String

Класс: `java.lang.String`

Описание

Представляет собой строку символов. Если точнее, то элемент строки (`char`) представляет собой кодовую единицу кодировки UTF-16. В пределах базовой многоязыковой плоскости (BMP, U+0000 – U+FFFF) символ представляется одним `char`'ом, в остальных плоскостях (U+10000 – U+1FFFF) — двумя.

Создание

Обычно в пользовательском коде объекты `String` не создаются явно, а возвращаются как результат выполнения других операций. Все строковые константы в Java (например, `"Hello World!"`) имеют тип `String`. Кроме того, строки возвращаются при построении строкового представления объекта (`Object.toString`), как результат соединения строк (операция + и объект `StringBuilder`), а также при работе с текстовыми потоками (`Scanner` и `Reader`).

Кроме того, для создания строк полезно семейство фабричных методов `valueOf`:

`static String valueOf(<любой_тип> value)`

Для значения `null` этот метод возвращает строку `"null"`, для объектов — результат `value.toString()`, а для числовых типов и типа `boolean` — тот же результат, что и при прибавлении их к строке оператором +:

```
String.valueOf(null)            // "null"
String.valueOf(LocalDate.now()) // то же, что и LocalDate.now().toString()
                                // например, "2016-02-24"
String.valueOf(5)               // "5"
String.valueOf(5L)              // "5"
String.valueOf(true)            // "true"
```

Важно! Массивы типа `char[]` можно преобразовать в `String` только через конструктор `String(char[])` или метод `valueOf`. Вызов `toString` или попытка присоединить массив к строке оператором `+` выдаст совсем не то, что нужно:

```
char[] array = {'H', 'e', 'l', 'l', 'o'};
System.out.println(String.valueOf(array));
System.out.println(new String(array));
```

```
System.out.println(array.toString());
System.out.println(array + " World");
```

В результате выведется что-то вроде:

```
Hello
Hello
[C@279f2327
[C@279f2327 World
```

Байты и символы

Строку можно раскодировать из массива байт, указав нужную кодировку в конструкторе:

`String(byte[] bytes, Charset charset)`

Например:

```
byte[] bytes = Files.readAllBytes(Paths.get("C:\\test.txt"));
String str = new String(bytes, StandardCharsets.UTF_8);
```

Для обратного преобразования из строки в массив байт тоже нужно указать кодировку:

`byte[] getBytes(Charset charset)`

Например:

```
byte[] bytes = "Hello".getBytes(StandardCharsets.UTF_8);
Files.write(Paths.get("C:\\test.txt"), bytes);
```

И, наконец, метод

`char[] toCharArray()`

возвращает копию символов, содержащихся в строке, в виде массива. Именно копию; при изменении полученного массива исходная строка не изменится.

```
String str = "Hello";
char[] array = str.toCharArray();
array[4] = 's';

System.out.println(str);
System.out.println(array);
System.out.println(String.valueOf(array));
```

Результат:

```
Hello
Hells
Hells
```

Сравнение

Никогда не используйте оператор == для сравнения строк. Он возвращает `true` только в том случае, если оба сравниваемых выражения указывают на один и тот же объект. Для сравнения строк по

значению, а не по ссылке, используйте `equals`:

`boolean equals(Object other)`

Например:

```
"2" == String.valueOf(2)      // false; не делайте так!
"2".equals(String.valueOf(2)) // true
```

Важно! При сравнении со строковой константой принято использовать идиому, при которой метод `equals` вызывается для константы, особенно если строка получена из внешнего кода:

```
"Hello".equals(myString)  // правильно
myString.equals("Hello")  // неправильно
```

Это связано с тем, что если `myString == null`, то попытка вызвать для этой переменной любой метод выбросит исключение `NullPointerException`. В то же время `"Hello".equals(null)` вернёт `false`, не выбросив исключения.

Чтобы упорядочить строки в лексикографическом (то есть алфавитном, словарном) порядке, применяется метод

`int compareTo(String other)`

Для сравнения символов используются коды Unicode, например, `'D' < 'd'`, а также `'d' < 'Д'`. Метод `compareTo` возвращает:

- значение меньше 0, если эта строка меньше переданной строки
- 0, если строки равны
- значение больше 0, если эта строка больше переданной строки

Поэтому, чтобы проверить условие `str1 *op* str2`, где `*op*` - это `<`, `>` или `==`, нужно записать `str1.compareTo(str2) *op* 0`.

Пример:

```
"ABC".compareTo("ABCD") < 0  // true
"ABC".compareTo("ABC") == 0  // true
"ABC".compareTo("ABAB") > 0  // true
```

Регистр

У `equals` и `compareTo` также есть версии, сравнивающие строки без учёта регистра:

`boolean equalsIgnoreCase(String other)`

`int compareToIgnoreCase(String other)`

Например:

```
"ABC".equalsIgnoreCase("abc")    // true
"ABC".compareToIgnoreCase("abc") // 0
```

А если нужно преобразовать регистр в верхний или нижний, используются методы

`String toUpperCase()`

`String toLowerCase()`

Например:

```
"Hello".toUpperCase()  // "HELLO"
"Hello".toLowerCase()  // "hello"
```

Важно! Класс `String` является *неизменяемым*, поэтому любые операции преобразования строки возвращают новую строку, не изменяя ту, для которой был вызван метод. Например:

```
String str1 = "HELLO";
str1.toLowerCase();       // уважайте труд сборщика, не мусорьте :(
System.out.println(str1); // str1 не изменилась, так что HELLO
String str2 = str1.toLowerCase();
System.out.println(str2); // hello
```

Длина и индексы

Длина строки — это количество элементов `char` в ней.

`int length()`

Например:

```
"Hello".length()            // 5
"Привет".length()           // 6
"Hello\n".length()          // 6
"Hello\u0020World".length() // 11
"C:\\".length()             // 3
```

Обратите внимание, что escape-последовательности типа `\\`, `\n` и `\u0020` кодируют один `char` каждая, несмотря на то, что записываются в Java-коде несколькими символами.

Элементы `char` в строке имеют *индекс*, изменяющийся от `0` (первый символ) до `length() - 1` (последний символ). Чтобы получить `char`, находящийся в строке по определённому индексу, используется метод

`char charAt(int index)`

Например, следующий код печатает каждый `char` на отдельной строке:

```
String str = "Hello";

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

Строка нулевой длины называется *пустой строкой*. Проверить, является ли строка пустой, можно с помощью метода

`boolean isEmpty()`

Он аналогичен сравнению `length() == 0`.

Важно! Пустая строка `""` отличается от значения `null`. Если строковой переменной

присвоено значение `null`, то она не ссылается ни на один объект, и попытка вызвать для неё любой метод выбросит `NullPointerException`:

```
String str1 = "";
String str2 = null;
str1.length()  // 0
str2.length()  // выбрасывает NullPointerException
```

Иногда бывает полезно проверить, что строковая переменная не содержит ни `null`, ни пустую строку. Например, в серверных приложениях значение `null` может означать, что в строке запроса определённый параметр отсутствует, а пустая строка означает, что параметр присутствует и пуст:

```
http://example.com/article?name=Java
    request.getParameter("version") -> null

http://example.com/article?name=Java&version=
    request.getParameter("version") -> ""

http://example.com/article?name=Java&version=4
    request.getParameter("version") -> "4"
```

В этом случае можно использовать следующую идиому:

```
String version = request.getParameter("version");

if (version == null || version.isEmpty()) {
    // код обработки пустого параметра
}
```

Подстроки

Чтобы выделить из строки подстроку, используется семейство методов `substring`:

`String substring(int beginIndex)`

`String substring(int beginIndex, int endIndex)`

Здесь `beginIndex` — это индекс первого символа подстроки, а `endIndex` — индекс символа, *следующего за последним*. При этом всегда выполняется равенство `beginIndex + длина_подстроки == endIndex`. Обратите внимание, что индексы начинаются с 0.

Если `endIndex` не задан (первая форма метода `substring`), то возвращается подстрока от `beginIndex` до конца строки.

Например:

```
"Hello".substring(2);     // "llo"
"Hello".substring(2, 5);  // "llo"; то же самое
"Hello".substring(1, 2);  // "e"
"Hello".substring(5, 5);  // ""; вырожденный случай
```

Следующий код печатает все непустые подстроки строки `"Hello"`, начинающиеся с первого символа `H`:

```
String str = "Hello";

for (int endIndex = 1; endIndex <= str.length(); endIndex++) {
    System.out.println(str.substring(0, endIndex));
}
```

Результат:

```
H
He
Hel
Hell
Hello
```

Важно! Будьте внимательны с граничными условиями циклов. Если бы в предыдущем коде вместо `<=` было написано `<`, то строка `Hello` не вывелась бы, и вывод остановился бы на строке `Hell`. (Почему?)

Поиск

Семейство методов `indexOf` ищет первое вхождение указанного символа в строке. Если вместо символа указать строку, то вернётся индекс начала первой соответствующей подстроки в строке. Можно также указать индекс, с которого начинать поиск (по умолчанию — с начала строки).

`int indexOf(int ch)`

`int indexOf(int ch, int fromIndex)`

`int indexOf(String str)`

`int indexOf(String str, int fromIndex)`

Если символ или строка не найдены, возвращается -1. Например:

```
"Hello".indexOf('l')     // 2
"Hello".indexOf('g')     // -1
"Hello".indexOf('H', 1)  // -1
"Hello".indexOf("ell")   // 1
```

Есть ещё полностью аналогичное семейство `lastIndexOf`, которое ищет не с начала, а с конца (но для строкового параметра всё равно возвращает индекс начала подстроки).

```
"Hello".lastIndexOf('l')     // 3
"Hello".lastIndexOf("ello")  // 1
```

Если нужно только проверить, содержит ли строка подстроку, можно использовать метод `contains`:

`boolean contains(CharSequence s)`

Например:

```
"Hello".contains("ell")  // true; аналогично "Hello".indexOf("ell") != -1
```

Также можно проверить, начинается или заканчивается ли строка указанной подстрокой:

`boolean startsWith(String prefix)`

`boolean endsWith(String suffix)`

Например:

```
"Hello".startsWith("He")   // true
"Hello".endsWith("World")  // false
```

Замена

Для замены символов или подстрок служит семейство методов `replace`:

`String replace(char oldChar, char newChar)`

`String replace(CharSequence target, CharSequence replacement)`

Например:

```
System.out.println("Hello".replace("lo", "p"));  // Help
```

Если строка или символ встретились несколько раз, то заменяются все вхождения:

```
"Hello".replace("l", "")  // Heo
```

Опять же, методы `replace` возвращают новую строку, не изменяя ту, для которой они вызывались.

Есть ещё методы `replaceAll` и `replaceFirst`, которые в качестве выражений для поиска и замены принимают не обычные строки, а *регулярные выражения*. С ними мы познакомимся, изучив классы `Pattern` и `Matcher`. Если вам нужно заменить именно обычные строки, не используйте эти методы, иначе результат будет не тем, что вы ожидали:

```
String str = "Ночь. Улица. Фонарь. Аптека.";
System.out.println("replace: " + str.replace(".", "!"));
System.out.println("replaceAll: " + str.replaceAll(".", "!"));
```

Результат:

```
replace: Ночь! Улица! Фонарь! Аптека!
replaceAll: !!!!!!!!!!!!!!!!!!!!!!!!!!
```