String Processing and Comparison

The String Class

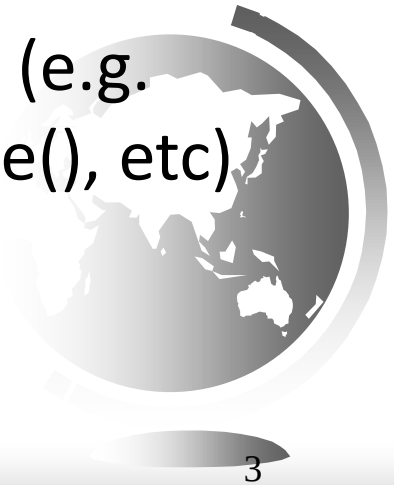
After reading Chapters 4.3, 4.4, and 10.10, you should be familiar with the following terms and concepts:

•Terms:

- What is a string (definition)? What is a string literal?
- What does it mean when we say that a String is a reference type?
- Why are string methods called "instance methods"?
- What are String indexes? What is a StringIndexOutOfBoundsException?
- What does it mean when we say that "strings are immutable"?
- What is an interned string?

Concepts:

- Counting characters in a string, retrieving a character in a string
- How characters are stored in a string object
- How string objects are created
- How string object are stored in memory
- String Methods - substring(), trim(), replace(), indexOf(), matches(), etc.
- The Character wrapper class and its methods (e.g. isDigit(), isLetter(), toLowerCase(), isUpperCase(), etc)



The String Class

After reading Chapters 4.3, 4.4, and 10.10, you should be familiar with the following terms and concepts:

•Terms:

- What is a string (definition)? What is a string literal?
- What does it mean when we say that a String is a reference type?
- Why are string methods called "instance methods"?
- What are String indexes? What is a StringIndexOutOfBoundsException?
- What does it mean when we say that "strings are immutable"?
- What is an interned string?

String Processing

Have a look at the documentation for the String class and the documentation for the Character class.
We will be demonstrating many of the methods in Chapters 4.3 and 4.4 during class. In particular:

Character class methods:

- compareTo()
- isDigit(), isLetter()
- isLetterOrDigit(), isSpaceChar()*, isWhiteSpace()*
- isLowerCase(), isUpperCase()
- toLowerCase(), toUpperCase()

String Processing

String class methods:length()

- charAt()
- substring()
- toLowerCase(), toUpperCase()
- trim()
- replace(), replaceAll()
- matches()
- indexOf(), lastIndexOf()

Exercise - String

Strings and Memory

Read chapter 10.10 and do all of the demonstrations and exercises.

One of the important concepts we often forget or ignore when working with strings is that strings are objects. When you create a string using a statement such as:

```
String s = "Java";
```

You are creating an object variable that points to a String object in memory. This would be just as if you had typed the statement:

```
String s = new String("Java");
```

The people who developed Java figured that since we work with Strings all the time, they'd make it easier to declare Strings as if they were primitive data, so you don't have to use the String() constructor all the time.

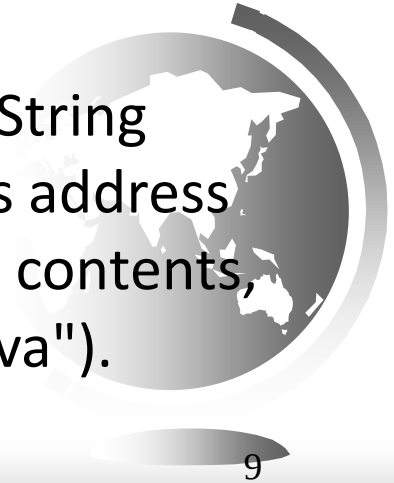
Strings and Memory

Another important concept is that **Strings are immutable**: once created, they can't be changed. Figure 10.15 in Chapter 10.10 also shows the effects of the following:

```
String s = "Java"; s = "HTML";
```

In this example, the String object referenced by the variable s is not being altered. In the first statement, a new String object with the value "Java" is created and its address stored in the variable s.

Then when the second statement executes, a new String object with the value "HTML" is constructed and it's address stored in the variable s. This wipes out the previous contents, which was the address to the first String object ("Java").



Chapter 10.10 also talks about **interned** strings. Since strings are used so often in programming, Java creates a pool of strings that are created in a program. When you create a string object, Java checks to see if there is a similar string object in the pool. If there is, Java will use that string object instead of creating a new one. See the code below Figure 10.15.

When the first statement executes, which assigns the string literal "Welcome to Java." to the String variable **s**, the String class adds the string "Welcome to Java" to the pool of strings. When the second statement executes, which constructs a new String object with the value "Welcome to Java" and assigns that object reference to the variable **s2**, a brand new object is created in memory, so the **s2** variable gets its own object reference. The third statement creates a String variable using the same string literal, "Welcome to Java". In this case, Java searches the pool of Strings to see if one matches. It finds the one that is being references by **s1**, so it uses that one. This causes **s3** to point to the same String object as **s1**.

This is why the two println() statements product the output you see in the middle of the page. The **s1** and **s2** variables are pointing to, or referencing, different objects, so they are not equal. The **s1** and **s3** variables are referencing the same object, so they are equal.

String Comparison

Read Chapter 4.4.7

You don't use relational operators to compare strings. For example, an expression such as:

```
"hello" > "world"
```

...would not work in Java. In this statement, we're attempting to check to see if the string "hello" is greater than the string "world".

To compare two strings in Java, we'd use the String class's compareTo() method. To understand how this method works, we need to understand more about strings.

Programming languages see characters in ASCII or Unicode, which are coding schemes that represent characters, numbers, and symbols as numeric values. When you compare strings, you're actually comparing these ASCII or Unicode values, not the letters. In addition, the comparison happens one character at a time. The first letters of each string are compared with each other, and if they are equal, the second letters are compared with each other, and if they are equal, the third letters are compared with each other, and so on. The comparisons stop when distinct found, or one string runs out of characters to compare.

String Comparison

So if we want to compare "hello" and "world", the value of "h" is compared with the value of "w". The value of "h" is 104 and the value of "w" is 119 (check the chart in Appendix B of your text). 104 is less than 119, so therefore "w" is greater than "h".

What about these ones (note these are just expressions, NOT valid Java code):

1. "disk" < "disc"
2. "iron" < "iron man"
3. "mop" < "broomstick"
4. "mouse" < "Mouse"
5. "234" < "9"
6. "" < "x"

In #1, we compare the d's and find that they're the same. We then move on to the i's and find that they're the same, also. Then we compare the s's and find that they're equal. We then come to the the "k" and the "c". The value of "c" is 99 and the value of "k" is 107, so therefore "disk" is greater than "disc". In fact, an easy way to figure this out without checking your chart is knowing that the further along a letter is in the alphabet, the greater it's value! This means that "a" is less than "m" and "m" is less than "z"!

In #2, we compare the first 4 letters i, r, o, and n, before we run out of characters in the first string. When this happens, the computer considers the next character in the first string as a null-string (""). A null-string is like a 0, except for strings. Therefore, the "" is less than any other string. This means the program will compare the first string's "" with the second string's " " (space). The space is an actual character and has a value of 32, so the string "iron" is less than "iron man".

Don't let the number of letters confuse you. "iron" had fewer letters, but it was less than "iron man" because of the value of the characters. In #3, we truly see how this works. When the comparison begins, the first letters "m" and "b" are compared. Right away, "b" is less than "m" because "b" is earlier in the alphabet than "m" (and the value of b is 98, while the value of m is 109). Therefore, the string "mop" is greater than "broomstick"!

The fourth example compares "mouse" to "Mouse". These two strings are not equal, as the first string's first character has a lower-case "m" and the second string starts with an upper-case "M". The value of the lower-case "m" is 109 and the value of the upper-case "M" is 77. Since "m" is greater than "M", "mouse" is greater than "Mouse". In fact, you can simply remember that upper-case letters are always less than lower-case letters.

The fifth example compares the string "234" to the string "9". Remember that these are strings, not numbers! If you check your chart, you'll see that "2" has a value of 50 and 9 has a value of 57. Therefore, the string "9" is greater than the string "234". Numbers are sorted differently than strings, which is why sometimes when you files with number-names sorted, they appear in a strange order, such as 1.txt, 11.txt, 14.txt, 2.txt, 21.txt, etc. In addition, string digits are always less than upper-case letter characters.

The last example compares a null-string to the string "x". Remember that a null string is a true "nothing" value for strings, so this statement must be false: the value "x" has to be greater than no value at all.

When you compare two strings using the compareTo() method, you invoke the method on one string object and pass in a second string object as an argument (just as you would with the equals() method):

```
"hello".compareTo("world");   // yes, you can invoke on a literal
```

The compareTo() method will go through the comparison process we discussed earlier. It will find that the "h" has a value of 104 and the "w" has a value of 119. Try this statement and see what the output is:

```
System.out.println("hello".compareTo("world"));
```

You will see the value -15 printed. Where do you think this value comes from?

104, the value of "h" minus 119, the value of "w", is -15! The compareTo() method takes the values of the first two distinct characters and subtracts the second one (parameter string object) from the first one.

We determined earlier that the string "iron" is less than "iron man". The compareTo() method will handle this example differently. When it sees that the strings are the same, except that one has more characters than the other, it measures the length of the string instead of the difference between the characters. In this case, s1.compareTo(s2) will return s1.length() - s2.length().

Therefore, we can conclude the following about the compareTo() method:

s1.compareTo(s2)	
For distinct characters:	
If...	**compareTo() returns...**
s1 greater than s2	a positive integer
s1 equal to s2	the integer value 0
s1 less than s2	a negative integer
For indistinct strings of different lengths:	
If...	**compareTo() returns...**
s1.length() < s2.length()	a negative integer
s1.length() > s2.length()	a positive integer

Exercise - String Comparison