for Loop

This lesson explains the for loop, its syntax and implementation in D.

WE'LL COVER THE FOLLOWING

`for` loop

The `for` loop serves the same purpose as the while loop. `for` makes it possible to put the definitions and expressions concerning the loop's iteration on the same line.
Although `for` is used much less than `foreach` in practice, it is important to understand the for loop first. We will see `foreach` in a later chapter.

The sections of the `while` loop

The `while` loop evaluates the loop condition and continues executing the loop as long as that condition is true. For example, a loop to print the numbers between 1 and 10 may check the condition less than 11. To be compilable as D code, `number` must have been defined before its first use:

```
int number = 1;
while (number < 11)
```

Iterating the loop can be achieved by incrementing the number at the end of the loop:

```
++number;
```

Finally, there is the actual work within the loop body:

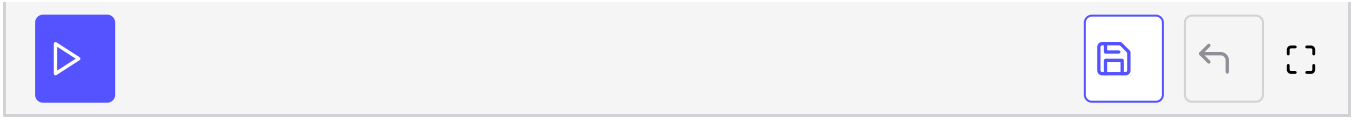
```
writeln(number);
```

These four sections can be combined into the desired loop as follows:

```
import std.stdio;

void main() {
    int number = 1;       // ← preparation

    while (number < 11) { // ← condition check
        writeln(number);  // ← actual work
        ++number;         // ← iteration
    }
}
```



The sections of the `while` loop are executed in the following order during the iteration of the while loop:

```
preparation
condition check
actual work
iteration
condition check
actual work
iteration
...
```

A `break` statement or an error can terminate the loop as well.

The sections of the `for` loop

The `for` loop brings three of these sections onto a single line. They are written within the parentheses of the for loop and separated by semicolons. The loop body contains only the actual work:

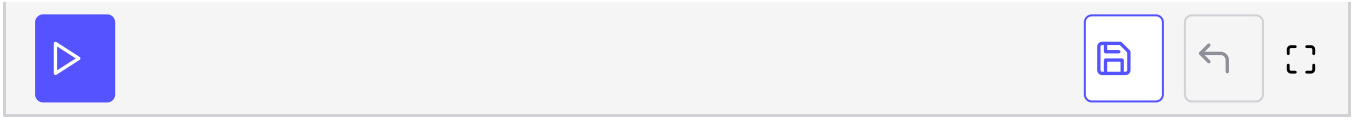
```
for (/* preparation */; /* condition check */; /* iteration */) {
    /* actual work */
}
```

Here is the same code written as a `for` loop:

Here is the same code written as a `for` loop:

```
import std.stdio;

void main() {
    for (int number = 1; number < 11; ++number) {
        writeln(number);
    }
}
```



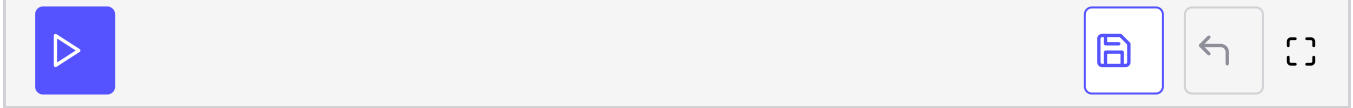
The benefits of the `for` loop are more obvious when the loop body has a large number of statements. The expression that increments the loop variable is visible on the `for` line instead of being mixed with the other statements of the loop. It is also clearer that the declared variable is used only as part of the loop, and not by any other surrounding code.

The sections of the `for` loop are executed in the same order as those in the `while` loop. The `break` and `continue` statements also work exactly the same way as they do in the `while` loop. The only difference between the `while` and `for` loops is the name scope of the loop variable. This is explained below.

Although very common, the iteration variable need not be an integer, nor it is modified only by increments. For example, the following loop is used to print the halves of the previous floating point values:

```
import std.stdio;

void main() {
    for (double value = 1; value > 0.001; value /= 2) {
        writeln(value);
    }
}
```



> **Note:** The information above is technically incorrect but better captures the spirit of how the `for` loop is used in practice. In reality, D's `for` loop does not have three sections that are separated by semicolons. It has two sections, the first of which consists of the preparation and the loop condition.

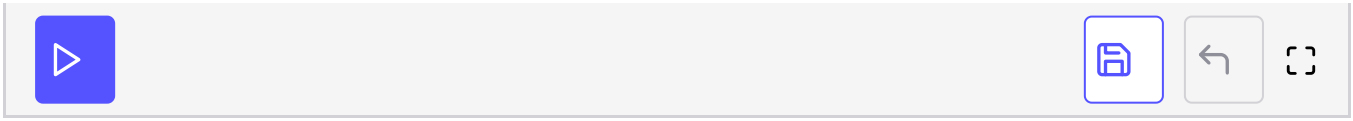
Without getting into the details of this syntax, here is how to define two variables of different types in the preparation section:

```
import std.stdio;

void main() {

    for ({ int i = 0; double d = 0.5; } i < 10; ++i) {
        writeln("i: ", i, ", d: ", d);
        d /= 2;
    }

}
```



> **Note:** The preparation section is the area within the highlighted curly brackets, and there is no semicolon between the preparation section and the condition section.

The sections may be empty

All three of the `for` loop sections may be left empty:

- Sometimes a special loop variable is not needed, possibly because of an already- defined variable would be used.
- Sometimes the loop would be exited by a `break` statement, instead of by relying on the loop condition.
- Sometimes the iteration expressions depend on certain conditions that would be checked within the loop body.

When all of the sections are empty, the `for` loop runs forever:

```
for ( ; ; ) {
// ...
}
```

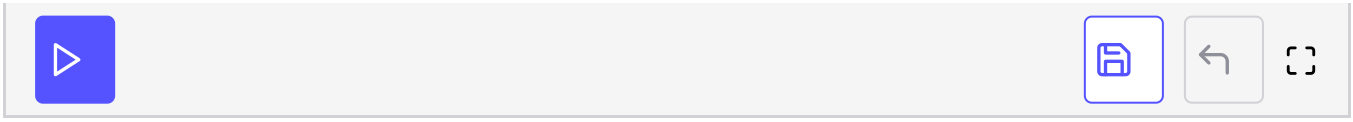
Such a loop may be designed to never end or end with a `break` statement.

The name scope of the loop variable

The only difference between the `for` and `while` loops is the name scope of the variable defined during loop preparation. The variable is accessible only within the `for` loop, not outside of it:

```
import std.stdio;

void main() {
    for (int i = 0; i < 5; ++i) {
        // ...
    }
    writeln(i);   // ← compilation ERROR
                  //   i is not accessible here
}
```



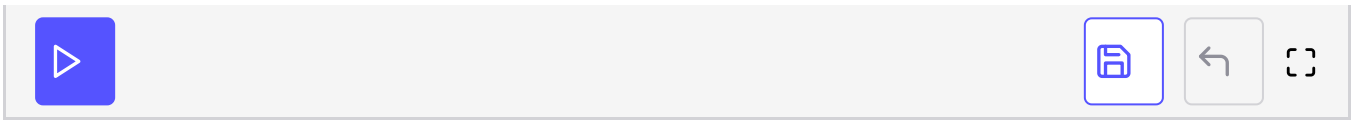
In contrast, when using a `while` loop, the variable is defined in the same name scope as that which contains the loop, and therefore, the name is accessible even after the loop:

```
import std.stdio;

void main() {

    int i = 0;
    while (i < 5) { // ...
    ++i;
    }
    writeln(i); // ← 'i' is accessible here

}
```



We have seen the benefits of defining names closest to their first use in the previous lesson. Based on the same rationale, the smaller the name scope of a variable the better. In this regard, when the loop variable is not needed outside the loop, a `for` loop is better than a `while` loop.

In the next lesson, you will find a coding challenge to print a parallelogram pattern using for loops.