Chapter 4: String Class Documentation

String Class Documentation

Java's String class is large (3,000 lines) but not terribly complex. It has some quirks that bear explaining, and we'll look at a few of those.

We're not going to delve into everything that the class has and does. That would take more time and space than we have left in this course! But we'll look at as much as we can in this chapter, and you can ask me any questions you have when we're done.

Like the Math class in the last chapter, String is part of the java.lang package. So if you still have that package open in the lower-left panel of your browser, you can scroll down a little and find the String class. If you've left that page, open the API again, and scroll down in the lower-left panel until you find String. Click it, and its documentation will come up in the main panel on the right.

Once you're there, you'll find a general description of the class and a few example lines that use it. I'm not going to discuss the whole description, but there are two points I'd like to hit. The first is a common misconception about String objects, and the second has to do with how String represents characters.

String Objects Are Immutable

The second paragraph of the description starts out like this: "Strings are constant; their values cannot be changed after they are created." In other words, String objects are *immutable*. Most new Java programmers think they can change the contents of a string (I certainly thought so), but Java doesn't allow that. Let me show you what I mean. Here are a couple of lines of code:

```
String s = "a string";
   // . . . further down in my code . . .
s = "another string";
```

The first time I saw this code, I thought it was changing the value of a String object named s. But that's not the way String objects work. For one thing, managing memory would become difficult since the new value is longer than the old value. How much memory would a String object need to allow for changes? They can be hundreds, even thousands, of characters long. You can see what a nightmare that would be.

What those lines of code do is replace one String object with another. The first line creates a variable named s with type String, and the variable refers to a String object containing "a string". The second part creates a new String object containing "another string" and sets s to refer to it. So Java discards the first object and replaces it with the second. The first object was unchangeable, but it was replaceable.

String Objects Can Store Thousands of Characters

Scroll down further in the description, and you'll find out that String objects store characters using Unicode's UTF-16 encoding scheme, which uses 16 bits, or two bytes, to store each character. That takes up a lot of memory for large strings, but memory is cheap these days, and flexibility is more important.

Older encoding schemes like ASCII and EBCDIC used one byte per character, which allows for only 256 different characters. That sounds like a lot, but with uppercase and lowercase characters, special characters, digits, and so on, we'd run out pretty quickly.

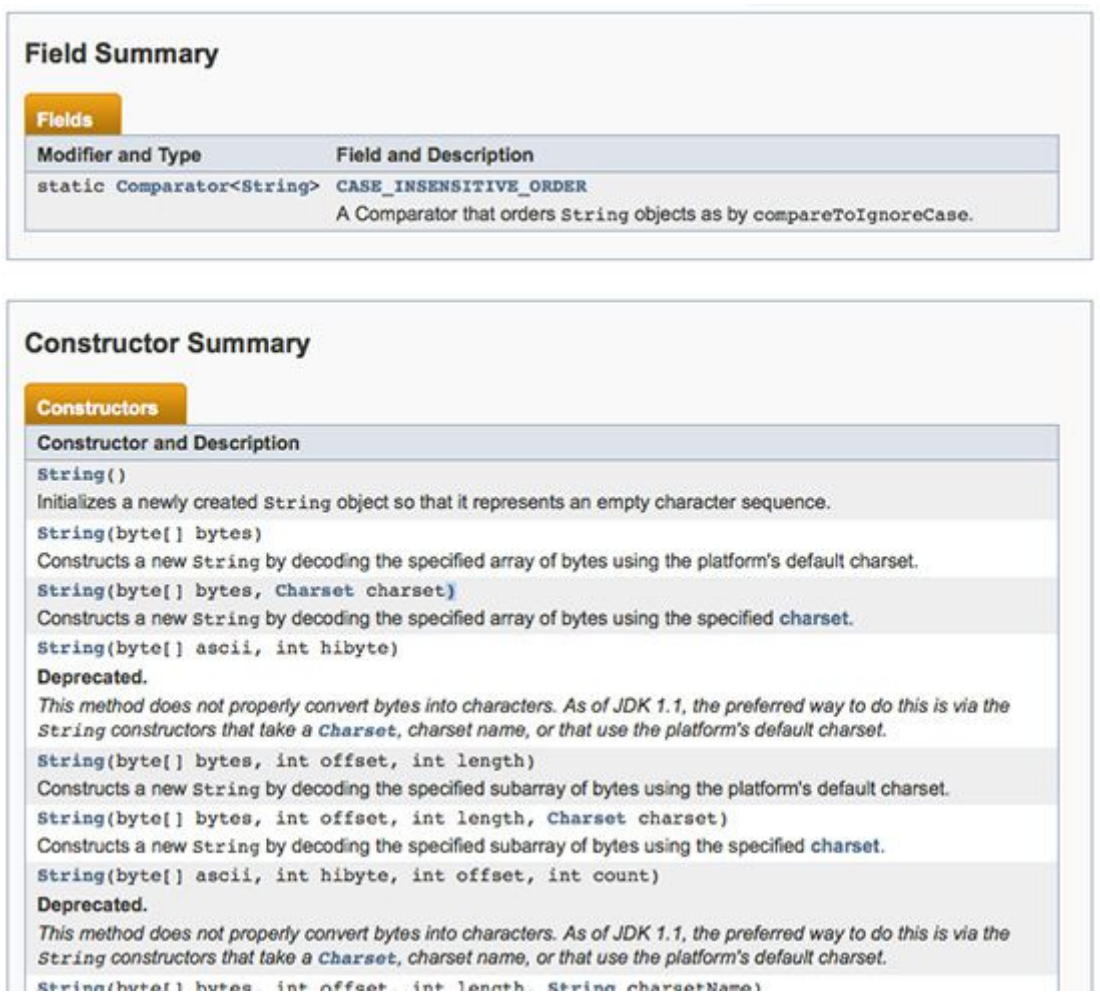
Unicode allows for thousands of characters. So Java strings can contain Cyrillic, Hebrew, Arabic, Greek, and Korean characters, to name a few. You can set values for unusual characters using a "\u" prefix like this:

String s = "\u03A3";

This statement puts the Greek uppercase letter sigma (Σ) into a String object.

Exploring String's Field Summary

If you're looking at the String documentation, scroll down past the general description until you see the words *Field Summary*. It should look something like this:



Field Summary

Fields

Modifier and Type	Field and Description
`static Comparator<String>`	`CASE_INSENSITIVE_ORDER` A Comparator that orders `String` objects as by `compareToIgnoreCase`.

Constructor Summary

Constructors

Constructor and Description
`String()` Initializes a newly created `String` object so that it represents an empty character sequence.
`String(byte[] bytes)` Constructs a new `String` by decoding the specified array of bytes using the platform's default charset.
`String(byte[] bytes, Charset charset)` Constructs a new `String` by decoding the specified array of bytes using the specified charset.
`String(byte[] ascii, int hibyte)` **Deprecated.** *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a `Charset`, charset name, or that use the platform's default charset.*
`String(byte[] bytes, int offset, int length)` Constructs a new `String` by decoding the specified subarray of bytes using the platform's default charset.
`String(byte[] bytes, int offset, int length, Charset charset)` Constructs a new `String` by decoding the specified subarray of bytes using the specified charset.
`String(byte[] ascii, int hibyte, int offset, int count)` **Deprecated.** *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a `Charset`, charset name, or that use the platform's default charset.*
`String(byte[] bytes, int offset, int length, String charsetName)`

java.lang.String field summary

The only public field that String has is a *Comparator object*. I don't have the space to get into how to use Comparators, but they allow alternate sort sequences when you're sorting lists using Java's Collections.sort() method. Specifically, this Comparator allows you to ignore case when sorting.

By default, Unicode sorts uppercase letters before lowercase ones. So "XYZ" will sort before "abc". The CASE_INSENSITIVE_ORDER comparator lets you sort them the way we'd expect: "abc" before "XYZ". If you'd like more information on sorting with comparators, visit the Supplementary Material.

Following the field summary and before the method summary, is String's constructor summary. If a class has public constructors—and most do—you'll find a list of them here. String has 15, each with a different parameter list. Why so many? People may need to create strings in quite a few ways, so the authors of the class have made it as easy as possible. We can create a string from character arrays, from integer arrays, from byte arrays, and from other strings, to name a few.

The easiest way to create a string in a program, though, is to use a simple assignment of a string literal, like this:

```
String s = "my string";
```

Understanding String's Method Summary

Let's look just a little further at some of the methods of the String class and at their documentation, and then we'll wrap it up.

Scroll down a bit more in the String class, and you'll see String's method summary:

Method Summary

Methods

Modifier and Type	Method and Description
char	**charAt**(int index) Returns the char value at the specified index.
int	**codePointAt**(int index) Returns the character (Unicode code point) at the specified index.
int	**codePointBefore**(int index) Returns the character (Unicode code point) before the specified index.
int	**codePointCount**(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this String.
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.
boolean	**contains**(CharSequence s) Returns true if and only if this string contains the specified sequence of char values.
boolean	**contentEquals**(CharSequence cs) Compares this string to the specified CharSequence.
boolean	**contentEquals**(StringBuffer sb) Compares this string to the specified StringBuffer.
static String	**copyValueOf**(char[] data) Returns a String that represents the character sequence in the array specified.
static String	**copyValueOf**(char[] data, int offset, int count) Returns a String that represents the character sequence in the array specified.
boolean	**endsWith**(String suffix) Tests if this string ends with the specified suffix.
boolean	**equals**(Object anObject) Compares this string to the specified object.

java.lang.String method summary

Just as there were in Math, there are quite a few methods in the String class too . . . about 70 of them. I'm going to touch on a couple of them, and then I'll let you explore the rest.

The most commonly used String methods are probably equals(), which compares strings for exact matches, and equalsIgnoreCase(), which lets you do comparisons while ignoring whether characters are uppercase or lowercase. Just so we're all on the same page, here are the details about those two methods:

equals

`public boolean equals(Object anObject)`

Compares this string to the specified object. The result is `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as this object.

Overrides:

`equals` in class `Object`

Parameters:

`anObject` - The object to compare this `String` against

Returns:

`true` if the given object represents a `String` equivalent to this string, `false` otherwise

See Also:

`compareTo(String)`, `equalsIgnoreCase(String)`

equalsIgnoreCase

`public boolean equalsIgnoreCase(String anotherString)`

Compares this `String` to another `String`, ignoring case considerations. Two strings are considered equal ignoring case if they are of the same length and corresponding characters in the two strings are equal ignoring case.

Two characters `c1` and `c2` are considered the same ignoring case if at least one of the following is true:

- The two characters are the same (as compared by the == operator)
- Applying the method `Character.toUpperCase(char)` to each character produces the same result
- Applying the method `Character.toLowerCase(char)` to each character produces the same result

Parameters:

`anotherString` - The `String` to compare this `String` against

Returns:

`true` if the argument is not `null` and it represents an equivalent `String` ignoring case; `false` otherwise

See Also:

`equals(Object)`

The java.lang.String equals() and equalsIgnoreCase() method details

These descriptions provide all the information we need to use the methods. They tell us the parameters they expect in our calls, information about what they do, the type of data to expect back, and how Java determines that value. So based on these descriptions, here are a couple of method calls and the values they return:

```
"abc".equals("ABC")                 // returns false
"def".equalsIgnoreCase("DeF")       // returns true
```

Here are the String methods I use most:

- compareTo() tells whether one string comes before another. It returns a negative value if the first string sorts before the second, a positive value if the second sorts before the first, and zero if they're the same. Example: "abc".compareTo("DEF") returns a positive number.

- compareToIgnoreCase() does the same as compareTo() but ignores whether each character is uppercase or lowercase. Example: "abc".compareToIgnoreCase("DEF") returns a negative number.

- contains() tells whether one string contains another. Example: "abc".contains("b") returns true.

- indexOf() searches for and returns the location of one string within another. It returns a negative value if the string it's looking for isn't there. Example: "abcdefg".indexOf("d") returns 3. (The first character is location 0 in Java.)

- length() returns the number of characters in the string. Example: "abcdefg".length() returns 7.

- toUpperCase() converts all lowercase characters in a string to uppercase. Example: "aBcDeFg".toUpperCase() returns "ABCDEFG".

- toLowerCase() converts all uppercase characters in a string to lowercase. Example: "aBcDeFg".toLowerCase() returns "abcdefg".

- substring() extracts part of a string based on index values. Examples: "abcdefg".substring(1,4) returns "bcd" and "abcdefg".substring(4) returns "efg".

If any of those examples don't make sense to you, look up the method details in the API, and see if you can figure out why they return what they return. You are welcome to share your questions with other students in the Discussion Area.