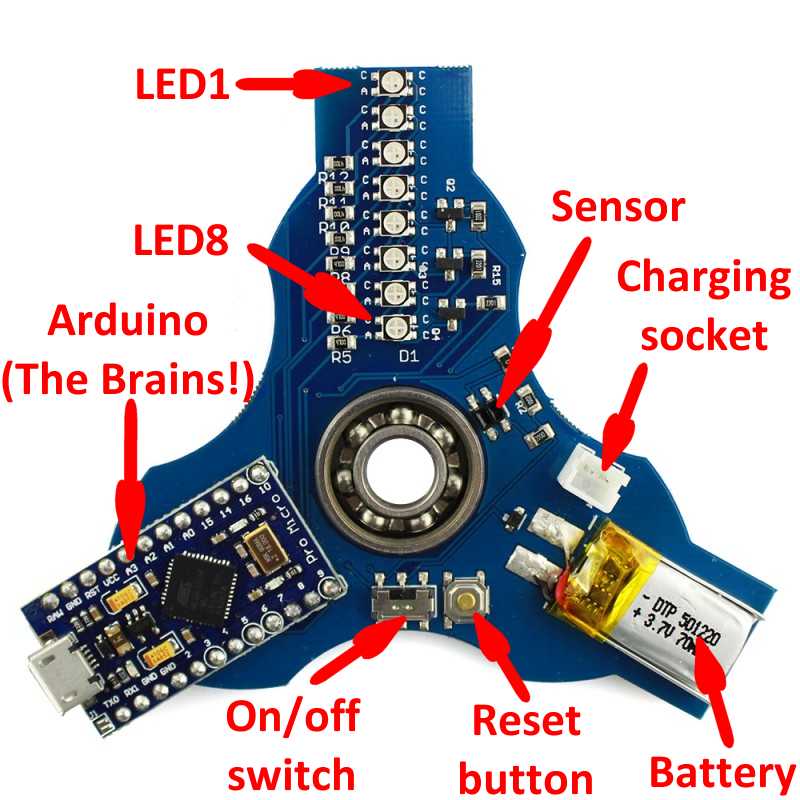
Lightino Finger Spinner

# Introduction

Lightino is a cool fidget spinner with eight 3-colour LEDS that you can program to display any patterns you like. You can even make it spell out messages as it spins. It also contains an infrared sensor that allows it to sense the start of each rotation so that you can make the pattern or the message always start in the same place.

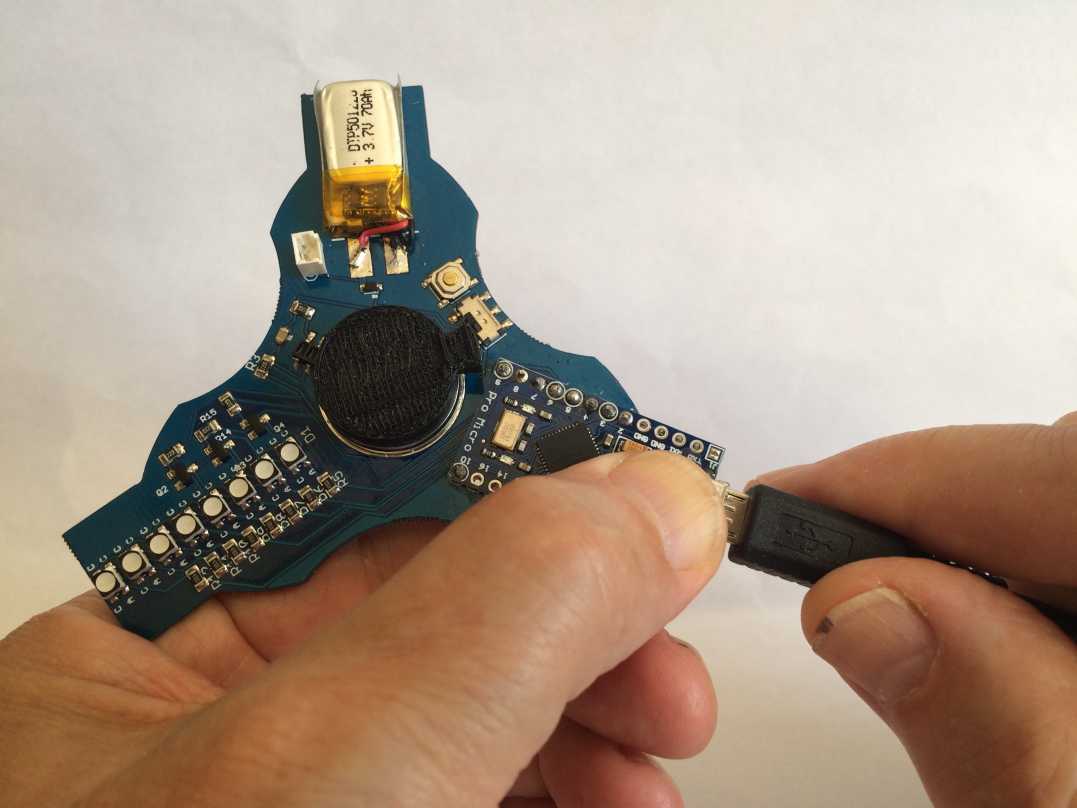
Take a look at the spinner. The 3 arms contain the 8 LEDs, a lithium-ion battery and an Arduino (the brains). There's an on/off switch between the battery and the Arduino. Slide the tiny knob with your finger nail.

You can program the Arduino through the micro USB port on the end. To write your program (called a "sketch" in Arduino terms) and upload it you need to run the Arduino IDE (IDE stands for Integrated Development Environment) which runs on a PC.

You will be programming in the "C" language. This is very powerful, but even so, it's simple to do simple things. You can find a brief introduction and survival guide in an appendix to this lesson set.

The files you will need are contained in this zip file. Double-click it to launch, then copy them out when you need them.



Please note: The USB socket on the end of the Arduino is easily broken and hard to repair. Always hold it firmly by the socket, between your thumb and index finger, when inserting or removing the USB lead.

# Lesson 0: How to program the Lightino

## Getting started

If you want to install the IDE on a computer at home you can find the instructions at:  
<https://coolcomponents.co.uk/blogs/news/getting-started-with-the-lightino-fidget-spinner>

On the Arduino IDE Tools menu, select Board: and then Arduino Leonardo. This is important!

## Your first program

The first program (or "sketch") anyone ever runs on an Arduino is called Blink. It simply flashes a LED on and off.

Here it is, below. Copy and paste it into the Arduino IDE, replacing the skeleton sketch it gives you to start with. (Click File - New from the menu if you need to create a new sketch.) Save it with name Blink.

Alternatively, extract it from StudenFiles.zip (see Introduction), open it with Notepad and copy and paste it from there.

/\*

Lightino Blink

Turns on an LED on and off for half a second each repeatedly.

\*/

// Arduino pin defines

#define RED A1

#define GREEN A0

#define BLUE A2

#define LED1 2

#define LED2 3

#define LED3 4

#define LED4 5

#define LED5 6

#define LED6 7

#define LED7 8

#define LED8 9

#define SENSOR 10

int led\_anodes[] = {LED1, LED2, LED3, LED4, LED5, LED6, LED7, LED8};

int led\_cathodes[] = {RED, GREEN, BLUE};

// The setup function runs once when you press reset or power the board

void setup() {

pinMode(LED1, OUTPUT);

pinMode(GREEN, OUTPUT);

digitalWrite(GREEN, HIGH);

}

// The loop function runs over and over again forever

void loop() {

digitalWrite(LED1, HIGH); // turn the LED on (HIGH is the voltage level)

delay(500); // wait for half a second

digitalWrite(LED1, LOW); // turn the LED off by making the voltage LOW

delay(500); // wait for half a second

}

## Understanding it

The Arduino has a number of programmable input/output pins which in our case are connected to the LEDs. So the first thing in the sketch is to define names for the pins with the #define statements.

Every Arduino sketch has two functions: the first is setup(). This is called once at the beginning and is used to set up things like which pins are to be inputs and which outputs. In this sketch it sets the colour to green (HIGH means On).

The other function is loop() which is then invoked repeatedly for ever. It switches LED1 on, waits a bit, switches it off and waits a bit.

## Making it work

On the Arduino Sketch menu, select Verify/Compile. This will check your sketch for syntax errors then compile it (i.e. translate it into binary code that the chip understands).

If you don't get any errors, plug the Lightino into a USB port (carefully! See the warning in the Introduction), and on the Sketch menu select Upload (*not* Upload using Programmer).

It will complete the compilation, then you should see it scanning COM ports for the Lightino. You should then see it writing the sketch to the Lightino, then reading it back to check, during each of these, displaying # characters to show progress and flashing 2 red LEDs on the Arduino.

Your sketch should now run! Unplug the USB lead and switch on the switch to make it run from its own battery.

## Going further

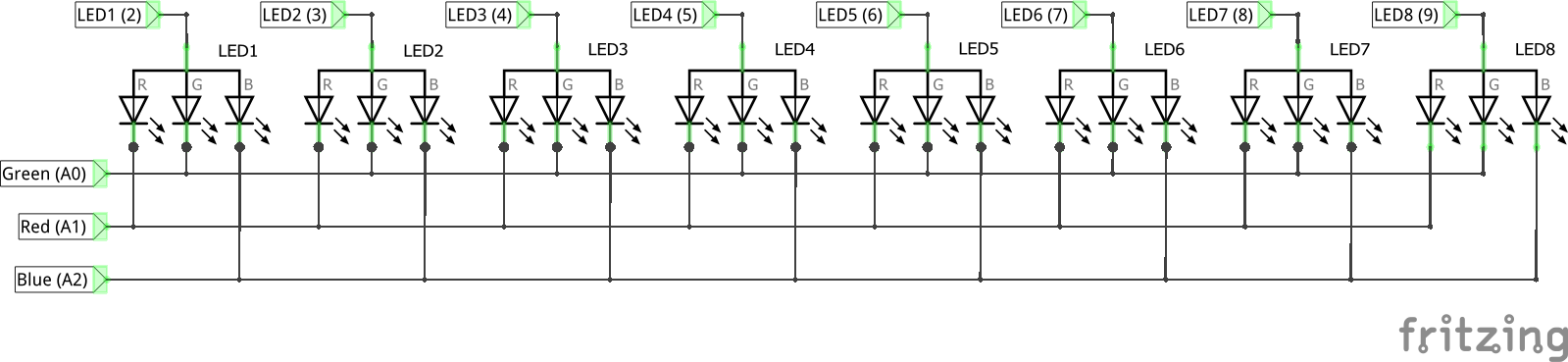
Can you make it flash one of the other LEDs (or more than one), or flash a different colour, or at a different rate?

# Lesson 1: Let's make patterns!

## How to drive the LEDs

We're now ready to start the fun! But first we need to understand how the LEDs work.

Each of the 8 LEDs is actually 3 LEDs in one package: a red, a green and a blue. Each has 2 connections: the anode (the positive one) and the cathode (the negative). All the red cathodes are connected together and all the green cathodes and all the blue cathodes. All 3 anodes in each package are connected together.

You can activate *either* the red *or* the green *or* the blue cathodes by writing something like digitalWrite(RED, HIGH). You can then activate whichever anodes you like with things like digitalWrite(LED1, HIGH). Only when you've done both will anything light up.

If you want to change the colour from red to another colour you first of all have to write digitalWrite(RED, LOW), and in just the same way you can switch individual LEDs off when you no longer want them.

## Your new sketch

In the Arduino IDE, click File - Open and browse to the file Pattern1 from the Student Files. You should see your new sketch Pattern1 which you can open.

Take a look at the function loop(). We saw that this is called repeatedly for ever, but within it are 3 more loops. We can summarise them in "pseudocode" (pretend computer code, just to help us understand what's going on) as follows:

Do 20 times:

| Do for each colour:

| | Switch on the colour (cathode)

| | Do 8 times:

| | | Turn on the next LED then wait a bit

| | Turn off all the LEDs

| | Turn off the colour.

Then there's another set of 3 loops doing something similar, but this time it draws diamonds. Can you see how it does it?

Now over to you! Can you change it to make hollow diamonds?

# Lesson 2: Let’s make hearts!

In the last lesson we used code to decide which LEDs to turn on or off to make a pattern. But what if we want to make it show a pattern such as a heart? If we draw the pattern out on squared paper we can take vertical slices to see which LEDs are to be lit for each slice, and tell the Arduino to light them in turn.

Here's the pattern I drew but you can change it if you think you can draw one you like better, or you could do a smiley face or anything else you like. It worked best for me with 7 columns plus a blank one to turn the LEDs off before the next heart.

The simplest way to do this is to code each column as a binary number, a 1 representing a LED on and a 0 for a LED off. (In C we write binary numbers starting "0b", so 3 in binary would be "0b11".)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| LED1 |  | ● | ● |  | ● | ● |  |  |
| LED2 | ● |  |  | ● |  |  | ● |  |
| LED3 | ● |  |  |  |  |  | ● |  |
| LED4 | ● |  |  |  |  |  | ● |  |
| LED5 | ● |  |  |  |  |  | ● |  |
| LED6 |  | ● |  |  |  | ● |  |  |
| LED7 |  |  | ● |  | ● |  |  |  |
| LED8 |  |  |  | ● |  |  |  |  |

So the first column will be 0b00011110, and the second will be 0b00100001. (We count the bits in a binary number from right to left, starting at zero, so bit 0, representing LED1, is 0 in the first column and 1 in the second.) Check you can see how I got that.

We store these binary numbers in an array called heart, which is simply a list of values. We declare it as having 8 elements and pre-set them like this:

int heart[8] = { 0b00011110, 0b00100001, … }

(We can put this before setup() after the other definitions, because we don't want it redefined every time loop() gets called.)

We can then refer to the first element as heart[0], the second as heart[1] and so on.

There's a function bitRead which extracts bit n (counting from zero at the right hand end) from a binary number. So bitRead(heart[0]) is 0, and bitRead(heart[1]) is 1.

Now HIGH and LOW as we usually write in digitalWrite are actually the same as 1 and 0, so we can directly use bitRead in a digitalWrite!

We're now ready to write our code. Let's display 3 hearts in red, green and blue. Here is the pseudocode:

For each colour:

| Switch on the colour (cathode)

| For each column 0 - 7:

| | For each row 0 - 7:

| | | Turn the LED on or off according to the bit (row)

| | | of the array element (column)

| | Wait a bit

Take the previous example (Pattern1), save it as Pattern2 and replace the code in loop() with your own. Now see if you can write the code.

# Lesson 3: Hold it! Stop your pattern from spinning!

You will have noticed that the patterns you've made so far spin round, because we haven't told the Lightino when to start during a rotation. This is what the sensor is for. We write to the LEDs with a digitalWrite() function, so you could probably guess that we read from the sensor with digitalRead(). This normally returns HIGH, but returns LOW if it sees something passing over the sensor.

## Good interruptions!

In our sketch, we could simply wait for the sensor to go LOW before starting the pattern, but there's a better method using "interrupts". Let me explain.

At any time on your PC you might hit a key on the keyboard, or move the mouse, or some data might arrive from the hard disk or the Internet. If it had to spend all its time watching for those things it'd never get anything else done! Instead, the hardware detects these events and when one of them happens it causes the processor to save where it's got to and what it's doing, execute a small piece of code called an Interrupt Service Routine (or ISR) to save away the incoming data or give the hard disk or the network card the data it's now ready to swallow, then resume what it was doing.

The Arduino IDE provides functions for managing interrupts, but unfortunately not on the input that the Lightino sensor is connected to. So we have to rely on one or two magic incantations invented by someone who knows a lot more about the processor chip than even I do!

## In your sketch…

Starting with Pattern2, save it as Interrupt.

Right at the beginning of the sketch we need to include the following line:

#include <avr/pgmspace.h>

This calls up definitions of the terms used in the magic incantations.

Somewhere before setup() we will need to declare some variables. These are only the sort of magic you're already used to.

// Sensor timings in microseconds

unsigned long curTime; // Most recent sensor detection

unsigned long lastTime = 0; // Previous sensor detection time

unsigned long revTime; // Time for last full rotation

unsigned long dwellTime; // Time between LED changes (based on rotation speed)

int dwellDegrees = 4; // Degrees of rotation between pixels

int revolutions; // Total number of revolutions, in case we want to know

boolean flag = false; // Set by the ISR

boolean overrun = false; // Overrun flag

Some of these are declared as type unsigned long, which means they're stored as 32 bits rather than 16 bits, so allowing values up to around 4 billion. We know they'll never be negative so we don't need to waste a bit to say they're not.

Then in setup() we need to include these lines of deep magic:

// Interrupt setting

PCMSK0 = bit (PCINT6); //Use PCINT6 = PB6

PCICR |= (1<<PCIE0); //Enable Pin change interrupt

sei();

This tells the chip to raise an interrupt whenever a change is detected on pin 10 (known to the chip as PB6), to which the sensor is connected.

At the start of loop() we need to add:

if (overrun) flag = overrun = false;

if (!flag)return;

dwellTime = (revTime \* dwellDegrees / 360);

The flag is set by the ISR at the start of a revolution and overrun is set if we start another revolution before completing the last pattern. So we first check we haven't had an overrun and reset the flag if we had. We then wait for the flag to be set, returning and allowing loop() to be called again and again until it is. And just before making our pattern, we calculate how long a LED must be lit to draw a single pixel, which we choose to be the time taken to spin through a number of degrees specified by dwellDegrees.

In the code that makes the pattern, change delay(2) to dwellDelay(2) and delay(15) to dwellDelay(12) so that the time for each column and the time between patterns is defined as angles of rotation rather than times.

After making our pattern we need to add

flag = false;

This must go just before the last curly bracket of loop().

More goes at the end of the sketch:

void dwellDelay(float n){

unsigned long uSecs;

uSecs = dwellTime \* n;

delay(uSecs / 1000);

delayMicroseconds(uSecs % 1000);

}

This defines the time to display each column in terms of the angle of rotation, rather than a time.

Finally, also at the end of the sketch, some more magic is needed:

ISR(PCINT0\_vect){ // Sensor input change detected

if (!digitalRead(SENSOR)){

if (flag) overrun = true;

flag = true;

revolutions += 1;

curTime = micros();

revTime = curTime - lastTime;

lastTime = curTime;

if (revTime < 20000 || revTime > 500000) revTime = 10000;

}

}

This is the interrupt service routine or ISR, called whenever an interrupt is raised by a change in the sensor output. If the sensor now reads false it sets flag for the code in loop() to act on, but if it’s is already set it sets the overrun flag to say that we're doing another revolution before loop() has finished processing the last.

Also in the ISR we calculate how long the last revolution took (revTime). We can't reliably do that in the main code as it may overrun and miss seeing the sensor. We make sure it's a sensible number, which it won't be if the spinner stops.

The function micros() returns the number of microseconds (millionths of a second) since the chip was last reset. This can be a big number, which is why we had to declare the variables we use here as type long.

Put it all together, compile it and upload it, and see if your hearts now stop spinning!

# Lesson 4: Let’s display messages!

You can display messages just as well as patterns since each letter, digit or symbol is just a pattern. Luckily for us, someone has already worked out all the patterns we need and written a function for displaying them!

Save your previous sketch (Interrupt) with a new name "Text". We can keep most of it.

We need a new file font.h containing the patters for the text characters. This is in the Student Files. Copy it into the folder Text.

Include it in your sketch by adding the following line at the top:

#include "font.h"

We also need 2 new functions displayChar() and displayCharLower() which you can also find in the file displayChar.c in the Student Files. Open it with Notepad then copy and paste them into your sketch after the end of the function loop().

The function displayChar() displays a character, and displayCharLower() does the same except the other way up and backwards. You can use it to display text the right way up along the bottom of the circle as the LEDs spin.

## And now to code!

Let's make it display "Hello" on the top and "Year8!" on the bottom.

Insert these lines somewhere before setup() as follows:

int rows= 8; // Total LED's in a row

const int charHeight = 8;

const int charWidth = 5;

char textString\_Up[] = "Hello";

char textString\_Down[] = "Year8!";

(You can change the message if you like but that's about as long as you can make it.)

Now you have to change the code in loop(), replacing your code for making a pattern. (Make sure you don’t delete the line flag = false; at the end and make sure your curly brackets still match up.)

First of all, switch on one of the colours with a digitalWrite() call. (Can you remember how to do that?) Now loop through the first string displaying the characters. This is how to do it:

for (int k=0; k<strlen(textString\_Up); k++){

displayChar(textString\_Up[k]);

}

Can you see how that works?

Before starting the other string we might want to change the colour. (Don’t forget to switch off the previous colour.

We want the other string to start 180⁰ on from the start of the first. The code at the start of loop() calculates the time for a revolution and records the start time of the last revolution, so we just wait until half a revolution's time from the start, after first checking we haven't already gone past it!

if((micros() < (lastTime + revTime / 2))){

// wait for it . . .

while((micros() < (lastTime + revTime / 2)) && !flag)continue;

//Now display the lower string

}

You can display the lower string (replace the comment) with a loop just like before, except that you'll be displaying the characters in reverse order (counting down instead of up). Can you work out how to do that? And to display the characters the right way up at the bottom, use displayCharLower instead of displayChar.

Don't forget to switch off the colour for the lower message afterwards if it was different from the upper one.

# Lesson 5: How fast is it spinning?

In previous lessons the code calculated the time for a revolution, so as a final exercise, let's display it!

Actually, rotational speed is more often measured in revolutions per minute (RPM), but we can easily calculate that.

We calculated the time for a revolution in microseconds (revTime) and there are 60,000,000 microseconds in a minute, so the speed in RPM will be 60,000,000 / revTime. But we need to convert that to text.

The easiest way to do that is using the built-in sprint() function. This takes 3 (or more) arguments:

1. Somewhere to put the resulting text
2. A pattern for formatting the text
3. A value or values to insert into the text.

For the place to put the text we can replace the definition of textString\_Up by:

char textString\_Up[10];

We've made it 10 characters long, which should be plenty. It's important not to make it too small otherwise we might run off the end and be overwriting who knows what!

In loop(), just before displaying the upper string we can call sprintf()with:

sprintf(textString\_Up, "%4d RPM", 60000000 / revTime);

The %4d means substitute this for the parameter expressed as a 4-digit decimal number. (That's just one of many types of substitution you can specify.)

Why not change textString\_Down in the same way to display the total number of revolutions?

**Warning:** Don't try too hard to see how fast you can get it to spin - it has hard edges and you could easily hurt your fingers!

# Appendix: C Language Survival Guide

## The Basics

* Statements are grouped together by curly brackets { }, not by indentation (as in Python).
* You can put newlines and spaces where you like, but your code will be easier to read if you indent it (like Python).
* Everything between // and the end of the line, or between /\* and \*/ is a comment.
* All variables must be declared, defining their type. They have no meaning outside the smallest enclosing { }. Declarations must be the first things after {.
* All statements (including declarations) must be terminated by a semicolon.
* Lines beginning with # are definitions of constants or directives to include a file from somewhere else containing stuff you'll be needing.

## Declarations

int a, b, c; // Declare 3 integers

float x; // Declare a floating point variable (needn't be a whole number)

boolean love\_marmite; // Takes the values true or false

char c; // this variable holds a single character

int blindmouse[3]; // declares blindmouse[0], blindmouse[1] and blindmouse[2]

## Statements

See the Arduino Reference (under Help in the Arduino IDE) for a list of operators.

### Assignment

x = (a + b) / c;

### Conditional

if ( a == 0 ) then { ….}

if (x >= 0 && x < 10} then { …. } else { …. }

### Loop

while ( sum < max ) do { …. }

for (i=0; i<10; i++) { ….} // The same as i = 0; while (i < 10) { ….; i = i+1; }

break; // Exit immediately from the smallest surrounding loop

continue; // Go back to the top of the smallest surrending loop and start the next time round.

## Functions

* A function has a type, but if it just does a job and doesn't return a result the type is void.
* We have to declare the type of the parameters (length and breadth), to which are assigned the values given in the function call.
* return() returns the result, or just exits from the function.
* See the Arduino Reference for standard functions you can use.

Example:

float area(float length, float breadth) {

return(length \* breadth);

}

roomsize = area(6, 9);

## Common mistakes

Syntax errors are very often in the line or lines immediately before the one that it's complaining about.

* Have you declared all your variables? Maybe you mis-spelled one when you used it.
* Have you forgotten to terminate a statement with a semicolon?
* Do all your brackets match up properly? (Place the cursor just before or just after an opening or closing round, curly or square bracket and a little box will appear around the matching one.)
* Remember, a single "=" is used for assigning a value to a variable (e.g. a = 3;) but you need "==" to test for equality ( e.g. if a == 3 { … } )