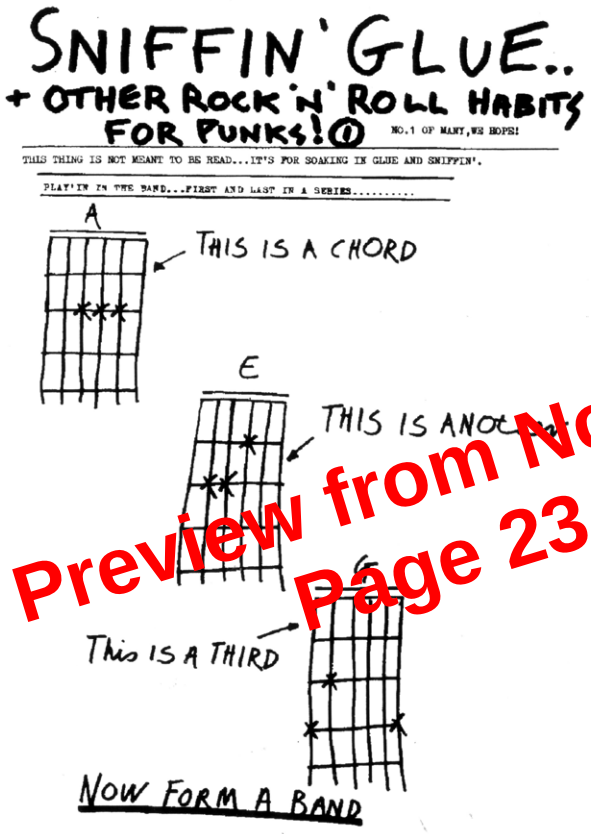
Tinkering

We believe that it is essential to play with technology, exploring different possibilities directly on hardware and software—sometimes without a very defined goal.

Reusing existing technology is one of the best ways of tinkering. Getting cheap toys or old discarded equipment and hacking them to make them do something new is one of the best ways to get to great results.



It's a bit like the *Sniffin' Glue* fanzine shown here: during the punk era, knowing three chords on a guitar was enough to start a band. Don't let the experts in one field tell you that you'll never be one of them. Ignore them and surprise them.

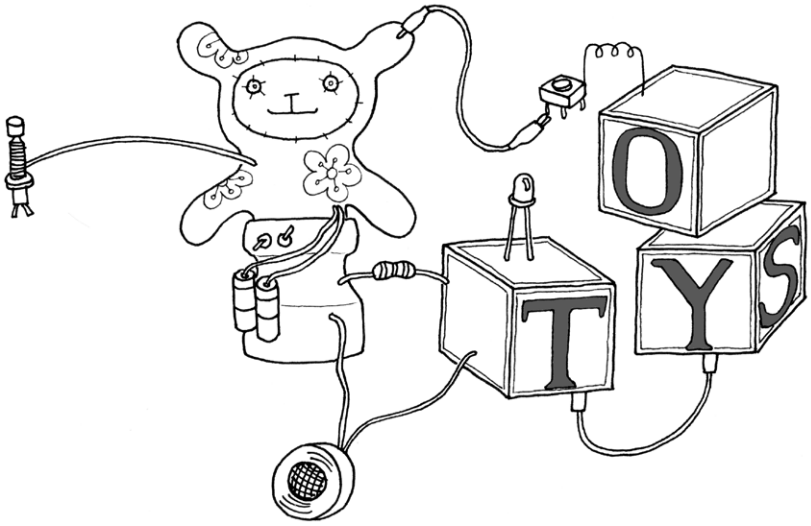
Computer keyboards are still the main way to interact with a computer after more than 60 years. Alex Pentland, academic head of the MIT Media Laboratory, once remarked: "Excuse the expression, but men's urinals are smarter than computers. Computers are isolated from what's around them."[1]

As tinkerers, we can implement new ways to interact with software by replacing the keys with devices that are able to sense the environment. Taking apart a computer keyboard reveals a very simple (and cheap) device. The heart of it is a small board. It's normally a smelly green or brown circuit with two sets of contacts going to two plastic layers that hold the connections between the different keys. If you remove the circuit and use a wire to bridge two contacts, you'll see a letter appear on the computer screen. If you go out and buy a motion sensing detector and connect this to your keyboard, you'll see a key being pressed every time somebody walks in front of the computer. Map this to your favourite software, and you have made your computer as smart as a urinal. Learning about keyboard hacking is a key building block of prototyping and Physical Computing.

[1]Quoted in Sara Reese Hedberg, "MIT Media Lab's quest for perceptive computers," *Intelligent Systems and Their Applications*, IEEE, Jul/Aug 1998.

Hacking Toys

Toys are a fantastic source of cheap technology to hack and reuse, as evidenced by the practise of circuit bending mentioned earlier. With the current influx of thousands of very cheap high-tech toys from China, you can build quick ideas with a few noisy cats and a couple of light swords. I have been doing this for a few years to get my students to understand that technology is not scary or difficult to approach. One of my favourite resources is the booklet "Low Tech Sensors and Actuators" by Usman Haque and Adam Somlai-Fischer (lowtech.propositions.org.uk). I think that they have perfectly described this technique in that handbook, and I have been using it ever since.



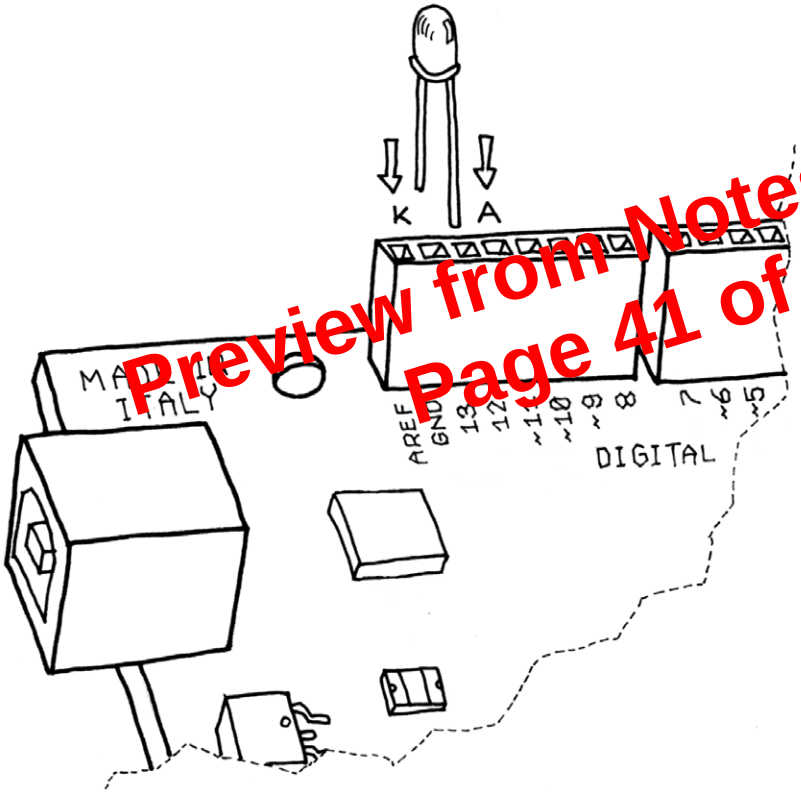


Figure 4-2.
Connecting an LED to Arduino

as our brain. So to group together a number of instructions, you stick a *{* before your code and an *}* after.

You can see that there are two blocks of code that are defined in this way here. Before each one of them there is a strange command:

```
void setup()
```

This line gives a name to a block of code. If you were to write a list of instructions that teach Arduino how to pass the Parmesan, you would write *void passTheParmesan()* at the beginning of a block, and this block would become an instruction that you can call from anywhere in the Arduino code. These blocks are called **functions**. If after this, you write *passTheParmesan()* anywhere in your code, Arduino will execute those instructions and continue where it left off.

Arduino Is Not for Quitters

Arduino expects two functions to exist—one called *setup()* and one called *loop()*.

setup() is where you put all the code that you want to execute once at the beginning of your program and *loop()* contains the core of your program, which is executed over and over again. This is done because Arduino is not like your regular computer—it cannot run multiple programs at the same time and programs can't quit. When you power up the board, the code runs; when you want to stop, you just turn it off.

Real Tinkerers Write Comments

Any text beginning with // is ignored by Arduino. These lines are comments, which are notes that you leave in the program for yourself, so that you can remember what you did when you wrote it, or for somebody else, so that they can understand your code.

It is very common (I know this because I do it all the time) to write a piece of code, upload it onto the board, and say "Okay—I'm never going to have to touch this sucker again!" only to realise six months later that you need to update the code or fix a bug. At this point, you open up the program, and if you haven't included any comments in the original program, you'll think, "Wow—what a mess! Where do I start?" As we move along, you'll see some tricks for how to make your programs more readable and easier to maintain.

chill. How does this work? Well, imagine that the battery is both a water reservoir and a pump, the switch is a tap, and the motor is one of those wheels that you see in watermills. When you open the tap, water flows from the pump and pushes the wheel into motion.

In this simple hydraulic system, shown in Figure 4-5, two factors are important: the pressure of the water (this is determined by the power of pump) and the amount of water that will flow in the pipes (this depends on the size of the pipes and the resistance that the wheel will provide to the stream of water hitting it).

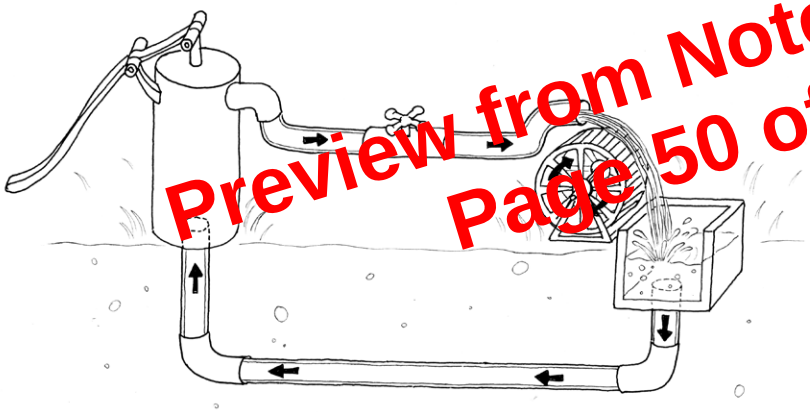


Figure 4-5.
A hydraulic system

You'll quickly realise that if you want the wheel to spin faster, you need to increase the size of the pipes (but this works only up to a point) and increase the pressure that the pump can achieve. Increasing the size of the pipes allows a greater flow of water to go through them; by making them bigger, we have effectively reduced the pipes' resistance to the flow of water. This approach works up to a certain point, at which the wheel won't spin any faster, because the pressure of the water is not strong enough. When we reach this point, we need the pump to be stronger. This method of speeding up the watermill can go on until the point when the wheel falls apart because the water flow is too strong for it and it is destroyed. Another thing you will notice is that as the wheel spins, the axle will heat up a little bit, because no matter how well we have mounted the wheel,

```
// Example 03B: Turn on LED when the button is pressed
// and keep it on after it is released
// Now with a new and improved formula!


const int LED = 13;   // the pin for the LED
const int BUTTON = 7; // the input pin where the
                      // pushbutton is connected
int val = 0;     // val will be used to store the state
                 // of the input pin
int old_val = 0; // this variable stores the previous
                 // value of "val"
int state = 0;   // 0 = LED off and 1 = LED on

void setup() {
  pinMode(LED, OUTPUT);  // tell Arduino LED is an output
  pinMode(BUTTON, INPUT); // and BUTTON is an input
}
void loop(){
  val = digitalRead(BUTTON); // read input value and store it
                             // yum, fresh

  // check if there was a transition
  if ((val == HIGH) && (old_val == LOW)){
    state = 1 - state;
  }

  old_val = val;  // val is now old, let's store it

  if (state == 1) {
    digitalWrite(LED, HIGH); // turn LED ON
  } else {
    digitalWrite(LED, LOW);
  }
}
```


Test it: we're almost there!

You may have noticed that this approach is not entirely perfect, due to another issue with mechanical switches. Pushbuttons are very simple devices: two bits of metal kept apart by a spring. When you press the

button, the two contacts come together and electricity can flow. This sounds fine and simple, but in real life the connection is not that perfect, especially when the button is not completely pressed, and it generates some spurious signals called **bouncing**.

When the pushbutton is bouncing, the Arduino sees a very rapid sequence of on and off signals. There are many techniques developed to do de-bouncing, but in this simple piece of code I've noticed that it's usually enough to add a 10- to 50-millisecond delay when the code detects a transition.

Example 03C is the final code:

Tilt switches

A simple electronic component that contains two contacts and a little metal ball (or a drop of mercury, but I don't recommend using those) An example of a tilt switch is called a tilt sensor. Figure 5-1 shows the inside of a typical model. When the sensor is in its upright position, the ball bridges the two contacts, and this works just as if you had pressed a pushbutton. When you tilt this sensor, the ball moves, and the contact is opened, which is just as if you had released a pushbutton. Using this simple component, you can implement, for example, gestural interfaces that react when an object is moved or shaken (bit.ly/ArduinoStoreTiltSensor).

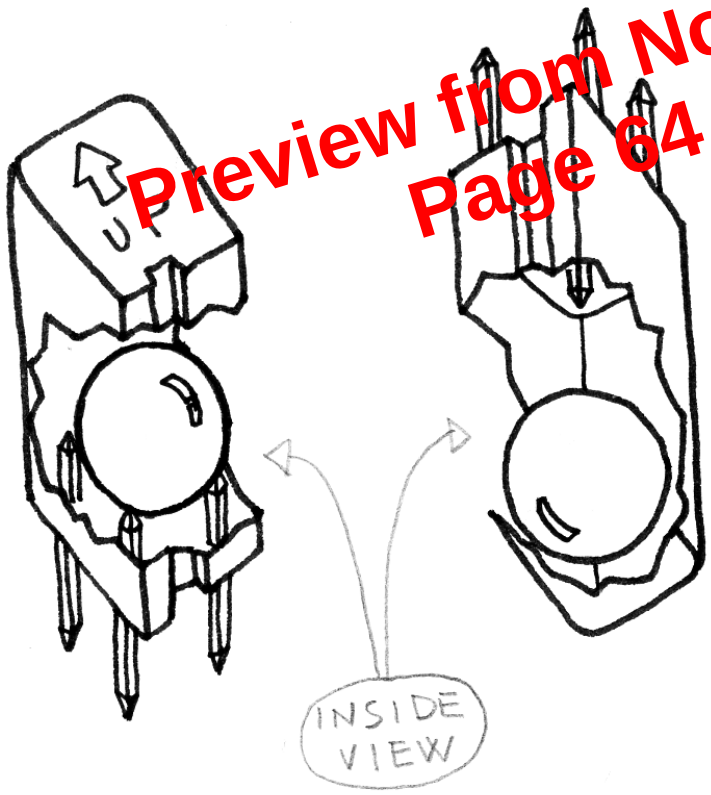


Figure 5-1.
The inside of a tilt sensor

For example, writing *analogWrite(9,128)* will set the brightness of an LED connected to pin 9 to 50%. Why 128? *analogWrite()* expects a number between 0 and 255 as an argument, where 255 means full brightness and 0 means off.

NOTE: Having three channels is very good, because if you buy red, green, and blue LEDs, you can mix their lights and make light of any colour that you like!

Let's try it out. Build the circuit that you see in Figure 5-4. Note that LEDs are polarized: the long pin (positive) should go to the right, and the short pin (negative) to the left. Also, most LEDs have a flattened negative side, as shown in the figure. Use a 270 ohm resistor (red violet brown).

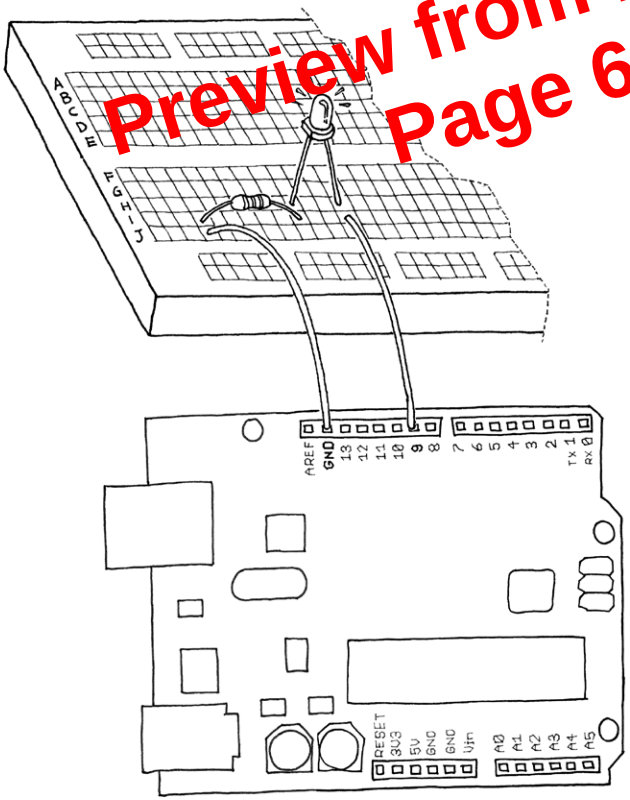


Figure 5-4.
LED connected to PWM pin

To create this circuit, you will need to combine the circuit you just built (shown in Figure 5-4) with the pushbutton circuit shown in Figure 4-6. If you'd like, you can simply build both circuits on different parts of the breadboard; you have plenty of room. However, one of the advantages of the breadboard (see Appendix A) is that there is a pair of rails running horizontally across the bottom and top. One is marked red (for positive) and the other blue or black (for ground).

These rails are used to distribute power and ground to where it's needed. In the case of the circuit you need to build for this example, you have two components (both of them resistors) that need to be connected to the GND (ground) pin on the Arduino. Because the Arduino has three GND pins, you could simply connect these two circuits exactly as shown in each of the two figures; just hook them both up to the Arduino at the same time. Or, you could connect one wire from the breadboard's ground rail to one of the GND pins on the Arduino, and then take the wires that are connected to GND in the figures and connect them instead to the breadboard ground rail.

If you're not ready to try this, don't worry: simply wire up both circuits to your Arduino as shown in Figures 4-6 and 5-4. You'll see an example that uses the ground and positive breadboard rails in Chapter 6.

Getting back to this next example, if we have just one pushbutton, how do we control the brightness of a lamp? We're going to learn yet another interaction design technique: detecting how long a button has been pressed. To do this, I need to upgrade example 03C from Chapter 4 to add dimming. The idea is to build an "interface" in which a press and release action switches the light on and off, and a press and hold action changes brightness.

Let's have a look at the sketch:

characters. Predictability also makes the code simpler: we wait until we see an #, then we read the six characters that follow into a *buffer* (a variable used as a temporary holding area for data). Finally, we turn each group of two characters into a byte that represents the brightness of one of the three LEDs.

Coding

There are two sketches that you'll be running: one Processing sketch, and one Arduino sketch. Here is the code for the Processing sketch. You can download it from www.makezine.com/getstartedarduino.

```
// Example 08A: Arduino networked lamp
// parts of the code are inspired
// by a blog post by Tod E. Kurt (todbot.com)

import processing.serial.*;

String feed = "http://blog.makezine.com/index.xml";

int interval = 10;  // retrieve feed every 60 seconds;
int lastTime;       // the last time we fetched the content

int love    = 0;
int peace   = 0;
int arduino = 0;

int light = 0;  // light level measured by the lamp

Serial port;
color c;
String cs;

String buffer = ""; // Accumulates characters coming from Arduino

PFont font;

void setup() {
  size(640,480);
  frameRate(10);    // we don't need fast updates

  font = loadFont("HelveticaNeue-Bold-32.vlw");
  fill(255);
  textFont(font, 32);
```

```
  // IMPORTANT NOTE:
  // The first serial port retrieved by Serial.list()
  // should be your Arduino. If not, uncomment the next
  // line by deleting the // before it, and re-run the
  // sketch to see a list of serial ports. Then, change
  // the 0 in between [ and ] to the number of the port
  // that your Arduino is connected to.
  //println(Serial.list());
  String arduinoPort = Serial.list()[0];
  port = new Serial(this, arduinoPort, 9600); // connect to Arduino

  lastTime = 0;
  fetchData();
}

void draw() {
  background( c );
  int n = (interval - (millis()-lastTime)/1000);

  // Build a colour based on the  values
  c = color(peace, love, arduino);
  cs = "#" + hex(c,6); // Prepare a string to be sent to Arduino

  text("Arduino Networked Lamp", 10,40);
  text("Reading feed:", 10, 100);
  text(feed, 10, 140);

  text("Next update in "+ n + " seconds",10,450);
  text("peace" ,10,200);
  text(" " + peace, 130, 200);
  rect(200,172, peace, 28);

  text("love ",10,240);
  text(" " + love, 130, 240);
  rect(200,212, love, 28);

  text("arduino ",10,280);
  text(" " + arduino, 130, 280);
  rect(200,252, arduino, 28);

  // write the colour string to the screen
  text("sending", 10, 340);
  text(cs, 200,340);
```

7/Troubleshooting

There will come a moment in your experimentation when nothing will be working and you will have to figure out how to fix it. Troubleshooting and debugging are ancient arts in which there are a few simple rules, but most of the results are obtained through a lot of work.

The more you work with electronics and Arduino, the more you will learn and gain experience, which will ultimately make the process less painful. Don't be discouraged by the problems that you will find—it's all easier than it seems at the beginning.

As every Arduino-based project is made both of hardware and software, there will be more than one place to look if something goes wrong. While looking for a bug, you should operate along three lines:

Understanding
Try to understand as much as possible how the parts that you're using work and how they're supposed to contribute to the finished project. This approach will allow you to devise some way to test each component separately.

Simplification and segmentation
The Ancient Romans used to say *divide et impera*: divide and rule. Try to break down (mentally) the project into its components by using the understanding you have and figure out where the responsibility of each component begins and ends.

Exclusion and certainty
While investigating, test each component separately so that you can be absolutely certain that each one works by itself. You will gradually build up confidence about which parts of project are doing their job and which ones are dubious.

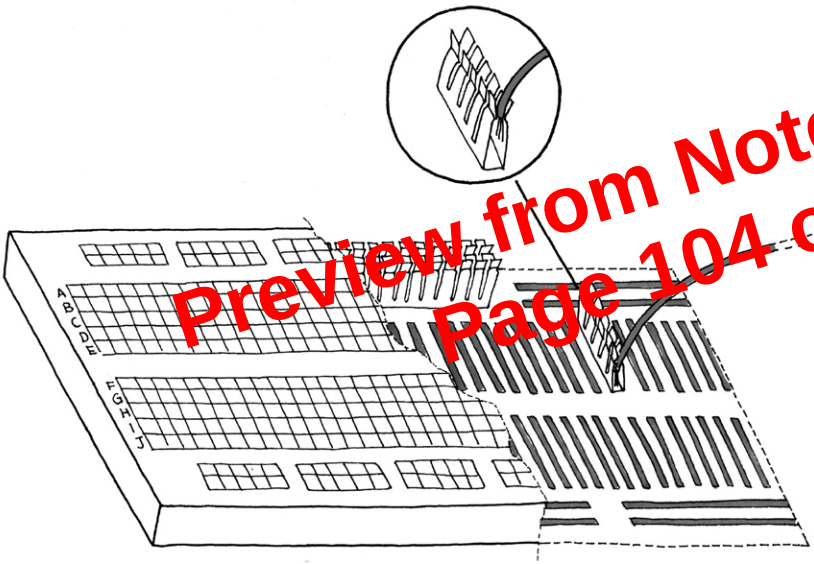


Figure A-1.
The solderless breadboard

Appendix B/Reading Resistors and Capacitors

In order to use electronic parts, you need to be able to identify them, which can be a difficult task for a beginner. Most of the resistors that you find in a shop have a cylindrical body with two legs sticking out and have strange coloured markings all around them. When the first commercial resistors were made, there was no way to print numbers small enough to fit on their body, so clever engineers decided that they could just represent the values with strips of coloured paint.

Today's beginners have to figure out a way to interpret these signs. The "key" is quite simple: generally, there are four stripes, and each colour represents a number. One of rings is usually gold coloured; this one represents the precision of that resistor. To read the stripes in order, hold the resistor so the gold (or silver in some cases) stripe is to the right. Then, read the colours and map them to the corresponding numbers. In the following table, you'll find a translation between the colours and their numeric values.

Colour	Value
Black	0
Brown	1
Red	2
Orange	3
Yellow	4
Green	5
Blue	6
Violet	7
Grey	8
White	9
Silver	10%
Gold	5%

For example, brown, black, orange, and gold markings mean 1 0 3 ±5%. Easy, right? Not quite, because there is a twist: the third ring actually represents the number of zeros in the value. Therefore 1 0 3 is actually 1 0 followed by 3 zeros, so the end result is 10,000 ohms ±5%. Electronics geeks tend to shorten values by expressing them in kilo ohm (for thousands

ARITHMETIC AND FORMULAS

You can use Arduino to make complex calculations using a special syntax. + and – work like you've learned in school, and multiplication is represented with an * and division with a /.

There is an additional operator called "modulo" (%), which returns the remainder of an integer division. You can use as many levels of parentheses as necessary to group expressions. Contrary to what you might have learned in school, square brackets and curly brackets are reserved for other purposes (array indexes and blocks, respectively).

Examples:

```
a =  2 + 2;
light = ((12 * sensorValue) - 5 ) / 2;
remainder = 3 % 2; // returns 1
```


COMPARISON OPERATORS

When you specify conditions or tests for *if*, *while*, and *for* statements, these are the operators you can use:

- == equal to
- != not equal to
- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to

BOOLEAN OPERATORS

These are used when you want to combine multiple conditions. For example, if you want to check whether the value coming from a sensor is between 5 and 10, you would write:

```
if ((sensor => 5) && (sensor <=10))
```

There are three operators: and, represented with **&&**; or, represented with **||**; and finally not, represented with **!**.

COMPOUND OPERATORS

These are special operators used to make code more concise for some very common operations like incrementing a value.

R

S

W

X