Static data types

Strings

If we want to work with strings in Java, we have the `String` class, with some useful methods that we can use: convert to upper or lower case, get a substring, find a text... You can have a whole list of available methods in the String official documentation. Let's explore some of them.

1. Basic string management

We can create a string in many different ways: with a constant value, asking the user to type something...

```
String text = "Hello world";
String name = scanner.nextLine();
```

We can also concatenate strings with the `+` operator, or in some cases with `+=` operator (if we want to add a string at the end of another).

```
text = text + ", how are you?";
```

We can't treat a string as a char array (as in C++ or C#), and get to each character with the corresponding index between square brackets. If we want to get the character at a given position, we need to use the `charAt` method. We can also get the length of a string with its `length()` method.

```
for (int i = 0; i < text.length(); i++)
    System.out.println(text.charAt(i));
```

Regarding conversions, we can easily convert a non-string variable into a string just joining it with an empty string. Alternatively, we can also use `String.valueof` instruction for this purpose:

```
int number = 23;
String text = "" + number;
String text2 = String.valueOf(number);
```

2. Some advanced operations with strings

In this section we are going to explore some more advanced features of strings, such as comparing strings, taking substrings in a string variable, splitting a string in parts and so on.

2.1. String comparison

We can compare two strings (alphabetically) in some different ways. If we want to know which is greater or lower, we can use the `compareTo` method. It returns a negative number if the string on the left is lower, 0 if both strings are equal, or a positive number if the string on the right is lower.

```
if (text1.compareTo(text2) < 0)
    System.out.println("Second text is greater");
```

If we want to check if two strings are equal, we use the `equals` method (remember, we must NOT use the `==` comparator for this purpose). We can also use `equalsIgnoreCase` method if we want to ignore if strings are in uppercase or lowercase.

```
if (text1.equals(text2))
    System.out.println("Texts are equal");

if ("hello".equalsIgnoreCase("HELLO"))
    System.out.println("Text are equal ignoring cases");
```

2.2. Finding texts in strings

We have a wide variety of instructions inside string elements to find texts. For instance, if we only want to know if a text contains a given subtext, we can use `contains` method, that returns a boolean:

```
if (text.contains("hello"))
    System.out.println("There is a 'hello' in the text");
```

If we want to know the index at which a given subtext appears, we can choose among `indexOf` (gets the first occurrence of the subtext, or -1 if it does not exist) or `lastIndexOf` (gets the last occurrence of the subtext, or -1 if it does not exist)

```
int pos = text.indexOf("hello");
if (pos >= 0)
    System.out.println("There is a 'hello' at position" + pos);
```

If we want to know if a text starts with a given prefix or ends with a given suffix, we use the `startsWith` or `endsWith` methods, which return a boolean

```
if (text.startsWith("Hello"))
    System.out.println("Text starts with 'Hello'");
```

2.3. String conversions

We can convert the whole string to upper and lower case with `toUpperCase` and `toLowerCase` methods:

```
String text = "Hello world";
String textUpper = text.toUpperCase(); // "HELLO WORLD"
```

We can get a substring of a given string with the `substring` method. It has two parameters: the index from which we must start getting the substring (starting at 0), and the index at which we must stop getting the string (excluded). If this second parameter is omitted, it returns the resulting string from the initial index to the end of the string. For instance, `"Welcome".substring(3, 5)` returns "co" (indexes 3 and 4 of the string).

We can replace a substring with another one with the `replace` method. It has two arguments: the old text and the new text, and it returns the resulting string. It replaces EVERY occurrence of the old string with the new string.

```
String result = text.replace("Hello", "Good morning");
```

There are some other options for this purpose, such as `replaceAll` method, which uses regular expressions to match the text to be replaced, or `replaceFirst`, which only replaces the first occurrence of the old text with the new one.

We can split a string using a delimiter with the `split` method. It returns an array with the resulting parts.

```
String text = "Hello world";
String[] parts = text.split(" ");
// Two parts, "Hello" and "world"
```

Finally, we can also do the opposite operation, this is, joining parts of a string with a common delimiter, using `String.join` method. We need to specify the delimiter, and then the array or sequence of texts to be joined.

```
String[] parts = {"One", "Two", "Three"};
String result = String.join(",", parts);   // "One,Two,Three"
```

> **Exercise 1**:
>
> Create a program called **SortJoin** that asks the user to enter a list of names separated by whitespaces. Then, the program must split the string, sort the names alphabetically and output them separated by commas. For instance, if the user types this name list: `Susan Kailey William John`, then the program must output `John, Kailey, Susan, William`.

> **Exercise 2**:
>
> Create a program called **CheckMessages** that asks the user to type 10 strings. The program must store them in an array, and then replace the text "Eclipse" with "IntelliJ" in every string that contains "Eclipse". The program must output the updated version of the strings stored in the array, once the replacement has been done.

> **Exercise 3**:
>
> Create a program called **LispChecker**. LISP is a programming languages where every instruction is enclosed in parentheses. This could be a set of instructions in LISP:

```
(let ((new (x-point a y))))
```

> You must implement a program that takes a string with LISP instructions (just one string) and then check if the parentheses are correct (this is, the number of opening parentheses and closing parentheses are the same).

3. Strings and static data

This section is about static data types. We have seen that these types don't change their size along program execution. However, we can increase or decrease the size of a string by concatenating or removing pieces of text. So, how can we say that a string is a static data size?

We need to take into account that Strings in Java are ALWAYS immutable, as in many other programming languages. So, every transformation that we need to do over a string needs to be assigned to a new (or old) variable. That's why methods such as `toUppeCase` or `substring` always return a new string, they never affect the original string. However, we can always re-assign the old string with this new value:

```
String text = "Hello world";

// This way we don't affect original text
String newText = text.toUpperCase();

// This way we modify original variable, and it points
// to this new text (old text is lost in memory)
text = text.toUpperCase();
```

Due to this, joining these instructions over strings is not very efficient, since we need to create a new string in memory for every new operation. Instead of this, we can also use *StringBuilder* element instead.

3.1. Using *StringBuilder*

Through `SceneBuilder` element we can create editable strings. This increases the speed of certains operations, and decreases the use of memory. In order to create these elements, we just need to use `new` operator with the initial value of the string to be stored:

```
StringBuilder text = new StringBuilder("Hello");
```

Then, we have some methods available to perform some editing tasks:

- `append` method lets us add new strings at the end of the existing one, just like `+` operator does with original strings
- `delete` method removes a fragment in the string, given the initial and final position to be removed.
- `insert` method adds a new string in the middle of the existing one (to be more precise, at the given starting index)
- `toString` method converts this *StringBuilder* element into a common *String*
- Besides, we have some other methods available, whose behavior is similar to the original methods in *String*: `charAt`, `length`, `indexOf`, `lastIndexOf` ...

Here you can find an example of usage:

```
StringBuilder text = new StringBuilder("Hello");
text.append(" world");                  // text = "Hello world"
text.insert(5, " my");                  // text = "Hello my world"
System.out.println(text.toString());    // "Hello my world"
System.out.println(text.indexOf("my")); // 6
text.delete(5, 8);                      // text = "Hello world"
```