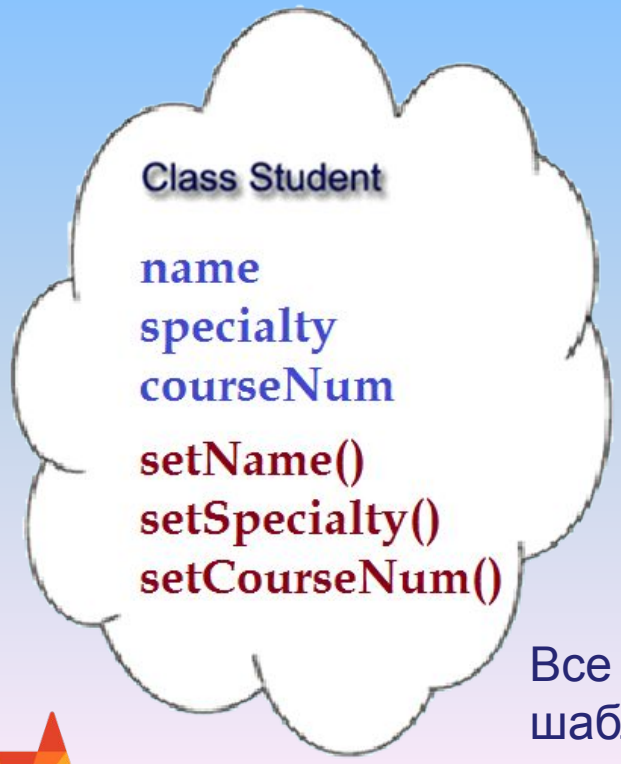
# Тема
# *Строки и регулярные выражения*

## Понятие класса.

**Класс** — в объектно-ориентированном программировании, представляет собой шаблон для создания объектов, обеспечивающий начальные значения состояний: инициализация полей-переменных и реализация поведения функций или методов.




объекты

Все экземпляры одного класса созданы по одному шаблону, поэтому имеют один и тот же набор полей и методов.

## Понятие класса.

Объектно-ориентированное программирование сводится к созданию некоторого количества классов, включая интерфейс и реализацию, и последующему их использованию.

Графическое представление некоторого количества классов и связей между ними называется диаграммой классов.

Объектно-ориентированный подход за время своего развития накопил множество рекомендаций (паттернов) по созданию классов и иерархий классов.

Используемые человеком классификации в зоологии, ботанике, химии, деталях машин, несут в себе основную идею, что любую вещь всегда можно представить частным случаем некоторого более общего понятия. Конкретное яблоко — это в целом некоторое яблоко, вообще яблоко, а любое вообще яблоко — фрукт.

```
class ИмяКласса{

    модификаторы тип переменнаяЭкземпляра1;
    ...
    модификаторы тип переменнаяЭкземпляраN;

    модификаторы ИмяКласса ( список параметров) {
    // тело метода конструктора1
    }
    …
    модификаторы ИмяКласса (список  параметров ) {
    // тело метода конструктораN
    }

    модификаторы тип имяМетода1 ( список параметров) {
    // тело метода1
    }
    …
    модификаторы тип имяМетодаN (список  параметров )   {
    // тело методаN
    }
}
```

Строки — представляют собой последовательность символов. Строки в Java широко используются и являются объектами.

Платформа Java предоставляет класс строк (class String) для создания и работы со строками.

**Module** java.base
**Package** java.lang

## Class String

java.lang.Object
    java.lang.String

**All Implemented Interfaces:**

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

```
class A extends String {
}
```

Класс терминальный (финальный), то есть создать подклассы на его основе невозможно.

Класс String предоставляет ряд методов для манипуляции строками. По факту объект String – ссылка на область в памяти, в которой последовательно размещены символы.

Строка – последовательность символов.

Для создания новой строки мы можем использовать один из 11 конструкторов класса String, либо напрямую присвоить строку в двойных кавычках (строковый литерал):

```
String str1 = "";                   // пустая строка
String str2 = new String();         // пустая строка
String str3 = "Hello World!";
String str4 = new String( original: "Hello World!");
String str5 = new String(new char[] {'H', 'e', 'l', 'l', 'o'});
String str6 = new String(new char[]{'H', 'e', 'l', 'l', 'o', ' ',
        'W', 'o', 'r', 'l', 'd', '!', 'e'}, offset: 6, count: 5);
        //6 -начальный индекс, 5 -кол-во символов
```

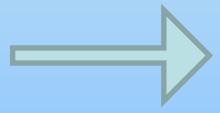
```
String str7 = new String("Hello World!", 6, 5);
```

```
System.out.println("str1 :" + str1);
System.out.println("str2 :" + str2);
System.out.println("str3 :" + str3);
System.out.println("str4 :" + str4);
System.out.println("str5 :" + str5);
System.out.println("str6 :" + str6);
```

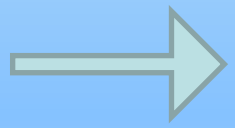
```
str1 :
str2 :
str3 :Hello World!
str4 :Hello World!
str5 :Hello
str6 :World
```

```
String str8 = null;
System.out.println("str8 :" + str8);

String str9;
System.out.println("str9 :" + str9);
```



```
str8 :null
```



Это не массив

При работе со строками важно понимать, что объект String является неизменяемым (**immutable**). То есть при любых операциях над строкой, которые изменяют эту строку, фактически будет создаваться новая строка.

В Java, 2 способа создания строк неявный, с помощью присваивания строкового литерала и явный путем создания объекта, при помощи оператора new

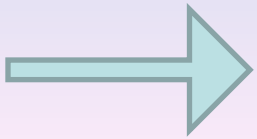
```
String s1 = "Hello";                // String literal
String s2 = new String("Hello");  // String object
```

Java предоставляет специальный механизм для хранения строковых литералов так называемый пул (*string common pool или string literal pool (общее хранилище строк)*.

Если несколько объектов имеют одинаковое содержимое, то они будут использовать одно и тоже значение из пула. Чтобы снизить число строковых объектов, создаваемых виртуальной машиной, каждый раз, при создании нового строкового литерала, JVM сначала проверяет пул если такой строки в пуле нет, то создается новый String объект, который ложится в пул.

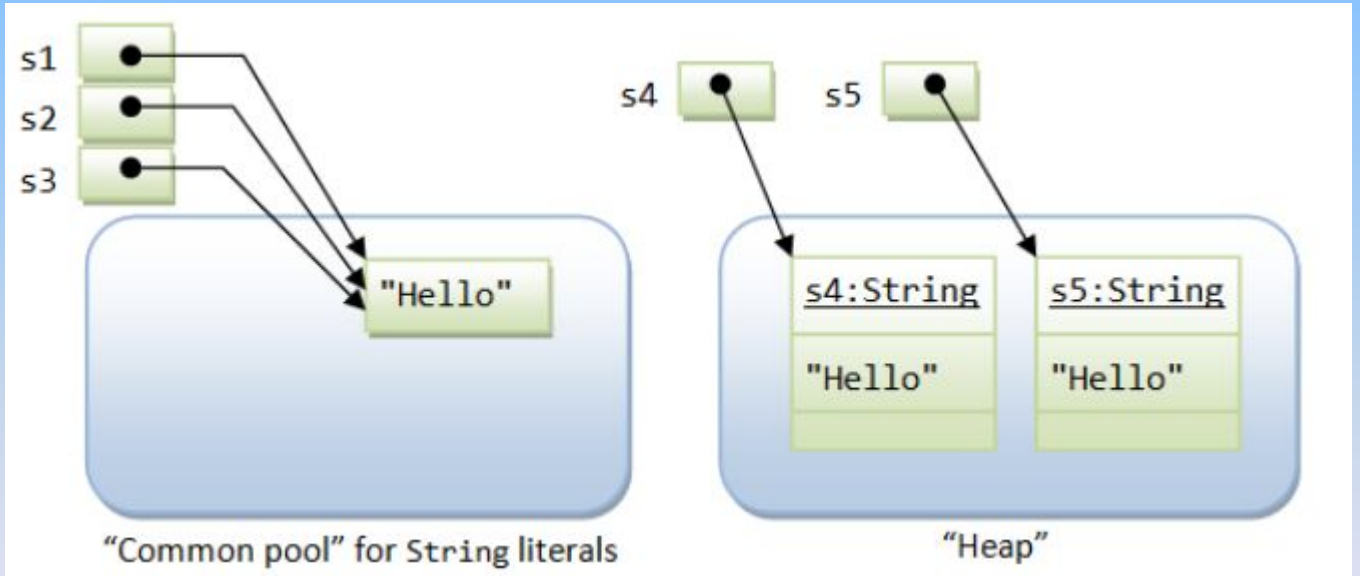
Это возможно так как строки неизменяемые (immutable), то есть могут взаимно использоваться без опасения о том, что данные будут изменены.

```
String str1 = "Hello World!";
String str2 = "Hello World!";
String str3 = new String( original: "Hello World!");
System.out.println(str1 == str2);
System.out.println(str1 == str3);
System.out.println(str1.equals(str2));
System.out.println(str1.equals(str3));
```



```
true
false
true
true
```

С другой стороны объекты созданные с помощью оператора new и конструктора хранятся не в пуле, а в heap - е. Каждый heap объект в хипе имеет свой собственный экземпляр, точно также, как любой другой объект. Нет механизма повторного использования данных в хипе



Такой способ создания объекта не рекомендуется:

**String s  = new String(“Hello”);**

JVM сначала проверит пул если в нем нет литерала “**Hello**”, создаст его. Затем будет создан новый объект, при помощи оператора new и конструктора с входным параметром “**Hello**”. Аргумент конструктора String - " **Hello** " - сам является экземпляром класса String и функционально равнозначен всем объектам, создаваемым конструктором. Если этот оператор попадает в цикл или в часто вызываемый метод, без всякой надобности могут создаваться миллионы экземпляров String.

Исправленная версия выглядит просто:

**String s ="Hello";**

В этом варианте используется единственный экземпляр String вместо создания новых при каждом проходе

Создание строк при помощи оператора **new** может быть полезным, только в очень специфических случаях.

Вот пример плохого использования. При использование java до v 7 update 6 метод substring при том, что возвращал новую строку, но продолжал хранить ссылку на входящее значение.

Сборщик мусора не мог удалить объект. То есть если мы получаем строку длиной 5000 символов и хотим лишь получить её префикс, используя метод substring(), то 5000 символов будут продолжать храниться в памяти.

Для систем, которые получают и обрабатывают множество сообщений, это может быть серьезной проблемой.

Для того, чтобы избежать данную проблему, можно использовать два варианта:

```
String prefix = new String( longString.substring(0,5) );
```

# Статический метод класса String - format() позволяет форматирование:

## %[аргумент_индекс][флаги][ширина][.точность]символ_преобразования

| |
|---|
| *%a* Шестнадцатеричное значение с плавающей точкой |
| *%b* Логическое (булево) значение аргумента |
| *%c* Символьное представление аргумента |
| *%d* Десятичное целое значение аргумента |
| *%h* Хэш-код аргумента |
| *%e* Экспоненциальное представление аргумента |
| *%f* Десятичное значение с плавающей точкой |
| *%g* Выбирает более короткое представление из двух: %e или %f |
| *%o* Восьмеричное целое значение аргумента |
| *%n* Вставка символа новой строки |
| *%s* Строковое представление аргумента |
| *%t* Время и дата |
| *%x* Шестнадцатеричное целое значение аргумента |
| *%%* Вставка знака % |

```
String fs;
String stringVar = "Hello World!";
int intVar = 10;
float floatVar = 10.0f;

fs = String.format("Значение переменной float = " +
        "%f, пока значение integer " +
        "переменная = %d, и string " +
        "= %s", floatVar, intVar, stringVar);
System.out.println(fs);
```

```
Значение переменной float = 10,000000, пока значение integer переменная = 10, и string = Hello World!
```

```
"%,.2f, пока значение integer " +
```

```
Значение переменной float = 10,00, пока значение integer переменная = 10, и string = Hello World!
```

| | |
|---|---|
| - | Выравнивание влево |
| # | Изменяет формат преобразования |
| 0 | Выводит значение, дополненное нулями вместо пробелов. Применим только к числам. |
| Пробел | Положительные числа предваряются пробелом |
| + | Положительные числа предваряются знаком +. Применим только к числам. |
| , | Числовые значения включают разделители групп. Применим только к числам. |
| ( | Отрицательные числовые значения заключаются в скобки. Применим только к числам. |

## Длина строки (length())

```
String str1 = "Hello World!";
System.out.println("str1 :" + str1.length());
```

```
str1 :12
```

## Объединение строк (+ или concat)

```
String str1 = "Hello ";
String str2 = "World!";
str1 = str1.concat(str2);
//str1 = str1 + str2;
System.out.println(str1);
```

```
Hello World!
```

| № | Методы с описанием |
|---|---|
| 1 | char charAt(int index)<br>Возвращает символ по указанному индексу. |
| 2 | int compareTo(Object o)<br>Сравнивает данную строку с другим объектом. |
| 3 | int compareTo(String anotherString)<br>Сравнивает две строки лексически. |
| 4 | int compareToIgnoreCase(String str)<br>Сравнивает две строки лексически, игнорируя регистр букв. |
| 5 | String concat(String str)<br>Объединяет указанную строку с данной строкой, путем добавления ее в конце. |
| 6 | boolean contentEquals(StringBuffer sb)<br>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)<br>Возвращает строку, которая представляет собой последовательность символов, в указанный массив. |
| 8 | static String copyValueOf(char[] data, int offset, int count)<br>Возвращает строку, которая представляет собой последовательность символов, в указанный массив. |
| 9 | boolean endsWith(String suffix)<br>Проверяет заканчивается ли эта строка указанным окончанием. |
| 10 | boolean equals(Object anObject)<br>Сравнивает данную строку с указанным объектом. |
| 11 | boolean equalsIgnoreCase(String anotherString)<br>Сравнивает данную строку с другой строкой, игнорируя регистр букв. |
| 12 | byte getBytes()<br>Кодирует эту строку в последовательность байтов с помощью платформы charset, сохраняя результат в новый массив байтов. |

| | |
|---|---|
| 13 | byte[] getBytes(String charsetName)<br>Кодирует эту строку в последовательность байтов с помощью платформы charset, сохраняя результат в новый массив байтов. |
| 14 | void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)<br>Копирует символы из этой строки в массив символов назначения. |
| 15 | int hashCode()<br>Возвращает хэш-код для этой строки. |
| 16 | int indexOf(int ch)<br>Возвращает индекс первого вхождения указанного символа в данной строке. |
| 17 | int indexOf(int ch, int fromIndex)<br>Возвращает индекс первого вхождения указанного символа в данной строке, начиная поиск с указанного индекса. |
| 18 | int indexOf(String str)<br>Возвращает индекс первого вхождения указанной подстроки в данной строке. |
| 19 | int indexOf(String str, int fromIndex)<br>Возвращает индекс первого вхождения указанной подстроки в данной строке, начиная с указанного индекса. |
| 20 | String intern()<br>Возвращает каноническое представление для строкового объекта. |
| 21 | int lastIndexOf(int ch)<br>Возвращает индекс последнего вхождения указанного символа в этой строке. |
| 22 | int lastIndexOf(int ch, int fromIndex)<br>Возвращает индекс последнего вхождения указанного символа в этой строке, начиная обратный поиск с указанного индекса. |
| 23 | int lastIndexOf(String str)<br>Возвращает индекс последнего вхождения указанной подстроки в данной строке. |
| 24 | int lastIndexOf(String str, int fromIndex)<br>Возвращает индекс последнего вхождения указанной подстроки в этой строке, начиная обратный поиск с указанного индекса. |
| 25 | int length()<br>Возвращает длину строки. |

| | |
|---|---|
| 26 | boolean matches(String regex)<br>Сообщает, соответствует ли или нет эта строка заданному регулярному выражению. |
| 27 | boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)<br>Проверяет равны ли две области строки. |
| 28 | boolean regionMatches(int toffset, String other, int ooffset, int len)<br>Проверяет равны ли две области строки. |
| 29 | String replace(char oldChar, char newChar)<br>Возвращает новую строку, в результате, заменив все вхождения oldChar в этой строке на newChar. |
| 30 | String replaceAll(String regex, String replacement)<br>Заменяет каждую подстроку строки, соответствующей заданному регулярному выражению с данной заменой. |
| 31 | String replaceFirst(String regex, String replacement)<br>Заменяет первые подстроки данной строки, которая соответствует заданному регулярному выражению с данной заменой. |
| 32 | String[] split(String regex)<br>Разделяет эту строку, окружая данным регулярным выражением. |
| 33 | String[] split(String regex, int limit)<br>Разделяет эту строку, окружая данным регулярным выражением. |
| 34 | boolean startsWith(String prefix)<br>Проверяет, начинается ли эта строка с заданного префикса. |
| 35 | boolean startsWith(String prefix, int toffset)<br>Проверяет, начинается ли эта строка с указанного префикса, начиная с указанного индекса. |
| 36 | CharSequence subSequence(int beginIndex, int endIndex)<br>Возвращает новую последовательность символов, которая является подпоследовательностью этой последовательности. |
| 37 | String substring(int beginIndex)<br>Возвращает новую строку, которая является подстрокой данной строки. |
| 38 | String substring(int beginIndex, int endIndex)<br>Возвращает новую строку, которая является подстрокой данной строки. |

| | |
|---|---|
| 39 | char[] toCharArray()<br>Преобразует эту строку в новый массив символов. |
| 40 | String toLowerCase()<br>Преобразует все символы в данной строке в нижний регистр, используя правила данного языкового стандарта. |
| 41 | String toLowerCase(Locale locale)<br>Преобразует все знаки в данной строке в нижний регистр, используя правила данного языкового стандарта. |
| 42 | String toString()<br>Этот объект (который уже является строкой!) возвращает себя. |
| 43 | String toUpperCase()<br>Преобразует все символы в строке в верхний регистр, используя правила данного языкового стандарта. |
| 44 | String toUpperCase(Locale locale)<br>Преобразует все символы в строке в верхний регистр, используя правила данного языкового стандарта. |
| 45 | String trim()<br>Возвращает копию строки с пропущенными начальными и конечными пробелами. |
| 46 | static String valueOf(primitive data type x)<br>Возвращает строковое представление переданного типа данных аргумента. |

**Метод charAt()** — возвращает символ, расположенный по указанному индексу строки (индексы строк начинаются с нуля).

```
String str1 = "Hello World!";
System.out.println("str1 :" + str1.charAt(1));
```

→ `str1 :e`

**Метод split()** — разделяет строку по признаку

```
String str1 = "Hello World!";
for (String retval : str1.split( regex: " ")) {
    System.out.println(retval);
}
```

→

```
Hello
World!
```

**Метод compareTo()** имеет два варианта. Первый: метод сравнивает строку с другим объектом, а второй: метод лексически сравнивает две строки.

Результат compareTo() получает значение 0, если аргумент является строкой лексически равной данной строке; значение меньше 0, если аргумент является строкой лексически большей, чем сравниваемая строка; и значение больше 0, если аргумент является строкой лексически меньшей этой строки.

```
String str1 = "Hello World!";
String str2 = "Hello World";
System.out.println("str1 :" + str1.compareTo(str2));
```

```
str1 :1
```

**Метод hashCode()** — возвращает хэш-код для данной строки. Хэш-код вычисляется для объекта String: **s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1],** где s[i] — i-ый символ строки, n — длина строки, и ^ возведение в степень.

```
String str1 = "Hello World!";
System.out.println("str1 :" + str1.hashCode());
```

```
str1 :-969099747
```

**Метод trim()** — возвращает копию строки с пропущенными начальными и конечными пробелами, другими словами метод позволяет в Java удалить пробелы в начале и конце строки.

**Метод indexOf()** имеет следующие варианты:

public int indexOf(int ch) — возвращает индекс в данной строке первого вхождения указанного символа или -1, если символ не встречается.

public int indexOf(int ch, int fromIndex) — возвращает индекс в данной строке первого вхождения указанного символа, начиная поиск по указанному индексу, или значение -1, если символ не встречается.

int indexOf(String str) — возвращает индекс в данной строке первого вхождения указанной подстроки. Если эта подстрока не встречается, возвращается -1.

int indexOf(String str, int fromIndex) — возвращает индекс в данной строке первого вхождения указанной подстроки, начиная с указанного индекса. Если не встречается, возвращается -1.

```
String str1 = "Hello World!";
System.out.println("str1 :" + str1.indexOf('e'));
```

→ `str1 :1`

**Метод replace()** — возвращает новую строку, заменив все вхождения oldChar, в данной строке, на newChar,

```
String str1 = "Hello World!";
System.out.println("str1 :" + str1.replace( oldChar: 'l', newChar: 'L'));
```

→ `str1 :HeLLo WorLd!`

**Метод toCharArray()** — в Java преобразует данную строку в новый массив символов.

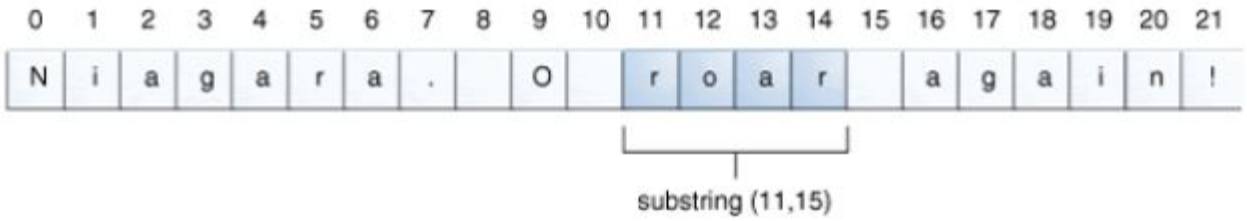
```
for (char c : "Привет МИР!".toCharArray()) {
    print(c);
}
```

**Метод substring()** возвращает новую строку, которая является подстрокой данной строки. Имеет два варианта. Подстрока начинается с символа, заданного индексом, и продолжается до конца данной строки или до endIndex-1, если введен второй аргумент.

```
String str1 = "Hello World!";
System.out.println("str1 :" + str1.substring(6));
```

→ `str1 :World!`

```
String palindrome = "Niagara. O roar again!" ;
String roar = palindrome.substring(11,15);
```



**Метод toLowerCase()** - два варианта.

Первый вариант преобразует все символы в данной строки в нижний регистр, используя правила языкового стандарта. Это эквивалент вызова toLowerCase(Locale.getDefault()).

Второй вариант принимает языковой стандарт в качестве аргумента для использования во время преобразования в нижний регистр.

```
String str1 = new String( original: "Hello World!");

System.out.print("str1: ");
System.out.println(str1.toLowerCase());

System.out.print("str1: ");
System.out.println(str1.toLowerCase(Locale.ENGLISH));
```

```
str1: hello world!
str1: hello world!
```

**Метод toUpperCase()** - два варианта.

Первый вариант преобразует все знаки в данной строке в верхний регистр, используя правила данного языкового стандарта. Это эквивалент вызова toUpperCase(Locale.getDefault()).

Второй вариант принимает языковой стандарт в качестве аргумента для использования во время преобразования в верхний регистр.

**Метод valueOf()** имеет много вариантов, которые зависят от передаваемых параметров. Этот метод возвращает строковое представление переданного аргумента:

- valueOf(boolean b) — возвращает строковое представление логического аргумента.
- valueOf(char c) — возвращает строковое представление char аргумента.
- valueOf(char[ ] data) — возвращает строковое представление массив char аргументов.
- valueOf(char[ ] data, int offset, int count) — возвращает строковое представление определенного подмассива массив char аргументов.
- valueOf(double d) — возвращает строковое представление double аргумента.
- valueOf(float f) — возвращает строковое представление float аргумента.
- valueOf(int i) — возвращает строковое представление int аргумента.
- valueOf(long l) — возвращает строковое представление long аргумента.
- valueOf(Object obj) — возвращает строковое представление объекта аргумента.

```
System.out.println("str1 :" + String.valueOf(2.0f));
```

```
str1 :2.0
```

## Классы StringBuilder и StringBuffer

Объекты String являются неизменяемыми, поэтому все операции, которые изменяют строки, фактически приводят к созданию новой строки, что ухудшает производительность приложения.

**Классы StringBuffer** и **StringBuilder** по сути напоминает расширяемую строку, которую можно изменять, а не пересоздавать.

**Классы StringBuffer и StringBuilder** используются, при необходимости сделать много изменений в строке символов.

**Различие.** Класс StringBuffer синхронизированный и потокобезопасный (принято использовать в многопоточных приложениях, где объект данного класса может меняться в различных потоках). Класс StringBuilder, работает быстрее, чем StringBuffer в однопоточных приложениях (потокоНЕбезопасный).

В целом оба класса имеют одинаковый набор конструкторов и методов

**Классы StringBuilder и StringBuffer имеют по** четыре конструктора:

- StringBuffer() - конструктор без параметров резервирует в памяти место для 16 символов
- StringBuffer(int capacity) - в качестве параметра принимает количество символов
- StringBuffer(String str) - принимают строку плюс при этом резервируя память для дополнительных 16 символов
- StringBuffer(CharSequence chars) - принимают набор символов, при этом резервируя память для дополнительных 16 символов

```
String str = "Java";
StringBuilder strBuilder = new StringBuilder(str);
System.out.println( "strBuilder = " + strBuilder );
System.out.println( "strBuilder.length() = " + strBuilder.length() );
System.out.println( "strBuilder.capacity() = " + strBuilder.capacity() );
```

```
strBuilder = Java
strBuilder.length() = 4
strBuilder.capacity() = 20
```

Метод length() возвращает реальную.
Метод capacity() возвращает строку плюс резерв

Получить реальную строку, которая хранится в **StringBuilder**, используют стандартный **метод toString():**

```
System.out.println( strBuilder.toString() );
```

Метод **ensureCapacity()** позволяет изменить минимальную емкость буфера символов:

```
String str = "Java";
StringBuilder strBuilder = new StringBuilder(str);
strBuilder.ensureCapacity( minimumCapacity: 32);
System.out.println( "strBuilder.length() = " + strBuilder.length() );
System.out.println( "strBuilder.capacity() = " + strBuilder.capacity() );
```

```
strBuilder.length() = 4
strBuilder.capacity() = 42
```

| № | Описание |
|---|---|
| 1 | int capacity()<br>Возвращает текущую вместимость буфера String. |
| 2 | char charAt(int index)<br>Возвращается указанный символ последовательности, в настоящее время представленный буфером строки, указанный индексом аргумент. |
| 3 | void ensureCapacity(int minimumCapacity)<br>Гарантирует вместимость буфера, по крайней мере равным указанному минимуму. |
| 4 | void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)<br>Символы копируются из этого буфера строки в символьный массив назначения dst. |
| 5 | int indexOf(String str)<br>Возвращает индекс в данной строке первого вхождения указанной подстроки. |
| 6 | int indexOf(String str, int fromIndex)<br>Возвращает индекс в данной строке первого вхождения указанной подстроки, начиная с указанного индекса. |
| 7 | int lastIndexOf(String str)<br>Возвращает индекс в данной строке последнего вхождения указанной подстроки. |
| 8 | int lastIndexOf(String str, int fromIndex)<br>Возвращает индекс в данной строке последнего вхождения указанной подстроки, начиная с указанного индекса. |
| 9 | int length()<br>Возвращает длину строкового буфера (количество символов). |
| 10 | void setCharAt(int index, char ch)<br>Символ с указанным индексом этого буфера строки имеет значение ch. |

| | |
|---|---|
| 11 | void setLength(int newLength)<br>Устанавливает длину буфера строки (Stringbuffer). |
| 12 | CharSequence subSequence(int start, int end)<br>Возвращает новую последовательность символов, которая является подпоследовательностью этой последовательности. |
| 13 | String substring(int start)<br>Возвращает новую строку, которая содержит подпоследовательность символов в данный момент содержащихся в StringBuffer. Подстрока начинается с указанного индекса и продолжается до конца StringBuffer. |
| 14 | String substring(int start, int end)<br>Возвращает новую строку, которая содержит подпоследовательность символов в данный момент содержащихся в этом StringBuffer. |
| 15 | String toString()<br>Преобразование в строку, представляющую данные в этой строке буфера. |

## Получение и установка символов

Метод **charAt() получает, а метод setCharAt() устанавливает символ по определенному индексу:**

```
1  StringBuffer strBuffer = new StringBuffer("Java");
2  char c = strBuffer.charAt(0); // J
3  System.out.println(c);
4  strBuffer.setCharAt(0, 'c');
5  System.out.println(strBuffer.toString()); // cava
```

Метод **getChars()** получает набор символов между определенными индексами:

```
1  StringBuffer strBuffer = new StringBuffer("world");
2  int startIndex = 1;
3  int endIndex = 4;
4  char[] buffer = new char[endIndex-startIndex];
5  strBuffer.getChars(startIndex, endIndex, buffer, 0);
6  System.out.println(buffer); // orl
```

## Добавление в строку

Метод **append()** добавляет подстроку в конец StringBuffer:

```
1  StringBuffer strBuffer = new StringBuffer("hello");
2  strBuffer.append(" world");
3  System.out.println(strBuffer.toString()); // hello world
```

Метод **insert()** добавляет строку или символ по определенному индексу в StringBuffer:

```
1  StringBuffer strBuffer = new StringBuffer("word");
2
3  strBuffer.insert(3, 'l');
4  System.out.println(strBuffer.toString()); //world
5
6  strBuffer.insert(0, "s");
7  System.out.println(strBuffer.toString()); //sworld
```

## Удаление символов

Метод **delete()** удаляет все символы с определенного индекса о определенной позиции, а метод **deleteCharAt()** удаляет один символ по определенному индексу:

```
StringBuffer strBuffer = new StringBuffer("assembler");
strBuffer.delete(0,2);
System.out.println(strBuffer.toString()); //sembler

strBuffer.deleteCharAt(6);
System.out.println(strBuffer.toString()); //semble
```

## Обрезка строки

Метод **substring()** обрезает строку с определенного индекса до конца, либо до определенного индекса:

```
StringBuffer strBuffer = new StringBuffer("hello java!");
String str1 = strBuffer.substring(6); // обрезка строки с 6 символа до конца
System.out.println(str1); //java!

String str2 = strBuffer.substring(3, 9); // обрезка строки с 3 по 9 символ
System.out.println(str2); //lo jav
```

## Изменение длины

Для изменения длины StringBuffer (не емкости буфера символов) применяется метод **setLength()**. Если StringBuffer увеличивается, то его строка просто дополняется в конце пустыми символами, если уменьшается - то строка по сути обрезается:

```
1  StringBuffer strBuffer = new StringBuffer("hello");
2  strBuffer.setLength(10);
3  System.out.println(strBuffer.toString()); //"hello     "
4
5  strBuffer.setLength(4);
6  System.out.println(strBuffer.toString()); //"hell"
```

## Замена в строке

Для замены подстроки между определенными позициями в StringBuffer на другую подстроку применяется метод **replace()**:

```
1  StringBuffer strBuffer = new StringBuffer("hello world!");
2  strBuffer.replace(6,11,"java");
3  System.out.println(strBuffer.toString()); //hello java!
```

Первый параметр метода replace указывает, с какой позиции надо начать замену, второй параметр - до какой позиции, а третий параметр указывает на подстроку замены.

## Обратный порядок в строке

Метод **reverse()** меняет порядок в StringBuffer на обратный:

```
1  StringBuffer strBuffer = new StringBuffer("assembler");
2  strBuffer.reverse();
3  System.out.println(strBuffer.toString()); //relbmessa
```

1. Дана строка символов. Показать номера символов, совпадающих с последним символом строки.

2.1 Дана строка символов. В данной строке найти количество цифр.

2.2 Дана строка символов. В данной строке найти количество цифр от 1 до 5.

3. Удалите в строке все 'abc', за которыми следует цифра.

4. Дан текст. Найти сумму имеющихся в нем цифр.

Регулярные выражения - инструмент для обработки строк.

Регулярные выражения позволяют задать шаблон, которому должна соответствовать строка или подстрока.

Есть методы класса String, которые принимают регулярные выражения и используют их для выполнения операций над строками.

Пакет java.util.regex поддерживает обработку регулярных выражений (regular expression).

Пакет содержит два класса - **Pattern** и **Matcher**. Класс **Patern** применяется для задания регулярного выражения. Класс **Matcher** сопоставляет шаблон с последовательностью символов.

Регулярное выражение состоит из обычных символов, наборов символов и групповых символов.

Обычные символы используются как есть. Если в шаблоне указать символы “abc”, то эти символы и будут искаться в строке.

Символы новой строки, табуляции и др. определяются при помощи стандартных управляющих последовательностей, которые начинаются с обратного слеша (\). Например, символ новой строки можно задать как **\n**.

Наборы символов заключаются в квадратные скобки. Например, **[abc]** совпадает с символами **a, b, c**. Если поставить символ **^** перед набором символов - **[^abc]**, то ищутся совпадения всех символов, кроме **a, b, c**.

Чтобы задать диапазон символов, используется дефис. Например, диапазон от 1 до 9 можно задать как **[1-9]**.

Символ точки является групповым символом, который совпадает с любым символом вообще.

Также можно задать, сколько раз совпадает выражение:
+ - совпадает один или более раз
* - совпадает ноль или более раз
? - совпадает ноль или один раз

```
System.out.println("-123".matches( regex: "-?\\d+"));
System.out.println("123".matches( regex: "-?\\d+"));
System.out.println("+123".matches( regex: "-?\\d+"));
System.out.println("+123".matches( regex: "(-|\\+)?\\d+"));
```

```
true
true
false
true
```

| Конструкция Regex | Что считается совпадением |
|---|---|
| `.` | Любой символ |
| `?` | Ноль (0) или одно (1)повторение предшествующего |
| `*` | Ноль (0) или более повторений предшествующего |
| `+` | Одно (1) или более повторений предшествующего |
| `[]` | Диапазон символов или цифр |
| `^` | Отрицание последующего (то есть, "не *что-то*") |
| `\d` | Любая цифра (иначе, `[0-9]`) |
| `\D` | Любой нецифровой символ (иначе, `[^0-9]`) |
| `\s` | Любой символ-разделитель (иначе, `[\n\t\f\r]`) |
| `\S` | Любой символ, отличный от разделителей |
| `\w` | Любая буква или цифра (иначе, `[A-Za-Z_0-9]`) |
| `\W` | Любой знак, отличный от буквы или цифры |

Примеры,
Выражение **-?\\d+** будет искать число, у которого может быть минус (а может и нет).

Выражение **a.c** позволит найти слова **abc**, **acc**, но не **abbc**.

```
String stringPattern = "[a-z]+";
String text = "code 2 learn java tutorial";
Pattern pattern = Pattern.compile(stringPattern);
Matcher matcher = pattern.matcher(text);
while (matcher.find()) {
    System.out.println(text.substring(matcher.start(), matcher.end()));
}
```

```
code
learn
java
tutorial
```

## Примеры регулярных выражений:

- a? - a один раз или ни разу
- a* - a ноль или более раз
- a+ - a один или более раз
- a{n} - a n раз
- a{n,}- a n или более раз
- a{n,m}- a от n до m

Регулярное выражение или соответствует тексту (его части) или нет. Если регулярное выражение совпадает с частью текста, то мы можем найти его. Если регулярное выражение составное, то мы можем легко выяснить, какая часть регулярного выражения совпадает с какой частью текста.

Например, строка:

Используются файлы file1.doc, file2.txt.
А еще было бы неплохо обратить внимание на файл file3.img.
Также просмотрите содержимое file4.doc.

Из строки нужно вырезать все имена файлов: file1.doc, file2.txt, file3.img, file4.doc.

Для нахождения используется регулярное выражение:

```
[a-zA-Z0-9]+\\.[a-z]{3}
```

Регулярное выражение [a-z]+ соответствует всем строчным буквам в тексте. [a-z] означает любой символ от a до z включительно, и + означает «один или более» символов.

## Pattern и Matcher классы

Pattern класс - объект класса составляет представление регулярного выражения. Класс Pattern не предусматривает никаких публичных конструкторов. Чтобы создать шаблон, необходимо сначала вызвать один из публичных статических методов, которые затем возвращают объект класса Pattern. Эти методы принимают регулярное выражение в качестве аргумента.

Matcher класс - объект «Искатель» является двигателем, который интерпретирует шаблон и выполняет операции сопоставления с входной строкой. Как и Pattern класс, Matcher не имеет публичных конструкторов. Вы получаете объект Matcher вызовом метода matcher(), на объекте класса Pattern.

Методы класса Matcher:

- matches() возвращает true, если шаблон соответствует всей строке, иначе false.
- lookingAt() возвращает true, если шаблон соответствует началу строки, и false в противном случае.
- find() возвращает true, если шаблон совпадает с любой частью текста.

## Сравнение регулярного выражения с текстом

```
Pattern pattern = Pattern.compile("a*b");
Matcher matcher = pattern.matcher( input: "aaab");
boolean b = matcher.matches();
System.out.println(b);
```

```
true
```

## Простой валидатор ссылки

```
    System.out.println(test( testString: "google.com"));
    System.out.println(test( testString: "reference1.ua"));
    System.out.println(test( testString: "reference1.org"));
}
public static boolean test(String testString) {
    Pattern pattern = Pattern.compile(".+\\.(com|ua|ru)");
    Matcher matcher = pattern.matcher(testString);
    return matcher.matches();
}
```

```
true
true
false
```

## Регулярное выражение для проверки email

```
String regex = "(\\w{6,})@(\\w+\\.)([a-z]{2,4})";
String s = "адреса эл.почты:ivan@gmail.com, ivanov@gmail.com, "
        + "sidorov@bsu.by!";
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(s);
while (matcher.find()) {
    System.out.println("e-mail: " + matcher.group());
}
```

```
e-mail: ivanov@gmail.com
e-mail: sidorov@bsu.by
```

## Методы Pattern.split(), String.split()

```
Pattern pattern = Pattern.compile("\\d+\\s?");
String[] words = pattern.split( input: "java5tiger 77 java6mustang");
System.out.print(Arrays.toString(words));

String str = "java5tiger 77 java6mustang";
String[] wordstr = str.split( regex: "\\d+\\s?");
System.out.print(Arrays.toString(wordstr));
```

```
[java, tiger , java, mustang]
[java, tiger , java, mustang]
```

***Спасибо за внимание!***