Compound assignment operators

The compound assignment operators are:

```
+=      -=      *=      /=      %=
```

Consider variable k:

```
int k= 5;
```

Each of the following two lines contains equivalent assignment statements, so it looks like the compound assignment operators simply provide syntactic sugar:

```
k += 20;        k= k + 20;        // both add 20 to k
k *= 20;        k= k * 20;        // both store in k the value k * 20.
```

The compound operators are different in two ways, which we see by looking more precisely at their definition. The Java language specification says that:

> The compound assignment E1 op= E2 is equivalent to [i.e. is syntactic sugar for]
>
> E1 = (T) ((E1) op (E2))
>
> where T is the type of E1, except that E1 is evaluated only once.

We note two important points:

(1) The expression is cast to the type of E1 before the assignment is made (the cast is in red above)

(2) E1 is evaluated once.

Investigation of the effect of (1)

For type int it doesn't matter because primitive operations of type int always produce an int. But consider type byte. A value of type byte lies in the range -128..127. The first line below is syntactically incorrect because the type of expression b1+1 is int, which is wider than byte. The second line is okay because the second statement is equivalent to b2= (byte) (b2 + 1). It changes b2 to -128 because of wraparound.

```
byte b1= 127;  b1= b1 + 1;      // syntactically incorrect: b1+1 has type int
byte b2;   b2 += 1;
```

Since char is a number type, you can even do this:

```
char c= 'a';   c += 1;              // after execution, c contains the character 'b'.
```

Investigation of the effect of (2)

Let d be an int-array and f(int) some function. In the first assignment below, f is called twice. In the second assignment, it is called only once. So the use of the compound operator is more efficient.

```
d[f(5)]= d[f(5)] + 1;
d[f(5)] += 1;
```

There is even more of a difference between using += instead of = if one considers side effects. Evaluation of d[k++] increments k after the value of d[k] is retrieved. So execution of the first line below stores d[1] + 1 in d[0] and adds 2 to k. Execution of the second line increments d[h] and h.

```
int k= 0;  d[k++]= d[k++] + 1;
int h= 0; d[h] += 1;                  (second statement equivalent to d[h]= d[h++] + 1;)
```

We don't approve of such side effects, for they are confusing. Theories of program correctness indicate that proof systems for programs that allow such side effects are far more complicated than those that don't.