Which number is larger?

25

8

We need to distinguish between numbers and the symbols that represent them, called numerals.

The number 25 is larger than 8, but the numeral 8 above is larger than the numeral 25.

The number twenty-five can be represented in many ways:

- Decimal system (base 10): 25
- Roman numerals: XXV
- Binary system (base 2): 11001
- Hexadecimal (base 16): 19

On a computer, the binary switch is easy to implement, so numbers are stored in a computer as binary (on and off).

In fact, everything stored in a computer is encoded using the binary number system.

- numbers, characters, strings, booleans, objects, instructions

We need to understand the binary system to understand computers.
Positional number representation

Choose a base (or radix) \( b \), and a set of \( b \) distinct symbols.

The radix \( b \) representation of a nonegative integer \( m \) is a string of digits chosen from the set \{ 0, 1, 2, 3, \ldots, b-1 \} so that if

\[ m = (d_kd_{k-1} \ldots d_3d_2d_1d_0)_b, \]

then \( m = d_kb^k + d_{k-1}b^{k-1} + \ldots + d_3b^3 + d_2b^2 + d_1b^1 + d_0b^0 \)

We can represent any nonnegative number in base \( b \) by forming such a polynomial where for all \( d_k, 0 \leq d_k < b \).

If \( b>10 \), new symbols must be used for digits above 9.

Examples

Decimal (\( b=10 \)): Set of digits = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}

\[ 493 = 4\cdot10^2 + 9\cdot10^1 + 3\cdot10^0 = 400 + 90 + 3 \]

Binary (\( b=2 \)): Set of digits = \{ 0, 1 \}

\[ 11010_2 = 1\cdot2^4 + 1\cdot2^3 + 0\cdot2^2 + 1\cdot2^1 + 0\cdot2^0 \]
\[ = 16 + 8 + 2 = 26_{10} \]

Octal (\( b=8 \)): Set of digits = \{ 0, 1, 2, 3, 4, 5, 6, 7 \}

\[ 2076_8 = 2\cdot8^3 + 0\cdot8^2 + 7\cdot8^1 + 6\cdot8^0 \]
\[ = 2\cdot512 + 56 + 6 \]
\[ = 1086_{10} \]
Hexadecimal \((b=16)\):

Set of digits = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F \}

\[ B3F_{16} = 11\cdot16^2 + 3\cdot16^1 + 15\cdot16^0 \]
\[ = 11\cdot256 + 48 + 15 \]
\[ = 2879_{10} \]

Note that the arithmetic is done in the decimal number system.

Number Conversions

1. **Converting a radix \(b\) numeral to decimal**
   Evaluate the corresponding polynomial.
   See examples above or use Horner’s Method.

   **Horner’s Method** for Evaluating Polynomials:
   Start with zero and then alternate adding a digit
   with multiplying by the base.

   **Binary Example**
   \[ 1011011_2 = 1\cdot2^6 + 0\cdot2^5 + 1\cdot2^4 + 1\cdot2^3 + 0\cdot2^2 + 1\cdot2^1 + 1\cdot2^0 \]
   \[ = (((((0+1)\cdot2+0)\cdot2+1)\cdot2+0)\cdot2+1)\cdot2+1 \]
   Multiply by 2 moving down; add a bit moving right.
   \[ 0 + 1 = 1 \]
   \[ 2 + 0 = 2 \]
   \[ 4 + 1 = 5 \]
   \[ 10 + 1 = 11 \]
   \[ 22 + 0 = 22 \]
   \[ 44 + 1 = 45 \]
   \[ 90 + 1 = 91_{10} \]
Octal Example

\[ 456_8 = 4 \cdot 64 + 5 \cdot 8 + 6 = 302_{10} \]
\[ 0 + 4 = 4 \]
\[ 32 + 5 = 37 \]
\[ 296 + 6 = 302_{10} \]

Hexadecimal Example \[ 37D2_{16} \]

\[ 0 + 3 = 3 \]
\[ \times 16 \]
\[ 48 + 7 = 55 \]
\[ \times 16 \]
\[ 880 + 13 = 893 \]
\[ \times 16 \]
\[ 14288 + 2 = 14290_{10} \]

2. Converting a decimal numeral to radix \( b \)

Suppose a decimal numeral \( m \) has the following representation in radix \( b \):

\[ m = d_k b^k + d_{k-1} b^{k-1} + \ldots + d_3 b^3 + d_2 b^2 + d_1 b + d_0 \]

Divide \( m \) by \( b \) using decimal arithmetic.

quotient = \( d_k b^{k-1} + d_{k-1} b^{k-2} + \ldots + d_3 b^2 + d_2 b + d_1 \)

remainder = \( d_0 \)

Continue dividing quotients by \( b \), saving the remainders, until the quotient is zero.
Pseudo-code Version of Algorithm

num = decimal numeral to be converted;
k = 0;

\textbf{do}
\begin{verbatim}
{ 
d_k = num \% b;
    write symbol for d_k; // k denotes positions of
    num = num / b; // symbols from right to left.
k++;
}
\textbf{while} (num > 0);
\end{verbatim}

Resulting numeral, base $b$

$d_9 \, d_8 \, d_7 \, d_6 \, d_5 \, d_4 \, d_3 \, d_2 \, d_1 \, d_0$

for as many digits as are produced until $num$ becomes zero.

Examples

• Decimal to Octal: $1492_{10}$

\begin{align*}
\begin{array}{r|c|c|c|c}
  & 186 & 23 & 2 & 0 \\
\hline
8 & 1492 & 186 & 23 & 2 \\
\hline
   & 8 & 16 & 16 & 0 \\
  & 69 & 26 & 7 & 2 \\
64 & 64 & 24 & & \\
52 & 52 & 2 & & \\
48 & 48 & & & \\
4 & 4 & & & \\
\end{array}
\end{align*}

Result = $2724_8$
• Decimal to Hexadecimal: $438_{10}$

\[
\begin{array}{c|c|c}
\text{16)438} & \text{16)27} & \text{16)1} \\
\hline
32 & 16 & 0 \\
118 & 11 & 0 \\
112 & 6 & 1 \\
\end{array}
\]

Result = $1B6_{16}$

• Decimal to Binary: $777_{10}$

<table>
<thead>
<tr>
<th>Quotients</th>
<th>Remainders</th>
</tr>
</thead>
<tbody>
<tr>
<td>777</td>
<td></td>
</tr>
<tr>
<td>388</td>
<td>1</td>
</tr>
<tr>
<td>194</td>
<td>0</td>
</tr>
<tr>
<td>97</td>
<td>0</td>
</tr>
<tr>
<td>48</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Result = $1100001001_{2}$

Some Useful Values

<table>
<thead>
<tr>
<th>$2 \cdot 16$ = 32</th>
<th>$6 \cdot 16$ = 96</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 \cdot 16$ = 48</td>
<td>$7 \cdot 16$ = 112</td>
</tr>
<tr>
<td>$4 \cdot 16$ = 64</td>
<td>$8 \cdot 16$ = 128</td>
</tr>
<tr>
<td>$5 \cdot 16$ = 80</td>
<td>$9 \cdot 16$ = 144</td>
</tr>
<tr>
<td>$16^2$ = 256 = $2^8$</td>
<td>$16^4$ = 65,536 = $2^{16}$</td>
</tr>
<tr>
<td>$16^3$ = 4,096 = $2^{12}$</td>
<td>$16^5$ = 1,048,576 = $2^{20}$</td>
</tr>
</tbody>
</table>
3. Converting Binary to Hexadecimal:
   Group binary digits, 4 bits at a time, proceeding from right to left, and transform the groups into hexadecimal digits.

   Binary to Hexadecimal
   \[11010011011010_2 = 11010011101101 = 34ED_{16} = 34ED_H\]

4. Converting Hexadecimal to Binary:
   Substitute correct 4 bit patterns for hexadecimal digits.

   Hexadecimal to Binary
   \[7A5F_{16} = 0111101001011111_2\]

**Octal and Hexadecimal**

These number systems provide us with a convenient form of shorthand for working with binary quantities.

\[
\text{int } m = 25979882;
\]

\(m\) is stored as 32 bits: 00000001100011000101111101010
Easier to represent as 8 hex digits: 018c6bea
Learn the four-bit binary patterns for the 16 hex digits.

**Note:** We are only dealing with nonnegative integers at this time.
We call them *unsigned* integers.
**Arithmetic**

Algorithms for addition and multiplication are essentially the same for any radix system.

The difference lies in the tables that define the sum and product of two digits.

### Binary Addition:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

### Binary Multiplication:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Example**

\[
\begin{array}{c}
1101101 \\
\times 11011 \\
\hline
1101101 \\
1101101 \\
0000000 \\
1101101 \\
+1101101 \\
\hline
101101111111
\end{array}
\]

Note: \(1+1+1+1+1 = 101\)

### Practice

Construct an addition table and a multiplication table in hexadecimal.
Finite Representation

Numbers inside a computer are constrained to a fixed finite number of bits, which affects range of numbers we can represent.

With \( n \) bits:

Number of possibilities = \( 2 \cdot 2 \cdot 2 \cdots 2 = 2^n \)

Unsigned numbers may range from 0 to \( 2^n - 1 \).

Example

Consider \( n = 8 \)

![Diagram](image)

Range: 0 to \( 2^8 - 1 \) (0 to 255 decimal)

Hexadecimal representations: 00 to FF

Representing Negative Integers

1. Sign / magnitude

Leftmost bit represents a sign, 0 for + and 1 for −.
The rest of the bits represent the absolute value of number.

With \( k=6 \) bits,

\[
\begin{align*}
9 &= 0 01001 \\
-9 &= 1 01001
\end{align*}
\]

Range of values: \(-(2^{k-1} - 1)\) to \( (2^{k-1} - 1)\).

For \( k=6 \), the values range from -31 to 31.

Only 5 bits available to represent the magnitude.
The maximum and minimum numbers, 011111 and 111111, have equal magnitude
Problems

- We have two forms of zero, 000000 and 100000.
- Arithmetic is expensive because we need an adder and a subtractor to do addition.
  
  If the signs are the same, add the magnitudes.
  
  If the signs are different, subtract the smaller magnitude from the larger and keep the sign from the larger magnitude.

2. Two’s complement

Nonnegative integers 0 to \((2^{k-1} - 1)\) are represented in binary the same way as with sign-magnitude, 0 followed by a \((k-1)\)-bit absolute value.

Negative integers \(-(2^{k-1})\) to -1 are represented by adding \(2^k\) and expressing the result in binary.

Example  \(k=6\)

Note that \(2^k = 64\).

\[
\begin{align*}
2 & = 000010_2 \\
13 & = 001101_2 \\
30 & = 011110_2 \\
31 & = 011111_2 \\
-2 & \Rightarrow 64 - 2 = 62 = 111110_2 \\
-13 & \Rightarrow 64 - 13 = 51 = 110011_2 \\
-30 & \Rightarrow 64 - 30 = 34 = 100010_2 \\
-32 & \Rightarrow 64 - 32 = 32 = 100000_2
\end{align*}
\]

Arithmetic: \(2 + (-2) = 000010 + 111110 = 000000\)
Ways of Interpreting Bit Patterns (k = 4)

<table>
<thead>
<tr>
<th>BIT PATTERN</th>
<th>UNSIGNED</th>
<th>SIGN &amp; MAGNITUDE</th>
<th>TWO'S COMPLEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-0</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-1</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-2</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-3</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-5</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-6</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-7</td>
<td>-1</td>
</tr>
</tbody>
</table>

Algorithm for Negating a Number in two's complement
   a) Complement (flip) each bit.
   b) Add 1

Complementing a k-bit number is same as subtracting it from $2^k - 1$. 
Examples

\[\text{k=6} \hspace{1cm} \text{Note } 2^k - 1 = 111111_2\]

13 = 001101_2
a) Complement bits: 110010
b) Add one: 110011

-13 = 110011_2
a) Complement bits: 001100
b) Add one: 001101

0 = 000000_2
a) Complement bits: 111111
b) Add one: 000000 (discard left carry)

The high order bit still represents sign of the number (0 for nonnegative and 1 for negative).

Now we have only one form of zero.

The magnitudes of the maximum and minimum values differ:
\[|\text{minimum}| = \text{maximum} + 1\]

Hence, one number cannot have the negation operation performed on it. Which one?

\textbf{int} num = -2147483648;  
-\text{num} is an erroneous operation now.

\textbf{long} big = -9223372036854775808;  
-\text{big} is an erroneous operation now.
Odometer principle for twos-complement.

Evaluating Twos Complement

The problem is converting negative twos complement to decimal.

Procedure
1. Find twos complement negative of number (a positive value).
2. Convert that number to decimal
3. Append a minus sign.
Examples

\( k = 8 \)

10010111

Negate to get 01101001

Convert to decimal: \( 64 + 32 + 8 + 1 = 105 \)

Append minus: -105

11111111

Negate to get 00000001

Convert to decimal: 1

Append minus: -1

11000000

Negate to get 01000000

Convert to decimal: 64

Append minus: -64

Exercise: Assuming 6 bit numbers, what values are represented by a) 011101 and b) 111010?

Addition of Signed Numbers

Sign / magnitude: Follow the same rules you use when working with the decimal arithmetic learned in grade school.

Add magnitudes when signs are the same.

Subtract magnitudes when signs are different.

Example 1

101101

+100111

---------

01101

+00111

---------

10100 Since the signs are the same, we add the magnitudes of the two numbers. Note that we do not add the signs.

10100 Now append the sign to get 110100
Example 2

Since the signs are different, we subtract the magnitudes of the two numbers.

Note that we do not subtract the signs.

Larger magnitude
Smaller magnitude

Now append sign of the larger to get 101000

Two’s complement Arithmetic

Add right to left, bit by bit, including the sign bits. Ignore any carry out of the sign position.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Explanation (optional)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01010</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>+00011</td>
<td>+ 3</td>
<td>---</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>------------------------</td>
</tr>
<tr>
<td>01101</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td>+ 7</td>
<td>25–32</td>
</tr>
<tr>
<td>+00101</td>
<td>+ 5</td>
<td>+ 5</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>------------------------</td>
</tr>
<tr>
<td>11110</td>
<td>−2</td>
<td>30–32</td>
</tr>
<tr>
<td>11001</td>
<td>−7</td>
<td>25–32</td>
</tr>
<tr>
<td>+11001</td>
<td>−7</td>
<td>+25–32</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>------------------------</td>
</tr>
<tr>
<td>110010</td>
<td>−14</td>
<td>50–32–32 = 32+18–32–32</td>
</tr>
</tbody>
</table>

* Carries are discarded
Overflow

- Add two nonnegative numbers and get a negative answer.
- Add two negative numbers and get a nonnegative answer.

Radix Conversions in Java

Converting decimal (int) to other bases

```java
int m = 123456;
Integer.toBinaryString(m) returns "1110001001000000"
Integer.toOctalString(m) returns "361100"
Integer.toHexString(m) returns "1e240"
Integer.toString(m) returns "123456"
```

Converting other bases to decimal

```java
Integer.parseInt("1110001001000000", 2)
Integer.parseInt("361100", 8)
Integer.parseInt("1e240", 16)
Integer.parseInt("123456")
```

each returns the int 123456

Octal and Hex Literals and Unicode

```java
int k = 0123; // leading 0 means octal digits
int m = 0xabc; // leading 0x means hex digits
char c = \u0041; // char 'A' = 41 hex
```
Logical Instructions

Java performs logical operations on integers of type `int` and `long`.

```java
int m1, m2, n;
m1 = some value;
m2 = some other value;
```

**OR**

```java
n = m1 | m2;
```

m1 and m2 are bit-wise OR-ed to produce the result.

- $1 | 1 = 1$
- $1 | 0 = 1$  \(1\) means true
- $0 | 1 = 1$  \(0\) means false
- $0 | 0 = 0$

**AND**

```java
n = m1 & m2;
```

m1 and m2 are bit-wise AND-ed to produce the result.

- $1 & 1 = 1$
- $1 & 0 = 0$
- $0 & 1 = 0$
- $0 & 0 = 0$

**Exclusive OR**

```java
n = m1 ^ m2;
```

m1 and m2 are bit-wise Exclusive OR-ed to produce the result.

- $1 ^ 1 = 0$  Equivalent to $\neq$
- $1 ^ 0 = 1$
- $0 ^ 1 = 1$
- $0 ^ 0 = 0$
NOT

\[ n = \overline{m1}; \]
All the bits in \( m1 \) are complemented to produce the result.

\[ \overline{1} = 0 \]
\[ \overline{0} = 1 \]

Examples
Suppose we have declarations

\[
\begin{align*}
\text{int } m &= 0x6ca64; \\
\text{int } n &= 0xB93DE; \\
\text{int } ans;
\end{align*}
\]

OR

\[
\begin{align*}
\text{ans } &= \text{m } \text{l } \text{n;} \\
\text{m } &= 0000 \ 0000 \ 0000 \ 0110 \ 1100 \ 1010 \ 0110 \ 0100 \\
\text{n } &= 0000 \ 0000 \ 0000 \ 1011 \ 1001 \ 0011 \ 1101 \ 1110 \\
\text{m } \text{l } \text{n } &= 0000 \ 0000 \ 0000 \ 1111 \ 1101 \ 1011 \ 1111 \ 1110
\end{align*}
\]

AND

\[
\begin{align*}
\text{ans } &= \text{m } \text{& } \text{n;} \\
\text{m } &= 0000 \ 0000 \ 0000 \ 0110 \ 1100 \ 1010 \ 0110 \ 0100 \\
\text{n } &= 0000 \ 0000 \ 0000 \ 1011 \ 1001 \ 0011 \ 1101 \ 1110 \\
\text{m } \text{& } \text{n } &= 0000 \ 0000 \ 0000 \ 0010 \ 1000 \ 0010 \ 0100 \ 0100
\end{align*}
\]
Exclusive OR (acts like not-equal)

\[ \text{ans} = m \oplus n; \]

\begin{align*}
m & = 0000 \ 0000 \ 0000 \ 0110 \ 1100 \ 1010 \ 0110 \ 0100 \\
n & = 0000 \ 0000 \ 0000 \ 1011 \ 1001 \ 0011 \ 1101 \ 1110 \\
m \oplus n & = 0000 \ 0000 \ 0000 \ 1101 \ 0101 \ 1001 \ 1011 \ 1010
\end{align*}

NOT

\[ \text{ans} = \neg m; \]

\begin{align*}
m & = 0000 \ 0000 \ 0000 \ 0110 \ 1100 \ 1010 \ 0110 \ 0100 \\
\neg m & = 1111 \ 1111 \ 1111 \ 1001 \ 0011 \ 0101 \ 1001 \ 1011
\end{align*}

**Bit Masks**

**Terminology**

- Clear a bit: Make it 0
- Set a bit: Make it 1

**Problem:** Clear all the bits in n.

\[ n = 0; \]

**Problem:** Clear bits in the first and third bytes in n.

\[ n = n \& 0x00FF00FF \quad \text{// lead zeros redundant} \]

**Problem:** Set all the bits in n.

\[ n = 0xFFFFFFFF; \quad \text{or} \quad n = -1; \]

**Problem:** Set the bits in the first and fourth bytes in n.

\[ n = n \mid 0xFF0000FF; \]
Problem: Flip all the bits in n.
    \[ n = \sim n; \]

Problem: Flip the bits in the fourth byte in n.
    \[ n = n \wedge 0x000000FF; \quad \text{// lead zeros redundant} \]

**Shift Operations**

Move the bits in a word to the left or right.

**Two Kinds**

Logical Shifts: Word filled with zeros as bits are moved.

Arithmetic Shifts: Left shift is same as logical shift.
                    Right shift replicates the sign bit

Shift operations take two operands.

```
int word; \quad \text{// Word to be shifted}
int numb; \quad \text{// Number of bits to shift}
```

Only the last six bits of the shift number are considered, giving shift distance \(0 \leq \text{numb} \leq 63\).

Logical shift left
    \[ \text{word} = \text{word} << \text{numb}; \]

Logical shift right
    \[ \text{word} = \text{word} >>> \text{numb}; \]

Arithmetic shift left
    \[ \text{word} = \text{word} <<< \text{numb}; \]
Arithmetic shift right
    word = word >> numb;

Examples

Suppose we have declarations
    int p = 500;
    int q = -12;
    int ans;

    ans = p << 2;
    p       = 0000 0000 0000 0000 0000 0000 0001 1111 0100
    p<<2    = 0000 0000 0000 0000 0000 0000 0111 1101 0000
    ans has the value 2000.

    ans = p >>> 2;    // same as p >> 2 in this case (p ≥ 0)
    p       = 0000 0000 0000 0000 0000 0000 0001 1111 0100
    p>>>2   = 0000 0000 0000 0000 0000 0000 0111 1101
    ans has the value 125.

    ans = q << 3;
    q       = 1111 1111 1111 1111 1111 1111 1111 0100
    q<<3    = 1111 1111 1111 1111 1111 1111 1111 1010 0000
    ans has the value -96.

    ans = q >> 3;
    q       = 1111 1111 1111 1111 1111 1111 1111 0100
    q>>3    = 1111 1111 1111 1111 1111 1111 1111 1110
    ans has the value -2.
ans = q >>> 3;

q = 1111 1111 1111 1111 1111 1111 1111 0100
q >>> 3 = 0001 1111 1111 1111 1111 1111 1111 1110
ans has the value 536870910.

**Sign Extension**

Observe what happens when integers are enlarged or truncated.

<table>
<thead>
<tr>
<th></th>
<th>Binary Storage</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>byte</strong> b1 = 100;</td>
<td>0110 0100</td>
<td>64</td>
</tr>
<tr>
<td><strong>byte</strong> b2 = -10;</td>
<td>1111 0110</td>
<td>F6</td>
</tr>
<tr>
<td><strong>int</strong> n1 = b1;</td>
<td>0000 0000 0000 0000 0000 0000 0110 0100 00000064</td>
<td></td>
</tr>
<tr>
<td><strong>int</strong> n2 = b2;</td>
<td>1111 1111 1111 1111 1111 1111 1111 0110 FFFFFFF6</td>
<td></td>
</tr>
<tr>
<td><strong>int</strong> n3 = -200;</td>
<td></td>
<td>FFFFFFF38</td>
</tr>
<tr>
<td><strong>byte</strong> b3 = (byte)n3;</td>
<td></td>
<td>38 (= 56_{10})</td>
</tr>
<tr>
<td><strong>int</strong> n4 = 150;</td>
<td></td>
<td>00000096</td>
</tr>
<tr>
<td><strong>byte</strong> b4 = (byte)n3;</td>
<td></td>
<td>96 (= -106_{10})</td>
</tr>
</tbody>
</table>
Character Data

Computers do not store character data directly.

IO devices provide for a mapping between the character symbols and internal form of the characters.

For portability between machines, standard encodings of characters are used.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Bits</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ascii</td>
<td>8</td>
<td>Up to 256</td>
</tr>
<tr>
<td>Unicode</td>
<td>16</td>
<td>Up to 65,536</td>
</tr>
</tbody>
</table>

Unicode is an extension of ascii:

Ascii 'A' is 01000001
Unicode 'A' is 00000000 01000001

Typing the letter A on the keyboard

sends the binary string 01000001 to the computer,
which stores it as 00000000 01000001 in Java.
### ASCII Character Set

#### In Hexadecimal

<table>
<thead>
<tr>
<th>0</th>
<th>00 nul</th>
<th>01 soh</th>
<th>02 stx</th>
<th>03 etx</th>
<th>04 eot</th>
<th>05 enq</th>
<th>06 ack</th>
<th>07 bel</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>08 bs</td>
<td>9</td>
<td>09 ht</td>
<td>a</td>
<td>0a nl</td>
<td>b</td>
<td>0b vt</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>0d cr</td>
<td>e</td>
<td>0e so</td>
<td>f</td>
<td>0f si</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>dle</td>
<td>11</td>
<td>dcl</td>
<td>12</td>
<td>dc2</td>
<td>13</td>
<td>dc3</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>nak</td>
<td>16</td>
<td>syn</td>
<td>17</td>
<td>etb</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>can</td>
<td>19</td>
<td>em</td>
<td>1a</td>
<td>sub</td>
<td>1b</td>
<td>esc</td>
<td>1c</td>
</tr>
<tr>
<td>1d</td>
<td>gs</td>
<td>1e</td>
<td>rs</td>
<td>1f</td>
<td>us</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>sp</td>
<td>21</td>
<td>!</td>
<td>22</td>
<td>&quot;</td>
<td>23</td>
<td>#</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>%</td>
<td>26</td>
<td>&amp;</td>
<td>27</td>
<td>'</td>
<td>28</td>
<td>(</td>
<td>29</td>
</tr>
<tr>
<td>2a</td>
<td>*</td>
<td>2b</td>
<td>+</td>
<td>2c</td>
<td>,</td>
<td>2d</td>
<td>-</td>
<td>2e</td>
</tr>
<tr>
<td>2f</td>
<td>/</td>
<td>30</td>
<td>0</td>
<td>31</td>
<td>1</td>
<td>32</td>
<td>2</td>
<td>33</td>
</tr>
<tr>
<td>34</td>
<td>4</td>
<td>35</td>
<td>5</td>
<td>36</td>
<td>6</td>
<td>37</td>
<td>7</td>
<td>38</td>
</tr>
<tr>
<td>39</td>
<td>9</td>
<td>3a</td>
<td>:</td>
<td>3b</td>
<td>;</td>
<td>3c</td>
<td>&lt;</td>
<td>3d</td>
</tr>
<tr>
<td>3e</td>
<td>&gt;</td>
<td>3f</td>
<td>?</td>
<td>40</td>
<td>@</td>
<td>41</td>
<td>A</td>
<td>42</td>
</tr>
<tr>
<td>43</td>
<td>C</td>
<td>44</td>
<td>D</td>
<td>45</td>
<td>E</td>
<td>46</td>
<td>F</td>
<td>47</td>
</tr>
<tr>
<td>48</td>
<td>H</td>
<td>49</td>
<td>I</td>
<td>4a</td>
<td>J</td>
<td>4b</td>
<td>K</td>
<td>4c</td>
</tr>
<tr>
<td>4d</td>
<td>M</td>
<td>4e</td>
<td>N</td>
<td>4f</td>
<td>O</td>
<td>50</td>
<td>P</td>
<td>51</td>
</tr>
<tr>
<td>52</td>
<td>R</td>
<td>53</td>
<td>S</td>
<td>54</td>
<td>T</td>
<td>55</td>
<td>U</td>
<td>56</td>
</tr>
<tr>
<td>57</td>
<td>W</td>
<td>58</td>
<td>X</td>
<td>59</td>
<td>Y</td>
<td>5a</td>
<td>Z</td>
<td>5b</td>
</tr>
<tr>
<td>5c</td>
<td>\</td>
<td>5d</td>
<td>]</td>
<td>5e</td>
<td>^</td>
<td>5f</td>
<td>_</td>
<td>60</td>
</tr>
<tr>
<td>61</td>
<td>a</td>
<td>62</td>
<td>b</td>
<td>63</td>
<td>c</td>
<td>64</td>
<td>d</td>
<td>65</td>
</tr>
<tr>
<td>66</td>
<td>f</td>
<td>67</td>
<td>g</td>
<td>68</td>
<td>h</td>
<td>69</td>
<td>i</td>
<td>6a</td>
</tr>
<tr>
<td>6b</td>
<td>k</td>
<td>6c</td>
<td>l</td>
<td>6d</td>
<td>m</td>
<td>6e</td>
<td>n</td>
<td>6f</td>
</tr>
<tr>
<td>70</td>
<td>p</td>
<td>71</td>
<td>q</td>
<td>72</td>
<td>r</td>
<td>73</td>
<td>s</td>
<td>74</td>
</tr>
<tr>
<td>75</td>
<td>u</td>
<td>76</td>
<td>v</td>
<td>77</td>
<td>w</td>
<td>78</td>
<td>x</td>
<td>79</td>
</tr>
<tr>
<td>7a</td>
<td>z</td>
<td>7b</td>
<td>{</td>
<td>7c</td>
<td></td>
<td>7d</td>
<td>}</td>
<td>7e</td>
</tr>
<tr>
<td>7f</td>
<td>del</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Converting a Digit Character to Numeric

Suppose `ch` contains an ascii digit.

```c
char ch = an ascii digit;
```

Conversion:

```c
int m = ch - '0';   // int arithmetic
```

or

```c
int n = ch & 0xF;
```

Examples

```
'6' - '0' = 54 - 48 = 6
00000036 & 0000000F = 0000006  // hexadecimal
```