Chapter Six

A Computing Machine

Our goal in this chapter is to show you how simple the computer that you're using really is. We will describe in detail a simple imaginary machine that has many of the characteristics of real processors at the heart of the computational devices that surround us.

You may be surprised to learn that many, many machines share these same properties, even some of the very first computers that were developed. Accordingly, we are able to tell the story in historical context. Imagine a world without computers, and what sort of device might be received with enthusiasm, and that is not far from what we have! We tell out story from the standpoint of scientific computing—there is an equally fascinating story from the standpoint of commercial computing that we touch on just briefly.

Next, our aim is to convince you that the basic concepts and constructs that we covered in Java programming are not so difficult to implement on a simple machine, using its own *machine language*. We will consider in detail conditionals, loops, functions, arrays, and linked structures. Since these are the same basic tools that we examined for Java, it follows that several of the computational tasks that we addressed in the first part of this book are not difficult to address at a lower level.

This simple machine is a link on the continuum between your computer and the actual hardware circuits that change state to reflect the action of your programs. As such it is preparation for learning how those circuits work, in the next chapter.

And that still is only part of the story. We end the chapter with a profound idea: we can use one machine to simulate the operation of another one. Thus, we can easily study imaginary machines, develop new machines to be built in future, and work with machines that may never be built.

6.1 Representing Information

THE FIRST STEP IN UNDERSTANDING HOW a computer works is to understand how information is represented within the computer. As we know from programming in Java, everything suited for processing with digital computers is represented as a sequence of 0s and 1s, whether it be numeric data, text, executable files, images, audio, or video. For each type of data, standard methods of encoding have come into widespread use: The ASCII standard associates a seven bit binary number with each of 128 distinct characters; the MP3 file format rigidly specifies how to encode each raw audio file as a sequence of 0s and 1s; the `.png` image format specifies the pixels in digital images ultimately as a sequence of 0s and 1s, and so forth.

Programs in this section

Within a computer, information is most often organized in *words*, which are nothing more than a sequence of bits of a fixed length (known as the *word size*). The word size plays a critical role in the architecture of any computer, as you will see. In early computers, 12 or 16 bits were typical; for many years 32-bit words were widely used; and nowadays 64-bit words are the norm.

The information content within every computer is nothing more nor less than a sequence of words, each consisting of a fixed number of bits, each either 0 or 1. Since we can interpret every word as a number represented in binary, all information is numbers, and all numbers are information.

The meaning of a given sequence of bits within a computer depends on the context. This is another of our mantras, which we will repeat throughout this chapter. For example, as you will see, depending on the context, we might interpret the binary string 1111101011001110 to mean the positive integer 64,206, the negative integer –1,330, the real number – 55744.0, or the two-character string `"eN"`.

Convenient as it may be for computers, the binary number system is extremely inconvenient for humans. If you are not convinced of this fact, try memorizing the 16-bit binary number 1111101011001110 to the point that you can close the book and write it down. To accommodate the computer's need to communicate in binary while at the same time accommodating our need to use a more compact representation, we introduce in this section the *hexadecimal* (base 16) number system, which turns out to be a convenient shorthand for binary. Accordingly, we begin by examining hexadecimal in detail.

Binary and Hex For the moment, consider nonnegative integers, or *natural numbers*, the fundamental mathematical abstraction for counting things. Since Babylonian times, people have represented integers using *positional notation* with a fixed *base*. The most familiar to you is certainly *decimal*, where the base is 10 and each positive integer is represented as a string of digits between zero and 9. Specifically, $d_n d_{n-1} ... d_2 d_1 d_0$ represents the integer

$$d_n 10^n + d_{n-1} 10^{n-1} + . . . + d_2 10^2 + d_1 10^1 + d_0 10^0$$

For example, `10345` represents the integer

$$10345 = 1 \cdot 10000 + 0 \cdot 1000 + 3 \cdot 100 + 4 \cdot 10 + 5 \cdot 1.$$

We can replace the base 10 by any integer greater than 1 to get a different number system where we represent any integer by a string of digits, all between 0 and one less than the base. For our purposes, we are particularly interested in two such systems: *binary* (base 2) and *hexadecimal* (base 16).

Binary. When the base is two, we represent an integer as a sequence of `0`s and `1`s. In this case, we refer to each binary (base 2) digit—either `0` or `1`—as a *bit* , the basis for representing information in computers. In this case the bits are coefficients of powers of 2. Specifically, the sequence of bits $b_n b_{n-1} ... b_2 b_1 b_0$ represents the integer

$$b_n 2^n + b_{n-1} 2^{n-1} + ... b_2 2^2 + b_1 2^1 + b_0 2^0$$

For example, `1100011` represents the integer

$$99 = 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1.$$

Note that the largest integer that we can represent in an n-bit word in this way is $2^n - 1$, when all n bits are 1. For example, with 8 bits, `11111111` represents

$$2^8 - 1 = 255 = 1 \cdot 128 + 1 \cdot 64 + 1 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 1 \cdot 1.$$

Another way of stating this limitation is that we can represent only 2^n nonnegative integers (0 through $2^n - 1$) in an n-bit word. We often have to be aware of such limitations when processing integers with a computer. Again, a big disadvantage of using binary notation is that the number of bits required to represent a number in binary is much larger than, for example, the number of digits required to represent the same number in decimal. Using binary exclusively to communicate with a computer would be unwieldy and impractical.

Hexadecimal. In hexadecimal (or just *hex* from now on) the sequence of hex digits $h_n h_{n-1} \ldots h_2 h_1 h_0$ represents the integer

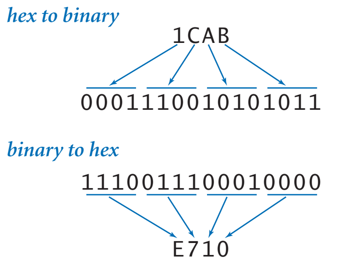
$$h_n 16^n \ + h_{n-1} 16^{n-1} \ + \ldots h_2 16^2 \ + h_1 16^1 \ + h_0 16^0$$

The first complication we face is that, since the base is 16, we need digits for each of the values 0 through 15. We need to have one character to represent each digit, so we use A for 10, B for 11, C for 12, and so forth, as shown in the table at left. For example, FACE represents the integer

$$64{,}206 = 15 \cdot 16^3 + 10 \cdot 16^2 + 12 \cdot 16^1 + 14 \cdot 16^0.$$

This is the same integer that we represented with 16 bits earlier. As you can see from this example, the number of hex digits needed to represent integers in hexadecimal is only a fraction (about one-fourth) of the number of bits needed to represent the same integer in binary. Also, the variety in the digits makes a number easy to remember. You may have struggled with 1111101011001110, but you certainly can remember FACE.

Conversion between hex and binary. Given the hex representation of a number, finding the binary representation is easy, and vice-versa, as illustrated in the figure at left. Since the hex base 16 is a power of the binary base 2, we can convert groups of four bits to hex digits and vice versa. To convert from hex to binary, replace each hex digit by the four binary bits corresponding to its value (see the table at right). Conversely, given the binary representation of a number, add leading 0s to make the number of bits a multiple of 4, then group the bits four at a time and convert each group to a single hex digit. You can do the math to prove that these conversions are always correct (see Exercise 5.1.8), but just a few examples should serve to convince you. For example the hex representation of the integer 39 is 27, so the binary representation is 00100111 (and we can drop the leading zeros); the binary representation of 228 is 11100100, so the hex representation is E4. This ability to convert quickly from binary to hex and from hex to binary is important as



Hex-binary conversion examples

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

Representations of integers from 0 to 15 an efficient way to communicate with the computer. You will be surprised at how quickly you will learn this skill, once you internalize the basic knowledge that `A` is equivalent to `1010`, `5` is equivalent to `0101`, `F` is equivalent to `1111`, and so forth.

Conversion from decimal to binary. We have considered the problem of computing the string of `0`s and `1`s that represent the binary number corresponding to a given integer as an early programming example. The following recursive program (the solution to Exercise 2.3.15) does the job, and is worthy of careful study:

```
public static String toString(int N)
{
    if (N == 0) return "";
    return toString(N/2) + (char) ('0' + (N % 2));
}
```

It is a recursive method based on the idea that the last digit is the character that represents `N % 2` (`'0'` if `N % 2` is `0` and `'1'` if `N % 2` is `1`) and the rest of the string is the string representation of `N / 2`. A sample trace of this program is shown at right. This method generalizes to handle hexadecimal (and any other base), and we also are interested in converting string representations to Java data-type values. In the next section, we consider a program that accomplishes such conversions.

```
toString(109)
   toString(54)
      toString(27)
         toString(13)
            toString(6)
               toString(3)
                  toString(1)
                     toString(0)
                        return ""
                     return "1"
                  return "11"
               return "110"
            return "1101"
         return "11011"
      return "110110"
   return "1101101"
```

Call trace for `toString(109)`

When we talk about what is going on within the computer, our language is hex. The contents of an n-bit computer word can be specified with $n/4$ hex digits and immediately translated to binary if desired. You likely have observed such usage already in your daily life. For example, when you register a new device on your network, you need to know its media access control (MAC) address. A MAC address such as `1a:ed:b1:b9:96:5e` is just hex shorthand (using some superflous colons and lowercase `a-f` instead of the uppercase `A-F` that we use) for a 48-bit binary number that identifies your device for the network.

Later in this chapter, we will be particularly interested in integers less than 256, which can be specified with 8 bits and 2 hex digits. For reference, we have

dec	*binary*	*hex*	*dec*	*binary*	*hex*	*dec*	*binary*	*hex*	*dec*	*binary*	*hex*
0	00000000	00	32	00100000	20	64	01000000	40	96	01100000	60
1	00000001	01	33	00100001	21	65	01000001	41	97	01100001	61
2	00000010	02	34	00100001	22	66	01000001	42	98	01100001	62
3	00000011	03	35	00100001	23	67	01000001	43	99	01100001	63
4	00000100	04	36	00100100	24	68	01000100	44	100	01100100	64
5	00000101	05	37	00100101	25	69	01000101	45	101	01100101	65
6	00000110	06	38	00100110	26	70	01000110	46	102	01100110	66
7	00000111	07	39	00100111	27	71	01000111	47	103	01100111	67
8	00001000	08	40	00101000	28	72	01001000	48	104	01101000	68
9	00001001	09	41	00101001	29	73	01001001	49	105	01101001	69
10	00001010	0A	42	00101010	2A	74	01001010	4A	106	01101010	6A
11	00001011	0B	43	00101011	2B	75	01001011	4B	107	01101011	6B
12	00001100	0C	44	00101100	2C	76	01001100	4C	108	01101100	6C
13	00001101	0D	45	00101101	2D	77	01001101	4D	109	01101101	6D
14	00001110	0E	46	00101110	2E	78	01001110	4E	110	01101110	6E
15	00001111	0F	47	00101111	2F	79	01001111	4F	111	01101111	6F
16	00010000	10	48	00110000	30	80	01010000	50	112	01110000	70
17	00010001	11	49	00110001	31	81	01010001	51	113	01110001	71
18	00010010	12	50	00110010	32	82	01010010	52	114	01110010	72
19	00010011	13	51	00110011	33	83	01010011	53	115	01110011	73
20	00010100	14	52	00110100	34	84	01010100	54	116	01110100	74
21	00010101	15	53	00110101	35	85	01010101	55	117	01110101	75
22	00010110	16	54	00110110	36	86	01010110	56	118	01110110	76
23	00010111	17	55	00110111	37	87	01010111	57	119	01110111	77
24	00010000	18	56	00110000	38	88	01010000	58	120	01111000	78
25	00010001	19	57	00110001	39	89	01010001	59	121	01111001	79
26	00010010	1A	58	00110010	3A	90	01010010	5A	122	01111010	7A
27	00010011	1B	59	00110011	3B	91	01010011	5B	123	01111011	7B
28	00010100	1C	60	00110100	3C	92	01010100	5C	124	01111100	7C
29	00010101	1D	61	00110101	3D	93	01010101	5D	125	01111101	7D
30	00011110	1E	62	00111110	3E	94	01011110	5E	126	01111110	7E
31	00011111	1F	63	00111111	3F	95	01011111	5F	127	01111111	7F

Decimal, 8-bit binary, and 2-digit hex representations of integers from 0 to 127

dec	*binary*	*hex*	*dec*	*binary*	*hex*	*dec*	*binary*	*hex*	*dec*	*binary*	*hex*
128	10000000	80	160	10100000	A0	192	11000000	C0	224	11100000	E0
129	10000001	81	161	10100001	A1	193	11000001	C1	225	11100001	E1
130	10000010	82	162	10100010	A2	194	11000010	C2	226	11100010	E2
131	10000011	83	163	10100011	A3	195	11000011	C3	227	11100011	E3
132	10000100	84	164	10100100	A4	196	11000100	C4	228	11100100	E4
133	10000101	85	165	10100101	A5	197	11000101	C5	229	11100101	E5
134	10000110	86	166	10100110	A6	198	11000110	C6	230	11100110	E6
135	10000111	87	167	10100111	A7	199	11000111	C7	231	11100111	E7
136	10001000	88	168	10101000	A8	200	11001000	C8	232	11101000	E8
137	10001001	89	169	10101001	A9	201	11001001	C9	233	11101001	E9
138	10001010	8A	170	10101010	AA	202	11001010	CA	234	11101010	EA
139	10001011	8B	171	10101011	AB	203	11001011	CB	235	11101011	EB
140	10001100	8C	172	10101100	AC	204	11001100	CC	236	11101100	EC
141	10001101	8D	173	10101101	AD	205	11001101	CD	237	11101101	ED
142	10001110	8E	174	10101110	AE	206	11001110	CE	238	11101110	EE
143	10001111	8F	175	10101111	AF	207	11001111	CF	239	11101111	EF
144	10010000	90	176	10110000	B0	208	11010000	D0	240	11110000	F0
145	10010001	91	177	10110001	B1	209	11010001	D1	241	11110001	F1
146	10010010	92	178	10110010	B2	210	11010010	D2	242	11110010	F2
147	10010011	93	179	10110011	B3	211	11010011	D3	243	11110011	F3
148	10010100	94	180	10110100	B4	212	11010100	D4	244	11110100	F4
149	10010101	95	181	10110101	B5	213	11010101	D5	245	11110101	F5
150	10010110	96	182	10110110	B6	214	11010110	D6	246	11110110	F6
151	10010111	97	183	10110111	B7	215	11010111	D7	247	11110111	F7
152	10010000	98	184	10110000	B8	216	11010000	D8	248	11111000	F8
153	10010001	99	185	10110001	B9	217	11010001	D9	249	11111001	F9
154	10010010	9A	186	10110010	BA	218	11010010	DA	250	11111010	FA
155	10010011	9B	187	10110011	BB	219	11010011	DB	251	11111011	FB
156	10010100	9C	188	10110100	BC	220	11010100	DC	252	11111100	FC
157	10010101	9D	189	10110101	BD	221	11010101	DD	253	11111101	FD
158	10011110	9E	190	10111110	BE	222	11011110	DE	254	11111110	FE
159	10011111	9F	191	10111111	BF	223	11011111	DF	255	11111111	FF

Decimal, 8-bit binary, and 2-digit hex representations of integers from 128 to 255 included on the presious two pages a complete table of their representations in decimal, binary and hex. A few minutes studying this table is worth your while, to give you confidence in working with such integers and understanding relationships among these representations. If you believe, after doing so, that the table is a waste of space, then we have achieved our goal!

Parsing and string representations Converting among different representations of integers is an interesting computational task, which we first considered in Program 1.3.7 and then revisited in Exercise 2.3.15. We have also been making use of Java's methods for such tasks throughout. Next, to cement ideas about positional number representations with various bases, we will consider a program for converting any number from one base to another.

Parsing. Converting a string of characters to an internal representation is called *parsing.* Since Section 1.1, we have been using Java methods like `Integer.parseInt()` and our own methods like `StdIn.readInt()` to convert numbers from the strings that we type to values of Java's data types. We have been using decimal numbers (represented as strings of the characters between `0` and `9`), now we look at a method to parse numbers written in any base. For simplicitly, we limit ourselves to bases no more than 36 and extend our convention for hex to use the letters `A` though `Z` to represent digits from 10 to 35. *Note*: Java's `Integer` class has a two-argument `parseInt()` method that has similar functionality, except that it also handles negative integers.

`i`	`N`	*characters seen*
0	1	1
1	3	11
2	6	110
3	13	1101
4	27	11011
5	54	110110
6	109	1101101

Trace of `parseInt(1101101, 2)`

One of the the hallmark features of modern data types is that *the internal representation is hidden from the user*, so we can only use defined operations on data type values to accomplish the task. Specifically, it is best to limit direct reference to the bits that represent a data type value, but to use data-type operations instead.

The first primitive operation that we need to parse a number is a method that converts a character into an integer. Exercise 6.1.12 gives a method `toInt()` that takes a character in the range `0-9` or `A-Z` as argument and returns an `int` value between 0 and 35 (0-9 for digits and 10-35 for letters). With this primitive, the rather simple method `parseInt()` in Program 6.1.1 parses the string representation of an integer in any base `b` from 2 to 36 and returns the Java `int` value for that integer. As usual, we can convince ourselves that it does so by reasoning about the effect

Program 6.1.1 Converting a natural number from one base to another

```
public class Convert
{
    public static int toInt(char c)
    { // See Exercise 5.1.12 }
    public static char toChar(int i)
    { // See Exercise 5.1.13 }
    public static int parseInt(String s, int d)
    {
        int N = 0;
        for (int i = 0; i < s.length(); i++)
            N = d*N + toInt(s.charAt(i));
        return N;
    }
    public static String toString(int N, int d)
    {
        if (N == 0) return "";
        return toString(N/d, d) + toChar(N % d);
    }
    public static String toString(int N, int d, int w)
    { // See Exercise 5.1.15 }
    public static void main(String[] args)
    {
        while (!StdIn.isEmpty())
        {
            String s = StdIn.readString();
            int baseFrom = StdIn.readInt();
            int baseTo = StdIn.readInt();
            int N = parseInt(s, baseFrom);
            StdOut.println(toString(N, baseTo));
        }
    }
}
```

```
% java Convert
1101101 2 10
109
FACE 16 10
64206
FACE 16 2
1111101011001110
109 10 2
1101101
64206 10 16
FACE
64206 10 32
1UME
1UME 32 10
64206
```

This general-purpose conversion program reads strings and pairs of bases from standard input and uses `parseInt()` *and* `toString()` *to convert the string from a representation of an integer in the first base to a representation of the same integer in the second base.*

of the code in the loop: each time through the loop the `int` value `N` is the integer corresponding to all the digits seen so far: to maintain this invariant, all we need to do is multiply by the base and add the value of the next digit. The trace shown here illustrates the process: each value of `N` is the base times the previous value of `N` plus the next digit (in blue). To parse `1101101`, we compute $0 \cdot 2 + 1 = 1$, $1 \cdot 2 + 1 = 3$, $3 \cdot 2 + 0 = 6$, $6 \cdot 2 + 1 = 13$, $13 \cdot 2 + 1 = 27$, $27 \cdot 2 + 0 = 54$, and $54 \cdot 2 + 1 = 109$. To parse `FACE` as a hex number, we compute $0 \cdot 16 + 15 = 15$, $15 \cdot 16 + 10 = 250$, $250 \cdot 16 + 12 = 4012$, and $4012 \cdot 16 + 14 = 64206$.

i	N	*characters seen*
0	15	"F"
1	250	"FA"
2	4012	"FAC"
3	64206	"FACE"

Trace of `parseInt(FACE, 16)`

For simplicity, we have not included error checks in this code. For example, `parseInt()` should raise an exception if the value returned by `toInt()` is not less than the base. Also, it should throw an exception on overflow, as the input could be a string that represents a number larger than can be represented as a Java `int`.

String representation. Using a `toString()` method to compute a string representation of a data-type value is also something that we have been doing since the beginning of this book. We use a recursive method that generalizes the decimal-to-binary method (the solution to Exercise 2.3.15) that we considered earlier in this section. Again, it is instructive to look at a method to compute the string representation of an integer in any given base, even though Java's `Integer` class has a two-argument `toString()` method that has similar functionality.

Again, the first primitive operation that we need is a method that converts an integer into a character (digit). Exercise 6.1.13 gives a method `toChar()` that takes an `int` value between 0 and 35 and returns a character in the range `0-9` (for values less than 10) or `A-Z` (for values from 10 to 35). With this primitive, the `toString()` method in Program 6.1.1 is even simpler than `parseInt()`. It is a recursive method based on the idea that the last digit is the character representation of `N % d` and the rest of the string is the string representation of `N / d`. The computation is essentially the inverse of the computation for parsing, as you can see from the call trace shown here.

```
toString(64206, 16)
   toString(4012, 16)
      toString(250, 16)
         toString(15, 16)
            toString(0, 16)
               return ""
            return "F"
         return "FA"
      return "FAC"
   return "FACE"
```

Call trace for `toString(64206, 16)`

When discussing the contents of computer words, we need to include leading zeros, so that we are specifying all the bits. For this reason, we include a three-argument version of `toString()` in `Convert`, where the third argument is the desired number of digits in the returned string. For example, the call `toString(15, 16, 4)` returns `000F` and the call `toString(14, 2, 16)` returns `0000000000001110`. Implementation of this version is left for an exercise (see Exercise 6.1.15).

Putting these ideas all together, Program 6.1.1 is a general-purpose tool for computing numbers from one base to another. While the standard input stream is not empty, the main loop in the test client reads a string from standard input, followed by two integers (the base in which the string is expressed and the base in which the result is to be expressed) and performs the specified conversion and prints out the result. To accomplish this task, it uses `parseInt()` to convert the input string to a Java `int`, then it uses `toString()` to convert that Java `int` to a string representation of the number in the specified base. You are encourage to download and make use of this tool to familiarize yourself with number conversion and representation.

Integer arithmetic The first operations that we consider on integers are basic arithmetic operations like addition and multiplication. Indeed, the primary purpose of early computing devices was to perform such operations repeatedly. In the next chapter, we will be studying the idea of building computational devices that can do so, since every computer has built-in hardware for performing such operations. For the moment, we illustrate that the basic methods that you learned in grade school for decimal work perfectly well in binary and hex.

Addition. In grade school you learned how to add two decimal integers: add the two least significant digits (rightmost digits); if the sum is more than 10, then carry a 1 and write down the sum modulo 10. Repeat with the next digit, but this time include the carry bit in the addition. The same procedure generalizes to any base. For example, if you are working in hex and the two summand digits are 7 and E, then you should write down a 5 and carry a 1 because 7 + E is 15 in hex. If you are working in binary and the two summand bits are 1 and the carry is 1 then you should write down a 1 and carry the 1 because 1+1+1 = 11 in binary. The examples at left illustrate how to compute the sum $4567_{10} + 366_{10} = 4933_{10}$ in decimal, hex, and binary. As in grade school, we supress leading zeros.

decimal

```
 0011  ← carries
 4567
  366
 ----
 4933
```

hex

```
 0011
 11D7
  16E
 ----
 1345
```

binary

```
00001111111110
 1000111010111
     101101110
 -------------
 1001101000101
```

Addition

Unsigned integers. If we want to represent integers within a computer word, we are immediately accepting a limitation on the number and size of integers that we can represent. As already noted, we can represent only 2^n integers in an n-bit word. If we want just nonnegative (or unsigned) integers the natural choice is to use binary for the integers 0 through $2^n - 1$, with leading 0s so that every word corresponds to an integer and every integer within the defined range to a word. The table at right shows the 16 unsigned integers we can represent in a 4-bit word, and the table at left shows the range of representable integers for the 16-bit, 32-bit, and 64-bit word sizes that are used in typical computers.

bits	*smallest*	*largest*
4	0	15
16	0	65,535
32	0	4,294,967,295
64	0	18,446,744,073,709,551,615

Representable unsigned integers

decimal	*binary*
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

4-bit integers (unsigned)

Overflow. As you have already seen with Java programming in SECTION 1.2, we need to pay attention that the value of the result of an arithmetic operation does not exceed the maximum possible value. This condition is called *overflow.* For addition in unsigned integers, overflow is easy to detect: if the last (leftmost) addition causes a carry, then the result is too large to represent. Testing the value of one bit is easy, even in computer hardware (as you will see), so computers and programming languages typically include low-level instructions to test for this possibility. Remarkably, Java does not do so (see the Q&A in SECTION 1.2).

```
carry out
indicates
overflow
 ↓
1000000111111000
 1111111111111000
 0000000000001000
 ----------------
 0000000000000000
```

Overflow (16-bit unsigned)

Multiplication. Similarly, as illustrated in the diagram at right, the grade-school algorithm for multiplication works perfectly well with any base. (The binary example is difficult to follow because of cascading carries: if you try to check it, add the numbers two at a time.) Actually, computer scientists have discovered multiplication algorithms that are much more suited to implementation in a computer and much more efficient than this method. In early computers, programmers had to do multiplication in *software* (we will illustrate such an implementation much later, in EXERCISE 5.3.38). Note that overflow is much more of a concern in developing a multiplication algorithm than for addition, as the number of bits in the result can be twice the number of bits in the operands. That is, when you multiply two n-bit numbers, you need to be prepared for a $2n$-bit result.

```
decimal
   4567
*   366
-------
  27402
 27402
13701
-------
1671522

hex
   11D7
*   16E
-------
   F9C2
  6B0A
 11D7
-------
 198162

binary
          1000111010111
*         0000101101110
  ---------------------
        1000111010111
       1000111010111
      1000111010111
    1000111010111
   1000111010111
 1000111010111
  ---------------------
  110010100000101100010
```

Multiplication examples

IN THIS BOOK, WE CERTAINLY CANNOT describe in depth all of the techniques that have been developed to perform arithmetic operations with computer hardware. Of course, you want your computer to perform division, exponentiation, and other operations efficiently. Our plan is to cover addition/subtraction in full detail and just some brief indication about other operations.

You also want to be able to compute with negative numbers and real numbers. Next, we briefly describe standard representations that allow for that.

Negative numbers Computer designers discovered early on that is is not difficult to modify the integer data type to include negative numbers, using a representation known as *two's complement*.

The first thing that you might think of would be to use a *sign-and-magnitude* representation, where the first bit is the sign and the rest of bits the magnitude of the number. For example, with 4 bits in this representation 0101 would represent +5 and 1101 would represent –5. By contrast, in n-bit two's complement, we represent positive numbers as before, but each negative number $-x$ with the (positive, unisigned) binary number $2^n - x$. For example, the table at left shows the 16 two's complement integers we can represent in a 4-bit word. You can see that 0101 still represents +5 but 1011 represents –5 because $2^4 - 5 = 11_{10} = 1011_2$.

decimal	*binary*
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

4-bit integers (two's complement)

With one bit reserved for the sign, the largest two's complement number that we can represent is about half the largest unsigned integer that we could represent with the same number of bits. As you can see from the 4-bit example, there is a slight asymmetry in two's complement: We represent the positive numbers 1 through 7 and the negative numbers – 8 through –1 and we have a single representation of 0. In general, in n-bits two's complement, the smallest possible negative number is -2^{n-1} and the largest possible positive number is $2^{n-1} - 1$. The table at left shows the smallest and largest (in absolute value) 16-bit two's complement integers.

decimal	*binary*
0	0000000000000000
1	0000000000000001
2	0000000000000010
3	0000000000000011
. . .	. . .
32765	0111111111111101
32766	0111111111111110
32767	0111111111111111
-32768	1000000000000000
-32767	1000000000000001
-32766	1000000000000010
-32765	1000000000000011
. . .	. . .
-3	1000000000001101
-2	1111111111111110
-1	1111111111111111

16-bit integers (two's complement)

There are two primary reasons that two's complement evolved as the standard over sign-and-magnitude. First, because there is only one representation of 0 (the binary string that is all 0s), testing whether a value is 0 is as easy as possible. Second, arithmetic operations are easy to implement—we discuss this for addition below. Moreover, as with sign-and-magnitude, the leading bit indicates the sign, so testing whether a value is negative is as easy as possible. Building computer hardware is sufficiently difficult that achieving these simplifications just by adopting a convention on how we represent numbers is compelling.

Addition. Also, adding two n-bit two's complement integers is easy: *add them as if they were unsigned integers*. For example, $2 + (-7)$ = `0010` + `1001` = `1011` = –5. Proving that this is the case when result is within range (between -2^{n-1} and $2^{n-1}-1$) is not difficult:

- If both integers are nonnegative then standard binary addition as we have described it applies, as long as the result is less than 2^{n-1}.
- If both are negative, then the sum is
  $$(2^n - x) + (2^n - y) = 2^n + 2^n - (x + y)$$
- If x is negative, y is positive, and the result is negative, then we have
  $$(2^n - x) + y = 2^n - (x - y)$$
- If x is negative, y is positive, and the result is positive, then we have
  $$(2^n - x) + y = 2^n + (y - x)$$

In the second and fourth cases, the extra 2^n term does not contribute to the n-bit result (it is the carry out), so a standard binary addition (ignoring the carry out) gives the result. Detecting overflow is a bit more complicated than for unsigned integers—we leave that for the Q&A.

```
 0000000000000000
 0000000001000000     64
 0000000000101010    +42
 ----------------   ----
 0000000001101010    106

 1111111111000000
 0000000001000000     64
 1111111111010110    -42
 ----------------   ----
 0000000000010110     22

 0000000000000000
 1111111111000000    -64
 0000000000101010    +42
 ----------------   ----
 1111111111101010    -22

 1111111111000000
 1111111111000000    -64
 1111111111010110    -42
 ----------------   ----
 1111111110010110   -106
```

Addition (16-bit two's complement)

Subtraction. To compute $x - y$ we compute $x + (-y)$. That is we can still use standard binary addition, if we know how to compute $-y$. It turns out that negating a number is very easy in two's complement: *flip the bits and then add 1*. Three examples of this process are shown at left—we leave the proof that it works for an exercise.

```
  0000000000001010    10
  1111111111110101    flip all bits
 +0000000000000001    add 1
  ----------------
  1111111111110110   -10

  1111111111110110   -10
  0000000000001001    flip all bits
 +0000000000000001    add 1
  ----------------
  0000000000001010    10

  0001001101001110    4942
  1110110010110001    flip all bits
 +0000000000000001    add 1
  ----------------
  1110110010110010   -4942
```

Negating two's complement numbers

Knowing two's complement is relevant for Java programmers because `short`, `int`, and `long` values are 16-, 32-, and 64-bit two's complement integers, respectively. This explains the bounds on values of thse types that Java programmers have to be aware of (shown in the table at the top of the next page).

Moreover, Java's two's complement representation explains the behavior on overflow in Java that we first observed in Section 1.2 (see the Q&A in that section, and Exercise 1.2.10). For example, we saw that, in any of Java integer types, the result of adding 1 to the largest positive integer, the result is the largest negative integer. In 4-bit two's complement, incrementing `0111` gives `1000`; in 16-bit two's complement, incrementing `0111111111111111` gives `1000000000000000`. (Note that this is the *only* case where incrementing a two's complement integer does not produce the expected result.) The behavior of the other cases in Exercise 1.2.10 are also as easily explained. For decades, such behavior has bewildered programmers who do not take the time to learn about two's complement. Here's one convincing example: in Java, the call `Math.abs(-2147483648)` returns `-2147483648`, a negative integer!

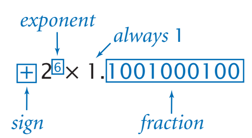
16-bit	*smallest*	−32,768
	largest	32,767
32-bit	*smallest*	−2,147,483,648
	largest	2,147,483,647
64-bit	*smallest*	−9,223,372,036,854,775,808
	largest	9,223,372,036,854,775,807

Representable two's complement integers

Real numbers How do we represent real numbers? This task is a bit more challenging than integers, as there are many choices to be made. Early computer designers tried many, many options and numerous competing formats evolved during the first few decades of digital computation. Arithmetic on real numbers was actually implemented in *software* for quite a while, and was quite slow by comparison with integer arithmetic.

By the mid 1980s, the need for a standard was obvious (different computers might get slightly different results for the same computation), so the Institute for Electrical and Electronic Engineers (IEEE) developed a standard known as the *IEEE 754* standard that is under development to this day. The standard is extensive—you may not want to know the full details—but we can describe the basic ideas briefly here. We illustrate with a 16-bit version known as the *IEEE 754 half-precision binary floating-point format* or `binary16` for short. The same essential ideas apply to the 32-bit and 64-bit versions used in Java.

Floating point. The real-number representation that is commonly used in computer systems is known as *floating-point*. It is just like scientific notation, except that everything is representing in binary. In scientific notation, you are used to working with numbers like



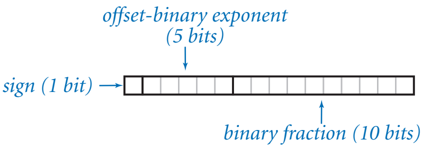
Anatomy of a floating point number

$+\ 6.0221413 \times 10^{23}$, which consist of a *sign*, a *coefficient*, and an *exponent*. Typically the number is expressed such that the coefficient is one (nonzero) digit. This is known as a *normalization* condition. In floating point, we have the same three elements.

Sign. The first bit of a floating point number is its *sign*. Nothing special is involved: the sign bit is `0` is the number is positive and `1` if it is negative. Again, checking whether a number is positive or negative is easy.

Exponent. The next t bits of a floating point number are devoted to its *exponent*. The number of bits used for `binary16`, `binary32`, and `binary64` are 5, 8, and 11, respectively. The exponent of a floating point number is not expressed in two's complement, but rather *offset-binary*, where we take $R = 2^{\,t-1}$ and represent any decimal number x between $-R$ and $R+1$ with the binary representation of $x+R$. The table at right shows the 5-bit offset binary representations of the numbers between -15 and $+16$. In the standard, `0000` and `1111` are used for other purposes.

Fraction. The rest of the bits in a floating point number are devoted to the coefficient: 10, 23, and 53 bits for `binary16`, `binary32`, and `binary64`, respectively. The normalization condition implies that the digit before the decimal place in the coefficient is always 1, so we need not include that digit in the representation!



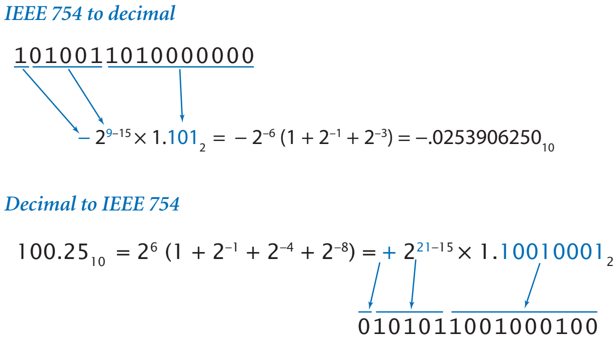
IEEE 754 half-precision format (`float`)

Given these rules, the process of *decoding* a number encoded in IEEE 754 format is straightforward, as illustrated in the top example in the figure at the top of the next page. According to the standard, the first bit in the given 16-bit quantity is the sign, the next five bits are the offset binary encoding of the exponent (-6_{10}), and the next 10 bits are the fraction, which defines the coefficient 1.101_2. The process of *encoding* a number, illustrated in the bottom example, is more complicated, due to the need to normalize and to extend binary conversion to include fractions. Again, the first bit is the sign bit, the next five bits are the exponent, and the next

decimal	*binary*
-15	00000
-14	00001
-13	00010
-12	00011
-11	00100
-10	00101
-9	00110
-8	00111
-7	01000
-6	01001
-5	01010
-4	01011
-3	01100
-2	01101
-1	01110
0	01111
1	10000
2	10001
3	10010
4	10011
5	10100
6	10101
7	10110
8	10111
9	11000
10	11001
11	11010
12	11011
13	11100
14	11101
15	11110
16	11111

5-bit integers (offset binary)

10 bits are the fraction. These tasks make for an challenging programming exercise even in a high-level language like Java (see Exercise 6.1.25, but first read about manipulating bits in the next subsection), so you can imagine why floating point numbers were not supported in early computer hardware and why it took so long for a standard to evolve.



Floating point-decimal conversion examples

The Java `Float` and `Double` data types include a `floatToIntBits()` method that you can use to check floating-point encoding. For example, the call

```
Convert.toString(Float.floatToIntBits(100.25), 2, 16)
```

prints the result `0101011001000100` as expected from the bottom example above.

Arithmetic. Performing arithmetic on floating point numbers also makes for an interesting programming exercise. For example, the following steps are required to multiply two floating point numbers:

- Exclusive or the signs.
- Add the exponents.
- Multiply the fractions.
- Normalize the result.

If you are interested, you can explore the details of this process and the corresponding process for addition and for multiplication by working Exercise 5.1.25. Addition is actually a bit more complicated than multiplication, because it is necessary to "unnormalize" to make the exponents match as the first step.

Computing with floating point numbers is often challenging because they are most often approximations to the real numbers of interest, and errors in the approximation can accumulate during a long series of calculations. Since the 64-bit format (used in Java's `double` data type) has more than twice as many bits in the fraction as the 32-bit format (used in Java's `float` data type), most programmers choose to use `double` to lessen the effects of approximations errors, as we do in this book.

Java code for manipulating bits As you can see from floating-point encoding of real numbers, encoding information in binary can get complicated. Next, we consider the tools available within Java that make it possible to write programs to encode and decode information. These tools are made possible because Java *defines* integer values to be two's complement integers, and makes explicit that the values of the `short`, `int`, and `long` data types are 16- 32- and 64-bit binary two's complement, respectively. Not all languages do so, leaving such code to a lower-level language, defining an explicit data type for bit sequences, and/or perhaps requiring difficult or expensive conversion. We focus on 32-bit `int` values, but the operations also work for `short` and `long` values.

Binary and hex literals. In Java, it is possible to specify integer literal values in binary (by prepending `0b`) and in hex (by prepending `0x`). This ability substantially clarifies code that is working with binary values. You can use literals like this anywhere that you can use a normal literal; it is just another way of specifying an integer value. If you assign a hex literal to an `int` variable and specify fewer than 8 digits, Java will fill in leading zeros. A few examples are shown in this table.

binary literal	*hex literal*	*shorter form*
`0b01000000010101000100111101011001`	`0x40544F59`	
`0b11111111111111111111111111111111`	`0x0000000F`	`0xF`
`0b00000000000000000001001000110100`	`0x00001234`	`0x1234`
`0b00000000000000001000101000101011`	`0x00008A2B`	`0x8A2B`

Shifting and bitwise operations in Java code. To allow clients to manipulate the bits in an `int` value, Java supports the following bitwise and shifting operations:

values	32-bit integers					
typical literals	`0b00000000000000000000000000001111` `0b1111` `0xF` `0x1234`					
operations	bitwise complement	bitwise and	bitwise or	bitwise xor	shift left	shift right
operators	`~`	`&`	`|`	`^`	`<<`	`>>`

Bit manipulation operators for Java's built-in `int` data type

We can complement the bits, do bitwise logical operations, and shift left or right a given number of bit positions.

Shifting and masking. One of the primary uses of such operations is *masking*, where we isolate a bit or a group of bits from the others in the same word. Going a bit further, we often do *shifting and masking* to extract the integer value that a contiguous group of bits represent, as follows:

- Use a *shift right* instruction to put the bits in the rightmost position.
- If we want *k* bits, create a literal mask whose bits are all `0` except its *k* rightmost bits, which are `1`.
- Use a *bitwise and* to isolate the bits. The `0`s in the mask lead to zeros in the result; the `1`s in the mask give the bits of interest in the result.

This sequence of operations puts us in a position to use the result as we would any other `int` value, which is often what is desired.

bitwise and

```
  01010001110101110000000000001111
& 00110001011011100011000101101110
  00010001010001100000000000001110
```

bitwise xor

```
  01010001110101110000000000001111
^ 00110001011011100011000101101110
  01100000101110010011000101100001
```

shift left 6

```
  01010001110101110000000000001111
<<00000000000000000000000000000110
  01110101110000000000001111000000
```

shift right 3

```
  01010001110101110000000000001111
>>00000000000000000000000000000011
  00001010001110101110000000000001
```

Bitwise instructions (32 bits)

Usually we prefer to specify masks as hex constants. For example, the mask `0x80000000` can be used to isolate the leftmost bit in a 32-bit word, the mask `0x000000FF` can be used to isolate the rightmost 8 bits, and the mask `0x007FFFFF` can be used to isolate the rightmost 23 bits. Later in this chapter we will be interested in shifting and masking to isolate hex digits, as shown in the examples at left.

expression	*value*	*comment*
`0x00008A2B & 0x00000F00`	`0x00000A00`	*isolates digit*
`0x00008A2B >> 8`	`0x0000008A`	*shift right*
`(0x00008A2B >> 8) & 0xF`	`0x0000000A`	*extracts digit*

Typical hex-digit-manipulation expressions

As an example of a practical application, Program 6.1.2 illustrates the use of shifting and masking to extract the sign, exponent and fraction from a floating point number. Most computer users are able to work comfortably without dealing with data representations at this level (indeed, we have hardly needed it so far in this book), but bit manipulation plays an important role in all sorts of applications.

Program 6.1.2 Extracting the components of a floating point number

```
public class ExtractFloat
{
   public static void main(String[] args)
   {
      while (!StdIn.isEmpty())
      {
         float x = StdIn.readFloat();
         int t = Float.floatToIntBits(x);
         if ((t & 80000000) == 1)
              StdOut.println("    Sign: -");
         else StdOut.println("    Sign: +");

         int exp = ((t >> 23) & 0xFF) - 127;
         StdOut.println("Exponent: " + exp);

         double frac =  1.0 * (t & 0x007FFFFF)   / (0b1 << 23);
         StdOut.println("Fraction: " + frac);

         StdOut.println((float) (Math.pow(2, exp) * (1 + frac)));
       }
   }
}
```

This program illustrates the use of Java bit maniulation operations by extracting the sign, exponent and fraction fields from `float` values entered on standard input, then using them to recompute the value.

```
% java ExtractFloat
100.25
    Sign: +
Exponent: 6
Fraction: 0.56640625
100.25
3.141592653589793
    Sign: +
Exponent: 1
Fraction: 0.5707963705062866
3.1415927410125732
```

Characters In order to process text, we need a binary encoding for characers. The basic method is quite simple: a table defines the correspondence between characters and *n*-bit unsigned binary integers. With six bits, we can encode 64 different characters; with seven bits, 128 different characters, with eight bits, 256 different characters and so forth. As with floating point, many different schemes evolved as computers came into use, and people still use different encodings in different situations.

ASCII. the *American Standard Code for Information Interchange* (*ASCII*) code was developed as a standard in the 1960s, and has been in widespread use ever since. It is a 7-bit code, though in modern computing it most often is used in 8-bit bytes with the leading bit ignored.

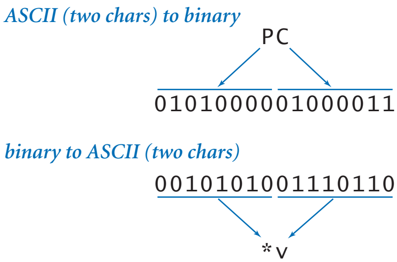
One of the primary reasons for the development of ASCII was for communication via teletypewriters that could send and receive text. Accordingly, many of the encoded characters are *control characters* for such machines. Some of the control characters were for communcations protocols (for example, `ACK` means "acknowledge"); others controlled the printing aspect of the machine (for example, `BS` means "backspace" and `CR` means "carriage return").

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1_	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2_	SP	!	"	#	$	%	&	‘	(	)	*	+	,	-	.	/
3_	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4_	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5_	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6_	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7_	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Hexadecimal-to-ASCII conversion table

The table at left is a definition of ASCII that provides the correspondence that you need to convert from 8-bit binary (equivalently, 2-digit hex) to a character and back. Use the first hex digit as a row index and the second hex digit as a column index to find the character that it encodes. For example, `31` encodes the digit `1`, `4A` encodes the letter `J`, and so forth. This table is for 7-bit ASCII, so the first hex digit must be `7` or less. Hex numbers starting with `0` and `1` (and the numbers `20` and `7F`) correspond to non-printing control characters such as `CR`, which now means "new line" (most of the others are rarely used in modern computing).

Unicode. In the connected world of the 21st century, it is necessary to work with many more than the 100 or so ASCII characters from the 20th century, so a new standard known as *Unicode* is emerging. By using 16 bits for most characters (and up to 24 or 32 for some characters), Unicode can support tens of thousands of characters and a broad spectrum of the world's languages. The UTF-8 encoding (from sequences of characters to sequences of 8-bit bytes and vice-versa, most characters mapping to two bytes) is rapidly emerging as a standard. The rules are complicated, but comprehensive, and fully implemented in most modern systems (such as Java) so programmers generally need not worry much about the details. ASCII survives within Unicode: the first 128 characters of Unicode are ASCII.



ASCII-binary conversion examples

We generally pack as much information as possible in a computer word, so it is possible to encode two ASCII characters in 16 bits (as shown in the example at right), four characters in 32 bits, eight characters in 64 bits, and so forth. In high-level languages such as Java, such details and UTF-8 encoding and decoding are implemented in the `String` data type, which we have been using throughout the book. Still, it is often important for Java programmers to understand some basic facts about the underlying representation, as it can certainly affect the resource requirements of programs. For example, many programmers discovered that the memory usage of their programs suddenly doubled when Java switched from ASCII to Unicode in the 2000s, and began using a 16-bit `char` to encode each ASCII character.

Summary Generally, it is wise to write programs that function properly independent of the data representation. Many programming languages fully support this point of view. But it can stand in direct opposition to the idea of taking full advantage of the capability of a computer, by using its hardware the way it was designed to be used. Java's primitive types are intended to support this point of view. For example, if the computer has hardware to add or multiply 64-bit integers, then, if we have a huge number of such operations to perform, we would like each add or multiply to reduce to a single instruction so that our program can run as fast as possible. For this reason, it is wise for the programmer to try to match data types having performance-critical operations with the primitive types that are implemented in the computer hardware. Achieving the actual match might involve deeper understanding of your system and its software, but striving for optimal performance is a worthwhile endeavor.

You have been writing programs that compute with various types of data. Our message in this section is that since every sequence of bits can be interpreted in many different ways, the meaning of any given sequence of bits within a computer depends on the context. You can write programs to interpret bits any way that you want. You cannot tell from the bits alone what type of data they represent, or even whether they represent data at all, as you will see.

To further emphasize this point, the table below gives several different 16-bit strings along with their values if interpreted as hex integers, unsigned integers, two's complement integers, binary16 floating point numbers, and pairs of characters. This are but a few early examples of the myriad available ways of representing information within a computer.

binary	*hex*	*unsigned*	*2's comp*	*floating point*	*ASCII chars*
0001001000110100	1234	4,660	4,660	0.00302886962890625	*DC2* 4
1111111111111111	FFFF	65,535	– 1	– 131008.0	*DEL* *DEL*
1111101011001110	FACE	64,206	– 1,330	– 55744.0	e N
0101011001000100	5644	22,052	22,052	100.25	V D
1000000000000001	8001	32,769	– 32,767	– .00012218952178955078	*NUL* *SOH*
0101000001000011	5043	20,547	20,547	34.09375	P C
0001110010101011	1CAB	7,339	7,339	0.0182342529296875	*FS* +

Five ways to interpret various 16-bit values

Q&A

Q. How do I find out the word size of my computer?

A. You need to find out the name of its processor, then look for the specifications of that processor. Most likely, you have a 64-bit processor. If not, it may be time to get a new computer!

Q. Why does Java use 32 bits for `int` values when most computers have 64-bit words?

A. That was a design decision made a long time ago. Java is unusual in that it completely specifies the representation of an `int`. The advantage of doing so is that old Java programs are more likely to work on new computers than in languages where machines might use different representations. The disadvantage is that 32 bits is often not enough. For example, in 2014 Google had to change from a 32-bit representation for view count after it became clear that the video *Gangham Style* would be watched more than 2,147,483,647 times. In Java, you can switch to `long`.

Q. This seems like something that could be taken care of by the system, right?

A. Some languages, for example Python, place no limit on the size of integers, leaving it to the system to use multiple words for integer values when necessary. In Java, you can use the `BigInteger` class.

Q. What's the `BigInteger` class?

A. It allows you to compute with integers without worrying about overflow. For example, if you import `java.math.BigInteger`, then the code

```
BigInteger x = new BigInteger("2");
StdOut.println(x.pow(100));
```

prints `1267650600228229401496703205376`, the value of 2^{100}. You can think of a `BigInteger` as a string (the internal representation is more efficient than that), and the class provides methods for standard arithmetic operations and many other operation. For example, this method is useful in cryptography, where arithmetic operations on numbers with hundreds of digits play a critical role in some systems. The implementation works with many digits as necessary, so overflow is not a concern. Of course, operations are much more expensive than built-in `long` or `int` operations, so Java programmers do not use `BigInteger` for integers that fit in the range supported by `long` or `int`.

Q. Why hexadecimal? Aren't there other bases that would do the job?

A. Base 8, or *octal*, was widely used for early computer systems with 12-bit, 24-bit, or 36-bit words, because the contents of a word could be expressed with 4, 8, or 12 octal digits, respectively. An advantage over hex in such systems was that only the familiar decimal digits 0-7 were needed, so that primitive I/O devices like numeric keypads could be used both for decimal numbers and octal numbers. But octal is not convenient for 32-bit and 64-bit word sizes, because those word sizes are not divisible by 3. (They are not divisible by 5 or 6 either, so no switch to a larger base is likely.)

Q. How can I guard against overflow?

A. It is not so easy, as a different check is needed for each operation. For example, if you know that `x` and `y` are both positive and you want to compute `x + y`, you could check that `x < Integer.MAX_VALUE - y`.

A. Another approach is to "upcast" to a type with a bigger range. For example, if you are calculating with in values, you could convert them to `long` values, then convert the result back to `int` (if it is not too big).

Q. How might hardware detect overflow for two's complement addition?

A. The rule is simple, though it is a bit tricky to prove: check the values of the carry *in* to the leftmost bit position and the carry *out* of the leading bit position. Overflow is indicated if they are different (see the examples at right).

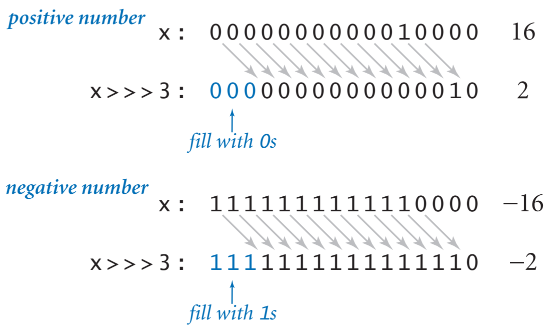
```
carry out
different
from carry in
 ∧
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  1111111111111000        -8
  1000000000000100    -32764
  1111111111111100        -4 ✗

carry out
different
from carry in
 ∧
0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
  0111111111111000     32760
  0000000000001000     +   8
  1000000000000000    -32768 ✗
```

Overflow (16-bit two's complement)

Q. What happens when we shift right a negative number?

A. Use an *arithmetic shift*, where the vacated positions are filled with the sign bit. This means that shifting right by 1 is the same as integer division by 2 for all two's complement numbers. You can use the operator >>> to invoke this operation in Java source. For example, the value of `(-16)>>>1` is `-2`, as illustrated at right. To test your understanding of this operator, figure out the values of `(-3)>>>1` and `(-1)>>>1`.



Arithmetic shift (16-bit two's complement)

Q. I never really understood the examples in the Q&A in Section 1.2 that claim that `(0.1 + 0.1 == 0.2)` is `true` but `(0.1 + 0.1 + 0.1 == 0.3)` is `false`. Can you elaborate, now?

A. A literal like `.1` or `.3` in Java source code is converted to the nearest 64-bit IEEE-754 number, a Java `double` value. Here are the values for the literals `.1`, `.2`, and `.3`:

literal	*nearest 64-bit IEEE 754 number*
`.1`	`0.1000000000000000055511151231257827021181583404541015625`
`.2`	`0.2000000000000000111022302462515654042363166809082031250`
`,3`	`0.2999999999999999888977697537484345957636833190917968750`

As you can see from the table, `.1 + .1` is equal to `.2`, but `.1 + .1 + .1` (which is equal to `.1 + .2`) is greater than `.3`. The situation is not so different from noticing that `2/5 + 2/5` is equal to `4/5`, but `2/5 + 2/5 + 2/5` is not equal to `6/5`.

Q. `System.out.println(.1)` prints `.1`, not the value in the above table. Why?

A. Few programmers need that much precision, so `println()` truncates for readability. You can use `printf()` for more precise control over the format, and the class `BigDecimal` for extended precision.

Exercises

6.1.1 Convert the decimal number `92` to binary.

Answer: `1011100`.

6.1.2 Convert the octal number `31415` to binary.

Answer: `011001100001101`.

6.1.3 Convert the octal number `314159` to decimal.

Answer: *That is not an octal number! You can do the computation, even with* `Convert`, *to get the result* `104561`, *but* `9` *is just not a legal octal digit. The version of* `Convert` *on the booksite includes such legality checks (see also* Exercise 5.1.12*). It is not unusual for a teacher to try this trick on a test, so beware!*

6.1.4 Convert the hexadecimal number `BB23A` to octal.

Answer: *First convert to binary* `1011 1011 0010 0011 1010`, *then consider the bits three at a time* `10 111 011 001 000 111 010`, *and convert to octal* `2731072`.

6.1.5 Add the two hexadecimal numbers `23AC` and `4B80` and give the result in hexadecimal. *Hint*: add directly in hex instead of converting to decimal, adding, and converting back.

6.1.6 Assume that m and n are positive integers. How many 1 bits are there in the binary representation of 2^{m+n}?

6.1.7 What is the only decimal integer whose hexadecimal representation has its digits reversed?

Answer: `53` *is* `35` *in hex.*

6.1.8 Prove that converting a hexadecimal number one digit at a time to binary and vice versa always gives the correct result.

6.1.9 IPv4 is the protocol developed in the 1970s that dictates how computers on the Internet communicate. Each computer on the Internet needs it own Internet address. IPv4 uses 32 bit addresses. How many computers can the Internet handle? Is this enough for every mobile phone and every toaster to have their own?

6.1.10 IPv6 is an Internet protocol in which each computer has a 128 bit address. How many computers would the Internet be able to handle if this standard is adopted? Is this enough?

Answer: 2^{128}. *That at least enough for the short term—5000 addresses per square micrometer of the Earth's surface!*

6.1.11 Fill in the values of the expressions in this table:

expression	`~0xFF`	`0x3 & 0x5`	`0x3 | 0x5`	`0x3 ^ 0x5`	`0x1234 << 8`
value					

6.1.12 Develop an implementation of the `toInt()` method specified in the text for converting a character in the range `0-9` or `A-Z` into an `int` value between 0 and 35.

Answer:

```
public static int toInt(char c)
{
    if ((c >= '0') && (c <= '9')) return c - '0';
    return c - 'A' + 10;
}
```

6.1.13 Develop an implementation of the `toChar()` method specified in the text for converting an `int` value between 0 and 35 into a character in the range `0-9` or `A-Z`.

Answer:

```
public static char toChar(int i)
{
    if (i < 10) return (char) ('0' + i);
    return (char) ('A' + i - 10);
}
```

6.1.14 Modify `Convert` (and the answers to the previous two exercises) to use `long`, test for overflow, and check that the digits in the input string are within the range specified by the base.

Answer: *See* `Convert.java` *on the booksite.*

6.1.15 Add to `Convert` a version of the `toString()` method that takes a third argument, which specifies the length of the string to be produced. If the specified length is less than needed, return only the rightmost digits; if it is greater, fill in with leading `0` characters. For example, `toString(64206, 16, 3)` should return `"ACE"` and `toString(15, 16, 4)` should return `"000F"`. *Hint*: First call the two-argument version.

6.1.16 Compose a Java program `TwosComplement` that takes an `int` value `i` and a word size `w` from the command line and prints the `w`-bit two's complement representation of i and the hex representation of that number. Assume that `w` is a multiple of 4. For example, your program should behave as follows:

```
% java TwosComplement -1 16
1111111111111111 FFFF
% java TwosComplement 45 8
00101101 2D
% java TwosComplement -1024 32
11111111111111111111110000000000 FFFFFC00
```

6.1.17 Modify `ExtractFloat` to develop a program `ExtractDouble` that accomplishes the same task for `double` values.

6.1.18 Write a Java program `EncodeDouble` that takes a double value from the command line and encodes it as a floating-point number according to the IEEE 754 `binary32` standard

6.1.19 Fill in the blanks in this table.

binary	*floating point*
0010001000110100	
1000000000000000	
	7.09375
	1024

6.1.20 Fill in the blanks in this table.

binary	*hex*	*unsigned*	*2's comp*	*ASCII chars*
1001000110100111				
	9201			
		1,000		
			– 131	
				? ?

Creative Exercises

6.1.21 *IP addresses and IP numbers* An IP address (IPV4) is comprised of integers w, x, y, and z and is typically written as the string `w.x.y.z`. The corresponding IP number is given by $16777216w + 65536x + 256y + z$. Given an IP number N, the corresponding IP address is derived from $w = (N / 16777216)$ mod 256, $x = (N / 65536)$ mod 256, $y = (N / 256)$ mod 256, $z = N$ mod 256. Write a function that takes an IP number and returns a `String` representation of the IP address. and another function takes an IP address and returns a `int` corresponding to the IP number. For example, given 3401190660 the first function should return `202.186.13.4`.

6.1.22 *IP address.* Write a program that takes a 32 bit string as a command line argument, and prints out the corresponding IP address in dotted decimal form. That is, take the bits 8 at a time, convert each group to decimal, and separate each group with a dot. For example, the binary IP address 010100000001000000000000 00000001 should be converted to 80.16.0.1.

6.1.23 *MAC address.* Write functions to convert back-and-forth between MAC addresses and 48-bit `long` values.

6.1.24 *Base64 encoding.* `Base64` encoding is a popular method for sending binary data over the Internet. It converts arbitrary data to ASCII text, which can be emailed back between systems without problems. Write a program to read in a arbitrary binary file and encode it using `Base64`.

6.1.25 *Floating point software.* Write a class `FloatingPoint` that has three instance variables `sign`, `exponent`, and `fraction`. Implement addition and multiplication. Include `toString()` and `parseFloat()`. Support 16-, 32-, and 64-bit formats.

6.1.26 *DNA encoding.* Develop a class `DNA` that supports an efficient representation of strings that are comprised exclusively of `a`, `c`, `t`, or `g` characters. Include a constructor that converts a string to the internal represention, a `toString()` method to convert the internal representation to a string, a `charAt()` method that returns the character at the specified index, and an `indexOf()` method that takes a `String` `p` as argument and returns the first occurence of `p` in the represented string. For the internal representation, use an array of `int` values, packing 16 characters in each `int` (two bits per character).