Solution: Make the Unit Test Pass

This lesson provides a solution to the challenge given in the previous lesson.

Solution

For demonstration purposes, let's write an obviously incorrect implementation that passes the first test by accident. The following function simply returns a copy of the input:

```
dstring toFront(dstring str, dchar letter) {
    dstring result;

    foreach (c; str) {
        result ~= c;
    }

    return result;
}

unittest {
    immutable str = "hello"d;

    assert(toFront(str, 'h') == "hello");
    assert(toFront(str, 'o') == "ohell");
    assert(toFront(str, 'l') == "llheo");
}

void main() {
}
```

The first test passes but the second one fails

Here is a correct implementation that passes all of the tests:

```d
dstring toFront(dstring str, dchar letter) {
    dchar[] firstPart;

    dchar[] lastPart;

    foreach (c; str) {
        if (c == letter) {
            firstPart ~= c;

        } else {
            lastPart ~= c;
        }
    }

    return (firstPart ~ lastPart).idup;
}

unittest {
    immutable str = "hello"d;

    assert(toFront(str, 'h') == "hello");
    assert(toFront(str, 'o') == "ohell");
    assert(toFront(str, 'l') == "llheo");
}

void main() {
}
```

The tests finally pass

This function can now be modified in different ways under the confidence that its tests will have to pass. The following two implementations are very different from the first one but they too are correct according to the tests.

- An implementation that takes advantage of `std.algorithm.partition`:

```d
import std.algorithm;

dstring toFront(dstring str, dchar letter) {
    dchar[] result = str.dup;
    partition!(c => c == letter, SwapStrategy.stable)(result);

    return result.idup;
}

unittest {
    immutable str = "hello"d;

    assert(toFront(str, 'h') == "hello");
    assert(toFront(str, 'o') == "ohell");
    assert(toFront(str, 'l') == "llheo");
}
```

```
void main() {
}
```

Implementation with std.algorithm.partition

> **Note:** The `=>` syntax that appears in the program above creates a lambda function.

- The following implementation first counts how many times the special letter appears in the string. That information is then sent to a separate function named `repeated()` to produce the first part of the result. Note that `repeated()` has a set of unit tests of its own:

```
dstring repeated(size_t count, dchar letter) {
    dstring result;

    foreach (i; 0..count) {
        result ~= letter;
    }

    return result;
}

unittest {
    assert(repeated(3, 'z') == "zzz");
    assert(repeated(10, 'é') == "éééééééééé");
}

dstring toFront(dstring str, dchar letter) {
    size_t specialLetterCount;
    dstring lastPart;

    foreach (c; str) {
        if (c == letter) {
            ++specialLetterCount;

        } else {
            lastPart ~= c;
        }
    }

    return repeated(specialLetterCount, letter) ~ lastPart;
}

unittest {
    immutable str = "hello"d;

    assert(toFront(str, 'h') == "hello");
    assert(toFront(str, 'o') == "ohell");
}


void main() {
}
```

Implementation using count of the special letters

In the next lesson, you will find a quiz based on the concepts of unit testing.