Bits and Bytes

Before getting into the details of cryptographic operators, we'll review some basics of working with bits and number bases. For those of you who have worked on operating systems, low-level protocols, and embedded systems, the material in this chapter will probably be a review; for the rest of you, whose day-to-day interactions with information technology and basic software development don't involve working directly with bits, this information will be an important foundation to the information contained in the rest of the book.

General Operations

Most cryptographic functions operate directly on machine representation of numbers. What follows are overviews of how numbers are presented in different bases, how computers internally represent numbers as a collection of bits, and how to directly manipulate bits. This material is presented in a computer- and language-generic way; the next section focuses specifically on the Java model.

Number Bases

In day-to-day operations, we represent numbers using base 10. Each digit is 0 through 9; thus, given a string of decimal digits $d_n d_{n-1} \cdots d_2 d_1 d_0$, the numeric value would be:

$$10^n d_n + 10^{n-1} d_{n-1} + \cdots + 10^2 d_2 + 10 d_1 + d_0$$

This can be generalized to any base x, where there are x different digits and a number is represented by:

$$x^n d_n + x^{n-1} d_{n-1} + \cdots + x^2 d_2 + x d_1 + d_0$$

In computer science, the most important base is base 2, or the binary representation of the number. Here each digit, or bit, can either be 0 or 1. The decimal number 30 can be represented in binary as 11110 or 16 + 4 + 2. Also common is hexadecimal, or base 16, where the digits are 0 to 9 and A, B, C, D, E, and F, representing 10 to 15, respectively. The number 30 can now be represented in hexadecimal as 1E or 16 + 14. The relationship between digits in binary, decimal, and hexadecimal is listed in Table 1.1.

Table 1.1 Binary, Decimal, and Hexadecimal Representations

BINARY	DECIMAL	HEXADECIMAL
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

When you are working with different bases, the base of a number may be ambiguous. For instance, is 99 the decimal or the hexadecimal 99 (= 9 × 16+9)? In this case, it's common to prefix a hexadecimal number with 0x or just x (e.g., 99 becomes 0x99 or x99). The same confusion can happen with binary numbers: Is 101 the decimal 101 or the binary 101 (4 + 1 = 5)? When a number in binary may be confused, it's customary to add a subscript 2 at the end of a string of binary digits, for example, $101_{2.}$

Any number can be represented in another base *b* if it is positive; however, doing the conversion isn't necessarily easy. We'll discuss general-purpose base conversion in a later section, but it's useful to note that conversion between two bases is especially easy if one of the bases is a power of the other. For instance, the decimal number 1234 has the canonical representation in base 10 as $1 \times 1000 + 2 \times 100 + 3 \times 10 + 4$. However, 1234 can also be thought as two "digits" in base 100: (12) and (34), with a value of $12 \times 1000 + 34 \times 100$. It's the same number with the same value; the digits have just been regrouped. This property is especially useful for base 2. Given a binary string, it's possible to convert to hexadecimal by grouping 4 bits and computing the hexadecimal value:

```
10011100 = 1001 1100 = 9C
```

Bits and Bytes

A *bit* is the smallest unit of information that a computer can work on and can take two values "1" and "0," although sometimes depending on context the values of "true" and "false" are used. On most modern computers, we do not work directly on bits but on a *collection* of bits, sometimes called a *word*, the smallest of which is a *byte*. Today, a byte by default means 8 bits, but technically it can range from 4 to 10 bits. The odd values are from either old or experimental CPU architectures that aren't really in use anymore. To be precise, many standards use *octet* to mean 8 bits, but we'll use the more common *byte*. Modern CPUs operate on much larger word sizes: The term *32-bit microprocessor* means the CPU operates primarily on 32-bit words in one clock cycle. It can, of course, operate on 8-bit words, but it doesn't mean it happens any faster. Many CPUs also have special instructions that sometimes can operate on larger words, such as the SSE and similar instructions for multimedia, as well as vector processing, such as on the PowerPC.

A byte has a natural numeric interpretation as an integer from 0 to 255 using base 2, as described earlier. Bit n represents the value $2n$, and the value of the byte becomes the sum of these bits. The bits are laid out exactly as expected for a numeric representation, with bit 0 on the right and bit 7 on the left, but the layout is backward when compared to Western languages.

$$(b_7b_6b_5b_4b_3b_2b_1b_0) = 2^7b_7 + 2^6b_6 + 2^5b_5 + 2^4b_4 + 2^3b_3 + 2^2b_2 + 2^1b_1 + 2^0b_0$$

or using decimal notation

$$(b_7b_6b_5b_4b_3b_2b_1b_0) = 128b_7 + 64b_6 + 32b_5 + 16b_4 + 8b_3 + 4b_2 + 2b_1 + b_0$$

For example, 00110111 = 32 + 16 + 4 + 2 + 1 = 55.

Bits on the left are referred to as the *most-significant bits,* since they contribute the most to the overall value of the number. Likewise, the rightmost bits are called the *least-significant bits*. This layout is also known as *Big-Endian,* which we'll discuss later.

Signed Bytes

Negative numbers can be represented in a few ways. The simplest is to reverse one bit to represent the sign of the number, either positive or negative. Bits 0 through 6 would represent the number, and bit 7 would represent the sign. Although this allows a range from –127 to 127, it has the quirk of two zeros: a "positive" zero and a "negative" zero. Having the two zeros is odd, but it can be worked around. The bigger problem is when an overflow occurs—for instance, adding 127 + 2 is 129 in unsigned arithmetic, or 1000001. However, in signed arithmetic, the value is –1.

The most common representation is known as *two's complement.* Given x, its negative is represented by flipping all the bits (turning 1s into 0s and vice versa) and adding 1, or computing $–1 – x$ (the same value). For example, note in Table 1.2 that adding 1 to 127 makes the value –128. While this method is a bit odd, there are many benefits. Microprocessors can encode just an addition circuit and a complementation circuit to do both addition and subtraction (in fact, many CPUs carry around the complement with the original value just in case subtraction comes up). The other main benefit is when casting occurs, or converting a byte into a larger word, as described in the following section.

Bitwise Operators

The usual arithmetic functions such as addition and multiplication interpret words as numbers and perform the appropriate operations. However, other operations work directly on the bits without regard to their representation as numbers. These are *bitwise* or *logical* operations. While examples shown in the next sections use 8-bit bytes, they naturally extend to any word size.

Table 1.2 Two's Complement Representation

UNSIGNED VALUE	SIGNED VALUE	HEXADECIMAL REPRESENTATION	BINARY REPRESENTATION
0	0	00	00000000
1	1	01	00000001
2	2	02	00000010
126	126	7d	01111110
127	127	7f	01111111
128	-128	80	10000000
129	-127	81	10000001
130	-126	82	10000010
253	-3	fd	11111101
254	-2	fe	11111110
255	-1	ff	11111111

As shown in Table 1.3, the operations are represented by different symbols depending if the context is programming or typographical. When required, this book will use the programming notations, with the exception of XOR, since the caret symbol (^) is used to denote exponents in some systems.

Table 1.3 Bitwise Operations and Notation

OPERATION	C-STYLE NOTATION	C-STYLE SELF-ASSIGNMENT	TYPOGRAPHICAL
NOT *a*	`~a`	n/*a*	$\neg a$
a AND *b*	`a & b`	`a &= b`	$a \wedge b$
a OR *b*	`a | b`	`a |= b`	$a \vee b$
a XOR *b*	`a ^ b`	`a ^= b`	$a \oplus b$
Shift *a* left by *n* bits	`a << n`	`a <<= n`	$a \ll n$

(continues)

Table 1.3 Bitwise Operations and Notation *(Continued)*

OPERATION	C-STYLE NOTATION	C-STYLE SELF-ASSIGNMENT	TYPOGRAPHICAL
Shift *a* right by *n* bits, preserve sign of *a*	`a >> n`	`a >>=n`	None; all shifts and words are assumed to be unsigned
Shift *a* right by *n* bits, unsigned	`a >>> n`	`a >>>=n`	$a >> n$
Rotate *a* right by *n* bits; w is number of bits of *a*	`(x >>> n) | (x << w-n)`	*n/a*	$\text{ROTR}^n(a)$
Rotate *a* left by *n* bits; w is the number of bits of *a*	`(x << n) | ( x >>> w-n)`	*n/a*	$\text{ROTL}^n(a)$
Concatenating *a* and *b*	`(a << shift) | b`	*n/a*	$a \parallel b$
Take *n* least-significant bits of *a*	`a & mask`	`a &= mask`	$\text{MSB}_n(a)$
Take *n* most-significant bits of *a*	`a & mask >>> shift`		$\text{LSB}_n(a)$

Complementation or Bitwise NOT

The simplest bit operation is complementation or the *bitwise NOT*. This simply flips bits within a word, where 0s become 1s and 1s become 0s—for example ~11001 = 00110. Cryptographically, this operation is not used much, primarily for implementing basic arithmetic in hardware.

Bitwise AND

AND is useful in bit testing and bit extraction. It's based on the usual truth table for logical AND, as shown in Table 1.4. You can remember the truth table by observing that it's the same as binary multiplication—anything times zero is zero.

Table 1.4 Bitwise AND

AND	0	1
0	0	0
1	0	1

To test if a bit is set, create a mask with 1s in positions you wish to test and perform the logical AND operation. If the result is nonzero, the bit is set; otherwise, the bit is not:

```
    01011101
AND 00001000
    ________
    00001000 // not == 0, bit 4 is set
```

It's also useful for bit clearing. A mask is created with 1s in positions to preserve and 0s where you want to clear. For instance, to clear the four least-significant bits in a byte, use the mask `11110000`:

```
    11011101
AND 11110000
    ________
    11010000
```

Bitwise OR

Bitwise OR extends the logical OR to a series of bits. In English, *or* means one or the other but not both. The bitwise version means one or the other or both (the same as the logical OR). See Table 1.5.

Logical OR is useful in joining nonoverlapping words. The following example joins two 4-bit words together:

```
   11010000
OR 00001111
___________
   11011111
```

In this case, the same effect could be done by using regular addition; however, it's pretty standard to use OR instead to make it clear that addition is not the point of the operation and that you are doing bit operations.

Table 1.5 Bitwise OR

OR	0	1
0	0	1
1	1	1

Table 1.6 Bitwise XOR

XOR	0	1
0	0	1
1	1	0

Bitwise Exclusive OR (XOR)

Exclusive OR is abbreviated as XOR, and its operation is denoted by ⊕. It is less common in normal programming operations but very common in cryptographic applications. Table 1.6 shows the Bitwise XOR table. This table is easy to remember, since it is the same as addition but without any carry (this is also known as addition modulo 2). It's also equivalent to the English usage of *or*—one or the other but not both.

XOR tests to see if two bits are different: If the bits are both 0s or both 1s, the result is 0; if they are different, the result is 1. XOR is useful in cryptographic applications, since unlike AND and OR, it's an invertible operation, so that A ^ B ^ B = A:

```
    11010011
XOR 10101010
————————————
    01111101
XOR 10101010
————————————
    11010011
```

Left-Shift

As its name implies, the left-shift operator shifts the bits within a word to the left by a certain number of positions. Left-shifts are denoted by $a << b$, where a is the value and b denotes the number of positions to shift left. Zeros are filled in on the least-significant positions:

```
11111111 << 1 = 11111110
11001100 << 3 = 01100000
```

Mathematically, each shift left can be thought of as multiplication by 2:

```
 3 × 2   = 00000011 << 1 = 00000110 = 4 + 2 =  6
 3 × 4   = 00000011 << 2 = 00001100 = 8 + 4 = 12
-3 × 2   = 11111101 << 1 = 11111010 = -6
-128 × 2 = 10000000 << 1 = 00000000 = 0 (overflow)
```

When you are working with large word sizes (such as 32-bit integers), left-shifts are used to compute large powers of 2, since $2^n = 1 << n$. A trick to test to see if only one bit is set (or rather the value is a power of 2) is using `x & -x == x`. In many cryptographic operations, we need to extract a certain number of the least-significant bits from a word. Normally you are working with word sizes much bigger than a byte. However, say you needed to extract the six least-significant bits from a byte *B*. You could perform *B* & 0x3f (*B* & 0011111). Computing this for larger word sizes is a bit clumsy, and worse, some algorithms may extract a variable number of bits, so the mask has to be dynamic. Hardwiring a hexadecimal mask is also prone to errors, and someone reading your code would have to think about how many bits you are extracting. How many bits are in 0x7fffff? The answer is 23, but it's not automatically clear, and staring at a computer monitor all day where the text is in a small font makes the task harder. Instead, a left-shift can be used, since $2^n - 1$ or $(1 << n) - 1$ has binary representation of with $(n - 1)$ digits of "1." Instead of 0x7fffff, we can use $((1 << 24) - 1)$, which will be perfectly clear to someone familiar with bit operations. Even better, you can make your intentions clearer by making the mask a constant:

```
public static final int mask23lsb = (1<<24) -1; // 23-bit mask
```

Right-Shift

Not surprisingly, right-shifts, denoted by >>, are the opposite of left-shifts, and positions are shifted to the right. However, there is another significant difference. In left-shift notation, zeros are placed in the least-significant position. With right-shift, the value of the most-significant bit is used to fill in the shifted positions. For example, 1000000 >> 2 = 1100000, but 0100000 >> 2 = 00100000. This is designed so that the sign is preserved and the right-shift is equivalent to division by 2 (dropping any fractional part), regardless of the sign. For example, we'll "undo" the previous left-shift examples:

```
   6 / 2  = 00000110 >> 1 = 00000011 =   3
  12 / 4  = 00001100 >> 2 = 00000011 =   3
  -6 / 2  = 11111010 >> 2 = 11111101 =  -3
-127 / 2  = 10000001 >> 1 = 11000000 = -64
```

This works well when you are treating the byte as a number. In cryptographic applications, the bits are just treated as bits without any numerical interpretation. In every case, it is assumed the byte or word is unsigned (e.g., unsigned long in C). If you are using a signed type (such as Java),

you'll want to use the *unsigned* right-shift, denoted by >>> (three less-than symbols). This just shifts the bits to the right and fills in zeros for the most significant bits. All cryptographic papers assume the shift is unsigned and normally use the plain >>. When coding an algorithm using signed types, make sure you convert >> to >>>.

Special Operations and Abbreviations

The previous operations, while useful, are fairly basic and typically directly implemented in hardware. By combining and composing them, you can create new higher-level bit operations. For cryptography, the most useful operations are rotations, concatenation, and extraction of most- and least-significant bits.

Rotations

Bit rotations are fairly self-explanatory: Bits are shifted either left or right, and the bits that "fall off the edge" roll over onto the other side. These are commonly used in cryptography, since this method provides an invertible operation that can aid in scrambling bits. The problem is that they aren't used for much else, so many CPUs do not have rotation instructions, or if they do, they are frequently slow. Even if they have a fast rotation, the only way to access them is by writing assembly language, since rotations do not have a standard operator like the shifts do. Common programming rotations can be accomplished with a combination of shifts and logical operators, which is the slowest method of all. In its most general case, a rotation will take one subtraction, two shifts, and a logical OR operation. Remember to use the unsigned right-shift operator:

```
(x >>> n) | (x << 32-n);   // rotate right n positions, x 32-bit int
(x << n) | (x >>> 32-n);   // rotate left n position, x 32-bit int
```

There is no special typographical symbol for rotations; we normally use the abbreviations ROTR or ROTL, although certain technical papers may define their own symbols in their work.

Bit Concatenation

Bit concatenation is the process of simply joining together two sets of bits into one word. If *b* is *n*-bits long, then a `||` b is `a << n | a:`

```
a = 101
b = 0011
a || b = 11 << 4 | 0011 = 1010000 | 0011 = 1010011
```

MSB and LSB operations

Two other common operations are extracting a certain number of most-significant bits (MSB) or least-significant bits (LSB). We'll use the notation $MSB_n(a)$ to denote extracting the n most-significant bits. Likewise with $LSB_n(a)$. These operations are accomplished by making the appropriate mask, and in the MSB case, by shifting appropriately:

```
MSB3(10111111) = (10111111 & 11100000) >>> 5 = 101
LSB2(11111101) = 11111111 & 00000011 = 00000001
```

Packed Words

Shifts are useful in creating *packed words*, where you treat the word as an array of bits rather than as a number, and where the bits represent anything you like. When doing this, you should use an unsigned numeric type. Unfortunately, Java and many scripting languages do not have unsigned types, so you must be extremely careful. As an example, suppose you have two variables that are numbers between 0 and 15. When transmitting or storing them, you might want to use the most space-efficient representation. To do this, you represent them not as 8-bit numbers from 0 to 255, but as two numbers each with 4 bits. For this example, we'll use pseudocode and use the type `int` to represent an unsigned 8-bit value:

```
int a = 5; // 00000101
int b = 13; // 00001101
int packed = a << 4 | b; // = 00000101 << 4 | 00001101
                         // = 01010000 | 00001101
                         // = 01011101
```

To undo the operation:

```
b = packed & 0x0F; // 01011101 & 00001111 = 00001101
a = packed >>> 4;  // note unsigned right-shift
                   // 01011101 >>> 4 = 00000101
```

Likewise, you might wish to view the byte as holding eight true/false values:

```
int c = b0 | (b1 << 1) | (b2 << 2) | (b3 << 3) | (b4 << 4)
           | (b5 << 5) | (b6 << 6) | (b7 << 7);
int mask = 0x01;
b0 =  c & mask
b1 = (c >>> 1) & mask;
b2 = (c >>> 2) & mask;
```

and so on. If you were using an array, you could do this dynamically in a loop as well.

```
for (int i = 0; i < 8; ++i)
    b[i] = (c >>> i) & mask;
```

It's quite easy to make mistakes, especially when the packed word has a complicated structure. If you are writing code and the answers aren't what you'd expect:

- Check to make sure you are using the unsigned right-shift operator >>> instead of the signed version >>.
- Check that the language isn't doing some type of automatic type conversion of signed values (e.g., turning bytes into integers before the shift operation happens).
- Check to make sure your masks are correct.
- If you are using dynamic shifts, make sure the shift amount isn't larger than the word size. For instance, if you are shifting a byte, make sure you aren't doing >> 9. How this is interpreted depends on the environment.

Integers and Endian Notation

Endian refers to how a string of bytes should be interpreted as an integer, and this notation comes is two flavors: *Little-Endian* and *Big-Endian*. The names come from Jonathan Swift's *Gulliver's Travels*, where the tiny people, Lilliputians, were divided into two camps based on how they ate eggs. The Little-Endians opened the eggs from the small end, and the Big-Endians opened theirs from the big end. As you might expect with that reference, there are pros and cons to Big-Endian and Little-Endian representations, but overall it doesn't make any difference except when you have to convert between the two formats. A comparison of the two is shown in Table 1.7.

Table 1.7 Comparison of Little-Endian versus Big-Endian Representation

ENDIAN TYPE	$B_0B_1B_2B_3 = 0XAABBCCDD$	SAMPLE MICROPROCESSORS
Little-Endian	aa bb cc dd	Intel x86, Digital (VAX, Alpha)
Big-Endian	dd cc bb aa	Sun, HP, IBM RS6000, SGI, "Java"

Big-Endian, also know as most-significant byte order (MSB) or network order, puts the most-significant or highest byte value first. This is equivalent to how we write decimal numbers: left to right. The downside is that many numerical algorithms have to work backward starting at the end of the array and working forward (just as you would manually with pencil and paper).

The Little-Endian, or least significant byte (LSB), format is the opposite. This makes it harder for humans to read hex dumps, but numeric algorithms are a little easier to implement. Adding capacity (or widening conversions) is also easier, since you just add bytes to the end of the array of bytes (i.e., 0xff becomes 0xff00).

Fortunately, regardless of what the byte Endian order is, the *bits* within bytes are always in Big-Endian format. For instance, 1 is always stored in a byte as 00000001_2 no matter what platform.

The Endian issue becomes critical when you are working with heterogeneous systems—that is, systems that use different Endian models. When shipping bytes between these machines, you must use a standard Endian format or an ASCII format. In many other programming languages, you must determine in advance what the Endian architecture is and adjust subsequent bit operations appropriately. For cryptographic applications this becomes critical, since you are often manipulating bits directly.

With Java, the underlying architecture is hidden and you program using a Big-Endian format. The Java virtual machine does any Endian conversion needed behind the scenes.

For C and C++ programmers, normally a `BIG_ENDIAN` or `LITTLE_ENDIAN` macro is defined by the compiler or from an include file. If not, you can use code similar to this for testing. It sets raw memory and then converts it to an integer type. The value will be different depending on the CPU's Endian type. This C code assumes an `int` is a standard 4 bytes or 32 bits, but you may wish to generalize:

```
int isBigEndian() {
  static unsigned char test[sizeof(unsigned int)] = {0, 0, 0, 1};
  unsigned int i = *(unsigned int) test;
  if (i == 0x0000001) return 1; // true, big-endian
  return 0;                     // false, little-endian
}
```

Java Numerics

We'll now apply the information in the previous section to the Java numeric model. While notation is similar to the C or C++ model, there are

Java-specific issues with using signed and unsigned types—specifically, byte arrays. Java also provides class wrappers around the native types, as well as an unlimited-capacity integer arithmetic.

Basic Types

Java provides the standard basic numeric types—integers with 8, 16, 32, and 64 bits, and floating-point types with 32- and 64-bit representations. Unlike C and other languages, all the types are signed—there are no native unsigned types. Table 1.8 shows Java's primitive numeric types.

For integer types, a literal integer can be expressed in decimal or hexadecimal formats (using lower- or uppercase letters for hex digits):

```
int i1 = 1000;
int i2 = 0x3e8; // == 1000 decimal
int i3 = 0x3E8; // same
```

For literal long values, a prefix of L or l should always be used, even if it appears unnecessary:

```
long l1 = 1000;            // compiles but is not recommended
long l1 = 1000L;           // recommended
long l2 = 0xfffffffff;     // won't compile, error
long l2 = 0xffffffffL;     // correct
```

Table 1.8 Primitive Numeric Types in Java

NAME	TYPE	LOGICAL SIZE	RANGE
byte	signed integer	8 bits	–128 to 127
short	signed integer	16 bits	–32768 to 32767
int	signed integer	32 bits	–2,147,483,648 to 2,147,483,647 (2.1 billion)
long	signed integer	64 bits	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 ($\pm 9.2 \times 10^{18}$)
float	ANSI/IEEE 754 floating point	32 bits	$\pm 1.4 \times 10^{-45}$ to $\pm 3.4028235 \times 10^{38}$ (6–7 significant decimal digits)
double	ANSI/IEEE 754 floating point	64 bits	$\pm 4.9 \times 10^{-324}$ to $\pm 1.7976931348623157 \times 10^{308}$ (15 significant decimal digits)

Specifying literal values for bytes can be tricky because a byte is signed from –128 to 127, while very often you'll be using constants specified from 0 to 255. If the value is within –128 to 127, the code will compile. If the value is from 128 to 255, you can either convert it to its negative equivalent or use a cast operation. The same principles apply to the `short` type:

```
byte b1 = 10;
byte b2 = 189;                  // error, out of bounds
byte b3 = -67;                  // = 189, ok
byte b4 = (byte) 189;           // ok
byte b4 = (byte) 0xbd;          // ok
```

Floating-type literals are assumed to be a `double` type unless suffixed by an `f` for `float`. Floating types can also be representing using scientific notation using `valEscale` = val × 10^{scale}.

```
float  f1 = 0.1;    // compiler error
float  f2 = 0.1f;   // by default "0.1" is a double type
double d1 = 0.1;
double d2 = 1.0E2   // == 100
```

In practice, the `short` type is rarely used because it almost always is converted into an `int` type before anything else is done. The `float` type also isn't used because it doesn't provide enough precision for most applications.

Type Conversion

Conversions between types can either be widening or narrowing. A *widening conversion* is where the new type can represent larger numbers and can "contain" the old type. For an integer-to-integer type conversion (i.e., `short` to `long`), this just means putting the same number in a larger container. Integer-to-floating-point conversions are also considered as widening, but some of the least-significant digits may be scrambled or zeroed due to how floating point numbers are presented. These types of conversion happen automatically and silently either at compile time or at run time. For example:

```
int    i1 = 123;
long   l1 = i;          // ok -- l = 123
float  f1 = i;          // ok -- f = 123

int    i2 = 123456789;
float  f2 = i;          // ok, but f = 123456792
```

Narrowing conversions may result in a loss of magnitude and precision. Any conversion from floating-point to an integer type is considered narrowing and is clearly a larger integer type to a smaller integer type. Java will *not* automatically do narrowing conversions, and a compiler error will be issued if the compiler detects such a conversion. To do these conversions, you must explicitly declare them with the cast operator.

Converting floating point to integers drops or truncates any digits to the right of the decimal point:

```
float  f1 = 0.123456;
int    i0 = f1;          // compiler error -- requires a cast.
int    i1 = (int) f1;    // == 0

float  f2 = 0.99999;
int    i2 = (int) f2;    // == 0

float  f3 = 100.9;
int    i3 = (int) f3;    // == 100

float  f4 = -100.9;
int    i4 = (int) f4;    // == -100

int    i5 = 0xffL;       // compiler error;  long to int conversion
```

For narrowing conversions from one integer type to another, the least-significant bits (or bytes) form the larger type:

```
int    i1 = 0xfffff01;
byte   b1 = (byte) b;     // b = 0x01;
```

These rules are summarized in Table 1.9, with *N* representing a narrowing conversion, *W* for a widening conversion, and *W** for a widening conversion that may result in a loss of precision.

Table 1.9 Primitive Type Conversion

	BYTE	SHORT	INT	LONG	FLOAT	DOUBLE
byte		W	W	W	W	W
short	N		W	W	W	W
int	N	N		W	W*	W
long	N	N	N		W*	W*
float	N	N	N	N		W
double	N	N	N	N	N	

Unsigned to Signed Conversions

There are no unsigned types in Java; however, you can still use the signed types as an unsigned *container*. Even though a `byte` is signed, it still has 8 bits and can represent an integer from 0 to 255 even if Java thinks otherwise. To use the unsigned value, it must be converted into a larger type using an AND mask.

To convert a unsigned byte:

```
byte  c = (byte) 254;         // b = -2 in Java.
short c = (short)(x & 0xff);  // c = 254
int   c = x & 0xff            // c = 254
long  c = x & 0xffL;          // c = 254, must put L at end of 0xffL
```

Unsigned short values are converted by the following:

```
int  c = x & 0xffff;
long c = x & 0xffffL;
```

Finally, the unsigned `int` is converted into a signed `long` by the following:

```
long c = x & 0xffffffffL
```

It's critical to put an `L` at the end of the mask when working with `long` types. Without it, Java will respond as if you are doing an `int` computation, then doing a widening conversion to a `long`:

```
byte b = (byte)0xff;

long b = (b & 0xff) << 56;              //  Wrong.  b = 0
long b = (long)((int)(b & 0xff) << 56); //  same as previous

long b = (b & 0xffL) << 56; // Correct. b = 0xff00000000000000
```

Overflow

When a computation exceeds an integer type's bounds, the value is "rolled over" *silently;* no exception is thrown:

```
byte b = (byte) 127;  // b = 0111111
b++;                  // b = 1000000, but b has the value of -128
```

While the silence may seem disturbing, in practice overflow is rarely a problem. For floating-point computations overflow is still silent, but the result is not rolled over. Instead, the type takes on a special `Infinity`

value that can be checked using normal comparison operators or by using the `Double.isInfinite` method. Also note that Java does not think 0.0 is the same as 0 (zero). Instead, 0.0 is treated as a number that is very close to zero, so division by 0.0 results in an infinite value. Keep in mind, however, that Java throws an exception if you try to divide by the *integer* 0. In the event that a computation doesn't make sense, the value is `NaN`, which stands for "not a number." If you wish to see if a value is `NaN`, then you *must* use the `Double.isNaN` method. Various examples of manipulating double types are shown as follows:

```
double d1 = 1.7E308 * 2.0;          // overflow = Infinity
double d2 = 1.0/0.0;                // =  Infinity
double d3 = -1.0/0.0;               // = -Infinity
int i = 1 / 0;                      // throws a DivisionByZero exception
double d4 = 0.0/0.0                 // = NaN
boolean b = (d4 == d4);             // = false,  always
boolean b = Double.isNan(d4);       // true;
boolean b = Double.isInfinite(d3); // true
```

Arrays

In Java, native-type arrays are treated and created as if there were objects created using a constructor with the `new` operator. Arrays can also be set with initial values, as shown in the following code snippet.

```
int[] a= new int[3]; // all three elements set to zero
int[] a = {1, 2, 3}; // a pre-set 3 element array.
```

Once constructed, an array is *not* resizable. If dynamically allocated storage is needed, you should use one of the collections in the `java.util` package. The index to arrays is with `int` types, and the first element starts at 0 (as in C). Small types will have widening conversions done to them, and the `long` type cannot be used without a cast operation. Thus, single dimensional arrays cannot be larger than 2^{31}, or roughly 2.1 billion, entries. Hopefully, this will be sufficient for your needs.

Since arrays are objects, assignment is done by *reference*, as shown in the following:

```
int[] a = {1,2};
int[] b = a;
b[0] = 100; // modifies the object that both a and b point too.
System.out.println(a[0]); // will print 100, not 1
a = null; // b is null, but a is intact.
```

In addition, they can be copied using the `clone` method. To use this, cast the result to the correct array type:

```
int[] a = {1, 2};
int[] b = (int[]) a.clone(); // b is a "deep copy" of a
```

The `System.arraycopy` method is extremely useful for copying parts of an array or concatenating arrays. It makes a native system call to copy memory directly instead of copying each element individually and is much faster than writing a custom `for` loop:

```
System.arrayCopy(Object input, int inputOffset,
                 Object output, int outputOffset, int length)
```

Even though the method has `Object` in its signature, this method only works on array types. Other useful methods for manipulating native arrays can be found in the class `java.util.Arrays` and are summarized in Table 1.10.

Table 1.10 Summary of Methods from `java.util.Arrays`

JAVA.UTIL.ARRAYS METHOD	DESCRIPTION
`static boolean equals(type[] a, type[] b)`	Returns true if and only if the arrays are the same length and every element is equal.
`static void fill(type[] a, type val)`	Fills an array with a single value.
`static void sort(type[] a)`	Performs a fast numeric sort. Array is modified.
`static void sort(type[] a, int fromIndex, int toIndex)`	Sorts only part of an array.
`static int binarySearch(type[] a, type val)`	Performs a binary search of a sorted array. Returns the array index if a match is found and –1 if no match. Arrays must be sorted first.

Numeric Classes

Each numeric type has a matching class that acts as an object wrapper, as shown in Table 1.11.

These classes do not have any support for mathematical operations—you can't add two integer objects directly. Instead, these classes are primarily used to allow numeric types to be used in methods that expect an `Object` type, such as in collections from the `java.util` package (e.g., `ArrayList`, `HashMap`). The wrapper classes also provide basic string formatting and parsing from strings for numbers. Since they are objects, they can also be `null`. Likewise, objects are always pass-by-reference.

```
public void changeInt(Integer i) {
     i = new Integer("1");
}
Integer I = new Integer("0");
changeInt(i);    // i is now 1
```

All of the classes share some common traits (examples are just shown for `Integer` and `Long`):

- Two public fields, `MAX_VALUE` and `MIN_VALUE`, in case you don't want to memorize the previous table.
- `byteValue`, `doubleValue`, `floatValue`, `intValue`, and `longValue` methods that return the underlying number in a native type.
- A static method valueOf that accepts a string and an optional radix to parse a string and return an object. The radix can be 2 to 32.

```
static Integer Integer.valueOf(int val)
static Integer Integer.valueOf(int val, int radix)
static Long Long.valueOf(long val)
static Long Long.valueOf(long val, int radix)
```

- A static method `parseClass` (where `Class` is the name of the class, such as `Byte` or `Integer`) that also accepts a string, and an optional radix to parse a string returns the *native type* instead of an object.

```
static int Integer.parseLong(int val)
static int Integer.parseFloat(int val, int radix)
static long Long.parseLong(long val)
static long Long.parseLong(long val, int radix)
```

- `toString` that returns the number as unformatted decimal number.

Table 1.11 Java Class Wrappers for Native Types

NATIVE TYPE	MATCHING JAVA.LANG CLASS
`int`	`Integer`
`byte`	`Byte`
`double`	`Double`
`float`	`Float`
`long`	`Long`
`short`	`Short`

The `Long` and `Integer` classes have a few other useful static methods that provide an *unsigned* representation of the number in binary, hex, or octal formats as shown:

```
static String Integer.toBinaryString(int val)
static String Integer.toHexString(int val)
static String Integer.toOctalString(int val)
static String Long.toBinaryString(long val)
static String Long.toHexString(long val)
static String Long.toOctalString(long val)
```

Binary representation is especially useful for debugging bit fields. These objects do not add any "leading zeros" to the output, so `new Integer(16).toHexString()` just returns `F` and not `0F` or `0x0F`.

Booleans and BitFields

Java provides a native `boolean` type that can either be `true` or `false`. Unlike C and C++, it is not a numeric type and does not have a numeric value, and it is not automatically cast. Like the numeric types, there is also a boolean class wrapper. If you want to create an array of boolean values, you could use `Boolean` with one of the collection classes (e.g., `ArrayList`). However, there is a special class `java.utl.BitField` specially designed for use with boolean types that provides a huge performance increase and memory savings. More specialized applications will convert a native type or byte array into a bit field directly.

Chars

Java has another native type `char` that represents a Unicode character and is used by `String` and `StringBuffer` internally. It is represented by an *unsigned* 16-bit integer, but it is not a numeric type. It is automatically cast to an integer in situations when needed, such in mathematical or bitwise operations, but it's not automatically cast to any other type. However, you can explicitly convert a char to another type by using a cast operator.

Since it's unsigned, it might be tempting to use `char` in places for mathematical purposes and for raw bit fields. In practice, there is not much point, since `char` types are automatically converted into an `int` before any operation is done, eliminating any advantage

Working with Bytes

Because the Java `byte` type is signed and because of the automatic conversion rules, working with bytes and byte array can be frustrating. Following are some tips and tricks to simplify byte manipulation.

Sign Problems

It's always easiest to specify a constant byte value by using a cast (later in this chapter you will see tricks on making byte arrays):

```
byte b = 0xff;        // Compiler error.  0xff is an 'int' type
byte b = (byte) 0xff; // right
```

Bytes range from –127 to 128, and not 0 to 255. Setting a byte with the value 128–255 and a cast is fine—the byte contains the correct bits. The problem is when you want to retrieve the unsigned value or when you perform a computation with bytes and ints. For instance:

```
byte b = 127;
int  i = 3;
System.out.println((i+b));  // Prints -126 and not 130
```

results in –127 being printed, not 129. In the addition step, byte *b* is automatically converted to an `int` type, including the sign. While b was set with 128, internally it's really –1, and this is its value when converted.

The key point to remember is that all bitwise and arithmetic operators work on `int` and `long` types only. All other types are automatically cast. If you are using a byte as an unsigned value, you must manually convert the type to an `int` or `long` *before* the operation using `val & 0xFF` or `val & 0xFFL` respectively.

The problem is more mysterious with the shift operators. Let's say you want to do a right-shift on 0xFF. In binary, this is 11111111, so you'd expect an unsigned right-shift to be 0111111, or 0x7F. The natural way to write this is:

```
byte b1 = (byte) 0xff;
byte b2 = b >>> 1;      // compiler error
```

However, this results in a strange error from the javac compiler, and the problem isn't clearly defined:

```
aprogram:java.6: possible loss of precision
found: int
required: byte
      byte b2 = b >>> 1;
                  ^
```

You might try and fix this by casting the result:

```
byte b1 = (byte) 0xff;
byte b2 = (byte)(b >>> 1);  // error #2
```

This performs the compile operation; however, the result is still wrong. The value of `b2` is *not* 0x7f, but strangely remains at 0xff or –1. Again, Java is converting the byte into an integer *before* the shift. Since bytes are signed, it converts the byte value of –1 into an `int` value of –1, which is 0xffffffff. *Then* it performs the shift, resulting in 0x7fffffff. Finally, the cast takes the least bits, or 0xff, and converts them to a `byte`. The step-by-step details are listed in Table 1.12. The correct solution is to treat the `byte` as an unsigned value, then convert to an `int`, and then do the shift and cast back down to a `byte`:

```
byte b1 = (byte) 0xff;
byte b2 = (byte)((b & 0xff) >>> 1); // correct:
```

Table 1.12 Comparison of Bit-Shifting a Byte Type

STEPS	INCORRECT	CORRECT
Initial value	`0xff` = –1	`0xff` = –1
Convert to `int`	`0xffffffff` = –1	`0x000000ff` = 255
Right-shift `>>>`	`0x7fffffff` = large int	`0x0000007f` = 127
Cast down to `byte`	`0xff` = –1	`0x7f` = 127

To convert a `byte` back to its unsigned value, you have to do a little trick; you must bitwise AND the `byte` with 0xff, that is, `b & 0xff`. For example:

```
byte b = 128;
int i = 1;
int unsignedByteValue = b & 0xff
System.out.println((i + unsignedByteValue);  // 129
System.out.println((i + (b & 0xff));         // combined
```

This trick works by observing that the least-significant bytes in an `int` match exactly with the `byte` type, as shown in Table 1.13.

Conversion of Integral Types to Byte Arrays

Java does not have any easy way of converting integral types into raw bytes. There are the serialization methods, but these involve a lot of work for something that should be relatively simple. For example:

```
public static byte[] intToByteArray(int i) {
     byte[] buf = new byte[4];
     b[0] = (byte) (i >>> 24);
     b[1] = (byte) (i >>> 16);
     b[2] = (byte) (i >>>  8);
     b[3] = (byte) i;
}

public static int byteArrayToInt(byte[]) {
     If (buf.length != 4)
          throw new RuntimeException("Bad Length");
     return ((b[0] & 0xff) << 24) | ((b[1] & 0xff) << 16) |
  (b[2] & 0xff) <<  8) | (b[3]);
}
```

Table 1.13 Comparison of Representation between `int` and `byte` Types

UNSIGNED VALUE	SIGNED VALUE	BYTE REPRESENTATION	INT REPRESENTATION
0	0	00	00000000
1	1	01	00000001
2	2	02	00000002
126	126	7d	0000007e
127	127	7f	0000007f
128	-128	80	00000080

Table 1.13 *(Continued)*

UNSIGNED VALUE	SIGNED VALUE	BYTE REPRESENTATION	INT REPRESENTATION
129	-127	81	00000081
130	-126	82	00000082
253	-3	fd	fffffffd
254	-2	fe	fffffffe
255	-1	ff	ffffffff
256	256	00	00000100

We could do the same thing for longs. Instead, we'll demonstrate a different technique that builds the result by incrementally shifting the value in question:

```
public static byte[] longToByteArray(long I) {
     byte[] buf = new byte[8];
     for (int j = 7; j >= 0; ++j) {
          buf[j] = (l & 0xffL); // !! must use 0xffL, not 0xff !!
          l >>>= 8;           // l = l >>> 8;
     }
}

public static long byteArrayToLong(byte[] buf)
{
     long i = 0;
if (buf.length != 8)
          throw new RuntimeException("Bad Length");
     for (int j = 0; j < 8; ++j) {
          i |= buf[j];
          i <<= 8;      // l = l << 8;
     }
     return i;
}
```

Converting to Hex Strings

Converting an array into a hexadecimal string is a fairly common task for printing, data interchange, and debugging. A naive way to do hex conversions is by using one of the previously mentioned byte `array-to-long` methods followed by `Long.toHexString`. However this is very slow

and is limited to arrays smaller than 8 bytes. Even worse would be using `BigInteger` (discussed later in the chapter). For example:

```
byte[] mybytes = ...;
BigInteger a = new BigInteger(1, mybytes);
return a.toString(16);
```

While mathematically correct, most programs operate on printing two hex digits per byte; for example, decimal 16 is converted to 0F, not F. The other problem is that this method is horrendously slow.

While these methods are useful in a pinch, for high performance applications they won't do. If one is willing to use a little memory, performance can be doubled or tripled. The conversion itself is fairly simple, but an implementation that uses tables has some important benefits:

- Provides typically higher performance (at the expense of some minor setup and memory costs).
- Allows using different alphabets instead of the standard. These issues are further detailed in Chapter 4.
- Typically allows simpler coding.

The conversion class starts out defining the hexadecimal alphabet (in our case, the standard one). The `hexDecode` table is the inverse mapping; for instance, character A is mapped to 10. Characters that are invalid are set to –1. For example:

```
public class Hexify
{
    protected static char[] hexDigits = {'0', '1', '2', '3', '4',
           '5' '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'},
     protected static int[]  hexDecode = new int[256];
    static {
     for (int i = 0; i < 256; ++i) hexDecode[i] = -1;
     for (int i = '0'; i <= '9'; ++i) hexDecode[i] = i - '0';
     for (int i = 'A'; i <= 'F'; ++i) hexDecode[i] = i - 'A' + 10;
     for (int i = 'a'; i <= 'f'; ++i) hexDecode[i] = i - 'a' + 10;
    }
```

Encoding is fairly straightforward. Each byte is split into two sections, and the table is used to determine the appropriate character:

```
    public static String encode(byte[] b) {
     char[] buf = new char[b.length * 2];
     int max = b.length;
     int j = 0;
```

```
 for (int i = 0; i < max; ++i) {
     buf[j++] = hexDigits[(b[i] & 0xf0) >> 4];
     buf[j++] = hexDigits[b[i] & 0x0f];
 }
 return new String[buf];
}
```

Decoding is more interesting. First, a function, given a hexadecimal character, returns the integer value. This is done via a table lookup, and the result is verified to make sure it's valid. It's always important to check the input for validity, especially if you don't have control of what the input is. You could easily add or inline these checks into the main decode routine for a minimal improvement in execution time, as follows:

```
protected static int getHexDecode(char c) {
 int x = hexDecode[c];
 if (x < 0) throw new RuntimeException("Bad hex digit " + c);
 return x;
}
```

While the encode function always encodes two characters per byte, you may receive a hexadecimal string that doesn't conform to this rule. If you are only processing strings that you created, you could check that the number of input characters is even and that an exception is thrown if it's not. The following implementation handles full generality. The string is checked to see if it has an odd number of characters, by computing `max & 0x01` (or, equivalently, `max % 2`), which just looks at the least-significant bit. If it's set, then the value is odd.

```
public static byte[] decode(String s) {
 char[] input = s.charArray[];
 int max = input.length;
 int maxodd = max & 0x01;
 byte b;
 byte[] buf = new byte[max/2 + odd];
 int i = 0, j = 0;
 if (maxodd == 1) {
     buf[j++] = getHexDecode[input[i++]]
 }
 while (i < max) {
     buf[j++] == (byte)(getHexDecode[input[i++]] << 4 |
      getHexdecode[input[i++]]);
 }
 return buf;
}           //end of class Hexity
```

BigInteger

The `BigInteger` class in the `java.math` package provides a way of doing computations with arbitrary large integer types. Unlike C++, Java does not overload operators, so to do basic math, you have to call methods to perform mathematical operations such as addition. All of the operations create new `BigInteger` objects as results. The objects used in the operations remain untouched. For instance:

```
BigInteger b1 = new BigInteger("1");
BigInteger b2 = new BigInteger("2");
BigInteger b3 = b1.add(b2);
```

At the end, `b1` is still 1, `b2` is still 2, and `b3` is what you'd expect it to be, 3. Modifying an object requires explicit code. For instance, if we wanted `b1` to contain the result of the addition, we would use self-assignment like this:

```
b1 = b1.add(b2); // b1 is now 3
```

While the notation is different, the behavior is no different than using normal operators. Adding 1 + 2 doesn't change the original values of 1 or 2.

All of the usual arithmetic operations are available: `add`, `subtract`, `multiply`, `divide`, and `remainder`. Another method, `divideAndRemainder`, returns a two-element `BigInteger` array where element 0 is the division result and element 1 is the remainder. A complete list and comparison of BigInteger operations is given in Table 1.14.

Table 1.14 BigInteger Arithmetic Operators

ARITHMETIC OPERATION	NATIVE TYPE NOTATION	BIGINTEGER NOTATION
Addition	`a + b`	`a.add(b)`
Subtraction	`a - b`	`a.subtract(b)`
Multiplication	`a * b`	`a.mult(b)`
Integer division	`a / b`	`a.divide(b)`
Remainder	`a % b`	`a.remainder(b)`
Division with remainder	`int[] result = { a / b, a % b }`	`BigInteger[] result = a.divideAndRemainder(b)`
Negation	`-a`	`a.negate()`

Table 1.14 *(Continued)*

ARITHMETIC OPERATION	NATIVE TYPE NOTATION	BIGINTEGER NOTATION
Exponentiation	`Math.pow(a,b)`	`a.pow(b)`

Random value	`Random r = new Random();`
	`a = r.getInt();`
		`Random r = ...`
		`int bits = ...;`
		`BigInteger a =`
		`new BigInteger(bits, r);`

Absolute value	`(a >= 0) ? a : b` ***or*** `Math.abs(a)`	`a.abs()`
Minimum of a, b	`(a < b) ? a : b` ***or*** `Math.min(a,b)`	`a.min(b)`
Maximum of a, b	`(a > b) ? a : b` ***or*** `Math.max(a,b)`	`a.max(b)`

Likewise, all the usual bit operations are available, as described in Table 1.15. There is no need for a special unsigned right-shift operator, since `BigInteger` has unlimited capacity; the result of shifting *n* places to the right is the equivalent of dividing by 2^n, regardless of sign. In addition are a few new methods that simplify working with bits: `testBit`, `setBit`, and `flipBit`. Two others called `bitCount` and `bitLength` need a bit more explaining. For positive integers the results are what you'd expect: `bitLength` returns the minimal number of bits required to represent the integer, and `BitCount` returns the number of one-bits in the representation. For negative numbers, `bitLength` gives a minimal length *excluding* the sign bit, and `bitCount` returns the number of zeros in the representation.

Creating and Converting

`BigInteger` objects can be converted by using a string representation of a number, a native type, or with a byte array. Using a native type, a string, or a byte array representation of a number creates `BigInteger` objects.

Strings

You can create `BigInteger` objects by using a construction that takes a string representation of a number. By default, the constructor expects the string representation to be in decimal (base 10) form. However, you can also use the standard base 16 representations, as follows:

```
BigInteger(String base10Rep)
BigInteger(String representation, int radix)
```

Table 1.15 BigInteger Bit Operations

BIT OPERATION	NATIVE TYPE NOTATION	BIGINTEGER NOTATION
AND	`a & b`	`a.and(b)`
ANDNOT	`a & ~b`	`a.andNot(b)`
NOT (complement)	`~a`	`a.not(b)`
OR	`a | b`	`a.or(b)`
XOR	`a ^ b`	`a.xor(b)`
Shift left *n* bits, signed	`a << n`	`a.shiftLeft(n)`
Shift right *n* bits, signed	`a >> n`	`a.shiftRight(n)`
Test bit *n*	`(a & (1 << n) != 0)`	`a.testBit(n)`
Set bit *n*	`(a | (1 << n))`	`a.setBit(n)`
Flip bit *n*	`(a ^ (1 << n))`	`a.flipBit(n)`

Conversely, you can retrieve a string representation in an appropriate base by using the `toString` method:

```
String toString()
String toString(int radix)
```

Numeric Types

The `BigInteger` class can be created from a long by using the static method:

```
Long val = 123456789123;
BigInteger bi = BigInteger.valueOf(val);
```

Once you have the `BigInteger` object, it can be converted directly into a numeric type by using `intValue`, `longValue`, `floatValue`, or `doubleValue`. Not that we'll mind, but there is no method to directly convert into a `short` or `char` value. If the value is too large to fit into the numeric type, a silent narrowing conversion is done (the high bits are chopped off).

Byte Arrays

Generating byte arrays and constructing byte *a*. Assuming you have a `BigInteger` object, a byte array can be generated with:

```
byte[] toByteArray()
```

And the results of this output can be used to create a new `BigInteger` object:

```
BigInteger(byte[] array)
```

However, because of signing issues, typically this format is not used for cryptographic purposes. If the bit length is a multiple of 8, and it always is for cryptographic applications, the byte array starts with an extra byte indicating the sign (0 for positive). Either your application can deal with a leading zero, or you can strip it away, as in the following:

```
byte ba[] = bi.toByteArray();
if (ba[0] == 0) {
     byte[] tmp = new byte[ba.length - 1];
     System.arraycopy(ba, 1, tmp, 0, tmp.length)
     ba = tmp;
}
```

A lower-speed option is to use strings as the common medium:

```
Byte ba[] = Hexify.decode(bi.toString(16);
```

Since `BigInteger` expects a sign-bit, you'll need to use a special constructor and manually indicate the sign by passing in 1, –1, or 0 for positive, negative, or zero:

```
BigInteger(int signum, byte[] magnitude)
```

BigInteger and Cryptography

`BigInteger` has a few special methods specifically designed for cryptography, listed in Table 1.16. These operations are discussed fully in Chapter 3.

Secret Methods in BigInteger

For Sun Microsystems to effectively test for primality, many generic methods and algorithms had to be implemented; however, they are *not* part of the public API. For people working in numbers theory or who are developing more advanced cryptographic algorithms, these "hidden" methods may be useful. Unfortunately, they are declared private; but with a few modifications to the source code, you can create your own enhanced `BigInteger` variant.

Table 1.16 BigInteger Operations Useful in Implementing Cryptographic Algorithms

OPERATION	BIGINTEGER NOTATION
Create a random nonnegative integer uniformly distributed from 0 to 2^n-1	`BigInteger (int numBits, Random r)`
Create a random integer that is prime with certainty $1 - 2^n$	`Random r = ...; int bitLength = ...; BigInteger a = BigInteger(bitLength, r)`
Check if prime with certainty using IEEE standard of $1 - 2^{100}$	`a.isProbablePrime()`
a mod b	`a.mod(b)`
a^n mod b	`a.modPow(b, n); // n is an int, not BigInteger`
find a^{-1}, such that $aa^{-1} = 1$ mod b	`a.modInv(b)`
Greatest common denominator of a, b	`a.gcb(b)`

The most interesting of these are the following methods:

```
int jacobiSymbol(int p, BigInteger n); // this is "package protected"
private boolean passesMillerRabin(int iterations)
private boolean passesLucasLehmer()
private static BigInteger lucasLehmerSequence(int z,
                                    BigInteger k, BigInteger n)
```

In addition, there is an implementation of sieve for testing primality of small numbers in the class `BitSieve`.

To "free" them:

1. Copy all the source files from this directory into a new directory.
2. Edit each file and change the package name from `sun.math` to one of your own choosing.
3. Make the `BitSieve` class "public."
4. Change the desired methods and classes to public. Note that the method `jacobiSymbol` in `BigInteger` and the class `BitSieve` do not have an access modifier, so you have to add `public` in front of the method.
5. Compile.

This code is copyrighted by Sun Microsystems. For commercial use and distribution, refer to the license agreement distributed with the source code.

Now that we have a full understanding of bit operations and the Java model, we'll discuss secret and public key cryptography.