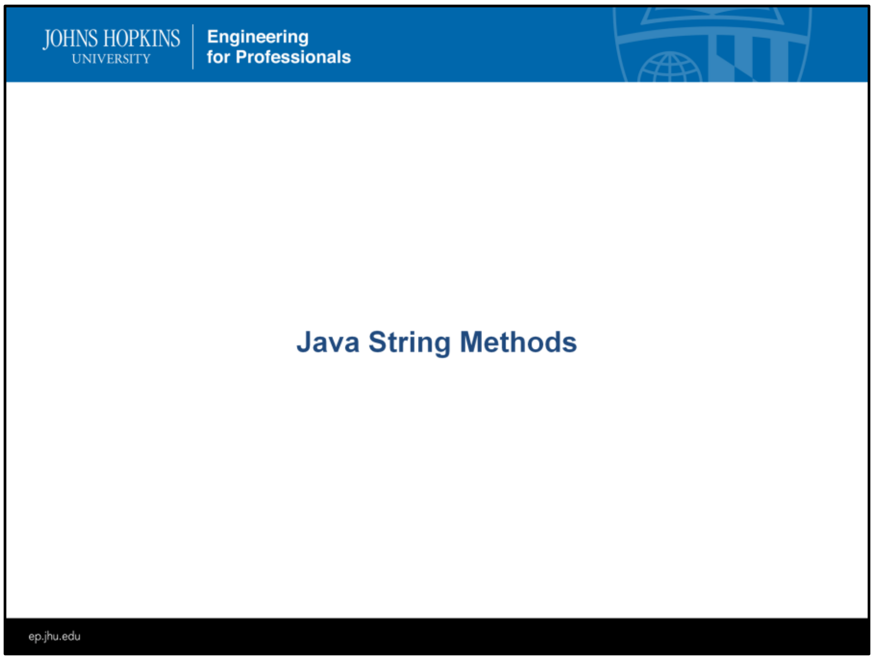
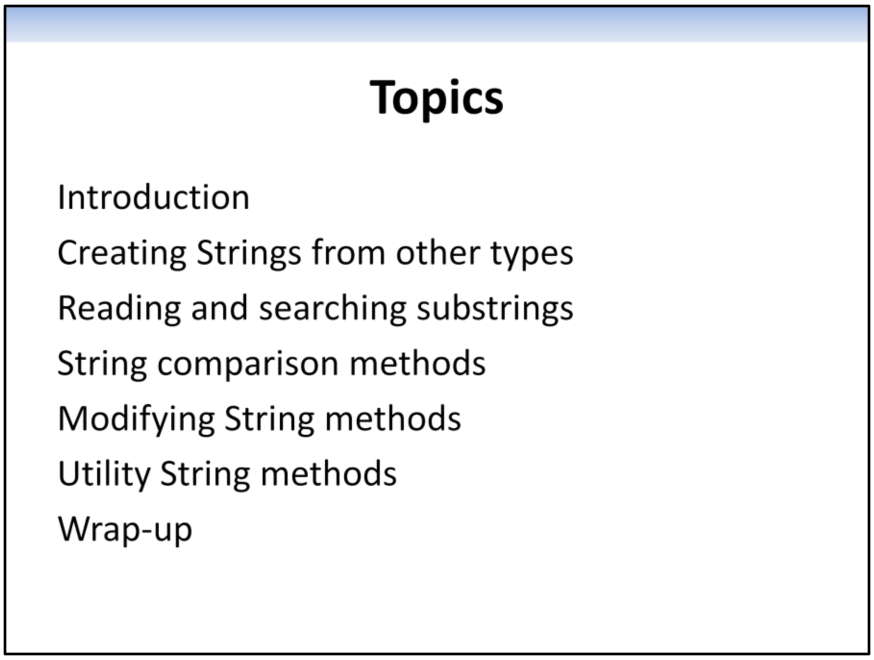


JOHNS HOPKINS UNIVERSITY | Engineering for Professionals

Java String Methods

ep.jhu.edu

This presentation discusses some of the Java methods associated with Strings. There are many more String methods than discussed in this presentation but the ones discussed should be enough to get a feel of the usefulness of Strings in your Java programs.



Topics

Introduction

Creating Strings from other types

Reading and searching substrings

String comparison methods

Modifying String methods

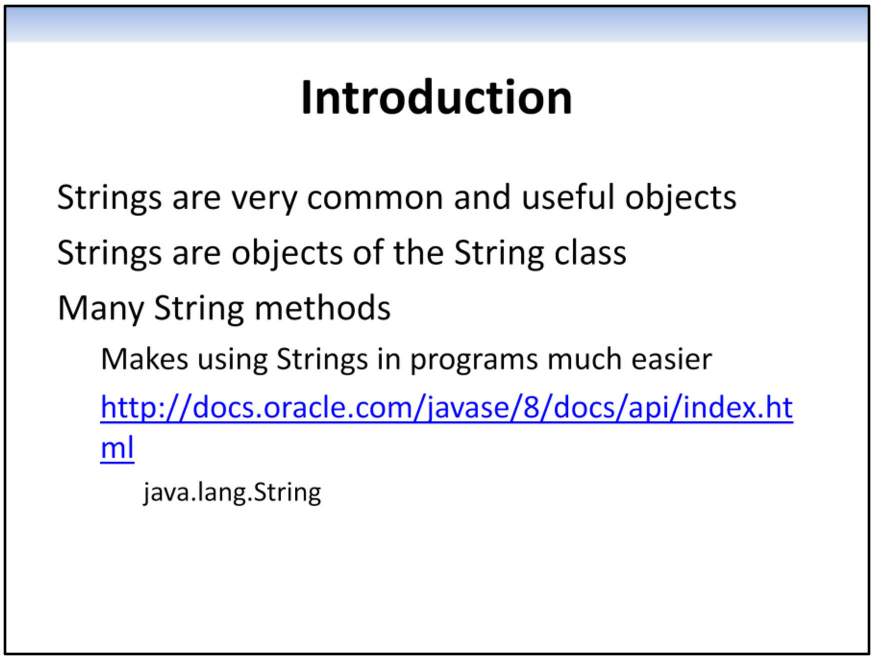
Utility String methods

Wrap-up

The main topics of this presentation are:

- Introduction to String methods
- Methods to convert other Java types to Strings
- Methods to read and searching for substrings
- Methods used to compare strings
- Methods which modify strings
- Some utility String methods
- A wrap-up summarizing the presentation

Quite a bit so lets get to it!



Introduction

Strings are very common and useful objects

Strings are objects of the String class

Many String methods

- Makes using Strings in programs much easier
- http://docs.oracle.com/javase/8/docs/api/index.html
  - java.lang.String

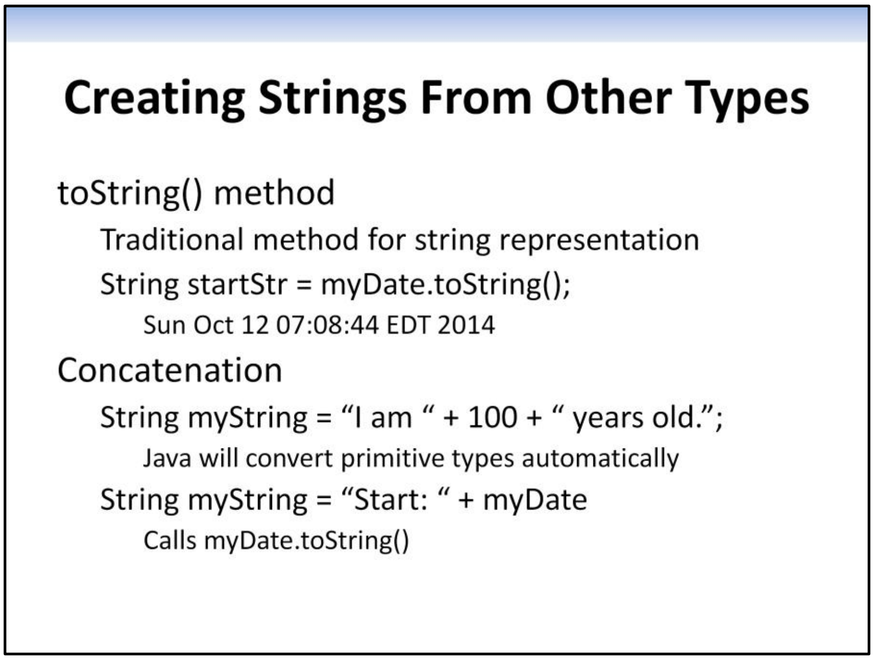
Strings are very common and useful objects in the Java programming language.

Strings are objects of the String class. Java classes are discussed in detail in separate presentations and we will leave many of the class aspects of Strings to those presentations. Here we will look at some of String methods which help make Strings so useful in Java programs.

There are many String methods not discussed in this presentation. For detailed information on all Java Standard Edition class methods, you can point your browser to the link here.

The String class is part of the java.lang package which is included in all Java programs by default.

Lets start by looking at methods which create Strings from other Java types.



Creating Strings From Other Types

toString() method

    Traditional method for string representation

    String startStr = myDate.toString();

        Sun Oct 12 07:08:44 EDT 2014

Concatenation

    String myString = “I am “ + 100 + “ years old.”;

        Java will convert primitive types automatically

    String myString = “Start: “ + myDate

        Calls myDate.toString()

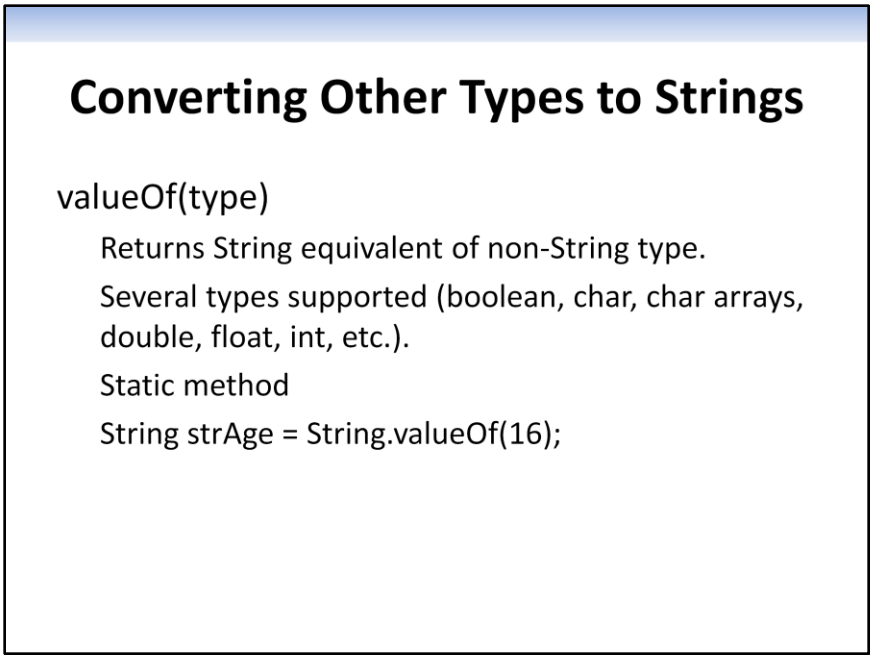
toString() method s is the traditional way to provide string representations of objects. Without getting into the details of objects, all Java objects have a toString() method although many maybe of limited usefulness to most Java programs.

An example of a very useful toString() method is for the Date object. This method provides a useful representation of a date as show here.

Concatenation is discussed in a different presentation but relevant here since it converts non-String types to strings on-the-fly as a part of building a new String from multiple parts. String concatenation will convert primitive types to their String representation an for objects call their toString() methods. Note at least one operand of the “+” operator must already be a String.

Here are a couple of examples of String concatenation. The first one demonstrates the conversion of an int variable to its String representation and joining it to the rest of the String being built.

The second example demonstrates the Date.toString() method being called for the myDate Date instance as part of building a new String.



Converting Other Types to Strings

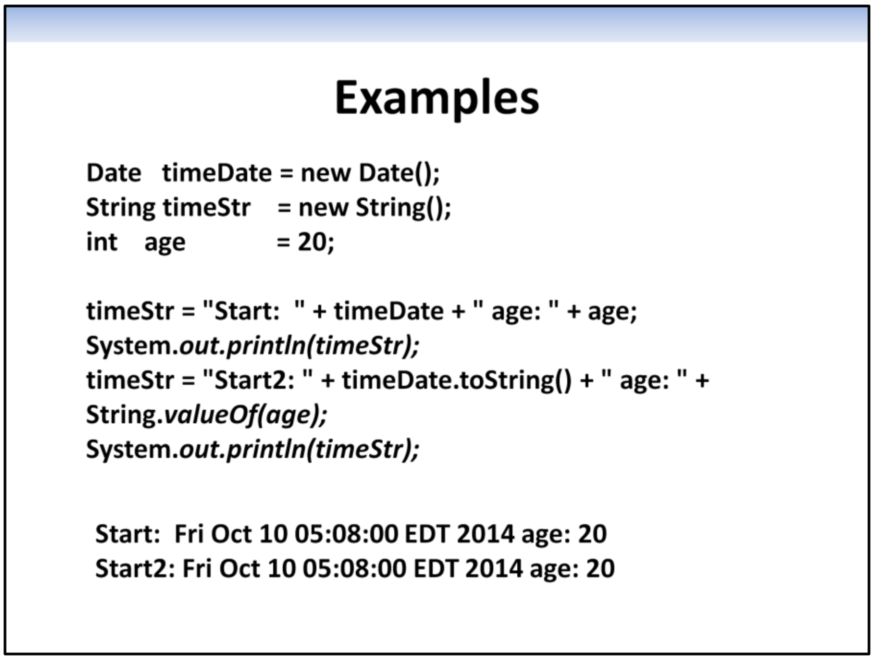
valueOf(type)

- Returns String equivalent of non-String type.
- Several types supported (boolean, char, char arrays, double, float, int, etc.).
- Static method
- String strAge = String.valueOf(16);

The String.valueOf() method is another way to convert a non-string type to a String. This is what actually gets called during String concatenation although for most objects the results of the object's toString() method is returned. For primitive types, valueOf() returns a String representation.

ValueOf() is a static method. A detailed discussion on static methods is in a separate presentation but for now this means you don't need a String object to use this method.

The example here will initialize the   String object **strAge** with the string representation of the integer 16.



Examples

```
Date   timeDate = new Date();
String timeStr    = new String();
int    age          = 20;

timeStr = "Start:  " + timeDate + " age: " + age;
System.out.println(timeStr);
timeStr = "Start2: " + timeDate.toString() + " age: " +
String.valueOf(age);
System.out.println(timeStr);
```

```
Start:  Fri Oct 10 05:08:00 EDT 2014 age: 20
Start2: Fri Oct 10 05:08:00 EDT 2014 age: 20
```

Here are some examples of converting non-String types to Strings.

Here we have defined a Date object, a String object, and an integer.

At this point, the program is concatenating the String constant “Start” with the results of String representation of the Date object **timeDate** via the implied String.valueOF() method . Then the String representation of the value contained in the integer **age** variable is tacked on to the end. The reference to this just-built String is assigned to the **timeString** variable.

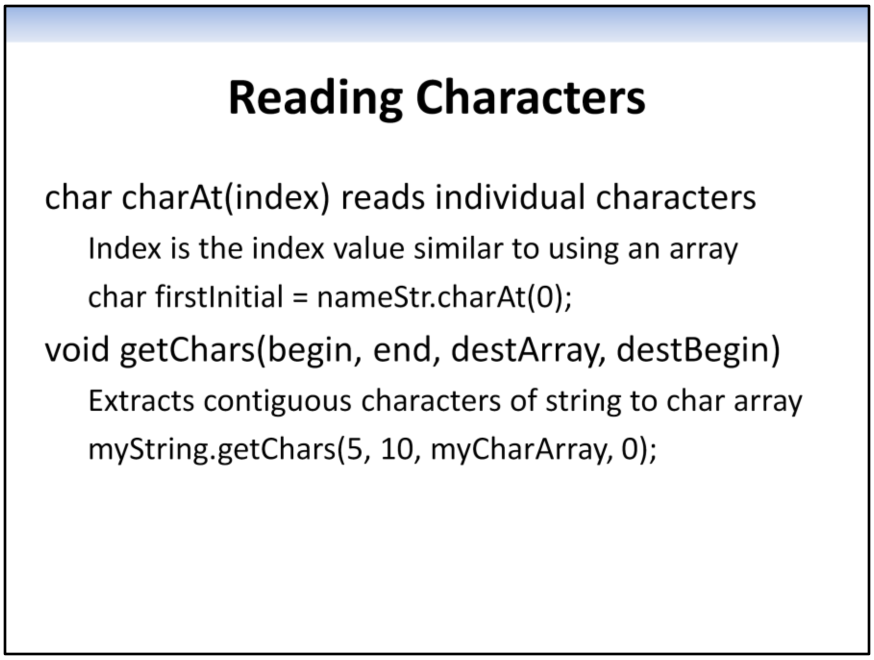
Here we display the results of our concatenation.

Here we recreate the same String as above but this time explicitly calling the toString() method of the **timeDate** Date object and using the String.valueOf() method to get the String representation of the value of the age integer variable.

Again we display the results.

Here is the output of one run of this program. Could you think of a possibility of the concatenation *not* being the same?

Lets move on to using String methods to read individual characters of Strings.



Reading Characters

char charAt(index) reads individual characters

- Index is the index value similar to using an array
- char firstInitial = nameStr.charAt(0);

void getChars(begin, end, destArray, destBegin)

- Extracts contiguous characters of string to char array
- myString.getChars(5, 10, myCharArray, 0);

You can read individual characters of Strings with a variety of methods of which two are discussed here.

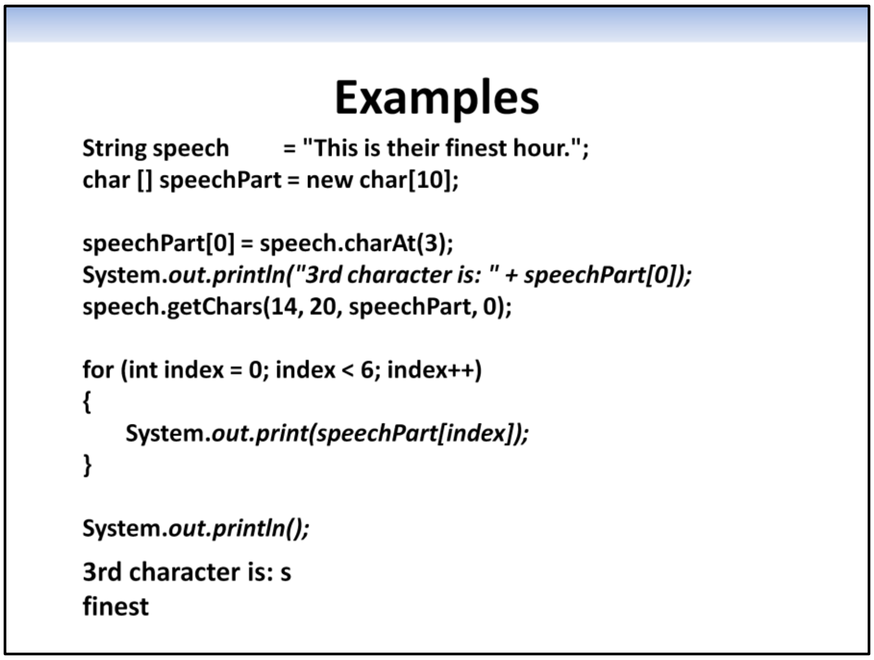
The charAt() String method returns an individual character of the string based on the supplied integer index value. This is similar to using an index to read a single character from a character array.

This example here assigns the first character (index zero) of the String name to the character variable firstInitial.

The getChars() String method copies a contiguous set of the String's characters to a character array. The parameters begin and end are the beginning and ending +1 index of the characters of the String to be copied. The destArray parameter is the name of the receiving character array. The destBegin parameter is the index value of the destination character array where the copied characters will be inserted.

The example here will copy five characters starting at index 5 of the String myString and assign the characters to the character array myCharArray starting at index zero (the beginning) of the array.

Note: using index values beyond the range of the String or array will result in an error (exception).



Examples

```
String speech        = "This is their finest hour.";
char [] speechPart = new char[10];

speechPart[0] = speech.charAt(3);
System.out.println("3rd character is: " + speechPart[0]);
speech.getChars(14, 20, speechPart, 0);

for (int index = 0; index < 6; index++)
{
    System.out.print(speechPart[index]);
}

System.out.println();
```

```
3rd character is: s
finest
```

Here is an example of using the Strings methods just discussed.

Here we define an initialized String variable and a character array.

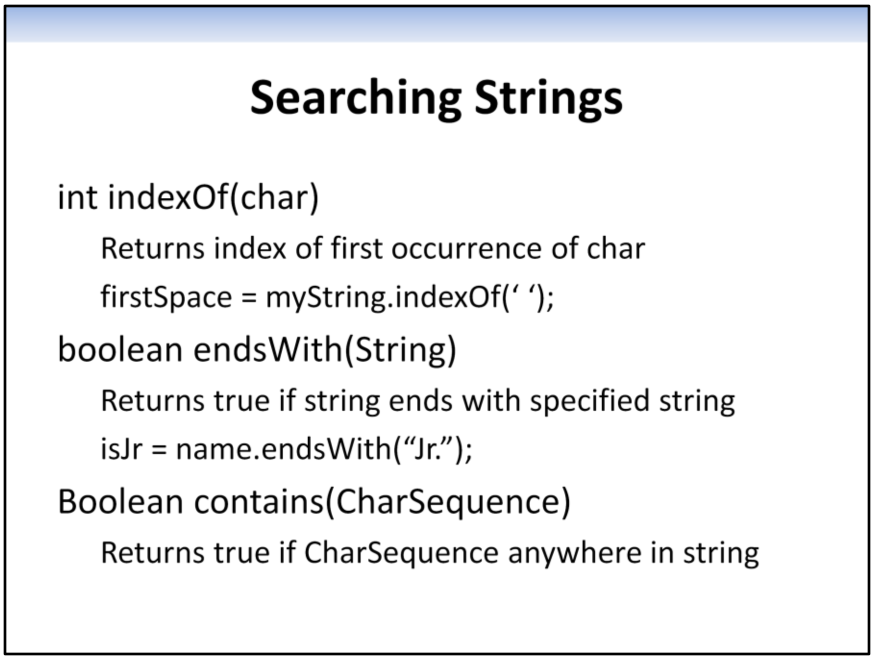
This line copies the character at index three of the speech String and assigning it to the first character (index zero) of the speechPart character array.

We then display the results of the above assignment.

Here we use the String method getChars() to copy six characters from the String speech starting at index 14 and assigning them to the first six characters of the speechPart character array.

Here we use a loop to display the first six characters of the speechPart character array.

Here is the output of this program. You may want to pause the presentation at this point to study how the results of this program were obtained.



Searching Strings

int indexOf(char)
- Returns index of first occurrence of char
- firstSpace = myString.indexOf(‘ ‘);

boolean endsWith(String)
- Returns true if string ends with specified string
- isJr = name.endsWith(“Jr.”);

Boolean contains(CharSequence)
- Returns true if CharSequence anywhere in string

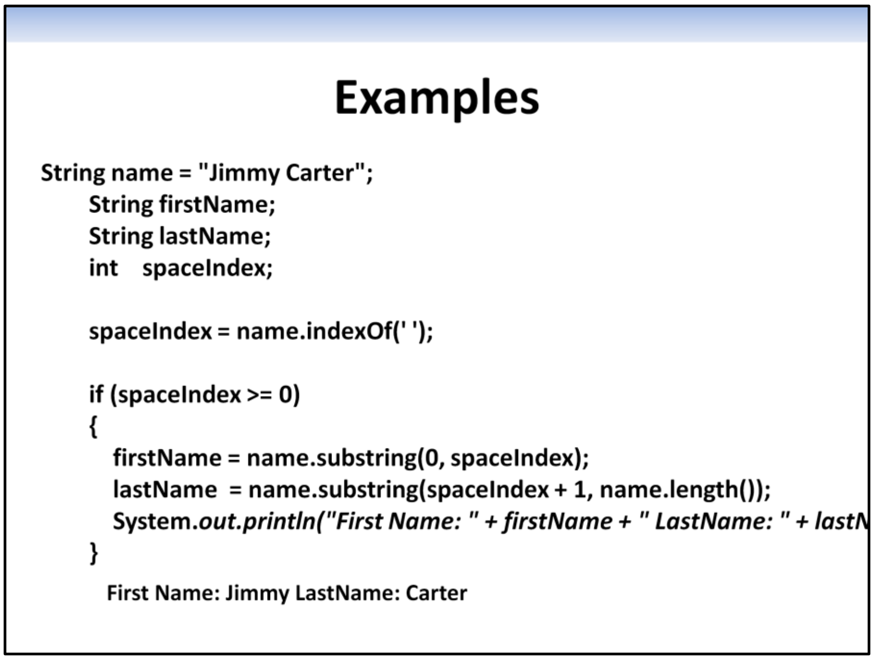
Strings can contain multiple pieces of information. It can be desirable to have the ability to find out if and where a particular set of characters are in a string. Discussed here are three String methods to help in this endeavor.

The first example is the indexOf() method. This method searches for the first occurance of the supplied character in the String and returns the index of the found character. If the character is not found, a -1 is returned.

The second example is the endsWith() method. This method returns true if the String ends with the supplied String. If the String does not end with the supplied string false is returned.

The third example is the contains() method which searches for the supplied character sequence anywhere in the String. If the target String does contain the character sequence, a true is returned. If it is not found, false is returned.

Lets look at an example which uses a couple of these methods.



Examples

```
String name = "Jimmy Carter";
       String firstName;
       String lastName;
       int    spaceIndex;

       spaceIndex = name.indexOf(' ');

       if (spaceIndex >= 0)
       {
          firstName = name.substring(0, spaceIndex);
          lastName  = name.substring(spaceIndex + 1, name.length());
          System.out.println("First Name: " + firstName + " LastName: " + lastN
       }
         First Name: Jimmy LastName: Carter
```

In this example we define and initialize the string name and define two non-initialize Strings firstName and LastName. We also create a non-initialized integer named spaceIndex.

Here we search for the first space in the String name. The goal is to find the end of the first name of the name contained in the name String object.

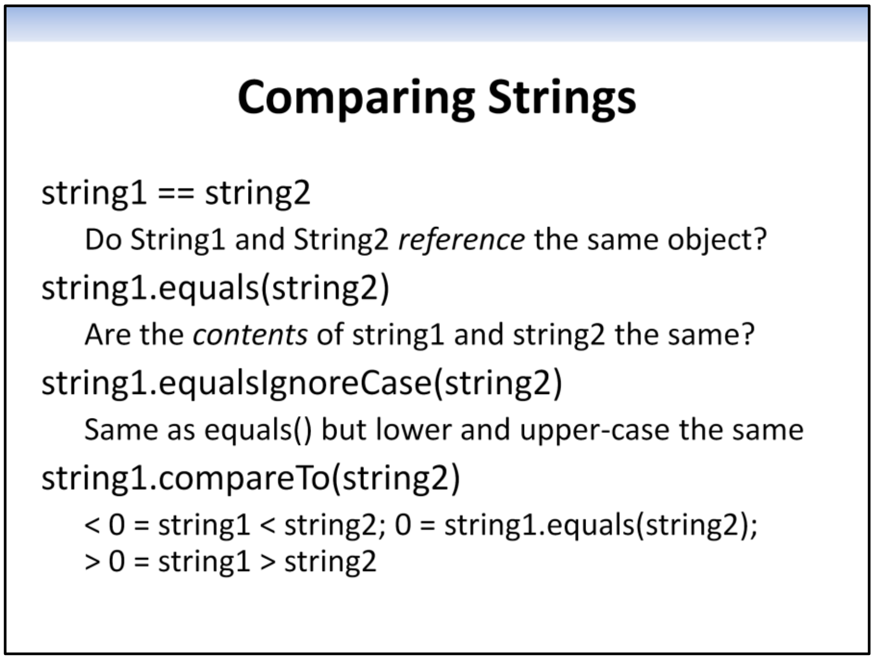
Next we check to see if the above method found a space in the name String. If so, we use the value of spaceIndex to copy the first name part of the name String and assign the firstName String object.

Then we use the spaceIndex value to calculate the beginning of the last name and assign the resulting sub-string to the lastName object.

Note the substring() method is discussed in more detail later in this presentation.

Here is the output of this program. Note the first and last names were effectively extracted from a single-string source.

Lets move on to look at methods which help when we need to compare Strings.



Comparing Strings

string1 == string2

Do String1 and String2 *reference* the same object?

string1.equals(string2)

Are the *contents* of string1 and string2 the same?

string1.equalsIgnoreCase(string2)

Same as equals() but lower and upper-case the same

string1.compareTo(string2)

< 0 = string1 < string2; 0 = string1.equals(string2);
> 0 = string1 > string2

Comparing Strings is a very common activity in Java programs. There are multiple String methods to help when comparing Strings are needed.

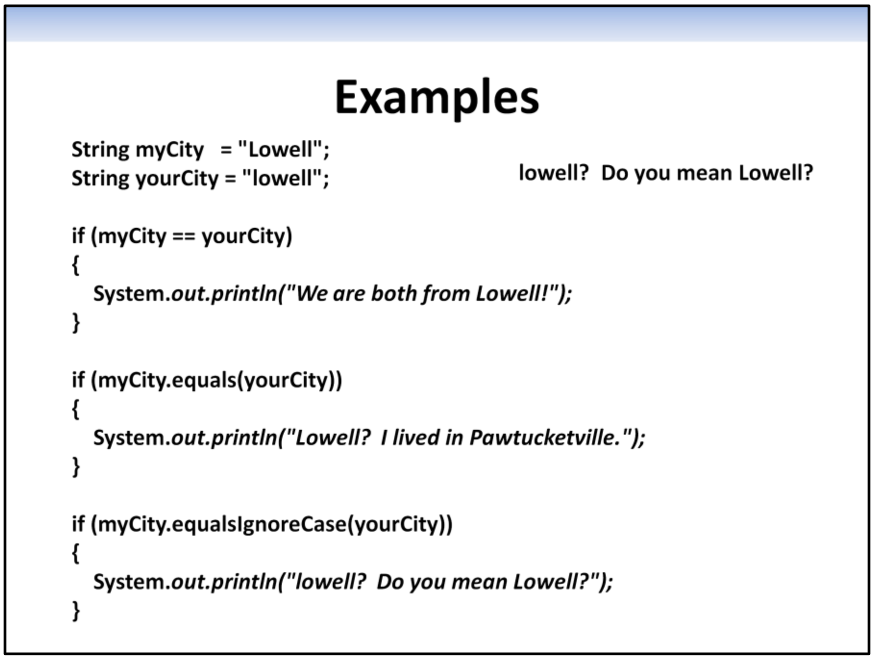
The first example here is a reminder using the == operator to compare strings evaluates of the two String operands refer to the same String object, not if the Strings contain the same values (although if the references are the same the values are the same).

The String equals() method compares the *values* of the two Strings involved. So the two Strings can refer to different String objects but if the values of the two String objects are the same a value of true will be returned from the method. This method is very commonly used since usually it is desired to compare the contents of Strings vs if the two Strings refer to the same objects.

The third example demonstrates the equalsIgnoreCase() method which works the same as the equals() method except it ignores any case-differences between the contents of the two strings.

The last method discussed is the compareTo() method. The purpose of this method is to provide a relative relationship between the two involved Strings. It does a lexically comparison between the two Strings and returns a value less than zero if the target string is less than the value of the String parameter, zero if the two Strings are equal, and a value greater than zero if the parameter string value is greater than the target string.

Lets see some examples.



Examples

```
String myCity   = "Lowell";
String yourCity = "lowell";                    lowell?  Do you mean Lowell?

if (myCity == yourCity)
{
   System.out.println("We are both from Lowell!");
}

if (myCity.equals(yourCity))
{
   System.out.println("Lowell?  I lived in Pawtucketville.");
}

if (myCity.equalsIgnoreCase(yourCity))
{
   System.out.println("lowell?  Do you mean Lowell?");
}
```

Here are some examples comparing Strings.

First we start off by defining and initializing a couple of Strings. Note the slight difference in the content of the initialization strings.

In this first comparison, we are checking to see if both **myCity** and **yourCity** refer to the same *object*. The contents are not examined. If the comparison returns true, a message is printed.

The second comparison uses the **equals()** method. This method does examine the contents of the two String objects, not what they are referring to. A message is printed if true.

The last comparison is using the **equalsIgnoreCase()** method. This method is similar to the **equals()** method except it considers upper-case and lower-case of the same letter equivalent. If the comparison returns true, a message is printed.

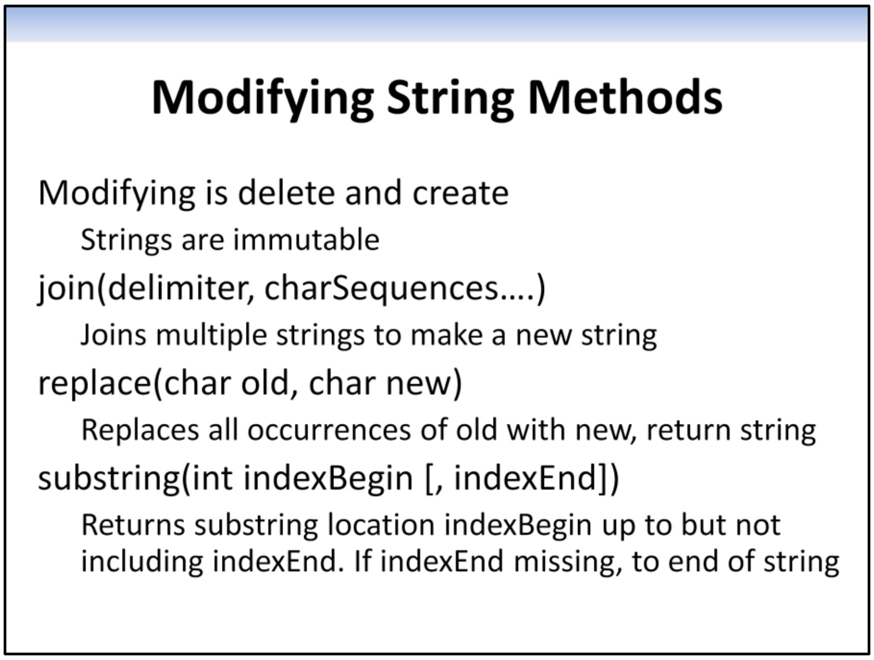
Here is the results after running the program. Note the first comparison returned false. This is because the two String objects do not refer to the same object since the initialization string constants are different requiring Java to create two different strings.

The second comparison returns false because the contents of the two Strings are not *exactly* the same. The leading lower-case and upper-case “L” are different.

The last comparison return true since, ignoring case, the String contents are the same.

Here is the program output. Note only the last compare returned a true result.

Lets look now at modifying Strings.



Modifying String Methods

Modifying is delete and create

Strings are immutable

join(delimiter, charSequences….)

Joins multiple strings to make a new string

replace(char old, char new)

Replaces all occurrences of old with new, return string

substring(int indexBegin [, indexEnd])

Returns substring location indexBegin up to but not including indexEnd. If indexEnd missing, to end of string

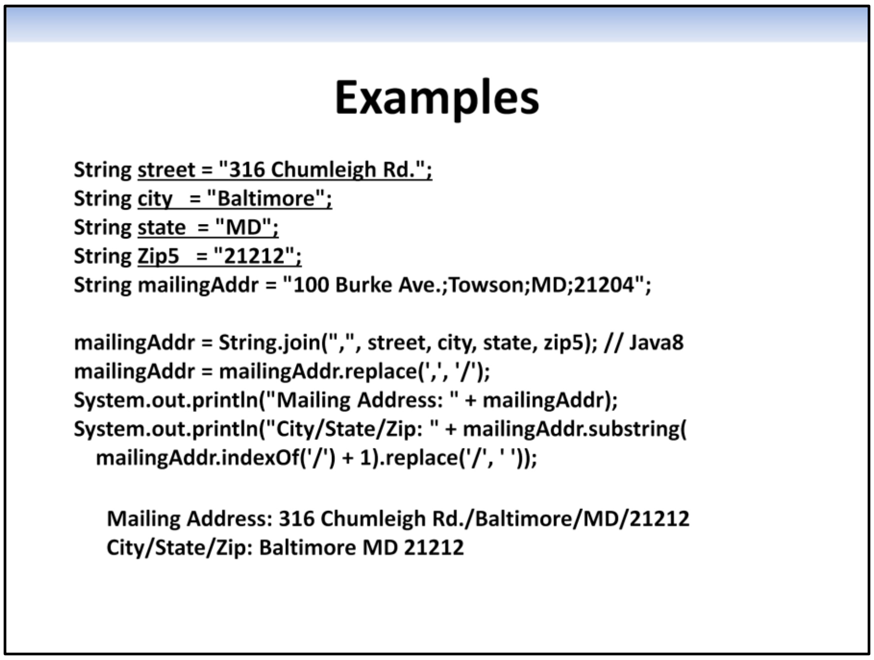
Remember Strings are immutable so they cannot be modified. If you specify Java to modify a String, the original string is deleted (unless there are other references to it) and a new String reflecting the modification s is created. Lets look at some methods which are involved in modifying strings.

The **join** method is new with Java 8. It concatenates all the parts specified in the second through last parameter but with the value of the first parameter placed between each part. The resulting String is returned.

The **replace** method will replace all occurrences of the value of the first parameter in the target String with the value of the second parameter. It returns a new string with these modifications.

The **substring()** method comes in two varieties as shown here. The first one will return the part of the target String which begins at the index of the parameter value to the end of the String. The second version includes a second parameter which instructs the method t o end the substring extracted at the index *before* the parameter value.

Lets take a look at some examples.



Examples

```
String street = "316 Chumleigh Rd.";
String city   = "Baltimore";
String state  = "MD";
String Zip5   = "21212";
String mailingAddr = "100 Burke Ave.;Towson;MD;21204";

mailingAddr = String.join(",", street, city, state, zip5); // Java8
mailingAddr = mailingAddr.replace(',', '/');
System.out.println("Mailing Address: " + mailingAddr);
System.out.println("City/State/Zip: " + mailingAddr.substring(
   mailingAddr.indexOf('/') + 1).replace('/', ' '));
```

```
    Mailing Address: 316 Chumleigh Rd./Baltimore/MD/21212
    City/State/Zip: Baltimore MD 21212
```

In this example we start off with defining a collection of initialized String objects which represent parts of an address.

The **join** method here is concatenating the street, city, state, and zip5 address parts and placing a comma between each part. The resulting String reference is assigned to the MailingAddr String.

The next statement takes the Mailing Addr String and replaces all commas with forward-slashes. The resulting String reference is assigned to the mailingAddr String. The previously referenced String is discarded.

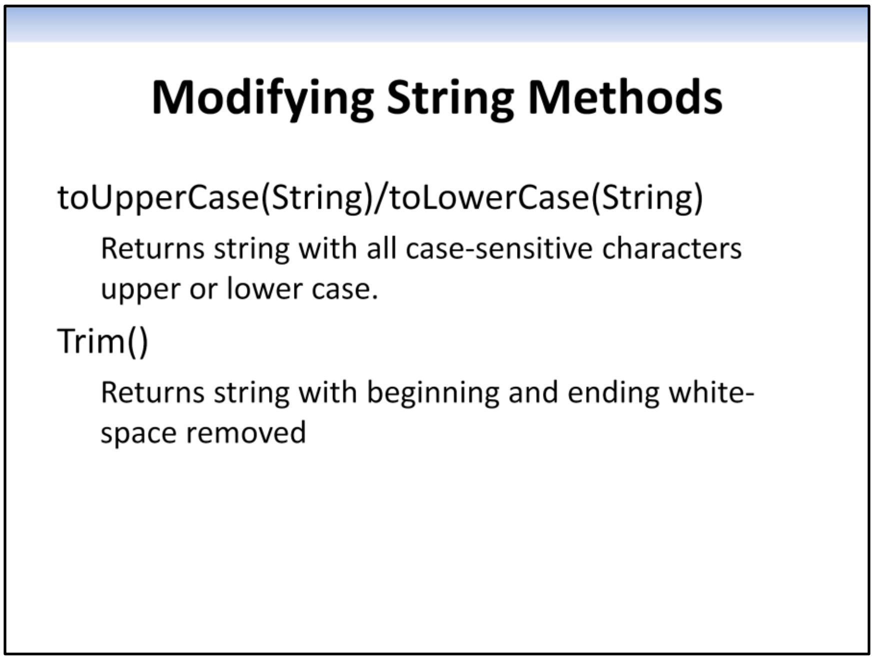
Next we display the mailingAddr string.

The last statement is a bit complex. After displaying a label, we take the mailingAddr String contents and extract a sub-string starting right after the first forward-slash, which is now the city-part of the address, until the end of the mailingAddr String. This returned sub-string is further modified by replacing the forward-slashes with spaces. The resulting temporary String is then displayed.

The output of the program is shown here.

Note this program results in several Strings being created and destroyed.

Lets look at a couple more modifying String methods before we leave the topic.



Modifying String Methods

toUpperCase(String)/toLowerCase(String)

Returns string with all case-sensitive characters upper or lower case.

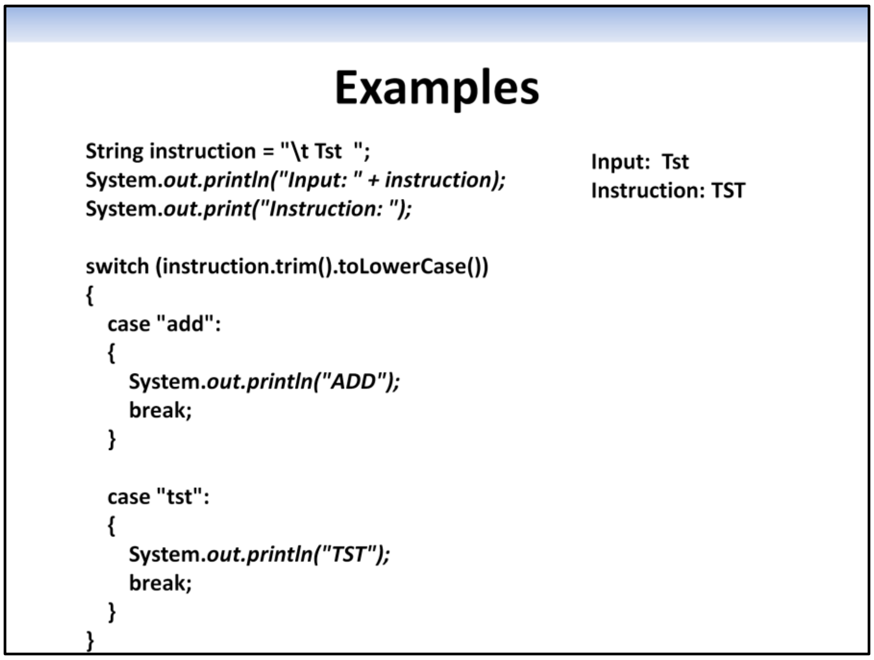
Trim()

Returns string with beginning and ending white-space removed

The **toUpperCase()** method modifies the target String by making all lower-case characters upper-case and returning the resulting String. The **toLowerCase()** method modifies the target String by making all upper-case characters lower-case an d returning the resulting String.

The **trim()** method removes “white space” from the beginning and end of the target String then returns the resulting String. White space is defined as any character who’s integer value is equal to or less than 0x20 (32) which is a space. This includes tabs, spaces, new-lines, etc. (generally characters you can’t see).

These routines can be convenient in parsing input values of a loose interface. Lets take a look at an example demonstrating this.



Examples

```
String instruction = "\t Tst  ";
System.out.println("Input: " + instruction);
System.out.print("Instruction: ");

switch (instruction.trim().toLowerCase())
{
   case "add":
   {
      System.out.println("ADD");
      break;
   }

   case "tst":
   {
      System.out.println("TST");
      break;
   }
}
```

```
Input:  Tst
Instruction: TST
```

For this example, lets say we are building a parser for an assembly language and one of the instructions is the TST instruction (which stands for TEST). Like most assembly languages, the keywords are case-insensitive and “white space” is parsed out.

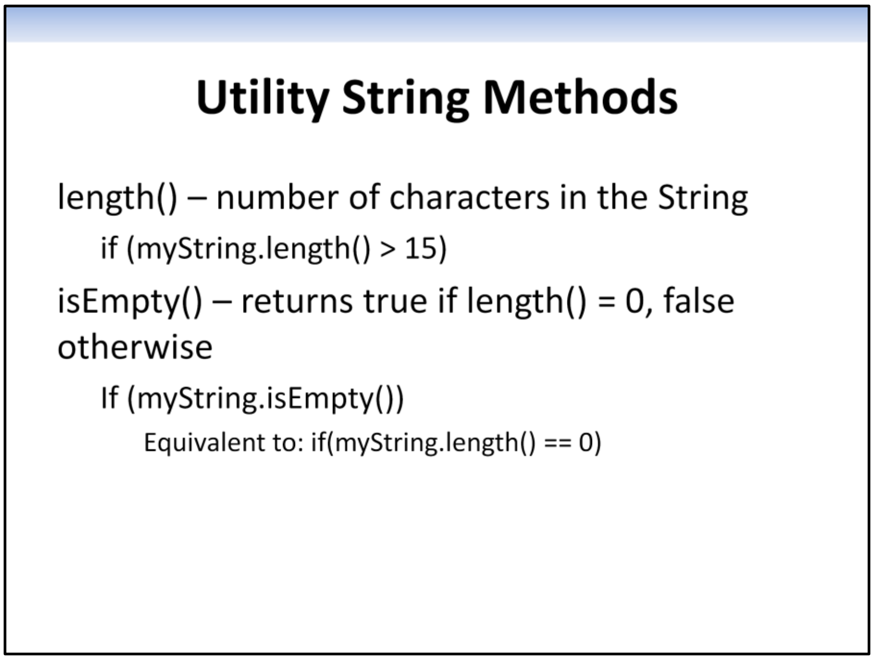
At the beginning we create and initialize the **instruction** String variable and display the just-initialized contents.

Next we start with the display of our parse results but there is more to come.

Next is a **switch** statement which call s the **toLowerCase()** method to force lower-case of the input text. We also call the **trim** method to remove any “white space” before or after the entered instruction text. Then in the comparison part of the **switch** statement we check for the string “tst” to see if we have a match. Since the input has been forced to lower case and our comparison String is also lower-case, a match with “tst” is found and the resulting println () executed.

Here is the output of this program. Note the input was mixed-case but by using the **toLowerCase()** method to force lower-case, the input was successfully parsed.

Lets look at some utility String functions to finish off this subject.



Utility String Methods

length() – number of characters in the String

    if (myString.length() > 15)

isEmpty() – returns true if length() = 0, false otherwise

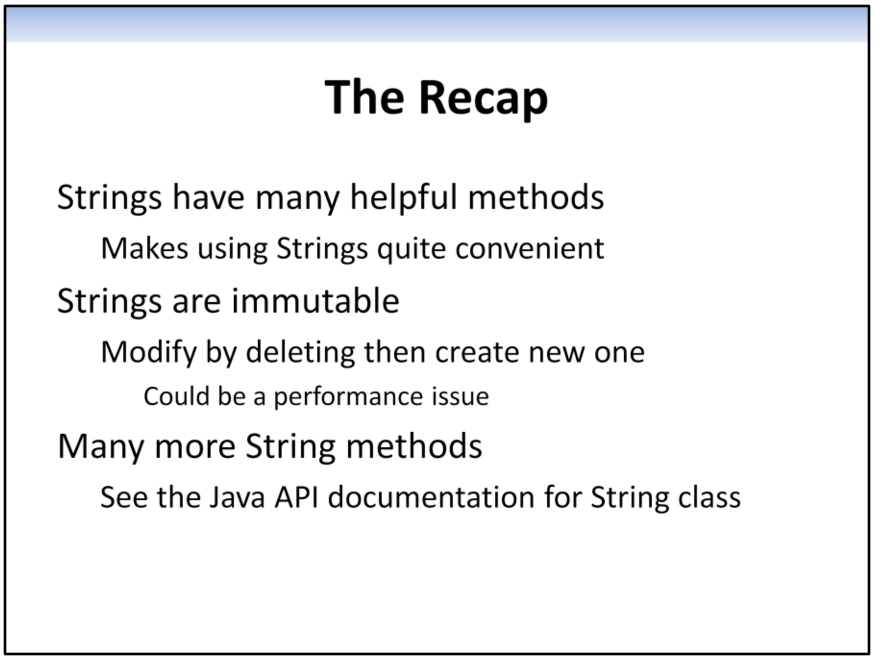
    If (myString.isEmpty())

        Equivalent to: if(myString.length() == 0)

Lets look at a couple of utility String methods, both dealing with the size of the contain character array.

The **length()** method returns the length of the target String. This is a very commonly used method. Previously we observed it being used to help determine the end-point of a desired sub-string. It is useful to ensure you do not try to process a String beyond it's end boundary. It important to remember it returns the number of characters not the last index. So if **length()** returns 12, the last valid index is 11.

The **isEmpty()** method returns true if the target String does not contain any characters and false otherwise.. It is a quick way to determine if a String is empty. It is a convenience method since you could use the **length()** method to perform the same function as shown here.



The Recap

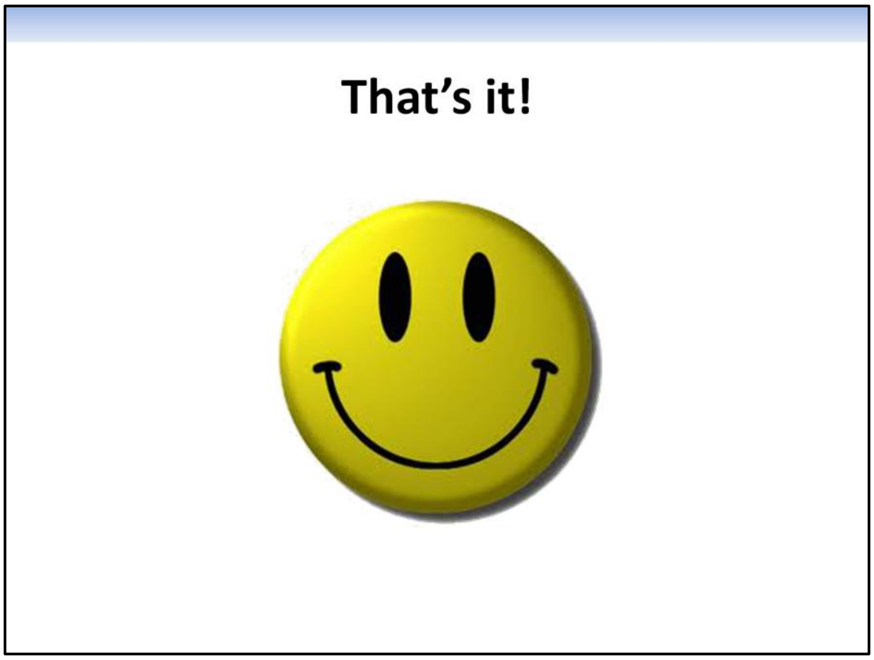
- Strings have many helpful methods
  - Makes using Strings quite convenient
- Strings are immutable
  - Modify by deleting then create new one
    - Could be a performance issue
- Many more String methods
  - See the Java API documentation for String class

OK here is the recap.

Strings have many useful methods which make them a flexible and easy to use type compared to character arrays.

Strings are immutable. Modifying Strings results in destroying the old one and creating a new one. This may have performance consequences depending on the frequency of String destruction and creation.

There are many more String methods beyond the ones discussed in this presentation. The Java API documentation on the String class provides more details.



That's it!

That's it!