Лекция №1

«Data types, variables, operators»

Автор: доц. каф. ПИ ХНУРЭ Колесников Д.О. v0.2

1.1. Ключевые слова

В языке Java зарезервированы ключевые слова, которые нельзя использовать в качестве идентификаторов (имен переменных, классов или методов.

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

Ключевые слова **const** и **goto** зарезервированы, но не используются. При попытке их использовать произойдет ошибка на этапе компиляции.

Замечание. Слово **main** не относится к ключевым.

Замечание. К ключевым словам не относятся литералы: **true**, **false** и **null**, которые также нельзя использовать в качестве идентификаторов.

1.2. Пробелы

К пробельным символам в Java относятся все символы, которые отделяют синтаксические единицы языка (лексемы) друг от друга.

К пробельным символам в Java относятся символы с десятичными кодами:

32 (SP - пробел);
9 (HT - горизонтальная табуляция);
12 (FF - перевод страницы);
10 (LF - новая строка);
13 (CR - возврат каретки);
упорядоченное сочетание CRLF.

Переводом строки в Java является любой из трех пробельных символов: CR, LF, CRLF

<u>Замечание</u>. Форматирование, сделанное с помощью пробельных символов, никак не влияет на работу компилятора. Следующие три примера кода эквивалентны относительно результата компиляции.

```
int x = 7;

int
x
=
7;

int          x          =          7;
```

<u>Замечание</u>. Многострочный комментарий также может выполнять функции пробельного символа. Следующий пример эквивалентен трем предыдущим.

```
int/**/x = 7;
```

1.3. Идентификаторы

Идентификаторы используются в качестве имен классов, методов и переменных.

Идентификатор может быть любой последовательностью букв нижнего и верхнего регистров национальных алфавитов, цифр, символа подчеркивания _ и знака $. Он не должен начинаться с цифры.

<u>Примеры правильных идентификаторов:</u>

count _x $test y5

<u>Примеры неправильных идентификаторов:</u>

#count 2x .test test-4

Замечание. При составлении идентификаторов рекомендуется использовать буквы только из латинского алфавита. Не рекомендуется для этих целей использовать знак $.

Замечание. Идентификаторы могут состоять из символов, которые не относятся к национальным алфавитам. Класс Character содержит методы

public static boolean **isJavaIdentifierPart**(int codePoint)
public static boolean **isJavaIdentifierStart** (int codePoint)

которые позволяют определить может ли быть использован символ с заданным кодом в качестве составной части идентификатора.

1.4. Типы данных

Все типы данных Java делятся на две категории - примитивные и ссылочные:

Примитивные типы

- числа: char, byte, int, short, long, float, double
- логический: boolean

Ссылочные типы

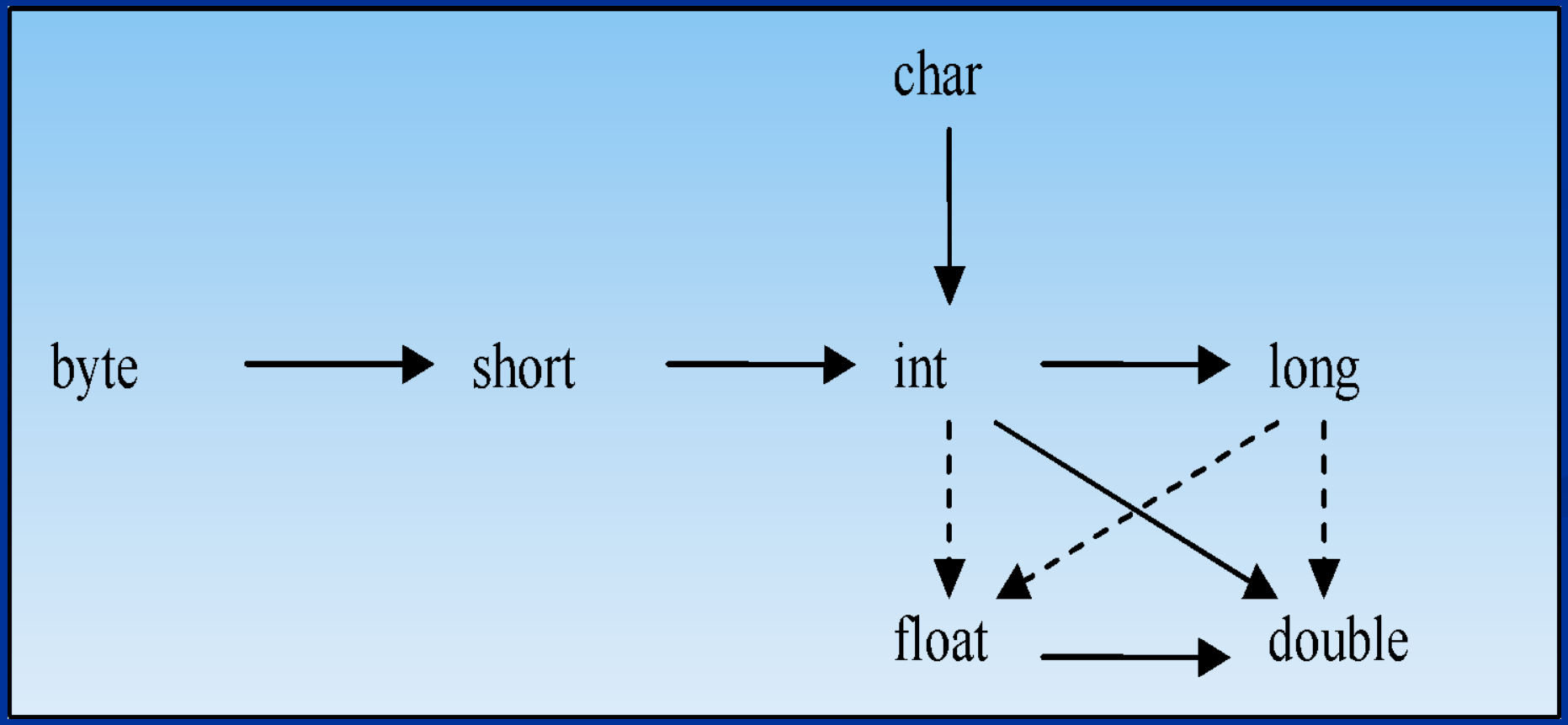
- классы
- интерфейсы
- массивы
- null тип

Существует 8 примитивных (простых) типов данных.

область значения	обозначение	требуемый объем памяти, байт	диапазон	структура в битах	константы
целые числа	char	2	$[0, 2^{16}]$	-	'a', '\u00ff'
	byte	1	[-128, 127]	1-й бит указывает знак	
	short	2	$[-2^{15}, 2^{15}-1]$		
	int	4	$[-2^{31}, 2^{31}-1]$		
	long	8	$[-2^{63}, 2^{63}-1]$		3L, 4l
вещественные числа	float	4	$\sim[-3.4 \cdot 10^{38}; -3.4 \cdot 10^{38}]$ Infinity, NAN	знак: 1 порядок: 8 мантисса: 23	3.4F, .4f
	double	8	$\sim[-1.79 \cdot 10^{308}; -1.79 \cdot 10^{308}]$ Infinity, NAN	знак: 1 порядок: 11 мантисса: 52	3.4, 0.4d, 3.4D
логические константы	boolean	1 бит	{false, true}	-	true, false

1.5. Автоматическое преобразование примитивных типов

Между примитивными типами Java разрешены следующие преобразования, которые выполняются автоматически.



Штриховой линией отмечены разрешенные автоматические преобразования, которые при определенных значениях переменной могут выполняться *с потерей* информации.

При преобразованиях, отмеченных сплошными стрелками потери информации не происходит.

Замечание. Тип int может содержать максимальное по величине число, которое может быть записано в двоичной форме с помощью 31-го бита.

Тип float для записи последовательности цифр использует мантиссу емкостью в 23 бита, а double – 52 бита.

Поэтому, любое число типа int может поместиться в мантиссу переменной типа double, но существуют такие числа типа int, которые не поместятся в мантиссу переменной типа float.

Аналогично, большие числа типа long, для записи которых применяются 64 бита, могут не поместиться в мантиссу вещественных переменных float (23 бита) и double (52 бита).

1.6. Тип числовой константы по умолчанию

По умолчанию (без суффиксов):

целочисленные константы имеют тип **int**;

вещественные константы имеют тип **double**.

Следующее выражение вызовет ошибку компиляции.

float y = 34.4; // 34.5 имеет тип double

1.7. Преобразования
byte <- int
short <- int

Переменным типа byte и short может быть присвоено значение целочисленной константы (которая по умолчанию имеет тип int) если ее значение не выходит за пределы диапазона соответствующего типа (при этом выполнится автоматическое преобразование типа).

```
byte x = 120;
short y = -1234;
```

1.8. Системы счисления записи числовых констант

При записи целочисленных констант может быть использовано три системы счисления: *десятичная*, *шестнадцатеричная* и *восьмеричная*.

Шестнадцатеричные константы предваряются комбинацией 0x и состоят из последовательности шестнадцатеричных цифр (0123456789ABCDEF).

int x = 0x1F; // эквивалентно int x = 31;

int y = -0x12; // эквивалентно int y = -18;

Восьмеричные константы предваряются 0 (нулем) и состоят из последовательности восьмеричных цифр (01234567).

int x = 011; // эквивалентно int x = 9;

int y = -002; // эквивалентно int y = -2;

1.9. Преобразование char <- int

Переменной типа char можно присваивать значение целочисленной константы, если ее значение не выходит за пределы интервала [0, 2^{16}-1] (при этом переменна типа char будет содержать символ Unicode с кодом, который соответствует целочисленной константе).

```
char ch1 = 70; // символ F
char ch2 = 65000; // не заполненная область Unicode
```

Замечание. Целочисленная константа должна быть определена в правой части присваивания либо непосредственно в виде числа, либо через арифметическое выражение, имеющее целый тип и не содержащее переменные (даже целочисленные).

```
int x = 70;
char ch = 23*3; // ch = 'B'
ch = 70; // ch = 'F'

ch = x; // ошибка компиляции
ch = x/2; // ошибка компиляции
```

1.10. Тип арифметического выражения

Арифметическое выражение, содержащее переменные и константы имеет тип, который определяется исходя из типов входящих в это выражение компонент (переменных и констант).

типы данных переменных констант входящих в выражение	тип результата
byte, short, int, char в любом сочетании	int
+ long к предыдущему	long
+float к предыдущему в любом сочетании	float
+double к предыдущему в любом сочетании	double

1.11. Преобразование int <- char

Переменной типа int можно присвоить значение переменной типа char, при этом в переменную будет загружен код символа.

```
char ch = 'F';
int x = ch; // x = 70
```

Замечание. Несмотря на то, что целочисленный тип short и тип char имеют одинаковую физическую емкость в 2 байта, преобразование из char в short автоматически не происходит, т.е. при присваивании необходимо применить явное преобразование типов.

```
char ch = 'F';
short x = (short)ch; // x = 70
```

Замечание. Так как тип int может быть преобразован без явного указания оператора преобразования типа в типы long, float и double, то таким же образом значение переменной типа char можно присваивать переменным указанных типов.

1.12. Вещественные константы

Десятичная целочисленная константа вместе со сразу следующей за ней десятичной точкой является вещественной константой.

```
int x = 123.; // вызовет ошибку компиляции,
              // т.к. 123. имеет тип double
```

Точка в конце шестнадцатеричных целочисленных констант <u>не допускается</u>.

Восьмеричная целочисленная константа вместе со сразу за ней следующей десятичной точкой является вещественной константой, при этом восьмеричные цифры интерпретируются как десятичные.

int x = 011; // x = 9 в десятичной сист. счисл.
double y = 011.; // y = 11.0 в десятичной сис. счисл.

float z = 09.; // вызовет ошибку компиляции, т.к. **09.** –
// вещественная константа типа double

Замечание. Точка в конце десятичных и восьмеричных целочисленных констант эквивалентна символу «d» или «D», т.е. указывает на то, что число относится к типу **double**.

1.13. Запрет преобразований вида boolean <- числовой тип числовой тип <- boolean

Логический тип boolean не может быть преобразован к целочисленному, верно и обратное, целочисленный тип не может быть преобразован к логическому.

1.14. Escape последовательности

Для записи неотображаемых на экране символов Unicode используются escape последовательности вида \uXXXX, где XXXX – *четыре* шестнадцатеричных цифры, составляющие код символа (диапазон [0, 2^{16}-1]). Escape последовательность представляет собой *символ* и может быть использована, в том числе, и в коде программы.

Для того, чтобы загрузить в переменную char символ, которому соответствует определенная escape последовательность, при присваивании последнюю следует заключить в одинарные кавычки.

ch\u0061r ch = '\u0061'; // эквивалентно char ch = 'a';

1.15. Главная функция программы

Главная функция программы **main** <u>должна</u>:

1) иметь уровень доступа *public* (чтобы быть доступной для JVM);

2) быть принадлежностью главного класса программы, т.е. *static*;

3) не возвращать никакого значения, т.е. иметь тип *void;*

4) иметь в точности один параметр типа *массив строк*.

public static void main(String args[])

В единственный параметр функции **main** заносятся значения параметров командной строки (если они есть).

Первый элемент массива с номером 0 содержит значение первого параметра, а не имя главного класса программы (как в языке C)

Замечание. Если в программе (отдельном самостоятельном приложении) отсутствует функция main или же эта функция определена с нарушением вышеописанных требований к ней, программа откомпилируется, но при попытке ее выполнить JVM выбросит исключение.

java.lang.NoSuchMethodError: main

1.16. Логические операции по краткой схеме

Если первый операнд логической операции OR по краткой схеме «||» равен true (истина) то и результат равен true *без вычисления* выражения, которое идет вторым операндом операции.

При вычислении результата операции OR по полной схеме всегда вычисляются оба операнда.

```
boolean A = true, B;
B = A || (A = false); // B = true, A = true
B = A | (A = false); B = true, A = false
```

Если первый операнд логической операции AND по краткой схеме «&&» равен false (ложь) то и результат равен false *без вычисления* выражения, которое идет вторым операндом операции.

При вычислении результата операции AND по полной схеме всегда вычисляются оба операнда.

```
boolean A, B = false;
A = B && (B = true); // A = false, B = false
A = B & (B = true); // A = false, B = true
```

Замечание. Применение операций «||» и «&&» позволяет добиться экономии вычислительных ресурсов и дает возможность реализовать ветвление программы наподобие оператора if.

Операция && очень часто применяется с целью избежать выполнения выражения, составляющего второй операнд, в том случае, когда его выполнение может выбросить исключение при определенном условии; это условие учитывается в первом операнде.

```
int[] array = … ;
int k = … ;
if (k < array.length && array[k] == 0) {// do something}
```

1.17. Параметр оператора switch

В качестве параметра оператора ветвления switch могут быть значения только типа int и любых других, которые к нему могут быть преобразованы автоматически (т.е. byte, short и char), также допускается использование значений перечислимого типа enum.

Таким образом, множество допустимых типов параметра оператора switch исчерпывается множеством {int, char, byte, short, enum}.

Если параметр будет иметь любой другой тип (объектный, вещественный, логический), то такой код не откомпилируется.

1.18. Унарные операции «++» и «--»

Унарные операции инкремента и декремента имеют две формы, различающиеся по записи и действию: *префиксную* и *постфиксную*. Префиксная форма увеличивает(уменьшает) значение переменной на единицу *до того*, как это значение будет использовано в выражении, постфиксная – *после* использования значения переменной.

```
int x = 1, y = 1, z;
z = ++x; // x = 2, z = 2
z = y++; // y = 2, z = 1
```

Замечание. Операции «++» и «--» могут применяться только к переменным.

```
int x = 5;
--(x++); // ошибка компиляции, т.к. выражение (x++)
         // не является переменной:
```

1.19. Арифметические операции побитового сдвига как замена умножению и делению

Арифметические операции побитового сдвига могут быть использованы в качестве замены умножению и целочисленному делению на числа, которые являются степенями 2-ки.

```
int x = 15;
x = x >> 2; // эквивалентно x = x/4;
x = x << 3; // если x ≠ 0 эквивалентно x = x*8;
```

1.20. Логические операции

Существует две группы логических операторов:

Булевы логические операции применяются к логическим (булевым) переменным/константам

действие	*название*	*обозначение*
NOT	булево отрицание	!
AND	булево AND по полной схеме	&
	булево AND по краткой схеме	&&
OR	булево OR по полной схеме	|
	булево OR по краткой схеме	||
XOR	булево XOR	^

Битовые логические операции применяются к целочисленным переменным/константам побитно.

действие	*название*	*обозначение*
NOT	битовое дополнение	~
AND	битовое AND	&
OR	битовое OR	|
XOR	битовое XOR	^

Булевы логические операции «!», «||» и «&&» нельзя применять к целочисленным константам/переменным.

Побитовую логическую операцию «~» нельзя применять к логическим константам/переменным.

Замечание. Результатом булевых логических операций всегда является константа типа boolean.

Замечание. Результатом битовых логических операций всегда является целочисленная константа.

1.21. Операторы «break» и «continue»

Оператор **break** может находится внутри тела циклов или операторов switch. Выполнение этого оператор внутри тела цикла (оператора switch) прерывает выполнение цикла (оператора switch) и выполнение передается следующей за циклом (оператором switch) строке кода.

Оператор **continue** допускается только внутри тела циклов. Выполнение этого оператора прерывает выполнение текущей итерации и следующей выполняемой строкой будет либо следующая итерация, либо, если прерванная итерация была последней, первая строка после цикла.

1.22. Параметры оператора цикла «for»

Оператор цикла **for** может в качестве первого и третьего параметров иметь выражения, состоящие из последовательности операторов присваивания (с операцией) или операций инкремента/декремента, разделенных запятыми.

```
int x, y, z;
for (x = 1, y = 2, z = 3; x+y+z < 100; x++, y++, z++) {
    System.out.println(x+" "+y+" "+z);
}
```

1.23. Операторы цикла

В Java существует три оператора цикла: **for**, **while** и **do/while**.

По выразительной мощности все три оператора эквиваленты, т.е., любой из трех может быть выражен с помощью любого из оставшихся двух, записанным соответствующим образом.

1.24. Объявление массивов

При объявлении массивов квадратные скобки могут стоять как перед переменной так и после.

Если перед переменной ставятся хотя бы одна пара квадратных скобок, то запись возможна без пробелов.

// x – трехмерный массив целых чисел, требующий
// дальнейшей инициализации:

```
int[]x[][] = new int[5][][];
```

Замечание. При выделении памяти под массив с помощью оператора new (т.е. при создании массива), элементы массива заполняются нулями только в том случае, когда в правой части от оператора **new** и типа элемента массива в квадратных скобках указаны все размерности массива.

Допускается указывать только *первые* размерности, но при этом потребуется дальнейшая процедура по размещению в памяти с помощью оператора new не созданных размерностей. Это позволяет создавать непрямоугольные массивы.

Пример непрямоугольного массива:

```
int[][]x = new int[5][];
for (int j = 0; j < 5; j++)
    x[j] = new int[j];
```

Массив x будет иметь «треугольную структуру»:

```
0
0 0
0 0 0
0 0 0 0
0 0 0 0 0
```

1.25. Массив – это объект

Массивы в Java являются *объектами*: каждый массив является наследником базового класса java.lang.Object и содержит методы, которые в нем определены.

Массивы наследовать нельзя.

Замечание. Элементы массива можно рассматривать как *поля* класса-массива. При создании массива с помощью оператора **new** (с указанием размерности) эти поля инициализируются значениями по умолчанию для соответствующих типов (для численных типов – 0; для boolean – false; для объектов – null):

```
boolean[] f = new boolean[3]; // f[0] == f[1] == f[2] == false
```

1.26. Инициализация массивов

Массивы можно инициализировать при объявлении двумя способами.

// правая часть равенства - массив-константа:

```
int[][] x = {{1, 2},{3, 4}};
```

и

// правая часть равенства - анонимный массив:

```
int[][] x = new int[][] {{1, 2},{3, 4}};
```

При передаче объекта-массива в качестве параметра метода может быть использован анонимный массив.

```
public static void main (String args []) throws Exception {
    // new int[] {1, 2, 3, 4, 5} – анонимный массив
    print(new int[] {1, 2, 3, 4, 5});
}

public static void print(int[] x) {
    for (int j = 0; j < x.length; j++) System.out.println(x[j]);
}
```

1.27. Оператор «switch»

Если в теле оператора **switch** выполняется блок код, соответствующий некоторому **case** выражению и этот код *не содержит* оператор **break**, то после выполнения кода управление будет передано следующей **case** конструкции, даже если значение селектора (выражения, которое прописано в заголовке оператора **switch**) *не совпадает* со значением выражения, стоящего в заголовке этой следующей **case** конструкции.

```
int x, y = 1;
switch (y) {
    case 1: x = 1;
    case 2: x = 2;
} // x = 2
```

Управление будет передаваться каждой последующей конструкции **case** вплоть до тех пор, пока не встретится оператор **break** или пока не исчерпаются все **case** выражения. В последнем случае будет выполнена конструкция **default** (если она есть) и оператор **switch** закончит свое выполнение.

Замечание. Конструкцию **default** обычно ставят после всех **case** конструкций, хотя ее можно ставить и между **case**-ами.

```
int x = 0, y = 3;
switch (y) {
    case 1: x = 1; break;
    default: x = 100;
    case 2: x = 2;
} // x = 2
```

В любом случае в конце тела **case/default** выражений следует ставить оператор **break**. Этот оператор не ставят только в том случае, когда нескольким значениям селектора **switch** должно соответствовать одно и тоже действие.

```
int x = 0, y = 3;
switch (y) {
    case 1:
    case 2: x = 1; break;
    case 3:
    case 4: x = 4; break;
    default: x = 0;
} // x = 4
```

1.28. Приоритет логических операций

Все логические операции упорядочены по приоритету.

Очередность выполнения булевых логических операций:

	название	*обозначение*	*ассоциативность*
1	отрицание	!	правая
2	AND по полной схеме	&	левая
3	XOR	^	левая
4	OR по полной схеме	|	левая
5	AND по краткой схеме	&&	левая
6	OR по краткой схеме	||	левая

Очередность выполнения битовых логических операций:

	название	*обозначение*	*ассоциативность*
1	битовое дополнение	~	правая
2	битовое AND	&	левая
3	битовое XOR	^	левая
4	битовое OR	|	левая

Замечание. Компилятор Java может оптимизировать вычисление булевых логических выражений, которые не содержат скобок. Например, если некоторое булево выражение не содержит скобки, но включает в качестве первого слагаемого значение true, причем применяется операция OR по краткой схеме, то остальные компоненты выражения не вычисляются в независимости от приоритета остальных операций.

1.29. Булева логическая операция «!»

Операция «!» предназначена для вычисления отрицания логической константы. Операнд и результат операции имеют тип boolean.

Пример правильного использования операции:

```
boolean x = false, y = !x;
```

Пример неправильного использования операции:

```
byte x = 0;
boolean y = !x;
```

1.30. Операции и операторы, результат которых может иметь логический тип

К операциям и операторам, которые могут иметь в качестве результата значение логического типа (boolean) относятся следующие.

1) оператор приведения типов: **(type)**
2) оператор проверки типа: **instanceof**
3) операции сравнения: **< <= > >= == !=**
4) булевы логические операции: **! & && | || ^**
5) оператор выбора: **?:**

1.31. Операции и операторы, операндами которых может иметь логический тип

К операциям и операторам, которые могут в качестве одного из своих операндов иметь значение логического типа, относятся следующие.

1) оператор приведения типов: **(type)**
2) операции сравнения: **==** **!=**
3) булевы логические операции: **!** **&** **&&** **|** **||** **^**
4) оператор выбора: **?:**

1.32. Операция деления « / »

Операция деления «/» в качестве операндов может иметь вещественные или целочисленные константы.

Результатом операции является числовая константа, тип которой зависит от типов операндов.

Если оба операнда (делимое и делитель) имеют *целочисленный* тип, то и результат будет иметь целочисленный тип, т.е. **int** (будет происходить отбрасывание остатка). Например блок кода

int x = 343; int y = 43; float z = x/y; // z = 7

загрузит в переменную **z** значение 7, т.к. 343/43 = 7,97...

1.33. Операция определения остатка от деления « % »

Операция «%» предназначена для определения остатка от деления двух целочисленных констант. Тип операндов и результата - целочисленный. Пример правильного использования операции:

```
int x = 17;
int y = 5;
float z = x % y; // z = 2
```

В результате выполнения этого блока кода в переменную z будет загружено значение 2, т.к. 17 = 3*5+2.

Практические задания

1.1. Написать программу, которая находит наибольший общий делитель двух целых положительных чисел.

1.2. Написать программу, которая находит сумму цифр произвольного целого числа.

1.3. Написать программу проверки того, что заданное число X – простое (т.е. не делится без остатка ни на какие числа, кроме себя и 1). Число X задавать в коде программы.

1.4. Сосчитать сумму ряда 1! - 2! + 3! – 4! + 5! - … + n! для заданного числа n > 0. Чем шире диапазон возможных значений n, тем лучше. Число n задавать в коде программы.

1.5. Подсчитать, сколько шестизначных цифр имеют равную сумму трех первых и трех последних цифр.

1.6. Разместить в памяти массив из 20 элементов и заполнить его рядом Фиббоначчи: 1, 1, 2, 3, 5, 8, 13, 21, … В этом ряду каждое следующее число является суммой двух предыдущих.

1.7. Создать целый массив из 100 элементов и заполнить его простыми числами: 2, 3, 5, 7, 11, 13, 17, …

1.8. Создать двумерный массив символов и заполнить его буквами 'Ч' и 'Б' в шахматном порядке.

1.9. Создать целый шестимерный массив с двумя значениями в каждом измерении. Заполнить массив числами из начала натурального ряда: 1, 2, 3, … Сказать, сколько потребуется чисел ?

1.10. Создать "треугольный" массив из 10 строк и заполнить его биномиальными коэффициентами (треугольник Паскаля)

1
1 2 1
1 3 3 1
1 4 6 4 1