# Обработка строк в Java. Часть I: String, StringBuffer, StringBuilder из песочницы tutorial

JAVA*

## Вступление

Что вы знаете о обработке строк в Java? Как много этих знаний и насколько они углублены и актуальны? Давайте попробуем вместе со мной разобрать все вопросы, связанные с этой важной, фундаментальной и часто используемой частью языка. Наш маленький гайд будет разбит на две публикации:

1. String, StringBuffer, StringBuilder (реализация строк)
2. Pattern, Matcher (регулярные выражения)

Реализация строк на Java представлена тремя основными классами: **String**, **StringBuffer**, **StringBuilder**. Давайте поговорим о них.

## String

Строка — объект, что представляет последовательность символов. Для создания и манипулирования строками Java платформа предоставляет общедоступный финальный (не может иметь подклассов) класс **java.lang.String**. Данный класс является неизменяемым (*immutable*) — созданный объект класса **String** не может быть изменен. Можно подумать что методы имеют право изменять этот объект, но это неверно. Методы могут только создавать и возвращать новые строки, в которых хранится результат операции. Неизменяемость строк предоставляет ряд возможностей:

- использование строк в многопоточных средах (**String** является потокобезопасным (thread-safe) )
- использование **String Pool** (это коллекция ссылок на **String** объекты, используется для оптимизации памяти)
- использование строк в качестве ключей в **HashMap** (ключ рекомендуется делать неизменяемым)

### Создание

Мы можем создать объект класса **String** несколькими способами:

**1. Используя строковые литералы:**

```
String habr = "habrahabr";
```

Строковый литерал — последовательность символов заключенных в двойные кавычки. Важно понимать, что всегда когда вы используете строковой литерал компилятор создает объект со значением этого литерала:

```
System.out.print("habrahabr"); // создали объект и вывели его значение
```

**2. С помощью конструкторов:**

```
String habr = "habrahabr";
char[] habrAsArrayOfChars = {'h', 'a', 'b', 'r', 'a', 'h', 'a', 'b', 'r'};
byte[] habrAsArrayOfBytes = {104, 97, 98, 114, 97, 104, 97, 98, 114};

String first = new String();
String second = new String(habr);
```

Если копия строки не требуется явно, использование этих конструкторов нежелательно и в них нет необходимости, так как строки являются неизменными. Постоянное строительство новых объектов таким способом может привести к снижению производительности. Их лучше заменить на аналогичные инициализации с помощью строковых литералов.

```
String third = new String(habrAsArrayOfChars); // "habrahabr"
String fourth = new String(habrAsArrayOfChars, 0, 4); // "habr"
```

Конструкторы могут формировать объект строки с помощью массива символов. Происходит копирование массива, для этого используются статические методы **copyOf** и **copyOfRange** (копирование всего массива и его части (если указаны 2-й и 3-й параметр конструктора) соответственно) класса **Arrays**, которые в свою очередь используют платформо-зависимую реализацию **System.arraycopy**.

```
String fifth = new String(habrAsArrayOfBytes, Charset.forName("UTF-16BE"));
// кодировка нам явно не подходит "桦抮慨慢"
```

Можно также создать объект строки с помощью массива байтов. Дополнительно можно передать параметр класса **Charset**, что будет отвечать за кодировку. Происходит декодирование массива с помощью указанной кодировки (если не указано — используется **Charset.defaultCharset()**, который зависит от кодировки операционной системы) и, далее, полученный массив символов копируется в значение объекта.

```
String sixth = new String(new StringBuffer(habr));
String seventh = new String(new StringBuilder(habr));
```

Ну и наконец-то конструкторы использующие объекты **StringBuffer** и **StringBuilder**, их значения (**getValue()**) и длину (**length()**) для создания объекта строки. С этими классами мы познакомимся чуть позже.

Приведены примеры наиболее часто используемых конструкторов класса **String**, на самом деле их пятнадцать (два из которых помечены как *deprecated*).

**Длина**

Важной частью каждой строки есть ее длина. Узнать ее можно обратившись к объекту **String** с помощью метода доступа (*accessor method*) **length()**, который возвращает количество символов в строке, например:

```
public static void main(String[] args) {
    String habr = "habrahabr";
    // получить длину строки
    int length = habr.length();
    // теперь можно узнать есть ли символ символ 'h' в "habrahabr"
    char searchChar = 'h';
    boolean isFound = false;
    for (int i = 0; i < length; ++i) {
        if (habr.charAt(i) == searchChar) {
            isFound = true;
            break; // первое вхождение
        }
    }
    System.out.println(message(isFound)); // Your char had been found!
    // ой, забыл, есть же метод indexOf
    System.out.println(message(habr.indexOf(searchChar) != -1)); // Your char
had been found!
}

private static String message(boolean b) {
    return "Your char had" + (b ? " " : "n't ") + "been found!";
}
```

## Конкатенация

Конкатенация — операция объединения строк, что возвращает новую строку, что есть результатом объединения второй строки с окончанием первой. Операция для объекта **String** может быть выполнена двумя способами:

#### 1. Метод concat

```
String javaHub = "habrhabr".concat(".ru").concat("/hub").concat("/java");
System.out.println(javaHub); // получим "habrhabr.ru/hub/java"
// перепишем наш метод используя concat
private static String message(boolean b) {
    return "Your char had".concat(b ? " " : "n't ").concat("been found!");
}
```

Важно понимать, что метод **concat** не изменяет строку, а лишь создает новую как результат слияния текущей и переданной в качестве параметра. Да, метод возвращает новый объект String, поэтому возможны такие длинные «цепочки».

#### 2. Перегруженные операторы "+" и "+="

```
String habr = "habra" + "habr"; // "habrahabr"
habr += ".ru"; // "habrahabr.ru"
```

Это одни с немногих перегруженных операторов в Java — язык не позволяет перегружать

операции для объектов пользовательских классов. Оператор "+" не использует метод **concat**, тут используется следующий механизм:

```
String habra = "habra";
String habr = "habr";
// все просто и красиво
String habrahabr = habra + habr;
// а на самом деле
String habrahabr = new
StringBuilder()).append(habra).append(habr).toString(); // может быть
использован StringBuffer
```

Используйте метод **concat**, если слияние нужно провести только один раз, для остальных случаев рекомендовано использовать или оператор "+" или **StringBuffer** / **StringBuilder**. Также стоит отметить, что получить NPE (**NullPointerException**), если один с операндов равен **null**, невозможно с помощью оператора "+" или "+=", чего не скажешь о методе **concat**, например:

```
String string = null;
string += " habrahabr"; // null преобразуется в "null", в результате "null
habrahabr"
string = null;
string.concat("s"); // логично что NullPointerException
```

**Форматирование**

Класс **String** предоставляет возможность создания форматированных строк. За это отвечает статический метод **format**, например:

```
String formatString = "We are printing double variable (%f), string ('%s')
and integer variable (%d).";
System.out.println(String.format(formatString, 2.3, "habr", 10));
// We are printing double variable (2.300000), string ('habr') and integer
variable (10).
```

**Методы**

Благодаря множеству методов предоставляется возможность манипулирования строкой и ее символами. Описывать их здесь нет смысла, потому что Oracle имеет хорошие статьи о манипулировании и сравнении строк. Также у вас под рукой всегда есть их документация. Хотелось отметить новый статический метод **join**, который появился в Java 8. Теперь мы можем удобно объединять несколько строк в одну используя разделитель (был добавлен класс **java.lang.StringJoiner**, что за него отвечает), например:

```
String hello = "Hello";
String habr = "habrahabr";
String delimiter = ", ";

System.out.println(String.join(delimiter, hello, habr));
// или так
System.out.println(String.join(delimiter, new
ArrayList<CharSequence>(Arrays.asList(hello, habr))));
// в обоих случаях "Hello, habrahabr"
```

Это не единственное изменение класса в Java 8. Oracle сообщает о улучшении производительности в конструкторе **String(byte[], *)** и методе **getBytes()**.

### Преобразование

#### 1. Число в строку

```
int integerVariable = 10;
String first = integerVariable + ""; // конкатенация с пустой строкой
String second = String.valueOf(integerVariable); // вызов статического метода
valueOf класса String
String third = Integer.toString(integerVariable); // вызов метода toString
класса-обертки
```

#### 2. Строку в число

```
String string = "10";
int first = Integer.parseInt(string);
/*
   получаем примитивный тип (primitive type)
   используя метод parseXxx нужного класса-обертки,
   где Xxx - имя примитива с заглавной буквы (например parseInt)
*/
int second = Integer.valueOf(string); // получаем объект wrapper класса и
автоматически распаковываем
```

## StringBuffer

Строки являются неизменными, поэтому частая их модификация приводит к созданию новых объектов, что в свою очередь расходует драгоценную память. Для решения этой проблемы был создан класс **java.lang.StringBuffer**, который позволяет более эффективно работать над модификацией строки. Класс является *mutable*, то есть изменяемым — используйте его, если Вы хотите изменять содержимое строки. **StringBuffer** может быть использован в многопоточных средах, так как все необходимые методы являются синхронизированными.

### Создание

Существует четыре способа создания объекта класса **StringBuffer**. Каждый объект имеет свою вместимость (*capacity*), что отвечает за длину внутреннего буфера. Если длина строки, что хранится в внутреннем буфере, не превышает размер этого буфера (*capacity*), то нет необходимости выделять новый массив буфера. Если же буфер переполняется — он автоматически становиться больше.

```
StringBuffer firstBuffer = new StringBuffer(); // capacity = 16
StringBuffer secondBuffer = new StringBuffer("habrahabr"); // capacity =
str.length() + 16
```

```
StringBuffer thirdBuffer = new StringBuffer(secondBuffer); // параметр -
любой класс, что реализует CharSequence
StringBuffer fourthBuffer = new StringBuffer(50); // передаем capacity
```

### Модификация

В большинстве случаев мы используем **StringBuffer** для многократного выполнения операций добавления (*append*), вставки (*insert*) и удаления (*delete*) подстрок. Тут все очень просто, например:

```
String domain = ".ru";
// создадим буфер с помощью String объекта
StringBuffer buffer = new StringBuffer("habrahabr"); // "habrahabr"
// вставим домен в конец
buffer.append(domain); // "habrahabr.ru"
// удалим домен
buffer.delete(buffer.length() - domain.length(), buffer.length()); //
"habrahabr"
// вставим домен в конец на этот раз используя insert
buffer.insert(buffer.length(), domain); // "habrahabr.ru"
```

Все остальные методы для работы с **StringBuffer** можно посмотреть в документации.

## StringBuilder

**StringBuilder** — класс, что представляет изменяемую последовательность символов. Класс был введен в Java 5 и имеет полностью идентичный API с **StringBuffer**. Единственное отличие — **StringBuilder** не синхронизирован. Это означает, что его использование в многопоточных средах есть нежелательным. Следовательно, если вы работаете с многопоточностью, Вам идеально подходит **StringBuffer**, иначе используйте **StringBuilder**, который работает намного быстрее в большинстве реализаций. Напишем небольшой тест для сравнения скорости работы этих двух классов:

```
public class Test {
    public static void main(String[] args) {
        try {
            test(new StringBuffer("")); // StringBuffer: 35117ms.
            test(new StringBuilder("")); // StringBuilder: 3358ms.
        } catch (java.io.IOException e) {
            System.err.println(e.getMessage());
        }
    }
    private static void test(Appendable obj) throws java.io.IOException {
        // узнаем текущее время до теста
        long before = System.currentTimeMillis();
        for (int i = 0; i++ < 1e9; ) {
            obj.append("");
        }
        // узнаем текущее время после теста
        long after = System.currentTimeMillis();
        // выводим результат
        System.out.println(obj.getClass().getSimpleName() + ": " + (after -
before) + "ms.");
    }
}
```