Unit Testing

This lesson explains how unit testing can be used for catching bugs and how to activate the unit tests. Then, it further explains the unit test blocks in detail.

WE'LL COVER THE FOLLOWING

- Unit testing for catching bugs
- Activating the unit tests
- unittest blocks

Unit testing for catching bugs

- Since programs are written by programmers and D is a compiled language, the programmers and the compiler will always be there to discover bugs. Those two aside, the earliest and the most effective (partly because it was the earliest) way of catching bugs is unit testing.
- Unit testing is an indispensable part of modern programming. It is the most effective method of reducing coding errors. According to some development methodologies, code that is not guarded by unit tests is buggy code. Unfortunately, the opposite is not true: Unit tests do not guarantee that the code is free of bugs. Although unit tests are very effective, they can only reduce the risk of bugs.
- Unit testing also enables refactoring the code (i.e. making improvements to it) with ease and confidence. Otherwise, it is common to accidentally break some of the existing functionality of a program when adding new features to it. Bugs of this type are called **regressions**. Without unit testing, regressions are sometimes discovered as late as the quality assurance (QA) testing of future releases, or worse, by the end-users.
- Risk of regressions discourages programmers from refactoring the code, sometimes preventing them from performing the simplest of improvements like correcting the name of a variable. This, in turn, causes *code rot*, a condition where the code becomes more and more

unmaintainable. For example, although some lines of code would better

be moved to a newly defined function in order to be called from more than one place, fear of regressions make programmers copy and paste the existing lines to other places instead, leading to the problem of code duplication. Phrases like “if it isn’t broken, don’t fix it” are related to fear of regressions. Although they seem to be conveying wisdom, such guidelines cause the code to rot slowly and become an untouchable mess.

- Modern programming rejects such “wisdom.” To prevent it from becoming a source of bugs, the code is supposed to be “refactored mercilessly.” The most powerful tool of this modern approach is **unit testing**.
- Unit testing involves testing the smallest units of code independently. When units of code are tested independently, it is less likely that there are bugs in higher-level code that use those units. When the parts work correctly, it is more likely that the whole will work correctly as well.
- Unit tests are provided as library solutions in other languages (e.g. JUnit, CppUnit, Unittest++, etc.). In D, unit testing is a core feature of the language. It is debatable whether or not a library solution is better than a language feature for unit testing. Because D does not provide some of the features that are commonly found in unit testing libraries, it may be worthwhile to consider library solutions as well.
- The unit testing features of D are as simple as inserting `assert` checks into unit test blocks.

Activating the unit tests

Unit tests are not a part of the actual execution of the program. They should be activated only during program development when explicitly requested.

The **dmd** compiler switch that activates unit tests is `-unittest`.

Assuming that the program is written in a single source file named deneme.d, its unit tests can be activated by the following command:

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

When a program that is built by the `-unittest` switch is started, its unit test

blocks are executed first. Only if all of the unit tests pass then the program execution continues with `main()`.

unittest blocks

The lines of code that involve unit tests are written inside `unittest` blocks. These blocks do not have any significance for the program other than containing the unit tests:

```
unittest {
    /* ... the tests and the code that support them ... */
}
```

Although unittest blocks can appear anywhere, it is convenient to define them right after the code that they test.
As an example, let's test a function that returns the ordinal form of a specified number as in "1st", "2nd", etc. A unittest block of this function can simply contain `assert` statements that compare the return values of the function to the expected values. The following function is being tested with the four distinct expected outcomes of the function:

```
string ordinal(size_t number) {
    // ...
}
unittest {
    assert(ordinal(1) == "1st");
    assert(ordinal(2) == "2nd");
    assert(ordinal(3) == "3rd");
    assert(ordinal(10) == "10th");
}
```

The four tests above test that the function works correctly at least for the values of 1, 2, 3 and 10 by making four separate calls to the function and comparing the returned values to the expected ones.
Although unit tests are based on `assert` checks, unittest blocks can contain any D code. This allows for preparations before actually starting the tests or any other supporting code that the tests may need. For example, the following block first defines a variable to reduce code duplication:

```
dstring toFront(dstring str, dchar letter) {
    // ...
}
```

```
unittest {
    immutable str = "hello"d;

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

The three `assert` checks above test that `toFront()` works according to its specification.
As these examples show, unit tests are also helpful in demonstrating how particular functions should be called. Usually, it is easy to get an idea of what a function does just by reading its unit tests.

The next lesson explains unit testing for exceptions.