The throw Statement

This lesson explains in-depth the working of a throw statement.

The `throw` statement

`throw` throws an exception object and this terminates the current operation of the program. The expressions and statements that are written after the throw statement are not executed. This behavior is according to the nature of exceptions: they must be thrown when the program cannot continue with its current task.

The exception types `Exception` and `Error`

Only the types that are inherited from the `Throwable` class can be thrown. `Throwable` is almost never used directly in programs. The types that are actually thrown are types that are inherited from `Exception` or `Error`, which themselves are the types that are inherited from `Throwable`. For example, all of the exceptions that “Phobos” throws are inherited from either `Exception` or `Error`.

`Error` represents unrecoverable conditions and is not recommended to be caught. Therefore, most of the exceptions that a program throws are the types that are inherited from `Exception`.

> **Note:** Inheritance is a topic related to classes.

`Exception` objects are constructed with a string value that represents an error message. You may find it easy to create this message with the `format()` function from the `std.string` module:

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

int[] randomDiceValues(int count) {
    if (count < 0) {
        throw new Exception(
            format("Invalid dice count: %s", count));
    }

    int[] values;

    foreach (i; 0 .. count) {
        values ~= uniform(1, 7);
    }

    return values;
}

void main() {
    writeln(randomDiceValues(-5));
}
```

Throwing exception using format() function

In most cases, instead of creating an exception object explicitly by `new` and throwing it explicitly by `throw`, the `enforce()` function is called. For example, the equivalent of the error check above is the following `enforce()` call:

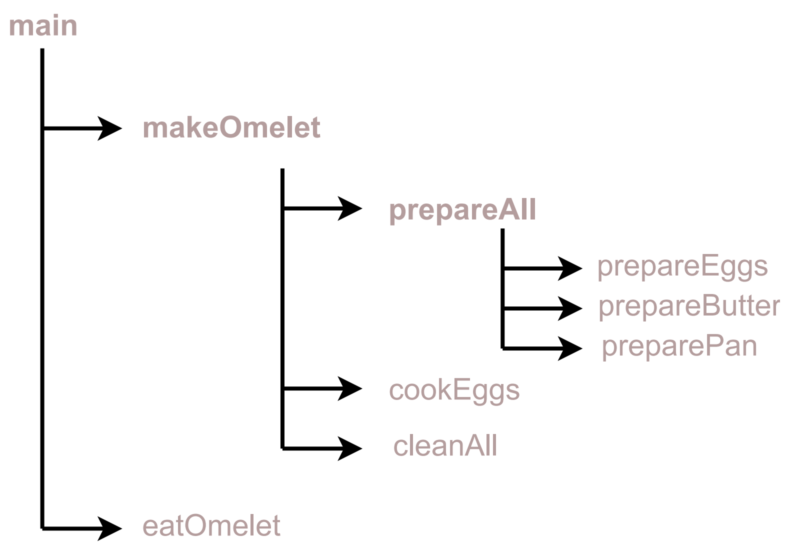
```
import std.stdio;
import std.random;
import std.string;
import std.exception;

int[] randomDiceValues(int count) {

    enforce(count >= 0, format("Invalid dice count: %s", count));

    int[] values;

    foreach (i; 0 .. count) {
        values ~= uniform(1, 7);
    }
    return values;
}


void main() {
    writeln(randomDiceValues(-5));
}
```

Throwing exception using format() function

We will see the differences between `enforce()` and `assert()` in a later chapter.

Exceptions and scope termination

We have seen that the program execution starts from the main function and branches into other functions from there. This layered execution of going deeper into functions and eventually returning from them can be seen as the branches of a tree.
For example, `main()` may call a function named `makeOmelet`, which in turn may call another function named `prepareAll`, which in turn may call another function named `prepareEggs`, etc. Assuming that the arrows indicate function calls, the branching of such a program can be shown as in the following function call tree:



Branching of function calls

The following program demonstrates the branching above by using different levels of indentation in its output. The program doesn't do anything useful other than producing an output suitable for our purposes:

```
import std.stdio;

void indent(int level) {
    foreach (i; 0 .. level * 2) {
        write(' ');
    }
}

void entering(string functionName, int level) {
    indent(level);
    writeln("▶ ", functionName, "'s first line");
}

void exiting(string functionName, int level) {
    indent(level);
    writeln("◁ ", functionName, "'s last line");
}

void main() {
    entering("main", 0);
    makeOmelet();
    eatOmelet();
    exiting("main", 0);
}

void makeOmelet() {
    entering("makeOmelet", 1);
    prepareAll();
    cookEggs();
    cleanAll();
    exiting("makeOmelet", 1);
}

void eatOmelet() {
    entering("eatOmelet", 1);
    exiting("eatOmelet", 1);
}

void prepareAll() {
    entering("prepareAll", 2);
    prepareEggs();
    prepareButter();
    preparePan();
    exiting("prepareAll", 2);
}

void cookEggs() {
    entering("cookEggs", 2);
    exiting("cookEggs", 2);
}

void cleanAll() {
    entering("cleanAll", 2);
    exiting("cleanAll", 2);
}

void prepareEggs() {
    entering("prepareEggs", 3);
    exiting("prepareEggs", 3);
}


void prepareButter() {
    entering("prepareButter", 3);
    exiting("prepareButter", 3);
}

void preparePan() {
    entering("preparePan", 3);
    exiting("preparePan", 3);
}
```

Program to demonstrate branching

The functions entering and exiting are used to indicate the first and last lines of functions with the help of the ▶ and ◁ characters. The program starts with the first line of `main()`, branches down to other functions and finally ends with the last line of `main`.

Let's modify the `prepareEggs` function to take the number of eggs as a parameter. Since certain values of this parameter would be an error, let's have this function throw an exception when the number of eggs is less than one.

In order to be able to compile the program, we must modify other lines of the program to be compatible with this change. The number of eggs to take out of the fridge can be handed down from function to function, starting with `main()`. The parts of the program that need to change are highlighted in the code given below. The invalid value of -8 is intentional to show how the output of the program will be different from the previous output when an exception is thrown:

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

void indent(int level) {
    foreach (i; 0 .. level * 2) {
        write(' ');
    }
}

void entering(string functionName, int level) {
    indent(level);
    writeln("▶ ", functionName, "'s first line");
}
```

```
}

void exiting(string functionName, int level) {
    indent(level);
    writeln("◁ ", functionName, "'s last line");
}

void RENAMED_main() {
    entering("main", 0);
    makeOmelet();
    eatOmelet();
    exiting("main", 0);
}

void makeOmelet() {
    entering("makeOmelet", 1);
    prepareAll();
    cookEggs();
    cleanAll();
    exiting("makeOmelet", 1);
}

void eatOmelet() {
    entering("eatOmelet", 1);
    exiting("eatOmelet", 1);
}

void prepareAll() {
    entering("prepareAll", 2);
    prepareEggs();
    prepareButter();
    preparePan();
    exiting("prepareAll", 2);
}

void cookEggs() {
    entering("cookEggs", 2);
    exiting("cookEggs", 2);
}

void cleanAll() {
    entering("cleanAll", 2);
    exiting("cleanAll", 2);
}

void prepareEggs() {
    entering("prepareEggs", 3);
    exiting("prepareEggs", 3);
}

void prepareButter() {
    entering("prepareButter", 3);
    exiting("prepareButter", 3);
}

void preparePan() {
    entering("preparePan", 3);
    exiting("preparePan", 3);
}
import std.string;

// ...
```

```
void prepareEggs(int count) {
    entering("prepareEggs", 3);

    if (count < 1) {
        throw new Exception(
            format("Cannot take %s eggs from the fridge", count));
    }

    exiting("prepareEggs", 3);
}
// ...

void main() {
    entering("main", 0);
    makeOmelet(-8);
    eatOmelet();
    exiting("main", 0);
}

void makeOmelet(int eggCount) {
    entering("makeOmelet", 1);
    prepareAll(eggCount);
    cookEggs();
    cleanAll();
    exiting("makeOmelet", 1);
}

// ...

void prepareAll(int eggCount) {
    entering("prepareAll", 2);
    prepareEggs(eggCount);
    prepareButter();
    preparePan();
    exiting("prepareAll", 2);
}

// ...
```

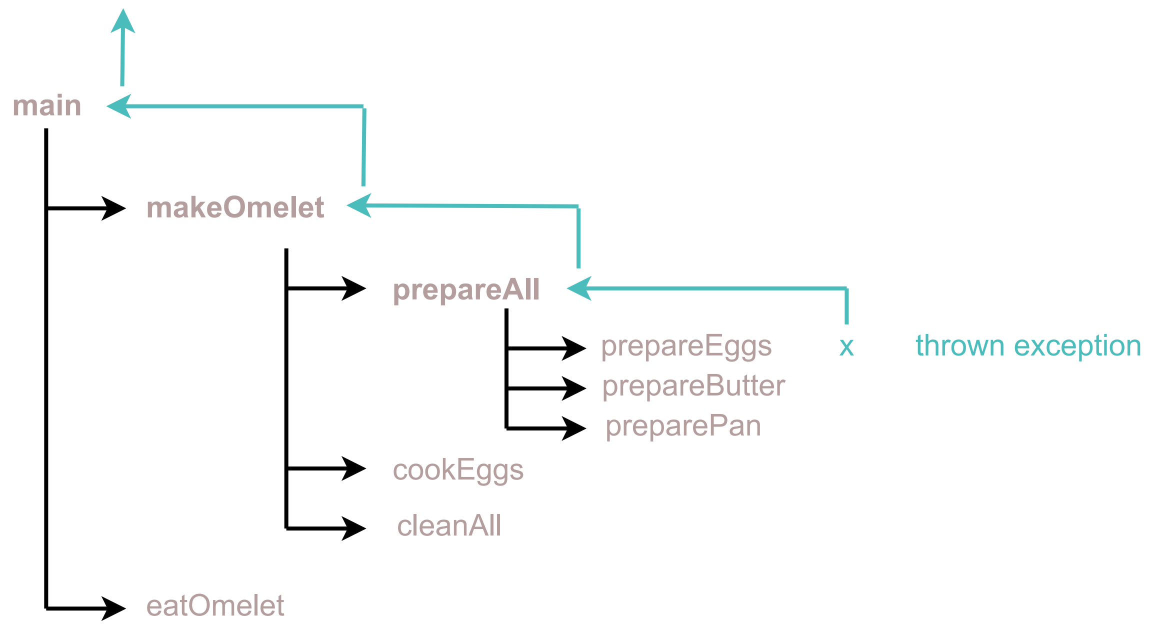
prepareEggs() throws an exception when the number of eggs is less than one

When we start the program now, we see that the lines that used to be printed after the point where the exception is thrown are missing.

When the exception is thrown, the program execution exits the `prepareEggs`, `prepareAll`, `makeOmelet` and `main()` functions in that order, from the bottom level to the top level. No additional steps are executed as the program exits these functions.

The rationale for such a drastic termination is that a failure in a lower-level function would mean that the higher-level functions that relied upon its

successful completion should also be considered as failed.
The exception object that is thrown from a lower-level function is transferred to the higher-level functions one level at a time and causes the program to finally exit the `main()` function. The path that the exception takes is shown as the highlighted path in the following tree:



Branching of function calls

The point of the exception mechanism is precisely this behavior of exiting all of the layers of function calls right away.
Sometimes it makes sense to catch the thrown exception to find a way to continue the execution of the program. I will introduce the `catch` keyword in the next lesson.

When to use throw

Use `throw` in situations when it is not possible to continue. For example, a function that reads the number of students from a file may throw an exception if this information is not available or incorrect. On the other hand, if the problem is caused by some user action like entering invalid data, it may make more sense to validate the data instead of throwing an exception. Displaying an error message and asking the user to re-enter the data is more appropriate in many cases.

In the next lesson, we will explore try-catch statements to catch exceptions.