Java Strings

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class. Strings are **immutable**; their values cannot be changed after they are created.

The class `String` includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase and so on.

```
System.out.println("abc");
String cde = "cde";
System.out.println("abc" + cde);
String c = "abc".substring(2,3);
String d = cde.substring(1, 2);
```

We can use *Text Blocks* by declaring the string with """ (three double quote marks):

```
String textBlock = """
    He who becomes the slave of habit,
    who follows the same routes every day,
    who never changes pace,
    who does not risk and change the color of his clothes,
    who does not speak and does not experience,
    dies slowly.""";
```

It is by far the most convenient way to declare a multiline string. Indeed, we don't have to deal with line separators or indentation spaces, as noted in our dedicated article.

Strings in memory

Being immutable, when created using `String s = "something"` they are stored in a special pool in which the same string is stored only once.

```
// memory use
String first = "Baeldung";

// no actual memory use
String second = "Baeldung";

// True
System.out.println(first == second);
```

Instead, when created using the new operator, Strings are stored in memory (heap) as standard objects. As such, each string occupies its own memory.

```
// memory use
String first = new String("Baeldung");

// memory use
String second = new String("Baeldung");

// memory use
String third = "Baeldung";

// False
System.out.println(first == second);

// False
System.out.println(first == third);
```

Integer caching

A similar mechanism has been implemented for Integers and called **Integer caching**. The most commonly used Integer values (-128, 127) are allocated within a *cache area* (inside heap area, some extra memory is allocated named as cache) while the rest of the integer values are allocated in the *heap area*.

If you create a new object with a value which is already present in *cache*, it will not create another object but a reference to the existing object will be returned. This might lead to apparently wrong behaviours.

```
public class IntegerCaching {
    public static void main(String[] args) {
        for (int i = -130; i <= 130; i++) {
            System.out.println(i + " --> " + (Integer.valueOf(i) ==
        }
    }
}
```

```
-130 --> false
-129 --> false
-128 --> true
-127 --> true
...
126 --> true
127 --> true
128 --> false
129 --> false
130 --> false
```

Comparing strings

The **== operator** verifies if two references objects point to the same object. On the contrary, the **equal() method** verifies if two objects (any object!) have the same internal state.

```
public class ComparingStrings {
    public static void checkStrings(String s1, String s2) {
        if (s1 == s2) {
            System.out.println("s1 and s2 point to the same object"
        } else {
            System.out.println("s1 and s2 point to different objects
        }

        if (s1.equals(s2)) {
            System.out.println("s1 and s2 have the same content");
        } else {
            System.out.println("s1 and s2 have different contents");
        }
    }
    public static void main(String[] args) {
        checkStrings("Hello World!", "Hello World!");
        checkStrings(new String("Hello World!"), new String("Hello W
        checkStrings(new String("Hello World!"), new String("Hello M
    }
}
```

Concatenating strings

The + operator

The + can be used to concatenate 2 or more strings. It can also concatenate variables of other types which are automatically converted to the String type before being concatenated.

```
String s = "This string" + " is made " + "by three substrings";

System.out.println("#students = " + 3);
System.out.println("PI = " + 3.1415);
```

StringBuilder

Being strings immutable, **when two Strings are concatenated using +, the two strings are actually discarded and a new one (containing their concatenation) is instantiated.** This process is **slow!**.

`StringBuilder` provides a better way for concatenating strings.

```
public class ConcatenateBenchmark {
    public static String concatenateSlow(int iterations) {
        // slow version
        String s = "";
        for (int i = 0; i < iterations; i++) {
            s += 'a';
        }
        return s;
    }

    public static String concatenateFast(int iterations) {
        // fast version using StringBuilder
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < iterations; i++) {
            sb.append('a');
        }
        return sb.toString();
    }

    public static void main(String[] args) {
        long start = System.nanoTime();
        concatenateSlow(10000);
        long end = System.nanoTime();
        System.out.println("Execution time: " + Duration.ofNanos(en
    }
}
```

String main methods

`charAt()` Returns the character at the specified index (position)

`compareTo()` Compares two strings lexicographically

`concat()` Appends a string to the end of another string

`contains()` Checks whether a string contains a sequence of characters

`endsWith()` Checks whether a string ends with the specified character(s)

`isEmpty()` Checks whether a string is empty or not

`length()` Returns the length of a specified string

`replace()` Searches a string for a specified value, and returns a new string where those values are replaced

`split()` Splits a string into an array of substrings

`startsWith()` Checks whether a string starts with specified characters

`substring()` Returns a new string which is the substring of a specified string

`valueOf()` Returns the string representation of the specified value

Standard streams

Java has 3 streams called `System.in`, `System.out`, and `System.err` which are commonly used to provide input to, and output from Java applications. Most commonly used is probably `System.out` for writing output to the console from console programs (command line applications). They are initialized by the Java runtime when a Java VM starts up, so you don't have to instantiate any streams yourself.

- `System.in` is an `InputStream` which is typically connected to keyboard input of console programs.
- `System.out` is a `PrintStream` to which you can write characters. It normally outputs the data you write to it to the console/terminal.
- `System.err` is a `PrintStream`. It works like `System.out` except it is normally only used to output error texts.

PrintStream Methods

Prints a string:

```
void print(String s)
```

Prints a string and adds a newline:

```
void println(String s)
```

Writes a formatted string using the specified format string and arguments:

```
void printf(String format, Object... args)
```

Returns a formatted string using the specified format string and arguments:

```
String String.format(String format, Object... args)
```

Acquiring primitive types from the keyboard

Scanner is a text scanner which can parse primitive types and strings using regular expressions. It **breaks its input into tokens** using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be **converted into values of different types** using the various *next* methods.

```
/* from stdin */
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();

/* from a file */
Scanner sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
    long aLong = sc.nextLong();
}
```

Resources

- https://www.baeldung.com/java-string
- https://www.baeldung.com/java-string-operations