Lab Report for Object-oriented Programming course

Lab 3: Linkage

Wang, Chen
16307110064
School of Software
Fudan University

May 12, 2019

Contents

Chapter 1

Understanding Internal and External Linkage

1.1 Requirements

In this part, we are going to analyze the running result of the following short program and will try to make some subtle modifications whose change will result in the complete change of the result. The codes are shown below.

1.1.1 header.h

```
#ifndef LAB3_HEAER_H                    1
#define LAB3_HEAER_H                    2
                                        3
static int variable = 0;                4
extern int variable2;                   5
#endif //LAB3_HEAER_H                   6
```

1.1.2 file1.h

```
#ifndef LAB3_FILE1_H                    1
#define LAB3_FILE1_H                    2
void function1();                       3
                                        4
#endif //LAB3_FILE1_H                   5
```

1.1.3 file2.h

```
#ifndef LAB3_FILE2_H                    1
#define LAB3_FILE2_H                    2
void function2();                       3
                                        4
#endif //LAB3_FILE2_H                   5
```

1.1.4 file1.cpp

```
#include "header.h"
```

1.1.5 file2.cpp

```
#include "header.h"
void function2() {
    variable = 2;
    variable2 = 2;

}
```

1.1.6 main.cpp

```
#include <iostream>
#include "header.h"
#include "file1.h"
#include "file2.h"
int variable2;

int main() {
    function1();
    function2();

    std::cout << variable << std::endl;
    std::cout << variable2 << std::endl;
    return 0;
}
```

1.2 Execution result

The result of the execution is shown in the Figure 1.1 below.

1.3 Analysis for the result of static variable

From the C++ language standards, we can know that the *static* keyword has different implications and effects in two different occasions. In this project, the keyword appears before the definition of the global variable outside from any methods. In this case, the *static* keyword has the effect that the variable is only accessible in the specified file, i.e. having a *file scope*.

More specifically, in this project there are three **cpp** files in this project, each of them including the *header.h* header file where a static variable is declared. In this case, each of the three files has a version of the static variable *variable*. Therefore, the methods *function1* and *function2* will have no effect to the variable *variable* in the *main* method. Hence, the printing result is unchanged.

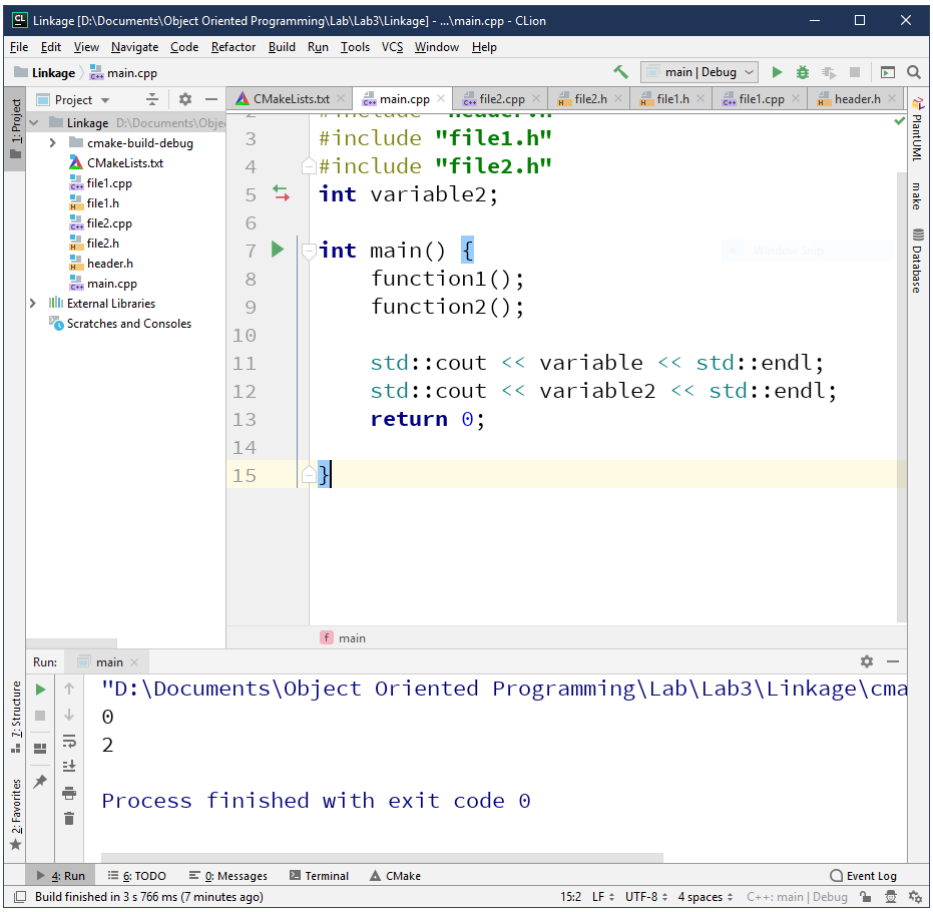

Figure 1.1: Execute Results

1.4 Analysis for the result of global variable

As to another part of this lab, the printing result of *variable2*, is a typical example of a global variable. Nevertheless, one slightly point making the analysis more complex is that the *external* keyword is used to make the global variable declared and defined at different places.

More specifically, in this project, the variable *variable2* is declared in *header.h* with an *external* keyword, indicating that this variable will be defined somewhere else in this project. Then in the file *main.cpp* this variable is defined. Therefore, this is a typical example of global variable and all the functions are modifying the same version of the variable. Therefore, the *main* function will print the result after being changed in the other functions.

1.5 Task 1.2: Duplicate variable definition

The result of the change is shown in the Figure 1.2 below, indicating an compiling error.

In this example, we have two places both defining the variable *variable2*. Therefore, the compiler will report a duplicate definition error.

1.6 Task 1.3: External declaration without definition

The result of the change is shown in the Figure 1.3 below, indicating an compiling error.

As per have stated above, an *extern* keyword indicates that this variable is defined somewhere else in this file, hence it will cause a compiling error if no definition is made through out the project file.

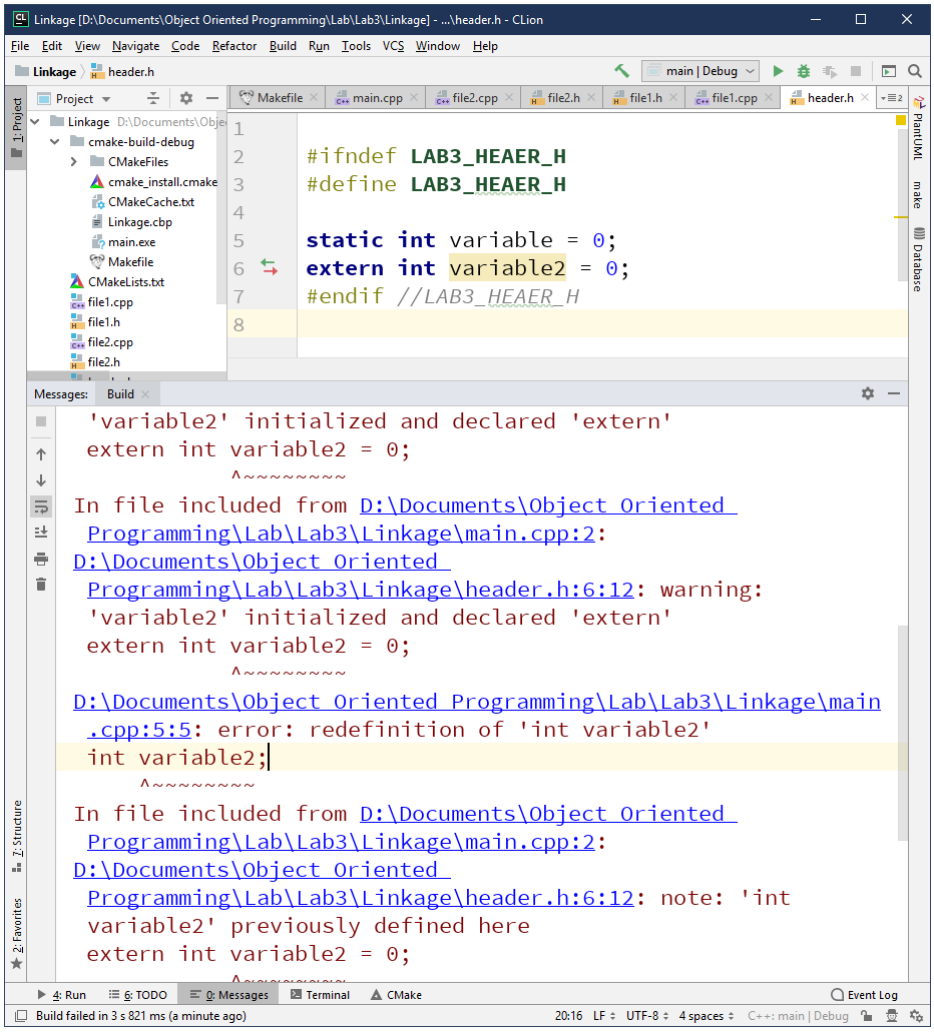

Figure 1.2: Execute Results of Task 1.2

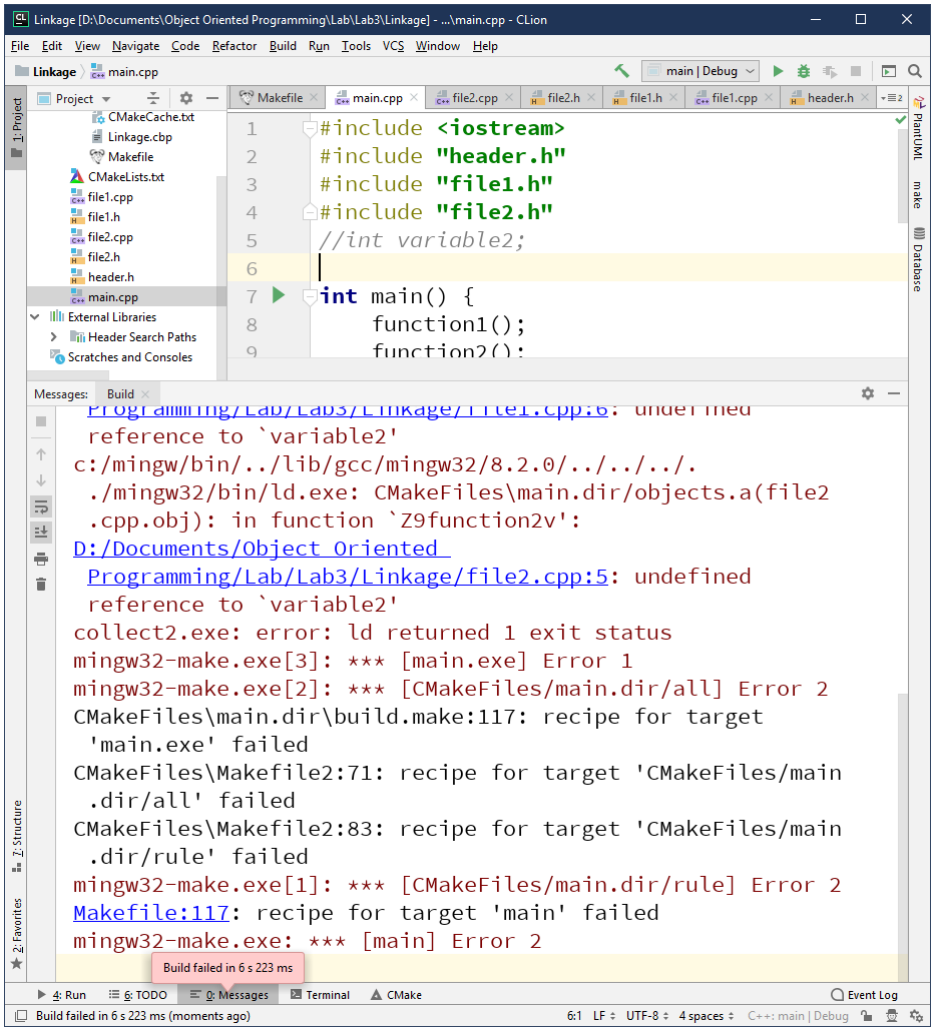

Figure 1.3: Execute Results of Task 1.3

Chapter 2

Understanding Name Space

2.1 Task 2.2 Duplicate variable naming in separate name spaces

In this task, we are going to use the name space mechanism, create two variables with the same name but different values and print them.

Here in my implementation, I declared two name spaces called **a** and **b** separately and then a variable named **a** in each name space but different value. Then in the *main* method, these two variables are printed separately, we can see the result that they have different values printed. The result is shown in the Figure 2.1 below.

2.2 Task 2.2 Extended namespace

In this task, I will need to duplicately declare the namespace of the same name in two files. The space of the first file defines "ab=1; cd=2", and the space of the second file is defined, "ab=3;bc=4" Finally, print the value in the namespace in the main function.

Here in my implementation, I declared a name space called **a** in two separate files and then assigned different values to the variables as stated. Then in the *main* method, these three variables are printed separately, we can see the result that this project cannot be compiled due to duplicate variable definition in the same name space. The result is shown in the Figure 2.2 below.

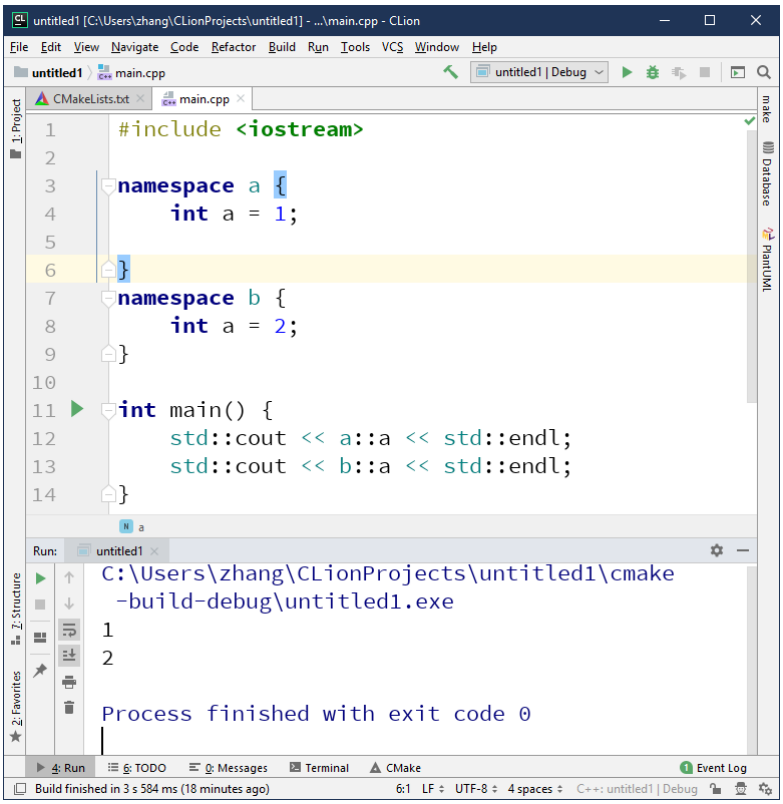

Figure 2.1: Execute Results of Task 2.1

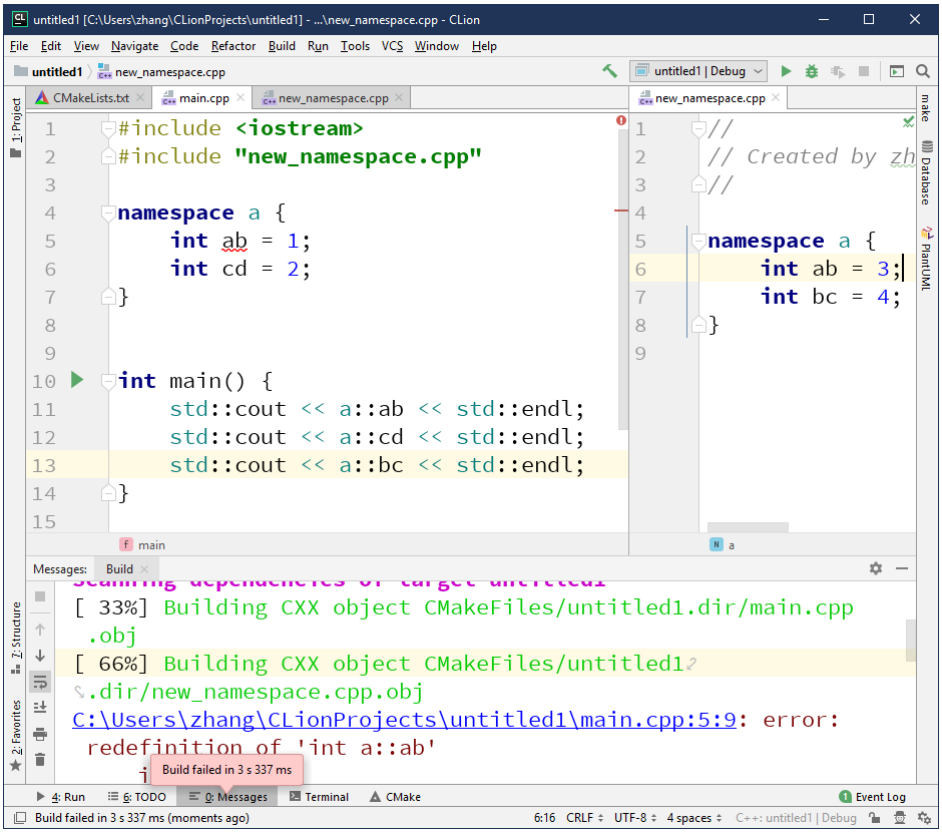

Figure 2.2: Execute Results of Task 2.2

Chapter 3

Compare with Typescript Name Space

3.1 Modules in TypeScript

***A note about terminology**: It's important to note that in TypeScript 1.5, the nomenclature has changed. "Internal modules" are now "namespaces". "External modules" are now simply "modules", as to align with ECMAScript 2015*[1] *'s terminology, (namely that* `module X {` *is equivalent to the now-preferred* `namespace X {` *).*

3.1.1 Introduction

Starting with ECMAScript 2015, JavaScript has a concept of modules. TypeScript shares this concept.

Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the *export* forms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the *import* forms.

Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level. Modules import one another using a module loader. At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it. Well-known modules loaders used in JavaScript are the CommonJS module loader for Node.js and require.js for Web applications.

In TypeScript, just as in ECMAScript 2015, any file containing a top-level *import* or *export* is considered a module. Conversely, a file without any top-level *import* or *export* declarations is treated as a script whose contents are available in the global scope (and therefore to modules as well).

[1] Ecma International. (2019, May 5). ECMAScript 2015 Language Specification – ECMA-262 6th Edition. In *Ecma International*. Retrieved 08:23, May 11, 2019, from `http://www.ecma-international.org/ecma-262/6.0/`

3.1.2 Export

Exporting a declaration

Any declaration (such as a variable, function, class, type alias, or interface) can be exported by adding the *export* keyword.

Export statements

Export statements are handy when exports need to be renamed for consumers.

Re-exports

Often modules extend other modules, and partially expose some of their features. A re-export does not import it locally, or introduce a local variable.

Optionally, a module can wrap one or more modules and combine all their exports using *export * from "module"* syntax.

3.1.3 Import

Importing is just about as easy as exporting from a module. Importing an exported declaration is done through using one of the *import* forms below:

1. Import a single export from a module
2. Import the entire module into a single variable, and use it to access the module exports
3. Import a module for side-effects only

Though not recommended practice, some modules set up some global state that can be used by other modules. These modules may not have any exports, or the consumer is not interested in any of their exports.

3.1.4 Default exports

Each module can optionally export a *default* export. Default exports are marked with the keyword *default*; and there can only be one *default* export per module. *default* exports are imported using a different import form. *default* exports are really handy. For instance, a library like jQuery might have a default export of *jQuery* or $, which we'd probably also import under the name $ or jQuery.

Classes and function declarations can be authored directly as default exports. Default export class and function declaration names are optional.

3.1.5 *export =* and *import = require()*

Both CommonJS and AMD generally have the concept of an *exports* object which contains all exports from a module.

They also support replacing the *exports* object with a custom single object. Default exports are meant to act as a replacement for this behavior; however, the two are incompatible. TypeScript supports *export =* to model the traditional CommonJS and AMD workflow.

The *export =* syntax specifies a single object that is exported from the module. This can be a class, interface, namespace, function, or enum.

When exporting a module using *export =*, TypeScript-specific *import module = require("module")* must be used to import the module.

3.1.6 Code Generation for Modules

Depending on the module target specified during compilation, the compiler will generate appropriate code for Node.js (**CommonJS**), require.js (**AMD**), **UMD**, **SystemJS**, or **ECMAScript 2015 native modules** (ES6) module-loading systems. For more information on what the *define*, *require* and *register* calls in the generated code do, consult the documentation for each module loader.

3.1.7 Optional Module Loading and Other Advanced Loading Scenarios

In some cases, you may want to only load a module under some conditions. In TypeScript, we can use the pattern shown below to implement this and other advanced loading scenarios to directly invoke the module loaders without losing type safety.

The compiler detects whether each module is used in the emitted JavaScript. If a module identifier is only ever used as part of a type annotations and never as an expression, then no *require* call is emitted for that module. This elision of unused references is a good performance optimization, and also allows for optional loading of those modules. The core idea of the pattern is that the *import id = require("...")* statement gives us access to the types exposed by the module. The module loader is invoked (through *require*) dynamically, as shown in the if blocks below. This leverages the reference-elision optimization so that the module is only loaded when needed. For this pattern to work, it's important that the symbol defined via an *import* is only used in type positions (i.e. never in a position that would be emitted into the JavaScript). To maintain type safety, we can use the *typeof* keyword. The *typeof* keyword, when used in a type position, produces the type of a value, in this case the type of the module.

3.2 Name space in TypeScript

3.2.1 Introduction

In this subsection I will try to outline the various ways to organize the code using namespaces (previously "internal modules") in TypeScript. As per alluded in previous section's note about terminology, "internal modules" are now referred to as "namespaces". Additionally, anywhere the module keyword was used when declaring an internal module, the namespace keyword can and should be used instead. This avoids confusing new users by overloading them with similarly named terms.

3.2.2 Namespacing

As we have more and more methods of the same or of similiar functions, we're going to want to have some kind of organization scheme so that we can keep track of our types and not worry about name collisions with other objects. Instead of putting lots of different names into the global namespace, we can wrap up our objects into a namespace.

If we want the interfaces and classes in the namespace to be visible outside the namespace, we can preface them with export. Conversely, the implementation details can be left unexported and will not be visible to code outside the namespace.

3.2.3 Splitting Across Files

As our application grows, we'll want to split the code across multiple files to make it easier to maintain.

Multi-file namespaces

We can split our namespace across many files. Even though the files are separate, they can each contribute to the same namespace and can be consumed as if they were all defined in one place. Because there are dependencies between files, we'll add reference tags to tell the compiler about the relationships between the files.

3.2.4 Aliases

Another way that you can simplify working with namespaces is to use *import q = x.y.z* to create shorter names for commonly-used objects. Not to be confused with the *import x = require("name")* syntax used to load modules, this syntax simply creates an alias for the specified symbol. We can use these sorts of imports (commonly referred to as aliases) for any kind of identifier, including objects created from module imports.

Notice that we don't use the *require* keyword; instead we assign directly from the qualified name of the symbol we're importing. This is similar to using *var*, but also works on the type and namespace meanings of the imported symbol. Importantly, for values, *import* is a distinct reference from the original symbol, so changes to an aliased *var* will not be reflected in the original variable.

Bibliography

[1] Wikipedia contributors. (2019, May 5). TypeScript. In *Wikipedia, The Free Encyclopedia*. Retrieved 08:23, May 11, 2019, from `https://en.wikipedia.org/w/index.php?title=TypeScript&oldid=895568490`

[2] Wikipedia contributors. (2019, May 9). JavaScript. In *Wikipedia, The Free Encyclopedia*. Retrieved 08:24, May 11, 2019, from `https://en.wikipedia.org/w/index.php?title=JavaScript&oldid=896215789`

[3] Wikipedia contributors. (2019, April 24). C++. In *Wikipedia, The Free Encyclopedia*. Retrieved 08:25, May 11, 2019, from `https://en.wikipedia.org/w/index.php?title=C%2B%2B&oldid=893975016`

[4] Wikipedia contributors. (2019, February 27). Object-oriented programming. In *Wikipedia, The Free Encyclopedia*. Retrieved 10:22, March 23, 2019, from `https://en.wikipedia.org/w/index.php?title=Object-oriented_programming&oldid=885274966`

[5] Wikipedia contributors. (2019, March 18). Namespace. In *Wikipedia, The Free Encyclopedia*. Retrieved 08:27, May 11, 2019, from `https://en.wikipedia.org/w/index.php?title=Namespace&oldid=888324103`