in Blocks for Preconditions

This lesson defines contract programming and then explains the working of in blocks in contract programming.

Contract programming

Contract programming is a software design approach that treats parts of software as individual entities that provide services to each other. This approach realizes that software can work according to its specifications as long as the *provider* and the *consumer* of the service both obey a contract.

D's contract programming features involve functions as units of software services. Similar to unit testing, contract programming is also based on `assert` checks.

Contract programming in D is implemented by three types of code blocks:

- Function `in` blocks
- Function `out` blocks
- Struct and class invariant blocks

`in` blocks

The correct execution of functions usually depends on whether the values of their parameters are valid. For example, a square root function may require that its parameter cannot be negative. A function that deals with dates may require that the number of the month must be between 1 and 12. Such requirements of a function are called its **preconditions**.

We have already seen such condition checks in the assert and enforce chapter. Conditions on parameter values can be enforced by `assert` checks within function definitions:

```
import std.stdio;
import std.format;

string timeToString(int hour, int minute) {
    assert((hour >= 0) && (hour <= 23));
    assert((minute >= 0) && (minute <= 59));

    return format("%02s:%02s", hour, minute);
}
void main() {
    writeln(timeToString(12,34));
}
```

In contract programming, the same checks are written inside the `in` blocks of functions. When an `in` or `out` block is used, the actual body of the function must be specified as a `do` block:

```
import std.stdio;
import std.string;

string timeToString(int hour, int minute)
in {
    assert((hour >= 0) && (hour <= 23));
    assert((minute >= 0) && (minute <= 59));

} do {
    return format("%02s:%02s", hour, minute);
}

void main() {
    writeln(timeToString(12, 34));
}
```

Use of in and do blocks

Note: In earlier versions of D, the `body` keyword was used for this purpose instead of `do`.

A benefit of an `in` block is that all of the preconditions can be kept together and separate from the actual body of the function. This way, the function body would be free of `assert` checks about the preconditions. As needed, it is still possible and advisable to have other `assert` checks inside the function body as unrelated checks that guard against potential programming errors in the function body.

The code that is inside the `in` block is executed automatically every time the function is called. The actual execution of the function starts only if all of the `assert` checks inside the `in` block pass. This prevents executing the function with invalid preconditions and as a consequence, avoids producing incorrect results.

> An `assert` check that fails inside the `in` block indicates that the contract has been violated by the caller.

In the next lesson, you will learn `out` blocks for postcondition.