## Integer Types and Literals

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Signed?</th>
<th>Literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8</td>
<td>Yes</td>
<td>Cast from <code>int</code>: (byte) 3</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>Yes</td>
<td>None. Cast from <code>int</code>: (short) 4096</td>
</tr>
<tr>
<td>char</td>
<td>16</td>
<td>No</td>
<td>'a' // (char) 97</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'\n' // newline ((char) 10)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'\t' // tab ((char) 8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'\' // backslash</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'A', '\101', '\u0041' // == (char) 65</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>Yes</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0100 // Octal for 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x3f, 0xffffffff // Hexadecimal 63, −1 (!)</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>Yes</td>
<td>123L, 01000L, 0x3fL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1234567891011L</td>
</tr>
</tbody>
</table>

- Negative numerals are just negated (positive) literals.

- "$N$ bits" means that there are $2^N$ integers in the domain of the type:
  - If signed, range of values is $-2^{N-1} .. 2^{N-1} - 1$.
  - If unsigned, only non-negative numbers, and range is $0 .. 2^N - 1$. 
Overflow

• **Problem:** How do we handle overflow, such as occurs in $10000 \times 10000 \times 10000$?

• Some languages throw an exception (Ada), some give undefined results (C, C++)

• Java *defines* the result of any arithmetic operation or conversion on integer types to “wrap around”—*modular arithmetic*.

• That is, the “next number” after the largest in an integer type is the smallest (like “clock arithmetic”).

• E.g., (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 $2^n$.
  - (That means that $x - x'$ is a multiple of $2^n$.)
Modular Arithmetic

- Define $a \equiv b \pmod{n}$ to mean that $a - b = kn$ for some integer $k$.

- Define the binary operation $a \mod n$ as the value $b$ such that $a \equiv b \pmod{n}$ and $0 \leq b < n$ for $n > 0$. (Can be extended to $n \leq 0$ 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 $a \mod n$).
  
  $$a'' = a'$$
  $$a' + b'' = (a' + b)' = a + b'$$
  $$(a' - b')' = (a' + (-b)')' = (a - b)'$$
  $$(a' \cdot b')' = a' \cdot b' = a \cdot b'$$
  $$(a^k)' = ((a')^k)' = (a \cdot (a^{k-1})')', \text{ for } k > 0.$$
Modular Arithmetic: Examples

- (byte) $(64\times8)$ yields 0, since $512 - 0 = 2 \times 2^8$.
- (byte) $(64\times2)$ and (byte) $(127+1)$ yield -128, since $128 - (-128) = 1 \times 2^8$.
- (byte) $(101\times99)$ yields 15, since $9999 - 15 = 39 \times 2^8$.
- (byte) $(-30\times13)$ yields 122, since $-390 - 122 = -2 \times 2^8$.
- (char) $(-1)$ yields $2^{16} - 1$, since $-1 - (2^{16} - 1) = -1 \times 2^{16}$.
Modular Arithmetic and Bits

• Why wrap around?

• Java’s definition is the natural one for a machine that uses binary arithmetic.

• For example, consider bytes (8 bits):

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1100101</td>
</tr>
<tr>
<td>× 99</td>
<td>1100011</td>
</tr>
<tr>
<td>9999</td>
<td>100111</td>
</tr>
<tr>
<td>− 9984</td>
<td>100111</td>
</tr>
<tr>
<td>15</td>
<td>00001111</td>
</tr>
</tbody>
</table>

• In general, bit $n$, counting from 0 at the right, corresponds to $2^n$.

• The bits to the left of the vertical bars therefore represent multiples of $2^8 = 256$.

• So throwing them away is the same as arithmetic modulo 256.
Negative numbers

• Why this representation for -1?

\[
\begin{array}{c|c}
1 & 00000001_2 \\
+ & 11111111_2 \\
= & 0100000000000000_2
\end{array}
\]

Only 8 bits in a byte, so bit 8 falls off, leaving 0.

• The truncated bit is in the \(2^8\) place, so throwing it away gives an equal number modulo \(2^8\). All bits to the left of it are also divisible by \(2^8\).

• On unsigned types (char), arithmetic is the same, but we choose to represent only non-negative numbers modulo \(2^{16}\):

\[
\begin{array}{c|c}
1 & 0000000000000001_2 \\
+ & 2^{16} - 1 \quad 1111111111111111_2 \\
= & 2^{16} + 0 \quad 10000000000000000_2
\end{array}
\]
Conversion

• In general Java will silently convert from one type to another if this makes sense and no information is lost from value.

• Otherwise, cast explicitly, as in (byte) x.

• Hence, given

```java
byte aByte; char aChar; short aShort; int anInt; long aLong;
```

// OK:
```java
aShort = aByte; anInt = aByte; anInt = aShort;
anInt = aChar; aLong = anInt;
```

// Not OK, might lose information:
```java
anInt = aLong; aByte = anInt; aChar = anInt; aShort = anInt;
aShort = aChar; aChar = aShort; aChar = aByte;
```

// OK by special dispensation:
```java
aByte = 13;  // 13 is compile-time constant
aByte = 12+100 // 112 is compile-time constant
```
Promotion

- Arithmetic operations (+, *, ...) promote operands as needed.
- Promotion is just implicit conversion.
- For integer operations,
  - if any operand is long, promote both to long.
  - otherwise promote both to int.
- So,

  aByte + 3 == (int) aByte + 3  // Type int
  aLong + 3 == aLong + (long) 3  // Type long
  'A' + 2 == (int) 'A' + 2  // Type int
  aByte = aByte + 1  // ILLEGAL (why?)

- But fortunately,

  aByte += 1;  // Defined as aByte = (byte) (aByte+1)

- Common example:

  // Assume aChar is an upper-case letter
  char lowerCaseChar = (char) ('a' + aChar - 'A');  // why cast?
Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No "conversion to bits" needed: they already are.

- Operations and their uses:

<table>
<thead>
<tr>
<th>Mask</th>
<th>Set</th>
<th>Flip</th>
<th>Flip all</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101100</td>
<td>00101100</td>
<td>00101100</td>
<td></td>
</tr>
<tr>
<td>&amp; 10100111</td>
<td>10100111</td>
<td>~ 10100111</td>
<td>~ 10100111</td>
</tr>
<tr>
<td>00100100</td>
<td>10101111</td>
<td>10001011</td>
<td>01011000</td>
</tr>
</tbody>
</table>

- Shifting:

<table>
<thead>
<tr>
<th>Left</th>
<th>Arithmetic Right</th>
<th>Logical Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101101 &lt;&lt; 3</td>
<td>10101101 &gt;&gt; 3</td>
<td>10101100 &gt;&gt;&gt; 3</td>
</tr>
<tr>
<td>01101000</td>
<td>11110101</td>
<td>00010101</td>
</tr>
</tbody>
</table>

  (-1) >>> 29?

- What is:

  - $x << n$?
  - $x >> n$?
  - ($x >>> 3$) & (($1<<5)-1$)?
Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No “conversion to bits” needed: they already are.

- Operations and their uses:

<table>
<thead>
<tr>
<th>Mask</th>
<th>Set</th>
<th>Flip</th>
<th>Flip all</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101100</td>
<td>00101100</td>
<td>00101100</td>
<td></td>
</tr>
<tr>
<td>&amp; 10100111</td>
<td>10100111</td>
<td>~ 10100111</td>
<td>~ 10100111</td>
</tr>
<tr>
<td>00100100</td>
<td>10101111</td>
<td>10001011</td>
<td>01011000</td>
</tr>
</tbody>
</table>

- Shifting:

<table>
<thead>
<tr>
<th>Left</th>
<th>Arithmetic Right</th>
<th>Logical Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101101 &lt;&lt; 3</td>
<td>10101101 &gt;&gt; 3</td>
<td>10101100 &gt;&gt;&gt; 3</td>
</tr>
<tr>
<td>01101000</td>
<td>11110101</td>
<td>00010101</td>
</tr>
</tbody>
</table>

  \((-1) >>> 29?) = 7.

- What is:

  \(x << n?)
  \(x >> n?)
  \((x >>> 3) & ((1<<5)-1)?)
Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No “conversion to bits” needed: they already are.

- Operations and their uses:

<table>
<thead>
<tr>
<th>Mask</th>
<th>Set</th>
<th>Flip</th>
<th>Flip all</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101100</td>
<td>00101100</td>
<td>00101100</td>
<td></td>
</tr>
<tr>
<td>&amp; 10100111</td>
<td>10100111</td>
<td>~ 10100111</td>
<td>~ 10100111</td>
</tr>
<tr>
<td>00100100</td>
<td>10101111</td>
<td>10001011</td>
<td>01011000</td>
</tr>
</tbody>
</table>

- Shifting:

  \[
  \begin{array}{c|c|c|c}
  \text{Left} & \text{Arithmetic Right} & \text{Logical Right} \\
  10101101 \ll 3 & 10101101 \gg 3 & 10101100 \ggg 3 \\
  01101000 & 11110101 & 00010101 \\
  \end{array}
  \]

  \((-1) \ggg 29? = 7.
  x \ll n?
  x \gg n?
  (x \ggg 3) \& ((1\ll5)-1)?
Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No “conversion to bits” needed: they already are.

- Operations and their uses:

<table>
<thead>
<tr>
<th>Mask</th>
<th>Set</th>
<th>Flip</th>
<th>Flip all</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101100</td>
<td>00101100</td>
<td>00101100</td>
<td>00101100</td>
</tr>
<tr>
<td>&amp; 10100111</td>
<td>10100111</td>
<td>~ 10100111</td>
<td>~ 10100111</td>
</tr>
<tr>
<td>00100100</td>
<td>10101111</td>
<td>10001011</td>
<td>01011000</td>
</tr>
</tbody>
</table>

- Shifting:

<table>
<thead>
<tr>
<th>Left</th>
<th>Arithmetic Right</th>
<th>Logical Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101101 &lt;&lt; 3</td>
<td>10101101 &gt;&gt; 3</td>
<td>10101100 &gt;&gt;&gt; 3</td>
</tr>
<tr>
<td>01101000</td>
<td>11110101</td>
<td>00010101</td>
</tr>
</tbody>
</table>

- What is:

  \[-1 \ggg 29?\]
  \[x \ggg n?\]
  \[x \ggg n?\]

  \[x \ggg 3) \& ((1<><5)-1)?\]

  = 7.

  = \(x \cdot 2^n\).

  = \lfloor x/2^n \rfloor \text{ (i.e., rounded down).}
Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No “conversion to bits” needed: they already are.

- Operations and their uses:

<table>
<thead>
<tr>
<th>Mask</th>
<th>Set</th>
<th>Flip</th>
<th>Flip all</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101100 &amp; 10100111</td>
<td>00101100</td>
<td>00101100</td>
<td>00101100</td>
</tr>
<tr>
<td>00100100</td>
<td>10100111</td>
<td>^ 10100111</td>
<td>~ 10100111</td>
</tr>
<tr>
<td>10101101 &lt;&lt; 3</td>
<td>10101111</td>
<td>10001011</td>
<td>01011000</td>
</tr>
</tbody>
</table>

- Shifting:

<table>
<thead>
<tr>
<th>Left</th>
<th>Arithmetic Right</th>
<th>Logical Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101101 &lt;&lt; 3</td>
<td>101011101 &gt;&gt; 3</td>
<td>10101100 &gt;&gt;&gt; 3</td>
</tr>
<tr>
<td>01101000</td>
<td>11110101</td>
<td>00010101</td>
</tr>
</tbody>
</table>

- What is:

  - \((-1) >>> 29\)?
    - \(x << n\)?
    - \(x >> n\)?
    - \((x >>> 3) & ((1<<5)-1)\)?

    \(-1) >>> 29 = 7.\n    \(x << n = x \cdot 2^n.\)
    \(x >> n = \lfloor x/2^n \rfloor \) (i.e., rounded down).
    \((x >>> 3) & ((1<<5)-1)\) \(5\)-bit integer, bits 3-7 of \(x.\)