Лекция 3. Типы данных

Введение

Java является строго типизированным языком. Это означает, что любая *переменная* и любое *выражение* имеют известный тип еще на момент компиляции. Такое строгое правило позволяет выявлять многие ошибки уже во *время компиляции*. *Компилятор*, найдя ошибку, указывает точное *место* (строку) и причину ее возникновения, а динамические "баги" (от английского bugs) необходимо сначала выявить с помощью тестирования (что может потребовать значительных усилий), а затем найти *место* в коде, которое их породило. Поэтому четкое понимание модели типов данных в *Java* очень помогает в написании качественных программ.

Все типы данных разделяются на две группы. Первую составляют 8 *простых*, или *примитивных* (от английского *primitive*), типов данных. Они подразделяются на три подгруппы:

- *целочисленные*
  - byte
  - short
  - int
  - long
  - char (также является целочисленным типом)
- *дробные*
  - float
  - double
- *булевые*
  - boolean

Вторую группу составляют *объектные*, или *ссылочные* (от английского reference), типы данных. Это все классы, интерфейсы и массивы. В стандартных библиотеках первых версий *Java* находилось несколько сот классов и интерфейсов, сейчас их уже тысячи. Кроме стандартных, написаны многие и многие классы и интерфейсы, составляющие любую *Java*-программу.

Иллюстрировать логику работы с типами данных проще всего на примере переменных.

Переменные

Переменные используются в программе для хранения данных.
Любая *переменная* имеет три базовые характеристики:

- имя;
- тип;
- значение.

Имя уникально идентифицирует переменную и позволяет обращаться к ней в программе. Тип описывает, какие величины может
хранить *переменная*. *Значение* – текущая величина, хранящаяся в переменной на данный момент.

Работа с переменной всегда начинается с ее *объявления* (*declaration*). Конечно, оно должно включать в себя имя объявляемой переменной. Как было сказано, в *Java* любая *переменная* имеет строгий тип, который также задается при объявлении и никогда не меняется. *Значение* может быть указано сразу (это называется инициализацией), а в большинстве случаев задание начальной величины можно и отложить.

Некоторые примеры объявления переменных примитивного типа `int` с инициализаторами и без таковых:

```
int a;
int b = 0, c = 3+2;
int d = b+c;
int e = a = 5;
```

Из примеров видно, что инициализатором может быть не только константа, но и арифметическое *выражение*. Иногда это *выражение* может быть вычислено во *время компиляции* (такое как `3+2`), тогда*компилятор* сразу записывает результат. Иногда это действие откладывается на момент выполнения программы (например, `b+c`). В последнем случае нескольким переменным присваивается одно и то же*значение*, однако объявляется лишь первая из них (в данном примере `e`), остальные уже должны существовать.

Резюмируем: ***объявление*** переменных и возможная инициализация при объявлении описываются следующим образом. Сначала указывается тип переменной, затем ее имя и, если необходимо, инициализатор, который может быть константой или выражением, вычисляемым во время компиляции или исполнения программы. В частности, можно пользоваться уже объявленными переменными. Далее можно поставить запятую и объявить новую переменную точно такого же типа.

После объявления *переменная* может применяться в различных выражениях, в которых будет браться ее текущее *значение*. Также в любой момент можно изменить *значение*, используя *оператор присваивания*, примерно так же, как это делалось в инициализаторах.

Кроме того, при объявлении переменной может быть использовано *ключевое слово* `final`. Его указывают перед типом переменной, и тогда ее необходимо сразу инициализировать и уже больше никогда не менять ее *значение*. Таким образом, `final` -переменные становятся чем-то вроде констант, но на самом деле некоторые инициализаторы могут вычисляться только во время исполнения программы, генерируя различные значения.

Простейший пример объявления `final` -переменной:

```
final double pi=3.1415;
```

Примитивные и ссылочные типы данных

Теперь на примере переменных можно проиллюстрировать различие между *примитивными* и *ссылочными* типами данных. Рассмотрим пример, когда объявляются две переменные одного типа, приравниваются друг другу, а затем *значение* одной из них изменяется. Что произойдет со второй переменной?

Возьмем *простой тип* `int`:

```
int a=5;    // объявляем первую переменную и
            // инициализируем ее
int b=a;    // объявляем вторую переменную и
            // приравниваем ее к первой
a=3;        // меняем значение первой
print(b);   // проверяем значение второй
```

Здесь и далее мы считаем, что *функция* `print(...)` позволяет нам некоторым (неважно, каким именно) способом узнать *значение* ее аргумента (как правило, для этого используют функцию из стандартной библиотеки `System.out.println(...)`, которая выводит *значение* на *системную консоль*).

В результате мы увидим, что *значение* переменной `b` не изменилось, оно осталось равным `5`. Это означает, что переменные *простого* типа хранят непосредственно свои значения и при приравнивании двух переменных происходит *копирование* данного значения. Чтобы еще раз подчеркнуть эту особенность, приведем еще один пример:

```
byte b=3;
int a=b;
```

В данном примере происходит *преобразование типов* (оно подробно рассматривается в соответствующей лекции). Для нас сейчас важно констатировать, что *переменная* `b` хранит *значение* `3` типа `byte`, а

переменная `a` – *значение* `3` типа `int`. Это два разных значения, и во второй строке при присваивании произошло *копирование*.

Теперь рассмотрим *ссылочный тип данных*. Переменные таких типов всегда хранят ссылки на некоторые объекты. Рассмотрим для примера *класс*, описывающий точку на координатной плоскости с целочисленными координатами. Описание класса – это отдельная тема, но в нашем простейшем случае оно тривиально:

```
class Point {
   int x, y;
}
```

Теперь составим пример, аналогичный приведенному выше для `int` - переменных, считая, что *выражение* `new Point(3,5)` создает новый *объект*-точку с координатами `(3,5)`.

```
Point p1 = new Point(3,5);
Point p2=p1;
p1.x=7;
print(p2.x);
```

В третьей строке мы изменили горизонтальную координату точки, на которую ссылалась *переменная* `p1`, и теперь нас интересует, как это сказалось на точке, на которую ссылается *переменная* `p2`. Проведя такой эксперимент, можно убедиться, что в этот раз мы увидим обновленное *значение*. То есть объектные переменные после приравнивания остаются "связанными" друг с другом, изменения одной сказываются на другой.

Таким образом, примитивные переменные являются действительными хранилищами данных. Каждая *переменная* имеет *значение*, не зависящее от остальных. Ссылочные же переменные хранят лишь ссылки на объекты, причем различные переменные могут ссылаться на один и тот же *объект*, как это было в нашем примере. В этом случае их можно сравнить с наблюдателями, которые с разных позиций смотрят на один и тот же *объект* и одинаково видят все происходящие с ним изменения. Если же один наблюдатель сменит *объект* наблюдения, то он перестает видеть и изменения, происходящие с прежним объектом:

```
Point p1 = new Point(3,5);
Point p2=p1;
p1 = new Point(7,9);
print(p2.x);
```

В этом примере мы получим `3`, то есть после третьей строки переменные `p1` и `p2` ссылаются на различные объекты и поэтому имеют разные значения.

Теперь легко понять смысл литерала `null`. Такое *значение* может принять *переменная* любого ссылочного типа. Это означает, что ее *ссылка* никуда не указывает, *объект* отсутствует. Соответственно, любая попытка обратиться к объекту через такую переменную (например, вызвать метод или взять *значение* поля) приведет к ошибке.

Также *значение* `null` можно передать в качестве любого объектного аргумента при вызове функций (хотя на практике многие методы считают такое *значение* некорректным).

Память в *Java* с точки зрения программиста представляется не нулями и единицами или набором байтов, а как некое *виртуальное пространство*, в котором существуют объекты. И *доступ* к памяти осуществляется не по физическому адресу или указателю, а лишь через ссылки на объекты. *Ссылка* возвращается при создании объекта и далее может быть сохранена в переменной, передана в качестве аргумента и т.д. Как уже говорилось, допускается наличие нескольких ссылок на один *объект*. Возможна и противоположная ситуация – когда на какой-то *объект* не существует ни одной ссылки. Такой *объект* уже недоступен программе и является "мусором", то есть без толку занимает аппаратные ресурсы. Для их освобождения не требуется никаких усилий. В состав любой виртуальной машины обязательно входит автоматический сборщик мусора *`garbage collector`* – *фоновый процесс*, который как раз и занимается уничтожением ненужных объектов.

Очень важно помнить, что *объектная переменная*, в отличие от примитивной, может иметь *значение* другого типа, не совпадающего с типом переменной. Например, если *тип переменной* – некий *класс*, то*переменная* может ссылаться на *объект*, порожденный от наследника этого класса. Все случаи подобного несовпадения будут рассмотрены в следующих разделах курса.

Теперь рассмотрим *примитивные* и *ссылочные* типы данных более подробно.

Примитивные типы

Как уже говорилось, существует 8 *простых типов данных*, которые делятся на *целочисленные* (*integer*), *дробные* (*floating-point*) и *булевы* (*boolean*).

Целочисленные типы

Целочисленные типы – это `byte, short, int, long`, также к ним относят и `char` . Первые четыре типа имеют длину 1, 2, 4 и 8 байт соответственно, длина `char` – 2 байта, это непосредственно следует из того, что все символы Java описываются стандартом Unicode. Длины типов приведены только для оценки областей значения. Как уже говорилось, память в Java представляется виртуальной и вычислить, сколько физических ресурсов займет та или иная переменная, так прямолинейно не получится.

4 основных типа являются знаковыми. `char` добавлен к целочисленным типам данных, так как с точки зрения JVM символ и его код – понятия взаимооднозначные. Конечно, код символа всегда положительный, поэтому `char` – единственный беззнаковый тип. Инициализировать его можно как символьным, так и целочисленным литералом. Во всем остальном `char` – полноценный числовой тип данных, который может участвовать, например, в арифметических действиях, операциях сравнения и т.п. В таблице 4.1 сведены данные по всем разобранным типам:

Таблица 4.1. Целочисленные типы данных.

Название типа	Длина (байты)	Область значений
`byte`	1	`-128 .. 127`
`short`	2	`-32.768 .. 32.767`
`int`	4	`-2.147.483.648 .. 2.147.483.647`
`long`	8	`-9.223.372.036.854.775.808 .. 9.223.372.036.854.775.807` (примерно 10^{19})
`char`	2	`'\u0000' .. '\uffff'`, или `0 .. 65.535`

Обратите внимание, что `int` вмещает примерно 2 миллиарда, а потому подходит во многих случаях, когда не требуются сверхбольшие числа. Чтобы представить себе размеры типа `long`, укажем, что именно он используется в Java для отсчета времени. Как и во многих языках, время отсчитывается от 1 января 1970 года в миллисекундах. Так вот, вместимость `long` позволяет отсчитывать время на протяжении миллионов веков(!), причем как в будущее, так и в прошлое.

Почему были выделены именно эти два типа, `int` и `long`? Дело в том, что целочисленные литералы имеют тип `int` по умолчанию, или тип `long`, если стоит буква `L` или `l`. Именно поэтому корректным литералом считается только такое число, которое укладывается в 4 или 8 байт, соответственно. Иначе компилятор сочтет это ошибкой. Таким образом, следующие литералы являются корректными:

```
1
-2147483648
2147483648L
0L
1111111111111111L
```

Над целочисленными аргументами можно производить следующие операции:

- операции сравнения (возвращают булево значение)
  - `<, <=, >, >=`
  - `==, !=`
- числовые операции (возвращают числовое значение)
  - унарные операции `+` и `-`
  - арифметические операции `+, -, *, /, %`
  - операции инкремента и декремента (в префиксной и постфиксной форме): `++` и `--`
  - операции битового сдвига `<<, >>, >>>`
  - *битовые операции* `~, &, |, ^`
- оператор с условием `?:`
- оператор приведения типов
- оператор конкатенации со строкой `+`

Операторы сравнения вполне очевидны и отдельно мы их рассматривать не будем. Их результат всегда *булева* типа (`true` или `false`).

Работа числовых операторов также понятна, к тому же пояснялась в предыдущей лекции. Единственное уточнение можно сделать относительно операторов `+` и `-`, которые могут быть как бинарными (иметь два операнда), так и унарными (иметь один операнд). Бинарные операнды являются операторами сложения и вычитания, соответственно. *Унарный оператор* `+` возвращает значение, равное аргументу (`+x`всегда равно `x`). *Унарный оператор* `-`, примененный к значению `x`, возвращает результат, равный `0-x`. Неожиданный эффект имеет место в том случае, если аргумент равен наименьшему возможному значению примитивного типа.

```
int x=-2147483648;   // наименьшее возможное
```

```
                                // значение типа int
int y=-x;
```

Теперь значение переменной `y` на самом деле равно не `2147483648`, поскольку такое число не укладывается в область значений типа `int`, а в точности равно значению `x`! Другими словами, в этом примере выражение `-x==x` истинно!

Дело в том, что если при выполнении числовых операций над целыми числами возникает переполнение и результат не может быть сохранен в данном примитивном типе, то Java не создает никаких ошибок. Вместо этого все старшие биты, которые превышают вместимость типа, просто отбрасываются. Это может привести не только к потере точной абсолютной величины результата, но даже к искажению его знака, если на месте знакового бита окажется противоположное значение.

```
int x= 300000;
print(x*x);
```

Результатом такого примера будет:

```
-194313216
```

Возвращаясь к инвертированию числа `-2147483648`, мы видим, что математический результат равен в точности $+2^{31}$, или, в двоичном формате, `1000 0000 0000 0000 0000 0000 0000 0000` (единица и 31 ноль). Но тип `int` рассматривает первую единицу как *знаковый бит*, и результат получается равным `-2147483648`.

Таким образом, явное выписывание в коде литералов, которые слишком велики для используемых типов, приводит к ошибке компиляции (см. лекцию 3). Если же переполнение возникает в результате выполнения операции, "лишние" биты просто отбрасываются.

Подчеркнем, что выражение типа `-5` не является целочисленным литералом. На самом деле оно состоит из литерала `5` и оператора `-`. Напомним, что некоторые литералы (например, `2147483648`) могут встречаться только в сочетании с унарным оператором `-`.

Кроме того, числовые операции в Java обладают еще одной особенностью. Хотя целочисленные типы имеют длину 8, 16, 32 и 64 бита, вычисления проводятся только с 32-х и 64-х битной точностью. А это значит, что перед вычислениями может потребоваться преобразовать тип одного или нескольких операндов.

Если хотя бы один *аргумент операции* имеет тип `long`, то все аргументы приводятся к этому типу, и результат операции также будет типа `long`. Вычисление будет произведено с точностью в 64 бита, а более старшие биты, если таковые появляются в результате, отбрасываются.

Если же аргументов типа `long` нет, то вычисление производится с точностью в 32 бита, и все аргументы преобразуются в `int` (это относится к `byte, short, char`). Результат также имеет тип `int`. Все биты старше 32-го игнорируются.

Никакого способа узнать, произошло ли переполнение, нет. Расширим рассмотренный пример:

```
int i=300000;
print(i*i);   // умножение с точностью 32 бита
long m=i;
print(m*m);   // умножение с точностью 64 бита
print(1/(m-i)); // попробуем получить разность
                // значений int и long
```

Результатом такого примера будет:

```
-194313216
90000000000
```

затем мы получим ошибку деления на ноль, поскольку переменные `i` и `m` хоть и разных типов, но хранят одинаковое математическое значение и их разность равна нулю. Первое умножение производилось с точностью в 32 бита, более старшие биты были отброшены. Второе – с точностью в 64 бита, ответ не исказился.

Вопрос приведения типов, и в том числе специальный оператор для такого действия, подробно рассматривается в следующих лекциях. Однако здесь хотелось бы отметить несколько примеров, которые не столь очевидны и могут создать проблемы при написании программ. Во-первых, подчеркнем, что результатом операции с целочисленными аргументами всегда является целое число. А значит, в следующем примере

```
double x = 1/2;
```

переменной `x` будет присвоено значение `0`, а не `0.5`, как можно было бы ожидать. Подробно операции с дробными аргументами рассматриваются ниже, но чтобы получить значение `0.5`, достаточно написать `1./2` (теперь первый аргумент дробный и результат не будет округлен).

Как уже упоминалось, время в Java измеряется в миллисекундах. Попробуем вычислить, сколько миллисекунд содержится в неделе и в месяце:

```
print(1000*60*60*24*7);
   // вычисление для недели
print(1000*60*60*24*30);
   // вычисление для месяца
```

Необходимо перемножить количество миллисекунд в одной секунде (1000), секунд – в минуте (60), минут – в часе (60), часов – в дне (24) и дней — в неделе и месяце (7 и 30, соответственно). Получаем:

```
604800000
-1702967296
```

Очевидно, во втором вычислении произошло переполнение. Достаточно сделать последний аргумент величиной типа `long`:

```
print(1000*60*60*24*30L);
   // вычисление для месяца
```

Получаем правильный результат:

```
2592000000
```

Подобные вычисления разумно переводить на 64-битную точность не на последней операции, а заранее, чтобы избежать переполнения.

Понятно, что типы большей длины могут хранить больший спектр значений, а потому Java не позволяет присвоить переменной меньшего типа значение большего типа. Например, такие строки вызовут ошибку компиляции:

```
// пример вызовет ошибку компиляции
int x=1;
byte b=x;
```

Хотя для программиста и очевидно, что переменная `b` должна получить значение `1`, что легко укладывается в тип `byte`, однако компилятор не может вычислять значение переменной `x` при обработке второй строки, он знает лишь, что ее тип – `int`.

А вот менее очевидный пример:

```
// пример вызовет ошибку компиляции
byte b=1;
byte c=b+1;
```

И здесь компилятор не сможет успешно завершить работу. При операции сложения значение переменной `b` будет преобразовано в тип `int` и таким же будет результат сложения, а значит, его нельзя так просто присвоить переменной типа `byte`.

Аналогично:

```
// пример вызовет ошибку компиляции
int x=2;
long y=3;
int z=x+y;
```

Здесь результат сложения будет уже типа `long`. Точно так же некорректна такая инициализация:

```
// пример вызовет ошибку компиляции
byte b=5;
byte c=-b;
```

Даже *унарный оператор* " `-` " проводит вычисления с точностью не меньше 32 бит.

Хотя во всех случаях инициализация переменных приводилась только для примера, а предметом рассмотрения были числовые операции, укажем корректный способ преобразовать тип числового значения:

```
byte b=1;
byte c=(byte)-b;
```

Итак, все числовые операторы возвращают результат типа `int` или `long`. Однако существует два исключения.

Первое из них – операторы инкрементации и декрементации. Их действие заключается в прибавлении или вычитании единицы из значения переменной, после чего результат сохраняется в этой переменной и значение всей операции равно значению переменной (до или после изменения, в зависимости от того, является оператор префиксным или постфиксным). А значит, и тип значения совпадает с типом переменной. (На самом деле, вычисления все равно производятся с точностью минимум 32 бита, однако при присвоении переменной результата его тип понижается.)

```
byte x=5;
byte y1=x++;      // на момент начала исполнения x равен 5
byte y2=x--;       // на момент начала исполнения x равен 6
byte y3=++x;     // на момент начала исполнения x равен 5
byte y4=--x;       // на момент начала исполнения x равен 6
println(y1);
```

```
println(y2);
println(y3);
println(y4);
```

В результате получаем:

```
5
6
6
5
```

Никаких проблем с присвоением результата операторов `++` и `--` переменным типа `byte`. Завершая рассмотрение этих операторов, приведем еще один пример:

```
byte x=-128;
print(-x);

byte y=127;
print(++y);
```

Результатом будет:

```
128
-128
```

Этот пример иллюстрирует вопросы преобразования типов при вычислениях и случаи переполнения.

Вторым исключением является оператор с условием `?`:. Если второй и третий операнды имеют одинаковый тип, то и результат операции будет такого же типа.

```
byte x=2;
byte y=3;
byte z=(x>y) ? x : y;
   // верно, x и y одинакового типа
byte abs=(x>0) ? x : -x;
   // неверно!
```

Последняя строка неверна, так как третий аргумент содержит числовую операцию, стало быть, его тип `int`, а значит, и тип всей операции будет `int`, и присвоение некорректно. Даже если второй аргумент имеет тип `byte`, а третий – `short`, значение будет типа `int`.

Наконец, рассмотрим оператор конкатенации со строкой. Оператор `+` может принимать в качестве аргумента строковые величины. Если одним из

аргументов является строка, а вторым – целое число, то число будет преобразовано в текст и строки объединятся.

```
int x=1;
print("x="+x);
```

Результатом будет:

```
x=1
```

Обратите внимание на следующий пример:

```
print(1+2+"text");
print("text"+1+2);
```

Его результатом будет:

```
3text
text12
```

Отдельно рассмотрим работу с типом `char`. Значения этого типа могут полноценно участвовать в числовых операциях:

```
char c1=10;
char c2='A';
   // латинская буква A (\u0041, код 65)
int i=c1+c2-'B';
```

Переменная `i` получит значение `9`.

Рассмотрим следующий пример:

```
char c='A';
print(c);
print(c+1);
print("c="+c);
print('c'+'='+c);
```

Его результатом будет:

```
A
66
c=A
225
```

В первом случае в метод `print` было передано значение типа `char`, поэтому отобразился символ. Во втором случае был передан результат сложения, то есть число, и именно число появилось на экране. Далее при сложении со

строкой тип `char` был преобразован в текст в виде символа. Наконец в последней строке произошло сложение трех чисел: `'c' (код 99), '=' (код 61)` и переменной c (т.е. код `'A' - 65`).

Для каждого примитивного типа существуют специальные вспомогательные *классы-обертки* (wrapper classes). Для типов `byte`, `short`, `int`, `long`, `char` это `Byte`, `Short`, `Integer`, `Long`, `Character`. Эти классы содержат многие полезные методы для работы с целочисленными значениями. Например, преобразование из текста в число. Кроме того, есть класс `Math`, который хоть и предназначен в основном для работы с дробными числами, но также предоставляет некоторые возможности и для целых.

В заключение подчеркнем, что единственные операции с целыми числами, при которых Java генерирует ошибки,– это деление на ноль (операторы `/` и `%`).

Дробные типы

Дробные типы – это `float` и `double` . Их длина - 4 и 8 байт, соответственно. Оба типа знаковые. Ниже в таблице сведены их характеристики:

Таблица 4.2. Дробные типы данных.

Название типа	Длина (байты)	Область значений
`float`	4	`3.40282347e+38f ; 1.40239846e-45f`
`double`	8	`1.79769313486231570e+308 ; 4.94065645841246544e-324`

Для целочисленных типов область значений задавалась верхней и нижней границами, весьма близкими по модулю. Для дробных типов добавляется еще одно ограничение – насколько можно приблизиться к нулю, другими словами – каково наименьшее положительное ненулевое значение. Таким образом, нельзя задать литерал заведомо больший, чем позволяет соответствующий тип данных, это приведет к ошибке***overflow***. И нельзя задать литерал, значение которого по модулю слишком мало для данного типа, компилятор сгенерирует ошибку *underflow*.

```
// пример вызовет ошибку компиляции
```

```
float f = 1e40f;
// значение слишком велико, overflow
double d = 1e-350;
// значение слишком мало, underflow
```

Напомним, что если в конце литерала стоит буква `F` или `f`, то литерал рассматривается как значение типа `float`. По умолчанию дробный литерал имеет тип `double`, при желании это можно подчеркнуть буквой `D` или `d`.

Над дробными аргументами можно производить следующие операции:

- операции сравнения (возвращают булево значение)
  - `<, <=, >, >=`
  - `==, !=`
- числовые операции (возвращают числовое значение)
  - унарные операции `+` и `-`
  - арифметические операции `+, -, *, /, %`
  - операции инкремента и декремента (в префиксной и постфиксной форме): `++` и `--`
- оператор с условием `?`:
- оператор приведения типов
- оператор конкатенации со строкой `+`

Практически все операторы действуют по тем же принципам, которые предусмотрены для целочисленных операторов (оператор деления с остатком `%` рассматривался в предыдущей лекции, а операторы `++` и`--` также увеличивают или уменьшают значение переменной на единицу). Уточним лишь, что операторы сравнения корректно работают и в случае сравнения целочисленных значений с дробными. Таким образом, в основном необходимо рассмотреть вопросы переполнения и преобразования типов при вычислениях.

Для дробных вычислений появляется уже два типа переполнения – *overflow* и *underflow*. Тем не менее, Java и здесь никак не сообщает о возникновении подобных ситуаций. Нет ни ошибок, ни других способов обнаружить их. Более того, даже деление на ноль не приводит к некорректной ситуации. А значит, дробные вычисления вообще не порождают никаких ошибок.

Такая свобода связана с наличием специальных значений дробного типа. Они определяются спецификацией *IEEE 754* и уже перечислялись в лекции 3:

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

Все эти значения представлены как для типа `float`, так и для `double`.

Положительную и отрицательную бесконечности можно получить следующим образом:

```
1f/0f     // положительная бесконечность,
          // тип float
-1d/0d    // отрицательная бесконечность,
          // тип double
```

Также в классах `Float` и `Double` определены константы `POSITIVE_INFINITY` и `NEGATIVE_INFINITY`. Как видно из примера, такие величины получаются при делении конечных величин на ноль.

Значение `NaN` можно получить, например, в результате следующих действий:

```
0.0/0.0          // деление ноль на ноль
(1.0/0.0)*0.0    // умножение бесконечности на ноль
```

Эта величина также представлена константами `NaN` в классах `Float` и `Double`.

Величины положительный и отрицательный ноль записываются очевидным образом:

```
0.0     // дробный литерал со значением
        // положительного нуля
+0.0    // унарная операция +, ее значение -
        // положительный ноль
-0.0    // унарная операция -, ее значение -
        // отрицательный ноль
```

Все дробные значения строго упорядочены. Отрицательная бесконечность меньше любого другого дробного значения, положительная – больше. Значения `+0.0` и `-0.0` считаются равными, то есть выражение`0.0==-0.0` истинно, а `0.0>-0.0` – ложно. Однако другие операторы различают их, например, выражение `1.0/0.0` дает положительную бесконечность, а `1.0/-0.0` – отрицательную.

Единственное исключение - значение `NaN`. Если хотя бы один из *аргументов операции* сравнения равняется `NaN`, то результат заведомо будет `false` (для оператора `!=` соответственно всегда `true`). Таким образом, единственное значение `x`, при котором выражение `x!=x` истинно,– именно `NaN`.

Возвращаемся к вопросу переполнения в числовых операциях. Если получаемое значение слишком велико по модулю (*overflow*), то результатом будет бесконечность соответствующего знака.

```
print(1e20f*1e20f);
print(-1e200*1e200);
```

В результате получаем:

```
Infinity
-Infinity
```

Если результат, напротив, получается слишком мал (*underflow*), то он просто округляется до нуля. Так же поступают и в том случае, когда количество десятичных знаков превышает допустимое:

```
print(1e-40f/1e10f);   // underflow для float
print(-1e-300/1e100);  // underflow для double
float f=1e-6f;
print(f);
f+=0.002f;
print(f);
f+=3;
print(f);
f+=4000;
print(f);
```

Результатом будет:

```
0.0
-0.0

1.0E-6
0.002001
3.002001
4003.002
```

Как видно, в последней строке был утрачен 6-й разряд после десятичной точки.

Другой пример (из спецификации языка Java):

```
double d = 1e-305 * Math.PI;
print(d);
for (int i = 0; i < 4; i++)
print(d /= 100000);
```

Результатом будет:

```
3.141592653589793E-305
3.1415926535898E-310
3.141592653E-315
3.142E-320
0.0
```

Таким образом, как и для целочисленных значений, явное выписывание в коде литералов, которые слишком велики (*overflow*) или слишком малы (*underflow*) для используемых типов, приводит к ошибке компиляции (см. лекцию 3). Если же переполнение возникает в результате выполнения операции, то возвращается одно из специальных значений.

Теперь перейдем к преобразованию типов. Если хотя бы один аргумент имеет тип `double`, то значения всех аргументов приводятся к этому типу и результат операции также будет иметь тип `double`. Вычисление будет произведено с точностью в 64 бита.

Если же аргументов типа `double` нет, а хотя бы один аргумент имеет тип `float`, то все аргументы приводятся к `float`, вычисление производится с точностью в 32 бита и результат имеет тип `float`.

Эти утверждения верны и в случае, если один из аргументов целочисленный. Если хотя бы один из аргументов имеет значение `NaN`, то и результатом операции будет `NaN`.

Еще раз рассмотрим простой пример:

```
print(1/2);
print(1/2.);
```

Результатом будет:

```
0
0.5
```

Достаточно одного дробного аргумента, чтобы результат операции также имел дробный тип.

Более сложный пример:

```
int x=3;
int y=5;
print (x/y);
print((double)x/y);
print(1.0*x/y);
```

Результатом будет:

```
0
0.6
0.6
```

В первый раз оба аргумента были целыми, поэтому в результате получился ноль. Однако поскольку оба операнда представлены переменными, в этом примере нельзя просто поставить десятичную точку и таким образом перевести вычисления в дробный тип. Необходимо либо преобразовать один из аргументов (второй вывод на экран), либо вставить еще одну фиктивную операцию с дробным аргументом (последняя строка).

Приведение типов подробно рассматривается в другой лекции, однако обратим здесь внимание на несколько моментов.

Во-первых, при приведении дробных значений к целым типам дробная часть просто отбрасывается. Например, число `3.84` будет преобразовано в целое `3`, а `-3.84` превратится в `-3`. Для математического округления необходимо воспользоваться методом класса `Math.round(…)`.

Во-вторых, при приведении значений `int` к типу `float` и при приведении значений типа `long` к типу `float` и `double` возможны потери точности, несмотря на то, что эти дробные типы вмещают гораздо большие числа, чем соответствующие целые. Рассмотрим следующий пример:

```
long l=111111111111L;
float f = l;
l = (long) f;
print(l);
```

Результатом будет:

```
111111110656
```

Тип `float` не смог сохранить все значащие разряды, хотя преобразование от `long` к `float` произошло без специального оператора в отличие от обратного перехода.

Для каждого примитивного типа существуют специальные вспомогательные *классы-обертки* (wrapper classes). Для типов `float` и `double` это `Float` и `Double`. Эти классы содержат многие полезные методы для работы с дробными значениями. Например, преобразование из текста в число.

Кроме того, класс `Math` предоставляет большое количество методов для операций над дробными значениями, например, извлечение квадратного корня, возведение в любую степень, тригонометрические и другие. Также в этом классе определены константы `PI` и основание натурального логарифма `E`.

Булев тип

Булев тип представлен всего одним типом `boolean`, который может хранить всего два возможных значения – `true` и `false` . Величины именно этого типа получаются в результате операций сравнения.

Над булевыми аргументами можно производить следующие операции:

- операции сравнения (возвращают булево значение)
  - `==, !=`
- логические операции (возвращают булево значение)
  - !
  - `&, |, ^`
  - `&&, ||`
- оператор с условием `?`:
- оператор конкатенации со строкой `+`

Логические операторы `&&` и `||` обсуждались в предыдущей лекции. В операторе с условием `?`: первым аргументом может быть только значение типа `boolean`. Также допускается, чтобы второй и третий аргументы одновременно имели булев тип.

Операция конкатенации со строкой превращает булеву величину в текст `"true"` или `"false"` в зависимости от значения.

Только булевы выражения допускаются для управления потоком вычислений, например, в качестве критерия условного перехода `if`.

Никакое число не может быть интерпретировано как булево выражение. Если предполагается, что ненулевое значение эквивалентно истине (по правилам языка C), то необходимо записать `x!=0`. Ссылочные величины можно преобразовывать в `boolean` выражением `ref!=null`.

Ссылочные типы

Итак, *выражение* ссылочного типа имеет *значение* либо `null`, либо ссылку, указывающую на некоторый *объект* в виртуальной памяти *JVM*.

Объекты и правила работы с ними

Объект (object) – это экземпляр некоторого класса, или экземпляр массива. Массивы будут подробно рассматриваться в соответствующей лекции. Класс – это описание объектов одинаковой структуры, и если в программе такой класс используется, то описание присутствует в единственном экземпляре. Объектов этого класса может не быть вовсе, а может быть создано сколь угодно много.

Объекты всегда создаются с использованием ключевого слова `new`, причем одно слово `new` порождает строго один объект (или вовсе ни одного, если происходит ошибка). После ключевого слова указывается имя класса, от которого мы собираемся породить объект. Создание объекта всегда происходит через вызов одного из конструкторов класса (их может быть несколько), поэтому в заключение ставятся скобки, в которых перечислены значения аргументов, передаваемых выбранному конструктору. В приведенных выше примерах, когда создавались объекты типа `Point`, выражение `new Point (3,5)` означало обращение к конструктору класса `Point`, у которого есть два аргумента типа `int`. Кстати, обязательное объявление такого конструктора в упрощенном объявлении класса отсутствовало. Объявление классов рассматривается в следующих лекциях, однако приведем правильное определение `Point`:

```
class Point {
   int x, y;

   /**
      * Конструктор принимает 2 аргумента,
      * которыми инициализирует поля объекта.
      */
   Point (int newx, int newy){
      x=newx;
      y=newy;
   }
}
```

Если конструктор отработал успешно, то выражение `new` возвращает ссылку на созданный объект. Эту ссылку можно сохранить в переменной, передать в качестве аргумента в какой-либо метод или использовать другим способом. JVM всегда занимается подсчетом хранимых ссылок на каждый объект. Как только обнаруживается, что ссылок больше нет, такой объект предназначается для уничтожения сборщиком мусора (*garbage collector*). Восстановить ссылку на такой "потерянный" объект невозможно.

```
Point p=new Point(1,2);
   // Создали объект, получили на него ссылку
Point p1=p;
   // теперь есть 2 ссылки на точку (1,2)
```

```
p=new Point(3,4);
   // осталась одна ссылка на точку (1,2)
p1=null;
```

Ссылок на объект-точку `(1,2)` больше нет, доступ к нему утерян и он вскоре будет уничтожен сборщиком мусора.

Любой объект порождается только с применением ключевого слова `new`. Единственное исключение – экземпляры класса `String`. Записывая любой строковый литерал, мы автоматически порождаем объект этого класса. Оператор конкатенации `+`, результатом которого является строка, также неявно порождает объекты без использования ключевого слова `new`.

Рассмотрим пример:

```
"abc"+"def"
```

При выполнении этого выражения будет создано три объекта класса `String`. Два объекта порождаются строковыми литералами, третий будет представлять результат конкатенации.

Операция создания объекта – одна из самых ресурсоемких в Java. Поэтому следует избегать ненужных порождений. Поскольку при работе со строками их может создаваться довольно много, компилятор, как правило, пытается оптимизировать такие выражения. В рассмотренном примере, поскольку все операнды являются константами времени компиляции, компилятор сам осуществит конкатенацию и вставит в код уже результат, сократив таким образом количество создаваемых объектов до одного.

Кроме того, в версии Java 1.1 была введена технология *reflection*, которая позволяет обращаться к классам, методам и полям, используя лишь их имя в текстовом виде. С ее помощью также можно создать объект без ключевого слова `new`, однако эта технология довольно специфична, применяется в редких случаях, а кроме того, довольно проста и потому в данном курсе не рассматривается. Все же приведем пример ее применения:

```
Point p = null;
try {
  // в следующей строке, используя лишь
  // текстовое имя класса Point, порождается
  // объект без применения ключевого слова new
p=(Point)Class.forName("Point").newInstance();

} catch (Exception e) { // обработка ошибок
   System.out.println(e);
}
```

Объект всегда "помнит", от какого класса он был порожден. С другой стороны, как уже указывалось, можно ссылаться на объект, используя ссылку другого типа. Приведем пример, который будем еще много раз использовать. Сначала опишем два класса, `Parent` и его наследник `Child`:

```
// Объявляем класс Parent
class Parent {
}

// Объявляем класс Child и наследуем
// его от класса Parent
class Child extends Parent {
}
```

Пока нам не нужно определять какие-либо поля или методы. Далее объявим переменную одного типа и проинициализируем ее значением другого типа:

```
Parent p = new Child();
```

Теперь переменная типа `Parent` указывает на объект, порожденный от класса `Child`.

Над *ссылочными значениями* можно производить следующие операции:

- обращение к полям и методам объекта
- оператор `instanceof` (возвращает булево значение)
- операции сравнения `==` и `!=` (возвращают булево значение)
- оператор приведения типов
- оператор с условием `?`:
- оператор конкатенации со строкой `+`

Обращение к полям и методам объекта можно назвать основной операцией над ссылочными величинами. Осуществляется она с помощью символа . (точка). Примеры ее применения будут приводиться.

Используя оператор `instanceof`, можно узнать, от какого класса произошел объект. Этот оператор имеет два аргумента. Слева указывается ссылка на объект, а справа – имя типа, на совместимость с которым проверяется объект. Например:

```
Parent p = new Child();
// проверяем переменную p типа Parent
// на совместимость с типом Child
print(p instanceof Child);
```

Результатом будет `true`. Таким образом, оператор `instanceof` опирается не на тип ссылки, а на свойства объекта, на который она ссылается. Но этот оператор возвращает истинное значение не только для того типа, от которого был порожден объект. Добавим к уже объявленным классам еще один:

```
// Объявляем новый класс и наследуем
// его от класса Child
class ChildOfChild extends Child { }
```

Теперь заведем переменную нового типа:

```
Parent p = new ChildOfChild();
print(p instanceof Child);
```

В первой строке объявляется переменная типа `Parent`, которая инициализируется ссылкой на объект, порожденный от `ChildOfChild`. Во второй строке оператор `instanceof` анализирует совместимость ссылки типа `Parent` с классом `Child`, причем задействованный объект не порожден ни от первого, ни от второго класса. Тем не менее, оператор вернет `true`, поскольку класс, от которого этот объект порожден, наследуется от `Child`.

Добавим еще один класс:

```
class Child2 extends Parent {
}
```

И снова объявим переменную типа `Parent`:

```
Parent p=new Child();
print(p instanceof Child);
print(p instanceof Child2);
```

Переменная `p` имеет тип `Parent`, а значит, может ссылаться на объекты типа `Child` или `Child2`. Оператор `instanceof` помогает разобраться в ситуации:

```
true
false
```

Для ссылки, равной `null`, оператор `instanceof` всегда вернет значение `false`.

С изучением свойств объектной модели Java мы будем возвращаться к алгоритму работы оператора `instanceof`.

Операторы сравнения `==` и `!=` проверяют равенство (или неравенство) объектных величин именно по ссылке. Однако часто требуется

альтернативное сравнение – по значению. Сравнение по значению имеет дело с понятием состояние объекта. Сам смысл этого выражения рассматривается в ООП, что же касается реализации в Java, то состояние объекта хранится в его полях. При сравнении по ссылке ни тип объекта, ни значения его полей не учитываются, `true` возвращается только в том случае, если обе ссылки указывают на один и тот же объект.

```
Point p1=new Point(2,3);
Point p2=p1;
Point p3=new Point(2,3);
print(p1==p2);
print(p1==p3);
```

Результатом будет:

```
true
false
```

Первое сравнение оказалось истинным, так как переменная `p2` ссылается на тот же объект, что и `p1`. Второе же сравнение ложно, несмотря на то, что переменная `p3` ссылается на объект-точку с точно такими же координатами. Однако это другой объект, который был порожден другим выражением `new`.

Если один из аргументов оператора `==` равен `null`, а другой – нет, то значение такого выражения будет `false`. Если же оба операнда `null`, то результат будет `true`.

Для корректного сравнения по значению существует специальный метод `equals`, который будет рассмотрен позже. Например, строки можно сравнивать следующим образом:

```
String s = "abc";
s=s+1;
print(s.equals("abc1"));
```

Операция с условием `?:` работает как обычно и может принимать второй и третий аргументы, если они оба одновременно ссылочного типа. Результат такого оператора также будет иметь объектный тип.

Как и простые типы, ссылочные величины можно складывать со строкой. Если ссылка равна `null`, то к строке добавляется текст `"null"`. Если же ссылка указывает на объект, то у него вызывается специальный метод (он будет рассмотрен ниже, его имя `toString()`) и текст, который он вернет, будет добавлен к строке.

Класс Object

В Java множественное наследование отсутствует. Каждый класс может иметь только одного родителя. Таким образом, мы можем проследить цепочку наследования от любого класса, поднимаясь все выше. Существует класс, на котором такая цепочка всегда заканчивается, это класс `Object`. Именно от него наследуются все классы, в объявлении которых явно не указан другой родительский класс. А значит, любой класс напрямую, или через своих родителей, является наследником `Object`. Отсюда следует, что методы этого класса есть у любого объекта (поля в `Object` отсутствуют), а потому они представляют особенный интерес.

Рассмотрим основные из них.

getClass()

Этот метод возвращает объект класса `Class`, который описывает класс, от которого был порожден этот объект. Класс `Class` будет рассмотрен ниже. У него есть метод `getName()`, возвращающий имя класса:

```
String s = "abc";
Class cl=s.getClass();
System.out.println(cl.getName());
```

Результатом будет строка:

```
java.lang.String
```

В отличие от оператора `instanceof`, метод `getClass()` всегда возвращает точно тот класс, от которого был порожден объект.

equals()

Этот метод имеет один аргумент типа `Object` и возвращает `boolean`. Как уже говорилось, `equals()` служит для сравнения объектов по значению, а не по ссылке. Сравнивается состояние объекта, у которого вызывается этот метод, с передаваемым аргументом.

```
Point p1=new Point(2,3);
Point p2=new Point(2,3);
print(p1.equals(p2));
```

Результатом будет `false`.

Поскольку сам `Object` не имеет полей, а значит, и состояния, в этом классе метод `equals` возвращает результат сравнения по ссылке. Однако при написании нового класса можно переопределить этот метод и описать правильный алгоритм сравнения по значению (что и сделано в большинстве

стандартных классов). Соответственно, в класс `Point` также необходимо добавить переопределенный метод сравнения:

```
public boolean equals(Object o) {
   // Сначала необходимо убедиться, что
   // переданный объект совместим с типом
   // Point
   if (o instanceof Point) {
      // Типы совместимы, можно провести
      // преобразование
      Point p = (Point)o;
      // Возвращаем результат сравнения
      // координат
      return p.x==x && p.y==y;
   }
   // Если объект не совместим с Point,
   // возвращаем false
   return false;
}
```

hashCode()

Данный метод возвращает значение `int`. Цель `hashCode()` – представить любой объект целым числом. Особенно эффективно это используется в хэш-таблицах (в Java есть стандартная реализация такого хранения данных, она будет рассмотрена позже). Конечно, нельзя потребовать, чтобы различные объекты возвращали различные хэш-коды, но, по крайней мере, необходимо, чтобы объекты, равные по значению (метод `equals()` возвращает `true`), возвращали одинаковые хэш-коды.

В классе `Object` этот метод реализован на уровне JVM. Сама виртуальная машина генерирует число хеш-кодов, основываясь на расположении объекта в памяти.

toString()

Этот метод позволяет получить текстовое описание любого объекта. Создавая новый класс, данный метод можно переопределить и возвращать более подробное описание. Для класса `Object` и его наследников, не переопределивших `toString()`, метод возвращает следующее выражение:

```
getClass().getName()+"@"+hashCode()
```

Метод `getName()` класса `Class` уже приводился в пример, а хэш-код еще дополнительно обрабатывается специальной функцией для представления в шестнадцатеричном формате.

Например:

```
print(new Object());
```

Результатом будет:

```
java.lang.Object@92d342
```

В результате этот метод позволяет по текстовому описанию понять, от какого класса был порожден объект и, благодаря хеш-коду, различать разные объекты, созданные от одного класса.

Именно этот метод вызывается при конвертации объекта в текст, когда он передается в качестве аргумента оператору конкатенации строк.

finalize()

Данный метод вызывается при уничтожении объекта автоматическим сборщиком мусора (*garbage collector*). В классе `Object` он ничего не делает, однако в классе-наследнике позволяет описать все действия, необходимые для корректного удаления объекта, такие как закрытие соединений с БД, сетевых соединений, снятие блокировок на файлы и т.д. В обычном режиме напрямую этот метод вызывать не нужно, он отработает автоматически. Если необходимо, можно обратиться к нему явным образом.

В методе `finalize()` нужно описывать только дополнительные действия, связанные с логикой работы программы. Все необходимое для удаления объекта JVM сделает сама.

Класс String

Как уже указывалось, класс `String` занимает в Java особое положение. Экземпляры только этого класса можно создавать без использования ключевого слова new. Каждый строковый литерал порождает экземпляр `String`, и это единственный литерал (кроме `null`), имеющий объектный тип.

Затем значение любого типа может быть приведено к строке с помощью оператора конкатенации строк, который был рассмотрен для каждого типа, как примитивного, так и объектного.

Еще одним важным свойством данного класса является неизменяемость. Это означает, что, породив объект, содержащий некое значение-строку, мы уже не можем изменить данное значение – необходимо создать новый объект.

```
String s="a";
s="b";
```

Во второй строке переменная сменила свое значение, но только создав новый объект класса `String`.

Поскольку каждый строковый литерал порождает новый объект, что есть очень ресурсоемкая операция в Java, зачастую компилятор стремится оптимизировать эту работу.

Во-первых, если используется несколько литералов с одинаковым значением, для них будет создан один и тот же объект.

```
String s1 = "abc";
String s2 = "abc";
String s3 = "a"+"bc";
print(s1==s2);
print(s1==s3);
```

Результатом будет:

```
true
true
```

То есть в случае, когда строка конструируется из констант, известных уже на момент компиляции, оптимизатор также подставляет один и тот же объект.

Если же строка создается выражением, которое может быть вычислено только во время исполнения программы, то оно будет порождать новый объект:

```
String s1="abc";
String s2="ab";
print(s1==(s2+"c"));
```

Результатом будет `false`, так как компилятор не может предсказать результат сложения значения переменной с константой.

В классе `String` определен метод `intern()`, который возвращает один и тот же объект-строку для всех экземпляров, равных по значению. То есть, если для ссылок `s1` и `s2` верно выражение`s1.equals(s2)`, то верно и `s1.intern()==s2.intern()`.

Разумеется, в классе переопределены методы `equals()` и `hashCode()`. Метод `toString()` также переопределен и возвращает он сам объект-строку, то есть для любой ссылки `s` типа `String`, не равной`null`, верно выражение `s==s.toString()`.

Класс Class

Наконец, последний класс, который будет рассмотрен в этой лекции.

Класс `Class` является метаклассом для всех классов Java. Когда JVM загружает файл `.class`, который описывает некоторый тип, в памяти создается объект класса `Class`, который будет хранить это описание.

Например, если в программе есть строка

```
Point p=new Point(1,2);
```

то это означает, что в системе созданы следующие объекты:

1. объект типа `Point`, описывающий точку `(1,2)` ;
2. объект класса `Class`, описывающий класс `Point` ;
3. объект класса `Class`, описывающий класс `Object`. Поскольку класс `Point` наследуется от `Object`, его описание также необходимо;
4. объект класса `Class`, описывающий класс `Class`. Это обычный Java-класс, который должен быть загружен по общим правилам.

Одно из применений класса `Class` уже было рассмотрено – использование метода `getClass()` класса `Object`. Если продолжить последний пример с точкой:

```
Class cl=p.getClass();
   // это объект №2 из списка
Class cl2=cl.getClass();
   // это объект №4 из списка
Class cl3=cl2.getClass();
   // опять объект №4
```

Выражение `cl2==cl3` верно.

Другое применение класса `Class` также приводилось в примере применения технологии reflection.

Кроме прямого использования *метакласса* для хранения в памяти описания классов, Java использует эти объекты и для других целей, которые будут рассмотрены ниже (статические переменные, синхронизация статических методов и т.д.).

Заключение

Типы данных – одна из ключевых тем курса. Невозможно написать ни одной программы, не используя их. Вот *список* некоторых операций, где применяются типы:

- объявление типов;
- создание объектов;
- при объявлении полей – тип поля;
- при объявлении методов – входные параметры, возвращаемое значение;
- при объявлении конструкторов – входные параметры;
- оператор приведения типов;
- оператор `instanceof` ;
- объявление локальных переменных;
- многие другие – обработка ошибок, `import` -выражения и т.д.

Принципиальные различия между примитивными и *ссылочными типами данных* будут рассматриваться и дальше по ходу курса. Изучение объектной модели *Java* даст основу для более подробного изложения объектных типов – обычных и абстрактных классов, интерфейсов и массивов. После приведения типов будут описаны связи между типом переменной и типом ее значения.

В обсуждении будущей версии *Java* 1.5 упоминаются темплейты (*templates*), которые существенно расширят понятия типа данных, если действительно войдут в *стандарт языка*.

В лекции было рассказано о том, что *Java* является строго типизированным языком, то есть тип всех переменных и выражений определяется уже компилятором. Это позволяет существенно повысить *надежность* и качество кода, а также делает необходимым понимание программистами объектной модели.

Все типы в *Java* делятся на две группы – фиксированные простые, или примитивные типы (8 типов) и многочисленная *группа* объектных типов (классов). Примитивные типы действительно являются хранилищами данных своего типа. Ссылочные переменные хранят ссылку на некоторый *объект* совместимого типа. Они также могут принимать *значение* `null`, не указывая ни на какой *объект*. *JVM* подсчитывает количество ссылок на каждый *объект* и активизирует *механизм автоматической сборки мусора* для удаления неиспользуемых объектов.

Были рассмотрены переменные. Они характеризуются тремя основными параметрами – имя, тип и *значение*. Любая *переменная* должна быть объявлена и при этом может быть инициализирована. Возможно использование модификатора `final`.

Примитивные типы состоят из пяти целочисленных, включая символьный тип, двух дробных и одного булевого. Целочисленные литералы имеют ограничения, связанные с типами данных. Были рассмотрены

все*операторы* над примитивными типами, тип возвращаемого значения и тонкости их использования.

Затем изучались объекты, способы их создания и *операторы*, выполняющие над ними различные действия, в частности принцип работы оператора instanceof. Далее были рассмотрены самые главные классы в *Java*– `Object, Class, String`.