DEVELHOPE

#codeforimpact

String Handling

DEVELHOPE

#codeforimpact

Strings

Java strings are *immutable objects* that represent a sequence of characters.

Strings in Java are made using the `String` objects, that come from the `java.lang` package.

A `String` object provides a lot of builtin methods for string handling.

You can create a `String` object in 2 ways:

1) using a *string literal* (a text between quotes `""`). For example, `String s1 = "Hello World";`
2) using the `new` keyword. For example, `String s1 = new String("Hello World");`

DEVELHOPE

#codeforimpact

`String` objects in memory

`String` objects are stored in the *heap*, in a dedicated area called *string constant pool.*

When you create a `String` object using a *string literal*, the *JVM* looks for the *string literal* into the *string constant pool.*

If the *string literal* is already in the *string constant pool*, then the *JVM* returns a reference to it.

If the *string literal* is not in the *string constant pool*, then the *JVM* places the new one in the *pool.*

DEVELHOPE

#codeforimpact

Concatenation operator

You have already seen the + as an arithmetic operator.

With `String` objects, the + becomes the *concatenation* operator, allowing to *concatenate* different strings.

```
String s1 = "Hello";
String s2 = "world";
String s3 = "Learn Java";

System.out.println(s1 + " " + s2 + ". " + s3 + "!");
```

This is a *string literal* representing an exclamation mark.

Prints the string
`Hello world. Learn Java!`

This is a *string literal* representing a space.

This is a *string literal* representing a dot followed by a space.

DEVELHOPE #codeforimpact

`String` methods: `charAt()`

The `String` method `charAt(int index)` returns the `char` value at the specified `index`.

Remember that first `char` value is at index `0`.

At the `index 2` of `s1` there's the first `l` character.

```
String s1 = "Hello";

System.out.println(s1.charAt(2));  // prints l

System.out.println(s1.charAt(5));  // throws an IndexOutOfBoundsException
```

`s1 index` goes from `0` to `4`, so there's no `index 5`.

DEVELHOPE

#codeforimpact

`String` methods: `compareTo()`

The `String` method `compareTo(String anotherString)` returns `0` if the argument `anotherString` is lexicographically equal to the `String` object invoking the method.

The comparison is based on the *Unicode* value of each character in the strings.

```
String s1 = "Hello";
String s2 = "World";
String s3 = "Hello";
String s4 = "world";

System.out.println(s1.compareTo(s2));   // prints -15 because s1 and s2 are different

System.out.println(s1.compareTo(s3));   // prints 0 because s1 and s2 are equal

System.out.println(s2.compareTo(s4));   // prints -32 because s2 and s4 are different
```

DEVELHOPE

#codeforimpact

`String` methods: `compareToIgnoreCase()`

The `String` method `compareToIgnoreCase(String str)` works like the `compareTo()` method, but it ignores the case differences.

```
String s1 = "Hello";
String s2 = "World";
String s3 = "Hello";
String s4 = "world";

System.out.println(s1.compareToIgnoreCase(s2));    // prints -15

System.out.println(s1.compareToIgnoreCase(s3));    // prints 0

System.out.println(s2.compareToIgnoreCase(s4));    // prints 0, ignoring the case
```

DEVELHOPE #codeforimpact

`String` methods: `concat()`

The `String` method `concat(String str)` returns the `String` made by concatenating the invoking `String` object with the `str` parameter.

```
String s1 = "Hello";
String s2 = " World";

System.out.println(s1.concat(s2)); // prints "Hello World"
```

DEVELHOPE

#codeforimpact

`String` methods: `contains()`

The `String` method `contains(CharSequence s)` returns `true` if and only if the invoking `String` object contains the specified sequence of `char` values.

```
String s1 = "Hello";

System.out.println(s1.contains("h"));    // returns false

System.out.println(s1.contains("H"));    // returns true

System.out.println(s1.contains("lo"));   // returns true
```

DEVELHOPE

#codeforimpact

`String` methods: `copyValueOf()`

The `String` method `copyValueOf(char[] data)` returns a `String` that represents the `char` sequence in the `data` array argument.

The correct way to define each `char` in the sequence is using the `'`single quotes`'`.

```
char[] charSequence = {'J', 'a', 'v', 'a'};
String finalString = "Final";

finalString = finalString.copyValueOf(charSequence);

System.out.println(finalString);        // prints Java
```

DEVELHOPE

#codeforimpact

`String` methods: `copyValueOf()`

The `String` method `copyValueOf(char[] data)` can accept other 2 parameters `offset` and `count` for returning the `char` sequence of a subarray of the `data` array argument.

The subarray is defined by `offset` (the start index of the subarray) and by `count` (the length of the subarray).

So , the method invocation can be: `copyValueOf(char[] data, int offset, int count);`

```
char[] charSequence = {'J', 'a', 'v', 'a'};
String finalString = "Final";

finalString = finalString.copyValueOf(charSequence, 1, 2);

System.out.println(finalString);        // prints av
```

DEVELHOPE #codeforimpact

String methods: endsWith()

The String method endsWith(String suffix) returns true if the String represented by the argument suffix is a suffix of the character sequence represented by the String object that invokes the method; false otherwise.

Note that the result will be true if the argument suffix is an empty String or is equal to the object that invokes the method.

```
String testString = "Hello World!";

System.out.println(testString.endsWith("Hello World!"));    // prints true
System.out.println(testString.endsWith(""));                // prints true
System.out.println(testString.endsWith("ld!"));             // prints true
System.out.println(testString.endsWith("Hello"));           // prints false
```

DEVELHOPE

#codeforimpact

`String` methods: `equals()`

The `String` method `equals(Object anObject)` compares the invoking `String` to the specified object `anObject`. The result is `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as the invoking object.

Use `equals()` if you need to compare string without considering the *Unicode* values.
Use `compareTo()` if you need to do a lexicographically comparison.

```
String s1 = "Hello";
String s2 = "World";
String s3 = "Hello";
String s4 = "world";

System.out.println(s1.equals(s2)); // prints false because s1 and s2 are different
System.out.println(s1.equals(s3)); // prints true because s1 and s2 are equal
System.out.println(s2.equals(s4)); // prints false because s2 and s4 are different
```

DEVELHOPE

#codeforimpact

String methods: equalsIgnoreCase()

The String method equalsIgnoreCase(String anotherString) works like equals(), but it ignores lower and upper case differences.

Use equalsIgnoreCase() if you need to compare string without considering the Unicode values.
Use compareToIgnoreCase() if you need to do a lexicographically comparison.

```
String s1 = "Hello";
String s2 = "World";
String s3 = "Hello";
String s4 = "world";

System.out.println(s1.equalsIgnoreCase(s2)); // prints false
System.out.println(s1.equalsIgnoreCase(s3)); // prints true
System.out.println(s2.equalsIgnoreCase(s4)); // prints true, ignoring the case
```

DEVELHOPE #codeforimpact

String methods: getBytes()

The String method getBytes() encodes and returns the invoking String into an array of byte using the platform's default charset.

getBytes can accept a charset argument: getBytes(Charset charset).

```
String s1 = "Alohà";

byte[] firstArr = s1.getBytes();   // no charset argument
for(int i=0; i< firstArr.length ; i++) {
     System.out.print(prova[i] +" ");   // prints 65 108 111 104 -61 -96
}

byte[] secondArr = s1.getBytes(Charset.forName("ASCII"));   // with charset argument
String s2 = new String(secondArr);

System.out.println(s2);  // prints Aloh? because the à char is not supported in ASCII
```

`String` methods: `getChars()`

The `String` method `getChars(int srcBegin, int srcEnd,char[] dst,int dstBegin)` copies characters from the invoking `String` into the `dst` destination character array.

- `srcBegin` is the index of the first character in the string to copy
- `srcEnd` is the index **after** the last character in the string to copy
- `dstBegin` is start offset in the destination array.

```
String s1 = "World";

char[] arr = {'H', 'e', 'l', 'l', 'o', ' ', 't', 'h', 'e', 'r', 'e' };

s1.getChars(0, 5, arr, 6);     // put World in the char array starting from dstBegin 6

System.out.println(arr); // prints Hello World
```

DEVELHOPE

#codeforimpact

`String` methods: `hashCode()`

The `String` method `hashCode()` returns a *hash code* (an `int` value) for the invoking `String`.

Hashing is useful for mapping object data to some representative `int` value.

```
String s1 = "Hello World";
String s2 = "How are you?";

int h1 = s1.hashCode();
int h2 = s2.hashCode();

System.out.println(h1);  // prints -862545276
System.out.println(h2);  // prints 1761539132
```

DEVELHOPE #codeforimpact

`String` methods: `indexOf()`

The `String` method `indexOf()` has different possible invocations:

- `indexOf(char ch)` returns the index of the **first occurrence** of the specified `ch`;
- `indexOf(char ch, int fromIndex)` returns the index of the **first occurrence** of the specified `ch`, starting the search at the specified `fromIndex` index;
- `indexOf(String subStr)` returns the index of the **first occurrence** of the specified `subStr`;
- `indexOf(String subStrint, int fromIndex)` returns the index of the **first occurrence** of the specified `subStr`, starting at the specified `fromIndex` index.

```
String s1 = "Hello, how are you? I hope you will have a good day!";

System.out.println(s1.indexOf('h'));          // prints 7
System.out.println(s1.indexOf('h', 9));       // prints 22
System.out.println(s1.indexOf("you"));        // prints 15
System.out.println(s1.indexOf("you", 22));    // prints 27
```

DEVELHOPE #codeforimpact

`String` methods: `length()`

The `String` method `length()` returns the length of the invoking `String`.

The length `int` is equal to the number of Unicode code units in the string.

```
String s1 = "Hello World";
String s2 = "How are you today?";

System.out.println(s1.length());    // prints 11
System.out.println(s2.length());    // prints 18
```

DEVELHOPE #codeforimpact

`String` methods: `isEmpty()`

The `String` method `isEmpty()` returns `true` if, and only if, `length()` is `0`.

```
String s1 = "Hello World";
String s2 = "";
String s3 = " ";

System.out.println(s1.isEmpty());  // prints false
System.out.println(s2.isEmpty());  // prints true
System.out.println(s3.isEmpty());  // prints false
```

DEVELHOPE #codeforimpact

String methods: lastIndexOf()

The `String` method `lastIndexOf()` has different possible invocations:

- `lastIndexOf(int ch)` returns the index of the **last occurrence** of the specified `ch`;
- `lastIndexOf(int ch, int fromIndex)` returns the index of the **last occurrence** of the specified `ch`, starting the search at the specified `fromIndex` index;
- `lastIndexOf(String subStr)` returns the index of the **last occurrence** of the specified `subStr`;
- `lastIndexOf(String subStrint, int fromIndex)` returns the index of the **last occurrence** of the specified `subStr`, starting at the specified `fromIndex` index.

```
String s1 = "Hello, how are you? I hope you will have a good day!";

System.out.println(s1.lastIndexOf('h'));          // prints 36
System.out.println(s1.lastIndexOf('e', 16));      // prints 13
System.out.println(s1.lastIndexOf("you"));        // prints 27
System.out.println(s1.lastIndexOf("how", 12));    // prints 7
```

DEVELHOPE

#codeforimpact

`String` methods: `replace()`

The `String` method `replace()` has different possible invocations:

- `replace(char oldChar, char newChar)` returns a new string resulting from replacing all occurrences of `oldChar` in the invoking `String` with `newChar`;
- `replace(CharSequence target, CharSequence replacement)` replaces each substring of the invoking string that matches the literal `target` sequence with the specified literal `replacement` sequence. The replacement proceeds from the beginning of the string to the end.

```
String s1 = "Hello World!";
String s2 = "Hellooo Wooorld!";

System.out.println(s1.replace('o','a'));            // Hella Warld!
System.out.println(s1.replace("oo", "a"));          // Hellao Waorld!
```

`String` methods: `replaceAll()`

The `String` method `replaceAll(String regex, String replacement)` replaces each substring of the invoking `String` that matches the given `regex` *regular expression* with the given `replacement`.

We will study more in detail *regular expressions*.

This is a `regex` for the space.

```
String s1 = "Hello World, how are you today?";

System.out.println(s1.replaceAll("\\s","")); // prints HelloWorld,howareyoutoday?

System.out.println(s1.replace("are you", "is your cat"));
// prints Hello World, how is your cat today?
```

DEVELHOPE

#codeforimpact

String methods: replaceFirst()

The String method replaceFirst(String regex, String replacement) replaces the first substring of the invoking String that matches the given regex *regular expression* with the given replacement.

Here there are two spaces

```
String s1 = "Hello  World";

System.out.println(s1.replaceFirst("\\s","")); // prints Hello World with just one space
```

DEVELHOPE

#codeforimpact

`String` methods: `split()`

The `String` method `split()` has different possible invocations:

- `split(String regex)` splits the invoking `String` around matches of the given *regular expression*;
- `split(String regex, int limit)` like before, but the `limit` parameter controls the number of times the pattern is applied and therefore affects the `length` of the resulting array.

```
String s1 = "How are you today?";

String[] newArray = s1.split("\\s");

System.out.println(Arrays.toString(newArray));          // prints [How, are, you, today?]
```

We use the `Arrays` class coming from `java.util.Arrays`

DEVELHOPE

#codeforimpact

`String` methods: `startsWith()`

The `String` method `startsWith()` has different possible invocations:

- `startsWith(String prefix)` returns `true` if the invoking `String` starts with the given `prefix`;
- `startsWith(String prefix, int toffset)` returns `true` if the invoking `String` starts with the given `prefix`, beginning at the specified `toffset` index.

```
String s1 = "How are you today?";

System.out.println(s1.startsWith("How"));          // prints true

System.out.println(s1.startsWith("you", 8));       // prints true

System.out.println(s1.startsWith("you", 7));       // prints false
```

`String` methods: `substring()`

The `String` method `substring()` has different possible invocations:

- `substring(int beginIndex)` returns a new string that is a substring of the invoking `String` and `beginIndex` is inclusive;
- `substring(int beginIndex, int endIndex)` like before, but the substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`.

```
String s1 = "How are you today?";

System.out.println(s1.substring(4));          // prints are you today?

System.out.println(s1.substring(4, 7));       // prints are
```

DEVELHOPE #codeforimpact

String methods: toCharArray()

The String method toCharArray() converts the invoking String to a new array of char.

```
String s1 = "Massachusetts";

char[] charArray = s1.toCharArray();

for(int i=0; i< charArray.length ; i++) {
    System.out.print(charArray[i] +" ");     // prints M a s s a c h u s e t t s
}
```

`String` methods: `toLowerCase()`

The `String` method `toLowerCase()` converts all of the characters in the invoking `String` to lower case.

```
String s1 = "HELLO!";
String s2 = "Hello!";

System.out.println(s1.toLowerCase());   // prints hello!

System.out.println(s2.toLowerCase());   // prints hello!
```

DEVELHOPE #codeforimpact

`String` methods: `toUpperCase()`

The `String` method `toUpperCase()` converts all of the characters in the invoking `String` to upper case.

```
String s1 = "hello!";
String s2 = "Hello!";

System.out.println(s1.toUpperCase());   // prints HELLO!

System.out.println(s2.toUpperCase());   // prints HELLO!
```

DEVELHOPE

#codeforimpact

`String` methods: `trim()`

The `String` method `trim()` returns a copy of the `String`, with leading and trailing whitespace omitted.

```
String s1 = " Hello! ";

System.out.println(s1.length());    // prints 8

String trimmedString = s1.trim();

System.out.println(trimmedString); // prints Hello!
```

DEVELHOPE #codeforimpact

`String` methods: `valueOf()`

The `String` method `valueOf(`**`type`**` param)` returns the string representation of the type argument.

`Type` can be `boolean`, `char`, `char[]`, `double`, `float`, `int`, `long` and `Object`.

```
boolean boolParam = true;
double doubleParam = 12.45;
char[] charArrayParam = {'h','e','l','l','o'};

System.out.println(String.valueOf(boolParam));          // prints a String "true"
System.out.println(String.valueOf(doubleParam));        // prints a String "12.45"
System.out.println(String.valueOf(charArrayParam));     // prints a String "hello"
```

DEVELHOPE

#codeforimpact

String Handling