Name Scopes

In this lesson, we discuss the name scopes.

WE'LL COVER THE FOLLOWING

Name scope

Any name is accessible from the point where it has been defined to the point where its scope ends, as well as in all of the scopes that its scope includes. In this regard, every scope defines a **name scope**.
Names are not available beyond the end of their scope:

```
import std.stdio;

void main() {
    bool aCondition = true;
    int outer;

    if (aCondition) { // ← curly bracket starts a new scope
        int inner = 1;
        outer = 2;       // ← 'outer' is available here
    } // ← 'inner' is not available beyond this point

    inner = 3; // ← compilation ERROR
               // 'inner' is not available in the outer scope
}
```

Error due to name scope

Because `inner` is defined within the scope of the `if` condition, it is available *only* in that scope. On the other hand, `outer` is available in both the if block scope and main function scope.

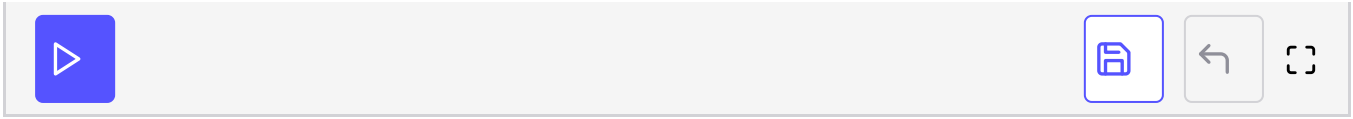
It is not legal to define the same name in an inner scope:

```
size_t length = oddNumbers.length;

if (aCondition) {
    size_t length = primeNumbers.length; // ← compilation ERROR
}
```

Defining names closest to their first use

As we have been doing in all of the programs so far, variables must be defined before their first use:

```
import std.stdio;

void main() {
    writeln(number);    // ← compilation ERROR
                        //   number is not known

    int number = 42;
}
```



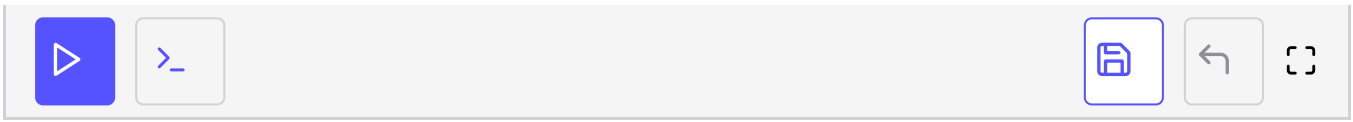
Variable must be defined before first use

For the code to be acceptable by the compiler, the number must be defined before it is used with `writeln`. Although there is no restriction on how many lines earlier it should be defined, it is accepted as good programming practice that variables be defined closest to where they are first used.

Let's see this in a program that prints the average of the numbers that it takes from the user. Programmers who are experienced in some other programming languages may be used to defining variables at top of the scopes as seen below:

```
import std.stdio;

void main() {
    int count;            // ← HERE
    int[] numbers;        // ← HERE
    double averageValue;  // ← HERE

    write("How many numbers are there? ");

    readf(" %s", &count);
```

```
    if (count >= 1) {
        numbers.length = count;

        // ... assume the calculation is here ...

    } else {
        writeln("ERROR: You must enter at least one possitive number!");
    }
}
```

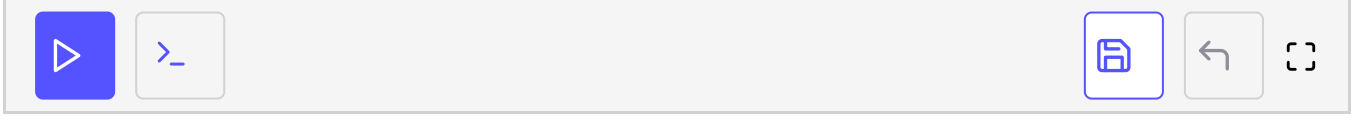


Variables defined at top of scope

> **Note:** You can ignore the concept of arrays i.e. `int[] numbers` for now, as arrays will be explained in the next chapter.

In contrast to the code above, the one given below defines the variables when they are actually needed in the program:

```
import std.stdio;

void main() {
    write("How many numbers are there? ");

    int count;
    readf(" %s", &count);

    if (count >= 1) {
        int[] numbers;
        numbers.length = count;

        double averageValue;

        // ... assume that the calculation is here ...

    } else {
        writeln("ERROR: You must enter at least one number!");
    }
}
```



Variables defined when about to be used

Although defining all of the variables at the top may look better structurally, there are several benefits of defining them as late as possible:

- **Speed:** Every variable definition tends to add a small speed cost to the program. As every variable is initialized in D, defining variables at the top will result in them always being initialized, even if they are only sometimes used later, wasting resources.
- **Risk of mistakes:** Every line between the definition and use of a variable carries a higher risk of programming mistakes. As an example, consider a variable using the common name *length*. It is possible to confuse that variable with some other length and use it inadvertently before reaching the line of its first intended use. When that line is finally reached, the variable may no longer have the desired value.
- **Readability:** As the number of lines in a scope increases, it becomes more likely that the definition of a variable is too far up in the source code, forcing the programmer to scroll back in order to look at its definition.
- **Code maintenance:** Source code is in constant modification and improvement: new features are added, old features are removed, bugs are fixed, etc. These changes sometimes make it necessary to extract a group of lines altogether into a new function.

For example, in the code above that followed this guideline, all of the lines within the `if` statement can be moved to a new function in the program. When that happens, having all of the variables defined close to the lines that use them makes it easier to move them as a coherent bunch.

On the other hand, when the variables are always defined at the top, if some lines ever need to be moved, the variables that are used in those lines must be identified one by one.

In the next lesson, we will explore another type of loop, i.e, the `for` loop.