Глава 4. Работа с числами в языке Java

Данная часть посвящена изучению работы с числами на более глубоком уровне. В ней рассматривается машинное представление целых и вещественных чисел, эффективное для аппаратной реализации, а также объясняются особенности и проблемы, к которым приводит такое представление.

4.1 Двоичное представление целых чисел

Позиционные и непозиционные системы счисления

Позиционная система счисления - это такой способ записи числа, при котором вес цифры зависит от занимаемой позиции и пропорционален степени некоторого числа. Основание степени называется основанием системы счисления.

Например, число 2006 в десятичной системе счисления представляется в виде суммы тысяч, сотен, десятков и единиц:

$2006 = 2*10^3 + 0*10^2 + 0*10^1 + 6*10^0$,

то есть слагаемых с различными степенями числа 10. По основанию степени - числу десять - система называется десятичной. Другие позиционные системы счисления отличаются только числом в основании степени.

При написании программ чаще всего используют десятичную, шестнадцатеричную (основание шестнадцать), восьмеричную (основание восемь) и двоичную (основание два) системы. Число различных знаков - цифр, используемых для записи чисел - в каждой системе равно основанию данной системы счисления.

0,1	- цифры двоичной системы
0,1,2,3,4,5,6,7	- цифры восьмеричной системы
0,1,2,3,4,5,6,7,8,9	- цифры десятичной системы
0,1,2,3,4,5,6,7,8,9,A,B,B,C,D,E,F	- цифры шестнадцатеричной системы

В шестнадцатеричной системе "обычных" десятичных цифр недостаточно, и для обозначения цифр, больших девяти, используются заглавные латинские буквы A,B,C,D,E,F.

В дальнейшем везде, где это необходимо, мы будем указывать основание системы счисления индексом рядом с числом: 95_{10} - в десятичной системе, $27F_{16}$ - в шестнадцатеричной системе, 6752_8 - в восьмеричной системе, 1000111_2 - в двоичной системе.

Существует множество непозиционных систем счисления, в которых числа изображаются и называются по своим правилам. Для римской системы чисел характерны сопоставление отдельного знака каждому большому числу (V - пять, X - десять, L - пятьдесят, C - сто, M - тысяча), повторение знака столько раз, сколько таких чисел во всем числе (III - три, XX - двадцать), отдельные правила для предшествующих и последующих чисел (IV - четыре, VI - шесть, IX - девять). Множество непозиционных систем счисления связано с традиционными способами измерения конкретных величин - времени (секунда, минута, час, сутки, неделя, месяц, год), длины (дюйм, фут, ярд, миля, аршин, сажень), массы (унция, фунт), денежных единиц. Выполнение арифметических действий в таких системах представляет собой крайне сложную задачу.

Приведём пример самой простой из возможных систем счисления – унарную. В ней имеется всего одна цифра 1. В унарной системе счисления число 1 изображается как 1, число 2 изображается как 11, число 3 как 111, число 4 как 1111, и так далее. Первоначально вместо единицы использовались палочки (помните детский сад?), поэтому такая система счисления иногда называется палочковой. Как ни странно, она является позиционной.

Позиционные системы счисления с основанием 2 и более удобны для алгоритмизации математических операций с числами (вспомните способ сложения и умножения "столбиком"). Двоичная система является естественным способом кодирования информации в компьютере, когда сообщение представляется набором нулей ("0" - нет сигнала на линии) и единиц ("1" - есть сигнал на линии). Для обозначения двоичных цифр применяется термин "бит", являющийся сокращением английского словосочетания "двоичная цифра" (BInary digiT).

Архитектура компьютера накладывает существенное ограничение на длину информации, обрабатываемой за одну операцию. Эта длина измеряется количеством двоичных разрядов и называется разрядностью. С помощью восьми двоичных разрядов можно представить 2^8=256 целых чисел. Порция информации размером 8 бит (8-ми битовое число) служит основной единицей измерения компьютерной информации и называется байтом (byte). Как правило, передача информацией внутри компьютера и между компьютерами идет порциями, кратными целому числу байт.

Машинным словом называют порцию данных, которую процессор компьютера может обработать за одну операцию (микрокоманду). Первые персональные компьютеры были 16-разрядными, т.е. работали с 16-битными (двухбайтными) словами. Поэтому операционные системы для этих компьютеров также были 16-разрядными. Например, MS DOS. Операционные системы для персональных компьютеров следующих поколений были 32-разрядны (Windows® ’95/’98/NT/ME/2000/XP, Linux, MacOS®), так как предназначались для использования с 32-разрядными процессорами. Современные операционные системы либо 32-разрядны, либо даже 64-разрядны (версии для 64-разрядных процессоров).

Представление чисел в двоичной и шестнадцатеричной системах счисления, а также преобразование из одной системы в другую часто необходимо при программировании аппаратуры для измерений, контроля и управления с помощью портов ввода-вывода, цифро-аналоговых и аналого-цифровых преобразователей.

Двоичное представление положительных целых чисел

Целые числа в компьютере обычно кодируются в двоичном коде, то есть в двоичной системе счисления. Например, число 5 можно представить в виде $5 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 101_2$.

Показатель системы счисления принято записывать справа снизу около числа.

Аналогично, $6 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 110_2$, $2 = 10_2$, $4 = 100_2$, $8 = 1000_2$, $9 = 1001_2$, и так далее.

Всё очень похоже на обозначение чисел в десятичной системе счисления:

$153 = 1 \cdot 10^2 + 5 \cdot 10^1 + 3 \cdot 10^0$. Но только в качестве основания системы счисления используется число $2 = 10_2$. У чисел, записанных в десятичной системе счисления, индекс 10 обычно не пишется, но его можно писать. Так что $2 = 2_{10}$, $10 = 10_{10}$, и так далее.

В двоичной арифметике всего две цифры, 0 и 1. Двоичный код положительного целого числа – это коэффициенты разложения числа по степеням двойки.

Умножение числа на двоичное десять, то есть на $10_2 = 2$, приводит к дописыванию справа нуля в двоичном представлении числа. Умножение на двоичное сто, то есть на $100_2 = 4$ - дописыванию двух нулей. И так далее.

Целочисленное деление на 10_2 с отбрасыванием остатка производится путём отбрасывания последнего (младшего) бита, деление на 100_2 - отбрасывания двух последних бит, и так далее.

Обычно такие операции называют побитовыми сдвигами на n бит влево (умножение на 2^n) или вправо (целочисленное деление на 2^n).

Сложение двоичных чисел можно производить “в столбик” по полной аналогии со сложением десятичных чисел. Единственное отличие – то, что в двоичной арифметике только две цифры, 0и1, а не десять цифр (от 0 до 9) как в десятичной. Поэтому если в десятичной арифметике единицу более старшего разряда даёт, к примеру, сложение 1 и 9, то в двоичной арифметике её

даст сложение 1 и 1. То есть
$1_2+1_2=10_2$
(в десятичной системе это равенство выглядит как 1+1=2). Аналогично, $10_2+10_2=100_2$, и так далее.

Примеры сложения “в столбик”:

$$\begin{array}{r} 0110_2 \\ +\ 1011_2 \\ \hline 10001_2 \end{array} \qquad \begin{array}{r} 1100_2 \\ +\ 0010_2 \\ \hline 1110_2 \end{array} \qquad \begin{array}{r} 111_2 \\ +\ 001_2 \\ \hline 1000_2 \end{array}$$

Совершенно аналогично выполняется умножение:

$$\begin{array}{r} 101_2 \\ \times\ 11_2 \\ \hline 101 \\ +\ 101 \\ \hline 1111_2 \end{array}$$

В машинной реализации целочисленного умножения используют побитовые сдвиги влево и сложения. Поскольку эти алгоритмы очень просты, они реализуются аппаратно.

Двоичное представление отрицательных целых чисел. Дополнительный код

Старший бит в целых без знака имеет обычный смысл, в целых со знаком – для положительных чисел всегда равен 0. В отрицательных числах старший бит всегда равен 1. В примерах для простоты мы будем рассматривать четырехбитную арифметику. Тогда в качестве примера целого положительного числа можно привести 0110_2.
Для хранения отрицательных чисел используется дополнительный код. Число (– *n*), где n положительно, переводится в число *n2=−n* по следующему алгоритму:

- этап 1: сначала число *n* преобразуется в число *n*1 путём преобразования $n \to n1$, во время которого все единицы числа *n* заменяются нулями, а нули единицами, то есть $1 \to 0, \quad 0 \to 1$;
- этап 2: перевод $n1 \to n2 = n1 + 1$, то есть к получившемуся числу *n1* добавляется единица младшего разряда.

Надо отметить, что дополнительный код отрицательных чисел зависит от разрядности. Например, код числа (–1) в четырёхразрядной арифметике будет 1111_2, а в 8-разрядной арифметике будет 11111111_2. Коды числа (–2) будут 1110_2 и 11111110_2, и так далее.
Для того, чтобы понять причину использования дополнительного кода, рассмотрим сложение чисел, представленных в дополнительном коде.

Сложение положительного и отрицательного чисел
Рассмотрим, чему равна сумма числа 1 и числа –1, представленного в дополнительном коде. Сначала переведём в дополнительный код число –1. При этом $n=1_{10}$, $n_2=-1_{10}$.
Этап1: $n=1_{10}=0001_2 \to n1=1110_2$;
Этап2: $n2=1110_2+1=1111_2$;
Таким образом, в четырёхбитном представлении $-1_{10}=1111_2$.
Проверка:

$n2+n=10000_2$. Получившийся пятый разряд, выходящий за пределы четырехбитной ячейки, отбрасывается, поэтому в рамках четырехбитной арифметики получается $n2+n=0000_2=0$.
Аналогично
$n=2_{10}=0010_2 \rightarrow n1=1101_2$; $n2=1110_2$;
$n=3_{10}=0011_2 \rightarrow n1=1100_2$; $n2=1101_2$;
$n=4_{10}=0100_2 \rightarrow n1=1011_2$; $n2=1100_2$;
Очевидно, во всех этих случаях $n2+n=0$.

Что будет, если мы сложим 3_{10} и -2_{10} (равное 1110_2, как мы уже знаем)?

$$\begin{array}{r} 0011_2 \\ + \quad 1110_2 \\ \hline 10001_2 \end{array}$$

После отбрасывания старшего бита, выходящего за пределы нашей четырёхбитовой ячейки, получаем $0011_2 + 1110_2=0001_2$, то есть $3_{10} + (-2_{10})=1_{10}$, как и должно быть.

Сложение отрицательных чисел
$(-1)+(-1)=1111_2+1111_2=11110_2 \rightarrow 1110_2$ из-за отбрасывания лишнего старшего бита, выходящего за пределы ячейки. Поэтому $(-1)+(-1)=1110_2=-2$.
Вычитание положительных чисел осуществляется путём сложения положительного числа с отрицательным, получившимся из вычитаемого в результате его перевода в дополнительный код.
Приведенные примеры иллюстрируют тот факт, что сложение положительного числа с отрицательным, хранящимся в дополнительном коде, или двух отрицательных, может происходить аппаратно (что значит очень эффективно) на основе крайне простых электронных устройств.

Проблемы целочисленной машинной арифметики

Несмотря на достоинства в двоичной машинной (аппаратной) арифметике имеются очень неприятные особенности, возникающие из-за конечной разрядности машинной ячейки.
Проблемы сложения положительных чисел
Пусть $a=3_{10}=0011_2$; $b=2_{10}=0010_2$; $a+b=0101_2=5_{10}$, то есть все в порядке.
Пусть теперь $a=6_{10}=0110_2$, $b=5_{10}=0101_2$. Тогда $a+b=1011_2=-3_2$.
То есть сложение двух положительных чисел может дать отрицательное, если результат сложения превышает максимальное положительное число, выделяемое под целое со знаком для данной разрядности ячеек! В любом случае при выходе за разрешённый диапазон значений результат оказывается неверным.
Если у нас беззнаковые целые, проблема остается в несколько измененном виде. Сложим $8_{10}+8_{10}$ в двоичном представлении. Поскольку $8_{10}=1000_2$, тогда $8_{10}+8_{10}=1000_2+1000_2=10000_2$. Но лишний бит отбрасывается, и получаем 0. Аналогично в четырёхбитной арифметике, $8_{10}+9_{10}=1_{10}$, и т.д.
Как уже говорилось, умножение двоичных чисел осуществляется путем сложений и сдвигов по алгоритму, напоминающему умножение “в столбик”, но гораздо более простому, так как умножить надо только на 0 или 1. При целочисленном умножении выход за пределы разрядности ячейки происходит гораздо чаще, чем при сложении или вычитании. Например, $110_2 \times 101_2=110_2 \times 100_2+110_2 \times 1_2=11000_2+110_2=11100_2$. Если наша ячейка четырехразрядная, произойдет выход за ее пределы, и мы получим после отбрасывания лишнего бита $1110_2=-2_{10}<0$. Таким образом, умножение целых чисел легко может дать неправильный результат. В том числе – даже отрицательное число. Поэтому при работе с целочисленными типами данных следует обращать особое внимание на то, чтобы в программе не возникало ситуаций арифметического переполнения. Повышение разрядности целочисленных переменных

позволяет смягчить проблему, хотя полностью её не устраняет. Например, зададим переменные

```
byte m=10,n=10,k=10;
```

Тогда значения `m*n`, `m*k` и `n*k` будут лежать в разрешённом диапазоне -128..127. А вот `m*n + m*k` из него выйдет. Не говоря уж об `m*n*k`.

Если мы зададим

```
int m=10,n=10,k=10;
```

переполнения не возникнет даже для `m*n*k`. Однако, при `m=n=k=100` значение `m*n*k` будет равно 10^6, что заметно выходит за пределы разрешённого диапазона –32768..32767. Хотя `m*n`, `m*k` и `n*k` не будут за него выходить (но уже `4*m*n` за него выйдет). Использование типа `long` поможет и в этом случае. Однако уже значения `m=n=k=2000` (не такие уж большие!) опять приведут к выходу `m*n*k` за пределы диапазона. Хотя для `m*n` выход произойдёт только при значениях около 50000.

Вычисление факториала с помощью целочисленной арифметики даст удивительные результаты! В таких случаях лучше использовать числа с плавающей точкой.
Пример:

```
byte i=127, j=1, k;
k=(byte)(i+j);
System.out.println(k);
```

В результате получим число (-128). Если бы мы попробовали написать

```
byte i=127,j=1,k;
System.out.println(i+j);
```

то получили бы +128. Напомним, что значения величин типа byte перед проведением сложения преобразуются в значения типа int.

Шестнадцатеричное представление целых чисел и перевод из одной системы счисления в другую

Во время программирования различного рода внешних устройств, регистров процессора, битовыми масками, кодировке цвета, и так далее, приходится работать с кодами беззнаковых целых чисел. При этом использование десятичных чисел крайне неудобно из-за невозможности лёгкого сопоставления числа в десятичном виде и его двоичных бит. А использование чисел в двоичной кодировке крайне громоздко – получаются слишком длинные последовательности нулей и единиц. Программисты используют компромиссное решение – шестнадцатеричную кодировку чисел, где в качестве основания системы счисления выступает число 16. Очевидно, $16_{10} = 10_{16}$. В десятичной системе счисления имеется только 10 цифр: 0,1,2,3,4,5,6,7,8,9. А в 16-ричной системе счисления должно быть 16 цифр. Принято обозначать недостающие цифры заглавными буквами латинского алфавита: A,B,C,D,E,F. То есть $10_{10} = A = A_{16}$, $11_{10} = B$, $12_{10} = C$, $13_{10} = D$, $14_{10} = E$, $15_{10} = F$. Таким образом, к примеру, $FF_{16} = 15 \cdot 16^1 + 15 \cdot 16^0 = 255$.

В Java для того, чтобы отличать 16-ричные числа, как мы уже знаем, перед ними ставят префикс 0x: 0x*FF* обозначает FF_{16}, а 0x10 – это 10_{16}, то есть 16.

Число N может быть записано с помощью разных систем счисления. Например, в десятичной:

$N = A_n 10^n + ... + A_2 10^2 + A_1 10^1 + A_0 10^0$ ($A_n = 0 .. 9$)

или в двоичной:

$N = B_n 2^n + ... + B_2 2^2 + B_1 2^1 + B_0 2^0$ ($B_n = 0$ или 1)

или в шестнадцатеричной:

$N = C_n 16^n + ... + C_2 16^2 + C_1 16^1 + C_0 16^0$ ($C_n = 0 .. F$)

Преобразование в другую систему счисления сводится к нахождению соответствующих коэффициентов. Например, B_n по известным коэффициентам A_n – при переводе из десятичной системы в двоичную, или коэффициентов A_n по коэффициентам B_n - из двоичной системы в десятичную.

Преобразование чисел из системы с меньшим основанием в систему с большим основанием

Рассмотрим преобразование из двоичной системы в десятичную. Запишем число N в виде

$$N = B_n 2^n + ... + B_2 2^2 + B_1 2^1 + B_0 2^0 \qquad (B_n = 0 \text{ или } 1)$$

и будем рассматривать как алгебраическое выражение в десятичной системе. Выполним арифметические действия по правилам десятичной системы. Полученный результат даст десятичное представление числа N.

Пример:

Преобразуем 01011110_2 к десятичному виду. Имеем:

$01011110_2 = 0\cdot2^7+1\cdot2^6+0\cdot2^5+1\cdot2^4+1\cdot2^3+1\cdot2^2+1\cdot2^1+0\cdot2^0=$
$= 0 + 64 + 0 + 16 + 8 + 4 + 2 + 0 =$
$= 94_{10}$

Преобразование чисел из системы с большим основанием в систему с меньшим основанием

Рассмотрим его на примере преобразования из десятичной системы в двоичную. Нужно для известного числа N_{10} найти коэффициенты в выражении

$$N = B_n 2^n + ... + B_2 2^2 + B_1 2^1 + B_0 2^0 \qquad (B_n = 0 \text{ или } 1)$$

Воспользуемся следующим алгоритмом: в десятичной системе разделим число N на 2 с остатком. Остаток деления (он не превосходит делителя) даст коэффициент B_0 при младшей степени 2^0. Далее делим на 2 частное, полученное от предыдущего деления. Остаток деления будет следующим коэффициентом B_1 двоичной записи N. Повторяя эту процедуру до тех пор, пока частное не станет равным нулю, получим последовательность коэффициентов B_n.

Например, преобразуем 345_{10} к двоичному виду. Имеем:

	частное	остаток	B_i
345 / 2	172	1	B_0
172 / 2	86	0	B_1
86 / 2	43	0	B_2
43 / 2	21	1	B_3
21 / 2	10	1	B_4
10 / 2	5	0	B_5
5 / 2	2	1	B_6
2 / 2	1	0	B_7
1 / 2	0	1	B_8

$345_{10} = 101011001_2$

Преобразование чисел в системах счисления с кратными основаниями

Рассмотрим число N в двоичном и шестнадцатеричном представлениях.

$$N = B_n 2^n + ... + B_2 2^2 + B_1 2^1 + B_0 2^0 \qquad (B_i = 0 \text{ или } 1)$$
$$N = H_n 16^n + ... + H_2 16^2 + H_1 16^1 + H_0 16^0 \quad (H_i = 0 .. F_{16}, \text{ где } F_{16} = 15_{10})$$

Заметим, что $16 = 2^4$. Объединим цифры в двоичной записи числа группами по четыре. Каждая группа из четырех двоичных цифр представляет число от 0 до F_{16}, то есть от 0 до15_{10}. От группы к группе вес цифры изменяется в $2^4=16$ раз (основание 16-ричной системы). Таким

образом, перевод чисел из двоичного представления в шестнадцатеричное и обратно осуществляется простой заменой всех групп из четырех двоичных цифр на шестнадцатеричные (по одному на каждую группу) и обратно :

$0000_2 = 0_{16}$
$0001_2 = 1_{16}$
$0010_2 = 2_{16}$
$0011_2 = 3_{16}$
$0100_2 = 4_{16}$
$0101_2 = 5_{16}$
$0110_2 = 6_{16}$
$0111_2 = 7_{16}$
$1000_2 = 8_{16}$
$1001_2 = 9_{16}$
$1010_2 = A_{16}$
$1011_2 = B_{16}$
$1100_2 = C_{16}$
$1101_2 = D_{16}$
$1110_2 = E_{16}$
$1111_2 = F_{16}$

Например, преобразуем 1011010111_2 к шестнадцатеричному виду:

$1011010111_2 = 0010\ 1101\ 0111_2 = 2D7_{16}$

4.2. Побитовые маски и сдвиги

Оператор	Название	Пример	Примечание
~	Оператор побитового дополнения (побитовое “не”, побитовое отрицание)	`~i`	
^	Оператор “ побитовое исключающее или” (XOR)	`i^j`	
&	Оператор “побитовое и” (AND)	`i&j`	
|	Оператор “побитовое или” (OR)	`i|j`	

<<	Оператор левого побитового сдвига
>>>	Оператор беззнакового правого побитового сдвига
>>	Оператор правого побитового сдвига с сохранением знака отрицательного числа

&=	y&=x эквивалентно y=y&x
|=	y|=x эквивалентно y=y|x
^=	y^=x эквивалентно y=y^x
>>=	y>>=x эквивалентно y= y>>x

>>>=	y>>>=x эквивалентно y= y>>>x
<<=	y<<=x эквивалентно y= y<<x

Побитовые операции – когда целые числа рассматриваются как наборы бит, где 0 и 1 играют роли логического нуля и логической единицы. При этом все логические операции для двух чисел осуществляются поразрядно – k-тый разряд первого числа с k-тым разрядом второго. Для простоты мы будем рассматривать четырёхбитовые ячейки, хотя реально самая малая по размеру ячейка восьмибитовая и соответствует типу byte.

а) установка в числе **a** нужных бит в 1 с помощью маски **m** операцией **a|m** (арифметический, или, что то же, побитовый оператор OR).
Пусть число $\mathbf{a = a_3*2^3 + a_2*2^2 + a_1*2^1 + a_0*2^0}$, где значения $\mathbf{a_i}$ – содержание соответствующих бит числа (то есть либо нули , либо единицы).

a	a_3	a_2	a_1	a_0
m	0	1	0	1
a|m	a_3	1	a_1	1

Видно, что независимо от начального значения в числе a в результате нулевой и второй бит установились в единицу. Таким образом, операцию **OR** с маской можно использовать для установки нужных бит переменной в единицу, если нужные биты маски установлены в единицу, а остальные – нули.

б) установка в числе **a** нужных бит в 0 с помощью маски m операцией **a&m** (арифметический, или, что то же, побитовый оператор AND):

a	a_3	a_2	a_1	a_0
m	0	1	0	1
a&m	0	a_2	0	a_0

Видно, что независимо от начального значения в числе **a** в результате первый и третий бит установились в нуль. Таким образом, операцию **AND** с маской можно использовать для установки нужных бит переменной в ноль, если нужные биты маски установлены в ноль, а остальные – единицы.

в) инверсия (замена единиц на нули, а нулей на единицы) в битах числа **a** , стоящих на задаваемых маской **m** местах, операцией **a^m** (арифметический, или, что то же, побитовый оператор **XOR**):

a	1	1	0	0
m	0	1	0	1
a^m	1	0	0	1

Видно, что если в бите, где маска **m** имеет единицу, у числа **a** происходит инверсия: если стоит 1, в результате будет 0, а если 0 – в результате будет 1. В остальных битах значение не меняется.
Восстановление первоначального значения после операции **XOR –** повторное **XOR** с той же битовой маской:

a^m	1	0	0	1
m	0	1	0	1
(a^m)^m	1	1	0	0

Видно, что содержание ячейки приняло то же значение, что было первоначально в ячейке a. Очевидно, что всегда (a ^ m) ^ m = a, так как повторная инверсия возвращает первоначальные значения в битах числа. Операция **XOR** часто используется в программировании для инверсии цветов частей экрана с сохранением в памяти только информации о маске. Повторное **XOR** с той же маской восстанавливает первоначальное изображение. - Имеется команда перевода вывода графики в режим XOR при рисовании, для этого используется команда graphics.setXORMode(цвет).
Ещё одна область, где часто используется эта операция – криптография.

Инверсия всех битов числа осуществляется с помощью побитового отрицания **~a**.

Побитовые сдвиги “<<”, “>>” и “>>>” приводят к перемещению всех бит ячейки, к которой применяется оператор, на указанное число бит влево или вправо. Сначала рассмотрим действие операторов на положительные целые числа.

Побитовый сдвиг на n бит влево m<<n эквивалентен быстрому целочисленному умножению числа m на 2^n. Младшие биты (находящиеся справа), освобождающиеся после сдвигов, заполняются нулями. Следует учитывать, что старшие биты (находящиеся слева), выходящие за пределы ячейки, теряются, как и при обычном целочисленном переполнении.

Побитовые сдвиги на n бит вправо m>>n или m>>>n эквивалентны быстрому целочисленному делению числа m на 2^n. При этом для положительных m разницы между операторами “>>” и “>>>” нет.

Рассмотрим теперь операции побитовых сдвигов для отрицательных чисел m. Поскольку они хранятся в дополнительном коде, их действие нетривиально. Как и раньше, для простоты будем считать, что ячейки четырёхбитовые, хотя на деле побитовые операции проводятся только для ячеек типа int или long, то есть для 32-битных или 64-битных чисел.

Пусть m равно -1. В этом случае $m=1111_2$. Оператор m<<1 даст $m=11110_2$, но из-за четырёхбитности ячейки старший бит теряется, и мы получаем $m=1110_2$=-2. То есть также получается полная эквивалентность умножению m на 2^n.

Иная ситуация возникает при побитовых сдвигах вправо. Оператор правого сдвига “>>” для положительных чисел заполняет освободившиеся биты нулями, а для отрицательных – единицами. Легко заметить, что этот оператор эквивалентен быстрому целочисленному делению числа m на 2^n как для положительных, так и для отрицательных чисел. Оператор m>>>n, заполняющий нулями освободившиеся после сдвигов биты, переводит отрицательные числа в положительные. Поэтому он не может быть эквивалентен быстрому делению числа на 2^n. Но иногда такой оператор бывает нужен для манипуляции с наборами бит, хранящихся в числовой ячейке. Само значение числа в этом случае значения не имеет, а ячейка используется как буфер соответствующего размера.

Например, можно преобразовать последовательность бит, образующее некое целое значение, в число типа float методом Float.intBitsToFloat(*целое значение*) или типа double методом Double.intBitsToDouble (*целое значение*). Так, Float.intBitsToFloat(0x7F7FFFFF) даст максимальное значение типа float.

4.3. Двоичное представление вещественных чисел

Двоичные дроби

Целое число 0101_2 можно представить в виде $0101_2=0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$

Аналогично можно записать двоичную дробь:

$11.0101_2 = 1*2^1 + 1*2^0 + 0*2^{-1} + 1*2^{-2} + 0*2^{-3} + 1*2^{-4}$

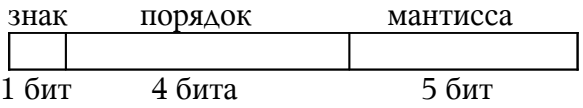
Заметим, что сдвиг двоичной точки на n разрядов вправо (чаще говорят о сдвиге самого числа влево) эквивалентен умножению числа на $(10_2)^n = 2^n$. Сдвиг точки влево (то есть сдвиг самого числа вправо) – делению на 2^n.

Мантисса и порядок числа

Рассмотрим сначала упрощенную схему хранения чисел в формате с плавающей точкой (floating point), несколько отличающуюся от реальной.

Число `x` с плавающей точкой может быть представлено в виде $x=s*m*2^p$. Множитель s – знак числа. Второй множитель m называется мантиссой, а число `p` – порядком числа.

Для простоты рассмотрим 10-битовую ячейку, состоящую из трёх независимых частей:



Первым идёт знаковый бит. Если он равен 0, число положительно, если равен 1 – отрицательно. Набор бит, хранящийся в мантиссе, задает положительное число m, лежащее в пределах $1 \le m < 2$. Оно получается из нашего двоичного числа путем переноса двоичной точки на место после первой значащей цифры числа. Например, числа 1.0101_2, 10.101_2 и 0.10101_2 и имеют одну и ту же мантиссу, равную 1.0101_2. При этом следующая за ведущей единицей точка в ячейке, выделяемой под мантиссу, не хранится - она подразумевается. То есть мантиссы приведённых чисел будут храниться в виде 10101_2.
Число сдвигов двоичной точки (с учетом знака) хранится в части ячейки, выделяемой под порядок числа. В нашем примере числа 1.0101_2, 10.101_2 и 0.10101_2 будут иметь порядки 0, 1 и -1, соответственно. При перемножении чисел их мантиссы перемножаются, а порядки складываются. При делении – мантиссы делятся, а порядки вычитаются. И умножение, и деление мантисс происходит по тем же алгоритмам, что и для целых чисел. Но при выходе за размеры ячейки отбрасываются не старшие, а младшие байты. В результате каждая операция умножения или деления даёт результат, отличающийся от точного на несколько значений младшего бита мантиссы. Аналогичная ситуация с потерей младших бит возникает при умножениях и делениях. Ведь если в ячейках для чисел данного типа хранится `k` значащих цифр числа, то при умножении двух чисел точный результат будет иметь `2k` значащих цифр, последние `k` из которых при записи результата в ячейку будут отброшены даже в том случае, если они сохранялись при вычислениях. А при делении в общем случае при точных вычислениях должна получаться бесконечная периодическая двоичная дробь, так что даже теоретически невозможно провести эти вычисления без округлений. С этим связана конечная точность вычислений на компьютерах при использовании формата с “плавающей точкой”. При этом чем больше двоичных разрядов выделяется под мантиссу числа, тем меньше погрешность в такого рода операциях.
Замечание: *системы символьных вычислений (или, что то же, – аналитических вычислений, или, что то же, системы компьютерной алгебры) позволяют проводить точные численные расчеты с получением результатов в виде формул. Однако они выполняют вычисления на много порядков медленнее, требуют намного больше ресурсов и не могут работать без громоздкой среды разработки. Поэтому для решения большинства практически важных задач они либо неприменимы, либо их использование нецелесообразно.*
При сложении или вычитании сначала происходит приведение чисел к одному порядку:

мантисса числа с меньшим порядком делится на 2^n, а порядок увеличивается на n, где n – разница в порядке чисел. При этом деление на 2^n осуществляется путем сдвига мантиссы на n бит вправо, с заполнением освобождающихся слева бит нулями. Младшие биты мантиссы, выходящие за пределы отведенной под нее части ячейки, теряются.

Пример:

сложим числа 11.011_2 и 0.11011_2. Для первого числа мантисса 1.1011_2, порядок 1, так как $11.011_2=1.1011_2*(10_2)^1$. Для второго – мантисса 1.1011_2, порядок -1, так как $0.11011_2 =1.1011_2*(10_2)^{-1}$. Приводим порядок второго числа к значению 1, сдвигая мантиссу на 2 места вправо, так как разница порядков равна 2:

$0.11011_2 = 0.011011_2* (10_2)^1$.

Но при таком сдвиге теряется два последних значащих бита мантиссы (напомним, хранится 5 бит), поэтому получаем приближенное значение $0.0110_2* (10_2)^1$. Из-за чего в машинной арифметике получается

$1.1011_2*(10_2)^1 + 0.011011_2*(10_2)^1 = (1.1011_2 + 0.011011_2)*(10_2)^1 \approx (1.1011_2 + 0.0110_2)*(10_2)^1 =10.0001_2* (10_2)^1 \approx 1.0000_2*(10_2)^2$

вместо точного значения $1.0000111_2*(10_2)^2$.

Таким образом, числа в описанном формате являются на деле рациональными, а не вещественными. При этом операции сложения, вычитания, умножения и деления выполняются с погрешностями, тем меньшими, чем больше разрядность мантиссы. Число двоичных разрядов, отводимых под порядок числа, влияет лишь на допустимый диапазон значений чисел, и не влияет на точность вычислений.

Научная нотация записи вещественных чисел

При записи программы в текстовом файле или выдачи результатов в виде “плоского текста” (plain text) невозможна запись выражений типа $1.5 \cdot 10^{14}$. В этом случае используется так называемая научная нотация, когда вместо основания 10 пишется латинская буква E (сокращение от Exponent – экспонента). Таким образом, $1.5 \cdot 10^{14}$ запишется как 1.5E14, а $0.31 \cdot 10^{-7}$ как 0.31E-7. Первоначально буква E писалась заглавной, что не вызывало проблем. Однако с появлением возможности набора текста программы в нижнем регистре стали использовать строчную букву e, которая в математике используется для обозначения основания натуральных логарифмов. Запись вида 3e2 легко воспринять как $3e^2$, а не $3 \cdot 10^2$. Поэтому лучше использовать заглавную букву.

Литерные константы для вещественных типов по умолчанию имеют тип double. Например, 1.5 , -17E2 , 0.0 . Если требуется ввести литерную константу типа float, после записи числа добавляют постфикс f (сокращение от “float”): 1.5f , -17E2f , 0.0f .

Минимальное по модулю не равное нулю и максимальное значение типа float можно получить с помощью констант

Float.MIN_VALUE - равна 2^{-149}

Float.MAX_VALUE - равна $(2-2^{-23})\cdot 2^{127}$

Аналогичные значения для типа double - с помощью констант

Double.MIN_VALUE - равна 2^{-1074}

Double.MAX_VALUE - равна $(2-2^{-52})\cdot 2^{1023}$.

Стандарт IEEE 754 представления чисел в формате с плавающей точкой*

**Этот параграф является необязательным и приводится в справочных целях*

В каком виде на самом деле хранятся числа в формате с плавающей точкой? Ответ даёт стандарт IEEE 754 (другой вариант названия IEC 60559:1989), разработанный для электронных счётных устройств. В этом стандарте предусмотрены три типа чисел в формате с плавающей точкой, с которыми могут работать процессоры: real*4, real*8 и real*10. Эти числа занимают 4, 8 и 10 байт, соответственно. В Java типу real*4 соответствует float, а типу real*8 соответствует

double. Тип real*10 из распространённых языков программирования используется только в диалектах языка PASCAL, в Java он не применяется.
Число r представляется в виде произведения знака s, мантиссы m и экспоненты 2^{p-d} :

$$r=s*m*2^{p-d}.$$

Число p называется порядком. Оно может меняться для разных чисел. Значение d, называемое сдвигом порядка, постоянное для всех чисел заданного типа. Оно примерно равно половине максимального числа p_{max}, которое можно закодировать битами порядка. Точнее, d= (p_{max}+1)/2-1.
Для чисел real*4: p_{max} = 255, d= 127.
Для чисел real*8: p_{max} = 2047, d= 1023.
Для чисел real*10: p_{max} =32767, d=16383.
Число называется *нормализованным* в случае, когда мантисса лежит в пределах 1≤m<2. В этом случае первый бит числа всегда равен единице. Максимальное значение мантиссы достигается в случае, когда все её биты равны 1. Оно меньше 2 на единицу младшего разряда мантиссы, то есть с практически важной точностью может считаться равным 2.

Согласно стандарту IEEE 754 все числа формата с плавающей точкой при значениях порядка в диапазоне от 1 до p_{max}-1 хранятся в нормализованном виде. Такое представление чисел будем называть ***базовым***. Когда порядок равен 0, применяется несколько другой формат хранения чисел. Будем называть его ***особым***. Порядок p_{max} резервируется для кодировки нечисловых значений, соответствующее представление будем называть ***нечисловым***. Об особом и нечисловом представлениях чисел будет сказано чуть позже.

Размещение чисел в ячейках памяти такое:

Тип	**Байт1**	**Байт2**	**Байт3**	**Байт4**	…	**Байт8**	**Байт9**	**Байт10**
real*4	sppp pppp	pmmm mmmm	mmmm mmmm	mmmm mmmm				
real*8	sppp pppp	pppp mmmm	mmmm mmmm	mmmm mmmm		mmmm mmmm		
real*10	sppp pppp	pppp pppp	1mmm mmmm	mmmm mmmm		mmmm mmmm	mmmm mmmm	mmmm mmmm

Буква s обозначает знаковый бит; p – биты двоичного представления порядка, m – биты двоичного представления мантиссы. Если знаковый бит равен нулю, число положительное, если равен единице – отрицательное. В числах real*4 (float) и real*8 (double) при базовом представлении ведущая единица мантиссы подразумевается, но не хранится, поэтому реально можно считать, что у них под мантиссу отведено не 23 и 52 бита, которые реально хранятся, а 24 и 53 бита. В числах real*10 ведущая единица мантиссы реально хранятся, и мантисса занимает 64 бит. Под порядок в числах real*4 отведено 8 бит, в числах real*8 отведено 11 бит, а в числах real*10 отведено 15 бит.

Тип IEEE 754	**Тип Java**	Число бит мантиссы	Число бит порядка	Сдвиг порядка
real*4	float	23+ подразумевается 1 ведущий бит	8	127
real*8	double	52+ подразумевается 1 ведущий бит	11	1023
real*10	-	64	15	16383

Чему равны минимальное и максимальное по модулю числа при их базовом представлении?

Минимальное значение достигается при минимальном порядке и всех нулевых битах мантиссы (за исключением ведущего), то есть при m=1 и p=1. Значит, минимальное значение равно 2^{1-d}.

Максимальное значение достигается при максимальном порядке и всех единичных

битах мантиссы, то есть при m≈2 и p= p_{max}-1. Значит, максимальное значение примерно равно $2 * 2^{p_{\max}-1-d} = 2^{p_{\max}-d}$

При значениях порядка в диапазоне от 1 до p_{max}-1 ***базовое представление*** позволяет закодировать

- числа real*4 примерно от 2.350989E-38 до 3.402824E38,
- числа real*8 примерно от 2.225074E-308 до 1.797693E308,
- числа real*10 примерно от 3.362103E-4932 до 1.189731E4932.

В случае, когда порядок равен 0 или p_{max}, используется ***особое*** представление чисел, несколько отличающееся от базового.

Если все биты порядка равны 0, но мантисса отлична от нуля, то порядок считается равным 1 (а не 0), а вместо единицы в качестве подразумеваемой ведущей цифры используется ноль. Это *ненормализованное* представление чисел. Максимальное значение мантиссы в этом случае на младший бит мантиссы меньше 1. Так что максимальное значение числа в особой форме представления равно (1- младший бит мантиссы)*2^{1-d}. То есть верхний предел диапазон изменения чисел в этом представлении смыкается с нижним диапазоном изменения чисел в базовом представлении.
Минимальное ненулевое значение мантиссы в особом представлении равно 2^{-n}, где n – число бит мантиссы после двоичной точки.
Минимальное отличное от нуля положительное число для некоторого типа чисел с плавающей точкой равно 2^{1-d-n}.

Таким образом, ***особое представление*** позволяет закодировать

- числа real*4 примерно от 1.401298E-45 до 2.350989E-38,
- числа real*8 примерно от 4.940656E-324 до 2.225074E-308,
- числа real*10 примерно от 3.6451995E-4951 до 3.362103E-4932.

Специальный случай особого представления – когда и порядок и мантисса равны нулю. Это значение обозначает машинный ноль. В соответствии со стандартом IEEE 754 имеются +0 и -0. Но во всех известных автору языках программирования +0 и -0 при вводе-выводе и сравнении чисел отождествляются.

Нечисловое представление соответствует случаю, когда p=p_{max}, то есть все биты порядка равны 1. Такое “число” в зависимости от значения мантиссы обозначает одно из трех специальных нечиселовых значений, которые обозначают как Inf (Infinity - “бесконечность”), NaN (Not a Number - “не число”), Ind (Indeterminate – “неопределённость”). Эти значения появляются при переполнениях и неопределённостях в вычислениях. Например, при делении 1 на 0 получается Inf, а при делении 0 на 0 получается Ind. Значение NaN может получаться при преобразовании строки в число, взятии логарифма от отрицательного числа, тригонометрической функции от бесконечности и т.п.

Значение Inf соответствует нулевым битам мантиссы.Согласно IEEE 754 бесконечность имеет знак. Если знаковый бит 0 это + Inf, если знаковый бит 1 это –Inf.

Значение Ind кодируется единицей в знаковом бите и битами мантиссы, равными 0 во всех разрядах кроме старшего (реального, а не подразумеваемого), где стоит 1. Все остальные сочетания знакового бита и мантиссы отведены под величины NaN. Значения NaN бывают двух типов – вызывающие возбуждение сигнала о переполнении (Signaling NaN) и не вызывающие (Quiet NaN). Значения обоих этих типов могут быть “положительными” (знаковый бит равен нулю) и “отрицательными” (знаковый бит равен единице).

В современных языках программирования поддерживается только часть возможностей, реализованных в процессорах в соответствии со стандартом IEEE 754. Например, в Java значения бесконечности различаются как по знаку, так и по типу: имеются

`Float.NEGATIVE_INFINITY`, `Float.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY`, `Double.POSITIVE_INFINITY`.

Но значение Ind вообще не употребляется и отождествляется с NaN, хотя `Float.NaN` и `Double.NaN` различаются.

Числа в формате с плавающей точкой занимают следующие диапазоны значений:

Название значения	**s (знак)**	**p (порядок)**	**m (мантисса)**
-NaN (отрицательное не-число)	1	11..11	11..11 : 10..01
Indeterminate (неопределённость)	1	11..11	10..00
Signaling -NaN (отрицательное не-число, возбуждающее ошибку)	1	11..11	01..11 : 00..01
-Infinity (отрицательное переполнение, плюс бесконечность)	1	11..11	00..00
Отрицательное нормализованное	1	11..10 : 00..01	11..11 : 00..00
Отрицательное ненормализованное	1	00..00	11..11 : 00..01
-0	1	00..00	00..00
+0	0	00..00	00..00
Положительное ненормализованное	0	00..00	00..01 : 11..11
Положительное нормализованное	0	00..01 : 11..10	00..00 : 11..11
+Infinity (положительное переполнение, плюс бесконечность)	0	11..11	00..00
Signaling +NaN (положительное не-число, возбуждающее ошибку)	0	11..11	00..01 : 01..11
Quiet +NaN (положительное не-число)	0	11..11	10..00 : 11..11

Имеются методы оболочечных классов, позволяющие преобразовывать наборы бит, хранящихся в ячейках типа int, в значения float, и наоборот – значения типа float в их битовое представление типа int. При этом содержание ячеек не меняется – просто

содержащиеся в ячейках наборы бит начинают по-другому трактоваться.
Аналогичные операции существуют и для значений типа long и double:
Float.intBitsToFloat(*значение типа int*)
Double.longBitsToDouble(*значение типа long*)

Float.floatToIntBits(*значение типа float*)
Double.doubleToLongBits(*значение типа double*)

Например, Float.intBitsToFloat(0x7F7FFFFF) даст значение, равное `Float.MAX_VALUE`,
Float.intBitsToFloat(0x7F800000) – значение `Float.POSITIVE_INFINITY`,
Float.intBitsToFloat(0xFF800000) – значение `Float.NEGATIVE_INFINITY`.
Если аргумент метода Float.intBitsToFloat лежит в пределах от 0xF800001 до 0xF800001, результатом будет `Float.NaN`.

Следует подчеркнуть, что данные операции принципиально отличаются от “обычных” преобразований типов, например, из int в float или из double в long. При “обычных” преобразованиях значение числа не меняется, просто меняется форма хранения этого значения и, соответственно, наборы битов, которыми кодируется это значение. Причём может измениться размер ячейки (скажем, при преобразовании значений int в значения double). А при рассматриваемых в данном разделе операциях сохраняется набор бит и размер ячейки, но меняется тип, который приписывается этому набору.

Краткие итоги по главе 4

- ✓ Отрицательные целые числа кодируются в дополнительном коде.
- ✓ При сложении, вычитании и, особенно, умножении целых чисел может возникать выход за пределы допустимого диапазона. В результате у результата может получиться значение, имеющее противоположный по сравнению с правильным результатом знак. Или нуль при сложении чисел одного знака.
- ✓ Побитовая маска AND (оператор "&") служит для сбрасывания в 0 тех битов числа, где в маске стоит 0, остальные биты числа не меняются. Побитовая маска OR (оператор "|") служит для установки в 1 тех битов числа, где в маске стоит 1, остальные биты числа не меняются.
- ✓ Побитовая маска XOR (оператор "^") служит для инверсии тех битов числа, где в маске стоит 1 (единицы переходят в нули, а нули – в единицы), остальные биты числа не меняются. Имеется команда перевода вывода графики в режим XOR при рисовании, для этого используется команда graphics.setXORMode(*цвет*).
- ✓ Инверсия всех битов числа осуществляется с помощью побитового отрицания **~a**.
- ✓ Побитовые сдвиги "<<", ">>" и ">>>" приводят к перемещению всех бит ячейки, к которой применяется оператор, на указанное число бит влево или вправо. Причём m<<n является очень быстрым вариантом операции $m \cdot 2^n$, а m>>n – целочисленному делению m на 2^n.
- ✓ Литерные константы для вещественных типов по умолчанию имеют тип double. Например, 1.5 , -17E2 , 0.0 . Если требуется ввести литерную константу типа float, после записи числа добавляют постфикс f (сокращение от "float"): 1.5f , -17E2f , 0.0f .
- ✓ Число в формате с плавающей точкой состоит из знакового бита, мантиссы и порядка. Все операции с числами такого формата сопровождаются накоплением ошибки порядка нескольких младших битов мантиссы ненормализованного результата.
- ✓ При проведении умножения и деления чисел в формате с плавающей точкой не происходит катастрофической потери точности результата. А при сложениях и вычитаниях такая потеря может происходить в случае, когда вычисляется разность двух значений, к которым прибавляется малая относительно этих значений добавка. Число порядков при потере точности равно числу порядков, на которые отличаются эти значения от малой добавки.

Типичные ошибки:

- При использовании целых типов не знают или забывают, что при выходе за пределы диапазона соответствующие биты отбрасываются без всякой диагностики переполнения. И что получившийся результат может быть неправильного знака или нуль. Особенно часто переполнение возникает при умножении нескольких целых величин друг на друга.
- Написание численных алгоритмов с катастрофической потерей точности.
- Сравнение двух чисел в формате с плавающей точкой на *равенство* или *неравенство* в случае, когда в точной арифметике они должны быть равны.

Задания

- Создать проект с графическим пользовательским интерфейсом, в котором имеется два пункта ввода и радиогруппа с выбором варианта для каждого из целочисленных типов, а также кнопки JButton с надписями "Сложить" и "Умножить". При выборе соответствующего варианта в пунктах ввода должны возникать случайным образом генерируемые числа, лежащее в пределах допустимых значений для этого типа. При

нажатии на кнопку содержимое пунктов ввода и результат действий должны быть преобразованы в значения соответствующего типа, и результат выполнения сложения или умножения должен быть показан в метке. Проверить наличие проблемы арифметического переполнения для разных значений каждого типа. Выяснить, в каком случае чаще происходит переполнение – при сложении или умножении. И насколько часто встречаются такие значения одного знака, чтобы в результате получился результат противоположного знака. Вручную ввести в пункты ввода такие значения одного знака, чтобы в результате получилось нулевое значение.

- Написать приложение с графическим пользовательским интерфейсом, в котором вычисляется выражение 1+x и присваиваются переменной float f, а также вычисляется выражение 1+y и присваиваются переменной double d. Величины x типа float и y типа double вводится пользователем с помощью пунктов ввода. Вывести в метку jLabel1 разность f-1 , и в метку jLabel2 разность d-1 . Провести вычисления для x и y, меняющихся в пределах от 1E-3 до 1E-18 . Объяснить результаты.
- * *Для студентов технических вузов и физико-математических факультетов университетов*. Написать приложение с графическим пользовательским интерфейсом, в котором вычисляются выражения f1(x) и f2(x), где f1(x)=a(x)-b(x), $a(x)=e^x$, b(x)=1+x, и $f2(x)=\frac{1}{2}x^2+\frac{1}{6}x^3+\frac{1}{24}x^4$. Поскольку f2(x) состоит из первых членов разложения f1(x) в ряд, то f1(x) и f2(x) должны быть примерно равны . Требуется сравнить значения выражения f1(x) и f2(x) при различных x. Все вычисления сначала проводить для переменных и функций типа float, а затем - для переменных и функций типа double. Величина x вводится пользователем. Вывести в метки значения f1_double (x), a_double(x), b_double(x), f1_float(x), a_float(x), b_float(x), а также разности f1_float(x)-f2_float(x) и f1_double (x) - f2_double (x). Провести такое сравнение для аргументов x, меняющихся в пределах от 1E-8 до 0.1. Объяснить результаты.