With std::optional

The lesson shows why std::optional is the right choice for nullable types.

WE'LL COVER THE FOLLOWING

- ^
- How to Refactor using std::optional?
- What are the advantages of the optional version?

How to Refactor using std::optional?

From the std::optional chapter:

std::optional is a wrapper type to express "null-able" types. It either
contains a value, or it's empty. It doesn't use any extra memory
allocation.

That seems to be the perfect choice for our code. We can remove the okvariable and rely on the semantics of the optional.

The new version of the code:

```
#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 }:
```

```
};
std::optional<SelectionData> CheckSelectionVer4(const ObjSelection &objList)
  if (!objList.IsValid())
    return std::nullopt;
  SelectionData out;
  // scan...
  return out;
}
int main(){
  ObjSelection sel;
  bool anyCivilUnits = false;
  bool anyCombatUnits = false;
  int numAnimating = 0;
  // for the caller site:
  if (auto ret = CheckSelectionVer4(sel); ret.has_value())
    // access via *ret or even ret->
    std::cout << "ok..." << ret->numAnimating << '\n';</pre>
  }
}
```







[]

What are the advantages of the optional version?

- Clean and Expressive Form optional expresses nullable types.
- **Efficiency** implementations of optional are not permitted to use additional storage, such as dynamic memory. The contained value shall be allocated in a region of the optional storage suitably aligned for the type T.

That's all for optional std, now its time to look into std::variant version.