Kinds of Errors

This lesson sheds light on different types of errors and explains how exceptions can be used to handle different types of errors.

WE'LL COVER THE FOLLOWING

Kinds of errors

We have seen how useful the exception mechanism is. It enables both the lower and higher-level operations to be aborted right away instead of letting the program continue with incorrect or missing data or behave in any other incorrect way. This does not mean that every error condition warrants throwing an exception. There may be better things to do depending on the kinds of errors.

User errors

Some errors are caused by the user. As we have seen above, the user may have entered a string like “hello” even though the program has been expecting a number. It may be more appropriate to display an error message and ask the user to enter appropriate data again.
Even so, it may be fine to accept and use the data directly without validating the data upfront; as long as the code that uses the data would throw anyway. However, it is important to be able to notify the user of why the data is not suitable.

For example, let’s look at a program that takes a file name from the user.

There are at least two ways of dealing with potentially invalid file names:

- **Validating the data before use:** We can determine whether the file with the given name exists by calling `exists()` of the `std.file` module:

  ```
  if (exists(fileName)) {
    // yes, the file exists

  } else {
    // no, the file doesn't exist
  }
  ```

  This gives us the chance to be able to open the data only if it exists. Unfortunately, it is still possible that the file cannot be opened even if `exists()` returns true if, for example, another process in the system deletes or renames the file before this program actually opens it.
  For that reason, the following method may be more useful.

- **Using the data without first validating it:** We can assume that the data is valid and start using it right away because `File` would throw an exception if the file cannot be opened anyway.

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

void useTheFile(string fileName) {
    auto file = File(fileName, "r");
    // ...
}

string read_string(string prompt) {
    write(prompt, ": ");
    return strip(readln());
}

void main() {
    bool is_fileUsed = false;

    while (!is_fileUsed) {
        try {
            useTheFile(
                read_string("Please enter a file name"));

            /* If we are at this line, it means that
             * useTheFile() function has been completed
             * successfully. This indicates that the file
             * name was valid.
             *
             * We can now set the value of the loop flag to
             * terminate the while loop. */
```

```
            is_fileUsed = true;
            writeln("The file has been used successfully");

        } catch (std.exception.ErrnoException exc) {
            stderr.writeln("This file could not be opened");
        }
    }
}
```

Using the data right away without first validating it

Programmer errors

Some errors are caused by programmer mistakes. For example, the programmer may think that a function that has just been written will always be called with a value greater than or equal to zero, and this may be true according to the design of the program. If the function is called with a value less than zero, it would indicate either a mistake in the design of the program or in the implementation of that design. Both of these can be thought of as programming errors.

It is more appropriate to use `assert` than the exception mechanism for errors that are caused by programmer mistakes.

> **Note:** We will cover `assert` in a later chapter

```
void processMenuSelection(int selection) {
    assert(selection >= 0);
    // ...
}

void main() {
    processMenuSelection(-1);
}
```

Using assert instead of the exception mechanism

The program terminates with an assert failure:

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

`assert` validates program state and prints the file name and line number of

the validation if it fails. The message above indicates that the assertion at line 2 of main.d has failed.

Unexpected situations

For unexpected situations that are outside of the two general cases above, it is still appropriate to throw exceptions. If the program cannot continue its execution, there is nothing else to do but throw.
It is up to the higher layer functions that call this function to decide what to do with thrown exceptions. They may catch the exceptions that we throw to remedy the situation.

Exceptions summary

- When faced with a user error either warn the user right away or ensure that an exception is thrown. The exception may be thrown anyway by another function when using incorrect data, or you may throw directly.
- Use `assert` to validate program logic and implementation.
- When in doubt, throw an exception with `throw` or `enforce()`.
- Catch exceptions if and only if you can do something useful about that exception. Otherwise, do not encapsulate code with a try-catch statement; instead, leave the exceptions to higher layers of the code that may do something about them.
- Order the catch blocks from the most specific to the most general.
- Put the expressions that must always be executed when leaving a scope, in `finally` blocks.

The next lesson explains the scope statement.