Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Java: StringBuffer & StringBuilder/5aabd63c26e8ba7398f58d6e

TUTORIAL

Java: StringBuffer & StringBuilder

Topics

1.6 Comparison of String/StringBuffer/StringBuilder

As Strings are immutable because String literals with same content share the same storage in the string common pool. Modifying the content of one String directly may cause adverse side-effects to other Strings sharing the same storage. JDK provides two classes to support mutable strings: StringBuffer and StringBuilder (in core package java.lang) . A StringBuffer or StringBuilder object is just like any ordinary object, which are stored in the heap and not shared, and therefore, can be modified without causing adverse side-effect to other objects.

StringBuilder class was introduced in JDK 1.5. It is the same as StringBuffer class, except that StringBuilder is not synchronized for multi-thread operations. However, for single-thread program, StringBuilder, without the synchronization overhead, is more efficient.

StringBuffer: The StringBuffer class is the force behind the scenes for most complex string manipulations. The compiler automatically declares and manipulates objects of this class to implement common string operations. The StringBuffer class provides three constructors: an empty constructor, a constructor with a specified initial buffer length, and a constructor that creates a StringBuffer object from a String object. In general, you will find yourself constructing StringBuffer objects from String objects, and the last constructor will be the one you use most often. The StringBuffer class provides several versions of the append() method to convert and append other objects and primitive data types to StringBuffer objects. It provides a similar set of insert() methods for inserting objects and

primitive data types into StringBuffer objects. It also provides methods to access the character-buffering capacity of StringBuffer and methods for accessing the characters contained in a string. It is well worth a visit to the StringBuffer API pages to take a look at the methods that it has to offer. Following code shows how StringBuffer objects can be manipulated using the append(), insert(), and setCharAt() methods.

Java

```
class Main
{
  public static void main(String args[])
  {
    StringBuffer sb = new StringBuffer(" is ");

    sb.append("Fun");
    sb.append('!');
    sb.insert(0,"Java");
    sb.append('\n');
    sb.append("This is ");
    sb.append("true");
    sb.setCharAt(21,'T');
    sb.append('\n');
    sb.append("Java is 1 Programming");
    sb.append("Language");

    String s = sb.toString();
    System.out.println(s);
  }
}

```

The program creates a StringBuffer object using the string " is ". It appends the string "Fun" using the append() method and the character '!' using an overloaded version of the same method. The insert() method is used to insert the string "Java" at the beginning of the string buffer. Three appends are used to tack on a newline character (\n), the string "This is ", and the boolean value true. The append() method is overloaded to support the appending of the primitive data types as well as arbitrary Java objects. The setCharAt()

method is used to replace the letter 't' at index 21 with the letter 'T'. The charAt() and setCharAt() methods allow StringBuffer objects to be treated as arrays of characters. Finally, another newline character is appended to sb, followed by the string "Java is 1 ProgrammingLanguage". The StringBuffer object is then converted to a string and displayed to the console window. The output of the program is as follows:

```
Java is Fun!
This is True
Java is 1 ProgrammingLanguage
```

StringBuffer class has 3 constructors to build Strings as below:

```
StringBuffer()             // an initially-empty StringBuffer
StringBuffer(int size)     // with the specified initial size
StringBuffer(String s)     // with the specified initial content
```

It has several methods used to manipulate the StringBuffer objects, some of these methods are listed below:

```
int length()         // Returns the length of String
StringBuffer append(type arg)  // type could be primitives,
char[], String, StringBuffer, etc
StringBuffer insert(int offset, arg)

StringBuffer delete(int start, int end)
StringBuffer deleteCharAt(int index)
void setLength(int newSize)
void setCharAt(int index, char newChar)
StringBuffer replace(int start, int end, String s)
StringBuffer reverse()

// Methods for extracting whole/part of the content
char charAt(int index)
String substring(int start)
String substring(int start, int end)
String toString()

// Methods for searching
int indexOf(String searchKey)
int indexOf(String searchKey, int fromIndex)
int lastIndexOf(String searchKey)
int lastIndexOf(String searchKey, int fromIndex)
```

StringBuffer object is just like any other object. You need to use a constructor to create a StringBuffer (instead of assigning to a String literal). Furthermore, '+' operator does not apply to objects, including the StringBuffer. You need to use a proper method such as append() or insert() to manipulating a StringBuffer.

To create a string from parts, It is more efficient to use StringBuffer (multi-thread) or StringBuilder (single-thread) instead of via String concatenation. For example,

```
// Create a string of YYYY-MM-DD HH:MM:SS
int year = 2018, month = 5, day = 5;
int hour = 08, minute = 55, second = 33;
String dateStr = new StringBuilder().append(year).append("-
").append(month).append("-").append(day).append("
").append(hour).append(":").append(minute).append(":").append(second).toString();
System.out.println(dateStr);
```

Rule of Thumb: Strings are more efficient if they are not modified (because they are shared in the string common pool). However, if you have to modify the content of a string frequently (such as a status message), you should use the StringBuffer class (or the StringBuilder described below) instead.

StringBuilder: JDK 1.5 introduced a new StringBuilder class (in package java.lang), which is almost identical to the StringBuffer class, except that it is not synchronized. In other words, if multiple threads are accessing a StringBuilder instance at the same time, its integrity cannot be guaranteed. However, for a single-thread program (most commonly), doing away with the overhead of synchronization makes the StringBuilder faster.

StringBuilder is API-compatible with the StringBuffer class, i.e., having the same set of constructors and methods, but with no guarantee of synchronization. It can be a drop-in replacement for StringBuffer under a single-thread environment.

Comparison of String/StringBuffer/StringBuilder

The following program compare the times taken to reverse a long string via a `String` object and a `StringBuffer`.

Java

```
// Reversing a long String via a String vs. a StringBuffer
public class Main
{
  public static void main(String[] args)
  {
    long beginTime, elapsedTime;

    // Build a long string
    String str = "";
    int size = 4623;
    char ch = 'a';
    beginTime = System.nanoTime();   // Reference time in nanoseconds
    for (int count = 0; count < size; ++count)
    {
      str += ch;
      ++ch;
      if (ch > 'z') {
        ch = 'a';
      }
    }
    elapsedTime = System.nanoTime() - beginTime;
    System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Build String)");

    // Reverse a String by building another String character-by-character in the reverse order
    String strReverse = "";
    beginTime = System.nanoTime();
    for (int pos = str.length() - 1; pos >= 0 ; pos--) {
      strReverse += str.charAt(pos);   // Concatenate
    }
    elapsedTime = System.nanoTime() - beginTime;
    System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Using String to reverse)");

    // Reverse a String via an empty StringBuffer by appending characters in the reverse order
    beginTime = System.nanoTime();
    StringBuffer sBufferReverse = new StringBuffer(size);
    for (int pos = str.length() - 1; pos >= 0 ; pos--) {
```

```
      sBufferReverse.append(str.charAt(pos));      // append
    }
    elapsedTime = System.nanoTime() - beginTime;
    System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Using StringBuffer's append() to reverse)");

    // Reverse a String by creating a StringBuffer with the given String and invoke its reverse()
    beginTime = System.nanoTime();
    StringBuffer sBufferReverseMethod = new StringBuffer(str);
    sBufferReverseMethod.reverse();     // use reverse() method
    elapsedTime = System.nanoTime() - beginTime;
    System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Using StringBuffer's reverse() method)");

    // Reverse a String via an empty StringBuilder by appending characters in the reverse order
    beginTime = System.nanoTime();
    StringBuilder sBuilderReverse = new StringBuilder(size);
    for (int pos = str.length() - 1; pos >= 0 ; pos--) {
      sBuilderReverse.append(str.charAt(pos));
    }
    elapsedTime = System.nanoTime() - beginTime;
    System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Using StringBuilder's append() to reverse)");

    // Reverse a String by creating a StringBuilder with the given String and invoke its reverse()
    beginTime = System.nanoTime();
    StringBuffer sBuilderReverseMethod = new StringBuffer(str);
    sBuilderReverseMethod.reverse();
    elapsedTime = System.nanoTime() - beginTime;
    System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Using StringBuidler's reverse() method)");
  }
}
```

Observe **StringBuilder** is significantly faster than **StringBuffer**, and String. The reverse() method is the fastest, which takes about the

same time for **StringBuilder** and **StringBuffer**.