Introduction
This document introduces the use of bit operations in Java. These operations are the basis for manipulating the fundamental binary data structure that defines the storage of an image.

Operators

- The bitwise AND operation
  - Operator - &
  - bit-by-bit logical AND of two operands (you may think of as logical multiplication)
  - ex. 10100011 & 00101101 = 00100001

- The bitwise OR operation
  - Operator |
  - bit-by-bit logical OR of two operands (you may think of as logical addition)
  - ex. 10100011 | 00101101 = 10101111

- The bitwise NEG operation
  - Unary operation
  - Operator ~
  - bit-by-bit inversion of the operand (1-complement)
  - ex. ~10101010 = 01010101

- The bitwise XOR operation
  - Operator ^
  - bit-by-bit EXCLUSIVE OR of two operands
  - ex. 10100011 ^ 00101101 = 10001110

- The bitwise SHIFT LEFT
  - Operator <<
  - Shifts the left operand left the number of times indicated by the right operand
  - Zero (0) is shifted into the least significant bit for each shift
  - ex. 10101011 << 2 = 10101100

- The bitwise ARITHMETIC SHIFT RIGHT
  - Operator >>
  - Shifts the left operand right the number of times indicated by the right operand
  - Preserves the sign of the value by replicating the most significant bit for each shift
  - ex1. 01010101 >> 2 = 00010101 (keeps value positive)
  - ex2. 10101010 >> 2 = 11101010 (keeps value negative)

- The bitwise LOGICAL SHIFT RIGHT
  - Operator >>>
  - Shifts the left operand right the number of times indicated by the right operand
Logical shift — a zero (0) is shifted in to the most significant bit for each shift

ex. 01010101 >>> 2 = 00101011

Note the size of the operands data type determines the number of bits shifted and Java’s arithmetic rules apply. That is you may have a shorter value promoted to an integer if you are not careful to cast.

Hexadecimal Numbers: A Shorthand for Writing Binary Numbers

Any binary value expressed using a multiple of four bits may be written more compactly using hexadecimal notation. First observe that there are 16 hexadecimal digits, 0 – 9, A – F. Second observe that if we shift a binary number right 4 times we multiply it by 16. Hence, if we take an eight bit binary number, $b_7b_6b_5b_4b_3b_2b_1b_0$, the number may be broken into two four bits starting at the right (least significant) end. The bits $b_7b_6b_5b_4$ have the same numerical value as one of the hexadecimal digits, say $h_1$. Hence, $b_7b_6b_5b_40000$ may be written in hexadecimal as $h_1\times16$. It follows that the hexadecimal number $h_1h_2$, where $h_2$ is the hexadecimal digit whose value is the same as the binary number $b_3b_2b_1b_0$, is the hexadecimal representation of $b_7b_6b_5b_4b_3b_2b_1b_0$. Consequently, any binary number can be expressed in hexadecimal by breaking it into groups of four bits (nibbles) starting at the right hand end and replacing nibble with the equivalent hexadecimal digit.

Assignment Operators

There is an assignment operator for each of the bitwise operators. For example, the Java statement

```java
mask = mask << bitPlane; // shift left bitPlane times
```

is equivalent to

```java
mask <<= bitPlane; // shift left bitPlane times
```

For completeness the other bitwise assignment operators are listed below:

- &=
- |=
- ^=
- ~=
- >>=
- >>>=

Note that these operators are not necessary. The do, however, provide a convenient shorthand and are heavily used in code.
Example
Set a bit in a given bit plane in a grayscale pixel.

Suppose that the bitPlane is an integer that indicates the bit plane whose value we want to set and the integer variable bit contains the bit we want to sit in the grayscale pixel whose name is pixel. Note that the value of the integer bit will be one if we want to set the bit to 1 and zero otherwise. Likewise, we will assume that the type of pixel matches that of the other variables. That is it is an integer.

In order to set the correct bit we have to:

1. Clear (set to zero) the appropriate bit in pixel.
   a. Create a mask whose value is 1.
   b. Shift the 1 in the mask left to the correct bit plane position.
   c. Negate the mask.
   d. AND the mask with the pixel to set the appropriate bit to zero.

2. Set the bit in pixel that was just cleared to the value in bit.
   a. Shift bit left to the correct bit plane position.
   b. OR the value of the shifted bit into pixel.

The code that implements this algorithm follows. It is surprisingly compact.

```java
public int setGrayPixel(int pixel, int bit, int bitPlane) {
    // clear pixel’s bit in position bitPlane
    int mask = 0x1; // set mask to 1
    mask = ~(mask << pitPlanes);
    pixel &= mask;

    //Set pixel’s bit in position bitPlane to bit
    pixel |= (bit << bitPlane);
}
```

Note that no if statements are required.