1. **Ответы на первое задание**
   1. **Зарезервированные слова в Javа**

В языке Java, как и в любом языке программирования, есть слова, которые имеют особый смысл. Например, return или if или while. Такие слова называются ключевыми словами (keywords) и считаются зарезервированными языком Java.

Вы не можете использовать эти слова в качестве имени переменной, имени метода или класса. Компилятор всегда будет интерпретировать их строго определенным образом. Всего в Java таких слов 57.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| abstract | boolean | break | Byte | Byvalue |
| case | cast | catch | Char | Class |
| const | continue | default | Do | Double |
| else | extends | false | Final | Finally |
| float | for | future | generic | Goto |
| If | implements | import | Inner | instanceof |
| Int | interface | long | Native | New |
| null | operator | outer | package | Private |
| protected | public | rest | Return | Short |
| static | super | switch | syncronized | This |
| throw | throws | transient | True | Try |
| var | void | volatile | While |  |

* 1. **Зарезервированные имена методов Java**

Отметим, что слова byvalue, cast, const, future, generic, goto, inner, operator, outer, rest, var зарезервированы в Java, но пока не используются Кроме этого, в Java есть зарезервированные имена методов (эти методы наследуются каждым классом, их нельзя использовать, за исключением случаев явного переопределения методов класса Object).

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

**1.3 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.

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

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.

**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**

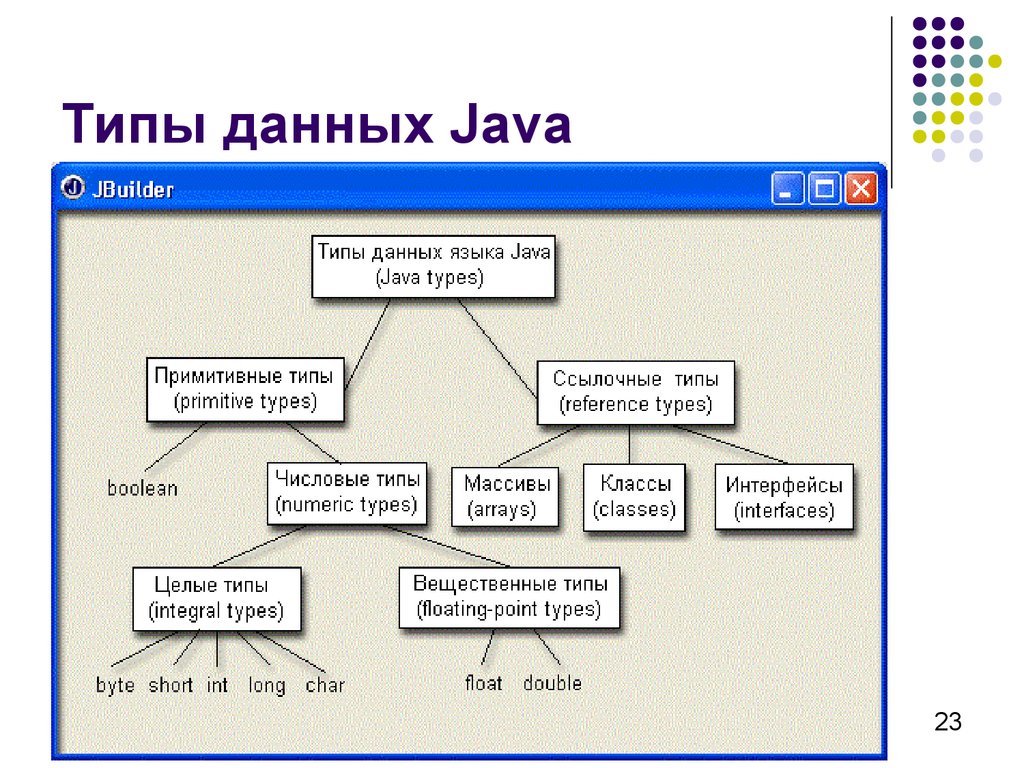
Символы в Java—это индексы в таблице символов UNICODE. Они представляют собой 16-битовые значения, которые можно преобразовать в целые числа и к которым можно применять операторы целочисленной арифметики, например, операторы сложения и вычитания. Символьные литералы помещаются внутри пары апострофов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

**1.5 Управляющие символы**

|  |  |
| --- | --- |
| **Управляющая последовательность** | **Описание** |
| \ddd | Восьмеричный символ (ddd) |
| \uxxxx | Шестнадцатеричный 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) |

**1.6 Операции и примитивные типы данных (целые типы, числа с плавающей точкой, символы и кодировки)**





Алфавит языка Java

ASCII, Unicode, UTF-16, 8

Латинские буквы ASCII:

ABCD...XYZ —заглавные (прописные); abcd...xyz—строчные.

\_$

Национальные буквы на примере русского алфавита:

АБВГ...ЭЮЯ абвг...эюя

0 1 2 3 4 5 6 7 8 9

( ) { } [ ] ; , .

**1.7 Разделители**

Java допускает применение нескольких символов в качестве разделителей. Чаще всего в качестве разделителя используется точка с запятой.

|  |  |  |
| --- | --- | --- |
| **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. |

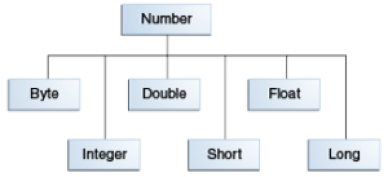
**1.8 Переменные**

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.

**1.9 Классы оболочки (ссылочные типы данных)**

Все эти классы являются наследниками класса **Number**.

Вот как выглядит иерархия класса **Number**.

[](https://i2.wp.com/proselyte.net/wp-content/uploads/2016/02/number.png)

Эти объекты “обёртывают” соответствующий им примитивы и преобразовывает его в объект (ссылочный тип данных). Этот процесс называется **автоупаковкой (boxing)**. Обратный процесс называется **распаковкой (unboxing)**.

**1.10 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

**1.11 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

##### 2. С помощью конструкторов:

String first = new String();

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"

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.

#### 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

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 "+="

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:

Используйте метод **concat**, если слияние нужно провести только один раз, для остальных случаев рекомендовано использовать или оператор "**+**" или **StringBuffer** / **StringBuilder**.

### 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.

### 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.

**1.12 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.

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.

### 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.

**1.13 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

**1.14 Code convention**

## **Имена файлов, пакетов**

* В именах пакетов используются только строчные буквы.
* Имена Java-классам даются согласно стандартной нотации Java.
* Имена классов должны быть существительными, первые буквы всех слов — заглавные.
* В именах web-папок и файлов используются только строчные буквы. Слова в многословных названиях разделяются подчеркиванием.

## **Имена методов, переменных**

* Названия методов должны быть глаголами, первая буква должна быть строчной, первые буквы внутренних слов — заглавные.
* Имена переменных должны начинаться со строчной буквы, внутренние слова — с заглавной.
* Имена констант составляются из всех заглавных букв, разделенных на слова символом подчеркивания.

## **Отступы, длина строки, переносы строк**

* Отступы должны составлять строго 4 пробела (не знак табуляции).
* Длина строки не должна превышать 80 символов.
* Если длина выражения превышает длину строки, то необходимо разбить его на несколько строк согласно следующим правилам:
  1. перенос после запятой;
  2. перенос перед оператором;
  3. необходимо использовать отступ 8 пробелов для обозначения второй строки разделенного выражения. Последующие строки выравниваются по второй строке либо добавляются новые 8 пробелов для обозначения вложенности.
  4. int result = function1(longExpression1,
  5. function2(longExpression2,

longExpression3));

## **Расположение блоков, операторов, пробелы, скобки**

* Определение переменных нужно располагать в начале блока, а не «ждать» первого использования переменной. Инициализация должна производиться, по возможности, сразу.
* void myMethod() {
* int count = 0; // beginning of method block
* if (condition) {
* int int2; // beginning of "if" block
* ...
* }

}

* Между именем метода и скобками для списка параметров нет пробела.

getStaffList(String name)

* Параметры разделяются пробелом.

getStaffList(String name, int count)

* Пробелы окружают любой оператор.
* res = getCount();
* (a > 10) ? b : c;
* Ключевое слово и следующая за ним скобка (должны разделяться пробелом.
* while (b < 100){
* …

}

* Открывающаяся скобка { располагается на той же строке, что и сигнатура метода/заголовок if, while-блока и т.п.
* Закрывающаяся скобка } выровнена по строке начала данного блока.
* public Collection getStaffList(String name) {
* Collection contacts = Contact.getList(name, RoleHelper.ROLE\_AGENT, null,
* AccessLevel.getStaffDiaryEditLevel(), 0, MAX\_NUMBER);
* return ListHelper.getLabelValueList(contacts);

}

* Методы разделяются пустой строкой, объявления свойств класса располагаются по одному на строку.
* На строке располагается только один оператор.
* if (b) {
* return result;

}

## **Структурирование кода**

* Методы должны быть короткими, и выполнять только одну задачу (к примеру, почти любой цикл уже достоин того, чтобы вынести его в особый метод).
* Имена методов должны быть самодокументированными.
* Шаблоны ООП должны применяться для структурирования и облегчения восприятия.

Аннотаций в Java, являются своего рода метками в коде, описывающими метаданные для функции/класса/пакета. Например, всем известная Аннотация @Override, обозначающая, что мы собираемся переопределить метод родительского класса. Да, с одной стороны можно и без неё, но, если у родителей не окажется этого метода, существует вероятность, что мы зря писали код, т.к. конкретно этот метод может и не вызваться никогда, а с Аннотацией @Override компилятор нам скажет, что: "Я не нашел такого метода в родителях... что-то здесь не чисто".