Internal Storage Representation of Primitive Types

This web bonus is referenced in the text on page 77 (page 65 in second edition).

This web bonus examines the underlying binary representation of primitive data types. Understanding the internal formats of data gives a programmer a better awareness of processing details and important insight when converting one data type to another.

The number of bits used to represent a value determines the range of values that can be represented. One bit can represent two things. Two bits can represent four things. In general, n bits can represent $2^n$ unique things. Appendix B in the textbook examines this issue in detail.

What does the following string of bits represent?

0011101000101010

Who knows? It might represent a character, or an integer, or a floating point value, or a piece of a larger structure, or several smaller values run together. There is nothing inherent in the string of ones and zeros itself that tells you what it is supposed to represent. Information in computer memory is just an organized collection of binary digits. They only have meaning when we take a specific string of bits and interpret them in a particular way. The rest of this web bonus describes how certain primitive data types are represented in Java.

Representing Integers

As we discussed in Chapter 2 of the text, the four types of Java integers all have a fixed size. A byte is represented using 8 bits, a short uses 16 bits, an int uses 32 bits, and a long uses 64 bits. All Java numeric types are signed, which means a sign (positive or negative) is always associated with the value. The leftmost bit in the storage space for the number is called the sign bit. If the sign bit is one, then the number is negative. If it is zero, the number is positive.

All Java integer types are stored in signed two’s complement format. The sign takes up one bit. The remaining bits store the value itself and dictate the range of values that can be stored for each type. For example, a short value uses one bit for the sign and 15 bits to store the value. Therefore it can represent values ranging from $-2^{15}$ to $2^{15}-1$ or from -32768 to 32767. The storage sizes for Java integer types are pictured in Figure 1.

![Figure 1. Java integer storage](image-url)
In two’s complement format, a positive value is represented as a straightforward binary number. For example, the value 38 is represented as a short as 0000000000100110. To represent a negative number in two’s complement, we invert all of the bits in the corresponding positive binary number, then add 1. The value -38 is represented by first inverting all of the bits, yielding 111111111011001, then adding 1, yielding 111111111011010. Note that the sign bit is automatically converted when transforming a positive value into its two’s complement negative representation. To decode a negative value, invert all of the bits, then add 1.

The use of two’s complement format makes internal arithmetic processing easier. Furthermore, two’s complement format prevents the case of “negative zero” from occurring. To demonstrate, let’s try to make the value zero, represented as the 16-bit short value 0000000000000000, into a negative number using two’s complement format. First, we invert all of the bits to 1111111111111111, then add 1, yielding 1000000000000000. There are 17 bits in this new string. The leftmost bit “falls off” the left edge of the 16-bit storage space and is lost, leaving only the regular zero format.

Figure 2 shows several integer values and their internal binary format using signed two’s complement representation.

<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>Type</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>119</td>
<td>byte</td>
<td>01110111</td>
</tr>
<tr>
<td>-95</td>
<td>byte</td>
<td>10100001</td>
</tr>
<tr>
<td>7526</td>
<td>short</td>
<td>00011101011001110</td>
</tr>
<tr>
<td>-347</td>
<td>short</td>
<td>1111111010100101</td>
</tr>
</tbody>
</table>

Figure 2. Some integer values and their internal representation

Because we only have a fixed size in which to store a value, two related problems can occur. *Overflow* occurs when a value grows so large that it cannot be stored in the space allocated. Similarly, *underflow* occurs when a value grows so small that it cannot be stored in the space allocated. Overflow and underflow errors produce the wrong results. The solution to these problems is to always use types that provide enough storage space for the values you are using.

Representing Floating Point Values

A floating point number can be described with three values: the *sign*, which determines if it is a positive or negative value; the *mantissa*, which is a positive value that defines the significant digits of the number; and the *exponent*, which determines how the decimal point is shifted relative to the mantissa. A decimal number is therefore defined by the following formula:

\[ \text{sign} \times \text{mantissa} \times 10^{\text{exponent}} \]

The sign is either 1 or -1. A positive exponent shifts the decimal point to the right, and a negative exponent shifts the decimal point to the left. For example, the number 148.69 is represented with a sign of 1, a mantissa of
14869, and an exponent of -2. The number -234000 is represented with a sign of -1, a mantissa of 234, and an exponent of 3.

Java follows a specific standard for representing floating point values. This standard was established by the International Electronic and Electrical Engineering (IEEE) organization and is commonly referred to as IEEE 754. It defines a floating-point number with a sign, mantissa, and exponent, stored in binary.

Because the base is different, a binary floating-point number is defined by the following formula:

\[ \text{sign} \times \text{mantissa} \times 2^{\text{exponent}} \]

Recall from Chapter 3 that there are two floating-point data types in Java: `float` and `double`. In IEEE 754, a single sign bit is used to represent the sign (as it is with the integer types) for both a `float` and a `double`. In a `float`, 23 bits are reserved for the mantissa and 8 bits are reserved for the exponent. In a `double`, 52 bits are used to store the mantissa and 11 bits are used to store the exponent. Figure 3 shows the storage of the Java floating point types.

![Floating Point Storage](image)

**Figure 3.** Java floating point storage

Three special values can be represented in the IEEE 754 floating point format: *positive infinity*, *negative infinity*, and *not a number*. The latter is displayed as NaN when printed. Unlike integer types, both positive and negative zero can be represented in a Java floating point value, but they are considered equal when compared using the equality operator `==`.

### Representing Characters

Java characters are represented as members of the Unicode character set. A character set is simply an ordered list of specific characters. Each character corresponds to a numeric value that represents its location in the ordered list. By storing that number, we essentially store the character. Appendix C examines the Unicode character set in more detail.

A Unicode character is stored as an unsigned 16-bit integer. Because a character is unsigned, all 16 bits contribute to the value, giving characters a numeric range of 0 to $2^{16}-1$. Therefore 65,536 unique characters can be represented in the Unicode character set, although only a little over half of those are currently defined. Unicode contains many special symbols and is designed to be a truly international character set, representing languages from all over the world. A programmer will typically use only a small fraction of all possible characters.
The internal representation of a Java character is a 16-bit binary string. A straightforward conversion to the decimal number system gives the decimal numeric value associated with a particular Unicode character. Figure 4 shows a few characters from the Unicode character set and their decimal and binary values.

<table>
<thead>
<tr>
<th>Character</th>
<th>Decimal Value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>36</td>
<td>0000000000100100</td>
</tr>
<tr>
<td>j</td>
<td>74</td>
<td>0000000001001010</td>
</tr>
<tr>
<td>z</td>
<td>122</td>
<td>0000000001111010</td>
</tr>
</tbody>
</table>

**Figure 4.** Some Unicode characters

Because characters are stored as numbers, you can perform arithmetic operations on them. If the `char` variable `ch` currently holds the character 'A', its internal representation is the decimal value 65.

When the following line of code is executed

```java
ch = ch + 5;
```

the value of `ch` now stores the character 'F' because the decimal value of `ch` is 70. This characteristic can occasionally be helpful. For instance, if you take any uppercase letter and add 32 (the difference between 'A' and 'a'), you will get the letter’s lowercase equivalent. If you subtract 48 (the value of '0') from any digit character ('0' through '9') and treat the result as an integer, you get the value of the digit.