Inputs, Comparisons & Branching

We saw how to read data from the serial port, but we also saw references to I/O 'pins' and something called GPIO, which is an acronym for General Purpose Input Output. We also saw how to turn on the built-in LED by writing a 1 to it. That LED is just connected to a GPIO pin, specifically pin #13. As you can see by looking at your Nano, all of the pins are labeled. Some have longer and somewhat obvious labels like 'GND' and '5V', while the others are just A or D and a number. We can reference that letter+number combination in our program to access these pins, then set a value on them using the statement `digitalWrite()`. If we want to read the value of a pin and see if it is `HIGH` (1) or `LOW` (0), we can use the statement `digitalRead()`.

The D# pins are digital only pins, and we can actually reference them in the Arduino environment using the *number only*. The A# pins are Analog pins, which means they have some special capabilities that we will look at later. They can also do everything the digital only pins can do, but they must be referenced using the full A# name.

Thus if we create a circuit between a switch, sensor, or other device and one of our pins, we can control and/or monitor said circuit/device.

We now know how to read and write to GPIO pins, and we know how to store values in variables and perform various calculations on them. But we haven't learned how to add any actual logic to our programs yet.

The most fundamental component needed to start building some logic is the ***comparison***. Comparisons are done using the comparison operators, such as:

Equal To	==
Not Equal To	!=
Greater than	>
Less than	<

A complete list is available in the Appendix with the other operators.

Now we can see if a variable is equal to a particular value or another variable, but how do we use that information? Earlier we looked at program flow and noted that the processor executes one instruction after another linearly, but that one type of instruction is to move to another set of instructions — what if you could control exactly where in the list of instructions execution would just based on a conditional? This is exactly what the 'if' statement does, and it's called ***conditional branching***.

In our programming language, the concept is simple: If a certain condition is true, then perform a certain action. And the syntax is pretty straightforward as well:

```
if ( x == 42 ) {
  Serial.println("Target reached");
}
```

In this case when the value of the variable x is equal to the number 42, then the string 'Target reached" will be printed to the serial link. The block could contain multiple statements, including more '`if`' statements as needed.

The standard '`if`' can be augmented with an '`else`' clause, which can specify a different block to be executed if the conditional is false.

```
if ( x == 42 ) {
  Serial.println("Target reached");
} else {
  Serial.println("Not yet at target");
}
```

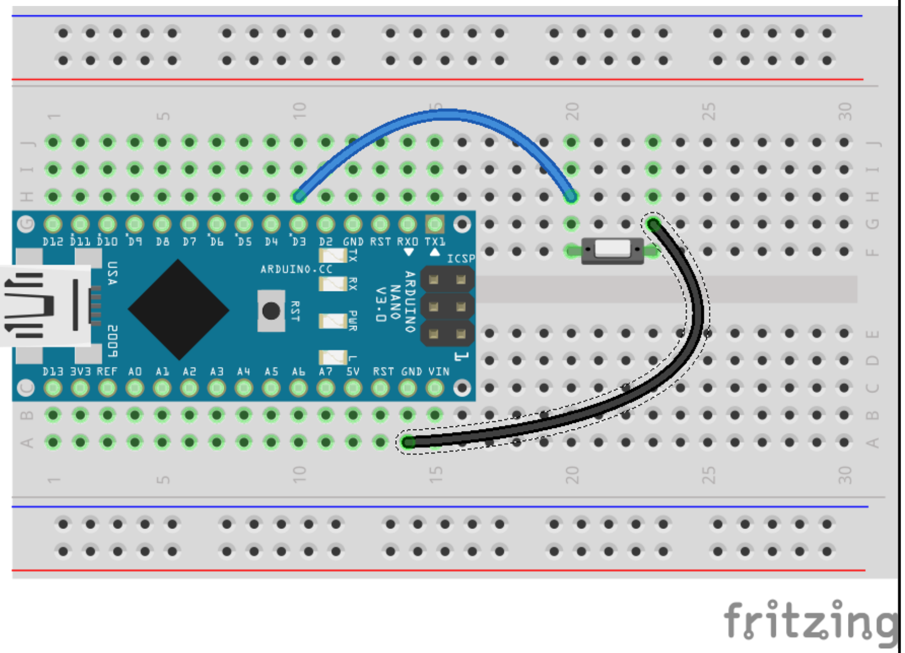
Of course, the conditional can be any combination of operators, as long as the entire combination can be evaluated as either *True* or *False*. It is common to join multiple comparison operators with the logical operators 'and' (&&) , 'or' (||) and 'not' (!). The code snippet below will only print "Target reached" when both x is 42 or greater and y is 42 or less. It will print "Cannot fire" if either the value of `ammo` is 0 or the value of `weaponOffline` is 1.

```
if ( x > 41 && y < 43) {
  Serial.println("Target reached");
  if ( ammo == 0 || weaponOffline == 1 ) {
      Serial.println("Cannot fire");
  }
}
```

A common pitfall is to accidentally use the assignment operator (=) in a comparison instead of the equal to operator (==). This does not result in a compile time or syntax error, because it is perfectly valid (though rarely useful) to make an assignment in conjunction with a comparison. However, not only will the assignment always evaluate as true, but now the variable you were using in the comparison is now set to the value you were comparing against.

Combining conditional branching and reading input from things connected to the GPIO pins, will allow our programs to respond dynamically to the physical world, as we'll see in this experiment.

Experiment: Rube Goldberg LED.

Components	**Wiring Diagram**
✓Computer ✓Arduino IDE ✓microswitch	

D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 GND RST RX0 TX1
D13 3V3 REF A0 A1 A2 A3 A4 A5 A6 A7 5V RST GND VIN
ARDUINO.CC
ARDUINO NANO V3.0
ICSP
USA
2009
RST
TX
RX
PWR
L
fritzing

Connection Instructions

Connect a pushbutton from your kit such that one pin goes to Ground (GND) and the other to Digital Pin #3 (D3), as shown in the diagram. Where exactly you place it and the wires doesn't matter. Just remember how the breadboard slots are connected (horizontally on the red and blue, vertically from A-E and F-J)

Make sure the wires are out of the way so the button is easy to push.

Double check that the wires are in the slots you think they are and that a circuit will be created from D3 to GND when the button is pushed.

Sketch(es)	RGLED.ino

Analysis Questions

Clearly this experiment is contrived and the LED could simply be connected with the button to a power source, but can you think of some situations where 'soft' or 'by wire' controls such as this might be beneficial?

Programming Tasks

This program is only a stub. Fill in the missing components such that when the button on the microswitch is depressed, the Nano's built in LED is illuminated, and it is dark when the button is not pressed.

Extra tasks if time permits:

- can you do achieve the same result without an 'if' statement at all?
- try to make each push of the button change the LED from on to off for a couple of seconds then back on.