# Immutability

This lesson explains the concept of immutability and immutable variables.

WE'LL COVER THE FOLLOWING



## What is immutability?

We have seen that variables represent concepts in programs. The interactions of these concepts are achieved by expressions that change the values of those variables:

```
// Pay the bill
totalPrice = calculateAmount(itemPrices);
moneyInWallet -= totalPrice;
moneyAtMerchant += totalPrice;
```

- Some concepts are *immutable* by definition. For example, there are always seven days in a week, the math constant Pi (π) never changes, the list of natural languages supported by a program may be fixed and small (e.g. only English and Turkish), etc.

- If every variable were modifiable, then any piece of code in the program could potentially modify it. Even if there was no reason to modify a variable in operation there would be no guarantee that this would not happen by accident.

Programs are difficult to read and maintain when there are no immutability

guarantees. For example, consider a function called *retire(office, worker)* that retires a worker of an office. All variables passed as parameters in functions are not modified by the function. However, just by looking at the function call, we cannot determine which variables are changed, and which are left unchanged.

The concept of immutability helps with understanding parts of programs by guaranteeing that certain operations do not change certain variables. It also reduces the risk of some types of programming errors.
The immutability concept is expressed in D by the `const` and `immutable` keywords. Although the two words themselves are close in meaning, their responsibilities in programs are different, and they are sometimes incompatible.
`const` and `immutable` are type qualifiers. We will see `inout` and `shared` in later chapters.

## Immutable variables #

Both of the terms “immutable variable” and “constant variable” are nonsensical when the word “variable” is taken literally to mean something that changes. However, in a broader sense, the word “variable” is often understood to mean any concept of a program which may be mutable or immutable.

There are three ways of defining variables that can never be mutated. `enum` constants, `immutable` variables and `const` variables.

## `enum` constants #

We have seen earlier in the enum lesson that `enum` defines named constant values:

```
enum fileName = "list.txt";
```

As long as their values can be determined at compile time, `enum` variables can be initialized with return values of functions as well:

```
import std.stdio;

int totalLines() {
    return 42;
}
```

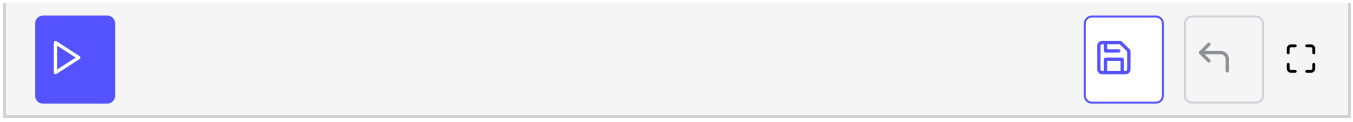
```
int totalColumns() {
    return 7;
}

string name() {
    return "list";
}

void main() {

    enum fileName = name() ~ ".txt";
    enum totalSquares = totalLines() * totalColumns();

    writeln(fileName);
    writeln(totalSquares);
}
```



Defining enums with return values of functions

The D feature that enables such initialization is *compile time function execution* (CTFE). As expected, the values of `enum` constants cannot be modified, although it is a very effective way of representing `immutable` values:

```
++totalSquares;    // ← compilation ERROR
```

An `enum` constant is a *manifest constant*, meaning that the program is compiled as if every mention of that constant had been replaced by its value. As an example, let's consider the following `enum` definition and the two expressions that make use of it:

```
enum i = 42;
writeln(i);
foo(i);
```

The code above is completely equivalent to the one below, where we replace every use of `i` with its value of 42:

```
writeln(42);
foo(42);
```

Although that replacement makes sense for simple types like `int` and makes no difference to the resulting program, `enum` constants can bring a hidden cost when they are used for arrays or associative arrays:

when they are used for arrays or associative arrays.

```
enum a = [ 42, 100 ];
writeln(a);
foo(a);
```

After replacing `a` with its value, the equivalent code that the compiler would be compiling is the following:

```
writeln([ 42, 100 ]); // an array is created at  run time
foo([ 42, 100 ]); // another array is created at run time
```

The hidden cost here is that there would be two separate arrays created for the two expressions above. For that reason, it may make more sense to define arrays and associative arrays as `immutable` variables if they are going to be used more than once in the program.
`enum` can only be used for compile time values.

## `immutable` variables #

Like `enum`, this keyword specifies that the value of a variable will never change. Unlike `enum`, an `immutable` variable is an actual variable with a memory address, which means that we can set its value during the execution of the program and that we can refer to its memory location.

The following program compares the uses of `enum` and `immutable`. The program asks for the user to guess a number that has been picked randomly. Since the random number cannot be determined at compile-time, it cannot be defined as an `enum`. Still, since the randomly picked value must never be changed after having been decided, it is suitable to specify that variable as `immutable`.
The program takes advantage of the `readInt()` function that was defined in the previous chapter:

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

int readInt(string message) {
    int result;
    write(message, "? ");
    readf(" %s", &result);
    return result;
```

```
}

void main() {

    enum min = 1;
    enum max = 10;

    immutable number = uniform(min, max + 1);

    writefln("I am thinking of a number between %s and %s.",
             min, max);

    auto isCorrect = false;
    while (!isCorrect) {
        immutable guess = readInt("What is your guess");
        isCorrect = (guess == number);
    }

    writeln("Correct!");
}
```

Use of immutable variable

### Observations:

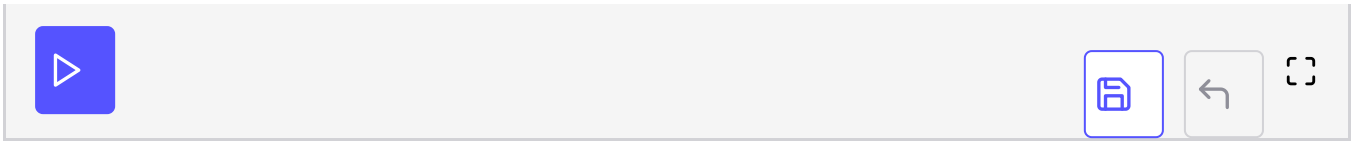
- `min` and `max` are integral parts of the behavior of this program, and their values are known at compile time. For that reason, they are defined as `enum` constants.
- `number` is specified as `immutable` because it would not be appropriate to modify it after its initialization at run time. Likewise for each user guess, once read, the `guess` should not be modified.
- Observe that the types of those variables are not specified explicitly. As with `auto` and `enum`, the type of an `immutable` variable can be inferred from the expression on the right-hand side.
  Although it is not necessary to write the type fully, `immutable` normally takes the actual type within parentheses, e.g. `immutable(int)`. The output of the following program demonstrates that the full names of the types of the three variables are in fact the same:

```
import std.stdio;

void main() {
    immutable      inferredType = 0;
    immutable int  explicitType = 1;
    immutable(int) wholeType    = 2;

    writeln(typeof(inferredType).stringof);
    writeln(typeof(explicitType).stringof);
    writeln(typeof(wholeType).stringof);
```

```
}
```



immutable variables type name

The use of parentheses has significance and specifies which parts of the type are `immutable`. We will see this when discussing the immutability of the whole slice vs. its elements.

## `const` variables #

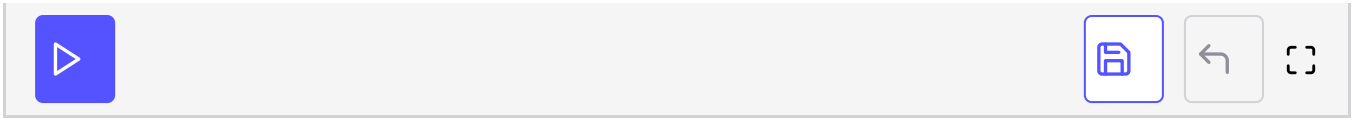
When defining variables the `const` keyword has the same effect as `immutable`. `const` variables cannot be modified:

```
import std.stdio;

void main() {
    int total = 100;

    const half = total / 2;
    half = 10; // ← compilation ERROR

}
```



const variables

I recommend that you prefer `immutable` over `const` for defining variables. The reason is that `immutable` variables can be passed to functions that have `immutable` parameters.

---

In the next lesson, we will see how `immutable` variables can be used with functions.