COP 3330 – Object Oriented Programming in Java
Use of Built In Class: String

Primitives vs. References
A variable in Java can store one of two things:

1) a primitive value
2) a reference to an object

Thus in Java, when you declare an object reference, such as

String name;

you are NOT creating a String object. (A reference is a similar to a pointer. It refers to a place in memory. But in Java, you don't get explicit control over memory addresses.)

In order to actually create a new object, you must instantiate one. In Java, this means calling a constructor using the new operator. (We did this with the Random class already.)

A constructor is a method that every class must have. Its job is to create and initialize an object. Many classes have more than one. Here is an example of a call to the constructor in the String class:

```
String fruit = new String("Watermelon");
```

All calls to constructors are of the form new

<Classname>(<parameter list>);

Often times, constructors are overloaded. This means there are multiple constructors, each of which takes in different parameters. You must choose the constructor appropriate to create the object that you want. In this example, the standard String class constructor takes in a String object. In this example, that object is a String literal.

Once you have created an object, you are allowed to call **instance methods** on that object. The word instance means, "belongs to the object." As previously mentioned, the word static means, "belongs to the class." An instance method automatically has access to the instance variables (the items that comprise the object) of that particular object. Typically, an instance method uses some of this information to do its task and in some instances an instance method may mutate (change) that specific object.

Before we move on, one observation about the String class must be made. No methods in the String class actually *change/modify* an object!!! All of the methods that appear to change a String object actually create a new object that is different than the object the method was called upon. These methods then return the newly created String object. When we create our own objects, it's likely that we'll have methods that can mutate the object the method was called on. There are specific

reasons the developers of Java chose to make String objects immutable, but at least for now, we won't go into those reasons.

Here is an example of a method call on the object we just created:

```
fruit.toUpperCase();
```

What this will do is return a new String object similar to fruit except that all of its uppercase letters are changed to their lowercase equivalents.

For example, if I executed the following line of code

```
String fruit2 = fruit.toUpperCase();
```

Our picture would look like the following:

fruit ----------→ ["Watermelon"]
fruit2 ----------→ ["WATERMELON"]

Also, something else that is "hidden" from the user with Strings is the call of the constructor. The following line of code is perfectly valid:

```
String fruit3 = "Strawberry";
```

It is misleading because one MIGHT think that Strings work like primitives and that fruit3 IS the object itself. But, this code is "automatically changed" to be interpreted as follows:

```
String fruit3 = new String("Strawberry");
```

Furthermore, since Strings are references, the following line of code is also deceiving:

```
fruit3 = fruit2;
```

Here's what the REAL picture looks like:

fruit ----------→ [Watermelon]
fruit2 ----------→ [WATERMELON] ←------ fruit3
[Strawberry]

Since no String reference points to the String object storing "Strawberry", this object will get "garbage collected" away. This means the memory for this object will get freed eventually so it can be used for other things. (In C, this would be a memory leak and that memory would be reserved for the duration of the program. But in Java, because it's interpreted, the interpreter can detect that this memory is unreachable and reclaim it.)

Some Methods in the String Class

Here are a few of the String Class constructors in Java:

Constructor	Description
`String()`	Initializes a newly created String object so that it represents an empty character sequence.
`String(char[] value)`	Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.
`String(String original)`	Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.
`String(StringBuffer buffer)`	Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

In this class we'll mostly use the third constructor, usually implicitly (without explicitly calling new since Java puts the call in for us), but in general, both the second and forth constructors are extremely useful. (The second takes in a character array which is different than a String object and the forth takes in another an object of another built in class, StringBuffer. A StringBuffer object is mutable and more efficiently adds characters to the end of a string.)

Most times though, we'll read in a String from input via the .next() method on a Scanner object and store what this method returns with a String reference.

Now let's take a look at some useful String methods:

Modifier and Type	Method	Description
`char`	`charAt()`	Returns the char value at the specified index.
`int`	`compareTo(String anotherString)`	Compares two strings lexicographically.
`int`	`compareToIgnoreCase(String str)`	Compares two strings lexicographically, ignoring case differences.
`String`	`concat(String str)`	Concatenates the specified string to the end of this string, returning a reference to the newly created String object.
`boolean`	`contains(CharSequence s)`	Returns true if and only if this string contains the specified sequence of char values.
`boolean`	`endsWith(String suffix)`	Tests if this string ends with the specified suffix.
`boolean`	`equals(Object anObject)`	Compares this string to the specified object.

boolean	equalsIgnoreCase(String anotherString)	Compares this String to anotherString, ignoring case considerations.
int	indexOf(int ch)	Returns the index within this string of the first occurrence of the specified character.
int	indexOf(int ch, int fromIndex)	Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
int	length()	Returns the length of this string.
String	replace(char oldChar, char newChar)	Returns a string resulting from replacing all occurrences of oldChar in this string with newChar.
String	substring(int beginIndex)	Returns a string that is a substring of this string, from beginIndex to the end of the string.
String	substring(int beginIndex, int endIndex)	Returns a string that is a substring of this string, starting at beginIndex and ending at endIndex-1, of this String object.
char[]	toCharArray()	Converts this string to a new character array.
String	toLowerCase()	Converts all of the characters in this String to lower case using the rules of the default locale.
String	toUpperCase()	Converts all of the characters in this String to upper case using the rules of the default locale.

There's a lot of functionality in the String class (and more generally in all of Java's built in classes). It's not necessarily an efficient teaching technique to go through each method one by one and show an example of the use of every method. The idea is that once someone has used a particular class a couple times, then they should be able to read through the whole Java Doc and see if there is a method that performs a task that might be helpful for them.

For the remainder of these notes, we'll illustrate the use of a few of these methods, but it's up to the reader to try out each of these methods in main testing out what they do.

String Example: concat, replace, substring

Here is a quick example illustrating the use of concat, replace and substring:

```
public class StringTest {

    public static void main(String[] args) {

        String test1=new String("Happy Birthday");
        String test2, test3, test4;

        test2 = test1.concat(" Trisha!")
        System.out.println("test1 = "+test1);
        System.out.println("test2 = "+test2);

        test3 = test1.replace('H', 'M');
        test3 = test3.replace('a', 'e');
        test3 = test3.replace('p', 'r');
        System.out.println("test1 = "+test1);
        System.out.println("test3 = "+test3);

        test4 = test2.substring(11, 20);
        System.out.println("test2 = "+test2);
        System.out.println("test4 = "+test4);
    }
}
```

Output:

```
test1 = Happy Birthday
test2 = Happy Birthday Trisha!
test1 = Happy Birthday
test3 = Merry Birthdey
test2 = Happy Birthday Trisha!
test4 = day Trish
```

One note about string concatenation. We've already used the plus sign earlier in printing out strings via the print and println methods. In essence, what the plus sign really does is invoke the concat method. Thus, secretly, in Java if we type the expression:

```
s + t
```

where s and t are both of type string, then what Java is really doing is calling

```
s.concat(t)
```

and returning this value. This is why you almost NEVER see the concat method explicitly called.

String Example: length, charAt

The following segment of code reads in a string from the user and then prints out all the letters in the string that are lowercase:

```
Scanner stdin = new Scanner(System.in);
System.out.println("Enter a word.");
String word = stdin.next();

for (int i=0; i<word.length(); i++) {

    char temp = a.charAt(i);
    if (temp >= 'a' && temp <= 'z')
        System.out.print(temp);
}
```

There are a couple key things to notice here:

1) length is a method and thus must have parentheses following it.

2) You can NOT use brackets to index into a string, you must make a call to the charAt method. This method does NOT allow you to CHANGE a character in a given string. Thus, it is ILLEGAL to do the following:

```
word.charAt(2) = 'a';
```

This is because the left-hand side of an assignment statement MUST BE a variable. A method call is NOT a variable.

String Example: Counting occurrences of a letter in a string 2 ways.

The most obvious way to count the number of times a letter occurs in a string is just to loop through each character of the string. But, we could ALSO use the indexOf method to do so. Here we look at both approaches. Here is the first way:

```
import java.util.*;

public class CountChar1 {
    public static void main(String[] args) {

        Scanner stdin = new Scanner(System.in);
        System.out.println("Please enter a string without spaces.");
        String word = stdin.next();
        System.out.println("What char in the string to count?");
        String myChar = stdin.next();
        char c = myChar.charAt(0);

        int res = 0;
        for (int i=0; i<word.length(); i++)
            if (word.charAt(i) == c)
                res++;

        System.out.println(c+" appears in "+word+" "+res+" number of times.");
    }
}
```

Once we get all of the relevant information, we just need to use a loop to reach each valid index into word. We retrieve the character at that index and compare it to the character we are looking for. If it's a match, we add 1 to a counter.

In the second approach, we see that the indexOf method naturally tells us the next location where a character appears. So, we can use it to "jump ahead" to the next location by telling it where to search from (the previous location plus 1). Here's the relevant code after the set up in that example:

```
int loc = word.indexOf(c);
int res = 0;

while (loc != -1) {
    res++;
    loc = word.indexOf(c, loc+1);
}
```

So we start by an initial search, and loop until the indexOf method returns -1, meaning that no further occurrence of the letter was found. If the letter was found at loc, we start looking for it starting at loc+1.

String Example: Character counts for all valid letters in a string.
Now, let's say we are interested in getting these character counts for ALL the letters in the string. This solution isn't efficient, but it will illustrate a rather clever use of the indexOf method.

We'll loop through the string. To determine if we've counted a character previously, we'll call indexOf. If it returns a value less than our current index, this means we've counted this character before. If it returns our current index, then we should count all of the occurrences of that letter.

```
import java.util.*;

public class CharCount3 {

    public static void main(String[] args) {

        Scanner stdin = new Scanner(System.in);
        System.out.println("Please enter a string without spaces.");
        String word = stdin.next();

        System.out.println("char\tcount");
        System.out.println("----\t-----");

        for (int i=0; i<word.length(); i++) {

            char c = word.charAt(i);
            int loc = word.indexOf(c);

            if (loc < i) continue;

            int res = 0;
            while (loc != -1) {
                res++;
                loc = word.indexOf(c, loc+1);
            }

            System.out.println(c+"\t"+res);
        }
    }
}
```

Though inefficient in run-time, the code is compact and makes nice use of the indexOf method.

String Example: First alphabetical string in a list of strings.

In this example we use the compareToIgnoreCase method to find the first alphabetic string without regard to case in a list of strings entered by the user:

```
import java.util.*;

public class FirstAlpha {

    public static void main(String[] args) {

        Scanner stdin = new Scanner(System.in);
        System.out.println("How many strings are you entering?");
        int n = stdin.nextInt();

        System.out.println("Please enter each string, 1 per line.");
        String best = stdin.next();

        for (int i=1; i<n; i++) {

            String cur = stdin.next();
            if (cur.compareToIgnoreCase(best) < 0)
                best = cur;
        }

        System.out.println("First alpha string was "+best);
    }
}
```

Keep in mind that that best and cur are references. Let's say that we enter "apple" for the first string, "Camel" for the second string, and "ant" for the third string. After entering the first two, our picture is:

best ----> ["apple"]
cur -----> ["Camel"]

If we were to do a regular compareTo, "Camel" would come before "apple" because the Ascii value for 'C' is smaller than the Ascii value for 'a'. But since we are ignoring case, this method will return a positive integer and best will not be reset. Since cur exists in the loop only, at the very end of the loop, the picture is only:

best ----> ["apple"]

and cur no longer exists. When i=2 and we enter "ant", the new picture is:

best ----> ["apple"]
cur -----> ["ant"]

Since ant comes before apple alphabetically, the compareToIgnoreCase returns a negative integer which triggers the if and our new picture is:

["apple"]
best -----> ["ant"] <---- cur

The string object storing "apple" will get garbage collected, the string reference cur will disappear, leaving just

best ----> ["ant"], we can then print out best and it'll be the first string alphabetically of the three entered.