Types of assert

This lesson explains why assert statements are needed and then explores different types of assert statements in D.

WE'LL COVER THE FOLLOWING

Need for `assert`

In the previous chapter, we have seen how exceptions and scope statements are used toward program correctness. `assert` is another powerful tool to achieve the same goal. `assert` ensures that the certain assumptions that the program is based on are valid.

It may sometimes be difficult to decide whether to throw an exception or to call `assert`. I will use `assert` in all of the examples below without much justification. We will look at the differences later in the chapter.

Although not always obvious, programs are full of assumptions. For example, the following function is written under the assumption that both age parameters are greater than or equal to zero:

```
import std.stdio;

double averageAge(double first, double second) {
    return (first + second) / 2;
}

void main() {
    auto result = averageAge(-1, 10);

    writeln(result);
}
```

Although it may be invalid for the program to ever have an age value that is negative, the function would still produce an average, which may be used in the program unnoticed, resulting in the program continuing with incorrect data.
As another example, the following function assumes that it will always be called with two commands: “sing” or “dance”:

```
import std.stdio;

void applyCommand(string command) {
    if (command == "sing") {
        robotSing();

    } else {
        robotDance();
    }
}

void robotSing() {

    writeln("robot singing");
}

void robotDance() {

     writeln("robot dancing");
}


void main() {
    string command = "teach";

    applyCommand(command);
}
```

Because of that assumption, the `robotDance()` function would be called for every command other than “sing,” valid or invalid.
When such assumptions are kept only in the programmer’s mind, the program may end up working incorrectly. *assert* statements check assumptions and terminate programs immediately when they are not valid.

`assert`

`assert` can be used in two ways:

```
assert(logical_expression);
assert(logical_expression, message);
```

The logical expression represents an assumption about the program. `assert` evaluates that expression to validate that assumption. If the value of the logical expression is true then the assumption is considered to be valid. Otherwise, the assumption is invalid and an `AssertError` is thrown.

As its name suggests, this exception is inherited from `Error`, and as you may remember from the Exceptions chapter, exceptions that are inherited from `Error` must never be caught. It is important for the program to be terminated right away instead of continuing under invalid assumptions.

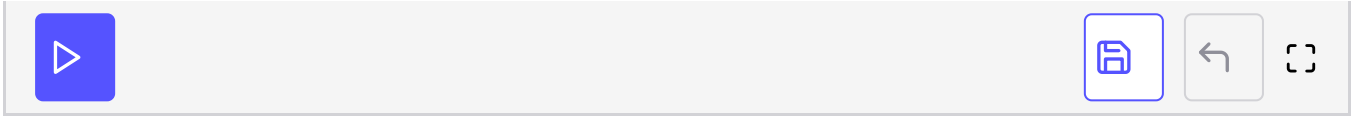
The two implicit assumptions of `averageAge()` above may be spelled out by two `assert` calls as in the following function:

```
import std.stdio;

double averageAge(double first, double second) {
    assert(first >= 0);
    assert(second >= 0);
    return (first + second) / 2;
}

void main() {
    auto result = averageAge(-1, 10);

    writeln(result);
}
```



Those `assert` checks carry the meaning “assuming that both of the ages are greater than or equal to zero.” It can also be thought of as meaning “this function can work correctly only if both of the ages are greater than or equal to zero.”

Each `assert` checks its assumption and terminates the program with an `AssertError` when it is not valid:

```
core.exception.AssertError@deneme(2): Assertion failure
```

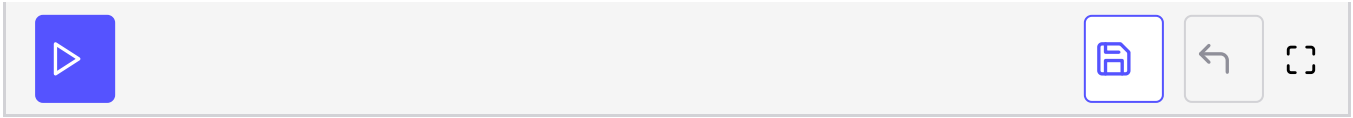
The part after the `@` character in the message indicates the source file and the line number of the assert check that failed. According to the output above, the assert that failed is on line 2 of file deneme.d.
The other syntax of assert allows printing a custom message when the assert check fails:

```
import std.stdio;

double averageAge(double first, double second) {
    assert(first >= 0, "Age cannot be negative.");
    assert(second >= 0, "Age cannot be negative.");
    return (first + second) / 2;
}

void main() {
    auto result = averageAge(-1, 10);

    writeln(result);
}
```



Sometimes it is thought to be impossible for the program to ever enter a code path. In such cases, it is common to use the literal `false` as the logical expression to fail an `assert` check. For example, to indicate that `applyCommand()` function is never expected to be called with a command other than “sing” and “dance” and to guard against such a possibility, an `assert(false)` can be inserted into the impossible branch:

```
import std.stdio;

void applyCommand(string command) {
    if (command == "sing") {
        robotSing();

    } else if (command == "dance") {
        robotDance();

    } else {
        assert(false);
    }
}

void robotSing() {

    writeln("robot singing");
```

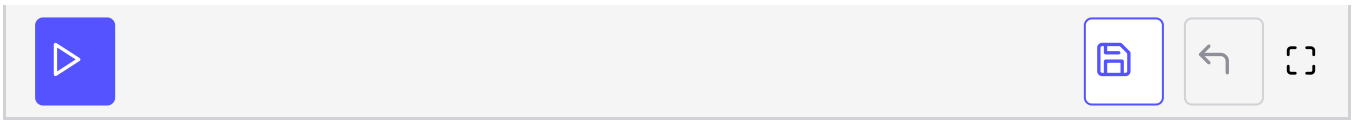
```
}

void robotDance() {


     writeln("robot dancing");
}


void main() {
    string command = "teach";

    applyCommand(command);
}
```



The function is guaranteed to work with the only two commands that it knows about.

> **Note:** An alternative choice here would be to use a final switch statement.

`static assert`

Since `assert` checks are for correct execution of programs, they are applied when the program is actually running. There can be other checks that are about the structure of the program, which can be applied even at compile time.

`static assert` is the counterpart of `assert` that is applied at compile time. The advantage of `static assert` is that it does not allow even compiling a program that would have otherwise run incorrectly. A natural requirement is that it must be possible to evaluate the logical expression at compile time. For example, assuming that the title of a menu will be printed on an output device that has limited width, the following `static assert` ensures that it will never be wider than that limit:

```
import std.stdio;

void main() {
    enum dstring menuTitle = "Command Menu";

    static assert(menuTitle.length <= 16);
}
```

Note that the `string` is defined as `enum` so that its length can be evaluated at compile time.
Let's assume that a programmer changes that title to make it more descriptive:

```
import std.stdio;

void main() {
    enum dstring menuTitle = "Directional Commands Menu";

    static assert(menuTitle.length <= 16);
}
```

The `static assert` check prevents the compiling of the program:

```
Error: static assert  (25u <= 16u) is false
```

This would remind the programmer of the limitations of the output device.
`static assert` is even more useful when used in templates.

In the next lesson, we will explore different properties of assert in detail.