out Blocks and Expression-based Contracts

This lesson explains the use of out blocks for postconditions and the expression-based contract. Furthermore, it teaches how to disable contract programming in D.

WE'LL COVER THE FOLLOWING

- out blocks for postconditions
- Expression-based contracts
- Disabling contract programming

out blocks for postconditions

This contract involves guarantees that the function provides. Such guarantees are called the **function's postconditions**. An example of a function with a postcondition would be a function that returns the number of days in February: The function can guarantee that the returned value would always be either 28 or 29.

> The postconditions are checked inside the `out` blocks of functions.

Because the value that a function returns by need not be defined as a variable inside the function, there is usually no name to refer to the return value. This can be seen as a problem because the `assert` checks inside the `out` block cannot refer to the returned variable by name.
D solves this problem by providing a way of naming the return value right after the `out` keyword. That name represents the very value that the function is in the process of returning:

```
int daysInFebruary(int year)
out (result) {
    assert((result == 28) || (result == 29));
```

```
} do {

    return isLeapYear(year) ? 29 : 28;
}
```

Although result is a reasonable name for the returned value, other valid names may also be used.
Some functions do not have return values or the return value need not be checked. In that case, the out block does not specify a name:

```
out {
    // ...
}
```

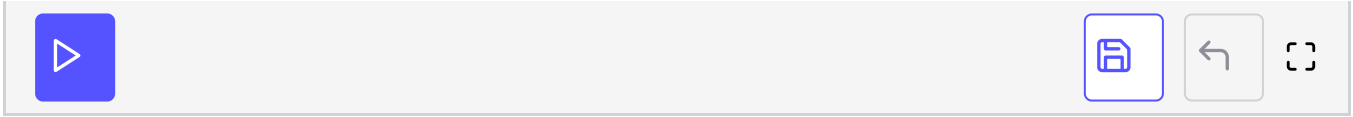
Similar to `in` blocks, the out blocks are executed automatically after the body of the function is executed.
An `assert` check that fails inside the `out` block indicates that the contract has been violated by the function.
As it has been obvious, `in` and `out` blocks are optional. Considering unittest blocks as well (which are also optional), D functions may consist of up to four blocks of code:

- `in`: optional
- `out`: optional
- `do`: mandatory, but the `do` keyword may be skipped if no `in` or `out` block is defined
- `unittest`: optional and technically not a part of a function's definition but commonly defined right after the function

Here is an example that uses all of these blocks:

```
import std.stdio;

/* Distributes the sum between two variables.
 *
 * Distributes to the first variable first, but never gives
 * more than 7 to it. The rest of the sum is distributed to
 * the second variable. */
void distribute(int sum, out int first, out int second)
in {
    assert(sum >= 0, "sum cannot be negative");
```

```
} out {
    assert(sum == (first + second));


} do {
    first = (sum >= 7) ? 7 : sum;
    second = sum - first;
}

unittest {
    int first;
    int second;

    // Both must be 0 if the sum is 0
    distribute(0, first, second);
    assert(first == 0);
    assert(second == 0);

    // If the sum is less than 7, then all of it must be given
    // to first
    distribute(3, first, second);
    assert(first == 3);
    assert(second == 0);

    // Testing a boundary condition
    distribute(7, first, second);
    assert(first == 7);
    assert(second == 0);

    // If the sum is more than 7, then the first must get 7
    // and the rest must be given to second
    distribute(8, first, second);
    assert(first == 7);
    assert(second == 1);

    // A random large value
    distribute(1_000_007, first, second);
    assert(first == 7);
    assert(second == 1_000_000);
}

void main() {
    int first;
    int second;

    distribute(123, first, second);
    writeln("first: ", first, " second: ", second);
}
```



Code with in, out, do, and unittest blocks

Although the actual work of the function consists of only two lines, there are a total of 19 nontrivial lines that support its functionality. It may be argued that so much extra code is too much for such a short function. However, bugs are

never intentional. The programmer always writes code that is expected to work correctly, which commonly ends up containing various types of bugs.

When expectations are laid out explicitly by unit tests and contracts, functions that are initially correct have a greater chance of staying correct. It is recommended that you take full advantage of any feature that improves program correctness. Both unit tests and contracts are effective tools for that goal. They help reduce the time spent on debugging, effectively increasing time spent on actually writing code.

Expression-based contracts

Although `in` and `out` blocks are useful for allowing any D code, precondition and postcondition checks are usually not more than simple `assert` expressions. As a convenience in such cases, there is a shorter expression-based contract syntax. Let's consider the following function:

```
int func(int a, int b)
in {
    assert(a >= 7, "a cannot be less than 7");
    assert(b < 10);

} out (result) {
    assert(result > 1000);

} do {
    // ...
}
```

The expression-based contract obviates curly brackets, explicit `assert` calls and the `do` keyword:

```
int func(int a, int b)
in (a >= 7, "a cannot be less than 7") in (b < 10)
out (result; result > 1000) {
    // ...
}
```

Note how the return value of the function is named before a semicolon in the `out` contract. When there is no return value or when the `out` contract does not refer to the return value, the semicolon must still be present:

```
out (; /* ... */)
```

```
out (; /* ... */)
```

Disabling contract programming

Contrary to unit testing, contract programming features are enabled by default. The `-release` compiler switch disables contract programming:

```
$ dmd deneme.d -w -release
```

When the program is compiled with the `-release` switch, the contents of `in`, `out` and invariant blocks are ignored.

In the next lesson, you will learn the use of `enforce` and `assert` with `in` blocks.