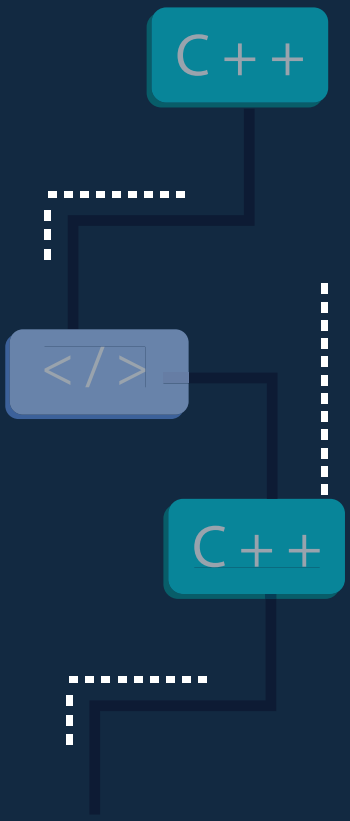


top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

Язык программирования JAVA

Урок № 5

Массивы и строки

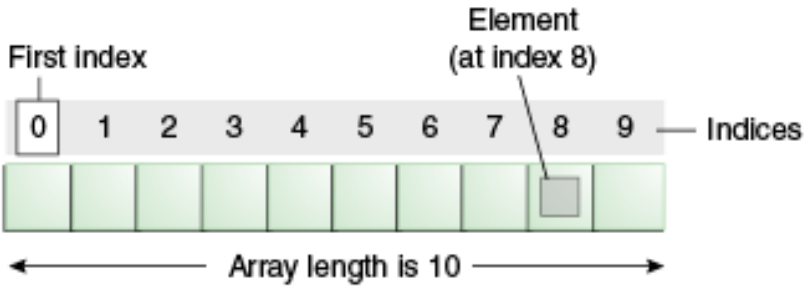
Содержание

1. Массивы 3
Одномерные массивы 3
Многомерные массивы 5
Метод Java.lang.System.arraycopy() 8
Класс Java.util.Arrays 9
2. Строки 12
Статические методы класса String 13
Escape-последовательности 15
Методы экземпляра 15
Класс StringBuffer 20
Методы класса StringBuffer 21
StringBuilder, его отличия от класса StringBuffer 22
Метод toString() 22
Класс StringTokinazer 23
Особенности конкатенации строк с другими типами 24
Сравнение строк 25
Регулярные выражения 27

1. Массивы

Массив в Java – это конечный набор элементов одного типа. Массив в Java это объект, поэтому переменная, указывающая на массив, является переменной ссылочного типа.

Одномерные массивы



Синтаксис объявления ссылки массива.

```
тип[] идентификатор;
или
тип идентификатор[].
```

Пример объявления ссылки на массив:

```
int[] mas; // объявляется переменная типа массив
   // с именем mas
или
int mas[];
```

Объект массива создается с помощью оператора new. Синтаксис создания массива элементов:

```
new int[<количество элементов>];
```

Пример:

```
int[] mas; // объявление переменной массива
mas = new int[3]; // создание массива
```

или

```
int[] mas = new int[3]; // создан массив на три
// элемента и ссылка на объект записана в
// переменную с именем mas.
```

***ВАЖНО!!!** Сразу после создания массива его элементы инициализируются значениями по умолчанию, для всех числовых типов это 0: для char – символ с юникодом 0, для boolean – false, для ссылочных типов – null.*

Обращение к элементам массива происходит по индексу. Индексация элементов массива начинается с 0.

Пример:

```
int[] mas = new int[3];
System.out.println(mas[0]);
```

Результат: 0.

Значение элементов массива можно установить в момент создания массива. Этот процесс называется инициализация массива.

Пример:

```
int[] mas = new int[]{3, 2, 1}; // будет создан массив
// на три элемента, его первый элемент
// будет иметь значение 3,
// второй 2, третий 1.
```

Внимание!!! *Нельзя одновременно инициализировать массив и указывать его размер при создании, так как его размер задается количеством элементов в блоке инициализации.*

Пример:

```
int[] mas = new int[5]{3, 2, 1}; // ошибка компиляции!!!
```

Для массивов существует упрощенная запись инициализации массива.

Пример:

```
int[] mas = {3, 2, 1};
```

Массив имеет свойство **length,** в котором хранится длина массива (количество элементов заданных при создании массива), изменить это свойство нельзя.

Пример:

```
int[] mas = {3, 2, 1};
mas.length = 5; // ошибка компиляции
System.out.println(mas.length);
mas = {1, 2, 3, 4, 5}; // ошибка компиляции, можно
// использовать только при объявлении.
```

Многомерные массивы

Многомерных массивов в языке Java нет. Для имитации работы с данными в n-мерном пространстве в Java используется вложение массивов.

Для создания многомерных массивов при объявлении добавляются дополнительные квадратные скобки.

Пример создания двухмерного массива:

```
int[][] mas = new int[5][5]; // создается массив на 5
// элементов, где элементами являются другие массивы
// на 5 элементов.
```

Пример создания трехмерного массива:

```
int[][][] mas = new int[5][3][3];
```

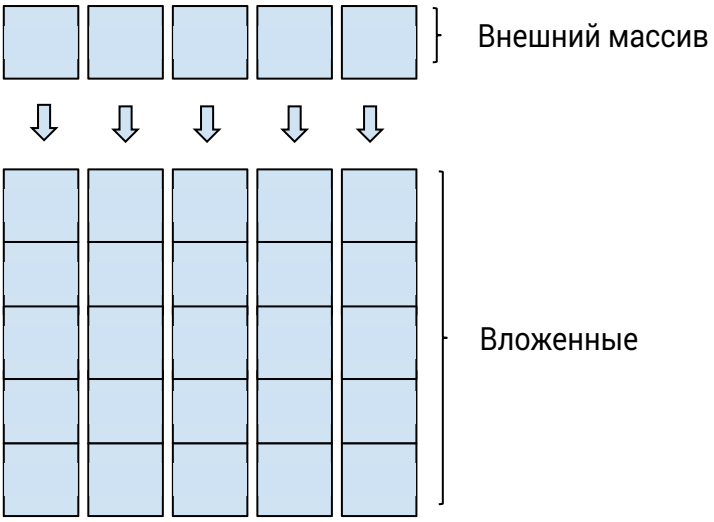


Рис. Структура двухмерного массива в Java

Обратиться к элементу многомерного массива можно, используя несколько индексов.

Пример записи значения в квадратный массив:*

```
int[][] mas = new int[3][3]; // создается массив на 5
mas[2][1] = 3; // запись значения в массив
```

* Квадратным называют двухмерный массив, у которого количество элементов во вложенных массивах равно количеству элементов в исходном масссиве.

Синтаксис многомерных массивов предусматривает отложенную инициализацию.

Пример:

```
int[][] mas = new int[3][]; // создается одномерный
// массив из трех элементов, значения
// которых null
mas[0] = new int[4];
mas[1] = new int[]{3, 3};
mas[2][2] = 7; // ошибка во время выполнения
```

Использование **цикла for-each** для работы с массивами.

Пример вывода значений массива:

```
int [] numbers = new int[] { 3, 2, 1 };
for (int number : numbers)
{
 System.out.print(number);
}
```

Пример:

```
int [] numbers = new int[] { 3, 2, 1 };
for (int number : numbers)
{
 number = 3; // переменная number может быть
     // использована только для чтения
    // значений элементов массива
}
System.out.print(Arrays.toString(numbers));
```

Результат: [3, 2, 1]

Метод Java.lang.System.arraycopy()

Метод **arraycopy** копирует значения из одного массива в другой. Метод нативный* (написан на языке C++), и благодаря этому копирует значения из одного массива в другой быстрее любого кода написанного с использованием языка Java. Используется в коллекциях, например в Java.util.ArrayList.

Синтаксис:

```
System.arraycopy(src, srcPos,
    dest, destPos, length)
```

где src – массив, откуда копируются значения, dest – массив, куда копируются значения, length – количество копируемых элементов из массива src, srcPos – индекс в массиве src, с которого начинается копирование, destPos – индекс в массиве dest, в который начнут копироваться элементы.

Пример:

```
int[] src = new int[]{1, 2, 3, 4, 5};
int[] dest = new int[]{5, 4, 3, 2, 1};
int length = 3;
int srcIndex = 1;
int destIndex= 2;
System.arraycopy(src, srcIndex, dest, destIndex, lenght);
System.out.println(Arrays.toString(dest));
```

Результат: [5, 4, 2, 3, 4]

Примечание. *Если указанное количество элементов отсутствует в исходном массиве или в массиве – приемнике, случится исключительная ситуация.*

* Нативный метод – имеет реализацию на другом языке, не на Java.

Класс Java.util.Arrays

Класс Arrays содержит статические методы для работы с массивами.

- toString – метод преобразует содержание массива в строку. Если в качестве элементов массив содержит объекты, то для каждого элемента будет вызван метод toString() и полученные строки будут конкатенированы*.

Пример вывода содержимого массива в консоль:

```
int [] mas = new int[]{1,2,3};
String content = Arrays.toString(mas);
System.out.println(content );
```

Массив с точки зрения Java является объектом специального класса. Как и все классы в Java, он наследуются от класса Object, но метод toString() в нем не переопределен, поэтому при передачи его в метод System.out.println, мы видим стандартную реализацию метода toString(). Поэтому для вывода на консоль многомерных массивов необходимо использовать метод Arrays.deepToString().

Пример вывода на консоль содержимого многомерного массива:

```
String [][] objects = new String[3][3];
String content = Arrays.deepToString(objects);
System.out.println(content);
```

* Конкатенация – объединение нескольких строк в одну строку.

- fill – метод заполняет массив одинаковыми значениями.

Пример:

```
boolean[] test1 = new boolean[3];
Arrays.fill(test1, true);
```

Пример заполнение элементов массива в диапазоне с 2 по 5 значением 2:

```
int[] mas = new int[10];
int startIndex = 1;
int endIndex = 4;
Arrays.fill(mas, startIndex, endIndex, 2);
```

- sort – сортирует значения массива в естественном порядке (от меньшего к большему).

Пример:

```
int [] mas = new int[] {3, 1, 4, 6, 2};
Arrays.sort(mas);
System.out.println(Arrays.toString(mas));
```

Результат: [1, 2, 3, 4, 6]

Если в массиве хранятся элементы ссылочных типов, в методе sort есть возможность задать правила сортировки с использованием объекта класса реализующего интерфейс Comporator.

Пример сортировки по убыванию:

```
Integer [] mas = new Integer[] {3, 1, 4, 6, 2};
Arrays.sort(mas, Collections.reverseOrder());
System.out.println(Arrays.toString(mas));
```

Результат: [6, 4, 3, 2, 1]

- equals – сравнивает содержимое двух массивов (по-элементно) и если количество элементов равно и все элементы эквиваленты, возвращает истину; иначе ложь.

Пример:

```
int [] mas1 = {1,2,3};
int [] mas2 = {1,2,3};
boolean isEqual = Arrays.equals(mas1, mas2);
System.out.println(isEqual);
```

- binarySearch – возвращает индекс, под которым был найден элемент, используя бинарный поиск.

Пример:

```
int index =Arrays.binarySearch(mas, 3);
```

- deepHashCode – возвращает хэш-код, вычисленный на основе глубокого анализа элементов массива. Можно использовать для массивов, хранящих ссылочные типы.
- asList – возвращает неизменяемый список (объект – адаптер), преобразуя элементы массива в элементы списка.

2. Строки

Класс String – это самый частоиспользуемый класс в Java, он предназначен для хранения набора (массива) символов. Состояние объектов класса String невозможно изменить после создания объекта (объекты класса являются неизменяемыми). Следуя из объявления класса – public final class String, запрещено наследовать подклассы от класса String.

Класс String реализует три интерфейса: Serializable (сериализация), Comparable<String> (сравнение), CharSequence (последовательность символов). Класс String в сущности является оберткой над массивом символов (char [] value).

Внутреннее устройство класса String:

```
public final class String{
private final char value[]; // массив символов
private final int offset; // смещение от начала массива
private final int count; // количество символов в строке
private int hash; // хэш код строки
}
```

Создать объект строка в Java можно с помощью оператора **new** или строкового литерала (символы, ограниченные двойными кавычками).

Пример со строковым литералом:

```
String name = "Вася";
name = ""; // пустая строка
```

Пример с оператором new :

```
String name = new String("Вася");
name = new String(); // пустая строка
```

При первом использовании строкового литерала в программе, в памяти создается новый объект класса String и ссылка на него присваивается в переменную. Объект строки создается в специально отведенном месте памяти, называемом пул строк*. При повторном использовании литерала, новый объект не создается, а в переменную присваивается ссылка на ранее созданный объект.

Пример:

```
String name1 = "Вася"; // создается объект новой строки и
// помещается в пул строк.
String name2 = "Вася"; // в переменную присваивается
// ссылка на раннее созданный объект.
```

При создании объекта строки оператором **new** всегда создается новый объект.

Статические методы класса String

- valueOf – возвращает строковое представление значений примитивных типов.

Пример:

```
String value = String.valueOf(3);
value = String.valueOf(0.5);
value = String.valueOf(true);
```

* Пул строк (String pool) – находится в разделе памяти называемой Permanent Generation. Необходим для уменьшения количества идентичных объектов.

- format – метод возвращает строку, в которой спецификаторы формата* заменены на значения параметров в методе.

  Синтаксис:

```
String.valueOf(<строка шаблона>,
    <подставляемые значения>);
```

Пример:

```
String result = String.valueOf("Жили у бабуси %d
        веселых гуся", 2);
```

При форматировании используются спецификаторы формата:

Спецификатор формата	Выполняемое форматирование
%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 %

* Последовательность символов, начинающаяся со знака %.

Количество спецификаторов формата должно соответствовать количеству параметров в методе. Тип значений должен соответствовать спецификатору формата.

Escape-последовательности

Некоторые символы нельзя вставлять в строку (например, двойные кавычки), для таких символов применяется экранирование с помощью косой черты (прямой слэш).

Пример:

```
String text = "\"Символ, которому предшествует
    наклонная черта влево (\\), является
    escape-последовательностью.\"";
```

Escape-последовательность	Описание
\t	Символ табуляции
\b	Символ Backspace
\n	Новая строка
\r	Возврат каретки
\f	Перевод формата
\'	Символ одинарной кавычки
\"	Символ двойной кавычки
\\	Символ наклонной черты влево

Методы экземпляра

Все методы класса String не изменяют строку, у которой они вызываются, а возвращают ссылку на новый объект класса String.

Неизменяемость (*immutable*) объекта строки обусловлена следующим:

- ***конструкцией языка*** (*Design*). Строки создаются в специальной области памяти в Java куче, известной как "String Intern pool". Когда Вы создаете новую строку с использованием строкового литерала (не с помощью конструктора или каких либо библиотечных функций), проверяется, существует ли уже объект такой строки в строковом пуле. Если строка существуют в пуле, то возвращается ссылка на уже существующий объект, иначе создается новый объект строки в пуле и возвращается ссылка на него. Если строку можно было бы изменить, работа строкового пула была бы не возможна;
- ***безопасность*** (*Security*). Строки широко используются в качестве параметров для открытия файлов и соединений к базам данных, подключения к сетевым ресурсам, в качестве паролей и т.д. Если бы строку можно было бы изменить во всех этих случаях – это бы привело к серьезным угрозам безопасности программы и данных.
- ***эффективность*** (*Efficiency*). Хэш-код строки часто используется в Java в качестве ключа, например в реализации ассоциативного массива Java.util.HashMap. Неизменяемость строки гарантирует нам, что ее хэш-код всегда будет одинаковым, что позволяет рассчитывать хэш-код один раз, во время создания строки, что при частом использовании существенно ускоряет работу с строками.

- charAt – возвращает символ из строки, находящийся по индексу.

- concat – возвращает конкатенацию (объединение двух строк в одну).

Пример:

```
String firstName= "Иван";
String lastName= "Иванов";
String fullname = fullname = cat.concat(name);
// аналогично
// firstName+ " " + lastName;
System.out.println(fullname);
```

- length – возвращает количество символов в строке.
- isEmpty – возвращает истину, если строка не содержит символов, иначе ложь. Работает быстрее, чем length.
- charAt – возвращает символ из строки по индексу.

Пример:

```
String testString = "test";
char myChar = testString.charAt(3);
System.out.println(myChar);
```

Результат: t

- contains – возвращает истину, если строка содержит хотя бы одно совпадение со сравниваемой строкой.

Пример:

```
String testString = "testing";
boolean test = testString.contains("st"));
System.out.println(test);
```

Результат: true

- startsWith – возвращает истину, если строка начинается с искомого символа или строки.

Пример:

```
String str1 = "Star Wars";
boolean test = str1.startsWith("Star");
System.out.println(test);
```

Результат: true

- endsWith – возвращает истину, если строка заканчивается на искомый символ или строку.

Пример:

```
String str1 = "Java.exe";
boolean test = str1.endsWith(".exe");
System.out.println(test);
```

Результат: true

- trim – возвращает строку с удаленными начальными и конечными пробелами.
- toLowerCase – возвращает строку, в которой все заглавные символы исходной строки заменены на строчные.
- toUpperCase – возвращает строку, в которой все строчные символы исходной строки заменены на заглавные.
- indexOf – возвращает индекс символа, с которого найдено первое совпадение с искомой строкой или символом. Поиск начинается с начала сроки. Если совпадение не найдено, возвращает –1.

Пример:

```
String text = "обороноспособность";
int index = text.indexOf('б');
System.out.println(index);
```

Результат: 1

Метод indexOf позволяет начать поиск не с начала, а с определенного символа.

Пример:

```
String text = "обороноспособность";
int index = text.indexOf('б');
index = text.indexOf('б', index + 1);
System.out.println(index);
```

Результат: 1

- lastIndexOf – возвращает индекс символа, с которого найдено первое совпадение с искомой строкой или символом. Поиск начинается с конца сроки. Если совпадение не найдено, возвращает –1.

Пример:

```
int index = "readme.txt".lastIndexOf(".");
System.out.println(index);
```

Результат: 6

- substring – возвращает часть строки из исходной.

Пример:

```
// вырезает строку с 6 символа и до конца строки.
String world = "Hello World".substring(6);
System.out.println(world);
```

Результат: World

Пример:

```
String world = "Hello World".substring(1, 4);
System.out.println(world);
```

Результат: ell

Примечание. *Метод substring возвращает новую строку, но массив с символами берется из исходной строки, изменяется только индекс смещения с начала массива (offset) и количество элементов от индекса смещения (count), в связи с этим сборщик мусора не сможет очистить память от неиспользуемой части строки.*

- replace – возвращает строку, заменяя в исходной строке символ или набор символов на другой символ или набор символов.
- getBytes – возвращает строку в виде массива байт.

Класс StringBuffer

Как уже упоминалось ранее, внутренние данные объекта класса **String** нельзя изменить после его создания, поэтому для работы со строками как с изменяемой структурой, применяется класс **StringBuffer**. В отличие от строк, объекты класса **StringBuffer** нельзя создавать с помощью строковых литералов, только с помощью оператора **new**.

Пример:

```
StringBuffer sb = "test"; // ошибка компиляции
StringBuffer sb = new StringBuffer("test");
```

Методы класса StringBuffer

- append – добавляет новую строку в объект класса StringBuffer и возвращает ссылку на этот же объект.

Пример:

```
StringBuffer sb = new StringBuffer("test");
// создание объекта
sb.append('-').append("test");
// добавление значений цепочкой
sb.append(true); // единичное добавление
sb.append(1);
System.out.println(sb);
```

Результат: test-testtrue1

Примечание. *Конкатенация строк в цикле объектов класса String с использованием оператора + может привести к проблемам с производительностью в связи с тем, что постоянно создаются новые объекты для строк. В такой ситуации предпочтительнее использовать объект класса StringBuffer или StringBuilder.*

- insert – вставляет строку или символ в объект класса StringBuffer.

Пример:

```
StringBuffer sb = new StringBuffer("I Java!");
sb.insert(2, "love"); // 2 - индекс символа, после
      // которого будет вставлена
      // строка
System.out.println(sb.toString());
```

Результат: I love Java!

- reverse – меняет порядок символов на обратный.

Пример:

```
StringBuffer sb = new StringBuffer("palindrome");
sb.reverse();
System.out.println(sb);
```

Результат: emordnilap.

- delete – удаляет часть строки начиная с указанного индекса.

Пример:

```
StringBuffer phrases = new StringBuffer("I do not
      like Java!");
phrases.delete(2, 7);
System.out.println(phrases);
```

- deleteCharAt – удаляет символ из строки, по указанному индексу.
- replace – заменяет подстроку в строке.

StringBuilder, его отличия от класса StringBuffer

Класс StringBuilder имеет точно такое же предназначение и такие же методы, как и StringBuffer. Класс StringBuilder отличается от StringBuffer отсутствием синхронизации данных для многопоточного программирования, поэтому его использование в однопоточной программе предпочтительнее.

Метод toString()

Все классы в Java неявно наследуются от класса Object. В классе Object имеется метод toString(), который воз-

вращает строковое представление состояния объекта. В классах StringBuilder и StringBuffer этот метод возвращает объект класса String, который содержит строку, хранящуюся в объекте классов StringBuilder или StringBuffer на момент вызова метода.

Класс StringTokinazer

Класс StringTokinazer предназначен для разбиения строки на части (токены). В качестве разделителя используется регулярное выражение, переданное вторым параметром в конструктор класса.

Пример разбиения строки на слова, используя в качестве разделителя символ пробела:

```
String s = "Best Java programming language.";
StringTokenizer st = new StringTokenizer(s);

while (st.hasMoreTokens())
{
 System.out.println(st.nextToken());
}
```

Пример разбиения строки на слова, используя в качестве разделителя запятую:

```
String s = "Best,Java,programming,language.";
StringTokenizer st = new StringTokenizer(s, ",");
while (st.hasMoreTokens())
{
 System.out.println(st.nextToken());
}
```

Особенности конкатенации строк с другими типами

Строки, полученные в результате конкатенации строковых литералов и переменных, в пул строк не помещаются.

Пример:

```
String t = "t";
String cat1 = "ca" + t;
String cat2 = "cat";
System.out.println(cat1 == cat2);
```

Результат: false

В случае конкатенации строк оператором + с другими примитивными типами данных, все последующие типы преобразуются в строчное представление. В связи с этим можно получить не то, что ожидается.

Пример:

```
String text = "sum =" + 1 + 2; // 1 и 2 преобразуются
        // в строку
System.out.println(text );
```

Результат: sum =12

Добавление круглых скобок в выражение меняют порядок выполнения. Сначала произойдет операция сложения литералов и только потом преобразование к строке.

Пример с скобками:

```
String text = "sum =" + (1 + 2);
System.out.println(text );
```

Результат: sum =3

Пример:

```
String text = 1 + 2 + " = 3";
System.out.println(text );
```

Результат: 3 = 3

Пример с комбинацией операторов:

```
String text = "mul =" + 3 * 2;
System.out.println(text );
```

Результат: mul =6

Сравнение строк

При применении оператора сравнения (==) для переменных ссылочного типа происходит сравнение ссылок на объект, хранящихся в переменных. В связи с этим, если создать две строки с идентичным содержанием, через оператор **new**, сравнение таких строк оператором == вернет **false**. В связи с этим для сравнение идентичности разных строк, следует использовать метод **equals**, который сравнивает строки на эквивалентность.

Пример сравнения строк, инициализированных литералами:

```
String text1 = "Java";
String text2 = "Java"; // в переменную присваивается
// ссылка на объект из пула строк
System.out.println(text1 == text2);
System.out.println(text1.equals(text2));
```

Результат: true, true

Пример сравнения строк, созданных через оператор new:

```
String text1 = new String("Java");
String text2 = new String("Java");
System.out.println(text1 == text2);
System.out.println(text1.equals(text2));
```

Результат: false, true

Строку, созданную с помощью конкатенации или оператора **new,** можно программно поместить в "пул строк", используя метод intern(). Метод intern() проверяет, есть ли данная строка в пуле строк, и если такая строка отсутствует, помещает ее в пул строк. Возвращает ссылку на строку из пула строк.

Пример:

```
String text1 = new String("Java");
text1 = text1.intern();
text2 = text2.intern();
String text2 = new String("Java");
System.out.println(text1 == text2);
System.out.println(text1.equals(text2));
```

Результат: true, true

***Важно!!!** В классах StringBuilder и StringBuffer метод equals не переопределен и, следуя стандартной реализации в классе Object, сравнивает ссылки на объекты. В связи с этим при необходимости сравнить два объекта классов StringBuilder или StringBuffer, преобразуйте оба объекта к строке методом toString.*

Пример:

```
StringBuffer sb1 = new StringBuffer("abc");
StringBuffer sb2 = new StringBuffer("abc");
System.out.println(sb1.equals(sb2)); // неверно
```

Результат: false

Регулярные выражения

*"Некоторые люди, столкнувшись с проблемой, думают: – "Я знаю как ее решить. Я буду использовать регулярные выражения". Теперь у таких людей две проблемы".**

Регулярные выражения** – это формальный язык, предназначенный для поиска и различных манипуляций с подстроками текста. Основой языка являются метасимволы. Регулярная строка является образцом или шаблоном, который задает правила поиска подстрок в исходной строке.

Регулярные выражения являются мощным и гибким инструментом для работы с текстом, в разы уменьшают количество исходного кода, но использование регулярных выражений часто усложняет понимание и чтение кода. Поэтому рекомендуется очень тщательно продумывать необходимость использования регулярных выражений.

Большинство символов в регулярных выражениях представляют самих себя, кроме метасимволов.

---

* Цитата принадлежит Джейми Завински (Jamie Zawinski, JWZ) – одному из первых разработчиков Netscape.

** Регулярные выражения (англ. regular expressions), в среде программистов, часто называют «регулярками» и «регэксп» (жаргонное).

Основные метасимволы регулярных выражений:

Мета-символ	Назначение
^	Начало проверяемой строки
$	Конец проверяемой строки
.	Сокращенная форма записи для символьного класса, совпадающего с любым символом
|	Подвыражения, объединенные этим способом, называются альтернативами (alternatives)
?	Предшествующий ему символ является необязательным
+	Обозначает «один или несколько экземпляров непосредственно предшествующего элемента»
*	Любое количество экземпляров элемента (в том числе и нулевое)
\\d	Цифровой символ
\\D	Нецифровой символ
\\s	Пробельный символ
\\S	Любой символ, кроме пробела
\\W	Любой символ, кроме буквенного или цифрового символа или знака подчёркивания
\\w	Буквенный или цифровой символ или знак подчёркивания

В классе String есть несколько методов использующих регулярные выражения:

- matches – метод возвращает истину, если исходная строка соответствует регулярному выражению, иначе возвращает ложь.

Пример проверяет, соответствует ли строка правилам составления электронного адреса:

```
String email = "unguryan@itstep.org";
// регулярное выражение для проверки правильности
// email адреса
```

```
String regular = "[a-zA-Z]{1}[a-zA-Z\\d\\u002E\\
     u005F]+@ ([a-zA-Z]+\\u002E){1,2}
     ((net)|(com)|(org))";
System.out.println(email.matches(regular));
```

Результат: true

- replaceAll – возвращает строку, в которой все подстроки, удовлетворяющие условию регулярного выражения, заменены на строку из второго параметра.

Пример:

```
String text= "I love coffee!";
text = text.replaceAll("[Cc]offee", "Java");
```

- split – возвращает массив строк, разбивая строку на части, используя в качестве разделителя регулярное выражение.

В Java есть пакет Java.util.regex, предназначенный для работы с регулярными выражениями.

Пример:

```
String url= "http://mystat.itstep.org/";
// регулярное выражение для проверки правильности
// URL адреса
String regular = "/^(?:([a-z]+):(?:([a-z]*):)?\/\/)?
     (?:([^:@]*)(?::([^:@]*))?@)?((?:[a-z0-
     9_-]+\.)+[a-z]{2,}|localhost|(?:(?:[01]?
     \d\d?|2[0-4]\d|25[0-5])\.){3}(?:(?:
     [01]?\d\d?|2[0-4]\d|25[0-5])))(?::(\
     d+))?(?:([^:\?\#]+))?(?:\?([^\#]+))?
     (?:\#([^\s]+))?$/i";
```

```
Pattern p = Pattern.compile(regular); // создает объект
          // образец
Matcher m = p.matcher(url); // создает объект совпадений
System.out.println(m.matches());
```

Результат: true