CS61B Lecture 14

Monday, February 24, 2020

So far, we have dealt with several different types of numbers in Java, and today we will examine what's different between them.

Type	Bits	Signed?	Literals
`byte`	8	Yes	Cast from `int`: `(byte) 3`
`short`	16	Yes	None. Cast from `int`: `(short) 4096`
`char`	16	No	`a /// (char) 97` `\n // newline ((char) 10)`

`int`	32	Yes	`123`
			`0100 // Octal for 64`
			`0x3f, 0xffffffff // Hexadecimaal 63, -1`

`long`	64	Yes	`123L`, `01000L`

Negative numbers are negated positive literals. "`N` bits" means that there are

$$2^N$$

integers in the domain of the type:

- If signed, range of values is:

$$-2^{N-1}, \ldots, 2^{N-1} - 1$$

- If unsigned, only non-negative numbers, and range is

$$0, \ldots, 2^N - 1$$

Overflow

How do we handle overflow, such as `1000*1000*1000`? Some languages throw an exception (Ada), others give undefined results (C, C++). Java "wraps" the numbers within the number's bounds: adding one to the maximum turns it into the minimum in the range.

For example, `(byte) 128 == (byte) (127+1) == (byte) (-128)`. In general:

- If the result of some arithmetic subexpression is supposed to have type `T`, an `n`-bit integer type,
- then we compute the real (mathematical) value, `x`,
- and yield a number, `x'`, that is in the range of `T`, and that is equivalent to `x` modulo `2n`.
- (That means that `x – x'` is a multiple of `2n`.)

Modular Arithmetic

This system is called **modular arithmetic**: when a computation would “overflow” the range of values for a type, the value yielded is a remainder of division by some modulus.

We define

$$a \equiv b(mod n)$$

to mean that

$$a - b = kn$$

for some integer k.

Define the binary operation

$$amodn$$

as the value b such that

$$a \equiv b(mod n)$$

and

$$0 \leq b < n$$

for

$$n > 0$$

(Can be extended to 0 or less as well, but we won't bother with that here.) This is not the same as Java's % operation!

Various facts: Here, let `a'` denote

$$amodn$$

$$a'' = a'$$

$$a' + b'' = (a' + b)' = a + b'$$

$$(a' - b')' = (a' + (-b)')' = (a - b)'$$

$$(a' \bullet b')' = a' \bullet b' = a \bullet b'$$

$$(a^k)' = ((a')^k)' = (a \bullet (a^k - 1)')', \forall k > 0$$

One odd thing about this is that there is no built-in system to represent negative numbers. As such, negative numbers in Java are two tokens mushed into one: the negative token, and the number itself.

Since the legal integers range from 0 to 2147483647, the legal negative integers range from -1 and go on to 2147483648, this system would not let you write the minimum number so easily. Thus, Java has the rather strange behavior that:

```
x = -2147483648; // is a legal expression
x = 0-2147483648; // is not
```

Character values

`char` values are non-negative integers because integer values are used to represent characters according to the Unicode specification set. However, just because you can represent characters as numbers doesn't mean you should.

Conversion

In general Java will silently convert from one type to another if this makes sense and no information is lost from value. Otherwise, you must cast explicitly. Given:

```
byte = b; char c; short s; int i; long l;
```

	Expression
OK``	`s = b;` `i = b;` `i = s;` `i = c;` `l = i;`
Not OK	`i = l;` `b = i;` `c = i;` `c = i;` `s = c;` `c = s;` `c = b;`
OK by special dispensation	`b = 13; // 13 is compile-time constant` `b = 12 + 100; // 112 is a compile-time constant`

Promotion

Arithmetic operations promote operands as needed, where **promotion** is just implicit conversion.

For integer operations:

- if any operand is a `long`, then promote both operands to `long`
- otherwise, promote both to `int`

Thus, given:

```
byte = b; char c; short s; int i; long l;
b + 3 = (int) b + 3; // Type int
l + 3 = l + (long) 3; // Type long
'A' + 2 == (int) 'A' + 2; // Type int
b = b + 1; // ILLEGAL
b += 1; // Equivalent to b = (byte) (b+1)
```

Bitwise Operations

Numbers in Java are represented by a series of bits, booleans that are represented by ones and zeros. You can actually directly fiddle around with those numbers if you'd like.

Much like how "and" works in the Java conditional spec, Java will return 1 for a bitwise operation only if the two numbers on either side of & are both 1, and 0 for every other value.

Similarly, for "or", Java will return 0 if the two numbers on either side of | are 0, and 1 for every other value.

"XOR" (^) stands for exclusive-or, and it means that the operator will return a value of 1 only if one of the two numbers is 1, and the other is zero. If both numbers are 1 or 0, it returns 0.

And "NOT" (¬) is simply the negation operation, reversing whatever value it operates on. Observe the below truth table:

x	y	x & y	x | y	x ^ y	¬ x
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

There is a slightly modified truth table that may be more useful:

	y	x & y	x | y	x ^ y
x	0	0	x	x
x	1	x	1	x
x	x	x	x	0

If you have more than one digit, perform bitwise operation on each individual digit to get the final result.

Left shift and right shift

Numbers in Java aren't just stored as 0, 1 or 10. They are stored as a sequence of 8 bits (`short`), 16 bits (`int`), or even more (`long`). That means that a number like 1 is stored as 00000001 as a `short`.

The left shift (<<) describes an operation where the current value is shifted to the left by adding zeros to the "beginning" (beginning means the right, as we are talking about numbers) of the number. For example, `1 << 4` turns 1 into 10000, which is 16 in decimal form.

The arithmetic right shift (>>) describes an operation where the value is shifted to the right, by adding ones to the "end" of the number. `100 >> 2` turns the binary 100 into 111. Because of how the number of bits is a fixed size, we're technically not "adding" digits to the end, so much as shifting the number to the right, which is why it's called right shift.

The logical right (>>>) describes an operation where the value is shifted to the right by adding zeros to the end of the number, instead of ones.

> **Quick questions**
>
> Compute the following numbers:
>
> 1. `(-1) >>> 29`
> 2. `x << n`
> 3. `x >> n`
> 4. `x >>> 3 & ((1 << 5) - 1)`