Зарезервированные слова в Javа

abstract  
assert\*\*\*  
boolean  
break  
byte  
case  
catch  
char  
class  
const\*  
continue  
default  
double  
do  
else  
enum\*\*\*\*  
extends  
final  
finally  
float  
for  
goto\*  
if  
implements  
import  
instanceof  
int  
interface  
long  
native  
new  
package  
private  
protected  
public  
return  
short  
static  
strictfp\*\*  
super  
switch  
synchronized  
this  
throw  
throws  
transient  
try  
void  
volatile  
while

Зарезервированные имена методов Java

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| clone | equals | finalize | getClass | hashCode |
| notify | notifyAll | toString | wait |  |

Идентификаторы

Идентификаторы используются для именования классов, методов и переменных. В качестве идентификатора может использоваться любая последовательность строчных и прописных букв, цифр и символов \_ (подчеркивание) и $ (доллар). Идентификаторы не должны начинаться с цифры, чтобы транслятор не перепутал их с числовыми литеральными константами, которые будут описаны ниже. Java — язык, чувствительный к регистру букв. Это означает, что, к примеру, Value и VALUE — различные идентификаторы.

Литералы (целые, с плавающей точкой, логические, символьные, строчные) и константы

**Литералы**

Константы в Java задаются их литеральным представлением. Целые числа, числа с плавающей точкой, логические значения, символы и строки можно располагать в любом месте исходного кода. Типы будут рассмотрены в [главе 4](http://www.mstu.edu.ru/study/materials/java/04.htm).

**Целые литералы**

Целые числа — это тип, используемый в обычных программах наиболее часто. Любое целочисленное значение, например, 1, 2, 3, 42 — это целый литерал. В данном примере приведены десятичные числа, то есть числа с основанием 10 — именно те, которые мы повседневно используем вне мира компьютеров. Кроме десятичных, в качестве целых литералов могут использоваться также числа с основанием 8 и 16 — восьмеричные и шестнадцатиричные. Java распознает восьмеричные числа по стоящему впереди нулю. Нормальные десятичные числа не могут начинаться с нуля, так что использование в программе внешне допустимого числа 09 приведет к сообщению об ошибке при трансляции, поскольку 9 не входит в диапазон 0.. 7, допустимый для знаков восьмеричного числа. Шестнадцатиричная константа различается по стоящим впереди символам нуль-х (0х или 0Х). Диапазон значений шестнадцатиричной цифры — 0.. 15, причем в качестве цифр для значений 10.. 15 используются буквы от А до F (или от а до f). С помощью шестнадцатиричных чисел вы можете в краткой и ясной форме представить значения, ориентированные на использование в компьютере, например, написав Oxffff вместо 65535.

Целые литералы являются значениями типа int, которое в Java хранится в 32-битовом слове. Если вам требуется значение, которое по модулю больше, чем приблизительно 2 миллиарда, необходимо воспользоваться константой типа long. При этом число будет храниться в 64-битовом слове. К числам с любым из названных выше оснований вы можете приписать справа строчную или прописную букву L, указав таким образом, что данное число относится к типу long. Например, Ox7ffffffffffffffL или 9223372036854775807L — это значение, наибольшее для числа типа long.

**Литералы с плавающей точкой**

Числа с плавающей точкой представляют десятичные значения, у которых есть дробная часть. Их можно записывать либо в обычном, либо экспоненциальном форматах. В обычном формате число состоит из некоторого количества десятичных цифр, стоящей после них десятичной точки, и следующих за ней десятичных цифр дробной части. Например, 2.0, 3.14159 и .6667 — это допустимые значения чисел с плавающей точкой, записанных в стандартном формате. В экспоненциальном формате после перечисленных элементов дополнительно указывается десятичный порядок. Порядок определяется положительным или отрицательным десятичным числом, следующим за символом Е или е. Примеры чисел в экспоненциальном формате: 6.022е23, 314159Е-05, 2е+100. В Java числа с плавающей точкой по умолчанию рассматриваются, как значения типа double. Если вам требуется константа типа float, справа к литералу надо приписать символ F или f. Если вы любитель избыточных определений — можете добавлять к литералам типа double символ D или d. Значения используемого по умолчанию типа double хранятся в 64-битовом слове, менее точные значения типа float — в 32-битовых.

**Логические литералы**

У логической переменной может быть лишь два значения — true (истина) и false (ложь). Логические значения true и false не преобразуются ни в какое числовое представление. Ключевое слово true в Java не равно 1, a false не равно 0. В Java эти значения могут присваиваться только переменным типа boolean либо использоваться в выражениях с логическими операторами.

**Символьные литералы**

Символы в Java — это индексы в таблице символов UNICODE. Они представляют собой 16-битовые значения, которые можно преобразовать в целые числа и к которым можно применять операторы целочисленной арифметики, например, операторы сложения и вычитания. Символьные литералы помещаются внутри пары апострофов (' '). Все видимые символы таблицы ASCII можно прямо вставлять внутрь пары апострофов: - 'a', 'z', '@'. Для символов, которые невозможно ввести непосредственно, предусмотрено несколько управляющих последовательностей.

Таблица 3. 2. Управляющие последовательности символов

|  |  |
| --- | --- |
| Управляющая последовательность | Описание |
| \ddd | Восьмеричный символ (ddd) |
| \uxxxx | Шестнадцатиричный символ UNICODE (xxxx) |
| \' | Апостроф |
| \" | Кавычка |
| \\ | Обратная косая черта |
| \r | Возврат каретки (carriage return) |
| \n | Перевод строки (line feed, new line) |
| \f | Перевод страницы (form feed) |
| \t | Горизонтальная табуляция (tab) |
| \b | Возврат на шаг (backspace) |

**Строчные литералы**

Строчные литералы в Java выглядят точно также, как и во многих других языках — это произвольный текст, заключенный в пару двойных кавычек (""). Хотя строчные литералы в Java реализованы весьма своеобразно (Java создает объект для каждой строки), внешне это никак не проявляется. Примеры строчных литералов: “Hello World!”; "две\строки; \ А это в кавычках\"". Все управляющие последовательности и восьмеричные / шестнадцатиричные формы записи, которые определены для символьных литералов, работают точно так же и в строках. Строчные литералы в Java должны начинаться и заканчиваться в одной и той же строке исходного кода. В этом языке, в отличие от многих других, нет управляющей последовательности для продолжения строкового литерала на новой строке.

Константа

-постоянное значение, известное до начала работы программы и заданное в коде один раз.

**Почему это удобно:**

* Если вы используете значение многократно, а в ходе разработки его потребуется изменить, достаточно будет одной правки.
* Вместо длинного значения можно ставить в коде короткое имя константы.
* Нет риска ошибиться при очередном вводе значения.

По логике это надо бы называть «постоянной» — в противоположность переменной, — но традиция уже сложилась.

Константа может хранить число, строку, символ, ссылку на объект и др. Значение константы еще называют литералом (от англ. literal — буквальный), потому что его записывают как есть — без предварительных вычислений.

При объявлении констант в Java используют ключевое слово final — оно показывает, что литерал не должен меняться. Именовать константы принято заглавными буквами со знаком подчеркивания вместо пробела: ВОТ\_ТАК.

Пример строковой константы:

**public** **static** **final** **String** MY\_CONSTANT="Мой текст";

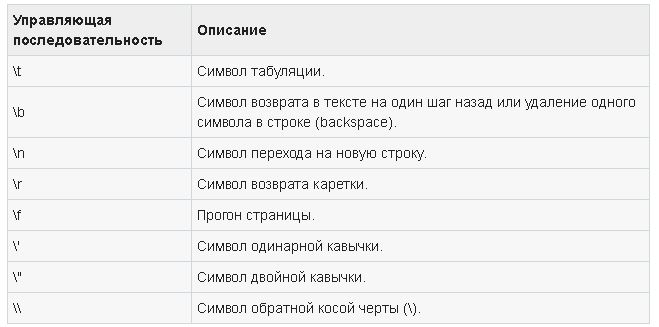
//создаем строку и присваиваем ей постоянное значение

Модификаторы public и static означают, что перед нами публичная константа уровня класса — без привязки к отдельным объектам, классам и т.д. Это аналог глобальной константы в Java. Значение константы всегда указывают при её создании. Если инициализация отложена, перед нами не константа, а «финальная переменная». Её значение тоже обычно задают только один раз, но:

* оно может быть получено в результате вычислений;
* если внутри финальной переменной лежит объект, его свойства можно менять, т.к. финализация действует на ссылку, а не на значение.

Важно следить, чтобы значение константы нигде не конфликтовало с логикой вычислений. Например, скорость звука может быть константой в одной среде, но в разных средах она разная. Если в коде мы сравниваем что-то со скоростью звука, хорошим вариантом будет создать финальную переменную, значение которой можно будет переопределять в зависимости от среды.

Управляющие символы



Операции и примитивные типы данных (целые типы,числа сплавающей точкой,символы и кодировки)

Примитивные типы Java не являются объектами. К ним относятся:

* boolean - булев тип, может иметь значения true или false
* byte - 8-разрядное целое число
* short - 16-разрядное целое число
* int - 32-разрядное целое число
* long - 64-разрядное целое число
* char - 16-разрядное беззнаковое целое, представляющее собой символ UTF-16 (буквы и цифры)
* float - 32-разрядное число в формате IEEE 754 с плавающей точкой
* double - 64-разрядное число в формате IEEE 754 с плавающей точкой

Примитивный в данном случае не оскорбление, а просто даёт понять, что речь идёт о простом типе, который не умеет прыгать, спать или мяукать. Да что он вообще умеет? Ой, всё.

## **Простые числовые типы**

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип** | **Разрядность** | **MIN** | **MAX** |
| byte | 8 бит | -128 | 127 |
| short | 16 бит | -32768 | 32767 |
| int | 32 бит | -2147483648 | 2147483647 |
| long | 64 бит | -9223372036854775808 | 9223372036854775807 |
| float | 32 бит | -3.4E+38 | 3.4E+38 |
| double | 64 бит | -1.7E+308 | 1.7E+308 |

## **Целочисленные типы**

Java определяет четыре целочисленных типа: **byte**, **short**, **int**, **long**. Они могут быть положительными и отрицательными (Java не поддерживает только положительные значения без знака, как некоторые языки программирования).

### **Тип byte**

Наименьший по размеру целочисленный тип - **byte**. Это 8-битовый тип с диапазоном допустимых значений от -128 до 127. Переменные типа **byte** часто используются при работе с потоком данных из сети или файла, а также при работе с необработанными двоичными данными или в массивах для экономии памяти.

Объявить переменную типа **byte** можно следующим образом:

byte c, a, t; // объявили сразу три переменные

В арифметических выражениях с переменными типа **short** вычисления выполняются как с типом **int**, т.е. с помощью 32-битовой арифметики, а полученный результат будет 32-битовым. Смотри пример с **short**.

Строку с числом перевести в данный тип можно через метод **parseByte(String)**:

byte x = Byte.parseByte("100");

Класс **Byte** является оболочкой для данного типа. Без необходимости не используйте в Android класс **Byte**.

### **Тип short**

Тип **short** - 16-битовый тип в диапазоне от -32768 до 32767. Используется очень редко.

short m;

В арифметических выражениях с переменными типа **short** вычисления выполняются как с типом **int**, т.е. с помощью 32-битовой арифметики, а полученный результат будет 32-битовым. Например, такой код не пройдёт.

// накорми кота

short fishNumber = 3; // три рыбки

short beefNumber = 2; // два кусочка говядины

short breakfast = 0;

breakfast = fishNumber + beefNumber; // завтрак чемпиона

Java будет ругаться на последнюю строчку, так как итоговый результат не может быть **short**. Как вариант, вам нужно преобразовать результат снова в 16-битовое число.

breakfast = (short) (fishNumber + beefNumber); // завтрак чемпиона

Это называется [приведением чисел](http://developer.alexanderklimov.ru/android/java/types.php#cast).

Явно перевести строку с числом в тип **short** можно через метод **parseShort(String)**:

short x = Short.parseShort("100");

Класс **Short** является оболочкой для данного типа. Без необходимости не используйте в Android класс **Short**.

### **Тип int**

Целые числа, представленные типом **int**, являются самым распространённым типом в программе, с которым вы будете работать. Поэтому нужно хорошенько изучить его и узнать его достоинства и ограничения. Это 32-битовый тип, имеющий диапазон допустимых значений от -2147483648 до 2147483647 (около двух миллиардов). Этого числа вполне достаточно, чтобы посчитать всех котов на свете. Часто используется в циклах, индексировании массивов, хотя может показаться, что для небольших операций в цикле и массивах проще использовать **short** или **byte**. Нужно запомнить, что тип **int** эффективен в этих случаях из-за особенностей структуры вычислительных процессоров. Просто примите на веру.

**Разделители**

Java допускает применение нескольких символов в качестве разделителей. Чаще всего в качестве разделителя используется точка с запятой. Как вы уже видели, она применяется для завершения строк операторов. Допустимые символы-разделители описаны в табл. 2.2.

Таблица 2.2. Допустимые символы-разделители

|  |  |  |
| --- | --- | --- |
| **Символ** | **Название** | **Назначение** |
| ( ) | Круглые скобки | Используются для передачи списков параметров в определениях и вызовах методов. Их применяют также для определения приоритета в выражениях, указания выражений в управляющих операторах и указания преобразования типов. |
| { } | Фигурные скобки | Используются для указания значений автоматически инициализируемых массивов. Их применяют также для определения блоков кода, классов, методов и локальных областей определения. |
| [ ] | Квадратные скобки | Используются для объявления типов массивов, а также при разыменовании значений массивов. |
| ; | Точка с запятой | Завершает операторы. |
| , | Запятая | Разделяет последовательные идентификаторы в объявлениях переменных. Этот символ-разделитель используют также для создания цепочек операторов внутри оператора for. |
| . | Точка | Используется для разделения имен пакетов от подпакетов и классов, а также для отделения переменной или метода от ссылочной переменной. |

Переменные

Переменная - это основной элемент хранения информации в Java- программе. Переменная характеризуется комбинацией идентификатора, типа и области действия. В зависимости от того, где объявлена переменная, она может быть локальной, например, для кода внутри метода, либо это может быть переменная экземпляра класса, доступная всем методам данного класса. Локальные области действия объявляются с помощью фигурных скобок.

Классы оболочки

(ссылочные типы данных)

http://proselyte.net/tutorials/java-core/wrapper-classes/

**Массивы (многомерные массивы)**

Массивом называется множество однотипных объектов, объединенных одним именем и доступ к каждому объекту в этом множестве осуществляется по порядковому номеру (индексу).

int[][] a = new int[5][5];// двумерный массив

int[][][] b = new int[3][4][5];// трехмерный массив

int[][][][] c = new int[2][4][5][5];// четырехмерный массив

**Тип 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 "桡扲慨慢�"

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](https://docs.oracle.com/javase/tutorial/java/data/manipstrings.html) и [сравнении](https://docs.oracle.com/javase/tutorial/java/data/comparestrings.html) строк. Также у вас под рукой всегда есть их [документация](http://docs.oracle.com/javase/8/docs/api/java/lang/String.html). 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](http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html) 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 parseXхх 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](http://docs.oracle.com/javase/8/docs/api/java/lang/StringBuffer.html).

### 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.");

}

}

**Консольный ввод -вывод**

Для получения данных, введенных пользователем, а также для вывода сообщений нам необходим ряд классов, через которые мы сможем взаимодействовать с консолью. Частично их использование уже рассматривалось в предыдущих темах. Для взаимодействия с консолью нам необходим класс System. Этот класс располагается в пакете *java.lang*, который автоматически подключается в программу, поэтому нам не надо дополнительно импортировать данный пакет и класс.

### Вывод на консоль

Для создания потока вывода в класс System определен объект out. В этом объекте определен метод println, который позволяет вывести на консоль некоторое значение с последующим переводом консоли на следующую строку:

|  |  |
| --- | --- |
| 1 | System.out.println("Hello world"); |

В метод println передается любое значение, как правило, строка, которое надо вывести на консоль. При необходимости можно и не переводить курсор на следующую строку. В этом случае можно использовать метод System.out.print(), который аналогичен println за тем исключением, что не осуществляет перевода на следующую строку.

|  |  |
| --- | --- |
| 1 | System.out.print("Hello world"); |

Но с помощью метода System.out.print также можно осуществить перевод каретки на следующую строку. Для этого надо использовать escape-последовательность *\n*:

|  |  |
| --- | --- |
| 1 | System.out.print("Hello world \n"); |

Если у нас есть два числа, и мы хотим вывести их значения на экран, то мы можем, например, написать так:

|  |  |
| --- | --- |
| 1  2  3 | int x=5;  int y=6;  System.out.println("x="+x +"; y="+y); |

Но в Java есть также функция для форматированного вывода, унаследованная от языка С: System.out.printf(). С ее помощью мы можем переписать предыдущий пример следующим образом:

|  |  |
| --- | --- |
| 1  2  3 | int x=5;  int y=6;  System.out.printf("x=%d; y=%d \n", x, y); |

В данном случае символы %d обозначают спецификатор, вместо которого подставляет один из аргументов. Спецификаторов и соответствующих им аргументов может быть множество. В данном случае у нас только два аргумента, поэтому вместо первого %d подставляет значение переменной x, а вместо второго - значение переменной y. Сама буква d означает, что данный спецификатор будет использоваться для вывода целочисленных значений типа int.

Кроме спецификатора *%d* мы можем использовать еще ряд спецификаторов для других типов данных:

* %x: для вывода шестнадцатеричных чисел
* %f: для вывода чисел с плавающей точкой
* %e: для вывода чисел в экспоненциальной форме, например, 1.3e+01
* %c: для вывода одиночного символа
* %s: для вывода строковых значений

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | String name = "Иван";  int age = 30;  float height = 1.7f;    System.out.printf("Имя: %s   Возраст: %d лет   Рост: %.2f метров \n", name, age, height); |

При выводе чисел с плавающей точкой мы можем указать количество знаков после запятой, для этого используем спецификатор на %.2f, где .2 указывает, что после запятой будет два знака. В итоге мы получим следующий вывод:

Имя: Иван Возраст: 30 лет Рост: 1,70 метров

### Консольный ввод

Для получения консольного ввода в классе System определен объект in. Однако непосредственно через объект System.in не очень удобно работать, поэтому, как правило, используют класс Scanner, который, в свою очередь использует System.in. Например, создадим маленькую программу, которая осуществляет ввод чисел:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import java.util.Scanner;    public class FirstApp {        public static void main(String[] args) {            Scanner in = new Scanner(System.in);          int[] nums = new int[5];          for(int i=0;i < nums.length; i++){              nums[i]=in.nextInt();          }            for(int i=0;i < nums.length; i++){              System.out.print(nums[i]);          }          System.out.println();      }  } |

Так как класс Scanner находится в пакете *java.util*, то мы вначале его импортируем. Для создания самого объекта Scanner в его конструктор передается объект System.in. После этого мы можем получать вводимые значения. Например, чтобы получить введенное число, используется метод in.nextInt();, который возвращает введенное с клавиатуры целочисленное значение.

В данном случае в цикле вводятся все элементы массива, а с помощью другого цикла все ранее введенные элементы массива выводятся в строчку.

Класс Scanner имеет еще ряд методов, которые позволяют получить введенные пользователем значения:

* next(): считывает введенную строку до первого пробела
* nextLine(): считывает всю введенную строку
* nextInt(): считывает введенное число int
* nextDouble(): считывает введенное число double
* hasNext(): проверяет, было ли введено слово
* hasNextInt(): проверяет, было ли введено число int
* hasNextDouble(): проверяет, было ли введено double

Кроме того, класс Scanner имеет еще ряд методов nextByte/nextShort/nextFloat/nextBoolean, которые по аналогии с nextInt считывают данные определенного типа данных.

**Документирование код**

**Javadoc** — стандартный генератор документации в HTML-формате из комментариев исходного кода.  
  
Для создания описания к элементу(поле, класс, метод) используются специальный комментарий, расположенный выше этого элемента:

/\*\* Описание \*/

Для документирования можно использовать дескрипторы, вот некоторые из них:  
**@author** — автор  
**@version** — версия  
**@since** — указывает с какой версии появился этот блок кода  
**@see** — ссылка на другое место в документации  
**@param** — передаваемый параметр методу  
**@return** — описание возвращаемого значения метода  
**@exception** и **@throws** — описание исключений  
**@deprecated** — документирование устаревших частей кода  
**{@link}** — создание ссылки, можно вставлять в любое место  
**{@value}** — описание значения переменной  
  
Рассмотрим пример:

/\*\* Класс служит для хранения объектов со свойствами  
 \* <b>maker</b> и <b>price</b>.  
 \* @autor Filippov Yakov  
 \* @version 1.0  
\*/  
class Product{  
        /\*\* Свойство - производитель \*/  
        private String maker;  
          
        /\*\* Свойство - цена \*/  
        public double price;  
          
        /\*\* Создает новый пустой объект  
         \* @see Product#Product(String, double)  
        \*/  
        Product(){  
                setMaker("");  
                price=0;  
        }  
          
        /\*\* Создает новый объект с заданными значениями  
         \* @param maker - производитель  
         \* @param price - цена  
         \* @see Product#Product()  
        \*/  
        Product(String maker,double price){  
                this.setMaker(maker);  
                this.price=price;  
        }  
          
        /\*\* Функция для получения значения поля {@link Product#maker}  
         \* @return Возвращает название производителя  
         \*/  
        public String getMaker() {  
                return maker;  
        }  
  
        public void setMaker(String maker) {  
                this.maker = maker;  
        }  
}

Как видно, в документации можно использовать HTML теги. При использовании ссылочных дескрипторов **@see** и **@link** нужно сначала указать имя класса и через символ "#" его метод или поле.  
Вот пример использования ссылок для документирования перегруженного конструктора:

/\*\* Создает новый объект с заданными значениями  
\* @param maker - производитель  
\* @param price - цена  
\* @see Product#Product()  
\*/

На выходе получаем:  
