

Evaluating a simple expression

Parameter *s* of function *eval* contains an expression involving unsigned integers and operators + and -. There is at least one integer, followed by zero or more operator-operand pairs. There may be any number of blanks at the beginning and end of *s* and between operators and integers.

Function *eval* will evaluate the expression in *s* and return the value. Here are examples:

```
eval("3")           returns 3
eval("3 + 4")       returns 7
eval("100 - 25+50") returns 125
eval("9")           returns 9
eval(" 7 + 7 +7 ") returns 21
```

The first point to note is that there may be spaces all over the place. If we removed them, then following each integer (except the last) would be an operator + or -, and following each operator would be a digit. That could simplify our task, so we start off with a statement to remove all blanks.

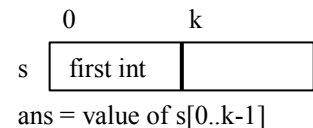
```
s = s.replace(" ", "");
```

Now, the tendency is to immediately write a loop to process parameter *s*, looking at one character at a time:

```
for (int k=0; k < s.length(); k=k+1) { Process s[k] }
```

This sudden introduction of a loop leads to all sorts of complications. Before introducing such a loop, think more about the situation.

Note that there is always a first operand, and that may be all there is, so we think of writing a statement-comment. It uses two new local variables. Variable *ans* now contains the value of the first integer, and *k* contains the index of the character following the first integer, as shown in the diagram to the right.



```
// Store the first integer in s into local variable ans and set k to the character following it.
```

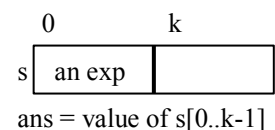
Now, the rest of *s* to process, *s*[*k*..], contains a sequence of zero or more operator-operand pairs, like

```
+234 -6 -6343 +12
```

Therefore, we think of writing a loop that, at each iteration, processes one operator-operand pair. This is the important step of having the structure of the loop reflect the structure of the data.

How is an operator-operand pair to be processed? Its contribution to the answer can be accumulated in variable *ans*. And *k* can be set to the position following the pair. Note how the obvious loop invariant, shown to the right below, is a simple generalization of the precondition given above. Instead of the first segment containing the first int, it contains the part of the expression processed thus far.

```
// invariant: s[0..k-1] is a complete expression, ans is its value, and
//           if s[k] exists, it starts an unsigned integer
while (k < s.length()) {
    // process the operator-operand pair that starts at s[k]:
    // Add its value to ans and change k to the index after that pair
}
```



Note that we avoided figuring out how to implement the two statement-comments, preferring to get the overall structure down first.

After the loop, of course, *ans* must be returned.

```
/** Return the value of the expression in s.
 * Precondition: s consists of a sequence of
 * at least one unsigned integer separated by
 * operators '+' or '-'. There may be space
 * characters at the beginning and end of the
 * expression and between the numbers and
 * operators. */
public static int eval(String s)
```

Evaluating a simple expression

The first statement-comment requires finding the end of the integer that starts at $s[0]$. The second one requires finding the end of the integer that starts at $s[k+1]$ —after the operator. The same task must be done in two places. So, we use the mañana principle and write a function header and specification.

```
/** s[h] is a digit. Return the integer t such that
 * s[h..t-1] is a list of digits but s[h..t] isn't. */
public static int getEnd(String s, int h)
```

It is given the *String* s and an index h . Character $s[h]$ is a digit. And it should return the index *following* the last digit of the integer that begins in $s[h]$.

We implement the first statement-comment. Call *getEnd* to find the index following the integer and store it in k . Then use function *parseInt* to extract the integer in $s[0..k-1]$ and store its value in ans .

```
/** Return the value of the expression in s. ... */
public static int eval(String s) {
    s = s.replace(" ", "");

    // Store the first integer into local variable ans,
    // set local variable k to index following first integer
    int k = getEnd(s, 0);
    int ans = Integer.parseInt(s.substring(0, k));

    while (k < s.length()) {
        // Process operator-operand pair that starts at s[k]:
        // Add value to ans, change k to index after that pair
        int h = k; // index of start of operator-operand pair
        k = getEnd(s, k+1);
        int n = Integer.parseInt(s.substring(h+1, k));
        ans = s.charAt(h) == "+" ? ans+n : ans-n;
    }
    return ans;
}
```

We implement the second statement-comment. First, save the index of the start of the operator-operand pair in h . Then call *getEnd* to find the index following the integer and store it in k . Then use function *parseInt* to extract the integer in $s[h..k-1]$ and store its value in n . Finally, use a conditional expression based on operator $s[h]$ to add n to ans or subtract n from ans .

This completes the development of function *eval*.

It remains to implement function *getEnd*! It has a loop that checks successive characters until either the end of the string or a non-digit is reached.

```
/** s[h] is a digit. Return the integer t such that
 * s[h..t-1] is a list of digits but s[h..t] isn't. */
public static int getEnd(String s, int h) {
    int t = h+1;
    while (t < s.length() &&
           Character.isDigit(s.charAt(t))) {
        t = t + 1;
    }
    return t;
}
```