Immutable Parameters

This lesson explains how immutability can be achieved in the functions having immutable parameters.

WE'LL COVER THE FOLLOWING

Immutable parameters

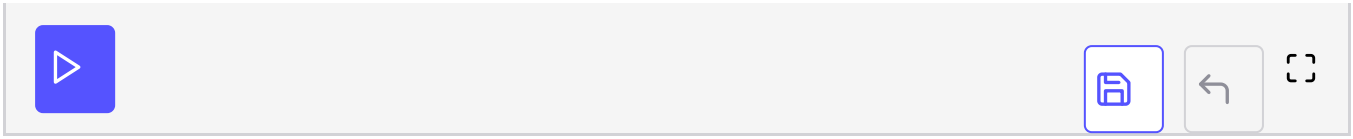
It is possible for functions to promise that they do not modify certain parameters they take, and the compiler will enforce this promise. Before seeing how this is achieved, let's first see how functions can modify the elements of slices that are passed as arguments to those functions.

As you remember from the slices and other array features lesson, slices do not own elements but provide access to them. There may be more than one slice at a given time that provides access to the same elements.

Although the examples in this section focus only on slices, this topic is applicable to associative arrays and classes as well.
A slice that is passed as a function argument is not the slice that the function is called with. The argument is a copy of the actual slice:

```
import std.stdio;

void main() {
    int[] slice = [ 10, 20, 30, 40 ];
    halve(slice);
    writeln(slice);
}

void halve(int[] numbers) {
    foreach (ref number; numbers) {
        number /= 2;
    }
```

```
}
```



Slices and functions

When the program execution enters the `halve()` function, there are two slices that provide access to the same four elements:

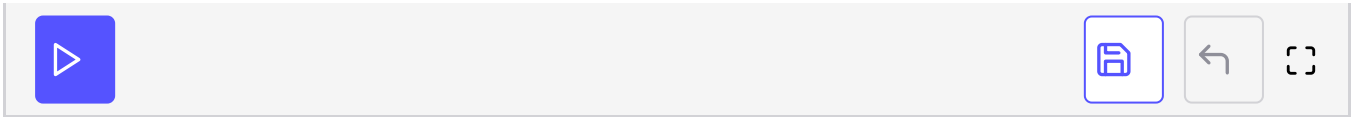
1. The slice named `slice` that is defined in `main()`, which is passed to `halve()` as its argument.
2. The slice named `numbers` that `halve()` receives as its argument, which provides access to the same elements as `slice`.

Since both slices refer to the same elements and given that we use the `ref` keyword in the `foreach` loop, the values of the elements get halved.

It is useful for functions to be able to modify the elements of the slices that are passed as arguments, as seen in this example.

The compiler does not allow passing `immutable` variables as arguments to such functions because we cannot modify an `immutable` variable:

```
import std.stdio;

void halve(int[] numbers) {
    foreach (ref number; numbers) {
        number /= 2;
    }
}

void main() {

    immutable int[] slice = [ 10, 20, 30, 40 ];
    halve(slice); // ← compilation ERROR

}
```



Error thrown because of modifying an immutable variable

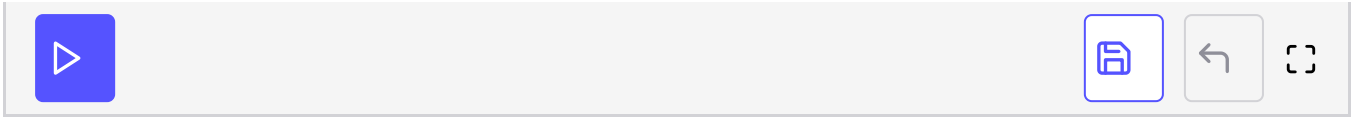
The compilation error indicates that a variable of type `immutable(int[])` cannot be used as an argument of type `int[]`:

```
Error: function deneme.halve (int[] numbers) is not callable using argumen
t types (immutable(int[]))
```

`const` parameters

It is important and natural that `immutable` variables be prevented from being passed to functions like `halve()`, which modify their arguments. However, it would be a limitation if they could not be passed to functions that do not modify their arguments in any way:

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

import std.stdio;

void main() {
    immutable int[] slice = [ 10, 20, 30, 40 ];
    print(slice);    // ← compilation ERROR
}

void print(int[] slice) {
    writefln("%s elements: ", slice.length);

    foreach (i, element; slice) {
        writefln("%s: %s", i, element);
    }
}
```



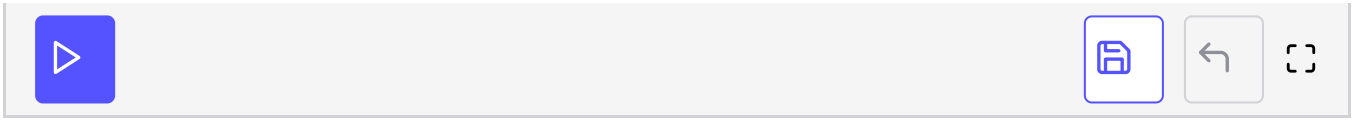
Compilation error while passing slice to a function

It does not make sense above that a slice is prevented from being printed just because it is `immutable`. The proper way of dealing with this situation is by using `const` parameters.

The `const` keyword specifies that a variable is not modified through that particular reference (e.g. a slice) of that variable. Specifying a parameter as `const` guarantees that the elements of the slice are not modified inside the function. Once `print()` provides this guarantee, the program can now be compiled:

```
import std.stdio;
```

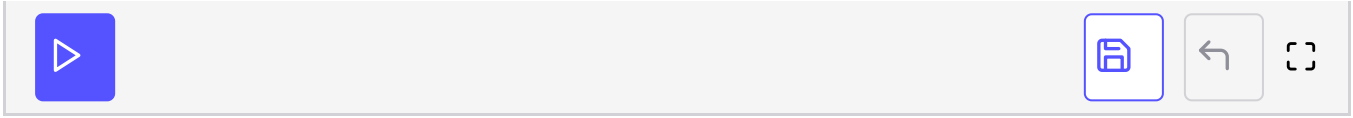
```
void main() {
    immutable int[] slice = [ 10, 20, 30, 40 ];
    print(slice);
}

void print(const int[] slice) {
    writefln("%s elements: ", slice.length);

    foreach (i, element; slice) {
        writefln("%s: %s", i, element);
    }
}
```



Using const while passing slice to a function

This guarantee allows passing both mutable and `immutable` variables as arguments:

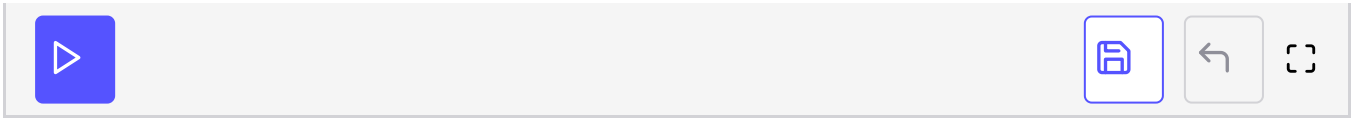
```
import std.stdio;

void main() {
    immutable int[] slice = [ 10, 20, 30, 40 ];
    print(slice); // compiles
    int[] mutableSlice = [ 7, 8 ];
    print(mutableSlice); // compiles
}

void print(const int[] slice) {
    writefln("%s elements: ", slice.length);

    foreach (i, element; slice) {
        writefln("%s: %s", i, element);
    }
}
```



A parameter that is not modified in a function or specified as `const` reduces the applicability of that function. Additionally, `const` parameters provide useful information to the programmer. Knowing that a variable will not be modified when passed to a function makes the code easier to understand. It also prevents potential errors because the compiler detects modifications to `const` parameters:

```
import std.stdio;
```

```
void main() {

    immutable int[] slice = [ 10, 20, 30, 40 ];

    print(slice);

}

void print(const int[] slice) {
    slice[0] = 42; // ← compilation ERROR

    writefln("%s elements: ", slice.length);

    foreach (i, element; slice) {
        writefln("%s: %s", i, element);
    }
}
```



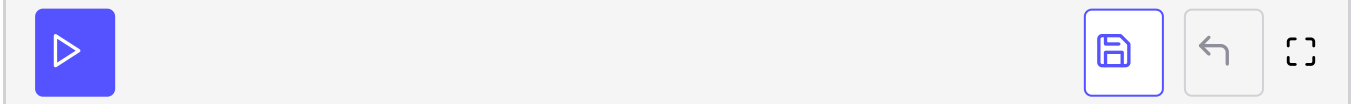
Compiler detects modifications to const parameters

The programmer would either realize the mistake in the function or rethink the design and perhaps remove the `const` specifier.
The fact that `const` parameters can accept both mutable and immutable variables has an interesting consequence. This is explained in the next lesson.

`immutable` parameters

As we have seen above, both mutable and immutable variables can be passed to functions as their `const` parameters. In a way, `const` parameters are welcoming.
In contrast, `immutable` parameters bring a strong requirement: only `immutable` variables can be passed to functions as arguments:

```
void func(immutable int[] slice) {
  // ...
}
void main() {
    immutable int[] immSlice = [ 1, 2 ];
    int[] slice = [8, 9];
    func(immSlice);  //compiles
    func(slice);     //compilation error
}
```



For that reason, the `immutable` specifier should be used only when this

requirement is actually necessary. We have been using the `immutable` specifier indirectly through certain `string` types. We have seen that the parameters that are specified as `const` or `immutable` promise not to modify the actual variable that is passed as an argument. This is relevant only for reference types because, only then, there is the actual variable's immutability to talk about.

Reference types and value types will be covered in a later chapter. Among the types that we have seen so far, only slices and associative arrays are reference types; the others are value types.

In the next lesson, we will see whether a parameter should be `const` or `immutable`.