ИНТУИТ
НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ
Национальный Открытый Университет "ИНТУИТ": www.intuit.ru
Вадим Монахов
Лекция 3. Примитивные типы данных и операторы для работы с ними

3.1. Булевский (логический) тип

Величины типа `boolean` принимают значения `true` или `false`.

Объявление булевских переменных:

```
boolean a;
boolean b;
```

Использование в выражениях при присваиваниях:

```
a=true;
b=a;
```

Булевские величины обычно используются в *логических операторах* и в *операциях отношения*.

Таблица 3.1. *Логические операторы*

Оператор	Название	Пример
&&	логическое "И" (and)	a&&b
||	логическое "ИЛИ" (or)	a||b
^	логическое "исключающее ИЛИ" (xor)	a^b
!	логическое "НЕ" (not)	!a

Значением логического выражения являются `true` или `false`. Например, если `a=true, b=true`, то `a && b` имеет *значение* `true`. А при `a=false` или `b=false` *выражение* `a && b` принимает *значение* `false`.

Для работы с логическими выражениями часто применяют так называемые *таблицы истинности*. В них вместо логической единицы (`true`) пишут 1, а вместо логического нуля (`false`) пишут 0. В приведенной ниже таблице указаны значения логических выражений при всех возможных комбинациях значений `a` и `b`.

Таблица 3.2.

a	0	0	1	1
b	0	1	0	1
Выражение	Значения			
a&&b	0	0	0	1
a||b	0	1	1	1
a^b	0	1	1	0
!a	1	1	0	0

Выполнение булевских операторов происходит на аппаратном уровне, а значит, очень быстро. Реально *процессор* оперирует числами 0 и 1, хотя в *Java* это осуществляется скрытым от программиста образом.

Логические выражения в *Java* вычисляются в соответствии с так называемым укороченным *оцениванием*: из приведенной выше таблицы видно, что если a имеет *значение* `false`, то *значение* оператора `a&&b` будет равно `false` независимо от значения `b`. Поэтому если `b` является *булевским выражением*, его можно не вычислять. Аналогично, если a имеет *значение* `true`, то *значение* оператора `a||b` будет равно `true` независимо от значения `b`.

Операции отношения

Это операторы сравнения и принадлежности. Они имеют результат типа `boolean`. Операторы сравнения применимы к любым величинам `a` и `b` одного типа, а также к произвольным числовым величинам `a` и `b`, не обязательно имеющим один тип.

Таблица 3.3.

Оператор	Название	Пример
==	равно	a==b
!=	не равно	a!=b
>	больше	a>b
<	меньше	a<b
>=	больше или равно	a>=b
<=	меньше или равно	a<=b

`instanceof` Принадлежность объекта классу `obj instanceof MyClass`

Про оператор `instanceof` будет рассказано в разделе, посвященном динамической проверке типов.

3.2. Целые типы, переменные, константы

Таблица 3.4.

Тип	Число байт	Диапазон значений	Описание
`byte`	1	$-128..127$	Однобайтовое целое число (8-битное целое со знаком)
`short`	2	$-2^{15}..2^{15}-1 = -32768..32767$	Короткое целое число (16- битное целое со знаком)
`char`	2	$\backslash u0000 \ldots \backslash uFFFF = 0 \ldots 65535$	Символьный тип (беззнаковое 16- битное целое)
`int`	4	$-2^{31}..2^{31}-1 = -2.147483648 * 10^9..2.147483647 * 10^9$	Целое число (32- битное целое со знаком)
`long`	8	$-2^{63}..2^{63}-1 = -9.22337203685478\Delta 10^{18}..9.22337203685478\Delta 10^{18}$	Длинное целое число (64- битное целое со знаком)

Для задания в тексте программы численных литерных констант типа `long`, выходящих за пределы диапазона чисел типа `int`, после написания числа следует ставить постфикс – букву `L`. Например, `6000000000000000L`. Можно ставить и строчную `l`, но ее хуже видно, особенно – на распечатках программы (можно перепутать с единицей). В остальных случаях для всех *целочисленных типов значение* указывается в обычном виде, и оно считается имеющим тип `int` – но при присваивании число типа `int` автоматически преобразуется в *значение* типа `long`.

Как уже говорилось, объявление переменных может осуществляться либо в классе, либо в методе.

В классе их можно объявлять как без явного присваивания начального значения, так и с указанием этого значения. Если начальное *значение* не указано, величина инициализируется нулевым значением. Если объявляется несколько переменных одного типа, после указания имени типа разрешается перечислять несколько переменных, в том числе – с присваиванием им начальных значений.

В методе все переменные перед использованием обязательно надо инициализировать – автоматической инициализации для них не происходит. Это надо делать либо при объявлении, либо до попытки использования значения переменной в подпрограмме. Попытка использования в подпрограмме значения неинициализированной переменной приводит к ошибке во *время компиляции*.

Рассмотрим примеры задания переменных в классе.

```
int i,j,k;
int j1;
byte i1,i2=-5;
short i3=-15600;
long m1=1,m2,m3=-100;
```

Заметим, что после указанных объявлений переменные `i,j,k,j1,i1,m2` имеют *значение* 0. Для `i1` это неочевидно – можно подумать, что обе переменные инициализируются значением -5. Поэтому лучше писать так:

```
byte i1=0,i2=-5;
```

Использование в выражениях:

```
i=5;
j=i*i + 1
m1=j
m2=255;
m1=m1 + m2*2;
```

Тип `char` в *Java*, как и в C/C++, является **числовым**, хотя и предназначен для хранения отдельных символов. Переменной *символьного типа* можно присваивать один символ, заключенный в одинарные кавычки, либо кодирующую символ *управляющую последовательность* Unicode. Либо можно присваивать числовой *код символа* Unicode (номер символа в кодовой таблице):

```
char c1='a';
char c2='\u0061';
char c3=97;
```

Все три приведенные декларации переменных присваивают переменным десятичное *значение* `97`, соответствующее латинской букве "a".

Существуют различные кодовые таблицы для сопоставления числам символов, которые могут быть отображены на экране или другом устройстве. В силу исторических причин самой распространенной является кодовая *таблица ASCII* для символов латиницы, цифр и стандартных специальных символов. Поэтому в таблице Unicode им были даны те же номера, и для них коды Unicode и *ASCII* совпадают:

'A' имеет код `65`, 'B' - `66`, 'Z' - `90`, 'a' - `97`, 'z' - `122`, '0' - `48`, '1' - `49`, '9' - `57`, ':' - `58`, ';' - `59`, '<' - `60`, '=' - `61`, '>' - `62`, и так далее.

К сожалению, с переменными и значениями *символьного типа* можно совершать все действия, которые разрешено совершать с целыми числами. Поэтому символьные значения можно складывать и вычитать, умножать и делить не только на "обычные" целые величины, но и друг на друга! То есть *присваивание* `c1='a'*'a'+1000/'b'` вполне допустимо, несмотря на явную логическую абсурдность.

Константами называются именованные ячейки памяти с неизменяемым содержимым. Объявление констант осуществляется в классе, при этом перед именем типа *константы* ставится комбинация зарезервированных слов `public` и `final`:

```
public final int MAX1=255;
public final int MILLENIUM=1000;
```

Константами можно пользоваться как переменными, доступными только по чтению. Попытка присвоить константе *значение* с помощью оператора присваивания `"="` вызывает ошибку компиляции.

Для того чтобы имена констант были хорошо видны в тексте программы, их обычно пишут в верхнем регистре (заглавными буквами).

Имеется следующее правило хорошего тона: никогда не используйте одно и то же числовое литерное *значение* в разных местах программы, вместо этого следует задать константу и использовать ее имя в этих местах. Например, мы пишем программу обработки текста, в которой во многих местах используется литерная константа `26` - число букв в английском алфавите. Если у нас возникнет задача модифицировать ее для работы с алфавитом, в котором другое число букв (например, с русским), придется вносить исправления в большое число мест программы. При этом нет никакой гарантии, что не будет забыто какое-либо из необходимых исправлений, или случайно не будет "исправлено" на новое *значение* литерное *выражение* `26`, не имеющее никакого отношения к числу букв алфавита. Оптимистам, считающим, что проблема решается поиском числа 26 по файлам проекта (в некоторых средах разработки такое возможно), приведу пример, когда это не поможет: символ подчеркивания `"_"` иногда (но не всегда) считается буквой. Поэтому в некоторых местах программы будет фигурировать число 27, а в некоторых 26. И исправления надо будет вносить как в одни, так и в другие места. Поэтому работа перестает быть чисто технической - при изменениях требуется вникать в смысл участка программы, и всегда есть шанс ошибиться.

Использование *именованных констант* полностью решают эту проблему. Если мы введем константу `CHARS_COUNT=26`, то вместо 27 будет записано `CHARS_COUNT +1`, и изменение значения `CHARS_COUNT` повлияет правильным образом и на это *место* программы. То есть достаточно внести одно изменение, и гарантированно получить правильный результат для всех мест программы, где необходимы исправления.

3.3. Основные операторы для работы с целочисленными величинами

Все перечисленные ниже *операторы* действуют на две целочисленные величины, которые называются операндами. В результате действия оператора возвращается целочисленный результат. Во всех перечисленных далее примерах `i` и `j` обозначают целочисленные выражения, а `v` - целочисленную переменную.

Таблица 3.5.

Оператор	Название	Пример	Примечание
`+`	Оператор сложения	`i+j`	В случае, когда операнды `i` и `j` имеют разные типы или типы `byte`, `short` или `char`, действуют правила *автоматического преобразования* типов.
`-`	Оператор вычитания	`i-j`	
`*`	Оператор	`i*j`	

	умножения		
/	Оператор деления	i/j	Результат округляется до целого путем отбрасывания дробной части как для положительных, так и для отрицательных чисел.
%	Оператор остатка от целочисленного деления	i%j	Возвращается остаток от целочисленного деления
=	Оператор присваивания	v=i	Сначала вычисляется выражение `i`, после чего полученный результат копируется в ячейку `v`
++	Оператор *инкремента* (увеличения на 1)	v++	`v++` эквивалентно `v=v+1`
--	Оператор *декремента* (уменьшения на 1)	v--	`v--` эквивалентно `v=v-1`
+=		v+=i	`v+=i` эквивалентно `v=v+i`
-=		v-=i	`v-=i` эквивалентно `v=v-i`
*=		v*=i	`v*=i` эквивалентно `v=v*i`
/=		v/=i	`v/=i` эквивалентно `v=v/i`
%=		v%=i	`v%=i` эквивалентно `v=v%i`

Также имеются важные методы классов `Integer` и `Long`, обеспечивающие преобразование строкового представления числа в целое *значение*:

```
Integer.parseInt( строка )
Long.parseLong( строка )
```

Например, если в *экранной форме* имеются текстовые пункты ввода `jTextField1` и `jTextField2`, преобразование введенного в них текста в числа может проводиться таким образом:

```
int n=Integer.parseInt(jTextField1.getText());
long n1=Long.parseLong(jTextField2.getText());
```

Функции `Integer.signum ( число )` и `Long.signum ( число )` возвращают знак числа – то есть 1 если число положительно, 0, если оно равно 0, и -1 если число отрицательно. Кроме того, в классах `Integer` и `Long` имеется ряд операторов для работы с числами на уровне их битового представления.

Классы `Integer` и `Long` являются так называемыми оболочечными классами (*wrappers*), о них речь пойдет чуть дальше.

3.4. Вещественные типы и класс Math

Таблица 3.6.

Тип	Размер	Диапазон	Описание
float	4 байта (32 бит)	$1.5 * 10^{-45}..3.4 * 10^{38}$	"Одинарная" точность, 7-8 значащих десятичных цифр *мантиссы*. Тип `real*4` стандарта IEEE754
double	8 байт (64 бит)	$5 * 10^{-324}..1.7 * 10^{308}$	"Двойная" точность, 15..16 значащих цифр *мантиссы*. Тип `real*8` стандарта IEEE754

Таблица 3.7.

Оператор	Название	Пример	Примечание
+	Оператор сложения	x+y	В случае, когда операнды `x` и `y` имеют разные типы, действуют правила *автоматического преобразования* типов.
-	Оператор вычитания	x-y	
*		x*y	

	Оператор умножения		
/	Оператор деления	x/y	Результат является вещественным. В случае, когда операнды x и y имеют разные типы, действуют правила *автоматического преобразования* типов.
%	Оператор остатка от целочисленного деления	x%y	Возвращается остаток от целочисленного деления x на y. В случае, когда операнды x и y имеют разные типы, действуют правила *автоматического преобразования* типов.

Таблица 3.8.

Оператор	Название	Пример	Примечание
=	Оператор присваивания	v=x	Сначала вычисляется выражение x, после чего полученный результат копируется в ячейку v
++	Оператор *инкремента* (увеличения на 1)	v++ ++v	эквивалентно v=v+1
--	Оператор *декремента* (уменьшения на 1)	v-- --v	эквивалентно v=v-1
+=		v+=x	эквивалентно v=v+x
-=		v-=x	эквивалентно v=v-x
*=		v*=x	эквивалентно v=v*x
/=		v/=x	эквивалентно v=v/x
%=		v%=x	эквивалентно v=v%x

Математические функции, а также *константы* "пи" (*Math*.PI) и "e" (*Math*.E) заданы в классе Math, находящемся в пакете java.lang.

Для того, чтобы их использовать, надо указывать *имя функции* или *константы*, квалифицированное впереди именем класса Math.

Например, возможны вызовы Math.PI или Math.sin(x). При этом *имя пакета* java.lang указывать не надо – он импортируется автоматически. К сожалению, использовать математические функции без квалификатора Math, не получается, так как это *методы класса*.

Константы в классе Math заданы так:

```
public static final double E = 2.7182818284590452354;
public static final double PI = 3.14159265358979323846;
```

Модификатор static означает, что это *переменная класса*; final означает, что в классе-наследнике переопределять это *значение* нельзя.

В приведенной ниже таблице величины x, y, angdeg, angrad имеют тип double, величина a – тип float, величины m, n – целые типов long или int. Математические функции возвращают значения типа double, если в примечании не указано иное.

Таблица 3.9.

Оператор класса Math	Примечание
Тригонометрические и обратные тригонометрические функции	
sin(x)	sin(x) – синус
cos(x)	cos(x) – косинус
tan(x)	tg(x) – тангенс
asin(x)	arcsin(x) – арксинус
acos(x)	arccos(x) – арккосинус
atan(x)	arctg(x) – арктангенс
atan2(y, x)	Возвращает угол, соответствующий точке с координатами x,y, лежащий в пределах $(-\pi, \pi]$.
toRadians(angdeg)	angdeg / 180.0 * PI; – перевод углов из градусов в

радианы.

`toDegrees(angrad)` angrad * 180.0 / PI; - перевод углов из радиан в градусы.

Степени, *экспоненты*, логарифмы

`exp(x)` e^x - *экспонента*

`expm1(x)` $e^x - 1$. При x, близком к 0, дает гораздо более точные значения, чем $exp(x) - 1$

`log(x)` ln(x) - *натуральный логарифм*.

`log10(x)` $\log_{10}(x)$ - десятичный логарифм.

`log1p(x)` $\ln(1 + x)$. При x, близком к 0, дает гораздо более точные значения, чем $\log(1 + x)$

`sqrt(x)` $\sqrt{}$ - квадратный корень

`cbrt(x)` $\sqrt[3]{}$ - кубический корень

`hypot(x,y)` $\sqrt{x^2 + y^2}$ - вычисление длины гипотенузы по двум катетам

`pow(x, y)` x^y - возведение x в степень y

`sinh(x)` $\mathrm{sh}(x) = \frac{e^x - e^{-x}}{2}$ - гиперболический синус

`cosh(x)` $\mathrm{ch}(x) = \frac{e^x + e^{-x}}{2}$ - гиперболический косинус

`tanh(x)` $\mathrm{th}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ - гиперболический тангенс

Модуль, знак, минимальное, максимальное число

`abs(m)` `abs(x)` Абсолютное значение числа. Аргумент типа `int`, `long`, `float` или `double`. Результат того же типа, что аргумент.

`signum(a)` `signum(x)` Знак числа. Аргумент типа `float` или `double`. Результат того же типа, что аргумент.

`min(m,n)` `min(x,y)` Минимальное из двух чисел. Аргументы одного типа. Возможны типы: `int`, `long`, `float`, `double`. Результат того же типа, что аргумент.

`max(m,n)` `max(x,y)` Максимальное из двух чисел. Аргументы одного типа. Возможны типы: `int`, `long`, `float`, `double`. Результат того же типа, что аргумент.

Округления

`ceil(x)` Ближайшее к **x** целое, большее или равное **x**

`floor(x)` Ближайшее к **x** целое, меньшее или равное **x**

`round(a)` `round(x)` Ближайшее к **x** целое. Аргумент типа `float` или `double`. Результат типа `long`, если аргумент `double`, и типа `int` - если `float`. То же, что `(int)floor(x + 0.5)`.

`rint(x)` Ближайшее к **x** целое.

`ulp(a)` `ulp(x)` Расстояние до ближайшего большего чем аргумент значения того же типа ("дискретность" изменения чисел в формате с плавающей точкой вблизи данного значения). Аргумент типа `float` или `double`. Результат того же типа, что аргумент.

Случайное число, остаток

`random()` *Псевдослучайное число* в диапазоне от 0.0 до 1.0. При этом $0 \leq \mathrm{Math.random}() < 1$

`IEEEremainder(x,y)` Остаток от целочисленного деления **x/y**, то есть **x-y*n**, где **n** - результат целочисленного деления

Также имеются методы оболочечных классов `Float` и `Double`, обеспечивающие преобразование строкового представления числа в *значение* типа `Float` или `Double`:

```
Float.parseFloat ( строка )
Double.parseDouble ( строка )
```

Например, если в *экранной форме* имеются текстовые пункты ввода `jTextField1` и `jTextField2`, преобразование введенного в них текста в числа может проводиться таким образом:

```
float f1= Float.parseFloat(jTextField1.getText()) ;
double d1= Double.parseDouble(jTextField2.getText()) ;
```

Иногда вместо класса `Math` ошибочно пытаются использовать пакет `java.math`, в котором содержатся классы `BigDecimal, BigInteger, MathContext, RoundingMode`. *Класс* `BigDecimal` обеспечивает работу с десятичными числами с произвольным числом значащих цифр – в том числе с произвольным числом цифр после десятичной точки. *Класс* `BigInteger` обеспечивает работу с целыми числами произвольной длины. Классы `MathContext` и `RoundingMode` являются вспомогательными, обеспечивающими настройки для работы с некоторыми методами классов `BigDecimal` и `BigInteger`.

3.5. Правила явного и автоматического преобразования типа при работе с числовыми величинами

Компьютер может проводить на аппаратном уровне целочисленные математические вычисления только с величинами типа `int` или `long`. При этом операнды в основных математических операциях (`+` , `-` , `*` , `/` , `%`) должны быть одного типа. Поэтому в тех случаях, когда операнды имеют разные типы, либо же типы `byte` , `short` или `char` , действуют правила *автоматического преобразования* типов: для величин типа `byte` , `short` или `char` сначала происходит преобразование в тип `int` , после чего производится их *подстановка* в качестве операндов. Если же один из операндов имеет тип `long` , действия производятся с числами типа `long` , поскольку второй *операнд* автоматически преобразуется к этому типу.

Аналогично, при работе с вещественными величинами в *Java* возможна работа на аппаратном уровне только с операндами типов `float` и `double`. При этом операнды в основных математических операциях должны быть одного типа. Если один из операндов имеет тип `double` , а другой `float` , действия производятся с числами типа `double` , поскольку *операнд* типа `float` автоматически преобразуется к типу `double`.

Если один из операндов целочисленный, а другой вещественный, сначала идет преобразование целочисленного операнда к такому же вещественному типу, а потом выполняется оператор.

Рассмотрим теперь правила *совместимости типов* по присваиванию. Они просты: *диапазон* значений типа левой части не должен быть уже, чем *диапазон* типа правой. Поэтому в присваиваниях, где тип в правой части не умещается в *диапазон* левой части, требуется указывать *явное преобразование типа*. Иначе *компилятор* выдаст *сообщение об ошибке* с не очень адекватной диагностикой `"possible loss of precision"` ("возможная *потеря точности*").

Для *явного преобразования* типа в круглых скобках перед преобразуемой величиной ставят имя того типа, к которому надо преобразовать. Например, пусть имеется

```
double d=1.5;
```

Величину `d` можно преобразовать к типу `float` таким образом:

```
(float)d
```

Аналогично, если имеется величина `f` типа `float` , ее можно преобразовать в величину типа `double` таким образом:

```
(double)f
```

Во многих случаях к явному преобразованию типов прибегать не нужно, так как действует *автоматическое преобразование*, если переменной, имеющий тип с более широким диапазоном изменения, присваивается *выражение*, имеющее тип с более узким диапазоном изменения.

Например, для вещественных типов разрешено *присваивание* только в том случае, когда слева стоит вещественная *переменная* более широкого диапазона, чем целочисленная или вещественная – справа. Например, переменной типа `double` можно присвоить *значение* типа `float` , но не наоборот. Но можно использовать *преобразование типов* для того, чтобы выполнить такое *присваивание*. Например:

```
double d=1.5;
float f=(float)d;
```

Другие примеры допустимых присваиваний:

Таблица 3.10.

byte	`byte0=1; //-128..127`
short	`short0=1;//- 32768.. 32767`
char	`char0=1;//0.. 65535`
int	`int0=1; //- 2.147483648E9.. 2.147483647E9`
long	`long0=1;//-9.223E18.. 9.223E18`

float float0=1;// ±(1.4E-45..3.402E38)

double double0=1;// ±(4.9E-324..1.797E308)

Таблица 3.11.

short0=byte0;

byte0=(**byte**)short0;

char0=(**char**)short0;

int0=short0;

int0=char0;

char0=(**char**)int0;

short0=(**short**)int0;

long0=byte0;

byte0=(**byte**)long0;

long0=char0;

long0=int0;

int0=(**int**)long0;

float0=byte0;

float0=int0;

float0=(**float**)double0;

double0=float0;

3.6. Оболочечные классы. Упаковка (boxing) и распаковка (unboxing)

В ряде случаев вместо значения *примитивного типа* требуется *объект*. Например, для работы со списками объектов. Это связано с тем, что работа с объектами в *Java* может быть унифицирована, поскольку все классы *Java* являются наследниками класса `Object` , а для примитивных типов этого сделать нельзя.

Для таких целей в *Java* каждому примитивному типу сопоставляется *объектный тип*, то есть *класс*. Такие классы называются оболочечными (`class wrappers`). В общем случае они имеют те же имена, что и *примитивные типы*, но начинающиеся не со строчной, а с заглавной буквы. *Исключение* почему-то составляют типы `int` и `char` , для которых имена оболочечных классов `Integer` и `Character` .

Таблица 3.12.

Примитивный тип	Оболочечный класс
byte	Byte
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double

Внимание! *Класс* `Character` несколько отличается от остальных оболочечных числовых классов потому, что тип `char` "не вполне числовой".

Основное назначение оболочечных классов – создание объектов, являющихся оболочками над значениями примитивных типов. Процесс создания такого объекта ("коробки" – *box*) из значения *примитивного типа* называется упаковкой (`boxing`), а *обратное преобразование* из объекта в величину *примитивного типа* – распаковкой (`unboxing`). Оболочечные объекты ("обертки" для значения *примитивного типа*) хранят это *значение* в *поле* соответствующего *примитивного типа*, доступном по чтению с помощью функции `имяТипаValue()` . Например, метода `byteValue()` для объекта типа `Byte` . Но во многих случаях можно вообще не обращать внимания на отличие переменных с типом оболочечных классов от переменных примитивных типов, так как упаковка и распаковка при подстановке такого объекта в *выражение* происходит автоматически, и *объект* оболочечного типа в этих случаях внешне ведет себя как число. Таким образом, если нам необходимо хранить в объекте числовое *значение*, следует создать *объект* соответствующего оболочечного типа.

Например, возможны такие фрагменты кода:

```
Integer obj1=10;
int i1= obj1*2;
Byte b=1;
obj1=i1/10;
b=2;
```

Но не следует забывать, что при операциях упаковки-распаковки происходят внешне невидимые дополнительные *операции* копирования значений в промежуточные буферные ячейки, поэтому соответствующие вычисления несколько медленнее операций с примитивными типами и требуют несколько больше памяти.

Поэтому в критических к быстродействию и занимаемым ресурсам местам программы их использование нежелательно. С другой стороны, автоматическая упаковка-распаковка (она появилась в пятой версии *JDK*) во многих случаях заметно упрощает *программирование* и делает текст программы более читаемым. Так что для участков программы, некритичных к быстродействию, ею вполне можно пользоваться.

Помимо создания объектов оболочечные классы имеют ряд других полезных применений. Например, в числовых оболочечных классах хранятся *константы*, с помощью которых можно получить максимальные и минимальные значения:

`Byte.MIN_VALUE` , `Byte.MAX_VALUE` , `Float.MIN_VALUE` , `Float.MAX_VALUE` , `Double.MIN_VALUE` , `Double.MAX_VALUE` и т.п.

В оболочечных классах также имеются методы классов, то есть такие, которые могут работать в отсутствии объекта соответствующего типа – для их вызова можно пользоваться именем типа. Например, как мы уже знаем, имеются методы

`Byte.parseByte` (*строка*)
`Short.parseShort` (*строка*)
`Integer.parseInt` (*строка*)
`Long.parseLong` (*строка*)
`Float.parseFloat` (*строка*)
`Double.parseDouble` (*строка*)

Они преобразуют строку в число соответствующего типа. Вызовы

`Byte.valueOf` (*строка*)
`Short.valueOf` (*строка*)
`Integer.valueOf` (*строка*)
`Long.valueOf` (*строка*)
`Float.valueOf` (*строка*)
`Double.valueOf` (*строка*)

аналогичны им, но возвращают не числовые значения, а объекты соответствующих оболочечных типов.

Примеры использования оболочечных классов:

```
int n1=Integer.MAX_VALUE;
double d1= Double.MIN_VALUE;
```

Отметим, что *присваивание*

```
double d2= Double.parseDouble(jTextField1.getText());
```

будет работать совершенно так же, как

```
double d2= Double.valueOf(jTextField1.getText());
```

несмотря на то, что во втором случае методом `valueOf` создается *объект* оболочечного типа `Double` . Поскольку в левой части присваивания стоит *переменная* типа `double` , происходит автоматическая распаковка, и переменной `d2` присваивается распакованное *значение*. Сам *объект* при этом становится мусором – программная *связь* с ним теряется, и он через некоторое время удаляется из памяти системой *сборки мусора*. В данном случае ни *быстродействие*, ни объем памяти некритичны, поскольку *операции* взаимодействия с пользователем по компьютерным меркам очень медленные, а один *объект* оболочечного типа занимает пренебрежимо мало места в памяти (около сотни *байт*). Так что с потерями ресурсов в этом случае можно не считаться, обращая внимание только на читаемость текста программы. Поэтому *автор* предпочитает второй вариант присваивания: хотя он и "неоптимальный" по затрате ресурсов, но более читаем.

3.7. Приоритет операторов

При вычислении выражений важен приоритет операторов. Для операторов сложения, вычитания, умножения и деления он "естественный": *умножение* и *деление* обладают одинаковым наиболее высоким приоритетом, а *сложение* и *вычитание* – одинаковым приоритетом, который ниже. Таким образом, например,

```
a*b/c+d
```

это то же, что

```
( (a*b)/c )+d
```

Круглые скобки позволяют группировать элементы выражений, при этом *выражение* в скобках вычисляется до того, как участвует в вычислении остальной части выражения. То есть скобки обеспечивают больший приоритет, чем все остальные *операторы*. Поэтому `(a+b)*c` будет вычисляться так: сначала вычислится сумма `a+b` , после чего полученный результат будет умножен на *значение* `c` .

Кроме перечисленных в *Java* имеется большое количество других правил, определяющих приоритеты различных операторов. *Автор* считает их изучение не только нецелесообразным, но даже вредным: программу следует писать так, чтобы все последовательности действий были очевидны и не могли вызвать сложностей в *понимании текста* программы и привести к *логической ошибке*. Поэтому следует расставлять скобки даже в тех случаях, когда они теоретически не нужны, но делают очевидной последовательность действий. Отметим, такие действия часто помогают заодно решить гораздо более сложные проблемы, связанные с арифметическим переполнением.

Далее в справочных целях приведена *таблица приоритета* операторов. Ей имеет смысл пользоваться в случае анализа плохо написанных программ, когда из текста программы неясна *последовательность*

операторов.

Таблица 3.13.

Приоритет	Группа операторов	Операторы		
1 *высший*	Постфиксные	( )	[ ]	.
2	Унарные	++ операнд операнд ++	--операнд операнд--	~ ! + операнд - операнд
3	Создания объектов и преобразования типа	new	(тип) операнд	
4	Мультипликативные	*	/	%
5	Аддитивные	+	-	
6	Сдвиги битов	>>	>>>	<<
7	Отношения	>	>=	< <= instanceof
8	Эквивалентности	==	!=	
9	Побитовое И	&		
10	Побитовое исключающее ИЛИ	^		
11	Побитовое ИЛИ	|		
12	Логическое И	&&		
13	*Логическое ИЛИ*	||		
14	Условный	? :		
15 *низший*	Присваивания	=	*Оператор* = (+=, -=, *=, /= и т.п.)	

Краткие итоги

Величины типа `boolean` принимают значения `true` или `false`.

Логические операторы `&& -"И", || - "ИЛИ", ^ - "Исключающее ИЛИ", ! - "НЕ"` применимы к величинам булевского типа. Логические выражения в Java вычисляются в соответствии с укороченным *оцениванием*.

Операторы сравнения применимы к любым величинам `a` и `b` одного типа, а также к произвольным числовым величинам `a` и `b`, не обязательно имеющим один тип. В качестве оператора сравнения на равенство используется составной символ, состоящий из двух подряд идущих символов равенства `"=="`.

В Java имеются встроенные примитивные целые типы `byte, short, int, long` и символьный тип `char`, в некотором смысле также являющийся целочисленным. При этом только тип `char` беззнаковый, все остальные – знаковые.

Для задания в тексте программы численных литерных констант типа `long`, выходящих за пределы диапазона чисел типа `int`, после написания числа следует ставить постфикс – букву L.

Константами называются именованные ячейки памяти с неизменяемым содержимым. Объявление констант осуществляется в классе, при этом перед именем типа константы ставится комбинация зарезервированных слов `public` и `final`.

В Java имеется два встроенных примитивных вещественных типа `float` и `double` (точнее, типы чисел в формате с плавающей точкой).

Математические функции, а также константы "пи" (`Math.PI`) и "е" (`Math.E`) заданы в классе `Math`, находящемся в пакете `java.lang.`

Целочисленные математические вычисления проводятся на аппаратном уровне только с величинами типа `int` или `long`. Для величин типа `byte`, `short` или `char` сначала происходит преобразование в тип `int`, после чего производится их подстановка в качестве операндов. Если же один из операндов имеет тип `long`, действия производятся с числами типа `long`, поскольку второй операнд автоматически преобразуется к этому типу.

При работе с вещественными величинами в Java возможна работа на аппаратном уровне только с операндами типов `float` и `double`. Если один из операндов имеет тип `double`, а другой `float`, действия производятся с числами типа `double`, поскольку операнд типа `float` автоматически преобразуется к типу `double`.

Если один из операндов целочисленный, а другой вещественный, сначала идет преобразование целочисленного операнда к такому же вещественному типу, а потом выполняется оператор.

В Java каждому примитивному типу сопоставляется *объектный тип*, то есть класс. Такие классы называются оболочечными (`class wrappers`). В общем случае они имеют те же имена, что и *примитивные типы*, но начинающиеся не со строчной, а с заглавной буквы. Исключение составляют типы `int` и `char`, для которых имена оболочечных классов `Integer` и `Character`.

Основное назначение оболочечных классов – создание объектов, являющихся оболочками над значениями примитивных типов. Процесс создание такого объекта ("коробки" – box) из значения *примитивного типа* называется *упаковкой* (boxing), а *обратное преобразование* из объекта в величину *примитивного типа* – *распаковкой* (unboxing). Упаковка и распаковка для числовых классов осуществляется автоматически.

В оболочечных классах имеется ряд полезных методов и констант. Например, минимальное по модулю не равное нулю и максимальное значение числового типа можно получить с помощью констант, вызываемых через имя оболочечного типа: Integer.MIN_VALUE, Integer.MAX_VALUE, Float.MIN_VALUE , Float.MAX_VALUE, Double.MIN_VALUE , Double.MAX_VALUE. и т.п.

В Java имеется 15 уровней приоритета операторов. В хорошо написанной программе ставятся скобки, повышающие читаемость программы, даже если они не нужны с точки зрения таблицы приоритетов.

Иногда требуется использовать элементы, которые не являются ни числами, ни строками, но ведут себя как имена элементов и одновременно обладают порядковыми номерами. Например, названия месяцев или дней недели. В этих случаях используют *перечисления*.

Типичные ошибки:

Очень часто встречается ошибка, когда вместо сравнения численных значений вида a==b программист пишет a=b .Чаще всего такая ошибка возникает в условных операторах: вместо if(a==b){...} пишут if(a=b){...} . В Java такая ошибка диагностируется на этапе компиляции для всех типов, кроме булевского. Для булевских a и b вызов if(a=b){...} приведет к тому, что величине a будет присвоено значение b , и результат присваивания возвратится равным этому значению. А диагностики ошибки, к сожалению, выдано не будет.

Начинающие программисты, изучавшие ранее язык BASIC, пытаются использовать выражение вида **a^b** для возведения в степень, то есть для вычисления выражения a^b . В Java для такой операции следует использовать выражение Math.pow(a,b) .

Программисты, изучавшие ранее C/C++, пытаются использовать логические выражения вида **a&b** или **a|b** для полной (не укороченной) оценки логических выражений. В Java нет полной (не укороченной) оценки логических выражений, а операторы & и | зарезервированы для арифметических побитовых операций "И" и "ИЛИ".

При использовании оператора instanceof пытаются написать instanceOf.

Путают класс java.lang.Math и пакет java.math.

Очень часто встречается ошибка, когда забывают, что для величин типа byte , short или char сначала происходит преобразование в тип int , и только после этого производится их подстановка в качестве операндов. Поэтому, в частности, побитовые операции с величинами этих типов дают те же результаты, что и при работе с величинами типа int .

Не ставятся скобки, группирующие операнды в длинных выражениях, где стороннему программисту неочевидна последовательность выполнения операндов.

Элементы перечисления пытаются задавать как строковые, или же пытаются предварительно задать числовые переменные с именами элементов перечисления.

Задания

На основе проекта с графическим пользовательским интерфейсом создать новый проект. В нем для каждого из целочисленных и вещественных типов задать переменные и кнопки с соответствующими надписями. При нажатии на кнопки должны показываться диалоговые панели с сообщением об имени и значении соответствующей переменной.

На основе проекта с графическим пользовательским интерфейсом создать новый проект. В нем сделать два пункта ввода, метку и кнопки **"Сложить"**, **"Умножить"**, **"Разделить"**, **"sin"**. По нажатию на кнопки **"Сложить"**, **"Умножить"**, **"Разделить"** в метку должен выводиться результат. Действия проводить с величинами типа double . По нажатию на кнопку **"sin"** в метку должен выводиться синус значения, показывавшегося до того в метке.

На основе проекта с графическим пользовательским интерфейсом создать новый проект. В нем сделать пункт ввода и радиогруппу с выбором варианта для каждого из *целочисленных типов*, а также кнопку **JButton** с надписью **"Преобразовать в число"**. При выборе соответствующего варианта в пункте ввода должно возникать случайным образом генерируемое число, лежащее в пределах допустимых значений для этого типа. При нажатии на кнопку содержимое пункта ввода должно быть преобразовано в число соответствующего типа, и это значение должно быть показано с помощью диалоговой панели с сообщением.

Работа с выбором вариантов осуществляется следующим образом:

```
if(jRadioButton1.isSelected() )
  оператор1;


if(jRadioButton2.isSelected())
  оператор2;


if(jRadioButton3.isSelected())
  оператор3;
```

Создать приложение Java с графическим пользовательским интерфейсом. В нем должно быть перечисление Spring ("весна"), в котором расположены весенние месяцы, и кнопки **"m2=m1"** и **"Вывести**

значение m2". Задать две переменные типа Spring - m1 и m2 . Переменную m1 инициализированной, со значением April , переменную m2 - не инициализированной. При нажатии на кнопку **" m2=m1"** переменной m2 должно присваиваться значение m1 . При нажатии на кнопку **"Вывести значение m2"** должно выводиться значение переменной m2 .

Внимание! Если Вы увидите ошибку на нашем сайте, выделите её и нажмите Ctrl+Enter.