Theory: StringBuilder

13 minutes 0 / 3 problems solved

Current topic:
StringBuilder Stage 3

Topic depends on:
Objects Stage 3

§1. Mutable strings

As you may know, in Java strings are **immutable**. It means that once created, a string cannot be changed. If we want to modify the content of a string object, we should create a new string. This may not be the best way when we perform a lot of modifications, because each operation creates a new object that is bad for performance.

Fortunately, there is a special class named `StringBuilder` that is used to create mutable string objects. An object of this class is similar to a regular string, except that it can be modified. As an example, it is better to use `StringBuilder` than `String` where lots of concatenation is being performed at runtime.

§2. Constructing objects

It is possible to create an empty object of `StringBuilder`

```
StringBuilder empty = new StringBuilder();
System.out.println(empty); // ""
```

or pass a string to it:

```
StringBuilder sb = new StringBuilder("Hello!");
System.out.println(sb); // "Hello!"
```

Note, we do not need to import anything to use this class in programs.

§3. Some important methods

The `StringBuilder` class provides a set of useful methods to manipulate objects. Let's consider some of them.

- `int length()` returns the length (characters count) like for a regular string. This method does not modify the object.

```
StringBuilder sb = new StringBuilder("I use Java");
System.out.println(sb.length()); // 10
```

- `char charAt(int index)` returns a character located at the specified index. The first character has the index 0. This method does not modify the object.

```
StringBuilder sb = new StringBuilder("I use Java");
System.out.println(sb.charAt(0)); // 'I'
System.out.println(sb.charAt(6)); // 'J'
```

- `void setCharAt(int index, char ch)` sets a character located at the specified index to **ch**.

```
StringBuilder sb = new StringBuilder("start");
sb.setCharAt(1, 'm');
System.out.println(sb); // "smart"
```

- `StringBuilder deleteCharAt(int index)` removes a character at the specified position.

```
StringBuilder sb = new StringBuilder("dessert");
sb.deleteCharAt(2);
System.out.println(sb); // "desert"
```

- `StringBuilder append(String str)` concatenates the given string to the end of the invoking StringBuilder object. There are also several overloadings to take primitive types and, even, arrays of characters.

```
StringBuilder sb = new StringBuilder("abc");
sb.append("123");
System.out.println(sb); // "abc123"
```

It is also possible to invoke this method multiple times on the same object in the same statement because this method returns the same modified object.

```
StringBuilder messageBuilder = new StringBuilder(); // empty

messageBuilder
        .append("From: Kate@gmail.com\n")
        .append("To: Max@gmail.com\n")
        .append("Text: I lost my keys.\n")
        .append("Please, open the door!");

System.out.println(messageBuilder);
```

Output:

```
From: Kate@gmail.com
To: Max@gmail.com
Text: I lost my keys.
Please, open the door!
```

- `StringBuilder insert(int offset, String str)` inserts the given string into the existing `StringBuilder` object at the given position, indicated by the offset. The method has a lot of overloadings for different types.

```
StringBuilder sb = new StringBuilder("I'm a programmer.");
sb.insert(6, "Java ");
System.out.println(sb); // I'm a Java programmer.
```

- `StringBuilder replace(int start, int end, String str)` replaces the substring from specified string index (inclusive) to the end index (exclusive) with a given string.

```
StringBuilder sb = new StringBuilder("Let's use C#");
sb.replace(10,12,"Java");
System.out.println(sb); // Let's use Java
```

- `StringBuilder delete(int start, int end)` removes the substring from the start index (inclusive) to the end index (exclusive).

```
StringBuilder sb = new StringBuilder("Welcome");
sb.delete(0,3);
System.out.println(sb); // "come"
```

- `StringBuilder reverse()` causes this character sequence to be replaced by the reverse of the sequence.

```
StringBuilder sb = new StringBuilder("2 * 3 + 8 * 4");
sb.reverse();
System.out.println(sb); // "4 * 8 + 3 * 2"
```

Note, when you have a `StringBuilder` object, you can get a `String` invoking the `toString` method.

For more details about methods see the documentation.

§4. The length() and capacity()

There are two methods that cannot be confused: `length` and `capacity`. The `length` returns the actual number of characters when `capacity` returns the amount of storage available for newly inserted characters, beyond which an

allocation will occur. The capacity is a part of the internal representation of `StringBuilder` and its value will dynamically change.

The following example will help to better distinguish these methods

```
StringBuilder sb = new StringBuilder(); // initial capacity is 16

System.out.println(sb.length());   // 0
System.out.println(sb.capacity()); // 16

sb.append("A very long string");

System.out.println(sb.length());   // 18
System.out.println(sb.capacity()); // 34
```

It is possible to specify the capacity when creating a `StringBuilder` object, but it is not very often used possibility:

```
StringBuilder sb = new StringBuilder(30);

System.out.println(sb.length());   // 0
System.out.println(sb.capacity()); // 30
```

§5. Conclusion

The `StringBuilder` class is used to create mutable strings which can be modified at runtime. You can perform different operations on `StringBuilder` objects like append, reverse, replace, delete, etc. No new object will be created. It is recommended to use this class instead of `String` where a lot of modifications is being performed. This will prevent the creation of multiple intermediate objects, therefore, it will work faster and require less memory. One common case for this is a sequence of concatenations.

Note, there is another similar class called `StringBuffer`. We will consider it in next topics.

Report a typo

366 users liked this theory. **1** didn't like it. **What about you?**

Start practicing

Table of contents:

Comments (3) Hints (0) Useful links (1) Show discussion