

# Control Flow in Arduino Programming

---

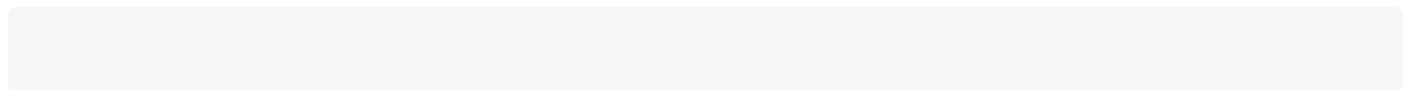
The term *control flow* may seem daunting at first, but at its core its essentially a group of statements and structures that govern the interactivity and processes of programs. You've already worked with one type of control flow in your Arduino experimentation- *functions*.

Two more types of control flow will be covered in this section, *conditional logic* and *looping*. This may seem like a lot of things to understand at first but are rather straightforward when they're broken down into component parts.

## The Block Statement

---

The first, and most basic, control flow statement is the *block statement*. You've actually already integrated these into your code when you've worked with *functions*. Consider this code contained within the **void loop()** function (you're going to have this code memorized by the end of the camp!):



With regard to the above example, you can think of the block statement as containing, or controlling, the code within the function. In this particular example, this would be the code that controls the on/off messages of the LED as well as the delays. As you move into the other types of control flow covered in this lesson, you'll see this form used each time. Just remember that code contained within { } is chunked

together for a reason.

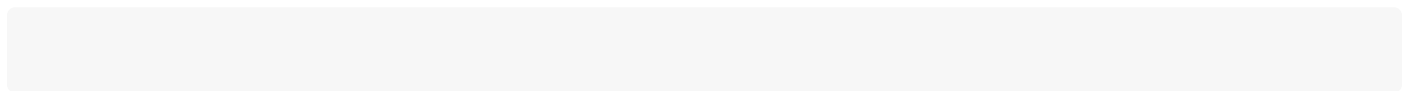
# Conditional Statements

---

*Conditional statements* are one of the core pillars of programming, whether it be for the web, video games or other applications. Developing a practical and functional understanding of working with conditional statements will greatly enhance the level of interactivity that you're able to develop and include in your code.

Let's take a look at two examples. The first is stripped down to demonstrate a concept, and the second is a snippet of functional Arduino code.

## Example one:



In this example, the `if` statement is evaluating a condition that is contained within the `( )` following the word *if*. The statement that is within the `( )` needs to be able to be evaluated as a Boolean value, meaning it must either evaluate to *true* or *false*.

In the above example, the statement would evaluate as true *if the value of the variable x is greater than 10*. If this returns true, then the code within the first set of `{ }` is executed.

This particular form includes an `else { }` block statement as well. This is the code that executes if the condition is false. Since the statement contained within the `( )` is Boolean, meaning it is either true **or** false, only one block statement of code will execute.

## Example two:

This is a snippet of code using a concept that you'll integrate into an upcoming project.



This example is checking to the state of a button to determine whether it is on or off. The setup is similar to the first example, as there is an expression inside the ( ) being checked.

The *buttonState* variable is checked to see if it is in a HIGH (on) state, and if that condition evaluates as true, then a *digitalWrite* message is sent to the board to turn on the LED.

In the else { } statement, which will execute when *buttonState* is NOT equal to HIGH, then sends a message to turn the LED off.

Notice the inclusion of *variables* in the above examples. Variables and conditional statements are commonly used together to execute commands that are determined by checking a particular state.

Now that you have an understanding of the structure of conditional statements, it's time to introduce another key concept, *operators*.

## Basic Operators

---

This section includes some of the basic operators that you'll use in your Arduino projects. There are lots of resources available online, particularly on the [Arduino Reference Page](#) for a more in-depth list if you're wanting to develop a deeper understanding of this tool.

## Assignment Operators

You've already worked with the first type of operator, which is the *assignment operator*. These are operators that assign a value to something on the left based on the evaluation of what's on the right. Here are some examples:

There are several more, but these are some of the most common that you'll use for now.

## Comparison Operators

Another type of operator is the *comparison operator*. This was used in the example above where the value of a "name" variable is checked to see if it is empty. These are commonly used in *if...else statements* because they return true or false based on the evaluation. Many of these are the same as what you've likely used in math classes. Here are some examples:

- Equal (==) : This returns true if the values are equal.
- Not equal (!=) : This returns true if the values aren't equal.
- Greater than (>) : This returns true if the value on the left is larger than the value on the right.
- Greater than or equal to (>=) : This returns true if the value on the left is larger or equal to the value on the right.
- Less than (<) : This returns true if the value on the left is less than the value on the right.
- Less than or equal to (<=) : This returns true if the value on the left is less than or equal to the value on the right.

One thing that you should remember is that when evaluating equality within conditional statements, you want to use the (==) operator. If you use only a single (=), Arduino thinks that you're attempting to assign a value and it can become confused about how to handle the statement.

There are other operator types that you'll encounter, but for now focus on developing an understanding of how these operator types are used with conditional statements to provide control flow to your code.

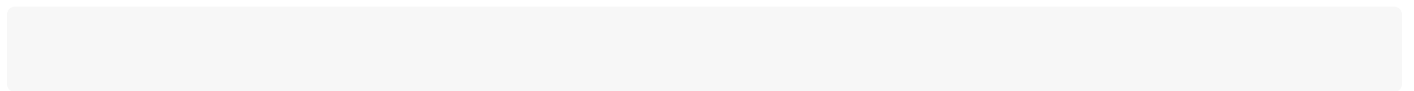
## Looping and Iterating

---

Looping is another commonly used control flow technique, and the last one that this lesson will cover. There are several types of loops and iterators in JavaScript, but this section includes the most basic and most common in Arduino programming: the *For Loop*.

Understanding the structure and purpose of the *For Loop* is important because it is a concept that is used in other languages as well.

Let's first consider an example, and then break down the code:



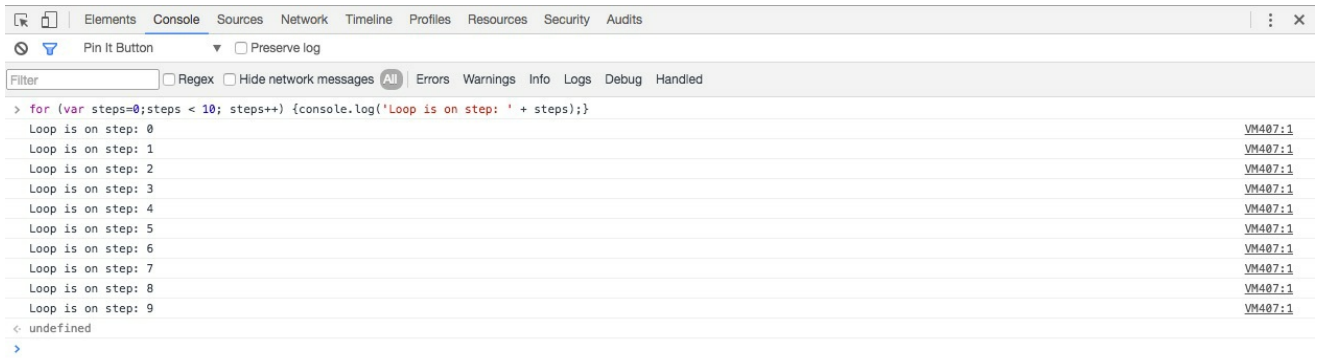
### Code Breakdown:

Let's first look at the expressions contained within the ( ).

- The first part, "var steps = 0;" is the initial value. The variable *i* is commonly used for this, but it doesn't need to be. The above example uses the variable *steps* for clarity. This first expression is the *starting point* of the loop. For this, the step counter begins at 0.
- The second part, "steps < 10" is the limit of the loop. The loop will continue until this evaluates as false. In this case, the loop will continue until the *steps* variable is *no longer less than 10*.
- The last part, "steps++" is the iterator of the loop. The "++" is shorthand for "steps = steps + 1" This determines how the steps variable increments with each pass through the loop.

The example behaves in the following way:

1. The steps variable starts with a value of 0.
2. The code inside the block statement executes and the following is logged to console:

A screenshot of a web browser's developer console. The 'Console' tab is selected, showing a series of log messages. The messages are: '> for (var steps=0; steps < 10; steps++) {console.log('Loop is on step: ' + steps);}', 'Loop is on step: 0', 'Loop is on step: 1', 'Loop is on step: 2', 'Loop is on step: 3', 'Loop is on step: 4', 'Loop is on step: 5', 'Loop is on step: 6', 'Loop is on step: 7', 'Loop is on step: 8', 'Loop is on step: 9', and 'undefined'. Each log message is followed by a link to the source file 'VM407:1'. The console also shows a 'Filter' input, a 'Pin It Button', and a 'Preserve log' checkbox. The 'All' filter is selected, and the 'Handled' column is visible.

```
> for (var steps=0; steps < 10; steps++) {console.log('Loop is on step: ' + steps);}
Loop is on step: 0
Loop is on step: 1
Loop is on step: 2
Loop is on step: 3
Loop is on step: 4
Loop is on step: 5
Loop is on step: 6
Loop is on step: 7
Loop is on step: 8
Loop is on step: 9
undefined
```

3. After the code in the block statement executes, the iterator progresses. In this example, the steps counter increments by 1 each time the loop executes.
4. The loop will run 10 times, beginning at 0 and ending at 9.
5. Note that the variable *steps* is used within the block statement to log the current loop step. You can use the iterator variable just like you could with any other variable *within the scope of the for loop*.