Introduction to robotics

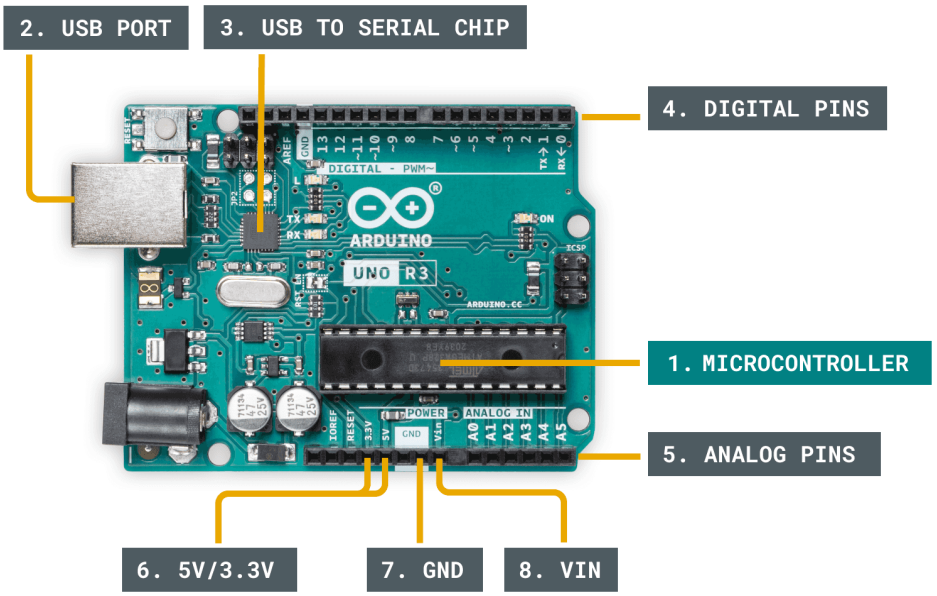
2nd lab

First of all, **get your kits**.

1. Arduino

Most information is taken from Getting started with Arduino

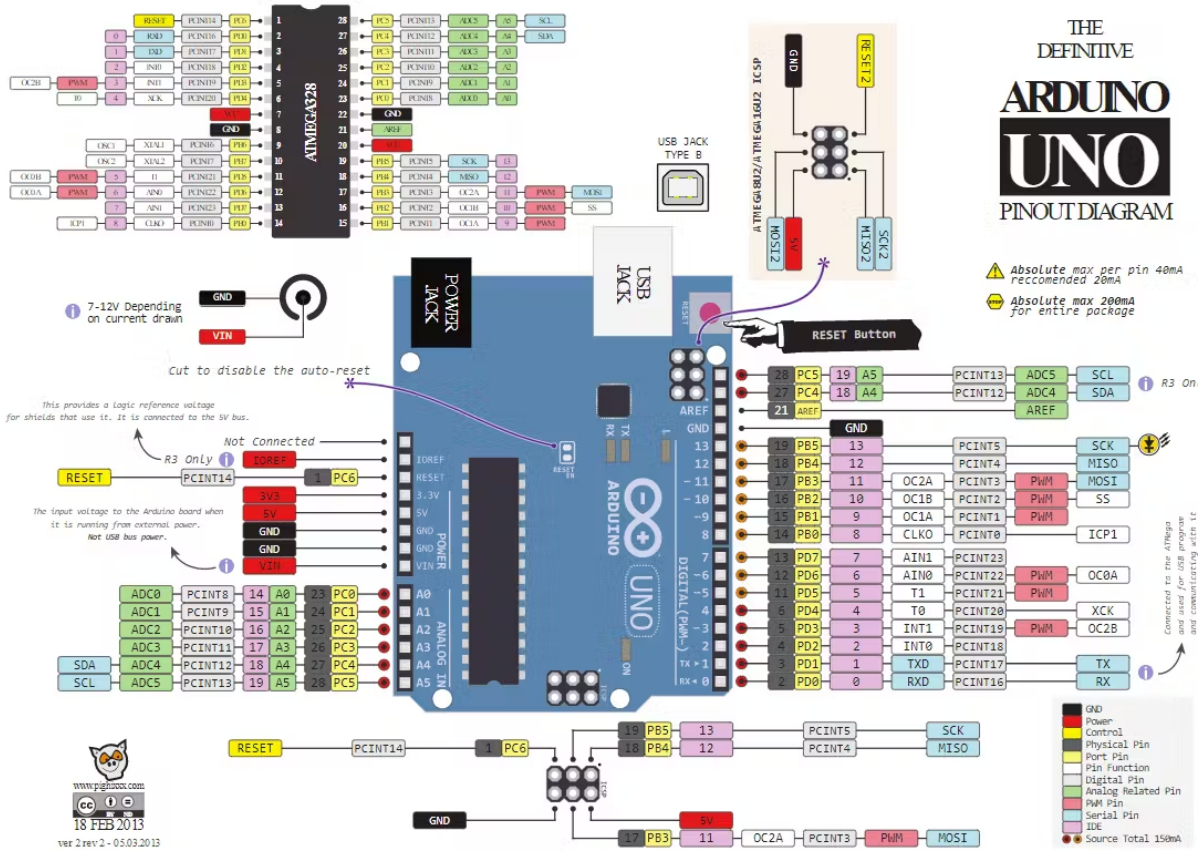
1.1 Arduino Board



1. **Microcontroller** - this is the brain of an Arduino, and is the component that we load programs into. Think of it as a tiny computer, designed to execute only a specific number of things.
2. **USB port** - used to connect your Arduino board to a computer.
3. **USB to Serial chip** - the USB to Serial is an important component, as it helps translating data that comes from e.g. a computer to the on-board microcontroller. This is what makes it possible to program the Arduino board from your computer.
4. **Digital pins** - pins that use digital logic (0,1 or LOW/HIGH). Commonly used for switches and to turn on/off an LED. Note the "~" sign that some have, signifying PWM capability.
5. **Analog pins** - pins that can read analog values in a 10 bit resolution (0-1023).
6. **5V / 3.3V** pins- these pins are used to power external components.
7. **GND** - also known as ground, negative or simply -, is used to complete a circuit, where the electrical level is at 0 volt.
8. **VIN** - stands for Voltage In, where you can connect external power supplies.

1.2 Arduino detailed pinout

The Arduino Uno pinout consists of 14 digital pins, 6 analog inputs, a power jack, USB connection and ICSP header. Many pins have multiple functionalities. We will go through most of these in time, but here is a detailed list of all of them.

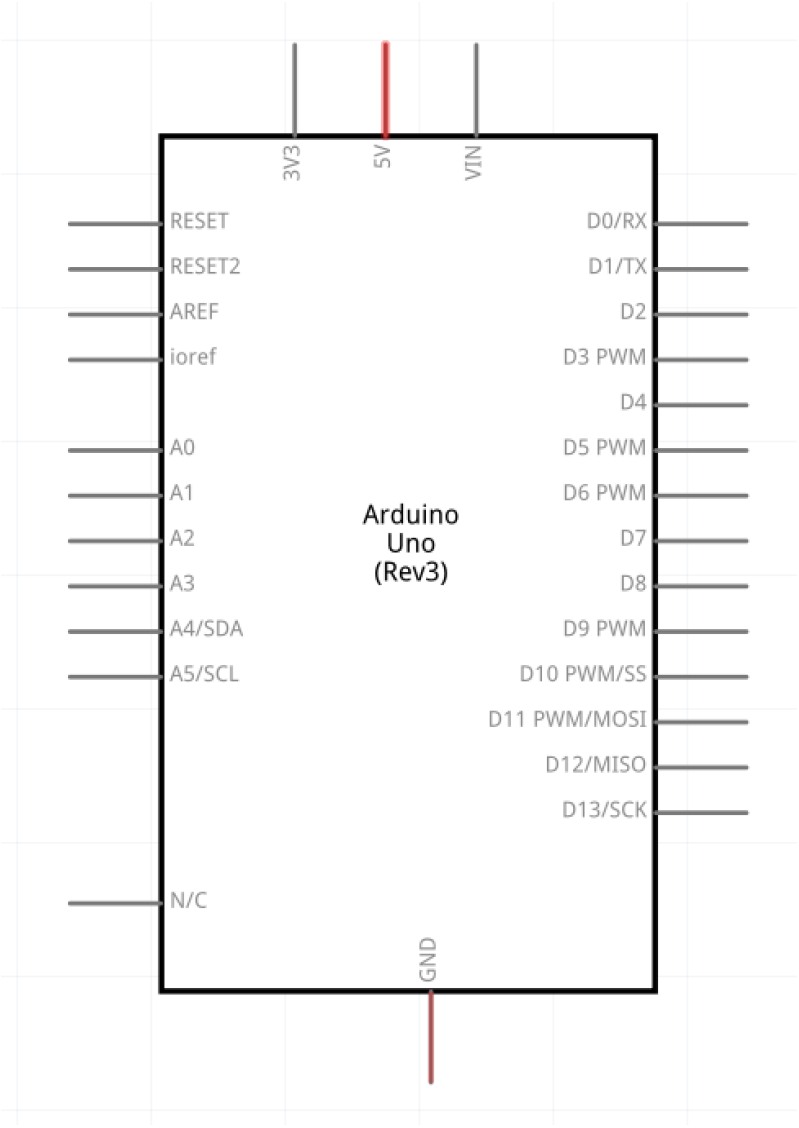


Source: [Full Arduino Uno pin details]

- **VIN** - This pin is used to power the Arduino Uno board using an external power source.
- **5v and 3.3v -** provide regulated 5 and 3.3v to power external components.
- **GND -** we have 5 interconnected GND pins. They are used to close the electrical circuit and provide a common logic reference level throughout your circuit. **Always make sure that all GNDs (of the Arduino, peripherals and components) are connected to one another and have a common ground.**
- **RESET -** resets the Arduino.
- **IOREF -** This pin is the input/output reference. It provides the voltage reference with which the microcontroller operates.
- **D0 - D13 - are** the digital pins.
- **"~" digital pins (3, 5, 6, 9, 10, 11):** have PWM capability. Each pin can provide/sink up to 40 mA max. But the recommended current is 20 mA. The absolute max current provided (or sank) from all pins together is 200mA
- **A0 - A5 -** are the Analog **input** pins (ADC0 - ADC5).
- **D0 (RX) and D1 (TX) -** are used for serial communication. RX = "receive" and TX = "transmit"

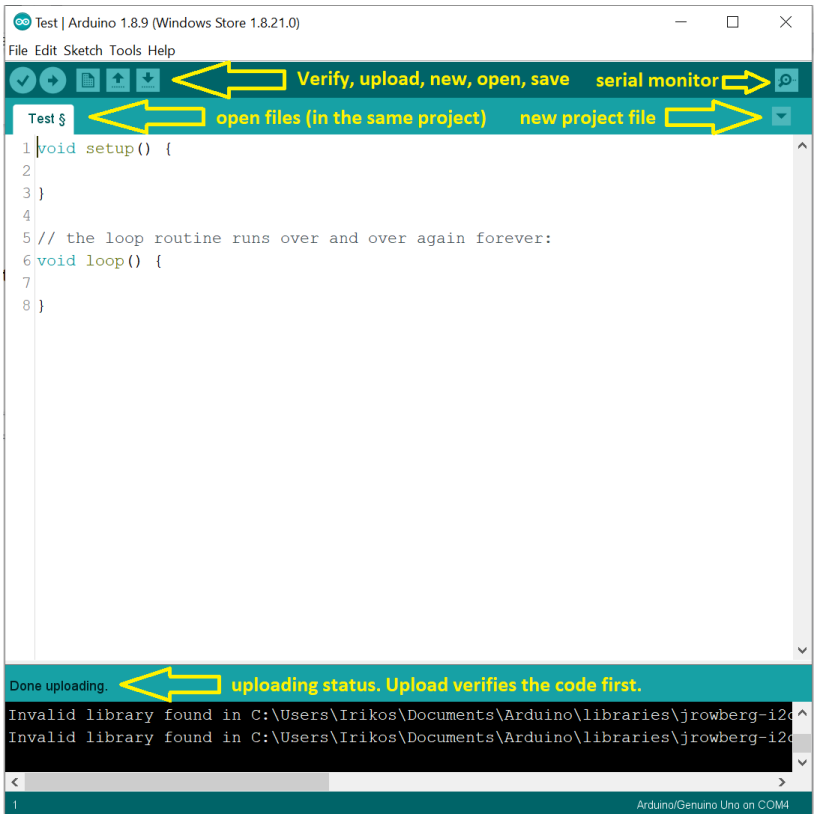
- **SPI - Serial Peripheral Interface - a serial data protocol used by microcontrollers for communication)**
    - **MISO (Master In Slave Out)**
    - **MOSI (Master Out Slave In)t**
    - **SCK (Serial Clock)**
- **I2C - communication protocol designed to enable communication between components on a single circuit board**
    - **SCL - Serial clock line - is the clock line which is designed to synchronize data transfers.**
    - **SDA - Serial data line - is the line used to transmit data.**
- **Aref - Reference voltage for the analog inputs.**
- **Interrupt - INT0 and INT1. Arduino Uno has two external interrupt pins. They are mapped to pins D2 and D3.**

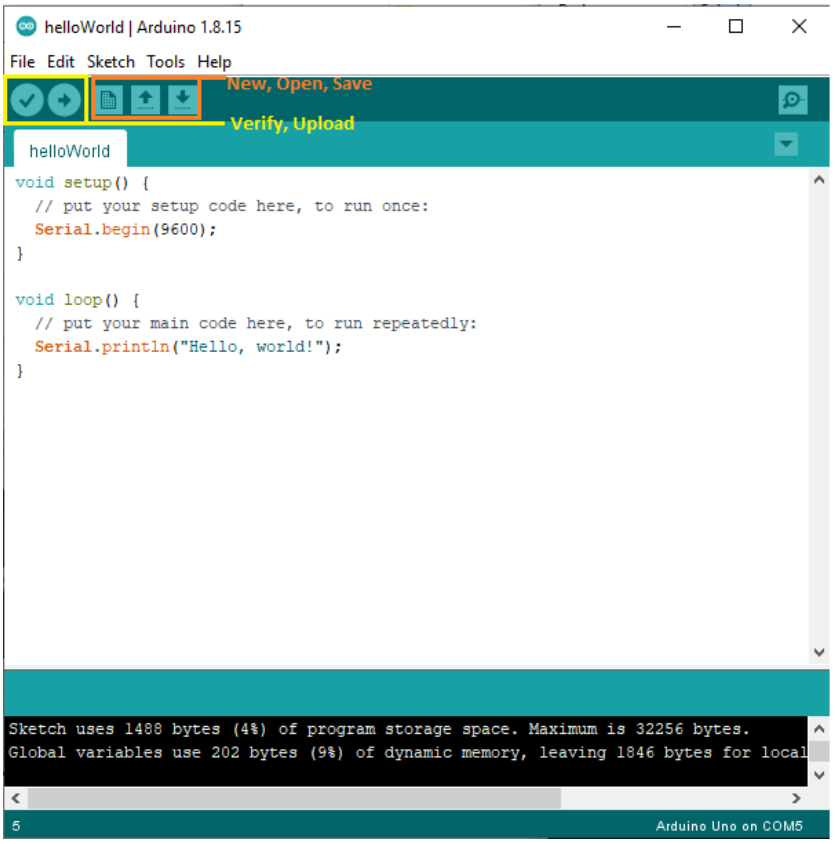
1.3 Arduino Uno electronic schematic from fritzing



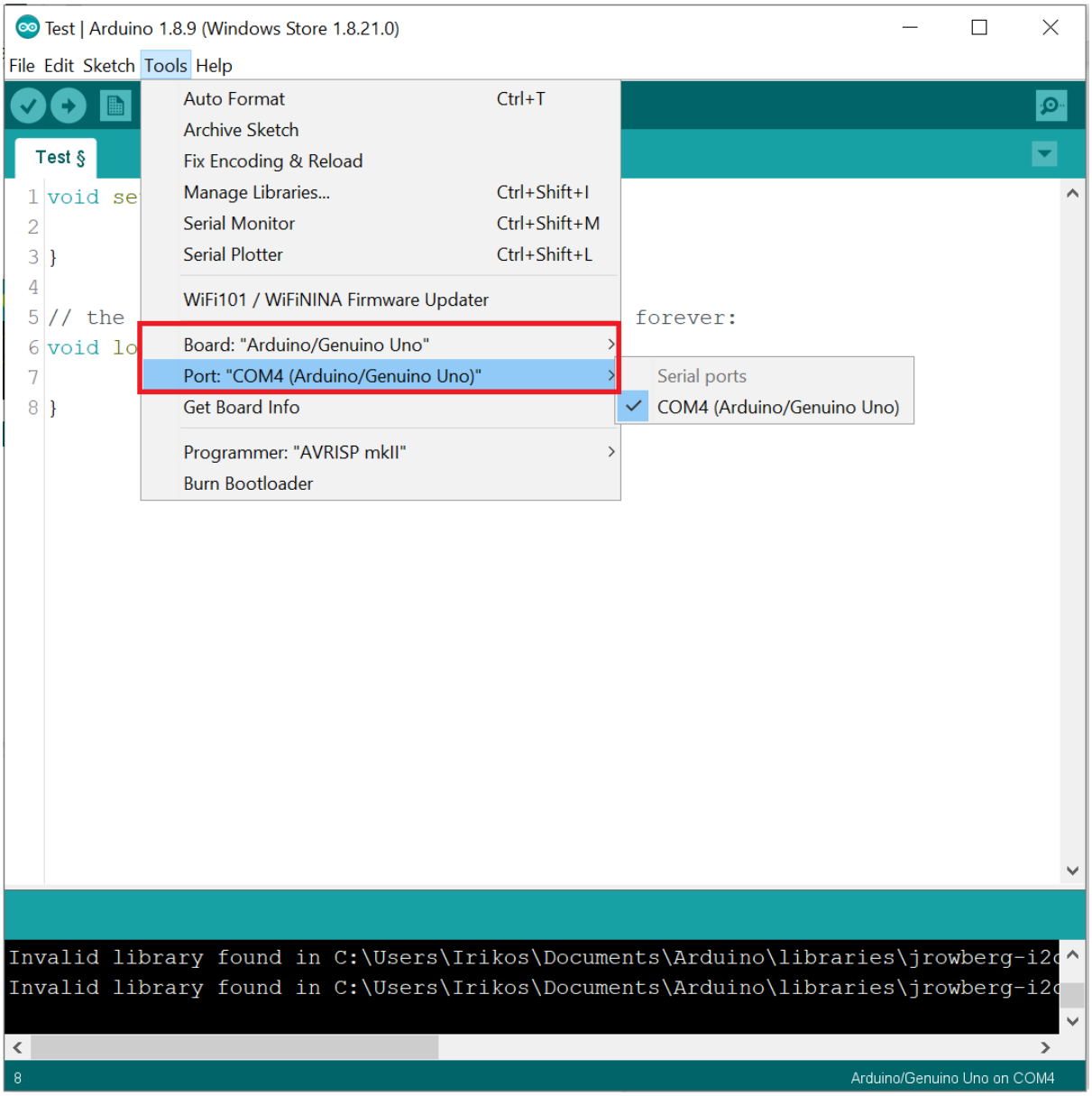
1.3 Arduino IDE

Learn the Arduino IDE interface.





Obs: Don't forget to check if the port is selected!



Tips:

- opening serial monitor **RESTARTS** your program
- uploading also verifies your code, no need to verify and (re)upload when doing small changes
- when uploading, one of the most common error sources is **not having the right port selected.** Make sure you have the right board and port selected. This can change even if the arduino loses power for a brief second
- If you have problems finding the arduino port, **run Arduino IDE as administrator (sudo in linux)**

2. Blinking led no. 13

This is the “Hello, world!” equivalent of the Arduino programming language.

2.1 Blinking the built-in led

Required components:

- Arduino Uno board
- Connection cable

```
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);  // turn the LED on (HIGH is the voltage level)
  delay(1000);                      // wait for a second
  digitalWrite(LED_BUILTIN, LOW);   // turn the LED off by making the voltage LOW
  delay(1000);                      // wait for a second
}
```

Source: https://docs.arduino.cc/built-in-examples/basics/Blink

Questions:

1. What’s with the delay?
   - A: We need to use it so that the LED stays “locked” in a state. Otherwise, we won’t see the difference. It will only stay ON, as there won’t be any time for the current to go to 0
2. What happens if we change only one? Does it matter which one?
   - A: Depending on which one we change, it will stay longer (or shorter) in the state before it
3. What happens if we remove it?
   - A: Depending on which one we remove, it will practically skip the state before it and only stay in the state with delay
4. What is LED_BUILTIN?
   - A: A system constant for number 13

The built in examples are helpful, but do not get into the habit of opening them from File -> Examples in the Arduino IDE. Instead, go to https://docs.arduino.cc/built-in-examples/, understand the schematic and the code, then go to the IDE and write it yourself. The built in examples code can be confusing due to lots of comments, and they also cover the basics of Arduino programming, which you should master. **There’s a big difference between understanding how it’s done and actually doing it yourself. Illusion of mastery is a real thing.**

Remember, **LED_BUILTIN** is actually an arduino constant for the number **13**. You can easily replace LED_BUILTIN with the number 13. It is, however, bad practice to write the number of the pin directly. Still, since we will be using the PIN no. 13 for other purposes besides controlling an LED, it's best to create our own variable. (do that now)

```
const int ledPin = 13;

void setup() {
  // put your setup code here, to run once:
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // put your main code here, to run repeatedly:
  digitalWrite(ledPin, HIGH);
  delay(1000);
  digitalWrite(ledPin, LOW);
  delay(1000);
}
```

Questions:

1. Although we'll leave it like that, what's a potential problem in the naming of "ledPin"?
   - **A:** When you have more than one LED in the project. Also, naming them ledPin1 is not a good naming scheme. Good naming schemes are specific to the project
2. What's another coding style mistake?
   - **A:** Magic numbers aka the "1000" from delay. If we want to change it, we need to change it everywhere. Should be renamed to "blinkInterval", but we can leave it like that for now
3. Why "const int" and not int?
   - **A:** You should never change the pin value inside a running code

Code naming schemes sometimes require inspiration. Pay attention to your project specifications. For example, when building a stoplight that has 3 LEDs of fixed colors, it is reasonable to name your led pins **redLedPin**, **yellowLedPin** and **greenLedPin** as you'll always need these 3 colors in the project. However, if you are just blinking a random LED where the color doesn't matter anymore, you need to find another source of inspiration for the name. Naming it **redLedPin**, because you chose a red LED leaves no room for portability and requires code intervention when changing the color of the LED.

I started an arduino style guide (since there really isn't a comprehensive one), but it's still a work in progress. Feel free to contribute to it: https://github.com/Irikos/arduino-style-guide

Now, as we've talked and as you can see, the built in led is actually connected to pin 13. So let's add an led to pin 13 as well.

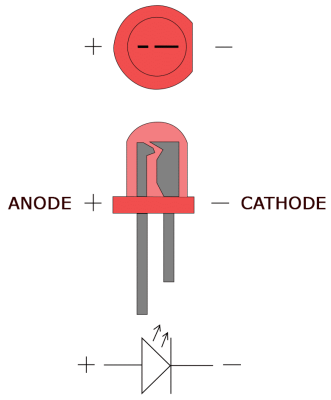
For this, we'll need 3 items:

- a LED (**green or maybe yellow. Blue and white ones are the brightest and can hurt your eyes**)
- a breadboard
- a 220 ohm resistor (330 works as well)

Take them all out.

2.2 Course and previous lab recap

2.2.1 Leds



A light-emitting diode (LED) is a semiconductor light source that emits light when current flows through it. Electrons in the semiconductor recombine with electron holes, releasing energy in the form of photons. The color of the light (corresponding to the energy of the photons) is determined by the energy required for electrons to cross the band gap of the semiconductor.

Interesting facts: Unlike a laser, the light emitted from an LED is neither spectrally coherent nor even highly monochromatic. However its spectrum is sufficiently narrow that it appears to the human eye as a pure (saturated) color.

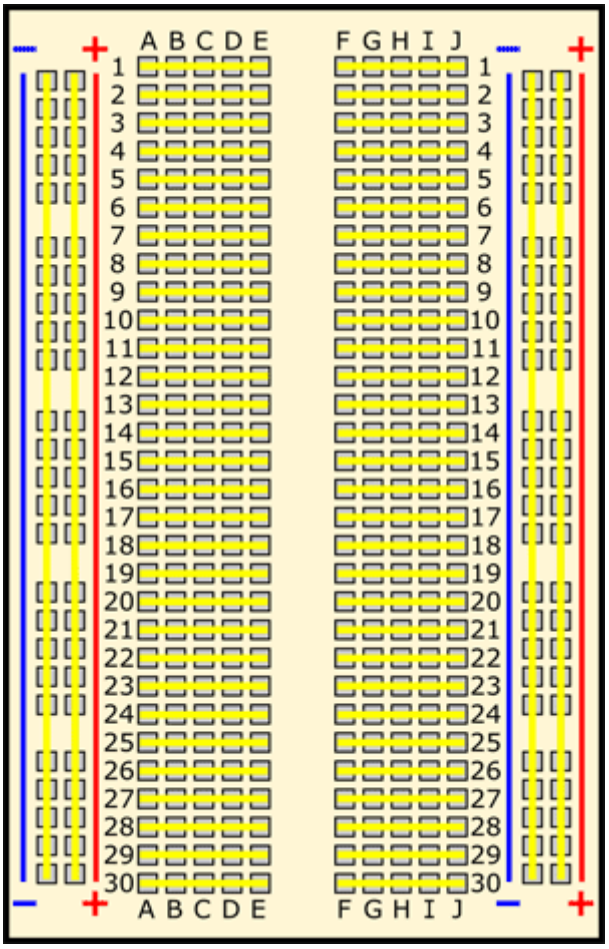
2.2.2 Resistors

Resistors are electrical components used to limit current in a circuit. They are passive components, meaning they only consume power (and can't generate it). The resistor's resistance limits the flow of electrons through a circuit.

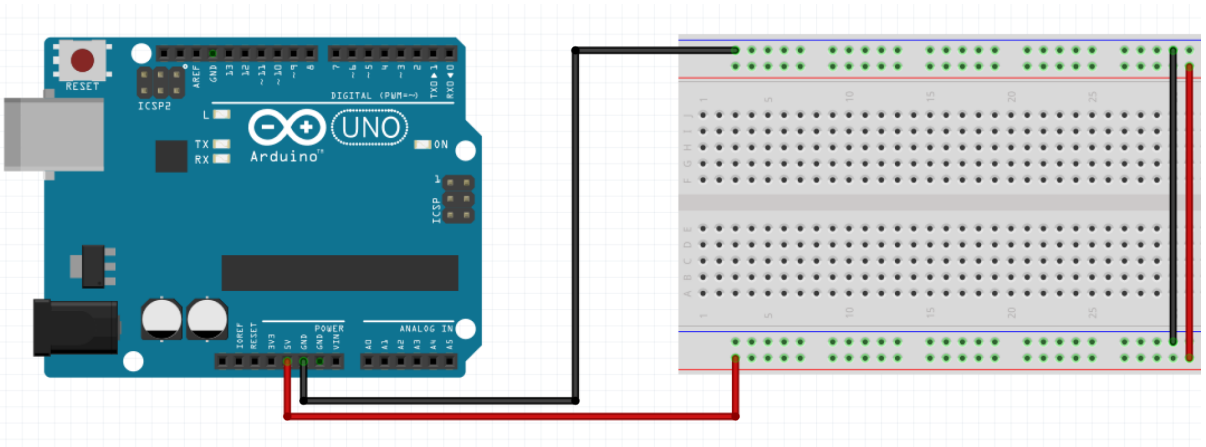
2.2.3 Breadboard

How does a breadboard work? How to use

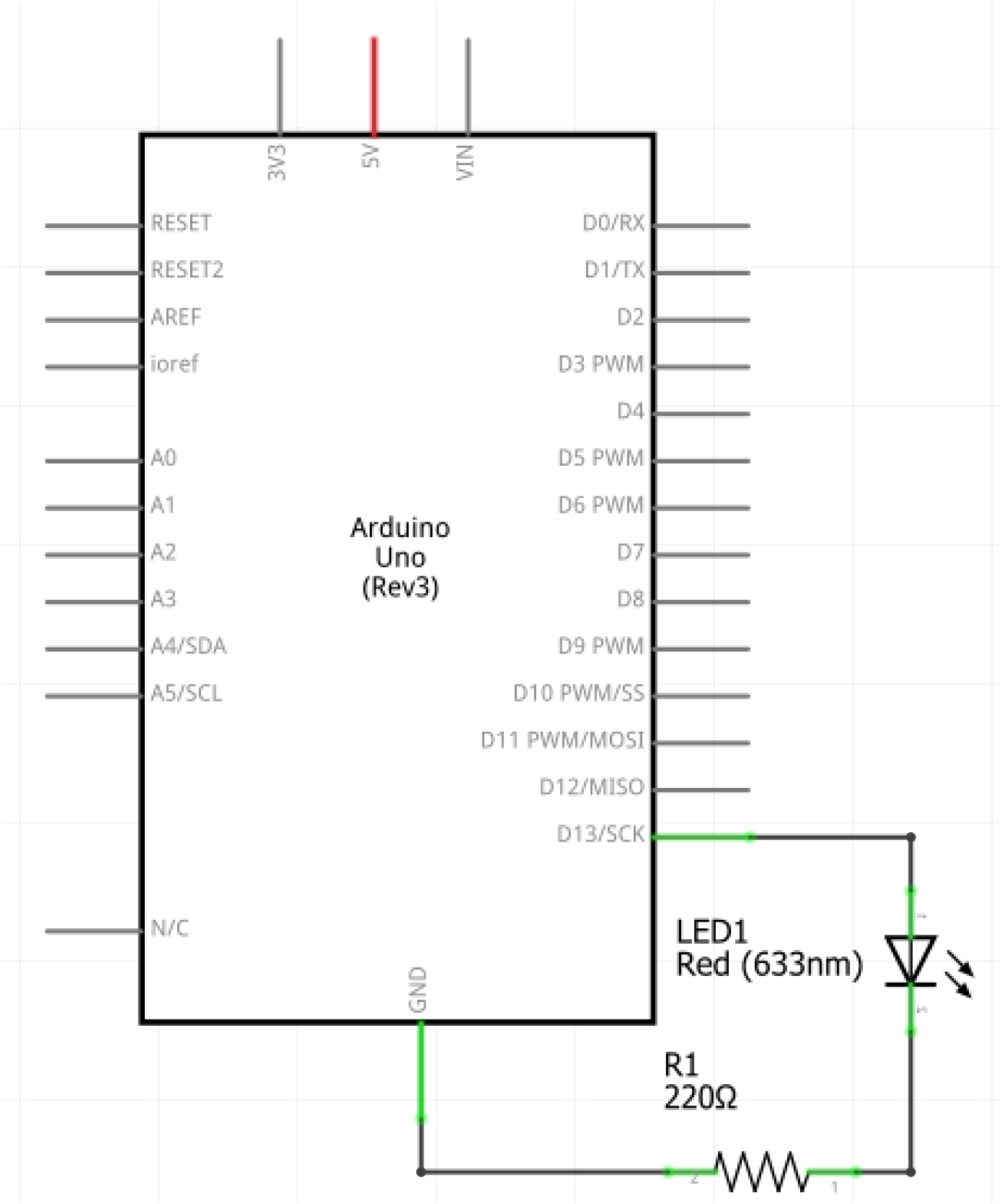
- Short tutorial (~ 1min)
- Properly detailed tutorial (~ 12 min)
- Blog post version of the detailed tutorial



Using the same code, let's add the LED. We can add the resistor to both the GND or the 13
This is the default setup we will usually use as a starting point.



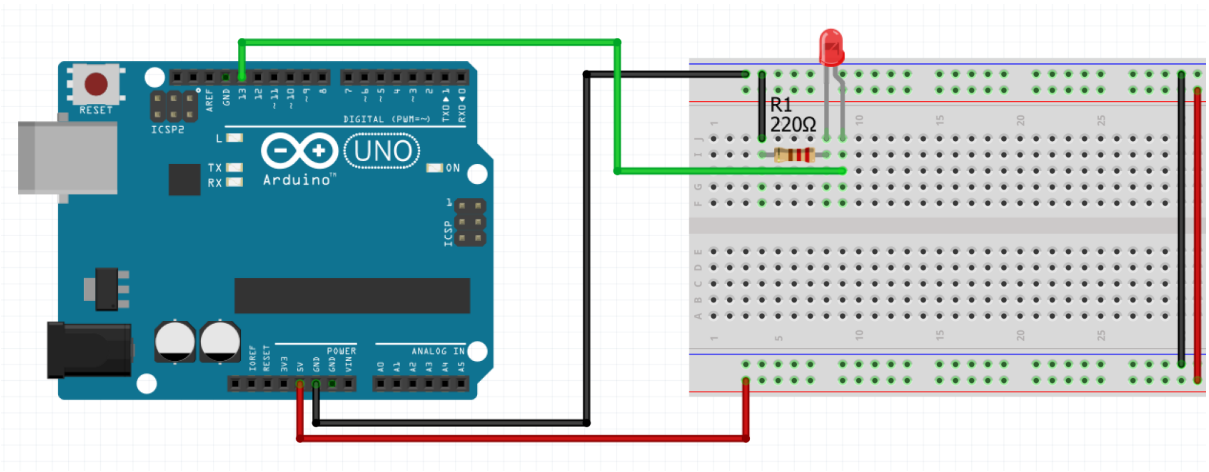
2.3 Blinking an external led



When possible, follow the **wires color scheme:**

- **BLACK** for **GND (dark colors if not available)**
- **RED** for **POWER (3.3V / 5V / VIN) (bright colors if not available)**
- Bright Colored for read and write signal (use **red** when none available and **black** only as a last option)
- We know it is not always possible to respect this due to lack of wires, but the first rule is **NOT USE BLACK FOR POWER OR RED FOR GND!**

Breadboard schematic



ATTENTION! Do not look directly perpendicular to the tip of the LED and do not look too much at the LED. It can and will affect your eyes - take this seriously!

3. Millis()

millis() is a widely used function in Arduino programming. It returns the number of milliseconds passed since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 50 days.

```
unsigned long myTime;

void setup() {
  Serial.begin(9600);
}

void loop() {
  Serial.print("Time: ");
  myTime = millis();

  Serial.println(myTime); // prints time since program started
  delay(1000);        // wait a second so as not to send massive amounts of data
}
```

Source: https://www.arduino.cc/reference/en/language/functions/time/millis/

3.1 Blinking an LED without using the delay() function

The blink example works well, but mostly because it's simple. Can you identify a problem with it?

The delay() function pauses the program for the amount of time specified as a parameter. While it is easy to create a blinking LED with the **delay()** function and many sketches use short delays for such tasks as switch debouncing, the use of delay() in a sketch has significant drawbacks. No other reading of sensors, mathematical calculations, or pin manipulation can go on during the delay function, so in

effect, it brings most other activity to a halt. More knowledgeable programmers usually avoid the use of **delay()** for timing of events longer than 10's of milliseconds unless the Arduino sketch is very simple.

But there's a better way.
"An analogy would be warming up a pizza in your microwave, and also waiting for some important email. You put the pizza in the microwave and set it for 10 minutes. The analogy to using delay() would be to sit in front of the microwave watching the timer count down from 10 minutes until the timer reaches zero. If the important email arrives during this time you will miss it.
What you would do in real life would be to turn on the pizza, and then check your email, and then maybe do something else (that doesn't take too long!) and every so often you will come back to the microwave to see if the timer has reached zero, indicating that your pizza is done." (source: arduino.cc)

```
const int ledPin =  13;
byte ledState = LOW; // could be bool
// Generally, you should use "unsigned long" for variables that hold time
// The value will quickly become too large for an int to store
unsigned long previousMillis = 0;
const long interval = 1000;       // interval at which to blink (milliseconds)
void setup() {
  pinMode(ledPin, OUTPUT);
}
void loop() {
  // check to see if it's time to blink the LED; that is, if the difference
  // between the current time and last time you blinked the LED is bigger than
  // the interval at which you want to blink the LED.
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval) {
    // save the last time you blinked the LED
    previousMillis = currentMillis;
    // if the LED is off turn it on and vice-versa:
    if (ledState == LOW) {
      ledState = HIGH;
    } else {
      ledState = LOW;
    }
    // set the LED with the ledState of the variable:
    digitalWrite(ledPin, ledState);
  }
}
```

Source: https://www.arduino.cc/en/tutorial/BlinkWithoutDelay

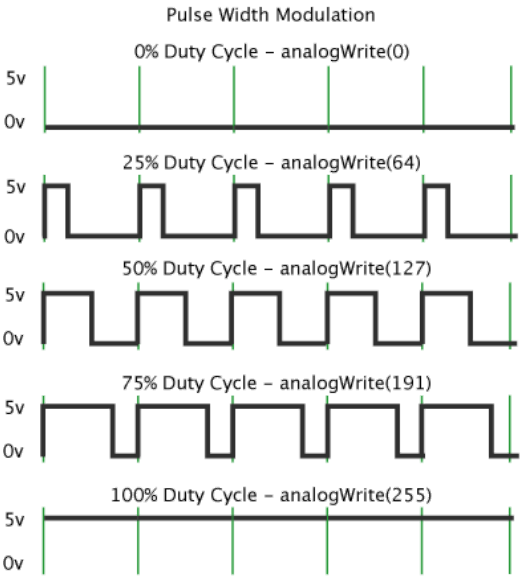
4. Pulse-width modulation

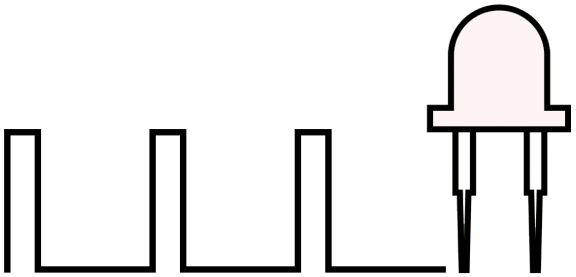
4.1 Fading an LED

Now that we know how to turn a light (LED) on and off, let's see how we can choose the intensity of the light as well.

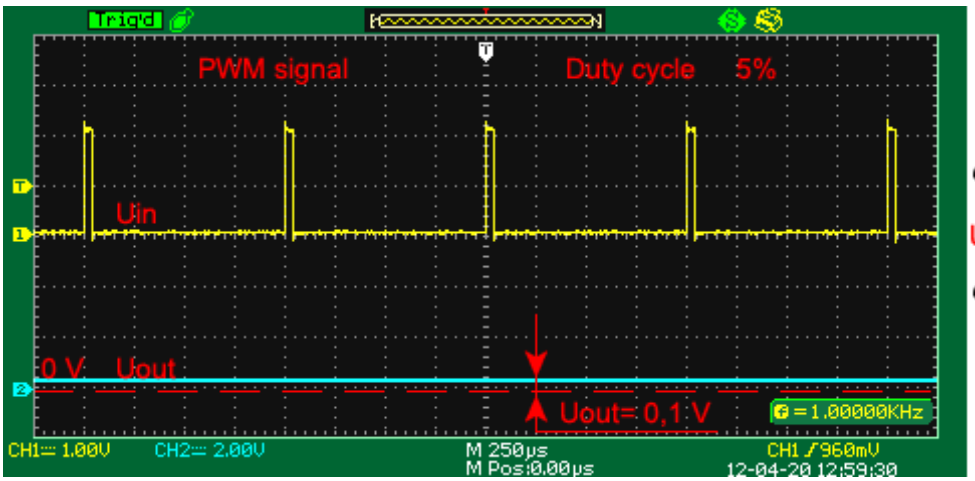
As you might remember from the course, the LED is a good example of a component that can be used both as a **digitalOuput** and an **analogOutput.** Basically, the intensity of the LED when using **digitalWrite(ledPin, HIGH) = analogWrite(ledPin, 255)**, but you can set the LED value anywhere between 0 and 255.

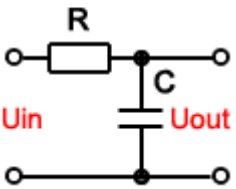
Let's recap the course a bit: PWM (pulse-width modulation) is a technique for getting analog results with digital means.





In practice, we use an RC low pass filter to convert the square wave into a continuous analog voltage (azzure line in the gif below)





That being said, let's change the code and play a bit with the LED intensity.

Since 255 is maximum, let's change it to 200.

```
const int ledPin = 13;

void setup() {
  pinMode(ledPin, OUTPUT);
}

void loop() {
  analogWrite(ledPin, 128);
}
```

What do we notice? What if we change the value to 127?

Questions:

1. Why did it turn off?
   - A: basically it treats it as HIGH if the value is >= 128 and LOW if the value is <= 127
2. Why didn't it fade?
   - A: it's digital not analog

Ok, so always remember that the PINS that can use PWM and on which you can use analogWrite are the ones with **~ (3, 5, 6, 9, 10, 11)**

So, let's change it to **digital pin 9 (D9)** and try again. After we have connected the **ANODE (+)** wire to D9, we can easily see the advantage of keeping the pinNumber in a variable. In the code, we only have to change the **ledPin** value once, instead of changing it both in setup (pinMode) and in loop (analogWrite)

```
const int ledPin = 9;

void setup() {
  pinMode(ledPin, OUTPUT);
}

void loop() {
  analogWrite(ledPin, 200); // change it to more values and test it's
                            // limits: 200, 128, 127, 100, 10 etc
}
```

4.2 Manual PWM (random fact)

Fun fact: you can also “manually” implement a PWM on any pin by repeatedly turning the pin on and off for the desired times. For example:

```
void setup()
{
  pinMode(13, OUTPUT);
}

void loop()
{
  digitalWrite(13, HIGH);
  delayMicroseconds(100); // Approximately 10% duty cycle @ 1KHz
  digitalWrite(13, LOW);
  delayMicroseconds(1000 - 100);
}
```

Source: https://docs.arduino.cc/tutorials/generic/secrets-of-arduino-pwm

This technique has the advantage that it can use any digital output pin. In addition, you have full control of the duty cycle and frequency. One major disadvantage is that any interrupts will affect the timing, which can cause considerable jitter unless you disable interrupts. A second disadvantage is you can't leave the output running while the processor does something else. Finally, it's difficult to determine the appropriate constants for a particular duty cycle and frequency unless you either carefully count cycles, or tweak the values while watching an oscilloscope.
A more elaborate example of manually PWMing all pins may be found here.

We will not be using manual PWM in the course (but you are welcome to try it and use it for very good reasons).

4.3 Fade a led on and off (optional)

```
const int ledPin = 9;      // the PWM pin the LED is attached to
int brightness = 0;        // how bright the LED is
int fadeAmount = 5;        // how many units to fade the LED by

// the setup routine runs once when you press reset:
void setup() {
  // declare pin 9 to be an output:
  pinMode(ledPin, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  // set the brightness of pin 9:
  analogWrite(ledPin, brightness);

  // change the brightness for next time through the loop:
  brightness = brightness + fadeAmount;

  // reverse the direction of the fading at the ends of the fade:
  if (brightness <= 0 || brightness >= 255) {
            fadeAmount = -fadeAmount;
  }
  // wait for 30 milliseconds to see the dimming effect
  delay(30);
}
```

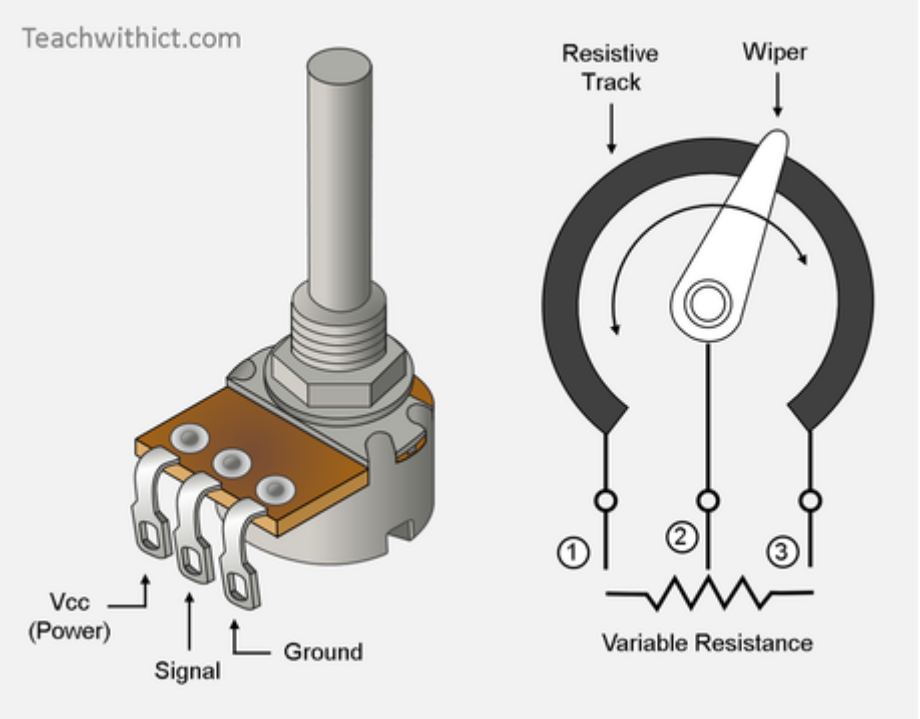
Source: https://www.arduino.cc/en/Tutorial/Fade

5. Analog Read

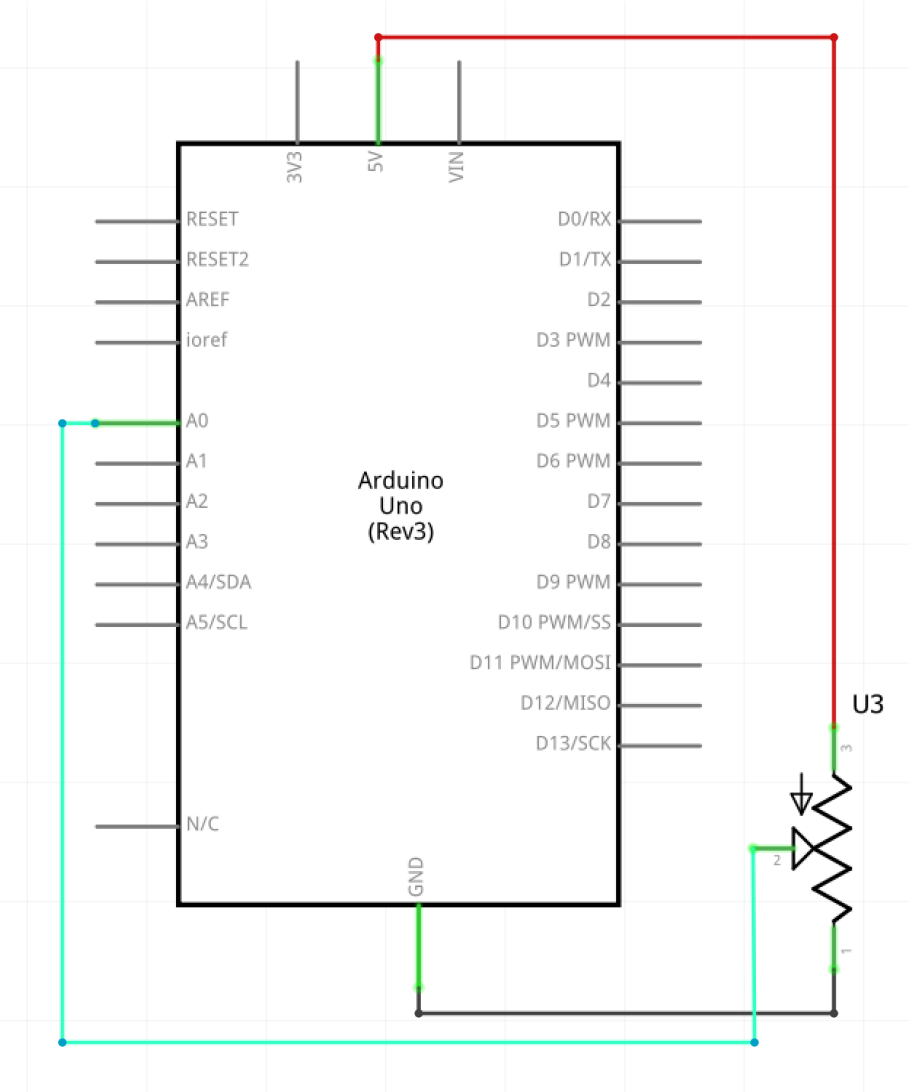
5.1 Reading and printing a potentiometer value

Controlling the LED intensity is great, but changing the value and re-uploading the code is not practical. Let's add a component which we can use to control the LED intensity.

Meet the potentiometer:



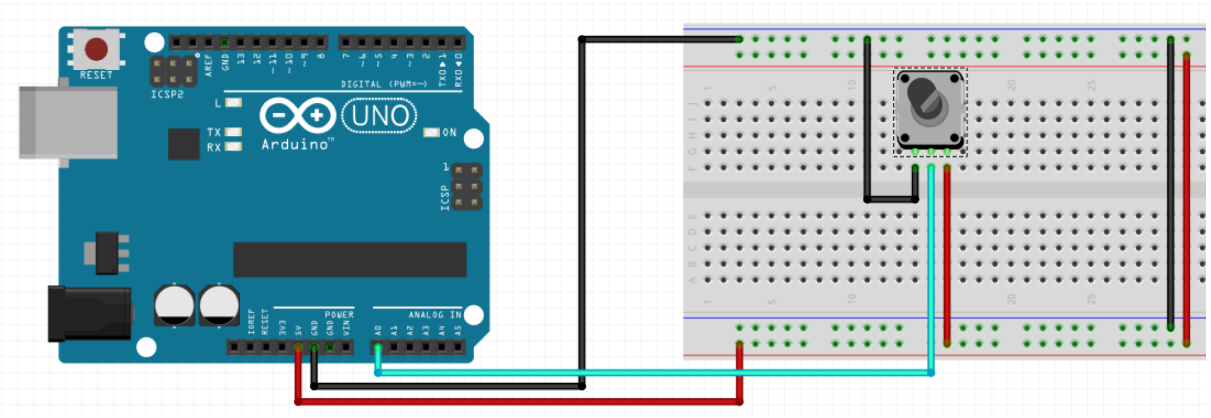
A potentiometer is a simple knob that provides a variable resistance, which we can read into the Arduino board as an analog value.



Electronic schematic of a connected potentiometer

Let's connect the potentiometer to the board as follows:

- the outermost pins go to **5V** and **GND**
- the middle pin goes to **A0**



This is the example from https://www.arduino.cc/en/Tutorial/AnalogReadSerial, but slightly modified.

```
const int potPin = A0;
int potValue = 0;

void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
}
// the loop routine runs over and over again forever:
void loop() {
  // read the input on analog pin 0:
  potValue = analogRead(potPin); // not the best practice
  // print out the value you read:
  Serial.println(potValue);
  delay(1);         // delay in between reads for stability
}
```

Q: Why do we use **const int** for pins but **int** for values?

- **A:** because the pin values do not change during the execution of the program, thus we make it immutable. **#DEFINE** can be used as well, but we use **const int** to match the example codes

Run the program and open **Serial Monitor.**
On the bottom right, you will see **9600 baud.**

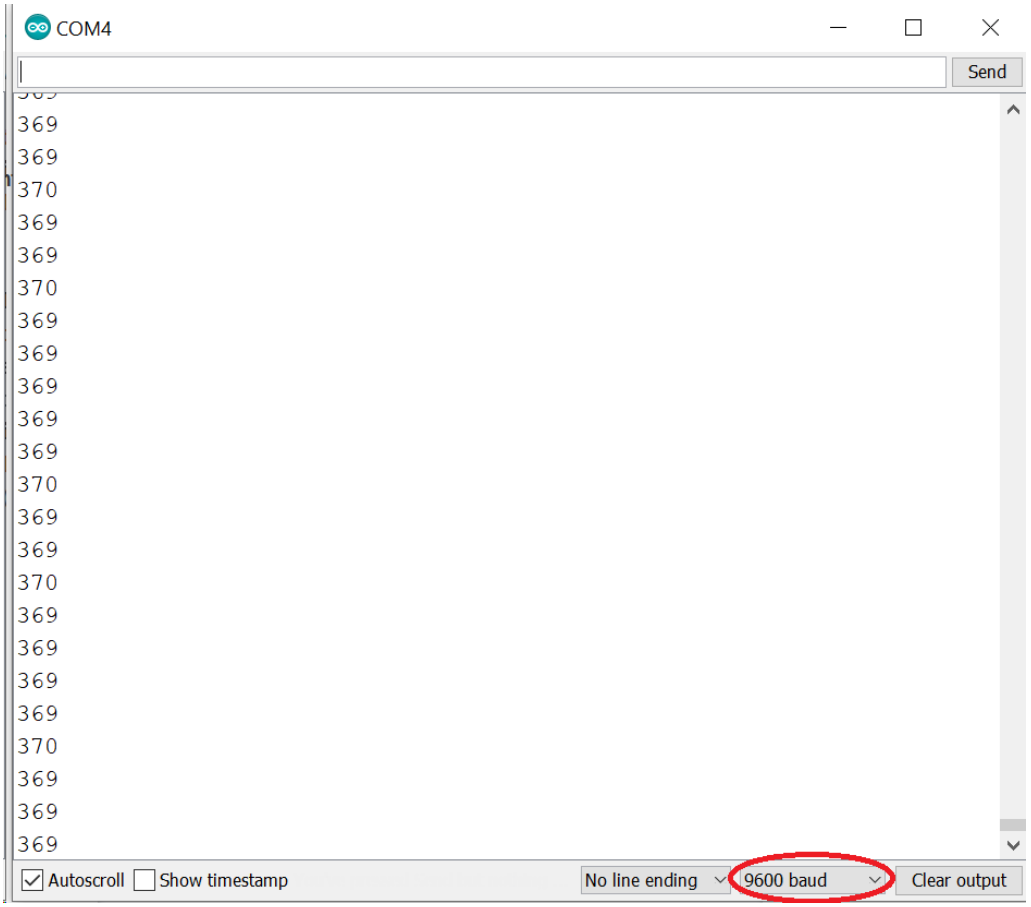
The **baud rate** is the **rate** at which information is transferred in a communication channel. In the serial port context, "9600 **baud**" means that the serial port is capable of transferring a maximum of 9600 bits per second.

At baud rates above 76,800, the cable length will need to be reduced. The higher the baud rate, the more sensitive the cable becomes to the quality of installation, such as how much of the wire is untwisted around each device.

Long story short, the number in Serial.begin(9600) must be the same as the baud rate.

Q: What do you think will happen if we change the baud rate in the serial monitor? Why does this happen?

- **A:** Because there's a mismatch between what the two systems think the speed is and the data is garbled.



We will be using the map() function quite a lot, so learn it.

While you can divide and multiply to get similar values, it is not best practice. You can easily forget a floating point etc.
map() re-maps a number from one range to another. That is, a value of fromLow would get mapped to toLow, a value of fromHigh to toHigh, values in-between to values in-between, etc.
Syntax:
map(value, fromLow, fromHigh, toLow, toHigh)

Parameters
value: the number to map.
fromLow: the lower bound of the value's current range.
fromHigh: the upper bound of the value's current range.
toLow: the lower bound of the value's target range.
toHigh: the upper bound of the value's target range.

Returns
The mapped value. Data type: long.

Example
val = map(val, 0, 1023, 0, 255);

Source: https://www.arduino.cc/reference/en/language/functions/math/map/

5.2 Reading the analog voltage

The microcontroller of the board has a circuit inside called an analog-to-digital converter or **ADC** that reads this changing voltage and converts it to a number between 0 and 1023. When the shaft is turned all the way in one direction, there are 0 volts going to the pin, and the input value is 0. When the shaft is turned all the way in the opposite direction, there are 5 volts going to the pin and the input value is 1023. In between, analogRead() returns a number between 0 and 1023 that is proportional to the amount of voltage being applied to the pin.

Therefore, we can calculate the voltage read on the **potPin** by adding one line.

```
float voltage = potValue * (5.0 / 1023.0);
```

So, our program should look like this:

```
const int potPin = A0;

int potValue = 0;
float voltage = 0;

void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
}

// the loop routine runs over and over again forever:
void loop() {
  // read the input on analog pin 0:
  potValue = analogRead(potPin); // not the best practice
// Convert the analog reading (which goes from 0 - 1023) to a voltage (0 - 5V):
  voltage = potValue * (5.0 / 1023.0);
  // print out the read voltage and comment value printing
  // Serial.println(potValue);
  Serial.println(voltage);
  delay(1);        // delay in between reads for stability
}
```

Source: https://docs.arduino.cc/built-in-examples/basics/ReadAnalogVoltage

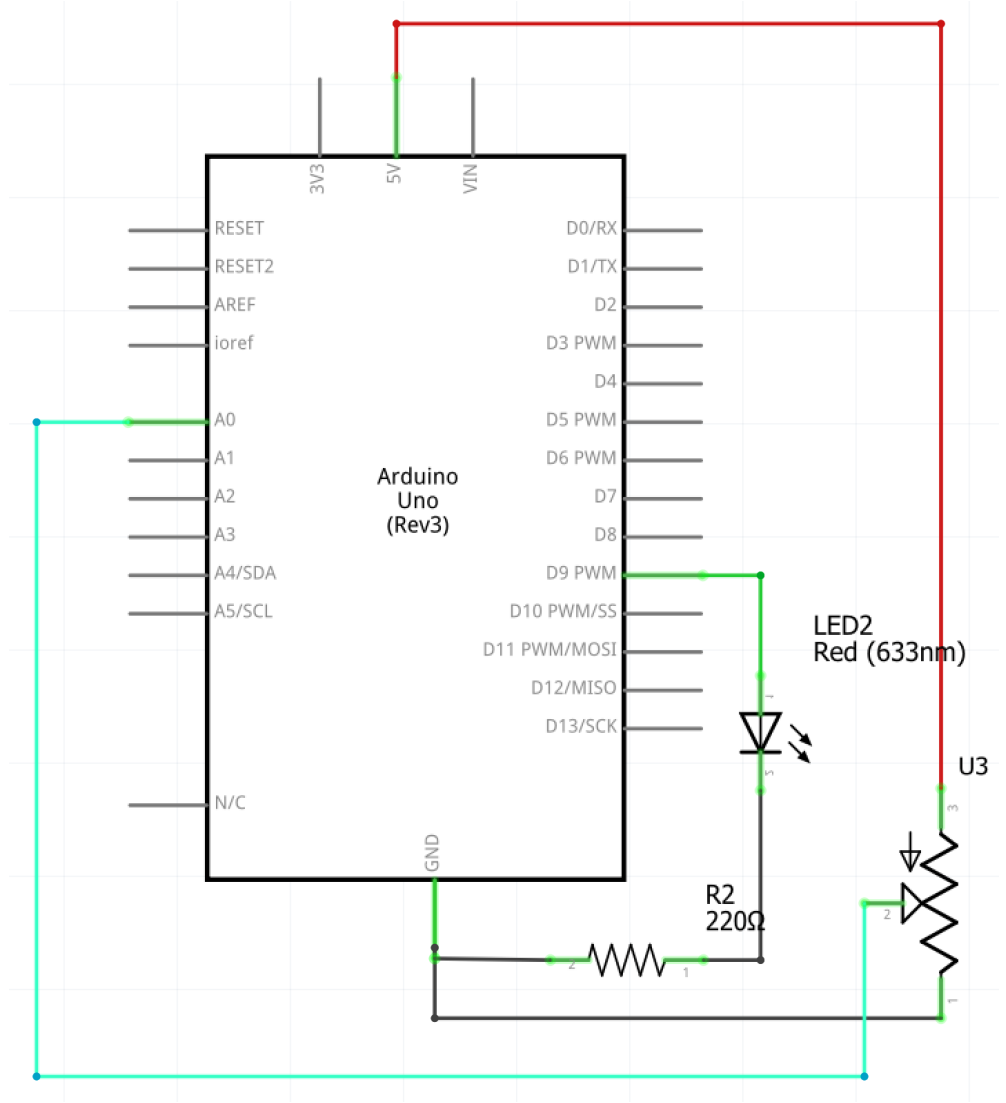
Let's recap the lab a bit:

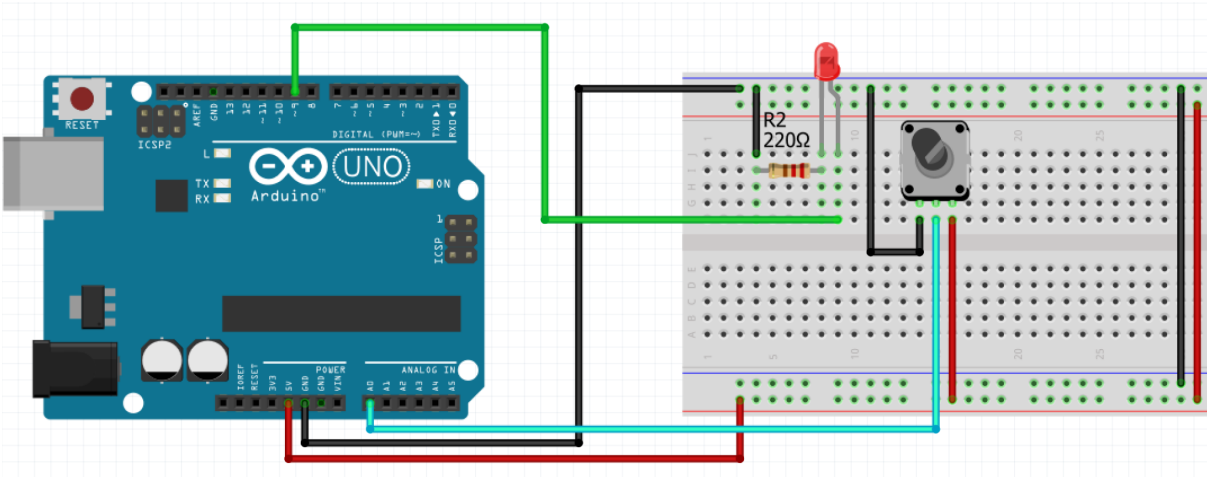
- we connected a **LED** to **D9**, which also has a **~**
- we used analogWrite(ledPin, ledValue), with ledValue = 0..255 to set the intensity of the LED

Now, let's use the potentiometer to control the intensity of the LED.

5.3 Fading an LED with the potentiometer

For this, we need to read the **potValue**, store it in a variable and write that variable to the LED.





The idea is simple, we have an analogInput and need to control an analogOutput with it

```
const int potPin = A0;
const int ledPin = 9;
// global variables are initialized to 0 by default
int potValue;

void setup() {
  pinMode(potPin, INPUT);
  pinMode(ledPin, OUTPUT);
}
void loop() {
  potValue = analogRead(potPin);
  analogWrite(ledPin, potValue);
}
```

Source: https://docs.arduino.cc/built-in-examples/analog/AnalogInOutSerial

Do that, see what happens.

Questions:

1. Why does it behave in this way?
   - A: because the read values are between 0..1023 (10 bits) and the written values are between 0..255 (8 bits). The final written value is modulo 256, so going over 255 takes us to 0
2. How can we fix it?
   - A: we can use the map(potValue, 0, 1023, 0, 255) function or we can divide the values / 4, as it is a good-enough approximation

```
const int potPin = A0;
const int ledPin = 9;

int potValue = 0;
int ledValue = 0;

void setup() {
  pinMode(potPin, INPUT);
  pinMode(ledPin, OUTPUT);
}

void loop() {
  potValue = analogRead(potPin);
  ledValue = map(potValue, 0, 1023, 0, 255);
  // ledValue = potValue / 4; // this works just as well
  analogWrite(ledPin, ledValue);
}
```

6. Wrap-up, Review and Q&A

Questions:

1. In the course, you discussed why “arduino is not an ideal environment for learning electronics”. Why is that?
    a. A: It's because Arduino often resorts to software-based solutions for most challenges.
2. Can you give an example from the stuff we did today, where arduino is an overkill solution?
    a. During our exercise, we adjusted the brightness of an LED using a potentiometer. We processed a 5V signal through a variable resistor, read the altered value, mapped this value to a different range, and then regulated the LED's brightness using a PWM method. A simpler method would have been to directly link the potentiometer's center pin to the LED's anode, thereby directly adjusting the voltage supplied to the LED.
3. Is there any advantage to our solution, then?
    a. A: Yes; first of all, we can program it to react differently to the values of the potentiometer. Secondly, the PWM method provides a potentially smoother linearity than the direct solution.

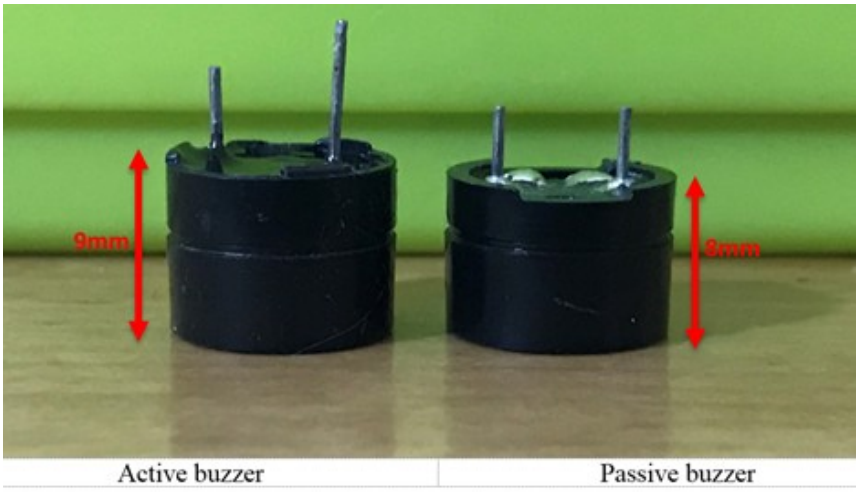
7. Basic sounds

7.1 Active and passive piezo buzzers

Active and Passive Buzzer - Discussed

In your kit you have **1 buzzer, either passive or active.**
They look quite similar, so be careful when telling them apart:

- The height of the two is slightly different
- **if you apply a DC voltage to them and it buzzes, it is active.**
- Pins side, you can see a green circuit board in the passive buzzer. No circuit board and closed with a black is an active buzzer.
- Using a multimeter, if the buzzer beeps when checking continuity, **it is active.**



-

Although they look similar, inside they are quite different. But let's start with what they have in common: they are both **piezo buzzers.** A piezo buzzer has a thin piezoelectric plate inside it that vibrates mechanically whenever a voltage is applied to it. It's the same principle as a quartz crystal that is used in watches, that vibrates whenever energy is applied to it.
They are sometimes called buzzers or speakers, but buzzers is more appropriate since they can only play tones, and not complicated sound effects.

As far as functionality is concerned, the **active speaker has active electrical components built into it.** The passive one is **just piezoelectric material** so it needs active components externally in order to generate the wave and work.

Let's make the simplest connection we can make. You can recognise the + and - pins by their length (+ is longer) or you can look at the case, as they both have a **+** written on the top.

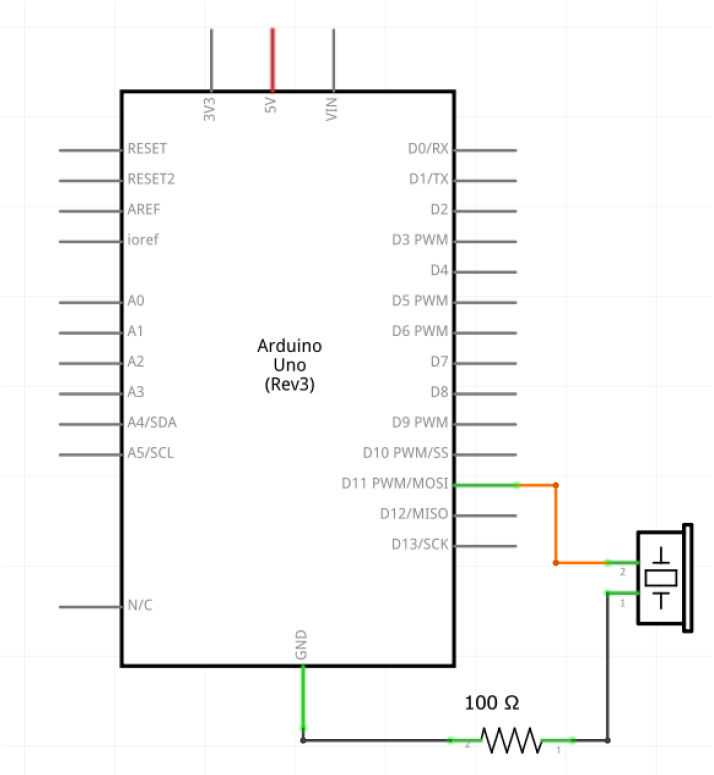
Check with the multimeter diode setting (continuity) which type of buzzer you have. If it buzzes, it's active. If not, it's passive.

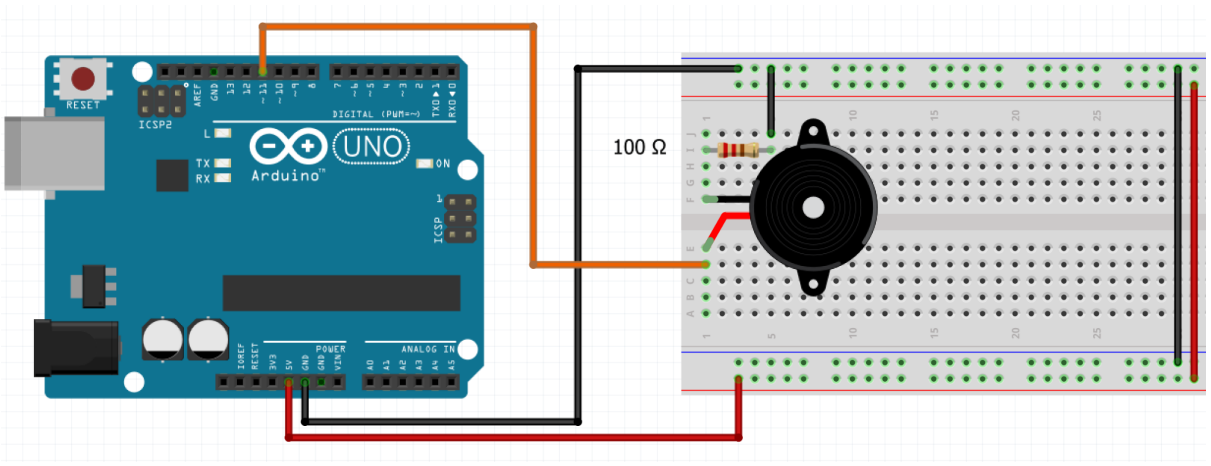
Why is that?

A: because the active buzzer can convert a straight line signal (3V, 5V etc) into sound because of the active components. The passive speaker needs a varying voltage (**PWM**). The good thing about the **PWM** style is that it works with both passive and active speakers.

7.1 Connecting the buzzer to arduino

Now, let's control the tone. Connect the **+** to **D11.** You can use either the passive or the active buzzer, but let's use the passive one just to see it works.





7.2 Introducing the tone() function

The **tone()** function generates a square wave of the specified frequency (and 50% duty cycle) on a pin. A duration can be specified, otherwise the wave continues until a call to noTone(). The pin can be connected to a piezo buzzer or other speaker to play tones.

Only one tone can be generated at a time. If a tone is already playing on a different pin, the call to tone() will have no effect. If the tone is playing on the same pin, the call will set its frequency.

Attention! Use of the tone() function will interfere with PWM output on pins 3 and 11.

Syntax:

```
tone(pin, frequency)
tone(pin, frequency, duration)
```

Parameters
pin: the Arduino pin on which to generate the tone.
frequency: the frequency of the tone in hertz. Allowed data types: unsigned int.
duration: the duration of the tone in milliseconds (optional). Allowed data types: unsigned long.

(source: https://www.arduino.cc/en/Tutorial/toneMelody)

The tone() function is all that is needed to play a tone. **Careful, this gets annoyingly fun fast!**

```
const int buzzerPin = 11;
int buzzerTone = 1000;

void setup() {
}

void loop() {
  tone(buzzerPin, buzzerTone, 500);
}
```

Now, let's cycle through the tones a bit.

```
const int buzzerPin = 11;
int buzzerTone = 1000;

void setup() {

}

void loop() {
  tone(buzzerPin, buzzerTone, 500);
```

```
  delay(100);

  noTone(buzzerPin);
  delay(100);
  buzzerTone += 50;
}
```

Change both delays to 10 instead of 100.

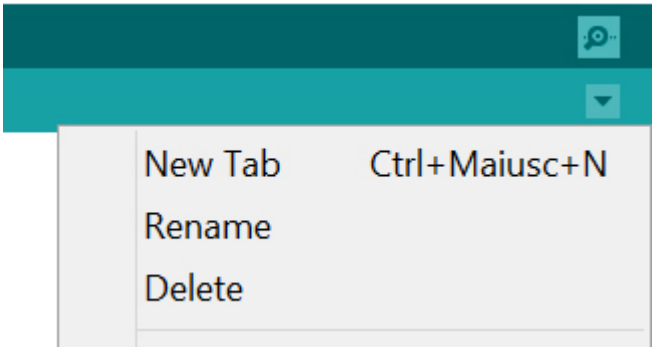
Question:

1. Where would these be useful?
    a. A: in a game where you need sounds (a matrix game is coming), in alarms and generally in projects where you need audio feedback.

7.3 Playing a song

Copy paste this code.

Prerequisite: in your project, create a new tab and add these: (CTRL + SHIFT + T)




```
/*************************************************
 * Public Constants
 *************************************************/

#define NOTE_B0  31
#define NOTE_C1  33
#define NOTE_CS1 35
#define NOTE_D1  37
#define NOTE_DS1 39
#define NOTE_E1  41
#define NOTE_F1  44
#define NOTE_FS1 46
#define NOTE_G1  49
#define NOTE_GS1 52
```

```
#define NOTE_A1  55
#define NOTE_AS1 58
#define NOTE_B1  62
#define NOTE_C2  65
#define NOTE_CS2 69
#define NOTE_D2  73
#define NOTE_DS2 78
#define NOTE_E2  82
#define NOTE_F2  87
#define NOTE_FS2 93
#define NOTE_G2  98
#define NOTE_GS2 104
#define NOTE_A2  110
#define NOTE_AS2 117
#define NOTE_B2  123
#define NOTE_C3  131
#define NOTE_CS3 139
#define NOTE_D3  147
#define NOTE_DS3 156
#define NOTE_E3  165
#define NOTE_F3  175
#define NOTE_FS3 185
#define NOTE_G3  196
#define NOTE_GS3 208
#define NOTE_A3  220
#define NOTE_AS3 233
#define NOTE_B3  247
#define NOTE_C4  262
#define NOTE_CS4 277
#define NOTE_D4  294
#define NOTE_DS4 311
#define NOTE_E4  330
#define NOTE_F4  349
#define NOTE_FS4 370
#define NOTE_G4  392
#define NOTE_GS4 415
#define NOTE_A4  440
#define NOTE_AS4 466
#define NOTE_B4  494
#define NOTE_C5  523
#define NOTE_CS5 554
#define NOTE_D5  587
#define NOTE_DS5 622
#define NOTE_E5  659
```

```
#define NOTE_F5  698
#define NOTE_FS5 740
#define NOTE_G5  784
#define NOTE_GS5 831
#define NOTE_A5  880
#define NOTE_AS5 932
#define NOTE_B5  988
#define NOTE_C6  1047
#define NOTE_CS6 1109
#define NOTE_D6  1175
#define NOTE_DS6 1245
#define NOTE_E6  1319
#define NOTE_F6  1397
#define NOTE_FS6 1480
#define NOTE_G6  1568
#define NOTE_GS6 1661
#define NOTE_A6  1760
#define NOTE_AS6 1865
#define NOTE_B6  1976
#define NOTE_C7  2093
#define NOTE_CS7 2217
#define NOTE_D7  2349
#define NOTE_DS7 2489
#define NOTE_E7  2637
#define NOTE_F7  2794
#define NOTE_FS7 2960
#define NOTE_G7  3136
#define NOTE_GS7 3322
#define NOTE_A7  3520
#define NOTE_AS7 3729
#define NOTE_B7  3951
#define NOTE_C8  4186
#define NOTE_CS8 4435
#define NOTE_D8  4699
#define NOTE_DS8 4978
```

```
#include "pitches.h"
const int buzzerPin = 11;
// notes in the melody:
int melody[] = {
  NOTE_C4, NOTE_G3, NOTE_G3, NOTE_A3, NOTE_G3, 0, NOTE_B3, NOTE_C4
};

// note durations: 4 = quarter note, 8 = eighth note, etc.:
int noteDurations[] = {
  4, 8, 8, 4, 4, 4, 4, 4
};

void setup() {
  // iterate over the notes of the melody:
  for (int thisNote = 0; thisNote < 8; thisNote++) {
          // to calculate the note duration, take one second divided by the note type.
          //e.g. quarter note = 1000 / 4, eighth note = 1000/8, etc.
          int noteDuration = 1000 / noteDurations[thisNote];
          tone(buzzerPin, melody[thisNote], noteDuration);

          // to distinguish the notes, set a minimum time between them.
          // the note's duration + 30% seems to work well:
          int pauseBetweenNotes = noteDuration * 1.30;
          delay(pauseBetweenNotes);
          // stop the tone playing:
          noTone(8);
  }
}

void loop() {
  // no need to repeat the melody.
}
```

Source: https://www.arduino.cc/en/Tutorial/toneMelody (with modified pin)