Лекция 9

Лексика языка

Лекция посвящена описанию лексики языка Java.

Лексика описывает, из чего состоит текст программы, каким образом он записывается, и на какие простейшие слова (лексемы) компилятор разбивает программу при анализе. Лексемы (или tokens в английском варианте) - это основные "кирпичики", из которых строится любая программа на языке Java.

1. Лексика языка

Лексика описывает, из чего состоит текст программы, каким образом он записывается, и на какие простейшие слова (лексемы) компилятор разбивает программу при анализе. **Лексемы** (или tokens в английском варианте) - это основные "кирпичики", из которых строится любая программа на языке Java.

1.1. Кодировка

Технология Java, как платформа, изначально спроектированная для Глобальной сети Интернет, должна быть многоязыковой, а значит обычный набор символов ASCII (American Standard Code for Information Interchange. Американский стандартный код обмена информацией), включающий в себя лишь латинский алфавит, цифры и простейшие специальные знаки (скобки, знаки препинания, арифметические операции и др.) не достаточен. Поэтому для записи текста программы применяется более универсальная кодировка Unicode.

Как известно, Unicode представляет символы кодом из 2 байт, описывая, таким образом, 65.535 символов. Это позволяет поддерживать практически все распространенные языки мира. Первые 128 символов совпадают с набором ASCII. Однако, понятно, что требуется некоторое специальное обозначение, чтобы иметь возможность задавать в программе любой символ Unicode, ведь никакая клавиатура не позволяет вводить более 65 тысяч различных знаков. Эта конструкция представляет символ Unicode, используя только символы ASCII. Например, если в программу нужно вставить знак с кодом 6917, необходимо его представить в шестнадцатеричном формате (1B05) и записать: \u1B05, причем буква u должна быть прописной, а шестнадцатеричные цифры A, B, C, D, E, F можно использовать произвольно, как заглавные, так и строчные. Таким образом можно закодировать все символы Unicode от \u0000 до \uFFFF. Буквы русского алфавита начинаются с \u0410 (только буква Ё имеет код \u0401) по \u044F (код буквы ё \u0451). В последних версиях JDK в состав демонстрационных приложений и апплетов входит небольшая программа SymbolTest, позволяющая просматривать весь набор символов Unicode. Ее аналог несложно написать самостоятельно. Для перекодирования больших текстов служит утилита native2ascii, также входящая в JDK. Она может работать как в прямом режиме - переводить из разнообразных кодировок в Unicode, записанный ASCII-символами. так и в обратном (опция -reverse) - из Unicode в стандартную кодировку операционной системы.

1.2. Анализ программы

Компилятор, анализируя программу, сразу разделяет ее на

- пробелы (white spaces);
- комментарии (comments);
- основные лексемы (tokens).

1.2.1. Пробелы

Пробелами в данном случае называют все символы, разбивающие текст программы на лексемы. Это как сам символ пробела (space, \u0020, десятичный код 32), так и знаки табуляции и перевода строки. Они используются для разделения лексем, а также для оформления кода. чтобы его было легче читать. Например, следующую часть программы (вычисление корней квадратного уравнения):

```
double a = 1, b = 1, c = 6;
double D = b * b - 4 * a * c;

if (D >= 0) {
double x1 = (-b + Math.sqrt (D)) / (2 * a);
double x2 = (-b - Math.sqrt (D)) / (2 * a);
}
```

можно записать и в таком виде:

```
double a=1,b=1,c=6;double D=b*b-4*a*c;if(D>=0){double
x1=(-b+Math.sqrt(D))/(2*a);
double x2=(-b-Math.sqrt(D))/(2*a);}
```

В обоих случаях компилятор сгенерирует абсолютно одинаковый код. Единственное соображение, которым должен руководствоваться разработчик - легкость чтения и дальнейшей поддержки такого кода.

Для разбиения текста на строки в ASCII используются два символа - "возврат каретки" (carriage return, CR, \u000d, десятичный код 13) и символ новой строки (linefeed, LF, \u000a, десятичный код 10). Чтобы не зависеть от особенностей используемой платформы, в Java применяется наиболее гибкий подход.

Завершением строки считается

- ASCII-символ LF, символ новой строки;
- ASCII-символ CR, "возврат каретки";
- символ CR, за которым сразу же следует символ LF.

Разбиение на строки важно для корректного разбиения на лексемы (как уже говорилось, завершение строки также служит разделителем между лексемами), для правильной работы со строковыми комментариями, а также для вывода отладочной информации (при выводе ошибок компиляции и времени исполнения указывается, на какой строке исходного кода они возникли).

Итак, пробелами в Java считаются:

- ASCII-символ SP, space, пробел, \u0020, десятичный код 32;
- ASCII-символ HT, horizontal tab, символ горизонтальной табуляции, \u0009, десятичный код 9;

- ASCII-символ FF, form feed, символ перевода страницы (был введен для работы с принтером), \u000c, десятичный код 12;
- завершение строки.

1.2.2. Комментарии

Комментарии не влияют на результирующий бинарный код и используются только для ввода пояснений к программе.

В Java комментарии бывают 2 видов:

- строчные;
- блочные.

Строчные комментарии начинаются с ASCII-символов // и длятся до конца текущей строки. Как правило, они используются для пояснения именно этой строки, например:

```
int y=1970; // год рождения
```

Блочные комментарии располагаются между ASCII-символами /* и */, могут занимать произвольное количество строк, например:

```
/*
Этот цикл не может начинаться с нуля
из-за особенностей алгоритма
*/
for (int i=1; i<10; i++) {
…
}
```

Часто блочные комментарии оформляют следующим образом (каждая строка начинается с *):

```
/*
*Описание алгоритма работы
*следующего цикла while
*/
while (x > 0) {
…
}
```

Блочный комментарий не обязательно должен располагаться на нескольких строках, он может даже находится в середине оператора:

```
float s = 2*Math.PI/*getRadius()*/; // Закомментировано для отладки
```

В этом примере блочный комментарий разбивает арифметические операции. Выражение `Math.PI` предоставляет значение константы `PI`, определенное в классе `Math`. Вызов метода `getRadius()` теперь закомментирован и не будет произведен, переменная `s` всегда будет принимать значение `2PI`. Завершает строку строчный комментарий.

Комментарии не могут находиться внутри символьных и строковых литералах. Комментарии не могут быть вложенными.

Любые комментарии полностью удаляются из программы по время компиляции, поэтому их можно использовать неограниченно, не опасаясь. что это повлияет на бинарный код. Основное их предназначение - сделать программу

простой для понимания, в том числе и для других разработчиков, которым придется в ней разбираться по какой-либо причине. Также комментарии зачастую используются для временного исключения частей кода, например

```
int x = 2; int y = 0;
/*
if (x > 0)
y = y + x*2; else y = -y - x*4;
*/
y = y*y;// + 2*x;
```

В этом примере закомментировано выражение if-else и оператор сложения +2*x.

Как уже говорилось выше, комментарии можно писать символами Unicode, то есть на любом языке, удобном разработчику.

Кроме этого, существует особый вид блочного комментария - комментарий разработчика. Он применяется для автоматического создания документации кода. В стандартную поставку JDK, начиная с версии 1.0, входит специальная утилита javadoc. На вход ей подается исходный код классов, а на выходе получается удобная документация в HTML формате, которая описывает все классы, все их поля и методы. При этом активно используются гиперссылки, что существенно упрощает изучение программы по ее такому описанию (например, читая описание метода, можно одним нажатием мыши перейти на описание типов, используемых в качестве аргументов или возвращаемого значения). Однако понятно, что одного названия метода и перечисления его аргументов не достаточно для понимания его работы. Необходимы дополнительные пояснения от разработчика.

Комментарий разработчика записывается так же, как и блочный. Единственное различие в начальной комбинации символов - для документации комментарий необходимо начинать с `/**`. Например:

```
/**
* Вычисление модуля целого числа.
* Этот метод возвращает
* абсолютное значение аргумента x.
*/
int getAbs(int x)   {
if (x>=0)
return x; else return -x;
}
```

Первое предложение должно содержать краткое резюме всего комментария. В дальнейшем оно будет использовано как пояснение этой функции в списке всех методов класса (ниже будут описаны все конструкции языка, для которых применяется комментарий разработчика).

Поскольку в результате создается HTML-документация, то и комментарий необходимо писать по правилам HTML. Допускается применение тегов, таких как <b> и <p> . Однако, теги заголовков с <h1> по <h6> и <hr> использовать нельзя, так как они активно применяются javadoc для создания структуры документации.

Символ * в начале каждой строки и предшествующие ему пробелы и знаки табуляции игнорируются. Их можно вообще не использовать, но они удобны, когда важно форматирование, например, в примерах кода.

```
/**
* Первое предложение - краткое описание метода.
* <p>
* Так оформляется пример кода:
* <blockquote>
* <pre>
* if (condition==true)    {
*    x = getWidht();
*    y = x.getHeight();
* }
* </pre></blockquote>
* А так описывается HTML-список:
* <ul>
* <li>Можно использовать наклонный шрифт <i>курсив</i>,
* <li>или жирный <b>жирный</b>.
* </ul>
*/
public void calculate (int x,   int y)   {
…
}
```

Из этого комментария будет сгенерирован HTML-код. выглядящий примерно так:

Первое предложение - краткое описание метода.

Так оформляется пример кода:

```
if (condition==true) {
     x = getWidht();
     y = x.getHeight();
}
```

А так описывается HTML-список:

- Можно использовать наклонный шрифт курсив,
- или жирный жирный.

Наконец? javadoc поддерживает специальные теги. Они начинаются с символа @. Подробное описание этих тегов можно найти в документации. Например, можно использовать тег @see, чтобы сослаться на другой класс, поле или метод, или даже на другой интернет-сайт.

```
/**
*Краткое описание.
*
```

```
*Развернутый комментарий.
*
*@see java.lang.String
*@see java.lang.Math#PI
*@see <a href="java.sun.com">Official Java site</a>
*/
```

Первая ссылка указывает на класс `String` (`java.lang` - название библиотеки, в которой находится этот класс), вторая - на поле `PI` класса `Math` (символ # разделяет название класса и его полей или методов), третья ссылается на официальный сайт Java.

Комментарии разработчика могут быть записаны перед объявлением классов, интерфейсов, полей, методов и конструкторов. Если записать комментарий `/** ... */` в другой части кода, то ошибки не будет, но он не попадет в документацию, генерируемую javadoc. Кроме этого, можно описать и пакет (так называются библиотеки, или модули, в Java). Для этого необходимо создать специальный файл package.html, сохранить в нем комментарий, и поместить его в директории пакета. HTML-текст, содержащийся между тегами <body> и </body> , будет перемещен в документацию, а первое предложение будет использовано для краткой характеристики этого пакета.

Все классы стандартных библиотек Java поставляются в виде исходного текста, и можно увидеть, как хорошо они комментированы. Стандартная документация по этим классам сгенерирована утилитой javadoc. Для любой программы можно также легко подготовить подобное описание, необходимы лишь грамотные и аккуратные комментарии в исходном коде. Кроме того, Java предоставляет возможность генерировать с помощью javadoc документацию с нестандартным внешним видом.

1.2.3. Лексемы

Итак, были рассмотрены пробелы (в широком смысле этого слова, т.е. все символы, отвечающие за форматирование текста программы) и комментарии, применяемые для ввода пояснений к коду. С точки зрения программиста они применяются для того, чтобы сделать программу более читаемой и понятной для дальнейшего развития.

С точки зрения компилятора, а точнее его части, отвечающей за лексический разбор, основная роль пробелов и комментариев - служить разделителями между лексемами, причем сами разделители далее отбрасываются и не влияют на компилированный код.

Конечно, лексемы очень разнообразны, и именно они определяют многие свойства языка.

1.3. Виды лексем

Ниже перечислены все виды лексем в Java:

- идентификаторы (identifiers);
- ключевые слова (key words);

- литералы (literals);
- разделители (separators);
- операторы (operators).

Рассмотрим их по отдельности.

1.3.1. Идентификаторы

Идентификаторы - это имена, которые даются различным элементам языка для упрощения доступа к ним. Имена имеют пакеты, классы, интерфейсы, поля, методы, аргументы и локальные переменные (все эти понятия подробно рассматриваются в дальнейших главах). Идентификаторы можно записывать символами Unicode, то есть, на любом удобном языке. Длина имени не ограничена.

Идентификатор состоит из букв и цифр. Имя не может начинаться с цифры. Java-буквы, используемые в идентификаторах, включают в себя ASCII-символы A-Z (\u0041-\u005a), a-z (\u0061-\u007a), а также знаки подчеркивания _ (ASCII underscore, \u005f) и доллара $ (\u0024). Знак доллара используется только при автоматической генерации кода (чтобы исключить случайное совпадение имен), либо при использовании каких-либо старых библиотек, в которых допускались имена с этим символом. Java-цифры включают в себя обычные ASCII-цифры 0-9 (\u0030-\u0039).

Для идентификаторов не допускаются совпадения с зарезервированными словами (это ключевые слова, булевские литералы true и false и null-литерал null). Конечно, если 2 идентификатора включают в себя разные буквы, которые одинаково выглядят (например, латинская и русская буквы A), то они считаются различными.

1.3.2. Ключевые слова

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

Ключевые слова `goto` и `const` зарезервированы, но не используются. Это сделано для того, чтобы компилятор мог правильно отреагировать на использование этих распространенных в других языках слов. Напротив, оба булевских литерала true, false и null-литерал null часто считают ключевыми словами (возможно потому. что многие средства разработки подсвечивают их таким же образом), однако это именно литералы.

1.3.3. Литералы

Литералы позволяют задать в программе значения для числовых, символьных и строковых выражений, а также null-литералов. Всего в Java определены следующие виды литералов:

- целочисленный (integer);

- дробный (floating-point);
- булевский (boolean);
- символьный (character);
- строковый (string);
- null-литерал (null-literal).

1.3.4. Разделители

```
( )    [ ]   { } ;   . ,
```

1.3.5. Операторы

Операторы используются в различных операциях – арифметических, логических, битовых, операции сравнения, присваивания. Следующие 37 лексем (все состоят только из ASCII-символов) являются операторами языка Java:

```
=    >    <    !    ~    ?    :
==   <=   >=   !=   &&   ||   ++   --
+    -*   /    &    |    ^    %    <<   >>   >>>
+=   -=   *=   /=   &=   |=   ^=   %=   <<=   >>=   >>>=
```

Большинство из них вполне очевидны и хорошо известны из других языков программирования, однако некоторые нюансы в работе с операторами в Java все же присутствуют.

1.3.6. Заключение

В этой главе были рассмотрены все типы лексем, из которых состоит любая Java-программа. Еще раз напомним, что использование Unicode возможно и необходимо в следующих конструкциях:

- комментарии;
- идентификаторы;
- символьные и строковые литералы.

Остальные же (пробелы, ключевые слова, числовые, булевские и null-литералы, разделители и операторы) легко записываются с применением лишь ASCII-символов. В то же время любой Unicode-символ можно задать в виде специальной последовательности лишь ASCII-символов (условное обозначение - \uhhhh, где hhhh - код символа в шестнадцатеричном формате).

В заключение для примера приведем простейшую программу (традиционное Hello, world!), а затем классифицируем и подсчитаем используемые лексемы:

```
public class Demo {
/**
*Основной метод, с которого начинается выполнение
*любой Java программы.
*/
public static void main (String args[])    {
System.out.println("Hello, world!");
}
}
```

Итак, в приведенной программе есть один комментарий разработчика, 7 идентификаторов, 5 ключевых слов, 1 строковый литерал, 13 разделителей и ни одного оператора.

1.4. Работа с операторами

Рассмотрим некоторые детали использования операторов в Java.

1.4.1. Операторы присваивания и сравнения

Во-первых, конечно же, различаются оператор присваивания = и оператор сравнения ==.

```
x = 1;    // присваиваем переменной x значение 1
x == 1 //  сравниваем значение переменной x с единицей
```

Оператор присваивания не имеет никакого возвращаемого значения. Оператор сравнения всегда возвращает булевское значение true или false. Поэтому обычная опечатка в языке C, когда эти операторы спутывают:

```
// пример вызовет ошибку компилятора
if (x=0) { //здесь должен применятся оператор сравнения ==
…
}
```

в Java легко устраняется. Поскольку выражение `x=0` не имеет никакого значения (и тем более не воспринимается как всегда истинное), то компилятор сообщает об ошибке (необходимо писать `x==0`).

Условие "не равно" записывается как !=. Например:

```
if    (x!=0)    {
float f = 1./x;
}
```

Сочетание какого-либо оператора с оператором присваивания = используется при изменении значения переменной, например, следующие две строки эквивалентны:

```
x = x + 1;
x += 1;
```

1.4.2. Арифметические операции

Наряду с 4 обычными арифметическими операциями +. -. *. /. есть оператор получения остатка от деления %. который может быть применен как к целочисленным аргументам, так и к дробным.

Работа с целочисленными аргументами подчиняется простым правилам. Если делится значение a на значение b, то выражение (a/b)*b+(a%b) должно в точности равняться a. Здесь, конечно, оператор деления целых чисел / всегда возвращает целое число. Например:

```
9/5 возвращает 1
9/(-5)   возвращает -1
(-9)/5 возвращает -1
(-9)/(-5)   возвращает 1
```

Остаток может быть положительным, только если делимое было положительным. Соответственно, остаток может быть отрицательным только в случае отрицательного делимого.

```
9%5 возвращает 4
9%(-5)   возвращает 4
(-9)%5 возвращает -4
(-9)%(-5)   возвращает -4
```

Попытка получить остаток от деления на 0 приводит к ошибке.

Деление с остатком для дробных чисел может быть произведено по двум различным алгоритмам. Один из них повторяет правила для целых чисел, и именно он представлен оператором %. Если в рассмотренном примере деления 9 на 5 перейти к дробным числам. значение остатка во всех вариантах не изменится (оно будет также дробным, конечно).

```
9.0%5.0 возвращает 4.0
9.0%(-5.0) возвращает 4.0
(-9.0)%5.0 возвращает -4.0
(-9.0)%(-5.0) возвращает -4.0
```

Унарные операторы инкрементации ++ и декрементации --, как обычно, можно использовать как справа, так и слева.

```
int x=1;
int y=++x;
```

В этом примере оператор ++ стоит перед переменной `x`, что означает, что сначала произойдет инкрементация, а затем значение `x` будет использовано для инициализации `y`. В результате после выполнения этих строк значения `x` и `y` будут равны 2.

```
int x=1;
int y=x++;
```

А в этом примере сначала значение `x` будет использовано для инициализации `y`, а лишь затем произойдет инкрементация. В результате значение `x` будет равно 2. а `y` будет равно 1.

1.4.3. Логические операторы

Логические операторы "и" и "или" (& и |) можно использовать в двух вариантах. Это связано с тем, что, как легко убедиться, для каждого оператора возможны случаи, когда значение первого операнда сразу определяет значение всего логического выражения. Если вторым операндом является значение некоторой функции, то появляется выбор - вызывать ее или нет, причем это решение может сказаться как на скорости, так и на функциональности программы.

Первый вариант операторов (&, |) всегда вычисляет оба операнда, второй же - (&&, ||) не будет продолжать вычисления, если значение выражения уже очевидно. Например:

```
int x=1;
(x>0) | calculate(x) // в таком выражении произойдет вызов
                        calculate
(x>0) || calculate(x) // а в этом - нет
```

Логический оператор отрицания "не" записывается как !, и конечно имеет только один вариант использования. Этот оператор меняет булевское значение на противоположное.

```
int x=1;
x>0 // выражение истинно
!(x>0)    // выражение ложно
```

Оператор с условием ? : состоит из трех частей - условия и двух выражений. Сначала вычисляется условие (булевское выражение), и на основании результата значение всего оператора определяется первым выражением в случае получения истины, и вторым - если условие ложно. Например, так можно вычислить модуль числа x:

```
x>0  ? x  :  -x
```

Лекция 9

Лексика языка

Лекция посвящена описанию лексики языка Java.

Лексика описывает, из чего состоит текст программы, каким образом он записывается, и на какие простейшие слова (лексемы) компилятор разбивает программу при анализе. Лексемы (или tokens в английском варианте) - это основные "кирпичики", из которых строится любая программа на языке Java.

1. Лексика языка

Лексика описывает, из чего состоит текст программы, каким образом он записывается, и на какие простейшие слова (лексемы) компилятор разбивает программу при анализе. **Лексемы** (или tokens в английском варианте) - это основные "кирпичики", из которых строится любая программа на языке Java.

1.1. Кодировка

Технология Java, как платформа, изначально спроектированная для Глобальной сети Интернет, должна быть многоязыковой, а значит обычный набор символов ASCII (American Standard Code for Information Interchange. Американский стандартный код обмена информацией), включающий в себя лишь латинский алфавит, цифры и простейшие специальные знаки (скобки, знаки препинания, арифметические операции и др.) не достаточен. Поэтому для записи текста программы применяется более универсальная кодировка Unicode.

Как известно, Unicode представляет символы кодом из 2 байт, описывая, таким образом, 65.535 символов. Это позволяет поддерживать практически все распространенные языки мира. Первые 128 символов совпадают с набором ASCII. Однако, понятно, что требуется некоторое специальное обозначение, чтобы иметь возможность задавать в программе любой символ Unicode, ведь никакая клавиатура не позволяет вводить более 65 тысяч различных знаков. Эта конструкция представляет символ Unicode, используя только символы ASCII. Например, если в программу нужно вставить знак с кодом 6917, необходимо его представить в шестнадцатеричном формате (1B05) и записать: \u1B05, причем буква u должна быть прописной, а шестнадцатеричные цифры A, B, C, D, E, F можно использовать произвольно, как заглавные, так и строчные. Таким образом можно закодировать все символы Unicode от \u0000 до \uFFFF. Буквы русского алфавита начинаются с \u0410 (только буква Ё имеет код \u0401) по \u044F (код буквы ё \u0451). В последних версиях JDK в состав демонстрационных приложений и апплетов входит небольшая программа SymbolTest, позволяющая просматривать весь набор символов Unicode. Ее аналог несложно написать самостоятельно. Для перекодирования больших текстов служит утилита native2ascii, также входящая в JDK. Она может работать как в прямом режиме - переводить из разнообразных кодировок в Unicode, записанный ASCII-символами. так и в обратном (опция -reverse) - из Unicode в стандартную кодировку операционной системы.

1.2. Анализ программы

Компилятор, анализируя программу, сразу разделяет ее на

- пробелы (white spaces);
- комментарии (comments);
- основные лексемы (tokens).

1.2.1. Пробелы

Пробелами в данном случае называют все символы, разбивающие текст программы на лексемы. Это как сам символ пробела (space, \u0020, десятичный код 32), так и знаки табуляции и перевода строки. Они используются для разделения лексем, а также для оформления кода. чтобы его было легче читать. Например, следующую часть программы (вычисление корней квадратного уравнения):

```
double a = 1, b = 1, c = 6;
double D = b * b - 4 * a * c;

if (D >= 0) {
double x1 = (-b + Math.sqrt (D)) / (2 * a);
double x2 = (-b - Math.sqrt (D)) / (2 * a);
}
```

можно записать и в таком виде:

```
double a=1,b=1,c=6;double D=b*b-4*a*c;if(D>=0){double
x1=(-b+Math.sqrt(D))/(2*a);
double x2=(-b-Math.sqrt(D))/(2*a);}
```

В обоих случаях компилятор сгенерирует абсолютно одинаковый код. Единственное соображение, которым должен руководствоваться разработчик - легкость чтения и дальнейшей поддержки такого кода.

Для разбиения текста на строки в ASCII используются два символа - "возврат каретки" (carriage return, CR, \u000d, десятичный код 13) и символ новой строки (linefeed, LF, \u000a, десятичный код 10). Чтобы не зависеть от особенностей используемой платформы, в Java применяется наиболее гибкий подход.

Завершением строки считается

- ASCII-символ LF, символ новой строки;
- ASCII-символ CR, "возврат каретки";
- символ CR, за которым сразу же следует символ LF.

Разбиение на строки важно для корректного разбиения на лексемы (как уже говорилось, завершение строки также служит разделителем между лексемами), для правильной работы со строковыми комментариями, а также для вывода отладочной информации (при выводе ошибок компиляции и времени исполнения указывается, на какой строке исходного кода они возникли).

Итак, пробелами в Java считаются:

- ASCII-символ SP, space, пробел, \u0020, десятичный код 32;
- ASCII-символ HT, horizontal tab, символ горизонтальной табуляции, \u0009, десятичный код 9;

- ASCII-символ FF, form feed, символ перевода страницы (был введен для работы с принтером), \u000c, десятичный код 12;
- завершение строки.

1.2.2. Комментарии

Комментарии не влияют на результирующий бинарный код и используются только для ввода пояснений к программе.

В Java комментарии бывают 2 видов:

- строчные;
- блочные.

Строчные комментарии начинаются с ASCII-символов // и длятся до конца текущей строки. Как правило, они используются для пояснения именно этой строки, например:

```
int y=1970; // год рождения
```

Блочные комментарии располагаются между ASCII-символами /* и */, могут занимать произвольное количество строк, например:

```
/*
Этот цикл не может начинаться с нуля
из-за особенностей алгоритма
*/
for (int i=1; i<10; i++) {
…
}
```

Часто блочные комментарии оформляют следующим образом (каждая строка начинается с *):

```
/*
*Описание алгоритма работы
*следующего цикла while
*/
while (x > 0) {
…
}
```

Блочный комментарий не обязательно должен располагаться на нескольких строках, он может даже находится в середине оператора:

```
float s = 2*Math.PI/*getRadius()*/; // Закомментировано для отладки
```

В этом примере блочный комментарий разбивает арифметические операции. Выражение `Math.PI` предоставляет значение константы `PI`, определенное в классе `Math`. Вызов метода `getRadius()` теперь закомментирован и не будет произведен, переменная `s` всегда будет принимать значение `2PI`. Завершает строку строчный комментарий.

Комментарии не могут находиться внутри символьных и строковых литералах. Комментарии не могут быть вложенными.

Любые комментарии полностью удаляются из программы по время компиляции, поэтому их можно использовать неограниченно, не опасаясь. что это повлияет на бинарный код. Основное их предназначение - сделать программу

простой для понимания, в том числе и для других разработчиков, которым придется в ней разбираться по какой-либо причине. Также комментарии зачастую используются для временного исключения частей кода, например

```
int x = 2; int y = 0;
/*
if (x > 0)
y = y + x*2; else y = -y - x*4;
*/
y = y*y;// + 2*x;
```

В этом примере закомментировано выражение if-else и оператор сложения +2*x.

Как уже говорилось выше, комментарии можно писать символами Unicode, то есть на любом языке, удобном разработчику.

Кроме этого, существует особый вид блочного комментария - комментарий разработчика. Он применяется для автоматического создания документации кода. В стандартную поставку JDK, начиная с версии 1.0, входит специальная утилита javadoc. На вход ей подается исходный код классов, а на выходе получается удобная документация в HTML формате, которая описывает все классы, все их поля и методы. При этом активно используются гиперссылки, что существенно упрощает изучение программы по ее такому описанию (например, читая описание метода, можно одним нажатием мыши перейти на описание типов, используемых в качестве аргументов или возвращаемого значения). Однако понятно, что одного названия метода и перечисления его аргументов не достаточно для понимания его работы. Необходимы дополнительные пояснения от разработчика.

Комментарий разработчика записывается так же, как и блочный. Единственное различие в начальной комбинации символов - для документации комментарий необходимо начинать с `/**`. Например:

```
/**
* Вычисление модуля целого числа.
* Этот метод возвращает
* абсолютное значение аргумента x.
*/
int getAbs(int x)   {
if (x>=0)
return x; else return -x;
}
```

Первое предложение должно содержать краткое резюме всего комментария. В дальнейшем оно будет использовано как пояснение этой функции в списке всех методов класса (ниже будут описаны все конструкции языка, для которых применяется комментарий разработчика).

Поскольку в результате создается HTML-документация, то и комментарий необходимо писать по правилам HTML. Допускается применение тегов, таких как <b> и <p> . Однако, теги заголовков с <h1> по <h6> и <hr> использовать нельзя, так как они активно применяются javadoc для создания структуры документации.

Символ * в начале каждой строки и предшествующие ему пробелы и знаки табуляции игнорируются. Их можно вообще не использовать, но они удобны, когда важно форматирование, например, в примерах кода.

```
/**
* Первое предложение - краткое описание метода.
* <p>
* Так оформляется пример кода:
* <blockquote>
* <pre>
* if (condition==true)    {
*    x = getWidht();
*    y = x.getHeight();
* }
* </pre></blockquote>
* А так описывается HTML-список:
* <ul>
* <li>Можно использовать наклонный шрифт <i>курсив</i>,
* <li>или жирный <b>жирный</b>.
* </ul>
*/
public void calculate (int x,   int y)   {
…
}
```

Из этого комментария будет сгенерирован HTML-код. выглядящий примерно так:

Первое предложение - краткое описание метода.

Так оформляется пример кода:

```
if (condition==true) {
     x = getWidht();
     y = x.getHeight();
}
```

А так описывается HTML-список:

- Можно использовать наклонный шрифт курсив,
- или жирный жирный.

Наконец? javadoc поддерживает специальные теги. Они начинаются с символа @. Подробное описание этих тегов можно найти в документации. Например, можно использовать тег @see, чтобы сослаться на другой класс, поле или метод, или даже на другой интернет-сайт.

```
/**
*Краткое описание.
*
```

```
*Развернутый комментарий.
*
*@see java.lang.String
*@see java.lang.Math#PI
*@see <a href="java.sun.com">Official Java site</a>
*/
```

Первая ссылка указывает на класс `String` (`java.lang` - название библиотеки, в которой находится этот класс), вторая - на поле `PI` класса `Math` (символ # разделяет название класса и его полей или методов), третья ссылается на официальный сайт Java.

Комментарии разработчика могут быть записаны перед объявлением классов, интерфейсов, полей, методов и конструкторов. Если записать комментарий `/** ... */` в другой части кода, то ошибки не будет, но он не попадет в документацию, генерируемую javadoc. Кроме этого, можно описать и пакет (так называются библиотеки, или модули, в Java). Для этого необходимо создать специальный файл package.html, сохранить в нем комментарий, и поместить его в директории пакета. HTML-текст, содержащийся между тегами <body> и </body> , будет перемещен в документацию, а первое предложение будет использовано для краткой характеристики этого пакета.

Все классы стандартных библиотек Java поставляются в виде исходного текста, и можно увидеть, как хорошо они комментированы. Стандартная документация по этим классам сгенерирована утилитой javadoc. Для любой программы можно также легко подготовить подобное описание, необходимы лишь грамотные и аккуратные комментарии в исходном коде. Кроме того, Java предоставляет возможность генерировать с помощью javadoc документацию с нестандартным внешним видом.

1.2.3. Лексемы

Итак, были рассмотрены пробелы (в широком смысле этого слова, т.е. все символы, отвечающие за форматирование текста программы) и комментарии, применяемые для ввода пояснений к коду. С точки зрения программиста они применяются для того, чтобы сделать программу более читаемой и понятной для дальнейшего развития.

С точки зрения компилятора, а точнее его части, отвечающей за лексический разбор, основная роль пробелов и комментариев - служить разделителями между лексемами, причем сами разделители далее отбрасываются и не влияют на компилированный код.

Конечно, лексемы очень разнообразны, и именно они определяют многие свойства языка.

1.3. Виды лексем

Ниже перечислены все виды лексем в Java:

- идентификаторы (identifiers);
- ключевые слова (key words);

- литералы (literals);
- разделители (separators);
- операторы (operators).

Рассмотрим их по отдельности.

1.3.1. Идентификаторы

Идентификаторы - это имена, которые даются различным элементам языка для упрощения доступа к ним. Имена имеют пакеты, классы, интерфейсы, поля, методы, аргументы и локальные переменные (все эти понятия подробно рассматриваются в дальнейших главах). Идентификаторы можно записывать символами Unicode, то есть, на любом удобном языке. Длина имени не ограничена.

Идентификатор состоит из букв и цифр. Имя не может начинаться с цифры. Java-буквы, используемые в идентификаторах, включают в себя ASCII-символы A-Z (\u0041-\u005a), a-z (\u0061-\u007a), а также знаки подчеркивания _ (ASCII underscore, \u005f) и доллара $ (\u0024). Знак доллара используется только при автоматической генерации кода (чтобы исключить случайное совпадение имен), либо при использовании каких-либо старых библиотек, в которых допускались имена с этим символом. Java-цифры включают в себя обычные ASCII-цифры 0-9 (\u0030-\u0039).

Для идентификаторов не допускаются совпадения с зарезервированными словами (это ключевые слова, булевские литералы true и false и null-литерал null). Конечно, если 2 идентификатора включают в себя разные буквы, которые одинаково выглядят (например, латинская и русская буквы A), то они считаются различными.

1.3.2. Ключевые слова

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

Ключевые слова `goto` и `const` зарезервированы, но не используются. Это сделано для того, чтобы компилятор мог правильно отреагировать на использование этих распространенных в других языках слов. Напротив, оба булевских литерала true, false и null-литерал null часто считают ключевыми словами (возможно потому. что многие средства разработки подсвечивают их таким же образом), однако это именно литералы.

1.3.3. Литералы

Литералы позволяют задать в программе значения для числовых, символьных и строковых выражений, а также null-литералов. Всего в Java определены следующие виды литералов:

- целочисленный (integer);

- дробный (floating-point);
- булевский (boolean);
- символьный (character);
- строковый (string);
- null-литерал (null-literal).

1.3.4. Разделители

```
( )    [ ]   { } ;   . ,
```

1.3.5. Операторы

Операторы используются в различных операциях – арифметических, логических, битовых, операции сравнения, присваивания. Следующие 37 лексем (все состоят только из ASCII-символов) являются операторами языка Java:

```
=    >    <    !    ~    ?    :
==   <=   >=   !=   &&   ||   ++   --
+    -*   /    &    |    ^    %    <<   >>   >>>
+=   -=   *=   /=   &=   |=   ^=   %=   <<=   >>=   >>>=
```

Большинство из них вполне очевидны и хорошо известны из других языков программирования, однако некоторые нюансы в работе с операторами в Java все же присутствуют.

1.3.6. Заключение

В этой главе были рассмотрены все типы лексем, из которых состоит любая Java-программа. Еще раз напомним, что использование Unicode возможно и необходимо в следующих конструкциях:

- комментарии;
- идентификаторы;
- символьные и строковые литералы.

Остальные же (пробелы, ключевые слова, числовые, булевские и null-литералы, разделители и операторы) легко записываются с применением лишь ASCII-символов. В то же время любой Unicode-символ можно задать в виде специальной последовательности лишь ASCII-символов (условное обозначение - \uhhhh, где hhhh - код символа в шестнадцатеричном формате).

В заключение для примера приведем простейшую программу (традиционное Hello, world!), а затем классифицируем и подсчитаем используемые лексемы:

```
public class Demo {
/**
*Основной метод, с которого начинается выполнение
*любой Java программы.
*/
public static void main (String args[])    {
System.out.println("Hello, world!");
}
}
```

Итак, в приведенной программе есть один комментарий разработчика, 7 идентификаторов, 5 ключевых слов, 1 строковый литерал, 13 разделителей и ни одного оператора.

1.4. Работа с операторами

Рассмотрим некоторые детали использования операторов в Java.

1.4.1. Операторы присваивания и сравнения

Во-первых, конечно же, различаются оператор присваивания = и оператор сравнения ==.

```
x = 1;    // присваиваем переменной x значение 1
x == 1 //  сравниваем значение переменной x с единицей
```

Оператор присваивания не имеет никакого возвращаемого значения. Оператор сравнения всегда возвращает булевское значение true или false. Поэтому обычная опечатка в языке C, когда эти операторы спутывают:

```
// пример вызовет ошибку компилятора
if (x=0) { //здесь должен применятся оператор сравнения ==
…
}
```

в Java легко устраняется. Поскольку выражение `x=0` не имеет никакого значения (и тем более не воспринимается как всегда истинное), то компилятор сообщает об ошибке (необходимо писать `x==0`).

Условие "не равно" записывается как !=. Например:

```
if    (x!=0)    {
float f = 1./x;
}
```

Сочетание какого-либо оператора с оператором присваивания = используется при изменении значения переменной, например, следующие две строки эквивалентны:

```
x = x + 1;
x += 1;
```

1.4.2. Арифметические операции

Наряду с 4 обычными арифметическими операциями +. -. *. /. есть оператор получения остатка от деления %. который может быть применен как к целочисленным аргументам, так и к дробным.

Работа с целочисленными аргументами подчиняется простым правилам. Если делится значение a на значение b, то выражение (a/b)*b+(a%b) должно в точности равняться a. Здесь, конечно, оператор деления целых чисел / всегда возвращает целое число. Например:

```
9/5 возвращает 1
9/(-5)   возвращает -1
(-9)/5 возвращает -1
(-9)/(-5)   возвращает 1
```

Остаток может быть положительным, только если делимое было положительным. Соответственно, остаток может быть отрицательным только в случае отрицательного делимого.

```
9%5 возвращает 4
9%(-5)   возвращает 4
(-9)%5 возвращает -4
(-9)%(-5)   возвращает -4
```

Попытка получить остаток от деления на 0 приводит к ошибке.

Деление с остатком для дробных чисел может быть произведено по двум различным алгоритмам. Один из них повторяет правила для целых чисел, и именно он представлен оператором %. Если в рассмотренном примере деления 9 на 5 перейти к дробным числам. значение остатка во всех вариантах не изменится (оно будет также дробным, конечно).

```
9.0%5.0 возвращает 4.0
9.0%(-5.0) возвращает 4.0
(-9.0)%5.0 возвращает -4.0
(-9.0)%(-5.0) возвращает -4.0
```

Унарные операторы инкрементации ++ и декрементации --, как обычно, можно использовать как справа, так и слева.

```
int x=1;
int y=++x;
```

В этом примере оператор ++ стоит перед переменной `x`, что означает, что сначала произойдет инкрементация, а затем значение `x` будет использовано для инициализации `y`. В результате после выполнения этих строк значения `x` и `y` будут равны 2.

```
int x=1;
int y=x++;
```

А в этом примере сначала значение `x` будет использовано для инициализации `y`, а лишь затем произойдет инкрементация. В результате значение `x` будет равно 2. а `y` будет равно 1.

1.4.3. Логические операторы

Логические операторы "и" и "или" (& и |) можно использовать в двух вариантах. Это связано с тем, что, как легко убедиться, для каждого оператора возможны случаи, когда значение первого операнда сразу определяет значение всего логического выражения. Если вторым операндом является значение некоторой функции, то появляется выбор - вызывать ее или нет, причем это решение может сказаться как на скорости, так и на функциональности программы.

Первый вариант операторов (&, |) всегда вычисляет оба операнда, второй же - (&&, ||) не будет продолжать вычисления, если значение выражения уже очевидно. Например:

```
int x=1;
(x>0) | calculate(x) // в таком выражении произойдет вызов
                        calculate
(x>0) || calculate(x) // а в этом - нет
```

Логический оператор отрицания "не" записывается как !, и конечно имеет только один вариант использования. Этот оператор меняет булевское значение на противоположное.

```
int x=1;
x>0 // выражение истинно
!(x>0)    // выражение ложно
```

Оператор с условием ? : состоит из трех частей - условия и двух выражений. Сначала вычисляется условие (булевское выражение), и на основании результата значение всего оператора определяется первым выражением в случае получения истины, и вторым - если условие ложно. Например, так можно вычислить модуль числа x:

```
x>0  ? x  :  -x
```

Лекция 9

Лексика языка

Лекция посвящена описанию лексики языка Java.

Лексика описывает, из чего состоит текст программы, каким образом он записывается, и на какие простейшие слова (лексемы) компилятор разбивает программу при анализе. Лексемы (или tokens в английском варианте) - это основные "кирпичики", из которых строится любая программа на языке Java.

1. Лексика языка

Лексика описывает, из чего состоит текст программы, каким образом он записывается, и на какие простейшие слова (лексемы) компилятор разбивает программу при анализе. **Лексемы** (или tokens в английском варианте) - это основные "кирпичики", из которых строится любая программа на языке Java.

1.1. Кодировка

Технология Java, как платформа, изначально спроектированная для Глобальной сети Интернет, должна быть многоязыковой, а значит обычный набор символов ASCII (American Standard Code for Information Interchange. Американский стандартный код обмена информацией), включающий в себя лишь латинский алфавит, цифры и простейшие специальные знаки (скобки, знаки препинания, арифметические операции и др.) не достаточен. Поэтому для записи текста программы применяется более универсальная кодировка Unicode.

Как известно, Unicode представляет символы кодом из 2 байт, описывая, таким образом, 65.535 символов. Это позволяет поддерживать практически все распространенные языки мира. Первые 128 символов совпадают с набором ASCII. Однако, понятно, что требуется некоторое специальное обозначение, чтобы иметь возможность задавать в программе любой символ Unicode, ведь никакая клавиатура не позволяет вводить более 65 тысяч различных знаков. Эта конструкция представляет символ Unicode, используя только символы ASCII. Например, если в программу нужно вставить знак с кодом 6917, необходимо его представить в шестнадцатеричном формате (1B05) и записать: \u1B05, причем буква u должна быть прописной, а шестнадцатеричные цифры A, B, C, D, E, F можно использовать произвольно, как заглавные, так и строчные. Таким образом можно закодировать все символы Unicode от \u0000 до \uFFFF. Буквы русского алфавита начинаются с \u0410 (только буква Ё имеет код \u0401) по \u044F (код буквы ё \u0451). В последних версиях JDK в состав демонстрационных приложений и апплетов входит небольшая программа SymbolTest, позволяющая просматривать весь набор символов Unicode. Ее аналог несложно написать самостоятельно. Для перекодирования больших текстов служит утилита native2ascii, также входящая в JDK. Она может работать как в прямом режиме - переводить из разнообразных кодировок в Unicode, записанный ASCII-символами. так и в обратном (опция -reverse) - из Unicode в стандартную кодировку операционной системы.

1.2. Анализ программы

Компилятор, анализируя программу, сразу разделяет ее на

- пробелы (white spaces);
- комментарии (comments);
- основные лексемы (tokens).

1.2.1. Пробелы

Пробелами в данном случае называют все символы, разбивающие текст программы на лексемы. Это как сам символ пробела (space, \u0020, десятичный код 32), так и знаки табуляции и перевода строки. Они используются для разделения лексем, а также для оформления кода. чтобы его было легче читать. Например, следующую часть программы (вычисление корней квадратного уравнения):

```
double a = 1, b = 1, c = 6;
double D = b * b - 4 * a * c;

if (D >= 0) {
double x1 = (-b + Math.sqrt (D)) / (2 * a);
double x2 = (-b - Math.sqrt (D)) / (2 * a);
}
```

можно записать и в таком виде:

```
double a=1,b=1,c=6;double D=b*b-4*a*c;if(D>=0){double
x1=(-b+Math.sqrt(D))/(2*a);
double x2=(-b-Math.sqrt(D))/(2*a);}
```

В обоих случаях компилятор сгенерирует абсолютно одинаковый код. Единственное соображение, которым должен руководствоваться разработчик - легкость чтения и дальнейшей поддержки такого кода.

Для разбиения текста на строки в ASCII используются два символа - "возврат каретки" (carriage return, CR, \u000d, десятичный код 13) и символ новой строки (linefeed, LF, \u000a, десятичный код 10). Чтобы не зависеть от особенностей используемой платформы, в Java применяется наиболее гибкий подход.

Завершением строки считается

- ASCII-символ LF, символ новой строки;
- ASCII-символ CR, "возврат каретки";
- символ CR, за которым сразу же следует символ LF.

Разбиение на строки важно для корректного разбиения на лексемы (как уже говорилось, завершение строки также служит разделителем между лексемами), для правильной работы со строковыми комментариями, а также для вывода отладочной информации (при выводе ошибок компиляции и времени исполнения указывается, на какой строке исходного кода они возникли).

Итак, пробелами в Java считаются:

- ASCII-символ SP, space, пробел, \u0020, десятичный код 32;
- ASCII-символ HT, horizontal tab, символ горизонтальной табуляции, \u0009, десятичный код 9;

- ASCII-символ FF, form feed, символ перевода страницы (был введен для работы с принтером), \u000c, десятичный код 12;
- завершение строки.

1.2.2. Комментарии

Комментарии не влияют на результирующий бинарный код и используются только для ввода пояснений к программе.

В Java комментарии бывают 2 видов:

- строчные;
- блочные.

Строчные комментарии начинаются с ASCII-символов // и длятся до конца текущей строки. Как правило, они используются для пояснения именно этой строки, например:

```
int y=1970; // год рождения
```

Блочные комментарии располагаются между ASCII-символами /* и */, могут занимать произвольное количество строк, например:

```
/*
Этот цикл не может начинаться с нуля
из-за особенностей алгоритма
*/
for (int i=1; i<10; i++) {
…
}
```

Часто блочные комментарии оформляют следующим образом (каждая строка начинается с *):

```
/*
*Описание алгоритма работы
*следующего цикла while
*/
while (x > 0) {
…
}
```

Блочный комментарий не обязательно должен располагаться на нескольких строках, он может даже находится в середине оператора:

```
float s = 2*Math.PI/*getRadius()*/; // Закомментировано для отладки
```

В этом примере блочный комментарий разбивает арифметические операции. Выражение `Math.PI` предоставляет значение константы `PI`, определенное в классе `Math`. Вызов метода `getRadius()` теперь закомментирован и не будет произведен, переменная `s` всегда будет принимать значение `2PI`. Завершает строку строчный комментарий.

Комментарии не могут находиться внутри символьных и строковых литералах. Комментарии не могут быть вложенными.

Любые комментарии полностью удаляются из программы по время компиляции, поэтому их можно использовать неограниченно, не опасаясь. что это повлияет на бинарный код. Основное их предназначение - сделать программу

простой для понимания, в том числе и для других разработчиков, которым придется в ней разбираться по какой-либо причине. Также комментарии зачастую используются для временного исключения частей кода, например

```
int x = 2; int y = 0;
/*
if (x > 0)
y = y + x*2; else y = -y - x*4;
*/
y = y*y;// + 2*x;
```

В этом примере закомментировано выражение if-else и оператор сложения +2*x.

Как уже говорилось выше, комментарии можно писать символами Unicode, то есть на любом языке, удобном разработчику.

Кроме этого, существует особый вид блочного комментария - комментарий разработчика. Он применяется для автоматического создания документации кода. В стандартную поставку JDK, начиная с версии 1.0, входит специальная утилита javadoc. На вход ей подается исходный код классов, а на выходе получается удобная документация в HTML формате, которая описывает все классы, все их поля и методы. При этом активно используются гиперссылки, что существенно упрощает изучение программы по ее такому описанию (например, читая описание метода, можно одним нажатием мыши перейти на описание типов, используемых в качестве аргументов или возвращаемого значения). Однако понятно, что одного названия метода и перечисления его аргументов не достаточно для понимания его работы. Необходимы дополнительные пояснения от разработчика.

Комментарий разработчика записывается так же, как и блочный. Единственное различие в начальной комбинации символов - для документации комментарий необходимо начинать с `/**`. Например:

```
/**
* Вычисление модуля целого числа.
* Этот метод возвращает
* абсолютное значение аргумента x.
*/
int getAbs(int x)   {
if (x>=0)
return x; else return -x;
}
```

Первое предложение должно содержать краткое резюме всего комментария. В дальнейшем оно будет использовано как пояснение этой функции в списке всех методов класса (ниже будут описаны все конструкции языка, для которых применяется комментарий разработчика).

Поскольку в результате создается HTML-документация, то и комментарий необходимо писать по правилам HTML. Допускается применение тегов, таких как <b> и <p> . Однако, теги заголовков с <h1> по <h6> и <hr> использовать нельзя, так как они активно применяются javadoc для создания структуры документации.

Символ * в начале каждой строки и предшествующие ему пробелы и знаки табуляции игнорируются. Их можно вообще не использовать, но они удобны, когда важно форматирование, например, в примерах кода.

```
/**
* Первое предложение - краткое описание метода.
* <p>
* Так оформляется пример кода:
* <blockquote>
* <pre>
* if (condition==true)    {
*    x = getWidht();
*    y = x.getHeight();
* }
* </pre></blockquote>
* А так описывается HTML-список:
* <ul>
* <li>Можно использовать наклонный шрифт <i>курсив</i>,
* <li>или жирный <b>жирный</b>.
* </ul>
*/
public void calculate (int x,   int y)   {
…
}
```

Из этого комментария будет сгенерирован HTML-код. выглядящий примерно так:

Первое предложение - краткое описание метода.

Так оформляется пример кода:

```
if (condition==true) {
     x = getWidht();
     y = x.getHeight();
}
```

А так описывается HTML-список:

- Можно использовать наклонный шрифт курсив,
- или жирный жирный.

Наконец? javadoc поддерживает специальные теги. Они начинаются с символа @. Подробное описание этих тегов можно найти в документации. Например, можно использовать тег @see, чтобы сослаться на другой класс, поле или метод, или даже на другой интернет-сайт.

```
/**
*Краткое описание.
*
```

```
*Развернутый комментарий.
*
*@see java.lang.String
*@see java.lang.Math#PI
*@see <a href="java.sun.com">Official Java site</a>
*/
```

Первая ссылка указывает на класс `String` (`java.lang` - название библиотеки, в которой находится этот класс), вторая - на поле `PI` класса `Math` (символ # разделяет название класса и его полей или методов), третья ссылается на официальный сайт Java.

Комментарии разработчика могут быть записаны перед объявлением классов, интерфейсов, полей, методов и конструкторов. Если записать комментарий `/** ... */` в другой части кода, то ошибки не будет, но он не попадет в документацию, генерируемую javadoc. Кроме этого, можно описать и пакет (так называются библиотеки, или модули, в Java). Для этого необходимо создать специальный файл package.html, сохранить в нем комментарий, и поместить его в директории пакета. HTML-текст, содержащийся между тегами <body> и </body> , будет перемещен в документацию, а первое предложение будет использовано для краткой характеристики этого пакета.

Все классы стандартных библиотек Java поставляются в виде исходного текста, и можно увидеть, как хорошо они комментированы. Стандартная документация по этим классам сгенерирована утилитой javadoc. Для любой программы можно также легко подготовить подобное описание, необходимы лишь грамотные и аккуратные комментарии в исходном коде. Кроме того, Java предоставляет возможность генерировать с помощью javadoc документацию с нестандартным внешним видом.

1.2.3. Лексемы

Итак, были рассмотрены пробелы (в широком смысле этого слова, т.е. все символы, отвечающие за форматирование текста программы) и комментарии, применяемые для ввода пояснений к коду. С точки зрения программиста они применяются для того, чтобы сделать программу более читаемой и понятной для дальнейшего развития.

С точки зрения компилятора, а точнее его части, отвечающей за лексический разбор, основная роль пробелов и комментариев - служить разделителями между лексемами, причем сами разделители далее отбрасываются и не влияют на компилированный код.

Конечно, лексемы очень разнообразны, и именно они определяют многие свойства языка.

1.3. Виды лексем

Ниже перечислены все виды лексем в Java:

- идентификаторы (identifiers);
- ключевые слова (key words);

- литералы (literals);
- разделители (separators);
- операторы (operators).

Рассмотрим их по отдельности.

1.3.1. Идентификаторы

Идентификаторы - это имена, которые даются различным элементам языка для упрощения доступа к ним. Имена имеют пакеты, классы, интерфейсы, поля, методы, аргументы и локальные переменные (все эти понятия подробно рассматриваются в дальнейших главах). Идентификаторы можно записывать символами Unicode, то есть, на любом удобном языке. Длина имени не ограничена.

Идентификатор состоит из букв и цифр. Имя не может начинаться с цифры. Java-буквы, используемые в идентификаторах, включают в себя ASCII-символы A-Z (\u0041-\u005a), a-z (\u0061-\u007a), а также знаки подчеркивания _ (ASCII underscore, \u005f) и доллара $ (\u0024). Знак доллара используется только при автоматической генерации кода (чтобы исключить случайное совпадение имен), либо при использовании каких-либо старых библиотек, в которых допускались имена с этим символом. Java-цифры включают в себя обычные ASCII-цифры 0-9 (\u0030-\u0039).

Для идентификаторов не допускаются совпадения с зарезервированными словами (это ключевые слова, булевские литералы true и false и null-литерал null). Конечно, если 2 идентификатора включают в себя разные буквы, которые одинаково выглядят (например, латинская и русская буквы A), то они считаются различными.

1.3.2. Ключевые слова

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

Ключевые слова `goto` и `const` зарезервированы, но не используются. Это сделано для того, чтобы компилятор мог правильно отреагировать на использование этих распространенных в других языках слов. Напротив, оба булевских литерала true, false и null-литерал null часто считают ключевыми словами (возможно потому. что многие средства разработки подсвечивают их таким же образом), однако это именно литералы.

1.3.3. Литералы

Литералы позволяют задать в программе значения для числовых, символьных и строковых выражений, а также null-литералов. Всего в Java определены следующие виды литералов:

- целочисленный (integer);

- дробный (floating-point);
- булевский (boolean);
- символьный (character);
- строковый (string);
- null-литерал (null-literal).

1.3.4. Разделители

```
( )    [ ]   { } ;   . ,
```

1.3.5. Операторы

Операторы используются в различных операциях – арифметических, логических, битовых, операции сравнения, присваивания. Следующие 37 лексем (все состоят только из ASCII-символов) являются операторами языка Java:

```
=    >    <    !    ~    ?    :
==   <=   >=   !=   &&   ||   ++   --
+    -*   /    &    |    ^    %    <<   >>   >>>
+=   -=   *=   /=   &=   |=   ^=   %=   <<=   >>=   >>>=
```

Большинство из них вполне очевидны и хорошо известны из других языков программирования, однако некоторые нюансы в работе с операторами в Java все же присутствуют.

1.3.6. Заключение

В этой главе были рассмотрены все типы лексем, из которых состоит любая Java-программа. Еще раз напомним, что использование Unicode возможно и необходимо в следующих конструкциях:

- комментарии;
- идентификаторы;
- символьные и строковые литералы.

Остальные же (пробелы, ключевые слова, числовые, булевские и null-литералы, разделители и операторы) легко записываются с применением лишь ASCII-символов. В то же время любой Unicode-символ можно задать в виде специальной последовательности лишь ASCII-символов (условное обозначение - \uhhhh, где hhhh - код символа в шестнадцатеричном формате).

В заключение для примера приведем простейшую программу (традиционное Hello, world!), а затем классифицируем и подсчитаем используемые лексемы:

```
public class Demo {
/**
*Основной метод, с которого начинается выполнение
*любой Java программы.
*/
public static void main (String args[])    {
System.out.println("Hello, world!");
}
}
```

Итак, в приведенной программе есть один комментарий разработчика, 7 идентификаторов, 5 ключевых слов, 1 строковый литерал, 13 разделителей и ни одного оператора.

1.4. Работа с операторами

Рассмотрим некоторые детали использования операторов в Java.

1.4.1. Операторы присваивания и сравнения

Во-первых, конечно же, различаются оператор присваивания = и оператор сравнения ==.

```
x = 1;    // присваиваем переменной x значение 1
x == 1 //  сравниваем значение переменной x с единицей
```

Оператор присваивания не имеет никакого возвращаемого значения. Оператор сравнения всегда возвращает булевское значение true или false. Поэтому обычная опечатка в языке C, когда эти операторы спутывают:

```
// пример вызовет ошибку компилятора
if (x=0) { //здесь должен применятся оператор сравнения ==
…
}
```

в Java легко устраняется. Поскольку выражение `x=0` не имеет никакого значения (и тем более не воспринимается как всегда истинное), то компилятор сообщает об ошибке (необходимо писать `x==0`).

Условие "не равно" записывается как !=. Например:

```
if    (x!=0)    {
float f = 1./x;
}
```

Сочетание какого-либо оператора с оператором присваивания = используется при изменении значения переменной, например, следующие две строки эквивалентны:

```
x = x + 1;
x += 1;
```

1.4.2. Арифметические операции

Наряду с 4 обычными арифметическими операциями +. -. *. /. есть оператор получения остатка от деления %. который может быть применен как к целочисленным аргументам, так и к дробным.

Работа с целочисленными аргументами подчиняется простым правилам. Если делится значение a на значение b, то выражение (a/b)*b+(a%b) должно в точности равняться a. Здесь, конечно, оператор деления целых чисел / всегда возвращает целое число. Например:

```
9/5 возвращает 1
9/(-5)   возвращает -1
(-9)/5 возвращает -1
(-9)/(-5)   возвращает 1
```

Остаток может быть положительным, только если делимое было положительным. Соответственно, остаток может быть отрицательным только в случае отрицательного делимого.

```
9%5 возвращает 4
9%(-5)   возвращает 4
(-9)%5 возвращает -4
(-9)%(-5)   возвращает -4
```

Попытка получить остаток от деления на 0 приводит к ошибке.

Деление с остатком для дробных чисел может быть произведено по двум различным алгоритмам. Один из них повторяет правила для целых чисел, и именно он представлен оператором %. Если в рассмотренном примере деления 9 на 5 перейти к дробным числам. значение остатка во всех вариантах не изменится (оно будет также дробным, конечно).

```
9.0%5.0 возвращает 4.0
9.0%(-5.0) возвращает 4.0
(-9.0)%5.0 возвращает -4.0
(-9.0)%(-5.0) возвращает -4.0
```

Унарные операторы инкрементации ++ и декрементации --, как обычно, можно использовать как справа, так и слева.

```
int x=1;
int y=++x;
```

В этом примере оператор ++ стоит перед переменной `x`, что означает, что сначала произойдет инкрементация, а затем значение `x` будет использовано для инициализации `y`. В результате после выполнения этих строк значения `x` и `y` будут равны 2.

```
int x=1;
int y=x++;
```

А в этом примере сначала значение `x` будет использовано для инициализации `y`, а лишь затем произойдет инкрементация. В результате значение `x` будет равно 2. а `y` будет равно 1.

1.4.3. Логические операторы

Логические операторы "и" и "или" (& и |) можно использовать в двух вариантах. Это связано с тем, что, как легко убедиться, для каждого оператора возможны случаи, когда значение первого операнда сразу определяет значение всего логического выражения. Если вторым операндом является значение некоторой функции, то появляется выбор - вызывать ее или нет, причем это решение может сказаться как на скорости, так и на функциональности программы.

Первый вариант операторов (&, |) всегда вычисляет оба операнда, второй же - (&&, ||) не будет продолжать вычисления, если значение выражения уже очевидно. Например:

```
int x=1;
(x>0) | calculate(x) // в таком выражении произойдет вызов
                        calculate
(x>0) || calculate(x) // а в этом - нет
```

Логический оператор отрицания "не" записывается как !, и конечно имеет только один вариант использования. Этот оператор меняет булевское значение на противоположное.

```
int x=1;
x>0 // выражение истинно
!(x>0)    // выражение ложно
```

Оператор с условием ? : состоит из трех частей - условия и двух выражений. Сначала вычисляется условие (булевское выражение), и на основании результата значение всего оператора определяется первым выражением в случае получения истины, и вторым - если условие ложно. Например, так можно вычислить модуль числа x:

```
x>0  ? x  :  -x
```