scope

This lesson defines the scope statement and explains how it is different from catch and finally.

scope

As we have seen in the previous chapter, expressions that must always be executed are written in the `finally` block, and expressions that must be executed when there are error conditions are written in `catch` blocks. We can make the following observations about the use of these blocks:

- `catch` and `finally` cannot be used without a `try` block.
- Some of the variables that these blocks need may not be accessible within these blocks:

```
void foo(ref int r) {
    try {
        int addend = 42;

        r += addend;
        mayThrow();

    } catch (Exception exc) {
        r -= addend;// ← compilation ERROR
    }
}
```

The function first modifies the reference parameter in a try block and then reverts this modification when an exception is thrown in the catch block. Unfortunately, `addend` is accessible only in the `try` block, where it is defined.

- Writing all of the potentially unrelated expressions in the single `finally` block at the bottom separates those expressions from the actual code that they are related to.

The scope statements have similar functionality to the `catch` and `finally` scopes, but scope statements are better in many respects. Like `finally`, the three different scope statements are about executing expressions when leaving scopes:

- `scope(exit)`: the expression is always executed when exiting the scope, regardless of whether it is successful or due to an exception
- `scope(success)`: the expression is executed only if the scope is being exited successfully
- `scope(failure)`: the expression is executed only if the scope is being exited due to an exception

Although these statements are closely related to exceptions, they can be used without a try-catch block.
As an example, let's write the function above with a `scope(failure)` statement:

```
void foo(ref int r) {
    int addend = 42;
    r += addend;
    scope(failure) r -= addend;
    mayThrow();
}
```

The `scope(failure)` statement above ensures that the `r -= addend` expression will be executed if the function's scope is exited due to an exception. A benefit of `scope(failure)` is the fact that the expression that reverts another expression is written close to it.
`scope` statements can be specified as blocks as well:

```
scope(exit) {
    // ... expressions ...
}
```

Here is another function that tests all three of these statements:

```
void test() {
    scope(exit) writeln("when exiting 1");

    scope(success) {
        writeln("if successful 1");
        writeln("if successful 2");
    }

    scope(failure) writeln("if thrown 1");
    scope(exit) writeln("when exiting 2");
    scope(failure) writeln("if thrown 2");

    throwsHalfTheTime();
}
```

scope statements

If no exception is thrown, the output of the function includes only the `scope(exit)` and `scope(success)` expressions:

```
when exiting 2
if successful 1
if successful 2
when exiting 1
```

If an exception is thrown, the output includes the `scope(exit)` and `scope(failure)` expressions:

```
if thrown 2
when exiting 2
if thrown 1
when exiting 1
object.Exception@...: the error message
```

As seen in the outputs, the blocks of the scope statements are executed in reverse order. This is because later code may depend on previous variables. Executing the scope statements in reverse order enables the undoing side effects of earlier expressions in a consistent order.

In the next lesson, you will find a quiz to test your understanding of the concepts covered in this chapter.