Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Java:Strings/5aabd42026e8ba7398f58d1e

TUTORIAL

Java: Strings

Topics

Conceptually, Java strings are sequences of Unicode characters. For example, the string "Java" consists of the four Unicode characters J, a, v, and a. Java does not have a built-in string type. Instead, the standard Java library contains a predefined class called, naturally enough, String. Each quoted string is an instance of the String class. String literals are strings that are specified using double quotes. "This is a string" and "xyz" are examples of string literals. String literals are different than the literal values used with primitive types. When the javac compiler encounters a String literal, it converts it to a String constructor. For example, this

```
String str = "text";
```

is equivalent to this

```
String str = new String("text");
```

Because the compiler automatically supplies String constructors, you can use String literals everywhere that you can use objects of the String class.

The String class provides a very powerful set of methods for working with String objects. These methods enable you to access individual characters and substrings; test and compare strings; copy,

concatenate, and replace parts of strings; convert and create strings; and perform other useful string operations.

The most important String methods are the length() method, which returns an integer value identifying the length of a string; the charAt() method, which allows the individual characters of a string to be accessed; the substring() method, which allows substrings of a string to be accessed; and the valueOf() method, which enables primitive data types to be converted into strings.

In addition to these methods, the Object class provides a toString() method for converting other objects to String objects. This method is often overridden by subclasses to provide a more appropriate object-to-string conversion.

The String and StringBuffer classes are used to support operations on strings of characters. The String class supports constant (unchanging or immutable) strings, whereas the StringBuffer class supports growable, modifiable strings. String objects are more compact than StringBuffer objects, but StringBuffer objects are more flexible.

String Literals

String literals are strings that are specified using double quotes. "This is a string" and "xyz" are examples of string literals. String literals are different than the literal values used with primitive types. When the javac compiler encounters a String literal, it converts it to a String constructor. For example, this

```
String str = "text";
```

is equivalent to this

```
String str = new String("text");
```

Because the compiler automatically supplies String constructors, you can use String literals everywhere that you can use objects of the String class.

The + Operator and StringBuffer

Strings in Java are immutable i.e. they cannot change once defined. But java provides + operator to concatenate strings with each other, actually JVM uses StringBuffer concept behind it and each time you call + operator on string to concatenate it, it creates a new string and adjust the addresses. In the following example, the code will result in the string "ab" being assigned to the s object:

```
String s = "";
s = s + "a" + "b";
```

How can this be possible if Strings are constant? The answer lies in the fact that the Java compiler uses StringBuffer objects to accomplish the string manipulations. This code would be rendered as something similar to the following by the Java compiler:

```
String s = "";
s = new StringBuffer("").append("a").append("b").toString();
```

A new object of class StringBuffer is created with the "" argument. The StringBuffer append() method is used to append the strings "a" and "b" to the new object, and then the object is converted to an object of class String via the toString() method. The toString() method creates a new object of class String before it is assigned to the s variable. In this way, the s variable always refers to a constant (although new) String object.

String Constructors

The String class provides several constructors for the creation and initialization of String objects. These constructors enable strings to be created from other strings, string literals, arrays of characters, arrays of bytes, and StringBuffer objects. Browse through the API page for the String class to become familiar with these constructors.

String Access Methods

The String class provides a very powerful set of methods for working with String objects. These methods enable you to access individual characters and substrings; test and compare strings; copy,

concatenate, and replace parts of strings; convert and create strings; and perform other useful string operations.

The most important String methods are the length() method, which returns an integer value identifying the length of a string; the charAt() method, which allows the individual characters of a string to be accessed; the substring() method, which allows substrings of a string to be accessed; and the valueOf() method, which enables primitive data types to be converted into strings.

In addition to these methods, the Object class provides a toString() method for converting other objects to String objects. This method is often overridden by subclasses to provide a more appropriate object-to-string conversion.

Character and Substring Methods

Several String methods enable you to access individual characters and substrings of a string. These include charAt(), getBytes(), getChars(), indexOf(), lastIndexOf(), and substring(). Whenever you need to perform string manipulations, be sure to check the API documentation to make sure that you don't overlook an easy-to-use, predefined String method.

String Comparison and Test Methods

Several String methods allow you to compare strings, substrings, byte arrays, and other objects with a given string. Some of these methods are compareTo(), endsWith(), equals(), equalsIgnoreCase(), regionMatches(), and startsWith().

Copy, Concatenation, and Replace Methods

The following methods are useful for copying, concatenating, and manipulating strings: concat(), copyValueOf(), replace(), and trim().

String Conversion and Generation

A number of string methods support String conversion. These are intern(), toCharArray(), toLowerCase(), toString(), toUpperCase(), and valueOf().

Following code shows some of the functions provided as above:

Java

```
class Main
{
  public static void main(String args[])
  {
    String s = " Java 2 Certification ";
    System.out.println(s);
    System.out.println(s.toUpperCase());
    System.out.println(s.toLowerCase());
    System.out.println("["+s+"]");

    s=s.trim();
    System.out.println("["+s+"]");

    s=s.replace('J','X');
    s=s.replace('C','Y');
    s=s.replace('2','Z');
    System.out.println(s);

    int i1 = s.indexOf('X');
    int i2 = s.indexOf('Y');
    int i3 = s.indexOf('Z');

    char ch[] = s.toCharArray();
    ch[i1]='J';
    ch[i2]='C';
    ch[i3]='2';
    s = new String(ch);
    System.out.println(s);
  }
}

```

This program performs several manipulations of a string s, which is initially set to " Java 2 Certification ". It prints the original string and then prints uppercase and lowercase versions of it, illustrating the use of the toUpperCase() and toLowerCase() methods. It prints the

string enclosed between two braces to show that it contains leading and trailing spaces. It then trims away these spaces using the trim() method and reprints the string to show that these spaces were removed. The program uses the replace() method to replace 'J', 'C', and '2' with 'X', 'Y', and 'Z', and prints out the string to show the changes. The replace() method is case sensitive. It uses the indexOf() method to get the indices of 'X', 'Y', and 'Z' within s. It uses the toCharArray() to convert the string to a char array. It then uses the indices to put 'J', 'C', and '2' back in their proper locations within the character array. The String() constructor is used to construct a new string from the character array. The new string is assigned to s and is printed. The program's output is as follows:

```
Java 2 Certification
JAVA 2 CERTIFICATION
java 2 certification
[ Java 2 Certification ]
[Java 2 Certification]
Xava Z Yertification
Java 2 Certification
```

Following is a program to check if string is equal to a predefined string or not: -

Java

```java
import java.util.Scanner;
import static java.lang.System.out;

class Main
{
  public static void main(String args[])
  {
    Scanner myScanner = new Scanner(System.in);
    String password = "defined";
    String userInput1, userInput2;
    out.print("Enter the password? ");
    userInput1 = "hello"; //myScanner.next();
    userInput2 = "defined"; //myScanner.next();

    if (password.equals(userInput1))
      out.println("You're okay!");
```

```
    else
      out.println("You're a not ok????.");

    if (password.equals(userInput2))
      out.println("You're okay!");
    else
      out.println("You're a not ok????.");
        }
}

```

It will test the user input with defined string and print the message according to the string is matched or not.

The Java API has a class named NumberFormat, and the NumberFormat class has a static method named getCurrencyInstance. When you call NumberFormat.getCurrencyInstance() with nothing inside the parentheses, you get an object that can mold numbers into U.S. currency amounts. Code below is an example.

```
import java.text.NumberFormat;
import java.util.Scanner;
class Main
{
 public static void main(String args[])
 {
   Scanner myScanner = new Scanner(System.in);
   double amount;
   boolean taxable;
   double total;
   NumberFormat currency = NumberFormat.getCurrencyInstance();
   String niceTotal;
   System.out.print("Amount: ");
   amount = myScanner.nextDouble();
   System.out.print("Taxable? (true/false) ");
   taxable = myScanner.nextBoolean();
   if (taxable)
     total = amount * 1.05;
   else
     total = amount;
   niceTotal = currency.format(total);
   System.out.println("Total: " + niceTotal);
```

```
 }
}
```

It will ask for an amount to the user and ask for whether it's a taxable amount or not, if it's a taxable amount then it prints the amount plus the tax calculated in program, otherwise it prints the amount only. Below are some of the output screens:

```
Amount: 3500
Taxable? (true/false) true
Total: $3,675.00

Amount: 4000
Taxable? (true/false) false
Total: $4,000.00
```

It is necessary to store Strings and other data types in a manner that allows the information to be found quickly. One of the best ways to store information for fast lookup is a hash table. A hash table stores information using a special calculation on the object to be stored that produces a hash code. The hash code is used to choose the location in the table at which to store the object. When the information needs to be retrieved, the same calculation is performed, the hash code is determined and a lookup of that location in the table results in the value that was stored there previously. Every object has the ability to be stored in a hash table. Class Object defines method hashCode to perform the hash code calculation. This method is inherited by all subclasses of Object. Method hashCode is overridden by String to provide a good hash code distribution based on the contents of the String. The example below demonstrates the hashCode method for two Strings containing "hello" and "Hello". Note that the hash code value for each String is different. That is because the Strings themselves are lexicographically different.

Java

```
class Main
{
  // test String hashCode method
  public static void main( String args[] )
  {
```

```
    String s1 = "CodeQuotient", s2 = "codequotient";
    String output = "The hash code for \"" + s1 + "\" is " +  s1.hashCode() + "\n The hash code for \"" + s2 + "\" is " + s2.hashCode();
    System.out.println("Demonstrating String Method hashCode \n " + output);
  }
}
```

The output of this program is as below:

```
Demonstrating String Method hashCode The hash code for "hello" is 99162322
 The hash code for "Hello" is 69609650
```

Often it is useful to search for a character or set of characters in a String. For example, if you are creating your own word processor, you might want to provide a capability for searching through the document. The below code demonstrates the many versions of String methods indexOf and lastIndexOf that search for a specified character or substring in a String. All the searches in this example are performed on the String letters (initialized with "abcdefghijklmabcdefghijklm") in method main of class StringIndexMethods.

Java

```
 class Main
{
  public static void main( String args[] )
  {
    String letters = "CodeQuotient$%";

    // test indexOf to locate a character in a string
    String output = "'Q' is located at index " + letters.indexOf('Q');
    output += "\n'C' is located at index " + letters.indexOf('C');
    output += "\n'$' is located at index " + letters.indexOf( '$' );

    // test lastIndexOf to find a character in a string
    output += "\n\nLast 'C' is located at index " + letters.lastIndexOf( 'C' );
```

```
14      output += "\nLast 'e' is located at index " +
    letters.lastIndexOf( 'e' );
15      output += "\nLast '$' is located at index " +
    letters.lastIndexOf( '$' );
16
17      // test indexOf to locate a substring in a string
18      output += "\n\n\"de\" is located at index " +
    letters.indexOf( "de" );
19      output += "\n\"quo\" is located at index " +
    letters.indexOf( "quo" );
20
21      // test lastIndexOf to find a substring in a string
22      output += "\n\nLast \"de\" is located at index " +
    letters.lastIndexOf( "de" );
23      output += "\nLast \"quo\" is located at index " +
    letters.lastIndexOf( "quo" );
24      System.out.println("Demonstrating String Class
    \"index\" Methods\n" + output);
25    }
26  }
27
```

Above code uses the method indexOf to locate the first occurrence of a character in a String. If indexOf finds the character, indexOf returns the index of that character in the String; otherwise, indexOf returns –1. There are two versions of indexOf that search for characters in a String. First version of indexOf is used that takes one integer argument, which is the integer representation of a character. Remember that a character constant in single quotes is of type char and specifies the integer representation of the character in the Unicode character set. The second version of method indexOf is also used which takes two integer arguments—the integer representation of a character and the starting index at which the search of the String should begin.

The statements also use method lastIndexOf to locate the last occurrence of a character in a String. Method lastIndexOf performs the search from the end of the String toward the beginning of the String. If method lastIndexOf finds the character, lastIndexOf returns the index of that character in the String; otherwise, lastIndexOf returns –1. There are two versions of lastIndexOf that search for characters in a String. The expression uses the first version of method lastIndexOf that takes one integer argument that is the integer

representation of a character. The second version of method lastIndexOf is also used which takes two integer arguments—the integer representation of a character and the highest index from which to begin searching backward for the character.

Other versions of methods indexOf and lastIndexOf used in this program which each take a String as the first argument. These versions of the methods perform identically to those described above except that they search for sequences of characters (or substrings) that are specified by their String arguments.

String is Immutable

Since string literals with the same contents share storage in the common pool, Java's String is designed to be immutable. That is, once a String is constructed, its contents cannot be modified. Otherwise, the other String references sharing the same storage location will be affected by the change, which can be unpredictable and therefore is undesirable. Methods such as toUpperCase() might appear to modify the contents of a String object. In fact, a completely new String object is created and returned to the caller. The original String object will be deallocated, once there is no more references, and subsequently garbage-collected. Because String is immutable, it is not efficient to use String if you need to modify your string frequently (that would create many new Strings occupying new storage areas). For example,

```
// inefficient codes
String str = "Hello";
for (int i = 1; i < 1000; ++i)
   str = str + i;
```

If the contents of a String have to be modified frequently, use the StringBuffer or StringBuilder class instead (discussed shortly).

StringTokenizer

Very often, you need to break a line of texts into tokens delimited by white spaces. The java.util.StringTokenizer class supports this. For

example, the following program reverses the words in a String.

Java

```java
import java.util.StringTokenizer;

class Main
{
  public static void main(String[] args)
  {
    String str = "Code Quotient Get Better at Coding";
    String strReverse;
    StringBuilder sb = new StringBuilder();
    StringTokenizer st = new StringTokenizer(str);

    while (st.hasMoreTokens())
    {
      sb.insert(0, st.nextToken());
      if (st.hasMoreTokens())
        sb.insert(0, " ");
    }
    strReverse = sb.toString();
    System.out.println(strReverse);
  }
}
```

The StringTokenizer class has following constructors and methods: -

```
// Constructors
StringTokenizer(String s)  // Constructs a StringTokenizer for
the given string, using the default delimiter set of " \t\n\r\f"
(i.e., blank, tab, newline, carriage-return, and form-feed).
                           // Delimiter characters themselves
will not be treated as tokens.
StringTokenizer(String s, String delimiterSet)  // Use characters
in delimiterSet as delimiters.

// Methods
boolean hasNextToken()     // Returns true if next token available
String nextToken()         // Returns the next token
```

According to JDK documentation, StringTokenizer is one of the retained classes for compatibility reasons and its use is discouraged.

We can use split() method of String class or the special regex package to achieve the same functionalities as shown below:

Java

```java
// Reverse the words in a String using split() method of the String class
class Main
{
  public static void main(String[] args)
  {
    String str = "Code Quotient Get Better at Coding";
    String[] tokens = str.split("\\s");  // white space '\s' as delimiter
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < tokens.length; ++i)
    {
      sb.insert(0, tokens[i]);
      if (i < tokens.length - 1)
        sb.insert(0, " ");
    }
    String strReverse = sb.toString();
    System.out.println(strReverse);
  }
}
```