Object lifetime

This lesson discusses how std::variant handles object lifetime.

```
we'll cover the following ^
std::variant and Object Lifetime
```

std::variant and Object Lifetime

When you use union, you need to manage the internal state: call constructors or destructors. This is error-prone, and it's easy to shoot yourself in the foot. But std::variant handles object lifetime as you expect. That means that if it's about to change the currently stored type, then a destructor of the underlying type is called.

```
// v allocates some memory for the string
std::variant<std::string, int> v { "Hello A Quite Long String" };

// we call destructor for the string!
v = 10; // no memory leak
```

Or see this example with a custom type:

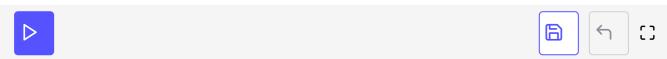
```
#include <iostream>
#include <variant>
using namespace std;

class MyType {
public:
    MyType() {
        std::cout << "MyType::MyType\n";
    }
    ~MyType() {
        std::cout << "MyType::~MyType\n";
    }
};

class OtherType {
public:</pre>
```

```
OtherType() {
    std::cout << "OtherType::OtherType\n";
}
    ~OtherType() {
    std::cout << "OtherType::~OtherType\n";
}
};

int main() {
    std::variant<MyType, OtherType> v;
    v = OtherType();
    return 0;
}
```



At the start, we initialize with a default value of type MyType. Then we change the value with an instance of OtherType, and before the assignment, the destructor of MyType is called. Later we destroy the temporary object and the object stored in the variant.

Next lesson will entail information on how to access stored values inside a variant using the helper functions.