Parameter Qualifiers: lazy, scope and shared

In this lesson you will learn three more parameter qualifiers which are lazy, scope, and shared.

WE'LL COVER THE FOLLOWING

- `lazy`
- `scope`
- `shared`
- `return`

`lazy`

It is natural to expect that arguments are evaluated before entering the functions that use the arguments. For example, the function `add()` below is called with the return values of two other functions:

```
result = add(anAmount(), anotherAmount());
```

In order for `add()` to be called, first `anAmount()` and `anotherAmount()` must be called. Otherwise, the values that `add()` needs would not be available.

Evaluating arguments before calling a function is called **eager evaluation**.

However, depending on certain conditions, some parameters may not get a chance to be used in the function at all. In such cases, evaluating the arguments eagerly would be wasteful.

A classic example of this situation is a logging function that outputs a message only if the importance of the message is above a certain configuration setting:

```
enum Level { low, medium, high }

void log(Level level, string message) {
    if (level >= interestedLevel) {
```

```
        writeln(message);

    }
}
```

For example, if the user is interested only in the messages that are `Level.high`, a message with `Level.medium` would not be printed. However, the argument would still be evaluated before calling the function. For example, the entire `format()` expression below, including the `getConnectionState()` call that it makes, would be wasted if the message is never printed:

```
if (failedToConnect) {
    log(Level.medium, format("Failure. The connection state is '%s'.", getConnectionState()));
}
```

The `lazy` keyword specifies that an expression that is passed as a parameter will be evaluated only if and when needed:

```
void log(Level level, lazy string message) {
    // ... the body of the function is the same as before ...
}
```

This time, the expression would be evaluated only if the message parameter is used.

One thing to be careful about is that a `lazy` parameter is evaluated every time that parameter is used in the function.

For example, because the `lazy` parameter of the following function is used three times in the function, the expression that provides its value is evaluated three times:

```
import std.stdio;

int valueOfArgument() {
    writeln("Calculating...");
    return 1;
}

void functionWithLazyParameter(lazy int value) {
    int result = value + value + value;
    writeln(result);
}

void main() {
```

```
    functionWithLazyParameter(valueOfArgument());
}
```

Use of lazy keyword

scope

This keyword specifies that a parameter will not be used beyond the scope of the function. At the time of this writing, the `scope` is effective only if the function is defined as `@safe` and if the -dip1000 compiler switch is used. DIP is short for D Improvement Proposal. DIP 1000 is experimental as of this writing; so it may not work as expected in all cases.

```
$ dmd -dip1000 deneme.d
```

```
/* NOTE: This program is expected to fail compilation. */

int[] globalSlice;

@safe int[] foo(scope int[] parameter) {
    globalSlice = parameter;    // ← compilation ERROR
    return parameter;           // ← compilation ERROR
}

void main() {
    int[] slice = [ 10, 20 ];
    int[] result = foo(slice);
}
```

scope keyword

The function above violates the promise of `scope` in two places: it assigns the parameter to a global variable, and it returns it. Both those actions would make it possible for the parameter to be accessed after the function finishes.

shared

This keyword requires that the parameter is shareable between threads of execution:

```
/* NOTE: This program is expected to fail compilation. */

void foo(shared int[] i) {

    // ...
}

void main() {
    int[] numbers = [ 10, 20 ];
    foo(numbers);     // ← compilation ERROR
}
```

The parameter should be shareable for compilation

The program above cannot be compiled because the argument is not shared. The following is the necessary change to make it compile:

```
void foo(shared int[] i) {
    // ...
}

void main() {
    shared int[] numbers = [ 10, 20 ];
    foo(numbers); // now compiles
}
```

The parameter should be shareable for compilation

`return`

Sometimes it is useful for a function to return one of its `ref` parameters directly. For example, the following `pick()` function picks and returns one of its parameters randomly so that the caller can mutate the lucky one directly:

```
import std.stdio;
import std.random;

ref int pick(ref int lhs, ref int rhs) {
    return uniform(0, 2) ? lhs : rhs;
}

void main() {
    int a;
    int b;

    pick(a, b) = 42;
```

```
    writefln("a: %s, b: %s", a, b);
}
```

Returning ref parameter directly

As a result, either `a` or `b` inside `main()` is assigned the value 42. Unfortunately, one of the arguments of `pick()` may have a shorter lifetime than the returned reference. For example, the following `foo()` function calls `pick()` with two local variables: effectively itself returning a reference to one of them:

```
import std.random;

ref int pick(ref int lhs, ref int rhs) {
    return uniform(0, 2) ? lhs : rhs;
}

ref int foo() {
    int a;
    int b;

    return pick(a, b);    // ← BUG: returning invalid reference
}

void main() {
    foo() = 42;           // ← BUG: writing to invalid memory
}
```

Undefined behavior

Since the lifetimes of both `a` and `b` end upon leaving `foo()`, the assignment in `main()` cannot be made to a valid variable. This results in *undefined behavior*.

The term *undefined behavior* describes situations where the behavior of the program is not defined by the programming language specification. Nothing can be said about the behavior of a program that contains undefined behavior. (In practice though, for the program above, the value 42 would most likely be written to a memory location that used to be occupied by either `a` or `b`, potentially a part of an unrelated variable, which effectively corrupts the value of that unrelated variable.)

The `return` keyword can be applied to a parameter to prevent such bugs. It specifies that a parameter must be a reference to a variable with a longer lifetime than the returned reference:

```
/* NOTE: This program is expected to fail compilation. */

import std.random;

ref int pick(return ref int lhs, return ref int rhs) {
    return uniform(0, 2) ? lhs : rhs;
}

ref int foo() {
    int a;
    int b;

    return pick(a, b);    // ← compilation ERROR
}

void main() {
    foo() = 42;
}
```

Sealed references

This time the compiler sees that the arguments to `pick()` have shorter lifetimes than the reference that `foo()` is attempting to return.
This feature is called **sealed references**.

> **Note:** Although it is conceivable that the compiler could inspect `pick()` and detect the bug even without the `return` keyword, it cannot do so in general because the bodies of some functions may not be available to the compiler during every compilation.

In the next lesson, you will find a summary of the function parameters.