Immutability of the Slice Versus the Elements

This lesson explains immutability of the slice versus the elements and use of immutability in general.

WE'LL COVER THE FOLLOWING

We have seen earlier in this chapter that the type of an immutable slice has been printed as `immutable(int[])`. As the parentheses after immutable indicate, it is the entire slice that is immutable. Such a slice cannot be modified in any way; elements may not be added or removed, their values may not be modified and the slice may not be changed to provide access to a different set of elements:

```
import std.stdio;

void main() {

    immutable int[] immSlice = [ 1, 2 ];
    immSlice ~= 3;  // ← compilation ERROR
    immSlice[0] = 3;  // ← compilation ERROR
    immSlice.length = 1;  // ← compilation ERROR
    immutable int[] immOtherSlice = [ 10, 11 ];
    immSlice = immOtherSlice; // ← compilation ERROR

}
```

An immutable slice cannot be modified

Taking immutability to that extreme may not be suitable in every case. In most cases, what is important is the immutability of the elements themselves.

Since a slice is just a tool to access the elements, it should not matter if we

make changes to the slice itself as long as the elements are not modified. This is especially true in the cases we have seen so far where the function receives a copy of the slice itself.

To specify that only the elements are immutable, we use the `immutable` keyword with parentheses that enclose just the element type. The code can be modified to make only the elements `immutable` and not the slice itself.

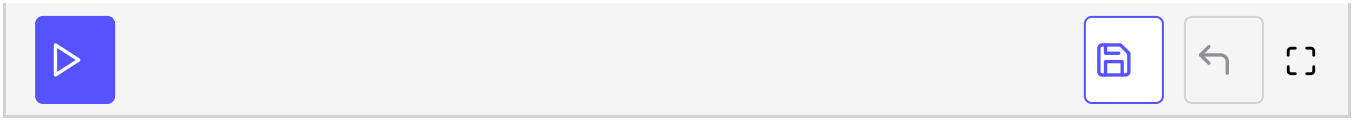
```
import std.stdio;

void main() {

    immutable(int)[] immSlice = [ 1, 2 ];
    immSlice ~= 3; // can add elements
    immSlice[0] = 3; // ← compilation ERROR
    immSlice.length = 1; // can drop elements

    immutable int[] immOtherSlice = [ 10, 11 ];
    immSlice = immOtherSlice; /* can provide access to other elements */

}
```



Just making the elements immutable

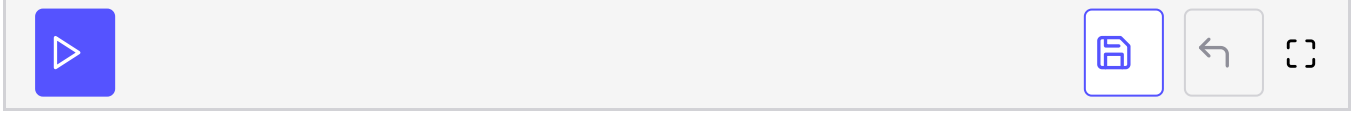
Although both the codes have similar looking syntax, they have different meanings.

```
import std.stdio;

void main() {

    immutable int[] a = [1]; /* Neither the elements nor the * slice can be modified */
    immutable(int[]) b = [1]; /* The same meaning as above */
    immutable(int)[] c = [1]; /* The elements cannot be modified but the slice can be */

}
```



This distinction has been in effect in some of the programs that we have written so far. As you may remember, the three `string` aliases involve immutability:

immutability.

- `string` is an alias for `immutable(char)[]`
- `wstring` is an alias for `immutable(wchar)[]`
- `dstring` is an alias for `immutable(dchar)[]`

Likewise, `string` literals are `immutable` as well:

- The type of literal "hello"c is `string`
- The type of literal "hello"w is `wstring`
- The type of literal "hello"d is `dstring`

According to these definitions, D strings are normally arrays of immutable characters.

`const` and `immutable` are transitive

As mentioned in the example code above, both slices `a` and `b` and their elements are `immutable`.
This is true for user-defined types such as structs and classes as well. For example, all members of a `const struct` variable are `const`, and all members of an `immutable struct` variable are `immutable` (likewise for classes).

`.dup` and `.idup`

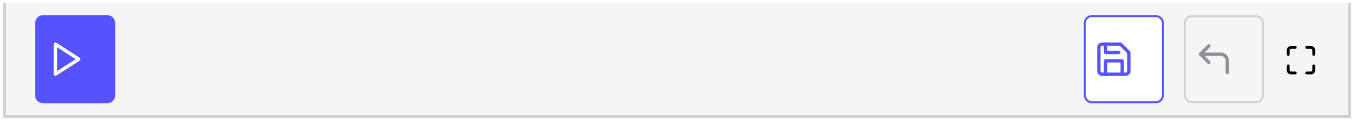
There may be mismatches in immutability when strings are passed to functions as parameters. The `.dup` and `.idup` properties make copies of arrays with the desired mutability:

- `.dup` makes a mutable copy of the array; its name comes from "duplicate."
- `.idup` makes an immutable copy of the array.

For example, a function that insists on the immutability of a parameter may have to be called with an `immutable` copy of a mutable `char[]`:

```
import std.stdio;
void foo(string s) {
// ...
}
void main() {
    char[] salutation;
```

```
    char[] salutation;
    foo(salutation);      // ← compilation ERROR
    foo(salutation.idup); // ← this compiles
}
```



Using .idup for an immutable copy

How to use

- As a general rule, prefer `immutable` variables over mutable ones. For functions that take mutable data and have to modify it, immutable variables will not serve the purpose.
- Define constant values as `enum` if their values can be calculated at compile time. For example, the constant value of seconds per minute can be an `enum`:

  ```
  enum int secondsPerMinute = 60;
  ```

  There is no need to specify the type explicitly if it can be inferred from the right hand side:

  ```
  enum secondsPerMinute = 60;
  ```

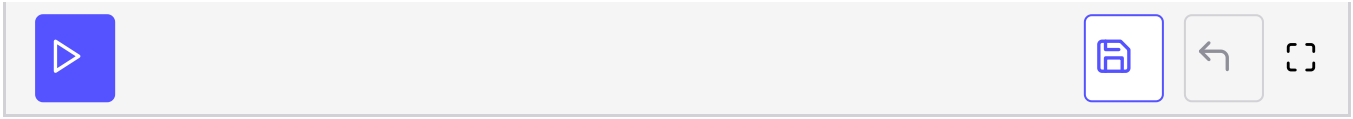
- Consider the hidden cost of `enum` arrays and `enum` associative arrays. Define them as `immutable` variables if the arrays are large and used more than once in the program.
- Specify variables as `immutable` if their values will never change but cannot be known at compile time. Again, the type can be inferred:

  ```
  immutable guess = readInt("What is your guess");
  ```

- If a function does not modify a parameter, specify that parameter as `const`. This would allow both mutable and immutable variables to be passed as arguments:

```
void foo(const char[] s) {
    // ...
}
```

```
void main() {
    char[] salutation;
    foo(salutation);
    foo(salutation.idup);
}
```



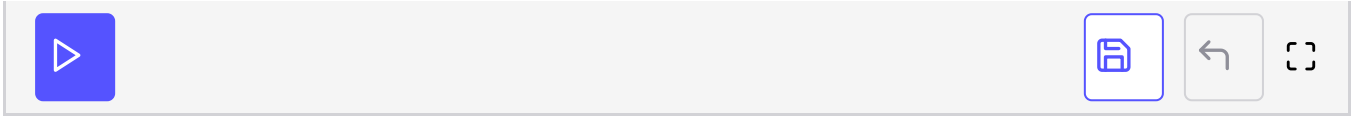
Passing mutable and immutable variables as const parameters

- Following the previous guideline, consider that `const` parameters cannot be passed to functions taking `immutable`. See the lesson should a parameter be const or immutable?.
- If the function modifies a parameter, leave that parameter as mutable (`const` or `immutable` would not allow modifications anyway):

```
import std.stdio;

void reverse(dchar[] s) {
    foreach (i; 0 .. s.length / 2) {
        immutable temp = s[i];
        s[i] = s[$ - 1 - i];
        s[$ - 1 - i] = temp;
    }
}

void main() {
    dchar[] salutation = "hello"d.dup;
    reverse(salutation);
    writeln(salutation);
}
```



Function allowed to modify variables

Summary

- `enum` variables represent immutable concepts that are known at compile time.
- `immutable` variables represent immutable concepts that must be calculated at run time or that must have some memory location that we can refer to.
- `const` parameters are the ones that functions do not modify. Both

mutable and immutable variables can be passed as arguments of `const` parameters.

- `immutable` parameters are the ones that functions specifically require them to be so. Only `immutable` variables can be passed as arguments of `immutable` parameters.

- `immutable(int[])` specifies that neither the slice nor its elements can be modified.

- `immutable(int)[]` specifies that only the elements cannot be modified.

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