From bits to bytes to ints

- At some level everything is stored as either a zero or a one
  - A bit is a binary digit a byte is a binary term (8 bits)
  - We should be grateful we can deal with Strings rather than sequences of 0's and 1's.
  - We should be grateful we can deal with an int rather than the 32 bits that make an int

- int values are stored as two's complement numbers with 32 bits, for 64 bits use the type long, a char is 16 bits
  - Standard in Java, different in C/C++
  - Facilitates addition/subtraction for int values
  - We don't need to worry about this, except to note:
    - Infinity + 1 = - Infinity
    - Math.abs(-Infinity) > Infinity
How are data stored?

- To facilitate compression coding we need to manipulate individual bits
  - Why do we need to read one bit?
  - Why do we need to write one bit?
  - When do we read 8 bits at a time? Read 32 bits at a time?

- We can't actually write one bit-at-a-time. We can't really write one char at a time either.
  - Output and input are buffered, minimize memory accesses and disk accesses
  - Why do we care about this when we talk about data structures and algorithms?
    - Where does data come from?
How do we buffer char output?

- **Done for us as part of InputStream and Reader classes**
  - InputStreams are for reading bytes
  - Readers are for reading char values
  - Why do we have both and how do they interact?
  
```java
Reader r = new InputStreamReader(System.in);
```

- Do we need to flush our buffers?

- **In the past Java IO has been notoriously slow**
  - Do we care about I? About O?
  - This is changing, and the java.nio classes help
    - Map a file to a region in memory in one operation
Buffer bit output

- To buffer bit output we need to store bits in a buffer
  - When the buffer is full, we write it.
  - The buffer might overflow, e.g., in process of writing 10 bits to 32-bit capacity buffer that has 29 bits in it
  - How do we access bits, add to buffer, etc.?

- We need to use bit operations
  - Mask bits -- access individual bits
  - Shift bits – to the left or to the right
  - Bitwise AND / OR / NEGATE bits
Bit Logical Operations

- Work on integers types in binary (by bit)
  - longs, ints, chars, and bytes
- Three binary operators
  - And: &
  - Or: |
  - Exclusive Or (xor): ^

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>a&amp;b</th>
<th>a</th>
<th>a^b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- What is result of
  - 27 & 14?
  - 27 | 14?
  - 27 ^ 14?
Bit Logical Operations

- **Need to work bit position by bit position**
  
  \[
  \begin{align*}
  11011 &= 27 & \text{(many leading zeros not shown)} \\
  01110 &= 14 \\
  \end{align*}
  \]

  \[
  \begin{align*}
  \& 01010 &= \\
  | 11111 &= \\
  ^ 10101 &= \\
  \end{align*}
  \]

- **Also have unary negation (not):** \(~\)
  
  \[
  \begin{align*}
  00000000000000000000000000011011 &= 27 \\
  ~11111111111111111111111111100100 &= -28 \\
  \end{align*}
  \]

- **Use “masks” with the various operators to**
  - Set or clear bits
  - Test bits
  - Toggle bits

  (Example later)
Bit Shift Operations

- Work on same types as logical ops
- One left shift and two right shifts
  - Left shift: <<
    \[
    \begin{align*}
    11011 & = 27 \\
    27 & << 2 \\
    1101100 & = 108
    \end{align*}
    \]
    (shifting left is like? )
  - Logical right shift: >>>
    \[
    \begin{align*}
    11011 & = 27 \\
    27 & >>> 2 \\
    110 & = 6
    \end{align*}
    \]
    (shifting right is like? )
  - Arithmetic right shift: >>
    \[
    \begin{align*}
    111111111111111111111111100100 & = -28 \\
    -26 & >> 2 \\
    11111111111111111111111111001 & = -7
    \end{align*}
    \]
    \[
    \begin{align*}
    11111111111111111111111111111111 & = -1 \\
    -1 & >>> 16 \text{ (for contrast)} \\
    0000000000000001111111111111111 & = 65535
    \end{align*}
    \]
Representing pixels

- A pixel typically stores RGB and alpha/transparency values
  - Each RGB is a value in the range 0 to 255
  - The alpha value is also in range 0 to 255
  
  ```java
  Pixel red = new Pixel(255, 0, 0, 0);
  Pixel white = new Pixel(255, 255, 255, 0);
  ```

- Typically store these values as int values, a picture is simply an array of int values

  ```java
  void process(int pixel){
      int blue = pixel & 0xff;
      int green = (pixel >> 8) & 0xff;
      int red = (pixel >> 16) & 0xff;
  }
  ```
Bit masks and shifts

```c
void process(int pixel) {
    int blue = pixel & 0xff;
    int green = (pixel >> 8) & 0xff;
    int red = (pixel >> 16) & 0xff;
}
```

- **Hexadecimal number:** 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f
  - Note that f is 15, in binary this is 1111, one less than 10000
  - The hex number 0xff is an 8 bit number, all ones

- **The bitwise & operator creates an 8 bit value, 0—255 (why)**
  - 1&1 == 1, otherwise we get 0, similar to logical and
  - Similarly we have |, bitwise or
Bit operations revisited

- How do we write out all of the bits of a number

```c
/**
 * writes the bit representation of a int
 * to standard out
 */
void bits(int val)
{
```
Swap two ints “in place”

- Swap contents of two int variables without requiring extra memory
- Still requires three statements (same time on most machines)
- Replace

```c
void swap(int[] a, int j, int k){
    int temp = a[j];
    a[j] = a[k];
    a[k] = temp;
}
```

With

```c
void swap(int[] a, int j, int k){
    a[j] = a[j] ^ a[k];
    a[k] = a[j] ^ a[k];
    a[j] = a[j] ^ a[k];
}
```

- Works because \(x ^ x = 0, x ^ 0 = x\)
  - Proof left to the student...
  - Once was useful; now more of a curiosity