I

Manipulação de bits

I.1 Introdução

Este apêndice apresenta uma extensa discussão sobre os operadores de manipulação de bits e também sobre a classe `BitSet` que permite a criação de objetos do tipo array de bits para configurar e obter valores de bits individuais. O Java fornece várias capacidades de manipulação de bits para programadores que precisam descer ao nível dos 'bits e bytes'. Sistemas operacionais, software de equipamento de teste, software de rede e muitos outros tipos de software exigem que o programador se comunique 'diretamente com o hardware'. Agora, discutiremos as capacidades de manipulação de bits do Java e os operadores de bits.

I.2 Manipulação de bits e os operadores de bits

Os computadores representam todos os dados internamente como seqüências de bits. Cada bit pode assumir o valor `0` ou o valor `1`. Na maioria dos sistemas, uma seqüência de oito bits forma um byte — a unidade de armazenamento padrão para uma variável do tipo `byte`. Outros tipos são armazenados em números maiores de bytes. Os operadores de bits podem manipular os bits de operandos integrais (operações do tipo `byte`, `char`, `short`, `int` e `long`), mas não os operandos de ponto flutuante.

Observe que as discussões sobre operador de bits nesta seção mostram as representações binárias dos operandos inteiros. Para uma explicação detalhada do sistema de números binário (também chamado de base 2), veja o Apêndice E, "Sistemas de numeração".

Os operadores de bits são **E sobre bits** (`&`), **OU inclusivo sobre bits** (`|`), **OU exclusivo sobre bits** (`^`), **deslocamento para a esquerda** (`<<`), **deslocamento para a direita com sinal** (`>>`), **deslocamento para a direita sem sinal** (`>>>`) e **complemento de bits** (`~`). Os operadores E sobre bits, OU inclusivo sobre bits e OU exclusivo sobre bits comparam seus dois operandos bit a bit. O operador E sobre bits configura cada bit no resultado como `1` se e somente se o bit correspondente nos dois operandos for `1`. O operador OU inclusivo sobre bits (`|`) configura cada bit no resultado como 1 se o bit correspondente em qualquer (ou ambos os) operando(s) for `1`. O operador OU exclusivo sobre bits configura cada bit no resultado como `1` se o bit correspondente em exatamente um operando for `1`. O operador de deslocamento para a esquerda desloca os bits de seu operando esquerdo para a esquerda pelo número de bits especificado em seu operando direito. O operador de deslocamento para a direita com sinal muda os bits em seu operando esquerdo para a direita pelo número de bits especificado em seu operando direito — se o operando esquerdo for negativo, `1`s são deslocados da esquerda; caso contrário, `0`s são deslocados da esquerda. O operador de deslocamento para a direita sem sinal desloca os bits no seu operando esquerdo para a direita de acordo com o número de bits especificado no seu operando à direita — `0`s são deslocados a partir da esquerda. O operador de complemento de bits configura todos os bits `0` em seu operando como `1` no resultado e configura todos os bits `1` em seu operando como `0` no resultado. Os operadores de bits estão resumidos na Figura I.1.

Quando se utilizam operadores de bits, é útil exibir valores em sua representação binária para ilustrar os efeitos desses operadores. O aplicativo da Figura I.2 permite que o usuário insira um inteiro pela entrada-padrão. As linhas 10–12 lêem o inteiro pela entrada-padrão. O inteiro é exibido em sua representação binária em grupos de oito bits cada. Freqüentemente, o operador E sobre bits é utilizado com um operando chamado de **máscara** — um valor inteiro com bits específicos configurados como `1`. As máscaras são utilizadas para ocultar alguns bits em um valor enquanto se selecionam outros bits. Na linha 18, é atribuído à variável de máscara `displayMask` o valor `1 << 31`, ou

```
10000000 00000000 00000000 00000000
```

As linhas 21–30 obtêm uma representação de string do inteiro, em bits. A linha 24 utiliza o operador E sobre bits para combinar a variável `input` com a variável `displayMask`. O operador de deslocamento para a esquerda desloca o valor `1` do bit de ordem inferior (mais à direita) para o bit de ordem superior (mais à esquerda) em `displayMask` e preenche com `0`s a partir da direita.

A linha 24 determina se o bit atual mais à esquerda da variável `value` é um `1` ou `0` e exibe `'1'` ou `'0'`, respectivamente, na saída-padrão. Suponha que `input` contenha `2000000000` (`01110111 00110101 10010100 00000000`). Quando `input` e `displayMask` são combinados utilizando `&`, todos os bits exceto o bit de ordem superior (mais à esquerda) na variável `input` são 'mascarados' (ocultos),

porque qualquer bit submetido ao operador E com 0 produz 0. Se o bit mais à esquerda for 1, a expressão `input & displayMask` é avaliada como 1 e a linha 24 exibe `'1'`; caso contrário, a linha 24 exibe `'0'`. Então, a linha 26 desloca a variável `input` para a esquerda por um bit com a expressão `input <<= 1`. (Essa expressão é equivalente a `input = input << 1`.) Esses passos são repetidos para cada bit na variável `input`. [*Nota*: A classe `Integer` fornece o método **`toBinaryString`** que retorna uma string contendo a representação binária de um inteiro.] A Figura I.3 resume os resultados da combinação de dois bits com o operador E sobre bits (`&`).

Operador	Nome	Descrição
`&`	E sobre bits	Os bits no resultado são configurados como 1 se os bits correspondentes nos dois operandos forem ambos 1.
`|`	OU inclusivo sobre bits	Os bits no resultado são configurados como 1 se pelo menos um dos bits correspondentes nos dois operandos for 1.
`^`	OU exclusivo sobre bits	Os bits no resultado são configurados como 1 se exatamente um dos bits correspondentes nos dois operandos for 1.
`<<`	deslocamento de bits para a esquerda	Desloca os bits do primeiro operando esquerdo pelo número de bits especificado pelo segundo operando; preenche a partir da direita com 0s.
`>>`	deslocamento para a direita com sinal	Desloca os bits do primeiro operando direito pelo número de bits especificado pelo segundo operando. Se o primeiro operando for negativo, os 1s são preenchidos a partir da esquerda; caso contrário, os 0s são preenchidos a partir da esquerda.
`>>>`	deslocamento para a direita sem sinal	Desloca os bits do primeiro operando direito pelo número de bits especificado pelo segundo operando; 0s são preenchidos a partir da esquerda.
`~`	complemento de bits	Todos os bits 0 são configurados como 1, e todos os bits 1 são configurados como 0.

Figura. I.1 Operadores de bits.

```
1  // Fig. I.2: PrintBits.java
2  // Imprimindo um inteiro sem sinal em bits.
3  import java.util.Scanner;
4
5  public class PrintBits
6  {
7     public static void main( String args[] )
8     {
9        // obtém o inteiro de entrada
10       Scanner scanner = new Scanner( System.in );
11       System.out.println( "Please enter an integer:" );
12       int input = scanner.nextInt();
13
14       // exibe a representação em bits de um inteiro
15       System.out.println( "\nThe integer in bits is:" );
16
17       // cria um valor inteiro com 1 no bit mais à esquerda e 0s em outros locais
18       int displayMask = 1 << 31;
19
20       // para cada bit exibe 0 ou 1
21       for ( int bit = 1; bit <= 32; bit++ )
22       {
23          // utiliza displayMask para isolar o bit
24          System.out.print(( input & displayMask ) == 0 ? '0' : '1');
25
26          input <<= 1; // desloca o valor uma posição para a esquerda
27
28          if ( bit % 8 == 0 )
29             System.out.print( ' ' ); // exibe espaço a cada 8 bits
30       } // fim do for
31    } // fim de main
32 } // fim da classe PrintBits
```

Figura I.2 Imprimindo os bits em um inteiro. (Parte 1 de 2.)

```
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 1111111
```

Figura I.2 Imprimindo os bits em um inteiro. (Parte 2 de 2.)

Erro de programação comum I.1

Utilizando o operador E condicional (&&) em vez do operador E sobre bits (&).

A Figura I.4 demonstra o operador E sobre bits, o operador OU inclusivo sobre bits, o operador OU exclusivo sobre bits e o operador de complemento de bits. O programa utiliza o método `display` (linhas 7–25) da classe de utilitário `BitRepresentation` (Figura I.5) para obter uma representação de string dos valores de inteiro. Observe que o método `display` realiza a mesma tarefa que as linhas 17–30 na Figura I.2. Declarar `display` como um método `static` da classe `BitRepresentation` permite que `display` seja reutilizado por outros aplicativos. O aplicativo da Figura I.4 pede para os usuários escolherem a operação que eles querem testar, obtém o(s) inteiro(s) da entrada, realiza a operação e exibe o resultado de cada operação tanto na representação de inteiros como na de bits.

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

Figura I.3 Operador E sobre bits (&) combinando dois bits.

```
1  // Fig. I.4: MiscBitOps.java
2  // Utilizando os operadores de deslocamento de bits.
3  import java.util.Scanner;
4
5  public class MiscBitOps
6  {
7     public static void main( String args[] )
8     {
9        int choice = 0; // armazena o tipo de operação
10       int first = 0; // armazena o primeiro inteiro da entrada
11       int second = 0; // armazena o segundo inteiro da entrada
12       int result = 0; // resultado da operação de armazenamento
13       Scanner scanner = new Scanner( System.in ); // cria o Scanner
14
15       // continua a execução até o usuário sair
16       while ( true )
17       {
18          // obtém a operação selecionada
19          System.out.println( "\n\nPlease choose the operation:" );
```

Figura I.4 Operadores E sobre bits, OU inclusivo sobre bits, OU exclusivo sobre bits e de complemento de bits. (Parte 1 de 4.)

```
20          System.out.printf( "%s%s", "1-AND\n2-Inclusive OR\n",
21             "3-Exclusive OR\n4-Complement\n5-Exit\n" );
22          choice = scanner.nextInt();
23
24          // realiza E sobre bits
25          switch ( choice )
26          {
27             case 1: // E
28                System.out.print( "Please enter two integers:" );
29                first = scanner.nextInt(); // obtém o primeiro inteiro de entrada
30                BitRepresentation.display( first );
31                second = scanner.nextInt(); // obtém o segundo inteiro de entrada
32                BitRepresentation.display( second );
33                result = first & second; // realiza E sobre bits
34                System.out.printf(
35                   "\n\n%d & %d = %d", first, second, result );
36                BitRepresentation.display( result );
37                break;
38             case 2: // OU inclusivo
39                System.out.print( "Please enter two integers:" );
40                first = scanner.nextInt(); // obtém o primeiro inteiro de entrada
41                BitRepresentation.display( first );
42                second = scanner.nextInt(); // obtém o segundo inteiro de entrada
43                BitRepresentation.display( second );
44                result = first | second; // realiza OU inclusivo sobre bits
45                System.out.printf(
46                   "\n\n%d | %d = %d", first, second, result );
47                BitRepresentation.display( result );
48                break;
49             case 3: // OU exclusivo
50                System.out.print( "Please enter two integers:" );
51                first = scanner.nextInt(); // obtém o primeiro inteiro de entrada
52                BitRepresentation.display( first );
53                second = scanner.nextInt(); // obtém o segundo inteiro de entrada
54                BitRepresentation.display( second );
55                result = first ^ second; // realiza OU exclusivo sobre  bits
56                System.out.printf(
57                   "\n\n%d ^ %d = %d", first, second, result );
58                BitRepresentation.display( result );
59                break;
60             case 4: // Complemento
61                System.out.print( "Please enter one integer:" );
62                first = scanner.nextInt(); // obtém o inteiro de entrada
63                BitRepresentation.display( first );
64                result = ~first; // realiza o complemento de bits no primeiro
65                System.out.printf( "\n\n~%d = %d", first, result );
66                BitRepresentation.display( result );
67                break;
68             case 5: default:
69                System.exit( 0 ); // encerra o aplicativo
70          } // fim de switch
71       } // fim do while
72    } // fim de main
73 } // fim da classe MiscBitOps
```

Figura I.4 Operadores E sobre bits, OU inclusivo sobre bits, OU exclusivo sobre bits e de complemento de bits. (Parte 2 de 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 BitSet sieve = new BitSet( 1024 );1

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

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
```

Figura I.4 Operadores E sobre bits, OU inclusivo sobre bits, OU exclusivo sobre bits e de complemento de bits. (Parte 3 de 4.)

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

Figura I.4 Operadores E sobre bits, OU inclusivo sobre bits Ou exclusivo sobre bits e de complemento de bits. (Parte 4 de 4.)

```
1  // Figura I.5: BitRepresentation.Java
2  // Classe de utilitário que exibe a representação de bits de um inteiro.
3
4  public class BitRepresentation
5  {
6     // exibe a representação de bits do valor inteiro especificado
7     public static void display( int value )
8     {
9        System.out.printf( "\nBit representation of %d is: \n", value );
10
11       // cria um valor inteiro com 1 no bit mais à esquerda e 0s em outros locais
12       int displayMask = 1 << 31;
13
14       // para cada bit exibe 0 ou 1
15       for ( int bit = 1; bit <= 32; bit++ )
16       {
17          // utiliza displayMask para isolar o bit
18          System.out.print( ( value & displayMask ) == 0 ? '0' : '1' );
19
20          value <<= 1; // desloca o valor uma posição para a esquerda
21
22          if ( bit % 8 == 0 )
23             System.out.print( ' ' ); // exibe espaço a cada 8 bits
24       } // fim do for
25    } // fim do método display
26 } // fim da classe BitRepresentation
```

Figura I.5 A classe utilitária que exibe a representação de bits de um inteiro.

A primeira janela da saída na Figura I.4 mostra os resultados da combinação do valor `65535` e do valor `1` com o operador E sobre bits (&; linha 33). Todos os bits exceto o bit de ordem inferior no valor `65535` são 'mascarados' (ocultos) pela operação E com o valor `1`.

O operador OU inclusivo sobre bits (|) configura cada bit no resultado como 1 se o bit correspondente em qualquer (ou ambos os) operando(s) for `1`. A segunda janela de saída na Figura I.4 mostra os resultados da combinação do valor `15` e do valor `241` utilizando o operador OU sobre bits (linha 44) — o resultado é `255`. A Figura I.6 resume os resultados da combinação de dois bits com o operador OU inclusivo sobre bits.

O operador OU exclusivo sobre bits (^) configura cada bit no resultado como `1` se *exatamente* um dos bits correspondentes nos seus dois operandos for `1`. A terceira janela de saída na Figura I.4 mostra os resultados da combinação do valor `139` e o do valor `199` utilizando o operador OU exclusivo (linha 55) — o resultado é `76`. A Figura I.7 resume resultados da combinação de dois bits com o operador OU exclusivo sobre bits.

O operador de complemento de bits (~) configura todos os bits `1` em seu operando como `0` no resultado e configura todos os bits `0` em seu operando como `1` no resultado — também referido como 'obter o complemento de um do valor'. A quarta janela de saída na Figura I.4 mostra os resultados da seleção complemento de um do valor `21845` (linha 64). O resultado é `-21846`.

O aplicativo da Figura I.8 demonstra o operador de deslocamento para a esquerda (<<), o operador de deslocamento para a direita com sinal (>>>) e o operador de deslocamento para a direita sem sinal (>>>). O aplicativo pede para o usuário inserir um inteiro e escolher a operação, então realiza o deslocamento de um bit e exibe os resultados desse deslocamento tanto na representação de inteiros como na de bits. Utilizamos a classe utilitária `BitRepresentation` (Figura I.5) para exibir a representação de bits de um inteiro.

O operador de deslocamento para a esquerda (<<) desloca os bits do seu operando esquerdo para a esquerda de acordo com o número de bits especificado no seu operando direito (realizado na linha 31 na Figura I.8). Os bits vagos à direita são substituídos por `0`s; `1`s deslocados à esquerda são perdidos. A primeira janela de saída na Figura I.8 demonstra o operador de deslocamento para a esquerda. Iniciando com o valor `1`, a operação de deslocamento para a esquerda foi escolhida, resultando no valor `2`.

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

Figura I.6 O operador OU inclusivo sobre bits (|) combinando dois bits.

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

Figura I.7 O operador OU exclusivo sobre bits (^) combinando dois bits.

```
1  // Fig. I.08: BitShift.java
2  // Utilizando os operadores de deslocamento de bits.
3  import java.util.Scanner;
4
5  public class BitShift
6  {
7     public static void main( String args[] )
8     {
9        int choice = 0; // armazena o tipo de operação
10       int input = 0; // armazena o inteiro de entrada
11       int result = 0; // resultado da operação de armazenamento
12       Scanner scanner = new Scanner( System.in ); // cria o Scanner
13
14       // continua a execução até o usuário sair
15       while ( true )
16       {
17          // obtém a operação de deslocamento
18          System.out.println( "\n\nPlease choose the shift operation:" );
19          System.out.println( "1–Left Shift (<<)" );
20          System.out.println( "2–Signed Right Shift (>>)" );
21          System.out.println( "3–Unsigned Right Shift (>>>)" );
22          System.out.println( "4–Exit" );
23          choice = scanner.nextInt();
24
25          // realiza a operação de deslocamento
26          switch ( choice )
27          {
28             case 1: // <<
29                System.out.println( "Please enter an integer to shift:" );
30                input = scanner.nextInt(); // obtém o inteiro de entrada
31                result = input << 1; // desloca para a esquerda uma posição
32                System.out.printf( "\n%d << 1 = %d", input, result );
33                break;
34             case 2: // >>
35                System.out.println( "Please enter an integer to shift:" );
36                input = scanner.nextInt(); // obtém o inteiro de entrada
37                result = input >> 1; // desloca para a direita com sinal uma posição
38                System.out.printf( "\n%d >> 1 = %d", input, result );
39                break;
40             case 3: // >>>
```

Figura I.8 Operações de deslocamento de bits. (Parte 1 de 2.)

```
41                System.out.println( "Please enter an integer to shift:" );
42                input = scanner.nextInt(); // obtém o inteiro de entrada
43                result = input >>> 1; // desloca para a direita sem sinal uma posição
44                System.out.printf( "\n%d >>> 1 = %d", input, result );
45                break;
46             case 4: default: // a operação padrão é <<
47                System.exit( 0 ); // exit application
48          } // fim de switch
49
50          // exibe o inteiro de entrada e o resultado em bits
51          BitRepresentation.display( input );
52          BitRepresentation.display( result );
53       } // fim do while
54    } // fim de main
55 } // fim da classe BitShift
```

```
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

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
```

Figura I.8 Operações de deslocamento de bits. (Parte 2 de 2.)

O operador de deslocamento para a direita com sinal (>>) desloca os bits do seu operando esquerdo para a direita de acordo com o número de bits especificado no seu operando direito (realizado na linha 37 na Figura I.8). Realizar um deslocamento para a direita faz com que os bits vagos à esquerda sejam substituídos por 0s se o número for positivo ou por 1s se o número for negativo. Quaisquer 1s deslocados para a direita são perdidos. A segunda janela de saída na Figura I.8 mostra os resultados do deslocamento para a direita com sinal do valor -2147483648, que é o valor 1 sendo deslocado para a esquerda 31 vezes. Observe que o bit mais à esquerda é substituído por 1 porque o número é negativo.

O operador de deslocamento para a direita sem sinal (>>>) desloca os bits do seu operando esquerdo para a direita de acordo com o número de bits especificado no seu operando direito (realizado na linha 43 da Figura I.8). Realizar um deslocamento para a direita faz com que os bits vagos à esquerda sejam substituídos por 0s se o número for positivo ou por 1s se o número for negativo. Quaisquer 1s deslocados para a direita são perdidos. A terceira janela de saída da Figura I.8 mostra os resultados do deslocamento para a direita sem sinal do valor -2147483648. Observe que o bit mais à esquerda é substituído por 0. Cada operador de bits (exceto o operador de complemento de bits) tem um operador de atribuição correspondente. Esses **operadores de atribuição de bits** são mostrados na Figura I.9.

I.3 Classe BitSet

A classe BitSet facilita criar e manipular **conjuntos de bits**, que são úteis para representar conjuntos de flags boolean. As classes BitSet são dinamicamente redimensionáveis — mais bits podem ser adicionados conforme necessário e um BitSet aumentará a fim de acomodar os bits adicionais. A classe BitSet fornece dois construtores — um construtor sem argumento que cria um BitSet vazio e um construtor que recebe um inteiro que representa o número de bits no BitSet. Por padrão, cada bit em um BitSet tem um valor false — o bit subjacente tem o valor 0. Um bit é configurado como true (também chamado de 'ativado') com uma chamada ao método **set** de BitSet, que recebe o índice do bit para configurar como um argumento. Isso torna o valor subjacente desse bit 1. Observe que índices de bits são baseados em zeros, como os arrays. Um bit é configurado como false (também chamado de 'desativado') chamando o método **clear** de BitSet. Isso torna o valor subjacente desse bit 0. Para obter o valor de um bit, utilize o método **get** de BitSet, que recebe o índice do bit para obter e retornar um valor booleano representando se o bit nesse índice está ativado (true) ou desativado (false).

A classe BitSet também fornece os métodos para combinar os bits em dois BitSet, utilizando o E lógico sobre bits (**and**), o OU inclusivo lógico sobre bits (**or**) e o OU exclusivo lógico sobre bits (**xor**). Supondo que b1 e b2 sejam BitSets, a instrução

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

realiza uma operação E lógico bit a bit entre os BitSets b1 e b2. O resultado é armazenado em b1. Quando b2 tem mais bits do que b1, os bits extras de b2 são ignorados. Conseqüentemente, o tamanho de b1 permanece inalterado. O OU inclusivo lógico sobre bits e o OU exclusivo lógico sobre bits são realizados pelas instruções

Operadores de atribuição de bits	
&=	Operador de atribuição E sobre bits.
|=	Operador de atribuição OU inclusivo sobre bits.
^=	Operador de atribuição OU exclusivo sobre bits.
<<=	Operador de atribuição de deslocamento para a esquerda.
>>=	Operador de atribuição de deslocamento para a direita com sinal.
>>>=	Operador de atribuição de deslocamento para a direita sem sinal.

Figura I.9 Operadores de atribuição de bits.

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

Quando b2 tem mais bits do que b1, os bits extras de b2 são ignorados. Conseqüentemente, o tamanho de b1 permanece inalterado.

O método **size** de BitSet retorna o número de bits em um BitSet. O método **equals** de BitSet compara dois BitSets quanto à igualdade. Dois BitSets são iguais se, e somente se, cada BitSet tiver valores idênticos nos bits correspondentes. O método **toString** de BitSet cria uma representação de string do conteúdo de um BitSet.

A Figura I.10 reexamina o Crivo de Eratóstenes (para encontrar números primos), que discutimos no Exercício 7.27. Este exemplo utiliza um BitSet em vez de um array para implementar o algoritmo. O aplicativo pede para o usuário inserir um inteiro entre 2 e 1023, exibe todos os números primos de 2–1023 e determina se esse número é primo.

```
1  // Fig. I.10: BitSetTest.java
2  // Utilizando um BitSet para demonstrar a Peneira de Eratóstenes.
3  import java.util.BitSet;
4  import java.util.Scanner;
5
6  public class BitSetTest
```

Figura I.10 Crivo de Eratóstenes, utilizando um BitSet. (Parte 1 de 3.)

```
{
   public static void main( String args[] )
   {
      // obtém o inteiro de entrada
      Scanner scanner = new Scanner( System.in );
      System.out.println( "Please enter an integer from 2 to 1023" );
      int input = scanner.nextInt();

                // realiza a Peneira de Eratóstenes
      BitSet sieve = new BitSet( 1024 );
      int size = sieve.size();

      // configura todos os bits de 2 a 1023
      for ( int i = 2; i < size; i++ )
         sieve.set( i );

      // realiza a Peneira de Eratóstenes
      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 );
         } // fim do if
      } // fim do for

      int counter = 0;

      // exibe os números primos entre 2 e 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" );
         } // fim do if
      } // fim do for

      // exibe o resultado
      if ( sieve.get( input) )
         System.out.printf( "\n%d is a prime number", input );
      else
         System.out.printf( "\n%d is not a prime number", input );
   } // fim de main
} // fim da classe BitSetTest
```

```
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      699     211     223     227
229     233     239     21      67      71      73
79      83      89      97      101     103     107
109     113     127     131     137     139     149
```

Figura I.10 Crivo de Eratóstenes, utilizando um `BitSet`. (Parte 2 de 3.)

```
151    157    163    167    173    179    181
191    193    197    141    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
```

Figura I.10 Crivo de Eratóstenes, utilizando um `BitSet`. (Parte 3 de 3.)

A linha 16 cria um `BitSet` de 1024 bits. Ignoramos os bits nos índices de zero e um nesse aplicativo. As linhas 20–21 configuram todos os bits no `BitSet` como 'ativados' com o método `set` de `BitSet`. As linhas 24–33 determinam todos os números primos entre 2 e 1023. O inteiro `finalBit` especifica quando o algoritmo está completo. O algoritmo básico é que um número é primo se ele não tiver nenhum divisor diferente de 1 e dele próprio. Começando com o número 2, uma vez que sabemos que é um número, podemos eliminar todos os múltiplos desse número. O número 2 é divisível somente por 1 e por ele próprio, então é primo. Portanto, podemos eliminar 4, 6, 8 e assim por diante. A eliminação de um valor consiste em configurar seu bit como 'desativado' com o método `clear` de `BitSet` (linha 31). O número 3 é divisível por 1 e por si próprio. Portanto, podemos eliminar todos os múltiplos de 3. (Tenha em mente que todos os números pares já foram eliminados). Depois que a lista de primos é exibida, as linhas 48–51 utilizam o método `get` de `BitSet` (linha 48) para determinar se o bit para o número que o usuário inseriu está configurado. Se estiver, a linha 49 exibe uma mensagem indicando que o número é primo. Caso contrário, a linha 51 exibe uma mensagem indicando que o número não é primo.

Resumo

- O operador E (`&`) sobre bits configura cada bit no resultado como `1` se o bit correspondente em ambos os operandos for `1`.
- O operador OU inclusivo sobre bits (`|`) configura cada bit no resultado como `1` se o bit correspondente em qualquer (ou ambos) operando (s) for `1`. O operador OU exclusivo sobre bits (`^`) configura cada bit no resultado como `1` se o bit correspondente em exatamente um operando for `1`.
- O operador de deslocamento para a esquerda (`<<`) desloca os bits de seu operando esquerdo para a esquerda pelo número de bits especificado em seu operando direito.
- O operador de deslocamento para a direita com sinal (`>>`) desloca os bits no seu operando esquerdo para a direita de acordo com o número de bits especificado no seu operando direito — se o operando esquerdo for negativo, `1`s são preenchidos a partir da esquerda; caso contrário, `0`s são deslocados para a esquerda.
- O operador de deslocamento para a direita sem sinal (`>>>`) desloca os bits no seu operando esquerdo para a direita de acordo com o número de bits especificado no seu operando direito — `0`s são preenchidos a partir da esquerda.
- O operador complemento de bits (`~`) configura todos os bits `0` em seu operando como `1` no resultado e configura todos os bits `1` como `0` no resultado.
- Cada operador de bits (exceto o de complemento) tem um operador de atribuição correspondente.
- O construtor `BitSet` sem argumento cria um `BitSet` vazio. O construtor `BitSet` com um argumento cria um `BitSet` com o número de bits especificado por seu argumento.
- O método `set` de `BitSet` configura o bit especificado como 'ativado'. O método `clear` configura o bit especificado como 'desativado'. O método `get` retorna `true` se o bit estiver ativado, e `false` se o bit estiver desativado.
- O método `and` de `BitSet` realiza um E lógico bit a bit entre os `BitSets`. O resultado é armazenado no `BitSet` que invocou o método. De maneira semelhante, o OU lógico sobre bits e o XOR lógico sobre bits são realizados pelos métodos `or` e `xor`, respectivamente. O método `size` de `BitSet` retorna o tamanho de um `BitSet`. O método `toString` converte um `BitSet` em uma `String`.

Terminologia

`and`, método da classe `BitSet`
conjunto de bits
`BitSet`, classe
operadores de atribuição de bits
  `&=` (operador E)
  `^=` OU exclusivo sobre bits
  `|=` (OU inclusivo sobre bits)
  `<=` deslocamento para a esquerda
  `>=` (deslocamento para a direita com sinal)
  `>>=` (deslocamento para a direita sem sinal)
operadores de manipulação de bits
  `&` (operador E)
  `~` (complemento de bits)
  `^=` OU exclusivo sobre bits
  `|` (OU inclusivo sobre bits)
  `<=` (deslocamento para a esquerda)
  `<<` (deslocamento para a direita com sinal)
  `>>` (deslocamento para a direita sem sinal)
`clear`, método da classe `BitSet`
`equals`, método da classe `BitSet`
`get`, método da classe `BitSet`
máscara
`or`, método da classe `BitSet`
`set`, método da classe `BitSet`
`size`, método da classe `BitSet`
`toBinaryString` da classe `Integer`
`toString`, método da classe `BitSet`
`xor`, método da classe `BitSet`

Exercícios de revisão

I.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Os bits no resultado de uma expressão utilizando o operador _____ são configurados como `1` se pelo menos um dos bits correspondentes em um dos operando estiver configurado como `1`. Caso contrário, os bits são configurados como `0`.
b) Os bits no resultado de uma expressão utilizando o operador _____ são configurados como `1` se os bits correspondentes em cada operando estiverem configurados como `1`. Caso contrário, os bits são configurados como zero.
c) Os bits no resultado de uma expressão utilizando o operador _____ são configurados como `1` se exatamente um dos bits correspondentes em um dos operando estiver configurado como `1`. Caso contrário, os bits são configurados como `0`.
d) O operador _____ desloca os bits de um valor para a direita com extensão de sinal e o operador _____ desloca os bits de um valor para a direita com extensão zero.
e) O operador _____ é utilizado para deslocar os bits de um valor para a esquerda.
f) O operador E sobre bits (`&`) costuma ser utilizado para _____ os bits, isto é, para selecionar certos bits em uma string de bits e, ao mesmo tempo, configurar outros como `0`.

Respostas dos exercícios de revisão

I.1 a) `|`. b) `&`. c) `^`. d) `>>`, `>>>`. e) `<<`. f) mascarar.

Exercícios

I.2 Explique a operação de cada um dos seguintes métodos de classe `BitSet`:

a) `set`
b) `clear`
c) `get`
d) `and`
e) `or`
f) `xor`
g) `size`
h) `equals`
i) `toString`

I.3 *(Deslocamento para a direita)* Escreva um aplicativo que desloca para a direita uma variável inteira por quatro bits para a direita com o deslocamento para a direita com sinal e, então, desloca essa mesma variável inteira por quatro bits para a direita com o deslocamento para a direita sem sinal. O programa deve imprimir o inteiro em bits antes e depois de cada operação de deslocamento. Execute seu programa uma vez com um inteiro positivo e outra com um inteiro negativo.

I.4 Mostre como o deslocamento de um inteiro para a esquerda por um pode ser utilizado para realizar a multiplicação por dois e como o deslocamento de um inteiro para a direita por um pode ser utilizado para realizar a divisão por dois. Tenha cuidado em considerar as questões relacionadas ao sinal de um inteiro.

I.5 Escreva um programa que inverta a ordem dos bits em um valor inteiro. O programa deve inserir o valor do usuário e o método de chamada `reverseBits` para imprimir os bits em ordem inversa. Imprima o valor em bits antes e depois de os bits serem invertidos para confirmar que os bits foram corretamente invertidos. Você pode querer implementar uma solução recursiva e uma iterativa.