Побитовые операции

[**Статья из группы Java Developer**](https://javarush.ru/groups/java-developer)

13529 участников

Присоединиться

Тебе наверняка знакомо слово “бит”. Если же нет, давай познакомимся с ним :) **Бит** — минимальная единица измерения информации в компьютере. Его название происходит от английского “*binary digit*” — “двоичное число”. Бит может быть выражен одним из двух чисел: 1 или 0. Существует специальная система счисления, основанная на единицах и нулях — **двоичная**.

Не будем углубляться в дебри математики и отметим лишь, что любое число в Java можно сконвертировать в его двоичную форму. Для этого нужно использовать классы-обертки. Например, вот как можно сделать это для числа int:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

**int** x = 342;

System.out.println(Integer.toBinaryString(x));

}

}

**Вывод в консоль: *101010110*** 1010 10110 (я добавил пробел для удобства чтения) — это число 342 в двоичной системе. Мы фактически разделили это число на отдельные биты — нули и единицы. Именно с ними мы можем выполнять операции, которые называются побитовыми.

* **~** — побитовый оператор “НЕ”.

Он работает очень просто: проходится по каждому биту нашего числа и меняет его значение на противоположное: нули — на единицы, единицы — на нули. Если мы применим его к нашему числу 342, вот что получится: *101010110 — число 342 в двоичной системe 010101001 — результат выражения ~342* Попробуем выполнить это на практике:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

**int** x = 342;

System.out.println(~x);

}

}

**Вывод в консоль: *-343*** -343 — это наш результат (число 010101001) в привычной десятичной системе :) ,

* **&** — побитовый оператор “И”

Он, как видишь, довольно похож по написанию на логический “И” (&&). Оператор &&, как ты помнишь, возвращает true только если оба операнда являются истинными. Побитовый & работает схожим образом: он сравнивает два числа по битам. Результатом этого сравнения является третье число. Для примера, возьмем числа 277 и 432: *100010101 — число 277 в двоичной форме 110110000 — число 432 в двоичной форме* Далее оператор & сравнивает первый бит верхнего числа с первым битом нижнего. Поскольку это оператор “И”, то результат будет равен 1 только в том случае, если оба бита равны 1. Во всех остальных случаях результатом будет 0. **100010101** & **110110000** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **100010000 — результат работы &** Мы сравниваем сначала первые биты двух чисел друг с другом, потом вторые биты, третьи и т.д. Как видишь, только в двух случаях оба бита в числах были равны 1 (первый и пятый по счету биты). Результатом всех остальных сравнений стал 0. Поэтому в итоге у нас получилось число 100010000. В десятичной системе ему соответствует число 272. Давай проверим:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

System.out.println(277&432);

}

}

**Вывод в консоль: *272***

* **|** — побитовое “ИЛИ”. Принцип работы тот же — сравниваем два числа по битам. Только теперь если **хотя бы один** из битов равен 1, результат будет равен 1. Посмотрим на тех же числах — 277 и 432:

**100010101** | **110110000** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **110110101 — результат работы |** Здесь уже результат другой: нулями остались только те биты, которые в обоих числах были нулями. Результат работы — число 110110101. В десятичной системе ему соответствует число 437. Проверим:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

System.out.println(277|432);

}

}

**Вывод в консоль: *437*** Мы все посчитали верно! :)

* **^** — побитовое исключающее “ИЛИ” (также известно как XOR)

С таким оператором мы еще не сталкивались. Но ничего сложного в нем нет. Он похож на обычное “или”. Разница в одном: обычное “или” возвращает true, если хотя бы один операнд является истинным. Но не обязательно один — если оба будут true — то и результат true. А вот исключающее “или” возвращает true **только если один из операндов является истинным.** **Если истинны оба операнда**, обычное “или” вернет true(“хотя бы один истинный“), а вот **исключающее или вернет** false. Поэтому он и называется исключающим. Зная принцип предыдущих побитовых операций, ты наверняка и сам сможешь легко выполнить операцию 277^432. Но давай лучше лишний раз разберемся вместе :) **100010101** ^ **110110000** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **010100101 — результат работы ^** Вот и наш результат. Те биты, которые были в обоих числах одинаковыми, вернули 0 (не сработала формула “один из”). А вот те, которые образовывали пару 0-1 или 1-0, в итоге превратились в единицу. В результате мы получили число 010100101. В десятичной системе ему соответствует число 165. Давай посмотрим, правильно ли мы посчитали:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

System.out.println(277^432);

}

}

**Вывод в консоль: *165*** Супер! Все именно так, как мы и думали :) Теперь самое время познакомиться с операциями, которые называют **битовыми сдвигами**. Название, в принципе, говорит само за себя. Мы возьмем какое-то число и будем двигать его биты влево и вправо :) Давай посмотрим как это выглядит:

**Сдвиг влево**

Сдвиг битов влево обозначается знаком **<<** Пример:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

**int** x = 64;//значение

**int** y = 3;//количество

**int** z = (x << y);

System.out.println(Integer.toBinaryString(x));

System.out.println(Integer.toBinaryString(z));

}

}

В этом примере число x=64 называется **значением**. Именно его биты мы будем сдвигать. Сдвигать биты мы будем влево (это можно определить по направлению знака <<) В двоичной системе число 64 = 1000000 Число y=3 называется **количеством**. **Количество** отвечает на вопрос “**на сколько** бит вправо/влево нужно сдвинуть биты числа **x**” В нашем примере мы будем сдвигать их на **3 бита** влево. Чтобы процесс сдвига был более понятен, посмотрим на картинке. У нас в примере используются числа типа int. Int’ы занимают в памяти компьютера 32 бита. Вот так выглядит наше изначальное число 64:

Побитовые операции - 2

А теперь мы, в прямом смысле слова, берем каждый из наших битов и сдвигаем влево на 3 ячейки:

Побитовые операции - 3

Вот что у нас получилось. Как видишь, все наши биты сдвинулись, а из-за пределов диапазона добавились еще 3 нуля. 3 — потому что мы делали сдвиг на 3. Если бы мы сдвигали на 10, добавилось бы 10 нулей. Таким образом, выражение x << y означает “сдвинуть биты числа х на y ячеек влево”. Результатом нашего выражения стало число 1000000000, которое в десятичной системе равно 512. Проверим:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

**int** x = 64;//значение

**int** y = 3;//количество

**int** z = (x << y);

System.out.println(z);

}

}

**Вывод в консоль: *512*** Все верно! Теоретически, биты можно сдвигать до бесконечности. Но поскольку у нас число int, в распоряжении есть всего 32 ячейки. Из них 7 уже заняты числом 64 (1000000). Поэтому если мы сделаем, например, 27 сдвигов влево, наша единственная единица выйдет за пределы диапазона и “затрётся”. Останутся только нули!

**public** **class** Main {

**public** **static** **void** main(String[] args) {

**int** x = 64;//значение

**int** y = 26;//количество

**int** z = (x << y);

System.out.println(z);

}

}

**Вывод в консоль: *0*** Как мы и предполагали, единичка вышла за пределы 32 ячеек-битов и исчезла. У нас получилось 32-битное число, состоящее из одних нулей.

Побитовые операции - 4

Естественно, в десятичной системе ему соответствует 0. Простое правило для запоминания сдвигов влево: **При каждом сдвиге влево выполняется умножение числа на 2.** Например, попробуем без картинок с битами посчитать результат выражения 111111111 << 3 Нам нужно трижды умножить число 111111111 на 2. В результате у нас получается 888888888. Давай напишем код и проверим:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

System.out.println(111111111 << 3);

}

}

**Вывод в консоль: *888888888***

**Сдвиги вправо**

Они обозначаются знаком **>>**. Делают то же самое, только в другую сторону! :) Не будем изобретать велосипед и попробуем сделать это с тем же числом int 64.

**public** **class** Main {

**public** **static** **void** main(String[] args) {

**int** x = 64;//значение

**int** y = 2;//количество

**int** z = (x >> y);

System.out.println(z);

}

}

Побитовые операции - 5

Побитовые операции - 6

В результате сдвига на 2 вправо два крайних нуля нашего числа вышли за пределы диапазона и затерлись. У нас получилось число 10000, которому в десятичной системе соответствует число 16 **Вывод в консоль: *16*** Простое правило для запоминания сдвигов вправо: **При каждом сдвиге вправо выполняется деление на два с отбрасыванием любого остатка.** Например, 35 >> 2 означает, что нам нужно 2 раза разделить 35 на 2, отбрасывая остатки 35/2 = 17 (отбросили остаток 1) 17:2 = 8 (отбросили остаток 1) Итого, 35 >> 2 должно быть равно 8. Проверяем:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

System.out.println(35 >> 2);

}

}

**Вывод в консоль: *8***

**Приоритет операций в Java**

В процессе написания или чтения кода тебе часто будут попадаться выражения, в которых одновременно выполняются несколько операций. Очень важно понимать, в каком порядке они будут выполнены, иначе результат может быть неожиданным. Поскольку операций в Java много, все они были выделены в специальную таблицу:

**Operator Precedence**

| **Operators** | **Precedence** |
| --- | --- |
|  |  |
| postfix | expr++ expr-- |
| unary | ++expr --expr +expr ~ ! |
| Multiplicative | \* / % |
| additive | + - |
| shift | << >> >>> |
| relational | < > <= >= instanceof |
| equality | == != |
| bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| logical AND | && |
| logical OR | || |
| ternary | ? : |
| assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

Все операции выполняются **слева направо**, однако с учетом своего приоритета. Например, если мы пишем: int x = 6 - 4/2; вначале будет выполнена операция деления (4/2). Хоть она и идет второй по счету, но у нее **выше приоритет**. Круглые или квадратные скобки меняют любой приоритет на максимальный. Это ты наверняка помнишь еще со школы. Например, если добавить их к выражению: int x = (6 - 4)/2; первым выполнится именно вычитание, поскольку оно вычисляется в скобках. У логического оператора && приоритет довольно низкий, что видно из таблицы. Поэтому чаще всего он будет выполняться последним. Например: boolean x = 6 - 4/2 > 3 && 12\*12 <= 119; Это выражение будет выполняться так:

* 4/2 = 2

**boolean** x = 6 - 2 > 3 && 12\*12 <= 119;

* 12\*12 = 144

**boolean** x = 6 - 2 > 3 && 144 <= 119;

* 6-2 = 4

**boolean** x = 4 > 3 && 144 <= 119;

* Далее будут выполнены операторы сравнения:

4 > 3 = true

**boolean** x = true && 144 <= 119;

* 144 <= 119 = false

**boolean** x = true && false;

* И, наконец, последним, будет выполнен оператор “И” &&.

boolean x = true && false;  
boolean x = false;

Оператор сложения (+), например, имеет более высокий приоритет, чем оператор сравнения != (“не равно”);

Поэтому в выражении:

boolean x = 7 != 6+1;

сначала будет выполнена операция 6+1, потом проверка 7!=7 (false), а в конце — присваивания результата false переменной **x**. У присваивания вообще самый маленький приоритет из всех операций — посмотри в таблице.

Фух! Лекция у нас получилась большая, но ты справился! Если ты не до конца понял какие-то части этой и предыдущей лекций — не переживай, мы еще не раз коснемся данных тем в будущем. Вот тебе несколько полезных ссылок :

* Отличная [статья](https://vertex-academy.com/tutorials/ru/ponyatie-bita-rabota-s-bitami/) в картинках про побитовые операции
* [Числовые операторы](https://javarush.ru/quests/lectures/questmultithreading.level10.lecture03)
* [Логические операторы](https://javarush.ru/quests/lectures/questmultithreading.level10.lecture05" \t "_blank) — лекция JavaRush о логических и числовых операциях. Мы до них еще нескоро дойдем, но почитать можно уже сейчас, вреда не будет