in Blocks Versus enforce Checks

This lesson teaches you when to use assert checks with in-blocks versus enforce checks inside function bodies.

Use `in` blocks or `enforce`?

We have seen in the assert and enforce chapter that sometimes it is difficult to decide whether to use `assert` or `enforce` checks. Similarly, sometimes it is difficult to decide whether to use `assert` checks within `in` blocks versus `enforce` checks within function bodies.

The fact that it is possible to disable contract programming is an indication that contract programming is for protecting against programmer errors. For that reason, the decision here should be based on the same guidelines that we saw in the `assert` and `enforce` chapter:

- If the check is guarding against a coding error, then it should be in the `in` block. For example, if the function is called only from other parts of the program, likely to help with achieving a functionality of it, then the parameter values are entirely the responsibility of the programmer. For that reason, the preconditions of such a function should be checked in its `in` block.

- If the function cannot achieve some task for any other reason, including invalid parameter values, then it must throw an exception, conveniently by `enforce`.

Example

To see an example of this, let's define a function that returns a slice of a

To see an example of this, let's define a function that returns a slice of a certain width from the middle of another slice. Let's assume that this function

is for the consumption of the users of the module, as opposed to being an internal function used by the module itself. Since the users of this module can call this function by various and potentially invalid parameter values, it would be appropriate to check the parameter values every time the function is called. It would be insufficient to only check them at program development time, after which contracts can be disabled by -release.

Another criterion to consider when deciding between `in` blocks versus `enforce` is to consider whether the condition is recoverable. If it is recoverable by the higher layers of code, then it may be more appropriate to throw an exception, conveniently by `enforce`.

The following function validates its parameters by calling `enforce` in the function body instead of an `assert` check in the `in` block:

```d
import std.exception;

inout(int)[] middle(inout(int)[] originalSlice, size_t width)
out (result) {
    assert(result.length == width);

} do {
    enforce(originalSlice.length >= width);

    immutable start = (originalSlice.length - width) / 2;
    immutable end = start + width;

    return originalSlice[start .. end];
}

unittest {
    auto slice = [1, 2, 3, 4, 5];

    assert(middle(slice, 3) == [2, 3, 4]);
    assert(middle(slice, 2) == [2, 3]);
    assert(middle(slice, 5) == slice);
}

void main() {
}
```

The function that returns a slice of the middle of another slice.

The `out` blocks do not face this problem. Since the return value of every function is the responsibility of the programmer, postconditions must always be checked in the `out` blocks. The function above follows this guideline.

In the next lesson, you will find a coding challenge to test the concepts covered in this chapter.