6.7 Argument Promotion and Casting

Another important feature of method calls is **argument promotion**—converting an *argument's value*, if possible, to the type that the method expects to receive in its corresponding *parameter*. For example, a program can call `Math` method `sqrt` with an `int` argument even though a `double` argument is expected. The statement

```
System.out.println(Math.sqrt(4));
```

correctly evaluates `Math.sqrt(4)` and prints the value `2.0`. The method declaration's parameter list causes Java to convert the `int` value `4` to the `double` value `4.0` *before* passing the value to method `sqrt`. Such conversions may lead to compilation errors if Java's **promotion rules** are not satisfied. These rules specify which conversions are allowed—that is, which ones can be performed *without losing data*. In the `sqrt` example above, an `int` is converted to a `double` without changing its value. However, converting a `double` to an `int` *truncates* the fractional part of the `double` value—thus, part of the value is lost. Converting large integer types to small integer types (for example, `long` to `int`, or `int` to `short`) may also result in changed values.

The promotion rules apply to expressions containing values of two or more primitive types and to primitive-type values passed as arguments to methods. Each value is promoted to the "highest" type in the expression. Actually, the expression uses a *temporary copy* of each value—the types of the original values remain unchanged. Figure 6.4 lists the primitive types and the types to which each can be promoted. The valid promotions for a given type are always to a type higher in the table. For example, an `int` can be promoted to the higher types `long`, `float` and `double`.

Converting values to types lower in the table of Fig. 6.4 will result in different values if the lower type cannot represent the value of the higher type (for example, the `int` value 2000000 cannot be represented as a `short`, and any floating-point number with digits after its decimal point cannot be represented in an integer type such as `long`, `int` or `short`). Therefore, in cases where information may be lost due to conversion, the Java compiler requires you to use a *cast operator* (introduced in Section 4.10) to explicitly force the conversion to occur—otherwise a compilation error occurs. This enables you to "take control" from the compiler. You essentially say, "I know this conversion might cause loss of information, but for my purposes here, that's fine." Suppose method `square` calculates the square of an integer and thus requires an `int` argument. To call `square` with a `double` argument named `doubleValue`, we would be required to write the method call as

Fig. 6.4

Promotions allowed for primitive types.

Type	Valid promotions
`double`	None
`float`	`double`
`long`	`float` or `double`
`int`	`long`, `float` or `double`
`char`	`int`, `long`, `float` or `double`
`short`	`int`, `long`, `float` or `double` (but not `char`)
`byte`	`short`, `int`, `long`, `float` or `double` (but not `char`)
`boolean`	None (`boolean` values are not considered to be numbers in Java)

```
  square((int) doubleValue)
```

This method call explicitly casts (converts) `doubleValue`'s value to a a temporary integer for use in method `square`. Thus, if `doubleValue`'s value is `4.5`, the method receives the value `4` and returns `16`, not `20.25`.

Common Programming Error 6.7

Casting a primitive-type value to another primitive type may change the value if the new type is not a valid promotion. For example, casting a floating-point value to an integer value may introduce truncation errors (loss of the fractional part) into the result.