The assignment statement

The assignment statement is used to store a value in a variable. As in most programming languages these days, the assignment statement has the form:

\[ \text{<variable>} = \text{<expression>;} \]

For example, once we have an \textbf{int} variable \textit{j}, we can assign it the value of expression \(4 + 6\):

\begin{verbatim}
int j;
j = 4+6;
\end{verbatim}

As a convention, we always place a blank after the = sign but \textit{not} before it. You don’t have to follow this convention. Our reasons for it are explained on p. 27 of Gries/Gries.

Once we have variables with values, we can use those variables in expressions. For example, we can evaluate the expression consisting simply of \(j\), or the expression \(2\ast j\), and we can store the value of expression \(j+1\) in another variable \(k\):

\begin{verbatim}
j
2*j
int k;
k = j + 1;
k
\end{verbatim}

You must memorize how the assignment statement is executed, or carried out. If asked, you should say:

Evaluate the \textit{<expression>} and store its value in the \textit{<variable>}.

Please memorize this definition of how to execute the assignment statement. In order to be sure that you understand it, we execute a series of assignments, showing how the variables change.

Here’s variables \(j\) and \(k\), with the values computed by what we have done so far. We now execute a sequence of three assignments. Add 2 to \(j\) and store the result in \(j\), subtract \(k\) from \(j\) and store the result in \(k\), and store 0 in \(j\).

\begin{verbatim}
j = j + 2;
k = j - k;
j = 0;
\end{verbatim}

As we carry out the assignments, we change the values of the variables. We do \textit{not} draw the variables again. There is only one variable \(j\), and its value is changed whenever \(j\) is assigned a new value.

The initializing declaration

We can abbreviate a declaration of \textbf{c} followed by an assignment of 25 to it, using an \textit{initializing declaration}:

\begin{verbatim}
int c = 25;
\end{verbatim}

Actually, any expression may be used — the expression need not be a constant.

It is important to realize that this is simply a combination of a declaration and an assignment. Writing two such initializing declarations for the same variable will not work because only one declaration per variable is allowed.

\begin{verbatim}
int m = c+1;
int m = 45;  // illegal because m has already been declared
\end{verbatim}

The types of variable and expression must match

In a Java assignment, the types of the variable and expression must match. For example, if one is a \textbf{boolean}, the other must be also, and if one is a \textbf{String}, the other must be a \textbf{String}. This is a consequence of the strong typing principle.

For numeric types, there is a bit more leeway. You know that there are types \textbf{byte}, \textbf{short}, \textbf{int}, and \textbf{long} which have increasingly larger sets of values, and there are two floating point, or real-number, types, \textbf{float} and \textbf{double}. These move from the so-called narrowest type \textbf{byte} to the widest type, \textbf{double}. 
The assignment statement

The rule for an assignment of an expression that is a number is that the type of the variable has to be at least as wide as the type of the expression.

For example, if we have a byte variable b and an int variable i, both of which contain 0, it is legal to assign b to i but illegal to assign i to b.

```java
byte b = 0;
int i = 0;
i = b;   // illegal
b = i;   // illegal
```

The reason for the rule should be clear. Assigning a wider-type value to a narrower-type variable may lose information or result in overflow of some sort. For example, how could 6000 be stored in a byte variable?

You might think that Java would allow an assignment of an int to a byte but would complain at runtime if the int value were too big. However, this would violate the strong typing principle, as designed in Java.