Announcement:

- Programming contest SATURDAY! You can still sign up.
- Hackers@Berkeley "HackJam"—a 12 hour hackathon hosted by Hackers@Berkeley and sponsored by Box.
  - There will be food served throughout the event and prizes awarded at the end.
  - Who should come: Anyone interested in hacking, regardless of experience. There will be helpful students and engineers from Box there to help anyone who wants to learn.
  - Time: 11am-11pm Saturday, September 29th.
  - Place: Wozniak Lounge, Soda Hall.

Readings for Upcoming Topics:
- Data Structures (Into Java), Chapter 1.

Modular Arithmetic

- 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 II

  - (byte) (64+8) yields 0, since $512 - 0 = 2 \cdot 2^8$.
  - (byte) (64+2) and (byte) (127+1) yield -128, since $128 - (-128) = 1 \cdot 2^8$.
  - (byte) (345+6) yields 22, since $2070 - 22 = 8 \cdot 2^8$.
  - (byte) (-30+13) yields 122, since $-390 - 122 = -2 \cdot 2^8$.
  - (char) (-1) yields 216 - 1, since $-1 - (2^{16} - 1) = -1 \cdot 2^{16}$.
  - Natural definition for a machine that uses binary arithmetic:

```

```
  | Type | 0 = 0000000000000000 |
  | char | 1 = 000000001 |
  | byte | 2^{16} - 1 = 1111111111111111 |

  - Terminology: rightmost (units) bit is bit 0, 2s bit is bit 1.
  - Hence, changing bit $n$ modifies value by $2^n$; truncating on left to $n$ bits computes modulo $2^n$.

Last modified: Wed Sep 26 14:43:33 2012
### Negative numbers

- Why this representation for -1?

\[
\begin{array}{c|c}
1 & 00000001_2 \\
+ -1 & 11111111_2 \\
= 0 & 100000000_2 \\
\end{array}
\]

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 $2^{16}$:

\[
\begin{array}{c|c}
1 & 0000000000000001_2 \\
+ 2^{16} - 1 & 1111111111111111_2 \\
= 2^{16} + 0 & 1000000000000000_2 \\
\end{array}
\]

### 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

\[
\begin{align*}
\text{byte} & \text{ aByte; } \text{char} \text{ aChar; short aShort; int anInt; long aLong;} \\
// \text{ OK:} \\
& \text{aShort = aByte; anInt = aByte; anInt = aShort; anInt = aChar;}
\text{aLong = anInt;} \\
// \text{ Not OK, might lose information:} \\
& \text{anInt = aLong; aByte = anInt; aChar = anInt; aShort = anInt;}
\text{aShort = aChar; aChar = aShort; aChar = aByte;}
\end{align*}
\]

// OK by special dispensation:

\[
\begin{align*}
& \text{aByte = 13; } // \text{13 is compile-time constant} \\
& \text{aByte = 12+100 } // \text{112 is compile-time constant}
\end{align*}
\]

### 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,

\[
\begin{align*}
\text{aByte + 3} &= (\text{int}) \text{aByte + 3} // \text{Type int} \\
\text{aLong + 3} &= \text{aLong} + (\text{long}) 3 \quad // \text{Type long} \\
\text{A' + 2} &= (\text{int}) 'A' + 2 \quad // \text{Type int} \\
\text{aByte} &= \text{aByte} + 1 \quad // \text{ILLEGAL (why?)}
\end{align*}
\]

- But fortunately,

\[
\text{aByte += 1; } // \text{Defined as aByte} = (\text{byte}) \text{(aByte+1)}
\]

- Common example:

\[
\begin{align*}
// \text{Assume aChar is an upper-case letter} \\
\text{char lowerCaseChar} &= (\text{char}) ('A' + \text{aChar} - 'A'); // \text{why cast?}
\end{align*}
\]

### 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:

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Mask} & \text{Set} & \text{Flip} & \text{Flip all} \\
\hline
00101100 & 00101100 & 00101100 & 00101100 \\
\& 10100111 & 10100111 & \sim 10100111 & \sim 10100111 \\
00100100 & 10101111 & 10001011 & 01011000 \\
\hline
\end{array}
\]

- Shifting:

\[
\begin{align*}
\text{Left} & \quad \text{Arithmetic Right} & \quad \text{Logical Right} \\
10101101 & << 3 & 10101101 & >> 3 & 10101100 & >> 3 \\
01101000 & & 11110101 & & 00010101 \\
\end{align*}
\]

- What is:

\[
\begin{align*}
\text{x} & << n? \\
\text{x} & >> n? \\
\text{(x >> 3) & ((1<<5)-1)}?
\end{align*}
\]