reference to strB
```

copies the information that is in `strA` to `strB`. It does not make a copy of the object.

D. The **`null`** Value

1. In most programs, objects are created and objects are destroyed, depending on the data and on what is being computed. A reference variable sometimes does and sometimes does not refer to an object. You may need a way to erase the reference inside a variable without creating a new reference. You do this by assigning **`null`** to the variable.

2. The value **`null`** is a special value that means "no object." A reference variable is set to **`null`** when it is not referring to any object.

```
String a =                // 1. an object is created;
    new String("stringy"); //    variable a refers to it
String b = null;          // 2. variable b refers to no
                          //    object.
String c =                // 3. an object is created
    new String("");       //    (containing no characters)
                          //    variable c refers to it
if (a != null)            // 4. statement true, so
   System.out.println(a); //    the println(a) executes.

if (b != null)            // 5. statement false, so the
   System.out.println(b); //    println(b) is skipped.

if (c != null)            // 6. statement true, so the
   System.out.println(c); //    println(c) executes (but
                          //    it has no characters to
                          //    print).
```

Run Output:

```
stringy
```

3. Variables `a` and `c` are initialized to object references. Variable `b` is initialized to **`null`**. Note that variable `c` is initialized to a *reference* to a String object containing no characters. Therefore `println(c)` executes, but it has no

characters to print. Having no characters is different from the value being **`null`**.

E. String Query Methods

Query Method	Sample Syntax
`int length();`	`String str1 = "Hello!";` `int len = str1.length();  // len == 6`
`char charAt(int index);`	`String str1 = "Hello!";` `char ch = str1.charAt(0); // ch == 'H'`

`int indexOf(String str);`	`String str2 = "Hi World!";`
	`int n = str2.indexOf("World"); // n == 3`
	`int n = str2.indexOf("Sun");   // n == -1`

`int indexOf(char ch);`	`String str2 = "Hi World!";`
	`int n = str2.indexOf('!');  // n == 8`
	`int n = str2.indexOf('T');  // n == -1`

1. The **`int`** `length()` method returns the number of characters in the `String` object.

2. The `charAt` method is a tool for extracting a character from within a `String`. The `charAt` parameter specifies the position of the desired character (0 for the leftmost character, 1 for the second from the left, etc.). For example, executing the following two instructions prints the char value `'X'`.

```
String stringVar = "VWXYZ"
System.out.println(stringVar.charAt(2));
```

3. The **`int`** `indexOf(String str)` method will find the first occurrence of `str` within this `String` and return the index of the first character. If `str` does not occur in this `String`, the method returns -1.

4. The **`int`** `indexOf(`**`char`** `ch)` method is identical in function and output to the other `indexOf` function except it is looking for a single character.

F. String Translation Methods

Translate Method	Sample Syntax

`String toLowerCase();`	`String greeting = "Hi World!";`
	`greeting = greeting.toLowerCase();`
	`// greeting <- "hi world!"`

`String toUpperCase();`	`String greeting = "Hi World!";`
	`greeting = greeting.toUpperCase();`
	`// greeting <- "HI WORLD!"`

`String trim();`	`String needsTrim = "  trim me!  ";`
	`needsTrim = needsTrim.trim();`
	`// needsTrim <- "trim me!"`

`String substring(int beginIndex)`	`String sample = "hamburger";`
	`sample = sample.substring(3);`
	`// sample <- "burger"`

`String substring(int beginIndex, int endIndex)`	`String sample = "hamburger";`
	`sample = sample.substring(4, 8);`
	`// sample <- "urge"`

1. `toLowerCase()` returns a `String` with the same characters as the `String` object, but with all characters converted to lowercase. Notice that in all of the above samples, the `String` object is placed on the left hand side of the assignment statement. This is necessary because `Strings` in Java are immutable. Please see section G for a full explanation of immutable.

2. `toUpperCase()` returns a `String` with the same characters as the `String` object, but with all characters converted to uppercase.

3. `trim()` returns a `String` with the same characters as the `String` object, but with the leading and trailing whitespace removed.

4. `substring(int beginIndex)` returns the substring of the `String` object starting from `beginIndex` through to the end of the `String` object.

5. `substring(int beginIndex, int endIndex)` returns the substring of the `String` object starting from `beginIndex` through, but not including, position `endIndex` of the `String` object. That is, the new `String` contains characters numbered `beginIndex` to `endIndex-1` in the original `String`.

G. Immutability of `Strings`

Immutability of `Strings` means you cannot modify any `String` object.

Notice the above example for the method `toLowerCase.` This method returns a new `String`, which is the lower case version of the object that invoked the method.

```
String greeting = "Hi World!";
greeting.toLowerCase();
System.out.println(greeting);
```

Run Output:

```
Hi World!
```

The object greeting did not change. To change the value of greeting, you need to assign the return value of the method to the object greeting.

```
greeting = greeting.toLowerCase();
System.out.println(greeting);
```

Run Output:

```
hi world!
```

H. Comparing Strings

1. The following methods should be used when comparing String objects:

Comparison Method	Sample Syntax

`boolean equals(String anotherString);`	`String aName = "Mat";`
	`String anotherName = "Mat";`
	`if (aName.equals(anotherName))`
	`  System.out.println("the same");`

`boolean equalsIgnoreCase(String anotherString);`	`String aName = "Mat";`
	`if (aName.equalsIgnoreCase("MAT"))`
	`  System.out.println("the same");`

`int compareTo(String anotherString)`	`String aName = "Mat"`
	`n = aName.compareTo("Rob"); // n < 0`
	`n = aName.compareTo("Mat"); // n == 0`
	`n = aName.compareTo("Amy"); // n > 0`

2. The `equals()` method evaluates the contents of two `String` objects to determine if they are equivalent. The method returns true if the objects have identical contents. For example, the code below shows two `String` objects and several comparisons. Each of the comparisons evaluate to **`true`**; each comparison results in printing the line `"Name's the same"`

```
String aName = "Mat";
String anotherName = new String("Mat");

if (aName.equals(anotherName))
```

```
  System.out.println("Name's the same");

if (anotherName.equals(aName))
  System.out.println("Name's the same");

if (aName.equals("Mat"))
  System.out.println("Name's the same");
```

Each `String` shown above, `aName` and `anotherName,` is an object of type `String`, so each `String` has access to the `equals()` method. The `aName` object can call `equals()` with `aName.equals(anotherName)`, or the `anotherName` object can call `equals()` with `anotherName.equals(aName)`. The `equals()` method can take either a variable `String` object or a literal `String` as its argument.

In all three examples above, the boolean expression evaluates to `true`.

3. The == operator can create some confusion when comparing objects. The == operator will check the reference value, or address, of where the object is being stored. It will not compare the data members of the objects. Because `Strings` are objects and not primitive data types, `Strings` cannot be compared with the == operator. However, due to the shortcuts that make `String` act in a similar way to primitive types, two `Strings` created without the `new` operator but with the same `String` literal will actually point to the same address in memory. Observe the following code segment and its output:

```
String aGreeting1 = new String("Hello");
String anotherGreeting1 = new String("Hello");

if (aGreeting1 == anotherGreeting1)
  System.out.println("This better not work!");
else
  System.out.println("This prints since each object " +
     "reference is different.");

String aGreeting2 = "Hello";
String anotherGreeting2 = "Hello";


if (aGreeting2 == anotherGreeting2)
  System.out.println("This prints since both " +
       "object references are the same!");
else
  System.out.println("This does not print.");
```

Run Output:

```
This prints since each object reference is different.
This prints since both object references are the same!
```

The objects `aGreeting1` and `anotherGreeting1` are each instantiated using the **`new`** command, which assigns a different reference to each object. The == operator compares the reference to each object, not their contents. Therefore the comparison `(aGreeting1 == anotherGreeting1)` returns **`false`** since the references are different.

The objects `aGreeting2` and `anotherGreeting2` are `String` literals (created without the **`new`** command – i.e. using the short-cut instantiation process unique to `String`s). In this case, Java recognizes that the contents of the objects are the same and it creates only one instance, with `aGreeting2` and `anotherGreeting2` each referencing that instance. Since their references are the same, `(aGreeting2 == anotherGreeting2)` returns **`true`**.

4. When comparing objects to see if they are equal, always use the `equals` method. It would be a rare occasion to care if they are occupying the same memory location. Remember that a `String` is an object!

5. The `equalsIgnoreCase()` method is very similar to the `equals()` method. As its name implies, it ignores case when determining if two `String`s are equivalent. This method is very useful when users type responses to prompts in your program. The `equalsIgnoreCase()` method allows you to test entered data without regard to capitalization.

6. The `compareTo()` method compares the calling `String` object and the `String` argument to see which comes first in the lexicographic ordering. Lexicographic ordering is the same as alphabetical ordering when both strings are either all uppercase or all lowercase. If the calling string is first lexicographically, it returns a negative value. If the two strings are equal, it returns zero. If the argument string comes first lexicographically, it returns a positive number.

```
String bob = "Bob";
String bob2 = "bob";
String steve = "Steve";
System.out.println(bob.compareTo(bob2));
System.out.println(bob2.compareTo(bob));
System.out.println(steve.compareTo(bob2));
System.out.println(bob.compareTo(steve));
```

The output for this block of code would be:

```
-32
32
-15
-17
```

I. Strings and Characters

1. It is natural to think of a **`char`** as a `String` of length 1. Unfortunately, in Java the **`char`** and `String` types are incompatible since a `String` is an object and a **`char`** is a primitive type. This means that you cannot use a `String` in place of a `char` or use a `char` in place of a `String`.

2. Extracting a **`char`** from within a `String` can be accomplished using the `charAt` method as previously described.

3. Conversion from **`char`** to `String` can be accomplished by using the `"+"` (concatenation) operator described previously. Concatenating any **`char`** with an *empty string* (`String` of length zero) results in a string that consists of that **`char`**. The java notation for an empty string is two consecutive double quotation marks. For example, to convert myChar to a String it is added to “”.

```
char myChar = 'X';
String myString = "" + myChar;
System.out.println(myString);
char anotherChar = 'Y';
myString += anotherChar;
System.out.println(myString);
```

The output of this block of code would be:

```
X
XY
```

J. The toString method

1. Wouldn't it be nice to be able to output objects that you have made using the simple line System.out.print(Object name)? Let's consider the example of the RegularPolygon class discussed in Lesson A6. It would be nice to be able to print out the statistics of your RegularPolygon objects without having to do a lot of System.out.print statements. Thanks to the toString method, you have the ability to do this.

2. You can create a toString method in any of your classes in the format of public String toString(). Within the toString() method, you can format your class variables into one String object and return that String. Then, when Java encounters your Object in a String format, it will call the toString() method. Let's look at an example using a RegularPolygon class.

```
public String toString(){
   String a = "Sides: " + getSides();
   a += " Length: " + getLength();
   a += " Area: " + getArea();
   return a;
}

RegularPolygon square = new RegularPolygon(4, 10);
System.out.println(square);
```

Run Output:

```
Sides: 4 Length: 10 Area: 100
```

3. You must be careful when using this, because you are fixing the format of the output. Oftentimes, you will still want to format your output depending on the specific problem you are solving, but the `toString()` method provides a simple and quick way to look at the state of your objects. There are also many times when the `toString()` method will be very useful. Consider a Student class that contains member variables for first name, middle name, last name, a list of classes being taken, the student's address

and phone number, etc. You could easily make a `toString()` method that would simply output the students first name, middle initial, and last name for quick reference. Every time you design a class, you should stop and think about whether or not your class would benefit from having a `toString()` method and how you should format this `String`.

K. `String` I/O

1. The `Scanner` class has two methods for reading textual input from the keyboard.

2. The `next` method returns a reference to a `String` object that has from zero to many characters typed by the user at the keyboard. The `String` will end whenever it reaches white space. White space is defined as blank spaces, tabs, or newline characters in the input stream. When inputting from the keyboard, `next` stops adding text to the `String` object when the first white space is encountered from the input stream.

3. A `nextLine` method returns a reference to a `String` object that contains from zero to many characters entered by the user. With `nextLine`, the `String` object may contain blank spaces and tabs but will end when it reaches a newline character. Therefore, `nextLine` will read in whole lines of input rather than only one word at a time.

4. String input is illustrated below.

```
Scanner keyboard = new Scanner(System.in);
String word1, word2, anotherLine;

// prompt for input from the keyboard
System.out.print("Enter a line: ");

// grab the first "word"
word1 = keyboard.next();

// grab the second "word"
word2 = keyboard.next();

// prompt for input from the keyboard
System.out.print("Enter another line: ");

// discard any remaining input from previous line
// and read the next line of input
anotherLine = keyboard.nextLine(); //skip to the next line
anotherLine = keyboard.nextLine(); //grab all of the next line

// output the strings
System.out.println("word1 = " + word1);
System.out.println("word2 = " + word2);
System.out.println("anotherLine = " + anotherLine);
```

Run Output:

```
Enter a line: Hello World! This will be discarded.
Enter another line: This line includes whitespace.
word1 = Hello
word2 = World!
```

```
anotherLine = This line includes whitespace.
```

5. Formatting Strings is done with the same style as using the `printf()` method discussed in Lesson A7, *Simple I/O*. In fact, now that you know more about Strings, you should be able to recognize that you are really manipulating String literals when you use the `printf()` formatting rules. If you want to alter how a String object is stored without actually printing it to the String, you can simply use a Formatter object (which is actually the object that `printf()` uses itself). An example of how to use Formatter is shown below. Note: Don't forget to import the Formatter class.

```
import java.util.Formatter;

Formatter f = new Formatter();
f.format("%10s","Bob");
String bob = f.toString();
System.out.println(bob.length());
System.out.println(bob);
```

Run Output:

```
10
       Bob
```

SUMMARY/ REVIEW: The use of pre-existing code (classes) has helped reduce the time and cost of developing new programs. In this lesson, you will use the `String` class without knowing its inner details. This is an excellent example of data type abstraction.

ASSIGNMENT:
Lab Assignment A10.1, *StringUtil*
Lab Assignment A10.2, *CarRental*
Lab Assignment A10.3, *RomanNumerals*
Worksheet A10.1, *String Review*
Worksheet A10.2, *Object References*