K

Bit Manipulation

K.1 Introduction

This appendix presents an extensive discussion of bit-manipulation operators, followed by a discussion of class **BitSet**, which enables the creation of bit-array-like objects for setting and getting individual bit values. Java provides extensive bit-manipulation capabilities for programmers who need to get down to the "bits-and-bytes" level. Operating systems, test equipment software, networking software and many other kinds of software require that the programmer communicate "directly with the hardware." We now discuss Java's bit-manipulation capabilities and bitwise operators.

K.2 Bit Manipulation and the Bitwise Operators

Computers represent all data internally as sequences of bits. Each bit can assume the value `0` or the value `1`. On most systems, a sequence of eight bits forms a byte—the standard storage unit for a variable of type `byte`. Other types are stored in larger numbers of bytes. The bitwise operators can manipulate the bits of integral operands (i.e., operations of type `byte`, `char`, `short`, `int` and `long`), but not floating-point operands. The discussions of bitwise operators in this section show the binary representations of the integer operands.

The bitwise operators are **bitwise AND** (**&**), **bitwise inclusive OR** (**|**), **bitwise exclusive OR** (**^**), **left shift** (**<<**), **signed right shift** (**>>**), **unsigned right shift** (**>>>**) and **bitwise complement** (**~**). The bitwise AND, bitwise inclusive OR and bitwise exclusive OR operators compare their two operands bit by bit. The bitwise AND operator sets each bit in the result to `1` if and only if the corresponding bit in both operands is `1`. The bitwise inclusive OR operator sets each bit in the result to 1 if the corresponding bit in either (or both) operand(s) is `1`. The bitwise exclusive OR operator sets each bit in the result to `1` if the corresponding bit in exactly one operand is `1`. The left-shift operator shifts the bits of its left operand to the left by the number of bits specified in its right operand. The signed right shift operator shifts the bits in its left operand to the right by the number of bits specified in its right operand—if the left operand is negative, `1`s are shifted in from the left; otherwise, `0`s are shifted in from the left. The unsigned right shift operator shifts the bits in its left operand to the right by the number of bits specified in its right operand—`0`s are shifted in from the left. The bitwise complement operator sets all `0` bits in its operand to `1` in the result and sets all `1` bits in its operand to `0` in the result. The bitwise operators are summarized in Fig. K.1.

Operator	Name	Description
&	bitwise AND	The bits in the result are set to 1 if the corresponding bits in the two operands are both 1.
|	bitwise inclusive OR	The bits in the result are set to 1 if at least one of the corresponding bits in the two operands is 1.
^	bitwise exclusive OR	The bits in the result are set to 1 if exactly one of the corresponding bits in the two operands is 1.
<<	left shift	Shifts the bits of the left operand left by the number of bits specified by the right operand; fill from the right with 0.
>>	signed right shift	Shifts the bits of the left operand right by the number of bits specified by the right operand. If the left operand is negative, 1s are filled in from the left; otherwise, 0s are filled in from the left.
>>>	unsigned right shift	Shifts the bits of the left operand right by the number of bits specified by the second operand; 0s are filled in from the left.
~	bitwise complement	All 0 bits are set to 1, and all 1 bits are set to 0.

Fig. K.1 | Bitwise operators.

When using the bitwise operators, it's useful to display values in their binary representation to illustrate the effects of these operators. The application of Fig. K.2 allows the user to enter an integer from the standard input. Lines 8–10 read the integer from the standard input. The integer is displayed in its binary representation in groups of eight bits each. Often, the bitwise AND operator is used with an operand called a **mask**—an integer value with specific bits set to 1. Masks are used to hide some bits in a value while selecting other bits. In line 16, mask variable `displayMask` is assigned the value `1 << 31`, or

```
      10000000 00000000 00000000 00000000
```

Lines 19–28 obtains a string representation of the integer, in bits. Line 21 uses the bitwise AND operator to combine variable `input` with variable `displayMask`. The left-shift operator shifts the value `1` from the low-order (rightmost) bit to the high-order (leftmost) bit in `displayMask` and fills in `0` from the right.

```
 1   // Fig. K.2: PrintBits.java
 2   // Printing an unsigned integer in bits.
 3   import java.util.Scanner;
 4
 5   public class PrintBits {
 6      public static void main(String[] args) {
 7         // get input integer
 8         Scanner scanner = new Scanner(System.in);
 9         System.out.println("Please enter an integer:");
10         int input = scanner.nextInt();
```

Fig. K.2 | Printing the bits in an integer. (Part 1 of 2.)

```
11
12        // display bit representation of an integer
13        System.out.println("\nThe integer in bits is:");
14
15        // create int value with 1 in leftmost bit and 0s elsewhere
16        int displayMask = 1 << 31;
17
18        // for each bit display 0 or 1
19        for (int bit = 1; bit <= 32; bit++) {
20           // use displayMask to isolate bit
21           System.out.print((input & displayMask) == 0 ? '0' : '1' );
22
23           input <<= 1; // shift value one position to left
24
25           if (bit % 8 == 0) {
26              System.out.print(' '); // display space every 8 bits
27           }
28        }
29     }
30  }
```

```
Please enter an integer:
0

The integer in bits is:
00000000 00000000 00000000 00000000
```

```
Please enter an integer:
-1

The integer in bits is:
11111111 11111111 11111111 11111111
```

```
Please enter an integer:
65535

The integer in bits is:
00000000 00000000 11111111 11111111
```

Fig. K.2 | Printing the bits in an integer. (Part 2 of 2.)

Line 21 determines whether the current leftmost bit of variable `value` is a `1` or `0` and displays `'1'` or `'0'`, respectively, to the standard output. Assume that `input` contains `2000000000` (`01110111 00110101 10010100 00000000`). When `input` and `displayMask` are combined using `&`, all the bits except the high-order (leftmost) bit in variable `input` are “masked off” (hidden), because any bit “ANDed” with `0` yields `0`. If the leftmost bit is `1`, the expression `input & displayMask` evaluates to 1 and line 21 displays `'1'`; otherwise, line 21 displays `'0'`. Then line 23 left shifts variable `input` to the left by one bit with the expression `input <<= 1`. (This expression is equivalent to `input = input << 1`.) These steps are repeated for each bit in variable `input`. [*Note:* Class `Integer` provides method **`toBinaryString`**, which returns a string containing the binary representation of an integer.] Figure K.3 summarizes the results of combining two bits with the bitwise AND (&) operator.

Common Programming Error K.1

Using the conditional AND operator (&&) instead of the bitwise AND operator (&) is a compilation error.

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Fig. K.3 | Bitwise AND operator (&) combining two bits.

Figure K.4 demonstrates the bitwise AND operator, the bitwise inclusive OR operator, the bitwise exclusive OR operator and the bitwise complement operator. The program uses the `display` method of the utility class `BitRepresentation` (Fig. K.5) to get a string representation of the integer values. Notice that method `display` performs the same task as lines in Fig. K.2. Declaring `display` as a `static` method of class `BitRepresentation` allows display to be reused by later applications. The application of Fig. K.4 asks users to choose the operation they would like to test, gets input integer(s), performs the operation and displays the result of each operation in both integer and bitwise representations.

```java
// Fig. K.4: MiscBitOps.java
// Using the bitwise operators.
import java.util.Scanner;

public class MiscBitOps {
   public static void main(String[] args) {
      int choice = 0; // store operation type
      int first = 0; // store first input integer
      int second = 0; // store second input integer
      int result = 0; // store operation result
      Scanner scanner = new Scanner(System.in); // create Scanner

      // continue execution until user exit
      while (true) {
         // get selected operation
         System.out.println("\n\nPlease choose the operation:");
         System.out.printf("%s%s", "1--AND\n2--Inclusive OR\n",
            "3--Exclusive OR\n4--Complement\n5--Exit\n");
         choice = scanner.nextInt();
```

Fig. K.4 | Bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 1 of 4.)

```

         // perform bitwise operation
         switch (choice) {
            case 1: // AND
               System.out.print("Please enter two integers:");
               first = scanner.nextInt(); // get first input integer
               BitRepresentation.display(first);
               second = scanner.nextInt(); // get second input integer
               BitRepresentation.display(second);
               result = first & second; // perform bitwise AND
               System.out.printf(
                  "\n\n%d & %d = %d", first, second, result);
               BitRepresentation.display(result);
               break;
            case 2: // Inclusive OR
               System.out.print("Please enter two integers:");
               first = scanner.nextInt(); // get first input integer
               BitRepresentation.display(first);
               second = scanner.nextInt(); // get second input integer
               BitRepresentation.display(second);
               result = first | second; // perform bitwise inclusive OR
               System.out.printf(
                  "\n\n%d | %d = %d", first, second, result);
               BitRepresentation.display(result);
               break;
            case 3: // Exclusive OR
               System.out.print("Please enter two integers:");
               first = scanner.nextInt(); // get first input integer
               BitRepresentation.display(first);
               second = scanner.nextInt(); // get second input integer
               BitRepresentation.display(second);
               result = first ^ second; // perform bitwise exclusive OR
               System.out.printf(
                  "\n\n%d ^ %d = %d", first, second, result);
               BitRepresentation.display(result);
               break;
            case 4: // Complement
               System.out.print("Please enter one integer:");
               first = scanner.nextInt(); // get input integer
               BitRepresentation.display(first);
               result = ~first; // perform bitwise complement on first
               System.out.printf("\n\n~%d = %d", first, result);
               BitRepresentation.display(result);
               break;
            case 5: default:
               System.exit(0); // exit application
         }
      }
   }
}
```

Fig. K.4 | Bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 2 of 4.)

```
Please choose the operation:
1--AND
2--Inclusive OR
3--Exclusive OR
4--Complement
5--Exit
1
Please enter two integers:65535 1

Bit representation of 65535 is:
00000000 00000000 11111111 11111111
Bit representation of 1 is:
00000000 00000000 00000000 00000001

65535 & 1 = 1
Bit representation of 1 is:
00000000 00000000 00000000 00000001

Please choose the operation:
1--AND
2--Inclusive OR
3--Exclusive OR
4--Complement
5--Exit
2
Please enter two integers:15 241

Bit representation of 15 is:
00000000 00000000 00000000 00001111
Bit representation of 241 is:
00000000 00000000 00000000 11110001

15 | 241 = 255
Bit representation of 255 is:
00000000 00000000 00000000 11111111

Please choose the operation:
1--AND
2--Inclusive OR
3--Exclusive OR
4--Complement
5--Exit
3

Please enter two integers:139 199

Bit representation of 139 is:
00000000 00000000 00000000 10001011
Bit representation of 199 is:
00000000 00000000 00000000 11000111

139 ^ 199 = 76
Bit representation of 76 is:
00000000 00000000 00000000 01001100
```

Fig. K.4 | Bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 3 of 4.)

```
Please choose the operation:
1--AND
2--Inclusive OR
3--Exclusive OR
4--Complement
5--Exit
4
Please enter one integer:21845

Bit representation of 21845 is:
00000000 00000000 01010101 01010101

~21845 = -21846
Bit representation of -21846 is:
11111111 11111111 10101010 10101010
```

Fig. K.4 | Bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 4 of 4.)

```java
// Fig K.5: BitRepresentation.java
// Utility class that displays bit representation of an integer.

public class BitRepresentation {
   // display bit representation of specified int value
   public static void display(int value) {
      System.out.printf("\nBit representation of %d is: \n", value);

      // create int value with 1 in leftmost bit and 0s elsewhere
      int displayMask = 1 << 31;

      // for each bit display 0 or 1
      for (int bit = 1; bit <= 32; bit++) {
         // use displayMask to isolate bit
         System.out.print((value & displayMask) == 0 ? '0' : '1');

         value <<= 1; // shift value one position to left

         if (bit % 8 == 0) {
            System.out.print(' '); // display space every 8 bits
         }
      }
   }
}
```

Fig. K.5 | Utility class that displays bit representation of an integer.

The first output window in Fig. K.4 shows the results of combining the value 65535 and the value 1 with the bitwise AND operator (&; line 29). All the bits except the low-order bit in the value 65535 are "masked off" (hidden) by "ANDing" with the value 1.

The bitwise inclusive OR operator (|) sets each bit in the result to 1 if the corresponding bit in either (or both) operand(s) is 1. The second output window in Fig. K.4 shows the results of combining the value 15 and the value 241 by using the bitwise OR operator (line 40)—the result is `255`. Figure K.6 summarizes the results of combining two bits with the bitwise inclusive OR operator.

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Fig. K.6 | Bitwise inclusive OR operator (`|`) combining two bits.

The bitwise exclusive OR operator (^) sets each bit in the result to `1` if *exactly* one of the corresponding bits in its two operands is `1`. The third output window in Fig. K.4 shows the results of combining the value `139` and the value `199` by using the exclusive OR operator (line 51)—the result is `76`. Figure K.7 summarizes the results of combining two bits with the bitwise exclusive OR operator.

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

Fig. K.7 | Bitwise exclusive OR operator (`^`) combining two bits.

The bitwise complement operator (~) sets all `1` bits in its operand to `0` in the result and sets all `0` bits in its operand to `1` in the result—otherwise referred to as "taking the one's complement of the value." The fourth output window in Fig. K.4 shows the results of taking the one's complement of the value `21845` (line 60). The result is `-21846`.

Figure K.8 demonstrates the left-shift operator (<<), the signed right-shift operator (>>) and the unsigned right-shift operator (>>>). The application asks the user to enter an integer and choose the operation, then performs a one-bit shift and displays the results of the shift in both integer and bitwise representation. We use the utility class `BitRepresentation` (Fig. K.5) to display the bit representation of an integer.

```java
// Fig. K.8: BitShift.java
// Using the bitwise shift operators.
import java.util.Scanner;

public class BitShift {
   public static void main(String[] args) {
      int choice = 0; // store operation type
      int input = 0; // store input integer
      int result = 0; // store operation result
```

Fig. K.8 | Bitwise shift operations. (Part 1 of 3.)

```java
      Scanner scanner = new Scanner(System.in); // create Scanner

      // continue execution until user exit
      while (true) {
         // get shift operation
         System.out.println("\n\nPlease choose the shift operation:");
         System.out.println("1--Left Shift (<<)");
         System.out.println("2--Signed Right Shift (>>)");
         System.out.println("3--Unsigned Right Shift (>>>)");
         System.out.println("4--Exit");
         choice = scanner.nextInt();

         // perform shift operation
         switch (choice) {
            case 1: // <<
               System.out.println("Please enter an integer to shift:");
               input = scanner.nextInt(); // get input integer
               result = input << 1; // left shift one position
               System.out.printf("\n%d << 1 = %d", input, result);
               break;
            case 2: // >>
               System.out.println("Please enter an integer to shift:");
               input = scanner.nextInt(); // get input integer
               result = input >> 1; // signed right shift one position
               System.out.printf("\n%d >> 1 = %d", input, result);
               break;
            case 3: // >>>
               System.out.println("Please enter an integer to shift:");
               input = scanner.nextInt(); // get input integer
               result = input >>> 1; // unsigned right shift one position
               System.out.printf("\n%d >>> 1 = %d", input, result);
               break;
            case 4: default: // default operation is <<
               System.exit(0); // exit application
         }

         // display input integer and result in bits
         BitRepresentation.display(input);
         BitRepresentation.display(result);
      }
   }
}
```

```
Please choose the shift operation:
1--Left Shift (<<)
2--Signed Right Shift (>>)
3--Unsigned Right Shift (>>>)
4--Exit
1
Please enter an integer to shift:
1
```

Fig. K.8 | Bitwise shift operations. (Part 2 of 3.)

```
1 << 1 = 2
Bit representation of 1 is:
00000000 00000000 00000000 00000001
Bit representation of 2 is:
00000000 00000000 00000000 00000010

Please choose the shift operation:
1--Left Shift (<<)
2--Signed Right Shift (>>)
3--Unsigned Right Shift (>>>)
4--Exit
2
Please enter an integer to shift:
-2147483648

-2147483648 >> 1 = -1073741824
Bit representation of -2147483648 is:
10000000 00000000 00000000 00000000
Bit representation of -1073741824 is:
11000000 00000000 00000000 00000000

Please choose the shift operation:
1--Left Shift (<<)
2--Signed Right Shift (>>)
3--Unsigned Right Shift (>>>)
4--Exit
3
Please enter an integer to shift:
-2147483648

-2147483648 >>> 1 = 1073741824
Bit representation of -2147483648 is:
10000000 00000000 00000000 00000000
Bit representation of 1073741824 is:
01000000 00000000 00000000 00000000
```

Fig. K.8 | Bitwise shift operations. (Part 3 of 3.)

The left-shift operator (<<) shifts the bits of its left operand to the left by the number of bits specified in its right operand (performed at line 27 in Fig. K.8). Bits vacated to the right are replaced with `0`s; `1`s shifted off the left are lost. The first output window in Fig. K.8 demonstrates the left-shift operator. Starting with the value `1`, the left shift operation was chosen, resulting in the value `2`.

The signed right-shift operator (>>) shifts the bits of its left operand to the right by the number of bits specified in its right operand (performed at line 33 in Fig. K.8). Performing a right shift causes the vacated bits at the left to be replaced by `0`s if the number is positive or by `1`s if the number is negative. Any `1`s shifted off the right are lost. Next, the output window the results of signed right shifting the value `-2147483648`, which is the value `1` being left shifted `31` times. Notice that the leftmost bit is replaced by `1` because the number is negative.

The unsigned right-shift operator (>>>) shifts the bits of its left operand to the right by the number of bits specified in its right operand (performed at line 39 in Fig. K.8). Performing an unsigned right shift causes the vacated bits at the left to be replaced by `0`s. Any `1`s shifted off the right are lost. The third output window of Fig. K.8 shows the results of unsigned right shifting the value `-2147483648`. Notice that the leftmost bit is replaced by

0. Each bitwise operator (except the bitwise complement operator) has a corresponding assignment operator. These **bitwise assignment operators** are shown in Fig. K.9.

Bitwise assignment operators	
`&=`	Bitwise AND assignment operator.
`|=`	Bitwise inclusive OR assignment operator.
`^=`	Bitwise exclusive OR assignment operator.
`<<=`	Left-shift assignment operator.
`>>=`	Signed right-shift assignment operator.
`>>>=`	Unsigned right-shift assignment operator.

Fig. K.9 | Bitwise assignment operators.

K.3 BitSet Class

Class `BitSet` makes it easy to create and manipulate **bit sets**, which are useful for representing sets of `boolean` flags. `BitSet`s are dynamically resizable—more bits can be added as needed, and a `BitSet` will grow to accommodate the additional bits. Class `BitSet` provides two constructors—a no-argument constructor that creates an empty `BitSet` and a constructor that receives an integer representing the number of bits in the `BitSet`. By default, each bit in a `BitSet` has a `false` value—the underlying bit has the value `0`. A bit is set to `true` (also called "on") with a call to `BitSet` method **`set`**, which receives the index of the bit to set as an argument. This makes the underlying value of that bit `1`. Bit indices are zero based, like arrays. A bit is set to `false` (also called "off") by calling `BitSet` method **`clear`**. This makes the underlying value of that bit `0`. To obtain the value of a bit, use `BitSet` method **`get`**, which receives the index of the bit to get and returns a boolean value representing whether the bit at that index is on (`true`) or off (`false`).

Class `BitSet` also provides methods for combining the bits in two `BitSet`s, using bitwise logical AND (**`and`**), bitwise logical inclusive OR (**`or`**), and bitwise logical exclusive OR (**`xor`**). Assuming that `b1` and `b2` are `BitSet`s, the statement

```
    b1.and(b2);
```

performs a bit-by-bit logical AND operation between `BitSet`s `b1` and `b2`. The result is stored in `b1`. When `b2` has more bits than `b1`, the extra bits of `b2` are ignored. Hence, the size of `b1` remain unchanged. Bitwise logical inclusive OR and bitwise logical exclusive OR are performed by the statements

```
    b1.or(b2);
    b1.xor(b2);
```

When `b2` has more bits than `b1`, the extra bits of `b2` are ignored. Hence the size of `b1` remains unchanged.

`BitSet` method **`size`** returns the number of bits in a `BitSet`. `BitSet` method **`equals`** compares two `BitSet`s for equality. Two `BitSet`s are equal if and only if each `BitSet` has identical values in corresponding bits. `BitSet` method **`toString`** creates a string representation of a `BitSet`'s contents.

Figure K.10 implements the Sieve of Eratosthenes (for finding prime numbers). We use a BitSet to implement the algorithm. The application asks the user to enter an integer between 2 and 1023, displays all the prime numbers from 2 to 1023 and determines whether that number is prime.

```java
// Fig. K.10: BitSetTest.java
// Using a BitSet to demonstrate the Sieve of Eratosthenes.
import java.util.BitSet;
import java.util.Scanner;

public class BitSetTest {
   public static void main(String[] args) {
      // get input integer
      Scanner scanner = new Scanner(System.in);
      System.out.println("Please enter an integer from 2 to 1023");
      int input = scanner.nextInt();

      // perform Sieve of Eratosthenes
      BitSet sieve = new BitSet(1024);
      int size = sieve.size();

      // set all bits from 2 to 1023
      for (int i = 2; i < size; i++) {
         sieve.set(i);
      }

      // perform Sieve of Eratosthenes
      int finalBit = (int) Math.sqrt(size);

      for (int i = 2; i < finalBit; i++) {
         if (sieve.get(i)  ) {
            for (int j = 2 * i; j < size; j += i) {
               sieve.clear(j);
            }
         }
      }

      int counter = 0;

      // display prime numbers from 2 to 1023
      for (int i = 2; i < size; i++) {
         if (sieve.get(i)  ) {
            System.out.print(String.valueOf(i));
            System.out.print(++counter % 7 == 0 ? "\n" : "\t");
         }
      }

      // display result
      if (sieve.get(input)  ) {
         System.out.printf("\n%d is a prime number", input);
      }
```

Fig. K.10 | Sieve of Eratosthenes, using a BitSet. (Part 1 of 2.)

```
47          else {
48             System.out.printf("\n%d is not a prime number", input);
49          }
50       }
51    }
```

```
Please enter an integer from 2 to 1023
773
2       3       5       7       11      13      17
19      23      29      31      37      41      43
47      53      59      61      67      71      73
79      83      89      97      101     103     107
109     113     127     131     137     139     149
151     157     163     167     173     179     181
191     193     197     199     211     223     227
229     233     239     241     251     257     263
269     271     277     281     283     293     307
311     313     317     331     337     347     349
353     359     367     373     379     383     389
397     401     409     419     421     431     433
439     443     449     457     461     463     467
479     487     491     499     503     509     521
523     541     547     557     563     569     571
577     587     593     599     601     607     613
617     619     631     641     643     647     653
659     661     673     677     683     691     701
709     719     727     733     739     743     751
757     761     769     773     787     797     809
811     821     823     827     829     839     853
857     859     863     877     881     883     887
907     911     919     929     937     941     947
953     967     971     977     983     991     997
1009    1013    1019    1021
773 is a prime number
```

Fig. K.10 | Sieve of Eratosthenes, using a `BitSet`. (Part 2 of 2.)

Line 14 creates a `BitSet` of 1024 bits. We ignore the bits at indices zero and one in this application. Lines 18–20 set all the bits in the `BitSet` to “on” with `BitSet` method `set`. Lines 23–31 determine all the prime numbers from 2 to 1023. The integer `finalBit` specifies when the algorithm is complete. The basic algorithm is that a number is prime if it has no divisors other than 1 and itself. Starting with the number 2, once we know that a number is prime, we can eliminate all multiples of that number. The number 2 is divisible only by 1 and itself, so it’s prime. Therefore, we can eliminate 4, 6, 8 and so on. Elimination of a value consists of setting its bit to “off” with `BitSet` method `clear` (line 28). The number 3 is divisible by 1 and itself. Therefore, we can eliminate all multiples of 3. (Keep in mind that all even numbers have already been eliminated.) After the list of primes is displayed, lines 44–49 use `BitSet` method `get` (line 44) to determine whether the bit for the number the user entered is set. If so, line 45 displays a message indicating that the number is prime.