15-string-handling-operations

Contents

- CharSequence
  - Strings
    - Constructors
    - Basics
    - Comparison
    - Searching
    - Reducing
  - StringBuffer/StringBuilder

CharSequence

`CharSequence` is the base interface from which most of everything in the core library that is considered a `string or a sequence` of some sorts implements. It provides the very basic interface which all sequences of characters must implement

```
int length();
String toString();
boolean isEmtpy();
char charAt(int index);
.....................
// and many many more
```

Strings

Constructors

Unless specified otherwise in the constructor the Java runtime will use the system encoding to create strings, when they are passed in under the form of char or byte arrays, usually that is either UTF-8 or UTF-16

1. String(char[] chars) - takes in array of 2-byte sized chars, to construct a new string object

2. String(byte[] bytes) - an alternative variant of the char[] constructor where 1-byte sized elements are used
3. String(StringBuilder builder) - a copy constructor for strings which copies the contents of a StringBuilder to a new string object
4. Literals - string literals are special type of String objects which are handled by the runtime, as already described these literals are pooled by the runtime which allows them to be re-used e.g. String lit = "literal" (literals are constant and compile time values defined in source code)

Basics

1. Concatenation - the java language does not allow any arithmetic operators to be used on strings except for the `+` operator which can be used to concatenate strings, string literals as well - if concatenation of a multiple string literals is performed, one final string object is generated from those and stored in the string pool.

```
String long = "one" + "two" + "three"; // will produce one string object
    pooled by the run-time
```

   Note, extra care needs to be taken when concatenating strings with other types, since there is no precedence the expression will be evaluated left to right, therefore one could face the following case which has different behavior based on the actual expression

```
String expr = "four " + 2 + 2; // this will produce the string four 22,
   not four 4
String expr = "four " + (2 + 2); // this will produce the string four 4,
    not four 22
```

`When concatenation is performed, Java would use the Object.toString or String.valueOf methods to convert the other data types to strings objects, that is how the language is able to concatenate seemingly unrelated to String objects`

2. Extraction - to extract characters from a string, there are several interfaces which the core library exposes - `charAt, getChars or getBytes`

   - charAt - extract single character from a string

```
String s = "This is a demo of the charAt method.";
char c = s.charAt(3); // extract char from index 3
```

   - getChars - extract a range of characters from a string

```
String s = "This is a demo of the getChars method.";
int start = 10; int end = 14; // starting and ending indices in the
   string
char buf[] = new char[end - start]; // make array with space for 4
   characters
s.getChars(start, end, buf, 0); // pull 4 characters from index 10 to 14
    non inclusive
```

   - getBytes - extract the byte representation of the string

```
String s = "This is a demo of the getBytes method.";
byte[] bytes = s.getBytes(); // converts the utf-16 encoded string into
   ascii byte array
```

   - toCharArray - convert the entire string to a char array

```
String s = "This is a demo of the toCharArray method.";
char[] bytes = s.toCharArray(); // convert the entire string to a 2-byte
   wide char array
```

Comparison

To compare the contents of two string objects, instead of their reference one has to use the `equals or equalsIgnoreCase`, which are pretty much self explanatory.

1. equals / equalsIgnoreCase - compare the strings character wise, either `ignoring or respecting the character casing`
2. regionMatches - compares a region from one string to another, the method is overloaded to accept a flag of `ignoreCase`
3. startsWith / endsWith - checks if the given string starts or ends with another string, a specialized case of `regionMatches`
4. compareTo / compareToIgnoreCase - implements comparable, and for strings that implies comparing them `lexicographically` or in `alphabetical` order

Searching

1. indexOf - finds the first index or -1 if not found, in a string that matches a given char or string/substring

```
int indexOf(int ch, int startIndex)
int indexOf(String str, int startIndex)
```

2. lastIndexOf - finds the last index or -1 if not found, in a string that matches a given char or string/substring

```
int lastIndexOf(int ch, int startIndex)
int lastIndexOf(String str, int startIndex)
```

Reducing

Since strings are `immutable` one can not simply modify a string, since none of the api methods work on the instance itself, each operation that produces a string creates a new string object in the java run-time. Therefore to reduce one string to another one could use various forms of the `substring` methods

1. substring - generate a new string object from the source string instance, starting at a given start index, and optional end index (non inclusive)

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

2. concat - to extend multiple string objects, and produce a new one, it is equivalent to using the `+ operator`

```
String concat(String other)
```

3. replace - replaces a single char or string in a string, generating a new instance representing the replacement result

```
String replace(char target, char replacement)
String replace(CharSequence target, CharSequence replacement)
```

4. trim - returns a copy of the source string in which all the leading and trailing white space characters are removed, including tabs and other unicode white space characters

```
String trim()
```

5. join - extension of the concat method which produces a join of a number of strings with a specific delimiter

```
String join(CharSequence delim, CharSequence... strings)
```

6. toUppser/LowerCase - produce a copy of the original string object where all character casings are converted to lower or upper case.

StringBuffer/StringBuilder

This is the `de-facto` structure to be used if one wishes to produce a mutating string object, `StringBuilder` provides a fast way to `append/preprend` to a string, without having to pay the cost of copying the string constantly as it is using an internal dynamic array instead and provides methods to mutate it directly, unlike the base `String` class. The `StringBuilder` also has another companion class `StringBuffer` which is essentially the same, the difference is that `StringBuilder is synchronized` it is equivalent to the difference between `Integer and AtomicInteger` classes provided by the Java core.

The default constructor for both takes in either a size (initial capacity), or the ability to construct it initially from an existing `CharSequence`, such as a String.

`Note that since the buffer/builder allows the string to be mutated, the api provides a way to reduce or expand the string at any point, using either other strings - append, prepend, insert, or explicitly setting the length of the string, with methods such as setLength, the internal buffer which stores the string is bound by a capacity property which signifies how much emtpy space the buffer contains before a new reallocation is required to expand it, shrinking the buffer is done with trimToSize which will reduce the capcity of the buffer to match the size or length of the string`

```
StringBuffer StringBuffer()
StringBuffer StringBuffer(int size)
StringBuffer StringBuffer(CharSequence chars)
```

1. capacity() - the current capacity the `buffer` can hold
2. length() - the current actual length of the `string` contained
3. ensureCapacity(int) - enlarge the capacity of the `buffer`
4. charAt(int)/setChaAt(int, char) - extract or update a given character at a given position in the string
5. setLength(int) - extends or reduces the length of the string, `null` chars are added to fill in empty space if length is bigger than current string, otherwise string is reduced to the given length
6. append(String) - appends a string to the end of the buffer
7. prepend(String) - prepends a string at the start of the buffer
8. insert(int start, String) - inserts a string at a given starting index
9. trimToSize() - reduce the internal capacity to match the length of the string
10. reverse() - reverse the string contents, this is fast since no re-allocation needs to be done
11. delete(int, int)/delCharAt(int) - deletes a substring given a start and end index, or a single char at position

12. replace(int, int, String) - replace a range within the buffer, given start and end positions for the range, taking care of overriding over the range with a given string