String. Работа со строками

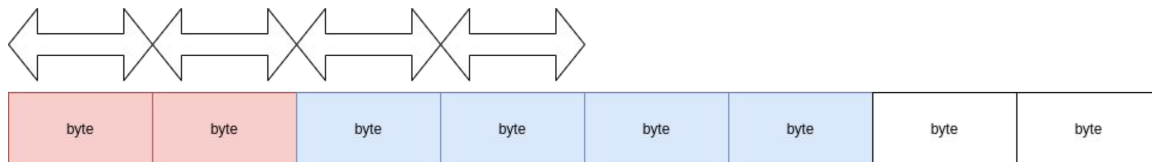
До Java 9

```
private final char[] value;  // Unicode (UTF-16)
```

После Java 9

```
private final byte[]
value;
private final byte coder;  // ASCII (Latin-1) - 1 byte / UTF-16 - 2/4 bytes
```

Latin-1



UTF-16

Особенности класса String

- это immutable (неизменный) класс
- это final класс

Это дает классу String несколько важных преимуществ:

1. Благодаря неизменности, хэшкод экземпляра класса String кэшируется. Его не нужно вычислять каждый раз, потому что значения полей объекта никогда не изменятся после его создания. Это дает высокую производительность при использовании данного класса в качестве ключа для `HashMap`.
2. Класс String можно использовать в многопоточной среде без дополнительной синхронизации.
3. Еще одна особенность класса String — для него перегружен оператор "+" в Java. Поэтому конкатенация (сложение) строк выполняется довольно просто:

```
String command = "Follow" + " " + "the" + " " + "white" + " " + "rabbit";
```

№	Методы с описанием
1	char charAt(int index) Возвращает символ по указанному индексу.
2	int compareTo(Object o) Сравнивает данную строку с другим объектом.
3	int compareTo(String anotherString) Сравнивает две строки лексически.
4	int compareToIgnoreCase(String str) Сравнивает две строки лексически, игнорируя регистр букв.
5	String concat(String str) Объединяет указанную строку с данной строкой, путем добавления ее в конце.
6	boolean contentEquals(StringBuffer sb) Возвращает значение true только в том случае, если эта строка представляет собой ту же последовательность символов как указанно в буфере строки (StringBuffer).
7	static String copyValueOf(char[] data) Возвращает строку, которая представляет собой последовательность символов, в указанный массив.
8	static String copyValueOf(char[] data, int offset, int count) Возвращает строку, которая представляет собой последовательность символов, в указанный массив.
9	boolean endsWith(String suffix) Проверяет заканчивается ли эта строка указанным окончанием.
10	boolean equals(Object anObject) Сравнивает данную строку с указанным объектом.

String Pool

Созданы литералы в пуле

```
String nameFirst = "Bohdan";
String nameSecond = "Bohdan";

assert nameFirst == nameSecond; // true
```

2) Созданы объекты вне пула

```
String nameFirst = new String("Bohdan");
String nameSecond = new String("Bohdan");
String nameThird = "Bohdan";

assert nameFirst == nameSecond; // false
```

Помещение строкового объекта в пул

```
String pooledNameFirst = nameFirst.intern();
```

java Main -ea

StringBuffer - mutable String

```
StringBuffer sb2 = new StringBuffer("Not empty");


StringBuffer sb = new StringBuffer();

 sb.append(new Integer(2))
         .append("; ")
         .append(false)
         .append("; ")
         .append(Arrays.asList(1,2,3))
         .append("; ");

 System.out.println(sb); // 2; false; [1, 2, 3];
```

Основные, но не единственные методы

- `delete(int start, int end)` — удаляет подстроку символов начиная с позиции `start`, заканчивая `end`
- `deleteCharAt(int index)` — удаляет символ в позиции `index`
- `insert(int offset, String str)` — вставляет строку `str` в позицию `offset`. Метод `insert` также перегружен и может принимать различные аргументы
- `replace(int start, int end, String str)` — заменит все символы начиная с позиции `start` до позиции `end` на `str`
- `reverse()` — меняет порядок всех символов на противоположный
- `substring(int start)` — вернет подстроку, начиная с позиции `start``substring(int start, int end)` — вернет подстроку, начиная с позиции `start` до позиции `end`

String vs StringBuffer vs StringBuilder

	String	StringBuffer	StringBuilder
Изменяемость	`Immutable` (нет)	`mutable` (да)	`mutable` (да)
Расширяемость	`final` (нет)	`final` (нет)	`final` (нет)
Потокобезопасность	Да, за счет неизменяемости	Да, за счет синхронизации	Нет
Когда использовать	При работе со строками, которые редко будут модифицироваться	При работе со строками, которые часто будут модифицироваться в многопоточной среде	При работе со строками, которые часто будут модифицироваться, в однопоточной среде

Сравнение с помощью compareTo()

1) Сравнение одинаковых по длине строк:

Сравнение идет по индексу начиная от начала. Если найдена пара отличающихся символов по индексу K - результатом сравнения будет сравнения двух s1.charAt(K) и s2.charAt(K)

1) Сравнение разных по длине строк:

Результатом compareTo() будет разница между длиной первой и второй строки

Regex (поиск совпадений)

```
new RegExp("^(((?=.*[a-z])(?=.*[A-Z]))|((?=.*[a-z])(?=.*[0-9]))|((?=.*[A-Z])(?=.*[0-9])))(?=.{6,})")
```

1) “string”.matches(String pattern);
2) java.util.regex.Matcher & java.util.regex.Pattern

abc -> “abc”

a+ -> “a”, “aaaa”

[abc] -> “a” , “b”, “c"

[^abc] -> НЕ “a” , “b”, “c”

[а-яА-ЯїЇ] -> “а” , “б”, “Ї"

[а-яА-ЯїЇ]* -> “”, “а”, “аааа”

[а-яА-ЯїЇ]+ -> “а”, “аааа”

[а-яА-ЯїЇ]? -> “”, “а”

[abc]{2,8} -> “ac” .. “abcabcab”

[abc] {2,} -> “ac” .. “abcabcabadasdas”

[abc]{2} -> “ac”

a|b -> “a, “b”

“обговорить”.matches(“`(?i)(?<=об)говорить`”) //true

“говорить”.matches(“`(?<=об)говорить`”) //true

Подвыражение	Обозначение
^	Соответствует началу строки.
$	Соответствует концу строки.
.	Соответствует любому одиночному символу, за исключением новой строки. Использование опции m делает возможным соответствие новой строке.
[...]	Соответствует любому одиночному символу в квадратных скобках.
[^...]	Соответствует любому одиночному символу вне квадратных скобок.
\A	Начало целой строки.
\z	Конец целой строки.
\Z	Конец целой строки, за исключением допустимого терминатора конца строки.
re*	Соответствует 0 либо более вхождений предыдущего выражения.
re+	Соответствует 1 либо более вхождений предыдущего выражения.
re?	Соответствует 0 либо 1 вхождению предыдущего выражения.
re{ n}	Соответствует заданному n числу вхождений предыдущего выражения.
re{ n,}	Соответствует n или большему числу вхождений предыдущего выражения.
re{ n, m}	Соответствует n как минимум и m в большинстве вложений предыдущего выражения.
a| b	Соответствует a или b.
(re)	Группирует регулярные выражения и запоминает сравниваемый текст.

Режимы

Режимы указываются в начале регулярного выражения. Можно комбинировать несколько режимов записывая их последовательно (?ismx).

- (?i) не чувствительность к регистру
- (?s) for "single line mode" символ точки так же указывает на символы переноса строки
- (?m) for "multi-line mode" символы ^ и $ указывают на начало и конец каждой строки многострочного текста
- (?x) single spacing mode - пробелы в регулярных выражениях игнорируются

Группирование - ()

```
(\d+)(abc)zzz
```

группа 0 - все выражение
группа 1 - подвыражение \d+
группа 2 - подвыражение abc

```
static void regexExample3() {
    String EXAMPLE_TEST = "Text .";
    String pattern = "(\\w)(\\s+)([\\.,])";
    System.out.println(EXAMPLE_TEST.replaceAll(pattern , "$1$3"));
}

// 1 - Text
// 2 - " "
// 3 - .
```

() Простая группа с захватом.

1) **(?:) -** Группа без захвата. То же самое, но заключённое в скобках выражение не добавляется к списку захваченных фрагментов. Например, если требуется найти или «здравствуйте», или «здраститя», но не важно, какое именно приветствие найдено, можно воспользоваться выражением `здра(?:ститя|вствуйте)`.

2) **(?=) -** Группа с положительной опережающей проверкой (positive lookahead assertion). Продолжает поиск только если справа от текущей позиции в тексте находится заключённое в скобки выражение. При этом само выражение не захватывается. Например, **`говор(?=ить)`** найдёт «говор» в «говорить», но не в «говорит». Иными словами, ищет в строке «говор», после которого сразу идут символы «ить» — если находит, выдает истину, иначе — ложь (FALSE).

3) **(?!) -** Группа с отрицательной опережающей проверкой (negative lookahead assertion). Продолжает поиск только если справа от текущей позиции в тексте не находится заключённое в скобки выражение. При этом само выражение не захватывается. Например, **`говор(?!ить)`** найдёт «говор» в «говорит», но не в «говорить».

4) **(?<=) -** Группа с положительной ретроспективной проверкой (positive lookbehind assertion). Продолжает поиск только если слева от текущей позиции в тексте находится заключённое в скобки выражение. При этом само выражение не захватывается. Например, **`(?<=об)говорить`** найдёт «говорить» в «обговорить», но не в «уговорить».

5) **(?<!) -** Группа с отрицательной ретроспективной проверкой (negative lookbehind assertion). Продолжает поиск только если слева от текущей позиции в тексте не находится заключённое в скобки выражение. При этом само выражение не захватывается. Например, **`(?<!об)говорить`** найдёт «говорить» в «уговорить», но не в «обговорить».

Исключение последовательности

`(?!pattern)`

```
a(?!b)
```

ac, ad, a3, но не ab

Именованные группы

```
static void regexExample3 () {
    String EXAMPLE_TEST =  "Text .";
    String pattern = "(?<group1>\\w)(?<group2>\\s+)(?<group3>[\\.,])";
    System.out.println(EXAMPLE_TEST.replaceAll(pattern , "${group1}${group3}"));
}
```

\k<name>

```
static void regexExample4 () {
    String pattern = "(?<group1>\\[a-zA-R])(?<group2>\\s+)\\k<group1>";
}
```

Поиск по всему тексту (?=.*)

(?=.{8,}) Строка должна быть длинной 8 символов или более

(?=.*[0-9]) Строка должна состоять как минимум с 1 числового символа

“Мой номер телефона 03”

(?=.*[0-9])(?=.*[а-яА-Я])^Мой.+$

\A

Пример. Валидация IP v4

```
\b(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.
  (25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.
  (25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.
  (25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\b
```

\.
\?
\(
\)

```
\b192\.168\.
  (25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.
  (25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\b
```

«Жадные» выражения

a+ , a*, a? , a{n} - жадные выражения (по умолчанию)

a+?, a*?, a{n}? - не жадный (ленивый) эквивалент

a++, a*+, a{n}+ - сверх жадные выражения

asdTTasdasTTT

\wT++

Методы класса Matcher

В классе Matcher есть ряд методов, для определения места совпадения.

Вот эти методы:

public int start()
Возврат начального индекса к предыдущему совпадению.

public int start(int group)
Возврат начального индекса к последовательности, захваченной данной группой в течение предыдущей операции установления соответствия.

public int end()
Возврат позиции смещения следом за последним совпадающим символом.

public String group()
Возврат совпадающей группы символов

public int end(int group)
Возврат позиции смещения следом за последним символом к последовательности, захваченной данной группой в течение предыдущей операции установления соответствия.

Методы поиска

Методы поиска предназначены для того, чтобы узнать есть ли в водимй строке указанный шаблон (pattern).

Вот список методов поиска:

public boolean lookingAt()
Предпринимает попытку поиска соответствия вводимой последовательности в начале области с шаблоном.

public boolean find()
Предпринимает попытку поиска следующей подпоследовательности в вводимой последовательности, соответствующей шаблону.

public boolean find(int start)
Сброс данного поиска соответствия и попытка поиска новой подпоследовательности в вводимой последовательности, соответствующей шаблону с указанного индекса.

public boolean matches()
Предпринимает попытку поиска совпадений во всей области с шаблоном.

Методы замещения

Для замещения текста в вводной строке в языке Java предусмотрены следующие методы:

public Matcher appendReplacement(StringBuffer sb, String replacement)
Производит нетерминальное присоединение и замену.

public StringBuffer appendTail(StringBuffer sb)
Производит терминальное присоединение и замену.

public String replaceAll(String replacement)
Заменяет каждую подпоследовательность в вводимой последовательности, совпадающей с шаблоном, указанным в замещающей строке.

public String replaceFirst(String replacement)
Замещает первую подпоследовательность в вводимой последовательности, совпадающей с шаблоном, указанным в замещающей строке.

public static String quoteReplacement(String s)
Возвращает литеральную замену Строки для указанной Строки. Данный метод производит сроку, которая будет функционировать в качестве литеральной замены s в методе appendReplacement класса Matcher.

Методы start и end

Далее представлен пример, в котором производится подсчет количества раз, когда в строке ввода встречается слово "кот".

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {

   private static final String REGEX = "\\bкот\\b";
   private static final String INPUT = "кот кот кот котёл кот";

   public static void main( String args[] ) {
      Pattern p = Pattern.compile(REGEX);
      Matcher m = p.matcher(INPUT);   // получение matcher объекта
      int count = 0;

      while(m.find()) {
         count++;
         System.out.println("Номер вхождения: "+count);
         System.out.println("Начальная позиция вхождения: "+m.start());
         System.out.println("Конечная позиция вхождения: "+m.end());
      }
   }
}
```

```
Номер вхождения: 1
Начальная позиция вхождения: 0
Конечная позиция вхождения: 3
Номер вхождения: 2
Начальная позиция вхождения: 4
```

Методы matches и lookingAt

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {

   private static final String REGEX = "Pro";
   private static final String INPUT = "ProgLang";
   private static Pattern pattern;
   private static Matcher matcher;

   public static void main( String args[] ) {
      pattern = Pattern.compile(REGEX);
      matcher = pattern.matcher(INPUT);

      System.out.println("Текущее регулярное выражение: "+ REGEX);
      System.out.println("Текущие входные данные: "+ INPUT);

      System.out.println("lookingAt(): " + matcher.lookingAt());
      System.out.println("matches(): " + matcher.matches());
   }
}
```

```
Текущее регулярное выражение: Pro
Текущие входные данные: ProgLang
lookingAt(): true
matches(): false
```

Методы appendReplacement и appendTail

Класс Matcher также предоставляет методы замены текста appendReplacement и appendTail.

Далее представлен пример, поясняющий их функциональность.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {

   private static String REGEX = "а*д";
   private static String INPUT = "аадProgLangаааадProgLangадProgLangд<<<<<<";
   private static String REPLACE = "-";
   public static void main(String[] args) {

      Pattern p = Pattern.compile(REGEX);

      // получение matcher объекта
      Matcher m = p.matcher(INPUT);
      StringBuffer sb = new StringBuffer();
      while(m.find()) {
         m.appendReplacement(sb, REPLACE);
      }
      m.appendTail(sb);
      System.out.println(sb.toString());
   }
  }
```

```
-ProgLang-ProgLang-ProgLang-<<<<<<
```

Методы класса PatternSyntaxException

PatternSyntaxException представляет непроверяемое исключение, которое отображает синтаксическую ошибку в шаблоне регулярного выражения. Класс PatternSyntaxException представлен следующими методами, которые помогут определить вам ошибку.

№.	Метод и описание
1	**public String getDescription()** Представляет описание ошибки.
2	**public int getIndex()** Представляет индекс ошибки.
3	**public String getPattern()** Представляет шаблон регулярного выражения, содержащего ошибку.
4	**public String getMessage()** Производит возврат многострочной строки, содержащей описание синтаксической ошибки и ее индекс, ошибочный образец регулярного выражения, а также визуальную индикацию индекса ошибки в шаблоне.