Reference Types

In this lesson, you will learn how variables of reference types maintain values.

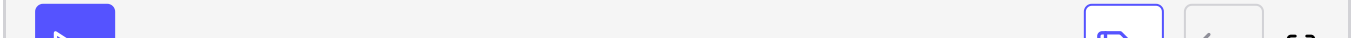
WE'LL COVER THE FOLLOWING

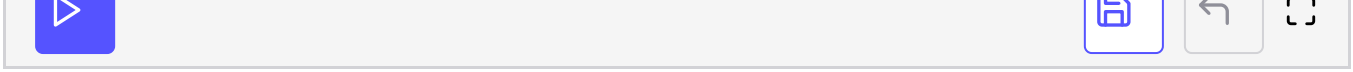
Variables of **reference types** have individual identities, but they do not have individual values. They provide access to existing variables.

Slices as reference types

We have already seen this concept with slices. Slices do not own elements, they provide access to existing elements:

```
void main() {
    // Although it is named as 'array' here, this variable is
    // a slice as well. It provides access to all of the
    // initial elements:
    int[] array = [ 0, 1, 2, 3, 4 ];

    // A slice that provides access to elements other than the
    // first and the last:
    int[] slice = array[1 .. $ - 1];

    // At this point slice[0] and array[1] provide access to
    // the same value:
    assert(&slice[0] == &array[1]);

    // Changing slice[0] changes array[1]:
    slice[0] = 42;
    assert(array[1] == 42);
}
```





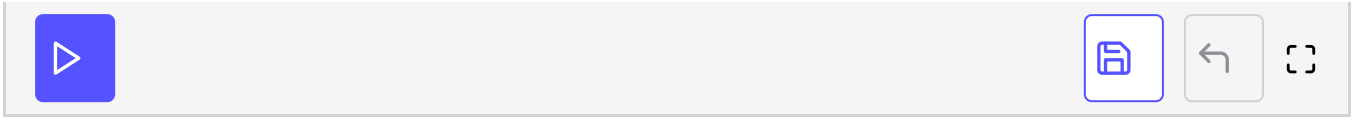
Slice providing access to array elements

Contrary to reference variables, reference types are not simply aliases. To see this distinction, let's define another slice as a copy of one of the existing slices:

```
int[] slice2 = slice;
```

The two slices have their own addresses. In other words, they have separate identities:

```
import std.stdio;

void main() {
    // Although it is named as 'array' here, this variable is
    // a slice as well. It provides access to all of the
    // initial elements:
    int[] array = [ 0, 1, 2, 3, 4 ];

    // A slice that provides access to elements other than the
    // first and the last:
    int[] slice = array[1 .. $ - 1];

    int[] slice2 = slice;

    assert(&slice != &slice2);
    writeln(&slice," : ",&slice2);
}
```

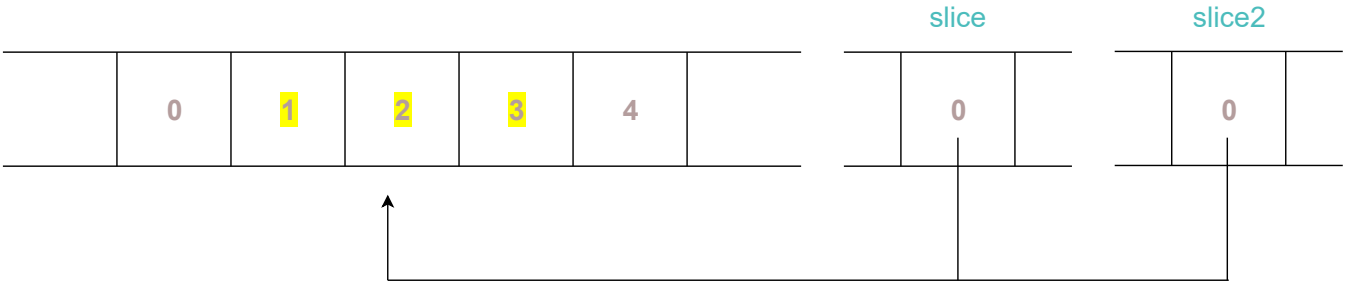


Slice providing access to array elements

The following list is a summary of the differences between reference variables and reference types:

- Reference variables do not have identities, they are aliases of existing variables.
- Variables of reference types have identities, but they do not own values; rather, they provide access to existing values.

The way `slice` and `slice2` live in memory can be illustrated as in the following figure:



Representation of slice and slice2 in the memory

The three elements that the two slices both reference are highlighted.

Class objects as reference types

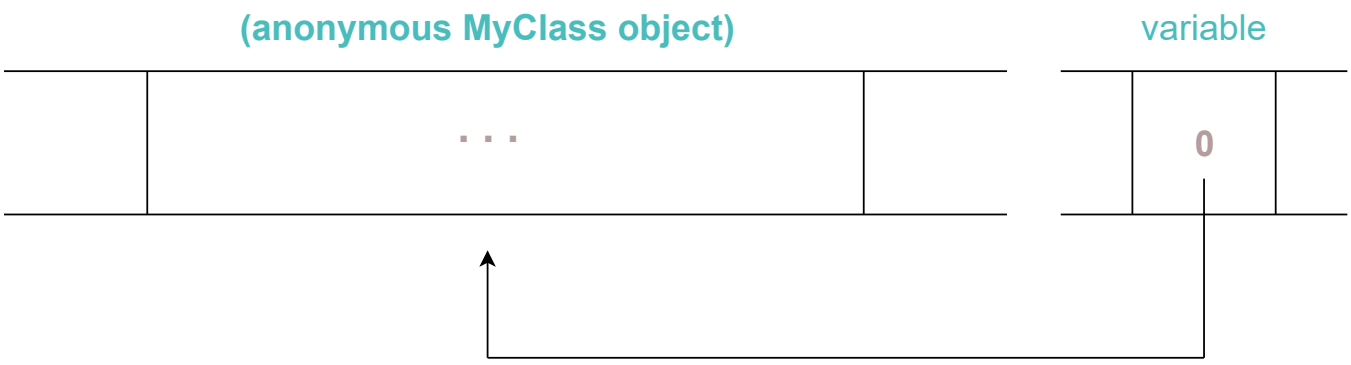
One of the differences between C++ and D is that classes are reference types in D. The following is a short example to demonstrate this fact:

```
class MyClass {
    int member;
}
```

Class objects are constructed by the `new` keyword:

```
auto variable = new MyClass;
```

`variable` is a reference to an anonymous `MyClass` object that has been constructed by `new`:



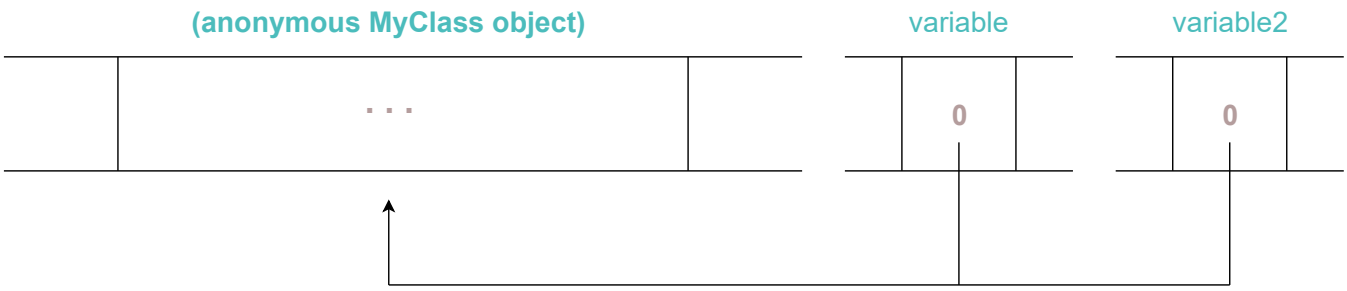
Myclass object referenced by variable

Just like with slices, when a variable is copied, the copy becomes another reference to the same object. The copy has its own address:

```
import std.stdio;

class MyClass {
    int member;
}

void main() {
    auto variable = new MyClass;
    auto variable2 = variable;
    assert(variable == variable2);
    assert(&variable != &variable2);

    writeln(&variable," : ",&variable2);
}
```

Address of a copied variable

The variables are equal from the point of view of referencing the same object, but they are separate variables:



variable and variable2 have different addresses

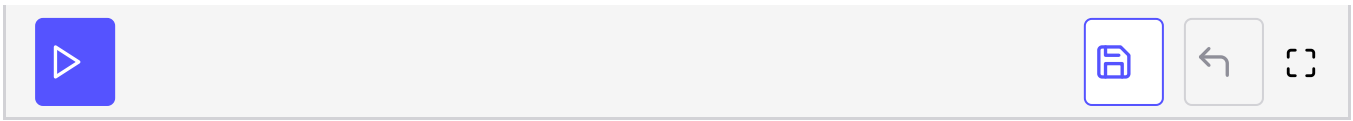
This can also be shown by modifying the member of the object:

```
import std.stdio;

class MyClass {
    int member;
}

void main() {
    auto variable = new MyClass;
    variable.member = 1;
    auto variable2 = variable; // They share the same object
    variable2.member = 2;
    assert(variable.member == 2); // The object that variable provides access to has changed.
}
```

Associative arrays as reference types

Another reference type is associative arrays. Like slices and classes, when an associative array is copied or assigned to another variable, both give access to the same set of elements:

```
import std.stdio;

void main() {
    string[int] byName =
    [
    1 : "one",
    10 : "ten",
    100 : "hundred",
    ];

    // The two associative arrays will be sharing the same set of elements
    string[int] byName2 = byName;

    // The mapping added through the second ...
    byName2[4] = "four";

    // ... is visible through the first.
    assert(byName[4] == "four");
}
```



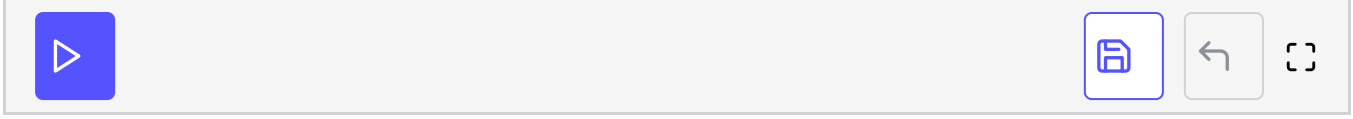
Copying associative arrays

There is no element sharing if the original associative array were null to begin with. This idea will be explained in the next chapter.

The difference in the assignment operation

With value types and reference variables, the assignment operation changes the actual value:

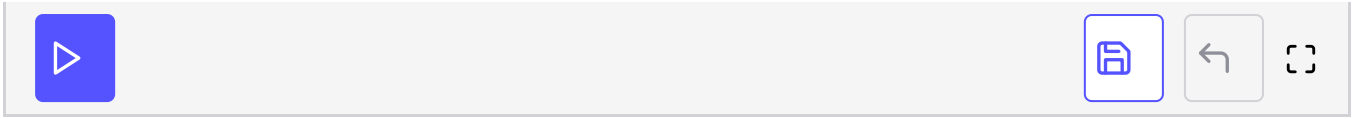
```
import std.stdio;

void main() {
    int number = 8;
    halve(number);

    writeln(number);
}
void halve(ref int dividend) {
    dividend /= 2;  // The actual value changes
}
```



Value of number variable is changed

On the other hand, with reference types, the assignment operation changes which value is being accessed. For example, the assignment of the `slice3` variable below does not change the value of any element; rather, it changes what elements `slice3` is now a reference of:

```
import std.stdio;

void main (){
    int[] slice1 = [ 10, 11, 12, 13, 14 ];
    int[] slice2 = [ 20, 21, 22 ];
    int[] slice3 = slice1[1 .. 3]; // Access to element 1 and element 2 of slice1

    slice3[0] = 777;
    assert(slice1 == [ 10, 777, 12, 13, 14 ]);

    // This assignment does not modify the elements that
    // slice3 is providing access to. It makes slice3 provide
    // access to other elements.
    slice3 = slice2[$ - 1 .. $]; // Access to the last element

    slice3[0] = 888;
    assert(slice2 == [ 20, 21, 888 ]);
}
```

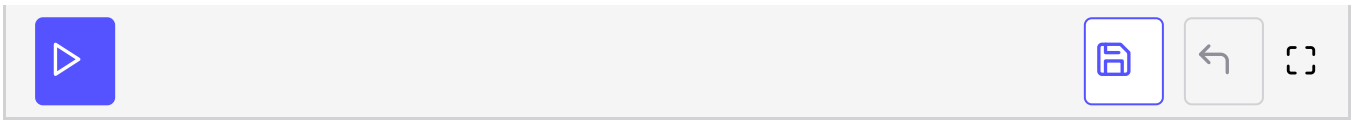


Successful assertion as values are not modified

This time, let's demonstrate the same effect with two objects of the `MyClass` type:

```
import std.stdio;

class MyClass {
    int member;
}

void main() {
    auto variable1 = new MyClass;
    variable1.member = 1;

    auto variable2 = new MyClass;
    variable2.member = 2;

    auto aCopy = variable1;
    aCopy.member = 3;
```

```
    aCopy = variable2;
    aCopy.member = 4;


    assert(variable1.member == 3);
    assert(variable2.member == 4);
}
```



Assignment operation on class objects

The `aCopy` variable above first references the same object as `variable1` and then the same object as `variable2`. As a consequence, the `.member` that is modified through `aCopy` is first `variable1`'s and then `variable2`'s.

Variables of reference types may not be referencing any object

With a reference variable, there is always an actual variable that it is an alias of; it can not start its life without a variable. On the other hand, variables of reference types can start their lives without referencing any object.
For example, a `MyClass` variable can be defined without an actual object being created by `new`:

```
MyClass variable;
```

Such variables have the special value of `null`. We will cover `null` and the `is` keyword in a later chapter.

In the next lesson, we will see how value type and reference type behave in different scenarios.