Лекция 2. Лексика языка

Кодировка

Технология *Java* как платформа, изначально спроектированная для Глобальной сети *Internet*, 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 т.д.), недостаточен. Поэтому для записи текста программы применяется более универсальная *кодировка Unicode UTF-16*.

Как известно, ***Unicode UTF-16*** представляет символы кодом из 2 байт, описывая, таким образом, 65535 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 строчной, 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`).

Итак, используя простейшую кодировку *ASCII*, можно ввести произвольную последовательность символов *Unicode*. Далее будет показано, что *Unicode* используется не для всех лексем, а только для тех, для которых важна *поддержка* многих языков, а именно: комментарии, *идентификаторы*, символьные и строковые *литералы*. Для записи остальных лексем вполне достаточно *ASCII* -символов.

Анализ программы

Компилятор, анализируя программу, сразу разделяет ее на:

- *пробелы* (white spaces);
- комментарии (comments);

- основные лексемы (tokens).

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 при дальнейшей поддержке такого кода.

Для разбиения текста на строки в *ASCII* используется 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 (см. следующую тему "Комментарии"), 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.

Комментарии

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

В Java комментарии бывают двух 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 находиться 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 `2 PI`. Завершает строку строчный комментарий.

Комментарии не могут находиться в символьных и строковых *литералах*, идентификаторах (эти понятия подробно рассматриваются далее в этой лекции). Следующий пример содержит случаи неправильного применения комментариев:

```
// В этом примере текст /*…*/ станет просто
// частью строки s
String s = "text/*just text*/";
/*
   Следующая строка станет причиной ошибки
   при компиляции, так как комментарий разбил
   имя метода getRadius()
*/
circle.get/*comment*/Radius();
```

А такой код допустим:

```
// Комментарий может разделять вызовы функций:
circle./*comment*/getRadius();

// Комментарий может заменять пробелы:
int/*comment*/x=1;
```

В последней строке между названием типа данных `int` и названием переменной `x` обязательно должен быть пробел или, как в данном примере, комментарий.

Комментарии не могут быть вложенными. Символы `/*`, `*/`, `//` не имеют никакого особенного значения внутри уже открытых комментариев, как строчных, так и блочных. Таким образом, в примере

```
/* начало комментария /* // /** завершение: */
```

описан только один блочный комментарий. А в следующем примере (строки кода пронумерованы для удобства)

```
1. /*
2.    comment
3.    /*
4.    more comments
5.    */
6.    finish
7. */
```

компилятор выдаст ошибку. Блочный комментарий начался в строке 1 с комбинации символов `/*`. Вторая открывающая комбинация `/*` на строке 3 будет проигнорирована, так как находится уже внутри комментария. Символы `*/` в строке 5 завершат его, а строка 7 породит ошибку – попытка закрыть комментарий, который не был начат.

Любые комментарии полностью удаляются из программы во 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 (например, читая описание метода, можно с помощью одного нажатия мыши перейти на описание типов,

используемых в качестве аргументов или возвращаемого значения). Однако понятно, что одного названия метода и перечисления его аргументов недостаточно 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 не использовать вообще, но они удобны, когда необходимо форматирование, скажем, в примерах кода.

```
/**
   * Первое предложение - краткое
   * описание метода.
   * <p>
   * Так оформляется пример кода:
   * <blockquote>
   * <pre>
   * if (condition==true) {
   *    x = getWidth();
   *    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 = getWidth();
   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 Internet-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 того, можно описать пакет (так называются библиотеки, или модули, в Java). Для этого необходимо создать специальный файл `package.html`, сохранить в нем комментарий и поместить его в каталог пакета. HTML-текст, содержащийся между тегами `<body>` и `</body>`, будет помещен в документацию, а первое предложение будет использоваться для краткой характеристики этого пакета.

Лексемы

Итак, мы рассмотрели *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 на компилированный код не влияют. Например, все следующие примеры объявления переменной эквивалентны:

```
// Используем пробел в качестве разделителя.
int x = 3 ;

// здесь разделителем является перевод строки
int
x
=
3
;

// здесь разделяем знаком табуляции
int x = 3 ;

/*
   * Единственный принципиально необходимый
   * разделитель между названием типа данных
   * int и именем переменной x здесь описан
   * комментарием блочного типа.
   */
```

```
int/**/x=3;
```

Конечно, лексемы очень разнообразны, и именно они определяют многие свойства языка. Рассмотрим все их виды более подробно.

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.

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 следующих лекциях). *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.

В этой лекции уже применялись следующие *идентификаторы*:

```
Character, a, b, c, D, x1, x2, Math, sqrt, x,
y, i, s, PI, getRadius, circle, getAbs,
calculate, condition, getWidth, getHeight,
```

```
java, lang, String
```

Также допустимыми являются *идентификаторы*:

```
Computer, COLOR_RED, _, aVeryLongNameOfTheMethod
```

Ключевые слова

Ключевые слова – это зарезервированные слова, состоящие из *ASCII* - символов и выполняющие различные задачи языка. Вот их полный список (48 слов):

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

Ключевые слова `goto` и `const` зарезервированы, но не используются. Это сделано для того, чтобы компилятор мог правильно отреагировать на их использование в других языках. 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*.

Значение всех ключевых слов будет рассматриваться в следующих лекциях.

Литералы

Литералы позволяют задать в программе значения для числовых, символьных и строковых выражений, а также null- *литералов* . Всего в Java определено 6 видов *литералов*:

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

Рассмотрим их по отдельности.

Целочисленные литералы

Целочисленные *литералы* позволяют задавать целочисленные значения в десятичном, восьмеричном и шестнадцатеричном виде. Десятичный формат традиционен и ничем не отличается от правил, принятых в других языках. Значения в восьмеричном виде начинаются с нуля, и, конечно, использование цифр `8` и `9` запрещено. Запись шестнадцатеричных чисел начинается с `0x` или `0X` (цифра 0 и латинская *ASCII* -буква `X` в произвольном регистре). Таким образом, ноль можно записать тремя различными способами:

```
0
0X0
0x0
```

Как обычно, для записи цифр `10` - `15` в шестнадцатеричном формате используются буквы `A`, `B`, `C`, `D`, `E`, `F`, прописные или строчные. Примеры таких *литералов*:

```
0xaBcDeF, 0xCafe, 0xDEC
```

Типы данных рассматриваются ниже, однако здесь необходимо упомянуть два целочисленных типа `int` и `long` длиной 4 и 8 байт, соответственно (или 32 и 64 бита, соответственно). Оба эти типа знаковые, т.е. тип `int` хранит значения от -2^{31} до $2^{31}-1$, или от `-2.147.483.648` до `2.147.483.647`. По умолчанию целочисленный *литерал* имеет тип `int`, а значит, в программе допустимо использовать *литералы* только от `0` до `2147483648`, иначе возникнет ошибка компиляции. При этом *литерал* `2147483648` можно использовать только как аргумент *унарного оператора* - :

```
int x = -2147483648;  \\ верно
int y = -5-2147483648; \\ здесь возникнет
                       \\ ошибка компиляции
```

Соответственно, допустимые *литералы* в восьмеричной записи должны быть от `00` до `017777777777` ($=2^{31}-1$), с унарным оператором - допустимо также `-020000000000` ($= -2^{31}$). Аналогично для шестнадцатеричного формата – от `0x0` до `0x7fffffff` ($=2^{31}-1$), а также `-0x80000000` ($= -2^{31}$).

Тип `long` имеет длину 64 бита, а значит, позволяет хранить значения от -2^{63} до $2^{63}-1$. Чтобы ввести такой *литерал*, необходимо в конце поставить латинскую букву `L` или `l`, тогда все значение будет трактоваться как `long`. Аналогично можно выписать максимальные допустимые значения для них:

```
9223372036854775807L
```

```
0777777777777777777L
0x7fffffffffffffL
// наименьшие отрицательные значения:
-9223372036854775808L
-01000000000000000000000L
-0x8000000000000000L
```

Другие примеры целочисленных *литералов* типа `long`:

```
0L, 123l, 0xC0B0L
```

Дробные литералы

Дробные *литералы* представляют собой числа с плавающей десятичной точкой. Правила записи таких чисел такие же, как и в большинстве современных языков программирования.

Примеры:

```
3.14
2.
.5
7e10
3.1E-20
```

Таким образом, дробный *литерал* состоит из следующих составных частей:

- целая часть;
- десятичная точка (используется *ASCII* -символ точка);
- дробная часть;
- порядок (состоит из латинской *ASCII* -буквы `E` в произвольном регистре и целого числа с опциональным знаком `+` или `-`);
- окончание-указатель типа.

Целая и дробная части записываются десятичными цифрами, а указатель типа (аналог указателя `L` или `l` для целочисленных *литералов* типа `long`) имеет два возможных значения – латинская *ASCII* -буква `D`(для типа `double`) или `F` (для типа `float`) в произвольном регистре. Они будут подробно рассмотрены ниже.

Необходимыми частями являются:

- хотя бы одна цифра в целой или дробной части;
- десятичная точка или показатель степени, или указатель типа.

Все остальные части необязательные. Таким образом, "минимальные" дробные *литералы* могут быть записаны, например, так:

```
1.
.1
1e1
1f
```

В Java есть два дробных типа, упомянутые выше, – `float` и `double`. Их длина – 4 и 8 байт или 32 и 64 бита, соответственно. Дробный *литерал* имеет тип `float`, если он заканчивается на латинскую букву `F`в произвольном регистре. В противном случае он рассматривается как значение типа `double` и может включать в себя окончание `D` или `d`, как *признак типа* `double` (используется только для наглядности).

```
// float-литералы:
1f, 3.14F, 0f, 1e+5F
// double-литералы:
0., 3.14d, 1e-4, 31.34E45D
```

В Java дробные числа 32-битного типа `float` и 64-битного типа `double` хранятся в памяти в бинарном виде в формате, стандартизированном спецификацией *IEEE 754* (полное название – IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985 (IEEE, New York)). В этой спецификации описаны не только конечные дробные величины, но и еще несколько особых значений, а именно:

- положительная и отрицательная бесконечности (positive/negative infinity);
- значение "не число", Not-a-Number, сокращенно NaN;
- положительный и отрицательный нули.

Для этих значений нет специальных обозначений. Чтобы получить такие величины, необходимо либо произвести арифметическую операцию (например, результатом деления ноль на ноль `0.0/0.0` является `NaN`), либо обратиться к константам в классах `Float` и `Double`, а именно `POSITIVE_INFINITY`, `NEGATIVE_INFINITY` и `NaN`. Более подробно работа с этими особенными значениями рассматривается в дальнейшем.

Типы данных накладывают ограничения на возможные значения *литералов*, как и для целочисленных типов. Максимальное положительное конечное значение дробного *литерала*:

- для `float`: 3.40282347e+38f
- для `double`: 1.79769313486231570e+308

Кроме того, для дробных величин становится важным еще одно предельное значение – минимальное положительное ненулевое значение:

- для `float`: 1.40239846e-45f
- для `double`: 4.94065645841246544e-324

Попытка указать *литерал* со слишком большим абсолютным значением (например, `1e40F`) приведет к ошибке компиляции. Такая величина должна представляться бесконечностью. Аналогично, указание *литерала* со слишком малым ненулевым значением (например, `1e-350`) также приводит к ошибке. Это значение должно быть округлено до нуля. Однако если округление приводит не к нулю, то компилятор произведет его сам:

```
// ошибка, выражение должно быть округлено до 0
0.00000000000000000000000000000000000000000001f
// ошибки нет, компилятор сам округляет до 1
1.00000000000000000000000000000000000000000001f
```

Стандартных возможностей вводить дробные значения не в десятичной системе в Java нет, однако классы `Float` и `Double` предоставляют много вспомогательных методов, в том числе и для такой задачи.

Логические литералы

Логические *литералы* имеют два возможных значения – `true` и `false`. Эти два зарезервированных слова не являются ключевыми, но также не могут использоваться в качестве идентификатора.

Символьные литералы

Символьные *литералы* описывают один символ из набора *Unicode*, заключенный в одиночные кавычки, или апострофы (*ASCII* -символ *single quote*, `\u0027`). Например:

```
'a' // латинская буква a
' ' // пробел
'K' // греческая буква каппа
```

Также допускается специальная запись для описания символа через его код (см. тему "Кодировка"). Примеры:

```
'\u0041' // латинская буква A
'\u0410' // русская буква A
'\u0391' // греческая буква A
```

Символьный *литерал* должен содержать строго один символ, или специальную последовательность, начинающуюся с `\`. Для записи

специальных символов (неотображаемых и служебных, таких как ", ', \ ) используются следующие обозначения:

```
\b  \u0008  backspace BS - забой
\t  \u0009  horizontal tab HT - табуляция
\n  \u000a  linefeed LF - конец строки
\f  \u000c  form feed FF - конец страницы
\r  \u000d  carriage return CR -
            возврат каретки
\"  \u0022  double quote " - двойная кавычка
\'  \u0027  single quote ' - одинарная кавычка
\\  \u005c  backslash \ - обратная косая черта
\шестнадцатеричный код  от \u0000 до \u00ff символа
                  в шестнадцатеричном формате.
```

Первая колонка описывает стандартные обозначения специальных символов, используемые в Java-программах. Вторая колонка представляет их в стандартном виде *Unicode* -символов. Третья колонка содержит английские и русские описания. Использование `\` в комбинации с другими символами приведет к ошибке компиляции.

Поддержка ввода символов через восьмеричный код обеспечивается для совместимости с C. Например:

```
'\101' // Эквивалентно '\u0041'
```

Однако таким образом можно задать лишь символы от `\u0000` до `\u00ff` (т.е. с кодом от 0 до 255), поэтому *Unicode* -последовательности предпочтительней.

Поскольку обработка *Unicode* -последовательностей (`\uhhhh`) производится раньше лексического анализа, то следующий пример является ошибкой:

```
'\u000a' // символ конца строки
```

Компилятор сначала преобразует `\u000a` в символ конца строки и кавычки окажутся на разных строках кода, что является ошибкой. Необходимо использовать специальную последовательность:

```
'\n' // правильное обозначение конца строки
```

Аналогично и для символа `\u000d` (возврат каретки) необходимо использовать обозначение `\r`.

Специальные символы можно использовать в составе как символьных, так и строковых *литералов*.

Строковые литералы

Строковые *литералы* состоят из набора символов и записываются в двойных кавычках. Длина может быть нулевой или сколь угодно большой. Любой символ может быть представлен специальной последовательностью, начинающейся с `\` (см. "Символьные *литералы* ").

```
"" // литерал нулевой длины
"\"" //литерал, состоящий из одного символа "
"Простой текст" //литерал длины 13
```

Строковый *литерал* нельзя разбивать на несколько строк в коде программы. Если требуется текстовое значение, состоящее из нескольких строк, то необходимо воспользоваться специальными символами `\n` и/или `\r`. Если же текст просто слишком длинный, чтобы уместиться на одной строке кода, можно использовать оператор конкатенации строк `+`. Примеры строковых *литералов*:

```
// выражение-константа, составленное из двух
// литералов
"Длинный текст " +
"с переносом"
/*
   * Строковый литерал, содержащий текст
   * из двух строк:
   * Hello, world!
   * Hello!
   */
"Hello, world!\r\nHello!"
```

На строковые *литералы* распространяются те же правила, что и на символьные в отношении использования символов новой строки `\u000a` и `\u000d`.

Каждый строковый *литерал* является экземпляром класса `String`. Это определяет некоторые необычные свойства строковых *литералов*, которые будут рассмотрены в следующей лекции.

Null-литерал

Null- *литерал* может принимать всего одно значение: `null`. Это *литерал* ссылочного типа, причем эта ссылка никуда не ссылается, объект отсутствует. Разумеется, его можно применять к ссылкам любого *объектного типа данных*. Типы данных подробно рассматриваются в следующей лекции.

Разделители

Разделители – это специальные символы, которые используются в служебных целях языка. Назначение каждого из них будет рассмотрено по ходу изложения курса. Вот их полный список:

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

Операторы

Операторы используются в различных операциях – арифметических, логических, битовых, операциях сравнения и 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, поэтому в конце лекции приводятся краткие комментарии к ним.

Пример программы

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. Этот текст можно сохранить в файле Demo.java, скомпилировать и запустить. Результатом работы будет, как очевидно:

```
Hello, world!
```

Дополнение. Работа с операторами

Рассмотрим некоторые детали использования операторов в *Java*. Здесь будут описаны подробности, относящиеся к работе самих операторов. В следующей лекции детально рассматриваются особенности, возникающие при использовании различных типов данных (например, *значение операции* `1/2` равно `0`, а `1/2.` равно `0.5`).

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 сравнения всегда возвращает булевское значение true или false. Оператор присваивания возвращает значение правого операнда. Поэтому обычная опечатка в языке C, когда эти операторы путают:

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

в Java легко устраняется. Поскольку выражение `x=0` имеет числовое значение 0, а не булевское (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 `=` (см. нижнюю строку в полном перечне в разделе "Операторы") используется при изменении значения переменной. Например, следующие две строки эквивалентны:

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

Арифметические операции

Наряду с четырьмя обычными арифметическими операциями `+`, `-`, `*`, `/`, существует 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
```

Однако стандарт *IEEE 754* определяет другие правила. Такой способ представлен методом стандартного класса `Math.IEEEremainder(double f1, double f2)`. Результат этого метода – значение, которое равно`f1-f2*n`, где `n` – целое число, ближайшее к значению `f1/f2`, а если два целых числа одинаково близки к этому отношению, то выбирается четное. По этому правилу значение остатка будет другим:

```
Math.IEEEremainder(9.0, 5.0) возвращает -1.0
Math.IEEEremainder(9.0, -5.0) возвращает -1.0
Math.IEEEremainder(-9.0, 5.0) возвращает 1.0
Math.IEEEremainder(-9.0, -5.0) возвращает 1.0
```

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`, это 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`, и лишь затем произойдет инкрементация. В результате значение `x` будет равно `2`, а `y` будет равно `1`.

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), а 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
```

Битовые операции

Прежде чем переходить к *битовым операциям*, необходимо уточнить, каким именно образом целые числа представляются в двоичном виде. Конечно, для неотрицательных величин это практически очевидно:

```
0   0
1   1
2   10
3   11
4   100
5   101
```

и так далее. Однако как представляются отрицательные числа? Во-первых, вводят понятие *знакового бита*. Первый бит начинает отвечать за знак, а именно `0` означает положительное число, `1` – отрицательное. Но не следует думать, что остальные биты остаются неизменными. Например, если рассмотреть 8-битовое представление:

```
-1   10000001   // это НЕВЕРНО!
-2   10000010   // это НЕВЕРНО!
-3   10000011   // это НЕВЕРНО!
```

Такой подход неверен! В частности, мы получаем сразу два представления нуля – `00000000` и `10000000`, что нерационально. Правильный алгоритм можно представить себе так. Чтобы получить значение `-1`, надо из `0` вычесть `1`:

```
00000000
- 00000001
------------
- 11111111
```

Итак, `-1` в двоичном виде представляется как `11111111`. Продолжаем применять тот же алгоритм (вычитаем 1):

```
0 00000000
-1 11111111
-2 11111110
-3 11111101
```

и так далее до значения `10000000`, которое представляет собой наибольшее по модулю отрицательное число. Для 8-битового представления наибольшее

положительное число `01111111 (=127)`, а наименьшее отрицательное `10000000 (=-128)`. Поскольку всего 8 бит определяет $2^8=256$ значений, причем одно из них отводится для нуля, то становится ясно, почему наибольшие по модулю положительные и отрицательные значения различаются на единицу, а не совпадают.

Как известно, *битовые операции* "и", "или", "исключающее или" принимают два аргумента и выполняют логическое действие попарно над соответствующими битами аргументов. При этом используются те же обозначения, что и для логических операторов, но, конечно, только в первом (одиночном) варианте. Например, вычислим выражение `5&6`:

```
00000101
& 00000110
-------------
  00000100

   // число 5 в двоичном виде
   // число 6 в двоичном виде

   //проделали операцию "и" попарно над битами
   // в каждой позиции
```

То есть выражение 5&6 равно 4.

Исключение составляет лишь оператор "не" или "NOT", который для побитовых операций записывается как `~` (для логических было !). Этот оператор меняет каждый бит в числе на противоположный. Например, `~(-1)=0`. Можно легко установить общее правило для получения битового представления отрицательных чисел:

Если `n` – целое положительное число, то `-n` в битовом представлении равняется `~(n-1)`.

Наконец, осталось рассмотреть лишь операторы побитового сдвига. В Java есть один оператор сдвига влево и два варианта сдвига вправо. Такое различие связано с наличием *знакового бита*.

При сдвиге влево оператором `<<` все биты числа смещаются на указанное количество позиций влево, причем освободившиеся справа позиции заполняются нулями. Эта операция аналогична умножению на 2^n и действует вполне предсказуемо, как при положительных, так и при отрицательных аргументах.

Рассмотрим примеры применения операторов сдвига для значений типа `int`, т.е. 32-битных чисел. Пусть положительным аргументом будет число `20`, а отрицательным `-21`.

```
// Сдвиг влево для положительного числа 20
20 << 00 = 00000000000000000000000000010100 = 20
20 << 01 = 00000000000000000000000000101000 = 40
20 << 02 = 00000000000000000000000001010000 = 80
20 << 03 = 00000000000000000000000010100000 = 160
20 << 04 = 00000000000000000000000101000000 = 320
...
20 << 25 = 00101000000000000000000000000000 = 671088640
20 << 26 = 01010000000000000000000000000000 = 1342177280
20 << 27 = 10100000000000000000000000000000 = -1610612736
20 << 28 = 01000000000000000000000000000000 = 1073741824
20 << 29 = 10000000000000000000000000000000 = -2147483648
20 << 30 = 00000000000000000000000000000000 = 0
20 << 31 = 00000000000000000000000000000000 = 0
// Сдвиг влево для отрицательного числа -21
-21 << 00 = 11111111111111111111111111101011 = -21
-21 << 01 = 11111111111111111111111111010110 = -42
-21 << 02 = 11111111111111111111111110101100 = -84
-21 << 03 = 11111111111111111111111101011000 = -168
-21 << 04 = 11111111111111111111111010110000 = -336
-21 << 05 = 11111111111111111111110101100000 = -672
...
-21 << 25 = 11010110000000000000000000000000 = -704643072
-21 << 26 = 10101100000000000000000000000000 = -1409286144
-21 << 27 = 01011000000000000000000000000000 = 1476395008
-21 << 28 = 10110000000000000000000000000000 = -1342177280
-21 << 29 = 01100000000000000000000000000000 = 1610612736
-21 << 30 = 11000000000000000000000000000000 = -1073741824
-21 << 31 = 10000000000000000000000000000000 = -2147483648
```

Как видно из примера, неожиданности возникают тогда, когда значащие биты начинают занимать первую позицию и влиять на знак результата.

При сдвиге вправо все биты аргумента смещаются на указанное количество позиций, соответственно, вправо. Однако встает вопрос – каким значением заполнять освобождающиеся позиции слева, в том числе и отвечающую за знак. Есть два варианта. Оператор `>>` использует для заполнения этих позиций значение *знакового бита*, то есть результат всегда имеет тот же знак, что и начальное значение. Второй оператор`>>>` заполняет их нулями, то есть результат всегда положительный.

```
// Сдвиг вправо для положительного числа 20
// Оператор >>
```

```
20 >> 00 = 00000000000000000000000000010100 = 20
20 >> 01 = 00000000000000000000000000001010 = 10
20 >> 02 = 00000000000000000000000000000101 = 5
20 >> 03 = 00000000000000000000000000000010 = 2
20 >> 04 = 00000000000000000000000000000001 = 1
20 >> 05 = 00000000000000000000000000000000 = 0
// Оператор >>>
20 >>> 00 = 00000000000000000000000000010100 = 20
20 >>> 01 = 00000000000000000000000000001010 = 10
20 >>> 02 = 00000000000000000000000000000101 = 5
20 >>> 03 = 00000000000000000000000000000010 = 2
20 >>> 04 = 00000000000000000000000000000001 = 1
20 >>> 05 = 00000000000000000000000000000000 = 0
```

Очевидно, что для положительного аргумента операторы `>>` и `>>>` работают совершенно одинаково. Дальнейший сдвиг на большее количество позиций будет также давать нулевой результат.

```
// Сдвиг вправо для отрицательного числа -21
// Оператор >>
-21 >> 00 = 11111111111111111111111111101011 = -21
-21 >> 01 = 11111111111111111111111111110101 = -11
-21 >> 02 = 11111111111111111111111111111010 = -6
-21 >> 03 = 11111111111111111111111111111101 = -3
-21 >> 04 = 11111111111111111111111111111110 = -2
-21 >> 05 = 11111111111111111111111111111111 = -1
// Оператор >>>
-21 >>> 00 = 11111111111111111111111111101011 = -21
-21 >>> 01 = 01111111111111111111111111110101 = 2147483637
-21 >>> 02 = 00111111111111111111111111111010 = 1073741818
-21 >>> 03 = 00011111111111111111111111111101 = 536870909
-21 >>> 04 = 00001111111111111111111111111110 = 268435454
-21 >>> 05 = 00000111111111111111111111111111 = 134217727
...
-21 >>> 24 = 00000000000000000000000011111111 = 255
-21 >>> 25 = 00000000000000000000000001111111 = 127
-21 >>> 26 = 00000000000000000000000000111111 = 63
-21 >>> 27 = 00000000000000000000000000011111 = 31
-21 >>> 28 = 00000000000000000000000000001111 = 15
-21 >>> 29 = 00000000000000000000000000000111 = 7
-21 >>> 30 = 00000000000000000000000000000011 = 3
-21 >>> 31 = 00000000000000000000000000000001 = 1
```

Как видно из примеров, эти операции аналогичны делению на 2^n. Причем, если для положительных аргументов с ростом `n` результат закономерно стремится к `0`, то для отрицательных предельным значением является `-1`.

Заключение

В этой лекции были рассмотрены основы лексического анализа программ *Java*. Для их записи применяется универсальная *кодировка Unicode,* позволяющая использовать любой язык помимо традиционного английского. 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 также можно задать в виде специальной последовательности *ASCII* -символов.

Во время анализа *компилятор* выделяет из текста программы < *пробелы* > (были рассмотрены все символы, которые рассматриваются как *пробелы*) и комментарии, которые полностью удаляются из кода (были рассмотрены все виды комментариев, в частности комментарий разработчика). *Пробелы* и все виды комментариев служат для разбиения текста программы на лексемы. Были рассмотрены все виды лексем, в том числе все виды *литералов*.

В дополнении были рассмотрены особенности применения различных операторов.