Язык Java – Лекция 4. Консольный ввод-вывод и операции

4.1 Консольный ввод-вывод

Для взаимодействия с консолью в Java применяется **класс System**[1,2], а его функциональность собственно обеспечивает консольный ввод и вывод.

4.1.1 Вывод на консоль

4.1.1.1 Вывод строки

Для создания **потока вывода** в класс System **определен объект out**. **В этом объекте** определен метод **println**, который позволяет вывести на консоль некоторое значение с **последующим переводом курсора** консоли на **следующую строку**. Например,

```
public class Program {
    public static void main(String[] args) {
            System.out.println("Hello world!");
     System.out.println("Bye world...");
   }
}
```

В метод println передается **любое значение**, ***как правило***, строка, которое надо вывести на консоль. И в данном случае мы получим следующий вывод: - проверить самостоятельно.

При необходимости можно и не переводить курсор на следующую строку. В этом случае можно использовать метод **System.out.print()**, который аналогичен println за тем исключением, что не осуществляет перевода на следующую строку.

Проверить, а также реализовать перевод печатающей каретки (головки) на следующую стоку без использования println.

[1] https://java-blog.ru/osnovy/klass-system-java
[2] https://javadevblog.com/rabota-s-klassom-java-system-java-lang-system.html

4.1.1.2 Вывод данных (не строк!)

Нередко необходимо подставлять в строку какие-нибудь данные. Например, у нас есть два числа, и мы хотим вывести их значения на экран. В этом случае мы можем, например, написать так:

```
public class Program {
   public static void main(String[] args) {
      int x=5;
      int y=6;
      System.out.println("x=" + x + "; y=" + y);
   }
}
```

Но в Java есть также функция для **форматированного** вывода, унаследованная от языка C: System.out.printf(). С ее помощью мы можем переписать предыдущий пример следующим образом:

```
int x=5;
int y=6;
System.out.printf("x=%d; y=%d \n", x, y);
```

В данном случае символы %d обозначают спецификатор формата, **вместо которого подставляет один из аргументов**. Спецификаторов и соответствующих им аргументов может быть множество. В данном случае у нас только два аргумента, поэтому вместо первого %d подставляет значение переменной x, а вместо второго - значение переменной y. Сама буква d означает, что данный спецификатор будет использоваться для вывода целочисленных значений.

Кроме спецификатора %d мы можем использовать еще ряд спецификаторов для других типов данных:

%x: для вывода шестнадцатеричных чисел

%f: для вывода чисел с плавающей точкой

%e: для вывода чисел в экспоненциальной форме, например, 1.3e+01

%c: для вывода одиночного символа

%s: для вывода строковых значений

Понятие спецификации

Спецификация[3,4] — (от позднелатинского *specificatio*, от лат. ***species* — вид**, **разновидность** и лат. *facio* — **делаю**) — **документ, устанавливающий требования** (ГОСТ Р ИСО 9000).

Варианты определения

- **Конструкторский документ**, содержащий перечень составных частей, входящих в специфицируемое изделие, а также других конструкторских документов, относящиеся к этому изделию и к его неспецифицируемым составным частям.
- **Точное описание потребностей**, которые необходимо удовлетворить, и важных требуемых характеристик.
- **Определение и перечень специфических особенностей** чего-либо, **уточненная классификация** чего-либо.
- **Набор требований и параметров**, которым удовлетворяет некоторый технический объект.

Спецификаторы формата в Java

Спецификаторы формата начинаются с символа процента (%) и заканчиваются “символом типа”, который указывает тип данных (int, float и т. Д.), Которые будут преобразованы основным способом представления данных (десятичный, шестнадцатеричный и т. Д.). Общий синтаксис спецификатора формата:

% [flags] [width] [.precision] [argsize] typechar

Например:

```
public class Program {
    public static void main(String[] args) {
        String name = "Tom";
        int age = 30;
        float height = 1.7f;
        System.out.printf("Name: %s  Age: %d  Height: %.2f \n", name, age, height);
```

[3] https://ru.wikipedia.org/wiki/Спецификация

[4] https://www.specialist.ru/course/tzit

```
    }
}
```

При выводе чисел с плавающей точкой мы можем указать количество знаков после запятой, для этого используем спецификатор на %.2f, где .2 указывает, что после запятой будет два знака.

Спецификатор формата	Применено преобразование
%%	Вставляет знак %
%x %X	Целое шестнадцатеричное число
%t %T	Время и дата
%s %S	Строка
%n	Вставляет символ новой строки
%o	Восьмеричное целое число
%f	Десятичное число с плавающей запятой
%e %E	Научная нотация
%g	Заставляет программу форматирования использовать либо %f, либо %e, в зависимости от того, что короче
%h %H	Хэш-код аргумента
%d	Десятичное целое число
%c	Символ
%b %B	Логическое
%a %A	Шестнадцатеричные числа с плавающей запятой

Рисунок 4.1. Спецификаторы формата Java

4.1.2. Ввод с консоли[5]

https://uproger.com/spring-boot-101-vvedenie-v-sozdanie-veb-prilozhenij/
https://mnogoblog.ru/primery-na-java-chast-1
https://developer.ibm.com/tutorials/j-introtojava2/

Для получения ввода с консоли в классе System определен объект in. Однако непосредственно через объект System.in не очень удобно работать, поэтому, как правило, используют класс **Scanner, который, в свою очередь использует System.in**. Например, напишем маленькую программу, которая осуществляет ввод чисел:

[5] https://mnogoblog.ru/java-urok-4-vvod-chisel-s-klaviatury-system-in

```java
import java.util.Scanner;

public class Program {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);
        System.out.print("Input a number: ");
        int num = in.nextInt();

        System.out.printf("Your number: %d \n", num);
        in.close();
    }
}
```

Рисунок 4.2. Пример ввода чисел

Так как класс **Scanner** находится в **пакете java.util**, то мы вначале его импортируем с помощью инструкции import **java.util.Scanner**.

Для создания самого объекта Scanner в его **конструктор передается объект System.in**. После этого мы можем получать вводимые значения. Например, в данном случае вначале выводим приглашение к вводу и затем получаем вводимое число в переменную num.

Чтобы получить введенное число, используется метод **in.nextInt()**, который возвращает введенное с клавиатуры целочисленное значение.

Для ввода значений каждого **примитивного** (?) типа в классе Scanner определен свой метод. Ниже – перечисление этих методов:

- next(): считывает введенную строку до первого пробела
- nextLine(): считывает всю введенную строку
- nextInt(): считывает введенное число int
- nextDouble(): считывает введенное число double
- nextBoolean(): считывает значение boolean
- nextByte(): считывает введенное число byte
- nextFloat(): считывает введенное число float
- nextShort(): считывает введенное число short

На рисунке 4.3 программа ввода информации о человеке: имя, возраст, рост. Замечание: вещественные числа вводятся с учётом текущей локализации системы Java – в России, Германии, Франции используется запятая: 20,5.

```java
import java.util.Scanner;

public class Program {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);
        System.out.print("Input name: ");
        String name = in.nextLine();
        System.out.print("Input age: ");
        int age = in.nextInt();
        System.out.print("Input height: ");
        float height = in.nextFloat();
        System.out.printf("Name: %s  Age: %d  Height: %.2f \n", name, age, height);
        in.close();
    }
}
```

Рисунок 4.3.

4.2 Операции в Java

Понятие операнда! Сущность, над которой выполняется какая-либо операция (переменная или число, участвующее в операции).

Есть унарные операции (выполняются над одним операндом), бинарные - над двумя операндами, а также тернарные - выполняются над тремя операндами.

4.2.1 Арифметические

В **арифметических операциях участвуют числа**. В Java есть бинарные арифметические операции (производятся над двумя операндами) и унарные (выполняются над одним операндом). К бинарным операциям относят следующие:

операция сложения(+), вычитания(-), умножения(*), деления(/) двух чисел (рисунок 4.4):

```java
int a = 10;
int b = 7;
int c = a + b;  // 17
int d = 4 + b;  // 11
```

```java
int a = 10;
int b = 7;
int c = a - b;  // 3
int d = 4 - a;  // -6
```

```java
int a = 10;
int b = 7;
int c = a * b;  // 70
int d = b * 5;  // 35
```

```java
int a = 20;
int b = 5;
int c = a / b;  // 4
double d = 22.5 / 4.5;  // 5.0
```

Рисунок 4.4

При делении стоит учитывать, что если в операции участвуют два целых числа, то результат деления будет округляться до целого числа, даже если результат присваивается переменной float или double. Чтобы результат

представлял число с плавающей точкой, один из операндов также должен представлять число с плавающей точкой.

```
double k = 10 / 4;        // 2
System.out.println(k);
double k = 10.0 / 4;     // 2.5
System.out.println(k);
```

Операция получения остатка от деления двух чисел (%):

```
int a = 33;
int b = 5;
int c = a % b;        // 3
int d = 22 % 4;     // 2 (22 - 4*5 = 2)
```

Унарные арифметические операции, которые производятся над одним числом: ++ (инкремент) и -- (декремент). Каждая из операций имеет две разновидности: **префиксная** и **постфиксная**:

++ (префиксный инкремент): предполагает увеличение переменной на единицу, например, z=++y (вначале значение переменной y увеличивается на 1, а затем ее значение присваивается переменной z).

```
int a = 8;
int b = ++a;
System.out.println(a);  // 9
System.out.println(b);  // 9
```

++ (постфиксный инкремент): также представляет увеличение переменной на единицу, например, z=y++ (вначале значение переменной y присваивается переменной **z**, а потом значение переменной **y** увеличивается на 1).

```
Int a = 8;
int b = a++;
System.out.println(a);  // 9
System.out.println(b);  // 8
```

-- (префиксный декремент): уменьшение переменной на единицу, например, z=--y (вначале значение переменной y уменьшается на 1, а потом ее значение присваивается переменной z):

```
int a = 8;
int b = --a;
System.out.println(a);  // 7
System.out.println(b);  // 7
```

-- (постфиксный декремент): z=y-- (сначала значение переменной y присваивается переменной z, а затем значение переменной y уменьшается на 1):

```
int a = 8;
int b = a--;
System.out.println(a);  // 7
System.out.println(b);  // 8
```

Приоритет арифметических операций

Одни операции имеют больший приоритет, чем другие, и поэтому выполняются вначале. Операции в **порядке уменьшения приоритета**:

++ (постфиксный инкремент), -- (постфиксный декремент)

++ (префиксный инкремент), -- (префиксный декремент)

* (умножение), / (деление), % (остаток от деления)

+ (сложение), - (вычитание)

Приоритет операций следует учитывать при выполнении набора арифметических выражений:

```
int a = 8;
int b = 7;
int c = a + 5 * ++b;
System.out.println(c);   // 48
```

Вначале будет выполняться операция инкремента ++b, которая имеет больший приоритет - она увеличит значение переменной b и возвратит его в качестве результата. Затем выполняется умножение 5 * ++b, и только в последнюю очередь выполняется сложение a + 5 * ++b.

Скобки позволяют переопределить порядок вычислений:

```
int a = 8;
int b = 7;
int c = (a + 5) * ++b;
System.out.println(c);   // 104
```

Несмотря на то, что операция сложения имеет меньший приоритет, но вначале будет выполняться именно сложение, а не умножение, так как операция сложения заключена в скобки.

Ассоциативность операций

Кроме приоритета операции отличаются таким понятием как ассоциативность. Когда операции имеют один и тот же приоритет, порядок вычисления определяется ассоциативностью операторов. В зависимости от ассоциативности есть два типа операторов:

Левоассоциативные операторы, которые выполняются слева направо

Правоассоциативные операторы, которые выполняются справа налево

Это надо учитывать при записи арифметических выражений в случае, когда некоторые операции, например, операции умножения и деления, имеют один и тот же приоритет: int x = 10 / 5 * 2;

Стоит нам трактовать это выражение как (10 / 5) * 2 или как 10 / (5 * 2)? Ведь в зависимости от трактовки мы получим разные результаты.

Все арифметические операторы (кроме префиксного инкремента и декремента) являются левоассоциативными, то есть выполняются слева направо. Поэтому выражение 10 / 5 * 2 необходимо трактовать как (10 / 5) * 2, то есть результатом будет 4.

Особенность работы с вещественными числами в Java!

Следует отметить, что **числа с плавающей точкой не подходят для финансовых и других вычислений, где ошибки при округлении могут быть критичными**.

```
double d = 2.0 - 1.1;
System.out.println(d);
```

В данном случае переменная d будет равна не 0.9, как можно было бы изначально предположить, а 0.8999999999999999. Подобные ошибки точности возникают из-за того, что на низком уровне для представления чисел с плавающей точкой применяется двоичная система, однако для числа 0.1 не существует двоичного представления, также, как и для других дробных значений. Поэтому в таких случаях обычно применяется класс **BigDecimal**, который позволяет обойти подобные ситуации.

4.2.2 Поразрядные операции – операции над целыми числами

Поразрядные операции выполняются над отдельными разрядами или битами чисел. В данных операциях в качестве операндов могут выступать только целые числа.

Каждое число имеет определенное двоичное представление. Например, число 4 в двоичной системе 100, а число 5 - 101 и так далее.

К примеру, возьмем следующие переменные:

```
byte b = 7;    // 0000 0111
short s = 7;   // 0000 0000 0000 0111
```

Тип byte занимает 1 байт или 8 бит, соответственно представлен 8 разрядами. Поэтому значение переменной b в двоичном коде будет равно 00000111.

Тип short занимает в памяти 2 байта или 16 бит, поэтому число данного типа будет представлено 16 разрядами. И в данном случае переменная s в двоичной системе будет иметь значение 0000 0000 0000 0111.

Для записи чисел со знаком в Java применяется **дополнительный код** (two's complement), при котором старший разряд является знаковым. Если его значение равно 0, то число положительное, и его двоичное представление не отличается от представления беззнакового числа. Например, 0000 0001 в десятичной системе 1.

Если старший разряд равен 1, то мы имеем дело с отрицательным числом. Например, 1111 1111 в десятичной системе представляет -1. Соответственно, 1111 0011 представляет -13.

Отрицательные числа в памяти компьютера - дополнительный код

Дополнительный код— наиболее распространённый способ представления отрицательных целых чисел в компьютерах. Он позволяет заменить операцию вычитания на операцию сложения и сделать операции сложения и вычитания одинаковыми для знаковых и беззнаковых чисел, чем упрощает архитектуру ЭВМ.

Дополнительный код отрицательного числа можно получить инвертированием модуля двоичного числа (первое дополнение) и прибавлением к инверсии единицы (второе дополнение), либо вычитанием числа из нуля. Дополнительный код (дополнение до 2) двоичного числа получается добавлением 1 к младшему значащему разряду его дополнения до 1.

Дополнение до 2 двоичного числа определяется как величина, полученная вычитанием числа из наибольшей степени двух (из 2^N для N-

битного дополнения до 2). **При записи числа в дополнительном коде старший разряд является знаковым**.

Если его значение равно 0, то в остальных разрядах записано положительное двоичное число, совпадающее с прямым кодом.

Если число, записанное в прямом коде, отрицательное, то все разряды числа инвертируются, а к результату прибавляется 1. К получившемуся числу дописывается старший (знаковый) разряд, равный 1.

Двоичное 8-ми разрядное число *со знаком* в дополнительном коде может представлять любое целое в диапазоне от −128 до +127. Если старший разряд равен нулю, то наибольшее целое число, которое может быть записано в оставшихся 7 разрядах равно 01111111, что равно 127 (рисунок 4.5).

Десятичное представление	Код двоичного представления (8 бит)		
	прямой	обратный	дополнительный
127	01111111	01111111	01111111
1	00000001	00000001	00000001
0	00000000	00000000	00000000
-0	10000000	11111111	---
-1	10000001	11111110	11111111
-2	10000010	11111101	11111110
-3	10000011	11111100	11111101
-4	10000100	11111011	11111100
-5	10000101	11111010	11111011
-6	10000110	11111001	11111010
-7	10000111	11111000	11111001
-8	10001000	11110111	11111000
-9	10001001	11110110	11110111
-10	10001010	11110101	11110110
-11	10001011	11110100	11110101
-127	11111111	10000000	10000001
-128	---	---	10000000

Рисунок 4.5. Целые числа в разном представлении

Поразрядные <u>логические</u> операции над числами

Логические операции над числами представляют поразрядные операции. В данном случае числа рассматриваются в двоичном представлении, например, 2 в двоичной системе равно 10 и имеет два разряда, число 7 - 111 и имеет три разряда.

<u>(&) - (логическое умножение)</u>

Умножение производится поразрядно, и если у обоих операндов значения разрядов равно 1, то операция возвращает 1, иначе возвращается число 0.

```
int a1 = 2; //010
 int b1 = 5;//101
 System.out.println(a1&b1); // результат 0

int a2 = 4; //100
int b2 = 5; //101
System.out.println(a2 & b2); // результат 4
```

В первом случае у нас два числа 2 и 5.

2 в двоичном виде представляет число 010, а 5 - 101. Поразрядное умножение чисел (0*1, 1*0, 0*1) дает результат 000.

Во втором случае у нас вместо двойки число 4, у которого в первом разряде 1, так же, как и у числа 5, поэтому здесь результатом операции (1*1, 0*0, 0 *1) = 100 будет число 4 в десятичном формате.

<u>(|) - (логическое сложение)</u>

Данная операция также производится по двоичным разрядам, но теперь возвращается единица, если хотя бы у одного числа в данном разряде имеется единица (операция "логическое ИЛИ").

```
int a1 = 2; //010
int b1 = 5;//101
System.out.println(a1|b1); // результат 7 - 111
int a2 = 4; //100
int b2 = 5;//101
System.out.println(a2 | b2); // результат 5 - 101
```

(^) - (логическое исключающее ИЛИ)

Также эту операцию называют **XOR**, нередко ее применяют для простого шифрования:

```
int number = 45; // 1001 Значение, которое надо зашифровать - в двоичной форме 101101
int key = 102; //Ключ шифрования - в двоичной системе 1100110
int encrypt = number ^ key; //Результатом будет число 1001011 или 75
System.out.println("Зашифрованное число: " +encrypt);

int decrypt = encrypt ^ key; // Результатом будет исходное число 45
System.out.println("Расшифрованное число: " + decrypt);
```

Здесь также производятся поразрядные операции. Если у нас значения текущего разряда у обоих чисел разные, то возвращается 1, иначе возвращается 0. Например, результатом выражения 9^5 будет число 12. А чтобы расшифровать число, мы применяем обратную операцию к результату.

(~) - (логическое отрицание)

Поразрядная операция, которая инвертирует все разряды числа: если значение разряда равно 1, то оно становится равным нулю, и наоборот.

```
byte a = 12;                // 0000 1100
System.out.println(~a);     //  1111 0011   или -13
```

Поразрядные операции сдвига

Операции сдвига также производятся над разрядами чисел. Сдвиг может происходить вправо и влево.

- **a<<b** - сдвигает число a влево на b разрядов. Например, выражение 4<<1 сдвигает число 4 (которое в двоичном представлении 100) на один разряд влево, в результате получается число 1000 или число 8 в десятичном представлении.
- **a>>b** - смещает число a вправо на b разрядов. Например, 16>>1 сдвигает число 16 (которое в двоичной системе 10000) на один разряд вправо, то есть в итоге получается 1000 или число 8 в десятичном представлении.
- **a>>>b** - в отличие от предыдущих типов сдвигов данная операция представляет беззнаковый сдвиг - сдвигает число a вправо на b разрядов. Например, выражение -8>>>2 будет равно 1073741822.

Таким образом, если исходное число, которое надо сдвинуть в ту или другую сторону, делится на два, то фактически получается умножение или деление на два. Поэтому подобную операцию можно использовать вместо непосредственного умножения или деления на два, так как операция сдвига на аппаратном уровне менее дорогостоящая операция в отличие от операции деления или умножения.

4.3 Условные выражения

Условные выражения представляют собой некоторое условие и возвращают значение типа boolean, то есть значение true (если условие истинно), или значение false (если условие ложно). **К условным выражениям относятся операции сравнения и логические операции**.

4.3.1 Операции сравнения (отношений)

В операциях сравнения сравниваются два операнда, и возвращается значение типа boolean - true, если выражение верно, и false, если выражение неверно.

(==) - сравнивает два операнда на равенство и возвращает true (если операнды равны) и false (если операнды не равны);

(!=) - сравнивает два операнда и возвращает true, если операнды НЕ равны, и false, если операнды равны;

(<) - меньше чем;

(>) - больше чем;

(>=) - больше или равно;

(<=) - меньше или равно.

```
int a = 10;
int b = 4;
boolean c = a == b;
boolean d = a == 10;
```

```
int a = 10;
int b = 4;
boolean c = a != b;
boolean d = a != 10;
```

```
int a = 10;
int b = 4;
boolean c = a < b;
```

```
int a = 10;
int b = 4;
boolean c = a > b;

boolean c = 10 >= 10;
boolean b = 10 >= 4;
boolean d = 10 >= 20;

boolean c = 10 <= 10;
boolean b = 10 <= 4;
boolean d = 10 <= 20
```

4.3.2 Логические операции

Логические операции также представляют условие и возвращают true или false и обычно **объединяют несколько операций сравнения**. К логическим операциям относят следующие:

- **(|) - c=a|b**; (c равно true, если либо a, либо b (либо и a, и b) равны true, иначе c будет равно false)
- **(&) - c=a&b**; (c равно true, если и a, и b равны true, иначе c будет равно false)
- **(!) - c=!b**; (c равно true, если b равно false, иначе c будет равно false)
- **(^) - c=a^b**; (c равно true, если либо a, либо b (но не одновременно) равны true, иначе c будет равно false)
- **(||) - c=a||b**; (c равно true, если либо a, либо b (либо и a, и b) равны true, иначе c будет равно false)
- **(&&) - c=a&&b**; (c равно true, если и a, и b равны true, иначе c будет равно false)

Здесь у нас две пары операций **|** и **||** (а также **&** и **&&**) выполняют похожие действия, **однако же они не равнозначны**.

Выражение c=a|b; будет вычислять **сначала оба значения** - a и b и на их основе выводить результат.

В выражении же c=a||b; **вначале** будет вычисляться значение **a**, и если оно равно true, то вычисление значения b уже смысла не имеет, так как у нас в любом случае уже c будет равно true. Значение b будет вычисляться только в том случае, если a равно false

То же самое касается пары операций &/&&. В выражении c=a&b; будут вычисляться оба значения - a и b.

В выражении же c=a&&b; сначала будет вычисляться значение a, и если оно равно false, то вычисление значения b уже не имеет смысла, так как

значение c в любом случае равно false. Значение b будет вычисляться только в том случае, если a равно true

Таким образом, операции || и && более удобны в вычислениях, позволяя сократить время на вычисление значения выражения и тем самым повышая производительность. А операции | и & больше подходят для выполнения поразрядных операций над числами.

```
boolean a1 = (5 > 6) || (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true
boolean a2 = (5 > 6) || (4 > 6); // 5 > 6 - false, 4 > 6 - false, поэтому возвращается false
boolean a3 = (5 > 6) && (4 < 6); // 5 > 6 - false, поэтому возвращается false (4 < 6 - true, но не вычисляется)
boolean a4 = (50 > 6) && (4 / 2 < 3); // 50 > 6 - true, 4/2 < 3 - true, поэтому возвращается true
boolean a5 = (5 < 6) ^ (4 > 6); // 5 < 6 - true, поэтому возвращается true (4 > 6 - false)
boolean a6 = (50 > 6) ^ (4 / 2 < 3); // 50 > 6 - true, 4/2 < 3 - true, поэтому возвращается false
```

4.4 Операции присваивания и приоритет операций

В завершении рассмотрим операции присваивания, которые в основном представляют комбинацию простого присваивания с другими операциями:

- **(=)** - просто приравнивает одно значение другому: c=b;
- **(+=)** - c+=b; (переменной c присваивается результат сложения c и b)
- **(-=)** - c-=b; (переменной c присваивается результат вычитания b из c)
- **(*=)** - c*=b; (переменной c присваивается результат произведения c и b)
- **(/=)** - c/=b; (переменной c присваивается результат деления c на b)
- **(%=)** - c%=b; (переменной c присваивается остаток от деления c на b)
- **(&=)** - c&=b; (переменной c присваивается значение c&b)
- **(|=)** - c|=b; (переменной c присваивается значение c|b)
- **(^=)** - c^=b; (переменной c присваивается значение c^b)
- **(<<=)** - c<<=b; (переменной c присваивается значение c<<b)
- **(>>=)** - c>>=b; (переменной c присваивается значение c>>b)
- **(>>>=)** - c>>>=b; (переменной c присваивается значение c>>>b)

```
int a = 5;
a += 10;       // 15
a -= 3;        // 12
a *= 2;        // 24
a /= 6;        // 4
a <<= 4;       // 64
a >>= 2;       // 16
System.out.println(a);  // 16
```

4.5 Приоритет операций

При работе с операциями важно понимать их приоритет, который можно описать следующей таблицей:

expr++ expr--
++expr --expr +expr -expr ~ !
* / %
+ -
<< >> >>>
< > <= >= instanceof
== !=
&
^
|
&&
||
? : (тернарный оператор)
= += -= *= /= %= &= ^= |= <<= >>= >>>= (операторы присваивания)

Рисунок 4.6 Приоритетность операций

Чем выше оператор в этой таблице, тем больше его приоритет. При этом скобки повышают приоритет операции, используемой в выражении.

4.6 Преобразования базовых типов данных

Каждый базовый тип данных занимает определенное количество байт памяти. Это накладывает ограничение на операции, в которые вовлечены различные типы данных. Рассмотрим следующий пример:

```
int a = 4;
byte b = a;        // ! Ошибка
```

В данном коде мы столкнемся с ошибкой. Хотя и тип byte, и тип int представляют целые числа. Более того, значение переменной a, которое присваивается переменной типа byte, вполне укладывается в диапазон значений для типа byte (от -128 до 127). Тем не менее мы сталкиваемся с ошибкой на этапе компиляции. Поскольку в данном случае мы пытаемся присвоить некоторые данные, которые занимают 4 байта, переменной, которая занимает всего один байт.

Тем не менее в программе может потребоваться, чтобы подобное преобразование было выполнено. В этом случае необходимо использовать операцию преобразования типов (операция ()):

```
int a = 4;
byte b = (byte)a;  // преобразование типов: от типа int к типу byte
System.out.println(b);   // 4
```

Операция преобразования типов предполагает указание в скобках того типа, к которому надо преобразовать значение. Например, в случае операции (byte)a, идет преобразование данных типа int в тип byte. В итоге мы получим значение типа byte.

Явные и неявные преобразования

Когда в одной операции вовлечены данные разных типов, не всегда необходимо использовать операцию преобразования типов. Некоторые виды преобразований выполняются неявно, автоматически.

Автоматические преобразования

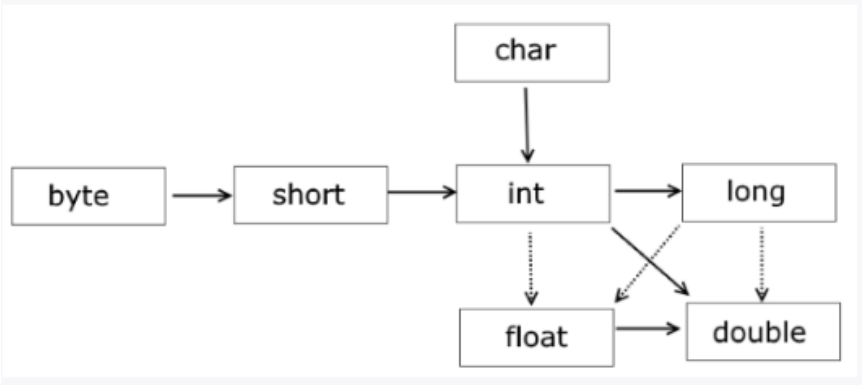

Рисунок 4.7.Граф автоматических преобразований типов.

Стрелками на рисунке показано, какие преобразования типов могут выполняться **автоматически**. Пунктирными стрелками показаны автоматические преобразования с потерей точности.

Автоматически без каких-либо проблем производятся **расширяющие преобразования** (widening) - они расширяют представление объекта в памяти.

```
byte b = 7;
int d = b;  // преобразование от byte к int
```

В данном случае значение типа byte, которое занимает в памяти 1 байт, расширяется до типа int, которое занимает 4 байта.

Расширяющие автоматические преобразования представлены следующими цепочками:

byte -> short -> int -> long

int -> double

short -> float -> double

char -> int

Автоматические преобразования с потерей точности

Некоторые преобразования могут производиться автоматически между типами данных одинаковой разрядности или даже от типа данных с большей разрядностью к типу с меньшей разрядностью. Это следующие цепочки преобразований: int -> float, long -> float и long -> double.

Они производятся без ошибок, но при преобразовании мы можем столкнуться с потерей информации.

```
int a = 2147483647;
float b = a;          // от типа int к типу float
System.out.println(b);    // 2.14748365E9
```

Явные преобразования

Во всех остальных преобразованиях примитивных типов явным образом применяется операция преобразования типов. Обычно это сужающие преобразования (**narrowing**) от типа с большей разрядностью к типу с меньшей разрядностью:

```
long a = 4;
int b = (int) a;
```

Потеря данных при преобразовании

При применении явных преобразований мы можем столкнуться с потерей данных. Например, в следующем коде у нас не возникнет никаких проблем:

```
int a = 5;
byte b = (byte) a;
System.out.println(b);       // 5
```

Число 5 вполне укладывается в диапазон значений типа byte, поэтому после преобразования переменная b будет равна 5. Но что будет в следующем случае:

```
int a = 258;
byte b = (byte) a;
System.out.println(b);       // 2
```

Результатом будет число 2. В данном случае число 258 вне диапазона для типа byte (от -128 до 127), поэтому произойдет усечение значения. Почему результатом будет именно число 2?

Число a, которое равно 258, в двоичной системе будет равно 00000000 00000000 00000001 00000010. Значения типа byte занимают в памяти только 8 бит. Поэтому двоичное представление числа int усекается до 8 правых разрядов, то есть 00000010, что в десятичной системе дает число 2.

Усечение рациональных чисел до целых

При преобразовании значений с плавающей точкой к целочисленным значениям, происходит усечение дробной части:

```
double a = 56.9898;
int b = (int)a;
```

Здесь значение числа b будет равно 56, несмотря на то, что число 57 было бы ближе к 56.9898. Чтобы избежать подобных казусов, надо применять функцию округления, которая есть в математической библиотеке Java:

```
double a = 56.9898;
int b = (int)Math.round(a);
```

Преобразования при операциях

Нередки ситуации, когда приходится применять различные операции, например, сложение и произведение, над значениями разных типов. Здесь также действуют некоторые правила:

если один из операндов операции относится к типу double, то и второй операнд преобразуется к типу double

если предыдущее условие не соблюдено, а один из операндов операции относится к типу float, то и второй операнд преобразуется к типу float

если предыдущие условия не соблюдены, один из операндов операции относится к типу long, то и второй операнд преобразуется к типу long

иначе все операнды операции преобразуются к типу int

Примеры преобразований:

```
int a = 3;
double b = 4.6;
double c = a+b;
```

Так как в операции участвует значение типа double, то и другое значение приводится к типу double и сумма двух значений a+b будет представлять тип double.

Другой пример:

```
byte a = 3;
short b = 4;
byte c = (byte)(a+b);
```

Две переменных типа byte и short (не double, float или long), поэтому при сложении они преобразуются к типу int, и их сумма a+b представляет значение

типа int. Поэтому если затем мы присваиваем эту сумму переменной типа byte, то нам опять надо сделать преобразование типов к byte.

Если в операциях участвуют данные типа char, то они преобразуются в int:

```
int d = 'a' + 5;
System.out.println(d);  // 102
```

Задание

Ответить на вопросы, получив ответы на основе результатов выполнения написанных вами программ.

Вопрос 1.

Какие преобразования типов НЕ выполняются автоматически (возможно, несколько вариантов):

Из short в int

Из int в short

Из boolean в int

Из byte в float

Вопрос 2

Что будет выведено на консоль в результате выполнения следующей программы и почему?

```
public class Program {
   public static void main(String[] args) {
      short shortNum = 257;
      byte byteNum = (byte)shortNum;
      System.out.println(byteNum);
   }
}
```