STUDENT OUTLINE

Lesson 12 – String Class

INTRODUCTION: Strings are needed in many large programming tasks. Much of the information that identifies a person must be stored as a string: name, address, city, social security number, etc.

Although the `String` class is technically not an actual part of the Java language, a standard `String` class is provided with Java by Oracle.

Your job in this lesson is to learn the specifications of the `String` class and use it to solve string-processing questions.

The key topics for this lesson are:

A. The String Class
B. String Constructors
C. String Query Methods
D. String Translate Methods
E. Comparing Strings
F. Strings and Characters

VOCABULARY: String Class

DISCUSSION:

A. The `String` Class

1. Character strings in Java are not represented by primitive types as are integers (`int`) or single characters (`char`). Strings are represented as 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 character string literals, such as `"Enter a value"` in earlier examples. Now we can begin to explore the `String` class and the capabilities that it offers.

STRING LITERALS ARE AUTOMATICALLY CONVERTED TO String OBJECTS AT COMPILATION

2. A `String` is the only Java reference data type that has a built-in syntax for constants. These constants, referred to as string literals, consist of any sequence of characters enclosed within double quotations. For example:

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

The characters that a `String` object contains can include control characters. The last example contains a tab (`\t`) and a newline (`\n`) character, specified with escape sequences.

String sentence ;

SEE NEXT PAGE

3. A second unique characteristic of the `String` type 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);
```

Its A String!

Run output:

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

4. `String` shares some of the characteristics with the primitive types, however, `String` *is a reference type, not a primitive type*. As a result, assigning one `String` to another copies the reference to the same `String` object. Similarly, each `String` method returns a new `String` object.

5. The rest of the student outline details a partial list of the methods provided by the Java `String` class. For a detailed listing of all of the capabilities of the class, please refer to Oracle's Java documentation at:

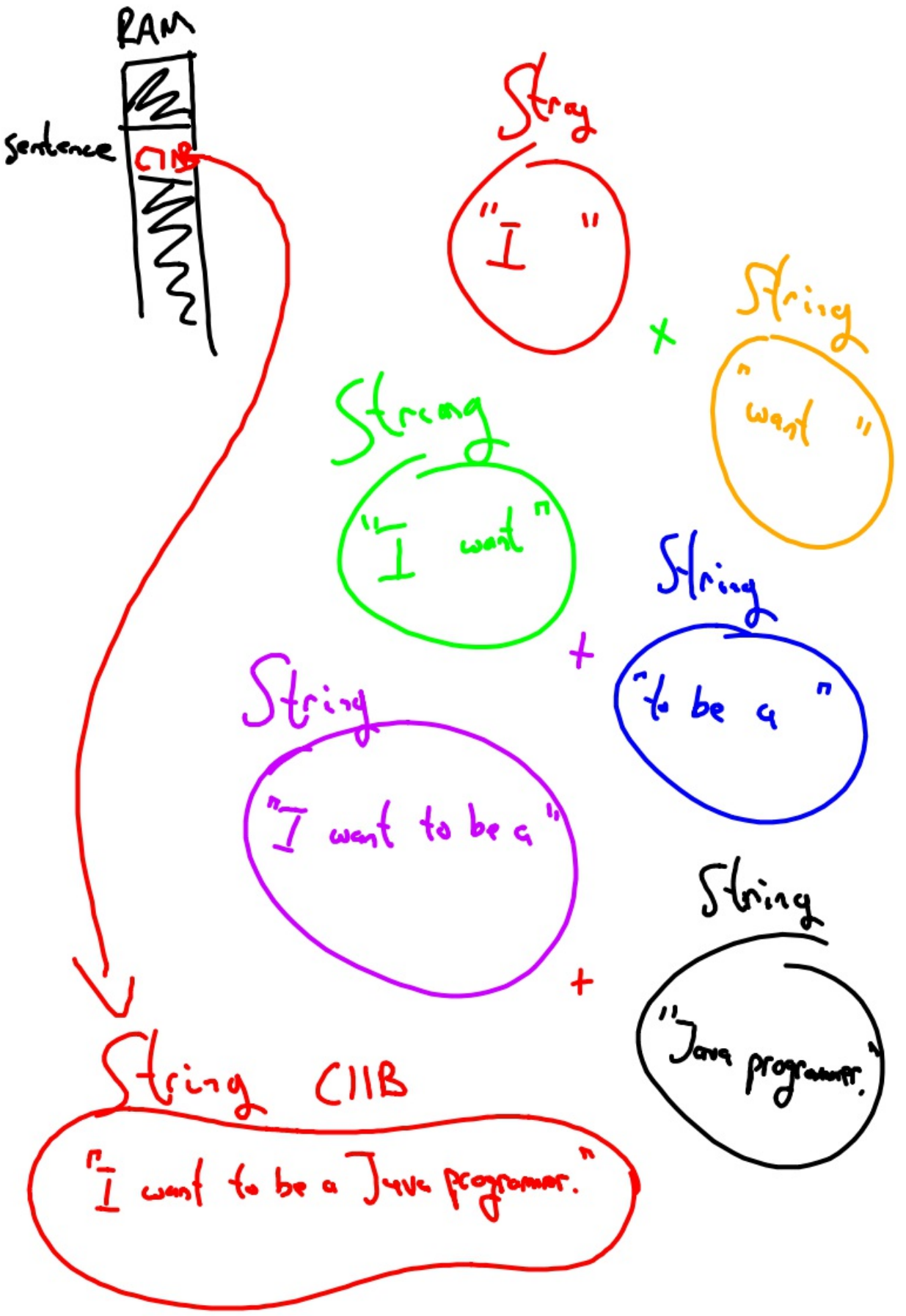
   https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

B. String Constructors

1. You create a `String` object by using the keyword **new** and the `String` constructor method, just as you would create an object of any other type.

Constructor	Sample syntax
`String();`	`// emptyString is a reference to an empty string` `String emptyString = new String();`

`String(String value);`	`String aGreeting;`
	
	`// aGreeting is reference to a String`
	`aGreeting = new String("Hello world");`
	
	`// Shortcut for constructing String objects.`
	`aGreeting = "Hello world";`
	
	`// anotherGreeting is a copy of aGreeting`
	`String anotherGreeting = new String(aGreeting);`


RAM
sentence
C11B
String
"I "
x
String
"want "
String
"I want "
+
String
"to be a "
String
"I want to be a "
+
String
"Java programmer."
String C11B
"I want to be a Java programmer."

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 short-cut like this. All other objects are constructed by using the **new** operator.

C. 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'`
	`***NOT IN AP SUBSET. DO NOT USE ON EXAMS ***`

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

`int indexOf(int 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.

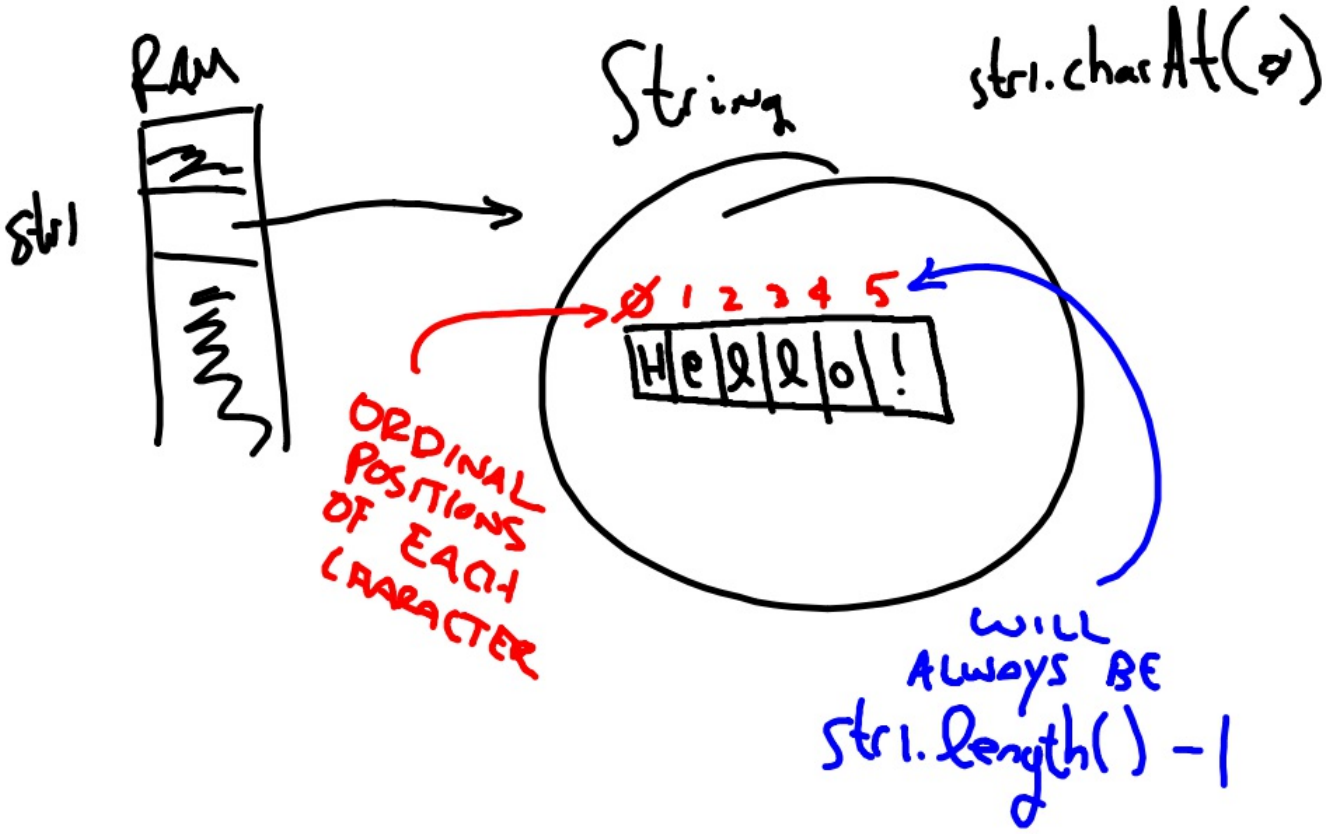
2. The `charAt` method is a tool for extracting a character from within a `String`. The `charAt` parameter specifies the position of the desired character (0 for the leftmost character, 1 for the second from the left, etc.). For example, executing the following two instructions, prints the char value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

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

While useful, this method is NOT in the AP subset and shouldn't be used on your exams and avoided if possible in later lesson labs!!!!

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


RAM
str1
String
str1.charAt(0)
0 1 2 3 4 5
H e l l o !
ORDINAL POSITIONS OF EACH CHARACTER
WILL ALWAYS BE
str1.length() - 1

String s1;
s1 = greeting.toLowerCase();

ORIGINAL OBJECT IS NOT ALTERED

D. String Translate Methods

Translate Method	Sample syntax

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

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

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

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

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

YOU REALLY WANT TO KNOW HOW TO USE THESE!

char INCLUDED

char EXCLUDED

1. `toLowerCase()` returns a `String` with the same characters as the `String` object, but with all characters converted to lowercase.
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.

E. 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");`
	`NOT IN AP SUBSET. DO NOT USE ON EXAMS`

`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`

COMPARES CONTENTS OF STRINGS

MAY USE FOR LABS 12.1 – 12.3

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 = "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");
```

ALIASES, ∴ == WOULD ALSO BE TRUE

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.

TESTS LOCATIONS, NOT CONTENTS!!

3. The == operator can create some confusion when comparing `String` objects. 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.");
```

ALIASES

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 `Strings`). 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**.

BECAUSE THIS COMPARES CONTENTS

Because of potential problems like those described above, you should always use the `equals()` method to compare the contents of two `String` objects.

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

While useful, this method is NOT in the AP subset and shouldn't be used on your exams and avoided if possible in later lesson labs!!!!

THINKING IN ASCII, CALLING OBJECT MINUS THE PARAMETER OBJECT YIELDS THE "0" VALUE

5. 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, it returns a negative value. If the two strings are equal, it returns zero. If the argument string is first, it returns a positive number.

E. 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 a reference type and a **`char`** is a primitive type. Some of the limitations of this incompatibility are:

   a. You *cannot* pass a **`char`** argument to a `String` parameter (nor the opposite).
   b. You *cannot* use a **`char`** constant in place of a `String` constant.
   c. You *cannot* assign a **`char`** expression to a `String` variable (nor the opposite).

   PARAMETER ARGUMENT

   String emp = "" ;

   The last restriction is particularly troublesome, because there are many times in programs when **`char`** data and `String` data must be used cooperatively.

2. 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, the following expression

```
"" + 'X';
```

   evaluates to a `String` of length `1`, consisting of the letter `"X"`

```
// charVar is a variable of type char
// stringVar is a variable of type String
stringVar = "" + charVar;
```

   The execution of this instruction assigns to `stringVar` a `String` object that consist of the single character from `charVar`.

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 Exercise, L.A.12.1, *Palindrome*
Lab Exercise, L.A.12.2, *Shorthand*
Lab Exercise, L.A.12.3, *Piglatinator*