With std::variant

The lesson describes how std::variant is better than std::optional in terms of error handling.

WE'LL COVER THE FOLLOWING

How to do Error Handling using std::variant?

The last implementation with `std::optional` omits one crucial aspect: **error handling**.

There's no way to know the reason why a value wasn't computed. For example with the version where `std::pair` was used, we were able to return an error code to indicate the reason. What can we do about that?

If you need full information about the error that might occur in the function, you can think about an alternative approach with `std::variant`.

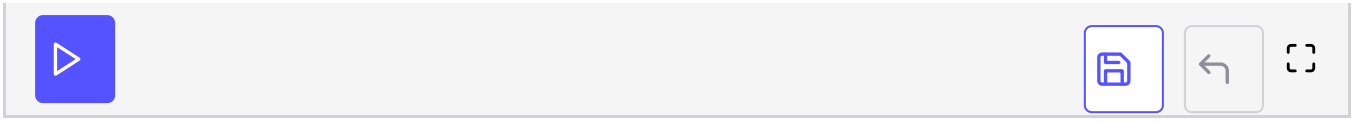
```
enum class [[nodiscard]] ErrorCode
{
  InvalidSelection,
  Undefined
};

variant<SelectionData, ErrorCode> CheckSelectionVer5(const ObjSelection &objList)
{
  if (!objList.IsValid())
    return ErrorCode::InvalidSelection;

  SelectionData out;
  // scan...
  return out;
}
```

As you see the code uses `std::variant` with two possible alternatives: either `SelectionData` or `ErrorCode.` It's almost like a pair, except that you'll always see one active value.

You can use the above implementation:

```
#include <iostream>
#include <tuple>
#include <optional>
#include <variant>
#include <vector>

class ObjSelection
{
  public:
  bool IsValid() const { return true; }
};

struct SelectionData
{
  bool anyCivilUnits { false };
  bool anyCombatUnits { false };
  int numAnimating { 0 };
};

enum class [[nodiscard]] ErrorCode
{
  InvalidSelection,
  Undefined
};

std::variant<SelectionData, ErrorCode> CheckSelectionVer5(const ObjSelection &objList)
{
  if (!objList.IsValid())
    return ErrorCode::InvalidSelection;

  SelectionData out;
  // scan...
  return out;
}

int main(){

  ObjSelection sel;

  bool anyCivilUnits = false;
  bool anyCombatUnits = false;
  int numAnimating = 0;

  if (auto retV5 = CheckSelectionVer5(sel); std::holds_alternative<SelectionData>(retV5))
  {
    std::cout << "ok..." << std::get<SelectionData>(retV5).numAnimating << '\n';
  }
  else
  {
    switch (std::get<ErrorCode>(retV5))
    {
      case ErrorCode::InvalidSelection:
        std::cerr << "Invalid Selection!\n";
        break;
      case ErrorCode::Undefined:
        std::cerr << "Undefined Error!\n";
        break;
    }
```

```
  }
}
```



As you can see, with `std::variant` you have even more information than when `std::optional` was used. You can return error codes and respond to possible failures.

Enough of Refactoring Now.
Let's have a quick overview of this module!