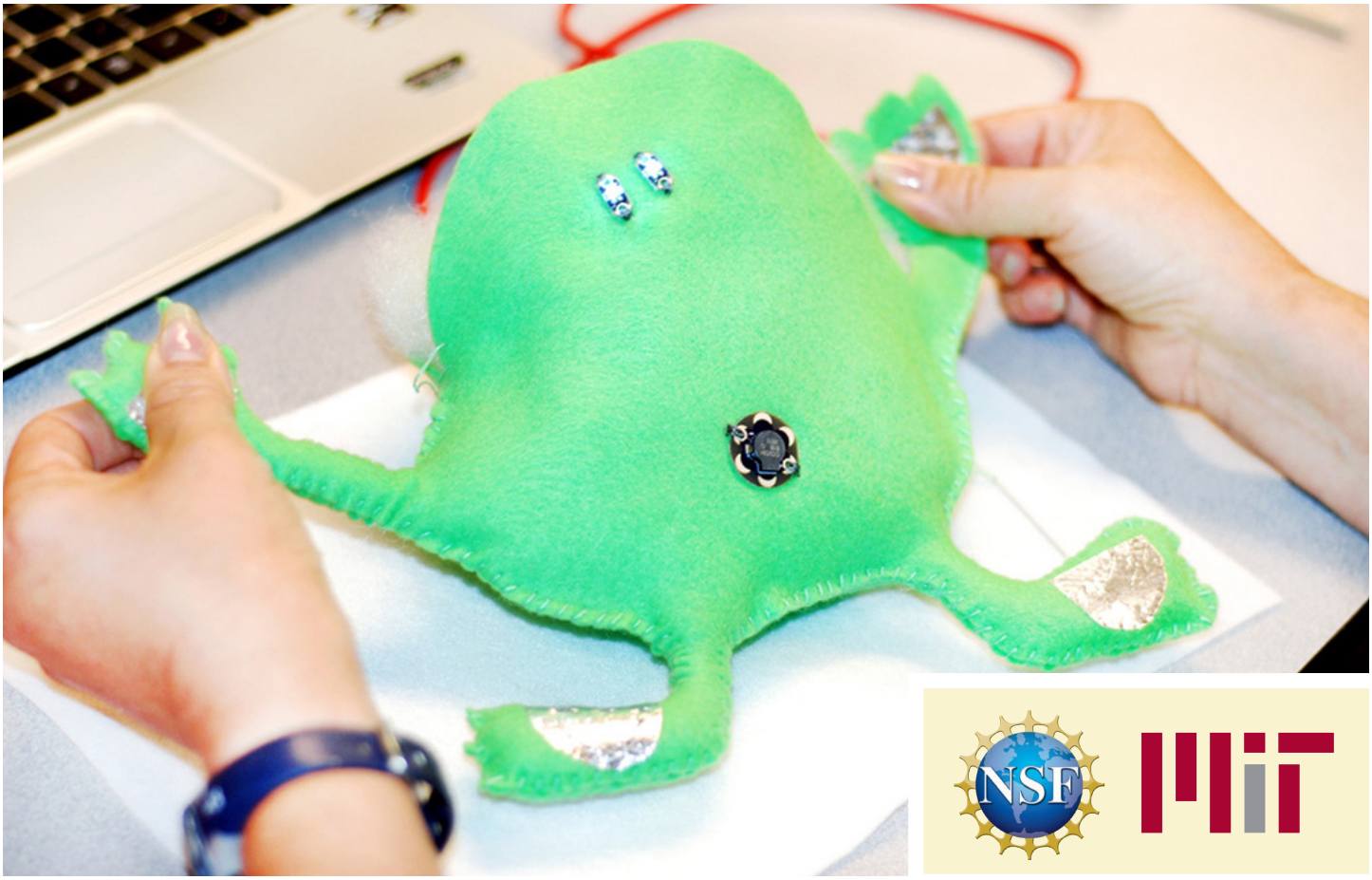
national center for women & INFORMATION TECHNOLOGY

E-TEXTILES-IN-A-BOX

WWW.NCWIT.ORG/ETEXTILES



The e-Textiles-in-a-Box tutorial helps adults get ready to teach young people about electronics and computing. Based on the Computational Textiles Curriculum and Sew Electric from MIT, e-Textiles-in-a-Box provides instructions for sewing soft circuits and programming an Arduino microprocessor on the way to creating a bookmark book light and an interactive felt monster that lights up and sings. NCWIT is pleased to offer e-Textiles-in-a-Box in cooperation with the MIT High-Low Tech Group, and with funding from the National Science Foundation.

NCWIT provides sample electronics and microprocessor kits to those wishing to learn e-Textiles in order to teach others. Learn more inside!

National Center for Women & Information Technology (NCWIT)
www.ncwit.org | info@ncwit.org | Twitter: @NCWIT | 303.735.6671

Practical Guide

About this Resource

NCWIT is pleased to offer e-Textiles-in-a-Box in cooperation with authors Leah Buechley and Kanjun Qiu from the High-Low Tech Group at MIT. Sonja de Boer contributed all illustrations. e-Textiles-in-a-Box was adapted for this resource from Buechley and Qui's Computational Textiles Curriculum by Jane Krauss and Stephanie Weber and was piloted by Thompson School District in Loveland, Colorado. For further exploration, readers are directed to the book Sew Electric, a collection of DIY projects that combine fabric, electronics, and programming. Learn more at http://sewelectric.org.

This practical guide introduces e-Textiles-in-a-Box and gives adult learners steps for getting started. It also includes a Materials Reference section and a Glossary of terms used in the lessons.

Introducing e-Textiles-in-a-Box

e-Textiles-in-a-Box is a set of four self-guided lessons designed for adults who wish to learn computer science and who anticipate teaching others. The activities in the "box" combine crafting with electronics and computer programming. All lessons leading to a culminating activity: the creation and programming of a stuffed "monster" that lights up and plays a tune.

The intent of this "program-in-a-box" is to help teachers and other instructors learn programming through e-Textiles so they can then introduce students — especially girls — to computer science in an engaging way.

e-Textiles-in-a-Box is a joint effort of MIT and NCWIT and is funded by the National Science Foundation. It is based on the work of Dr. Leah Buechley, Associate Professor at the MIT Media Lab. At MIT, Dr. Buechley directs the High-Low Tech research group, which explores the integration of high and low technology from cultural, material, and practical perspectives with the goal of engaging diverse groups of people in developing their own technologies.

Research indicates that hands-on, craft-centered computing projects such as e-Textiles are enticing to girls and serve as an impetus for continued exploration of computer science. e-Textiles-in-a-Box is intended as a turnkey project through which adults become agents for bringing these experiences to girls.

The grant that funds the project pays for sample Arduino kits (one per user) containing the computing and electronic components necessary for the project. Other components — mostly crafting supplies — are not included. See the Materials Reference section that follows for a complete list.

Before starting the tutorial, readers need to understand the parameters of the project and secure materials.

FOUR LESSONS

The four lessons are sequential and build on one another. Here is a brief description of each:

1. INTRODUCTION TO E-TEXTILES (SCREENCAST)

Time: 15 minutes

The first lesson is a short video that explains the history and development of e-textiles and introduces the lessons ahead. Watch the video here: [www.ncwit.org/intro_e-textiles].

2. BOOKMARK BOOK LIGHT: SEWING SIMPLE CIRCUITS

Time: 1–2 hours

The second lesson introduces basic electronics. Participants design a felt bookmark and sew circuits with electronic thread to form a circuit between a battery and an LED.

3. INTRODUCTION TO PROGRAMMING

Time: 1–3 hours if Arduino software is preinstalled, 3–5 hours if installing Arduino software

In the third lesson, participants learn basic computer science as they program an Arduino microprocessor to display a blink pattern. The Arduino programming environment can be downloaded for use on a PC or MacIntosh computer. The Arduino programming language is based on C++.

4. INTERACTIVE STUFFED MONSTER

Time: 10–15 hours

In the fourth lesson, prior learning comes together in the creation of an interactive monster that lights up and plays a tune. In this lesson, participants do creative crafting, sew electronic circuits, program light patterns, and create a touch-sensitive switch that causes a tune they have programmed to play.

MATERIALS

Through Sparkfun Electronics, NCWIT provides each user with a kit that includes the electronics and computing components for the e-Textiles tutorial. To order the technical materials from Sparkfun, please complete the resource request on the page where you downloaded this box: www.ncwit.org/e-textiles. You can expect delivery of the kit within two weeks of ordering.

The Materials Reference section below describes the other items that you will need to complete the tutorial. These items are not provided by NCWIT, but you probably have many of these items at home, and you can easily purchase the rest at a craft or hobby store.

Advice and Troubleshooting

The tutorial is designed for adult learners who have no prior experience in electronics and computer science. With time and perseverance, you can become a creative and crafty programmer!

If you have a background in electronics and computing, we recommend that you still go through each tutorial before you begin to teach others. By doing so, you will be better able to anticipate the challenges that students may face and be ready to supplement the lessons or give support as needed.

TIME

Set aside adequate blocks of time to dig in. As you will see, the lessons require concentration, flexible thinking, and problem solving. You will feel more productive and better equipped to teach others if you work in longer, uninterrupted intervals.

SPACE

Set up a space where you can lay out craft materials and work in good light.

GETTING HELP

Our first advice is to be your own best helper. Strategies for progressing independently include previewing each lesson in its entirety, rereading, tracking your trial-and-error attempts, looking ahead, and taking breaks. The tutorial includes a glossary and resource list to help you with unfamiliar terms and to identify technical components.

When you need more help with programming, refer to the Troubleshooting section at the end of each lesson. Both the e-Textiles and Arduino websites offer FAQs and troubleshooting pages, as well. These are noted in the lessons at appropriate spots.

As a last resort, call on people with more computer science or crafting experience to help you.

Our best advice may be to pair up with another new learner. Working with another person of similar ability will cause you to think aloud, and in doing so, resolve challenges that may stump you when working solo.

Beyond the Tutorial

With proper support, the e-textiles projects in this tutorial are appropriate for students in middle school through high school.

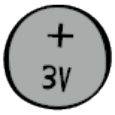
You can use elements of this tutorial in your teaching, but in addition, you should plan to explain concepts, model certain steps, and otherwise provide scaffolding for the learning experience.

The cost of offering a camp, club, or class e-Textiles experience is approximately $46.00 per student project. Consider grouping students into teams of two or three to reduce expense.

Materials Reference

ELECTRONIC MATERIALS & TOOLS

COIN CELL BATTERY

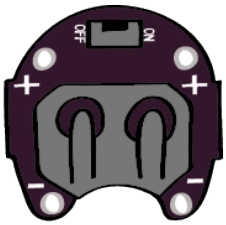


A 3-volt battery whose code name is CR2032. This code means that the battery is 20mm in diameter and 3.2mm thick. Flat, round batteries like these are called coin or button cell batteries because of their shape.

Where to buy? Online retailer like Amazon. Sparkfun sells this battery, product #338, but its slightly more expensive.

Approximate cost: $0.50

LILYPAD COIN CELL BATTERY HOLDER



A sewable battery holder for the CR2032 coin cell battery.

Where to buy? https://www.sparkfun.com/products/11285

Approximate cost: $6.00

Alternate options: If you want to make your own battery holder out of fabric you can find a tutorial here: http://www.kobakant.at/DIY/?p=52

CONDUCTIVE THREAD

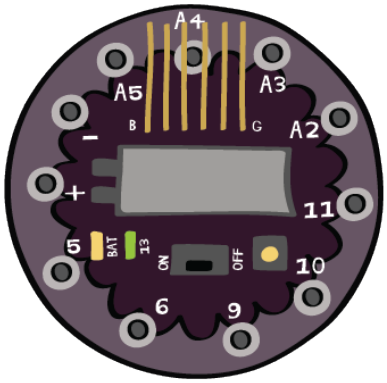
A thread capable of carrying electric current. The thread referred to in this book is a stainless steel thread spun from fine stainless steel wires. Other types of conductive thread include silver-plated threads and gold-wrapped embroidery threads.

Where to buy? https://www.sparkfun.com/products/10867

Approximate cost: $3.00

Alternate options: You can purchase silver-plated thread from plug and wear: http://www.plugandwear.com/

LILYPAD ARDUINO SIMPLESNAP

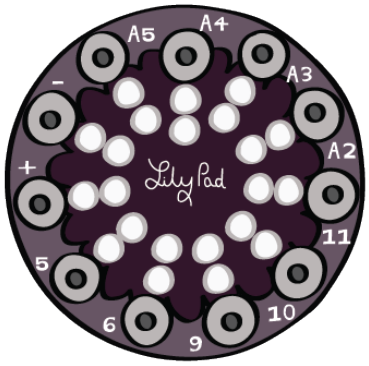


A small snap-on computer. This programmable LilyPad Arduino board contains an ATmega328 microcontroller and a built-in rechargeable battery. For more information see: http://lilypadarduino.org/?p=289

Where to buy? https://www.sparkfun.com/products/10941

Approximate cost: $30.00

Alternate options: You can use any other LilyPad Arduino board to replace the SimpleSnap and Protoboard combination. For more information see: http://lilypadarduino.org/?cat=4



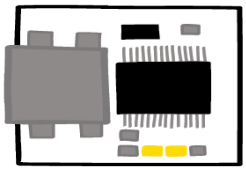
LILYPAD ARDUINO PROTOBOARD

A sewable board with a ring of male snaps on its outer edge that mates with the LilyPad Arduino SimpleSnap. Sew this board to your project to be able to snap the LilyPad Arduino SimpleSnap on and off of it.

Where to buy? https://www.sparkfun.com/products/10940

Approximate cost: $10.00

Alternate options: You can replace the protoboard with sew-on snaps (size 1/0) or riveting snaps (size 10).
For more information see: http://lilypadarduino.org/?p=289



FTDI BREAK OUT BOARD / PROGRAMMING BOARD

The board that connects a LilyPad Arduino board to a computer so that it can be programmed.

Where to buy? https://www.sparkfun.com/products/10275

Approximate cost: $15.00

Alternate options: You can replace the FTDI Breakout Board and USB cable with an integrated FTDI cable:
https://www.sparkfun.com/products/9718

LILYPAD SPEAKER / LILYPAD BUZZER

A small sewable speaker.

Where to buy? https://www.sparkfun.com/products/8463

Approximate cost: $8.00

LILYPAD LED

A small sewable LED (light-emitting diode).

Where to buy? https://www.sparkfun.com/products/10081

Approximate cost: $1.00

Alternate options: You can replace LilyPad LEDs with standard "through hole" LEDs. Simply twist their legs into sewable loops like so: You can purchase through hole LEDs from your local Radioshack or sparkfun:
https://www.sparkfun.com/products/9590

MINI-USB CABLE

A USB 2.0 A to Mini-B Cable. The cable that connects the FTDI Breakout Board/Programming board to your computer. This type of cable is used to connect many cameras and cell phones to computers and chargers. Check to see if you already own one before you buy a new one.

Where to buy? Online retailer like Amazon. Sparkfun sells a red USB cable, product #598.

Approximate cost: $5.00

Alternate options: You can replace the FTDI Breakout Board and USB cable with an integrated FTDI cable:
https://www.sparkfun.com/products/9718

Materials Reference

CRAFT MATERIALS & TOOLS

EMBROIDERY THREAD OR "FLOSS"

A special heavy-weight thread used for embroidery.

Where to buy? Your local craft store, or an online retailer like Amazon.

Approximate cost: $1.00

GLUE

Any standard craft glue will work well for sealing knots, including fabric glue or Elmers® glue. Use a fabric glue or Iron-On adhesive to attach one piece of fabric to another.

Where to buy? Your local craft store, or an online retailer like Amazon.

Approximate cost: $5.00

LARGE-EYED NEEDLE

"Chenille" needles sized 18-24 are very easy to thread and fit through the holes in LilyPad pieces. If you are a more experienced sewer, you may want to use a smaller needle, but be careful, smaller needles are much harder to thread.

Where to buy? https://www.sparkfun.com/products/10405

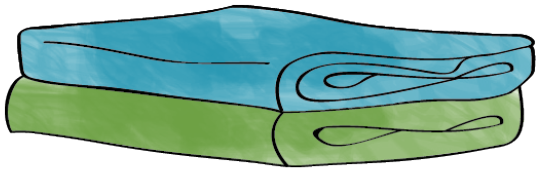
Approximate cost: $2.00

POLYESTER FILLING

The material that's inside pillows and stuffed animals. The soft squishy center of your stuffed monster.

Where to buy? Your local craft store, or an online retailer like Amazon.

Approximate cost: $15.00

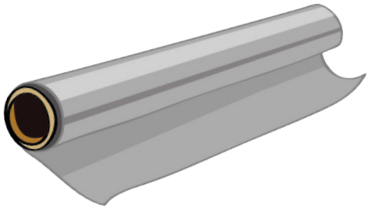


FLEECE OR FELT FABRIC

These thick non-stretch fabrics are easy to sew.

Where to buy? Your local craft store, or an online retailer ike Amazon.

Approximate cost: $10.00 per yard

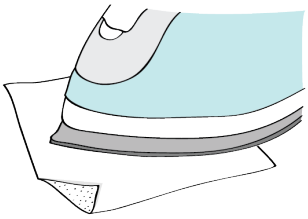


ALUMINUM FOIL

Standard aluminum foil. Do not try to use the non-stick variety.

Where to buy? Your local grocery store.

Approximate cost: $2.00



IRON-ON ADHESIVE

Thermoweb's Ultra Hold Heat-n-Bond ™ adhesive. You'll need the sheet, not the tape.

Where to buy? Your local craft store, or an online retailer like Amazon.

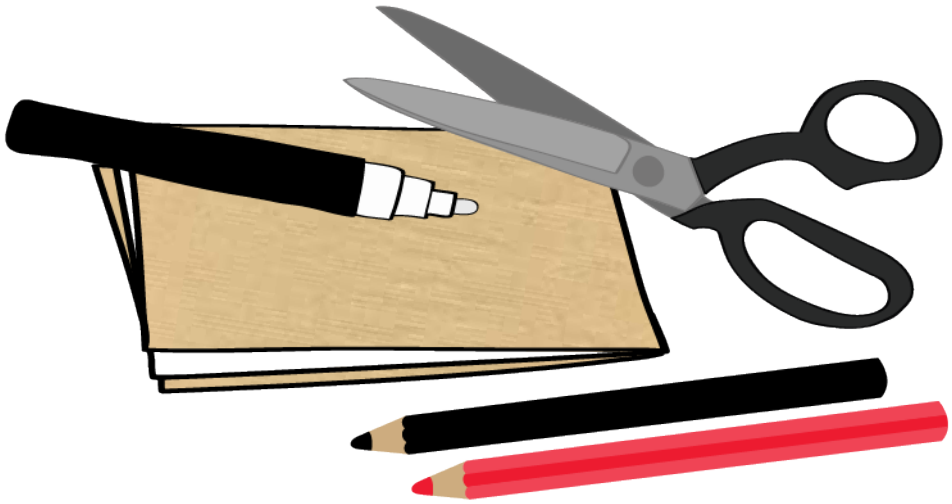
Approximate cost: $15.00

MISCELLANEOUS CRAFT MATERIALS

Scissors, paper, pencils, chalk, etc.

Where to buy? Your local craft store, or an online retailer like Amazon.

Approximate cost: $5-10.00



Glossary

AMP. A unit used to measure electric current (the flow of electrons).
See also: **ENERGY**.

AMP HOUR. A unit of electric charge based on the number of amps used in one hour. Household circuit breakers typically provide a maximum of 15 A (amps) or 20 A of current to a wall outlet.

ARDUINO DEVELOPMENT ENVIRONMENT (ADE). The programming environment for e-Textiles. ADE programs, or "sketches," have three parts: the variable declarations section, the setup section, and the loop section.

ARDUINO PROGRAMMING LANGUAGE. Used in the e-Textiles project, this programming language is a simplified version of C++.
See also: **PROGRAMMING** and **PROGRAMMING LANGUAGE.**

BIT. The basic unit of information in computing and telecommunications. A bit can have the value of either 1 or 0 only, so it acts as an on–off switch in electric current.

BOARD, MICROCONTROLLER BOARD. A small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals. The LilyPad Arduino SimpleSnap is a microcontroller board designed for wearable computing and e-textiles. It can be sewn to fabric and similarly mounted power supplies, sensors and actuators and wired using conductive thread.

BUG. A software bug is an error, flaw, failure, or fault in a computer program or system that produces an incorrect or unexpected result, or causes it to behave in unintended ways.
See also: **DEBUG**.

CALL (A PROCEDURE). "Calling" a procedure means putting the procedure in the main part of a program to use. For example, we can call Arduino's built-in delay procedure by including the statement `delay(1000);` in the loop part of the program.

CIRCUIT. Components such as a battery and an LED connected by a conductive wire or trace in which electric current flows. See also: **SHORT CIRCUIT**.

CODE OR SOURCE CODE. A collection of instructions that a computer (or LilyPad) will carry out and that are written in a programming language. Any piece of an Arduino program can be called code.

COMMENT. This is a piece of text that will be ignored by the computer when the code is compiled. A comment either begins with // characters or begins with a /* and ends with */. It is useful to "comment out" lines of code to temporarily deactivate them from the program.

COMPILE. To translate program code to machine code that the computer chip on the LilyPad board can understand. In the Arduino environment, clicking the check-mark icon in the toolbar compiles C code written by the programmer into hex code that a LilyPad can understand.

COMPILE ERROR. An error in a program that is detected when software attempts to compile the program. Compile errors are often syntax errors such as misspellings or missing semicolons. For example, since the word "delay" is misspelled in the statement `dellay(1000);`, the code containing this statement will generate a compile error.

CONDITIONAL STATEMENT. A block of code that does one thing if a condition is true and another thing if the condition is false. `if else` statements and `while` loops are examples of conditional statements.

CONDUCTIVE. A property within a material that allows electric current to run through it. Metals such as copper, silver, and aluminum are highly conductive. They are good *conductors*. Conductive materials in this book's projects include the tabs on the LilyPad components and the conductive (stainless steel) thread used to sew these components together. The opposite of a conductive material is an insulating material. Insulating materials — such as plastic, glass, and fabric — resist the flow of electricity. Electrical current does not flow through insulating materials.

CURRENT (ELECTRIC CURRENT). The rate of flow of electric charge past a certain point in a circuit. Current travels from the (+) side of a battery (power), through the components connected in the circuit, and back to the (–) side of the battery (ground). Current only flows through a circuit if the path is complete; that is, if there are no breaks in the circuit. Current is measured in amps.

DEBUG. The process of identifying and removing errors from computer hardware or software. See also: **BUG**.

DELAYTIME. This is a variable in the Arduino language, used to create the pause between blinks in the programming tutorial.

DRIVER. A bit of downloadable software that allows a computer to interact with other hardware, such as a printer.

ELECTRICITY. A form of energy that results from the presence of charged particles (such as electrons or protons) existing either statically as an accumulation of charge (as in lightning or a shock) or dynamically as a current.

ENERGY. Electrical energy is the ability of a power supply to run a circuit over time, light up an LED, or make sounds with a speaker. The amount of energy stored in a battery is equal to its amp-hour rating multiplied by its voltage rating. Energy is measured in watt-hours (Wh). For example, a 3-volt battery with an amp-hour rating of .25 amp-hours stores .75 watt-hours of energy.

E-TEXTILES (electronic textiles). Fabrics with embedded electronics including sensors, lights, motors, and small computers. Designers of e-textiles strive for softness and wearability by using new materials such as conductive thread, conductive fabric, and flexible circuit boards.

EXECUTE (a program). The act of carrying out the instructions specified by a program. A computer — such as the LilyPad Arduino — executes programs line by line in the exact order in which they are written.

FREQUENCY. The speed or pitch of a sound wave. Sound is created by vibrations of molecules in the air. When molecules vibrate very quickly — at a high frequency — we hear a high tone; when they vibrate more slowly — at a low frequency — we hear a low tone. Frequency is measured in pulses per second, or Hertz (Hz). Musical notes are associated with their frequency; for example, high C is 1046 Hz.

FTDI BREAKOUT BOARD. A printed circuit board that mechanically supports and electrically connects electronic components using conductive tracks, pads, and other features. In this project, the FTDI board associates the LilyPad Arduino SimpleSnap to the computer through a USB mini-cable.

GROUND (–). The negative terminal of a battery or other power supply in a circuit; 0 volts. Also, any part of a circuit that is at 0 volts. Ground is the reference point in a circuit from which all other voltages are measured. The color black is used to denote ground in circuit diagrams and drawings. In Arduino code, ground is referred to as `LOW`. See also: **LOW** and **POWER**.

HEX CODE (short for hexadecimal code). A numerical code based on 16 symbols in a base-16 number system. When Arduino code is compiled, it is translated into hex code. This code is then loaded onto the LilyPad Arduino, which can read and execute it.

HIGH or `HIGH`. In Arduino code, the term used to refer to power (+) in electrical circuits. Power, (+), and `HIGH` refer to the positive terminal of a battery or other power supply in a circuit. See also: **LOW** and **POWER**.

INPUT (device). An electrical component that gathers information from the world. The input device could be a switch that is flipped or a sensor that reads temperature or sound. In the Arduino environment, information is collected from inputs through digitalRead and analogRead statements.

INPUT (to a procedure) or **INPUT VARIABLE.** Information required by a procedure. In Arduino programming, an input is usually a number supplied in parentheses after the procedure name in a procedure call. For example, in the statement `delay(1000);`, the number 1000 is the input. Input variables help programmers write procedures that apply to a wide range of situations. Inputs make it possible to carry out the same basic set of instructions or statements with different values. For example, since the built-in procedure `delay` has an input we are able to delay for different amounts of time in our programs. When we call `delay` with an input of 1000, the program pauses for one second. When we call `delay` with an input of 100, it pauses for 1/10 of a second, and so on. Similarly, in the monster tutorial, the duration input variable to the song procedure lets us play a song at different speeds when we provide different values as inputs.

INT. Int stands for integer, which is any whole number. Int is a variable type in the Arduino language. All of the variables and numbers used in the e-Textiles lessons are of type `int`. For example, the statement `intled = 13;` declares a variable called `led` of type `int`.

LANGUAGE. A programming language is an artificial language designed to communicate instructions to a machine, particularly a computer. Programming languages express algorithms precisely to create programs that control a computer's behavior.

LED. Short for light-emitting diode. LEDs contain an electroluminescent material — a material that glows when electrical current flows through it. LEDs are polarized. That is, they have a (+) and a (–) side and will only light up when current flows from their (+) to (–) side. If an LED is attached backward in a circuit, it will not work. LEDs are more efficient than most other light sources. That is, they produce more light with less energy than most other types of lights.

LILYPAD ARDUINO SIMPLESNAP BOARD. A type of computer microprocessor that is used in e-Textiles.

LOW or `LOW`. In Arduino code, the term used to refer to ground (–) in electrical circuits. Ground (–) and `LOW` refer to the negative terminal of a battery or other power supply in a circuit. `LOW` is always 0 volts. See also: **GROUND** (–) and **HIGH**.

MICROCONTROLLER. A small computer chip that stores and executes programs and controls electronics. Microcontrollers, like most computers, have a memory that is used to store programs and program data and a processor that is used to interpret and execute programs. Microcontrollers also have pins that can be used to control input and output devices. When input and output devices such as sensors and LEDs are attached to the pins, the microcontroller can read electrical signals from the inputs and send electrical signals out to the outputs. The LilyPad Arduino SimpleSnap board contains an ATmega328 microcontroller. See also: **PIN**.

MILLISECONDS. 1/1000 of a second. There are 1000 milliseconds (1000 ms) in one second. The input to the Arduino procedure `delay` is in milliseconds. Thus, `delay(1000);` pauses program execution for one second, `delay(100);` pauses program execution for 1/10 of a second, and so on.

MULTIMETER. An instrument designed to measure electric current, voltage, and usually resistance, typically over several ranges of value.

NONCONDUCTIVE. A material that is not able to conduct electricity. For example, regular sewing thread is nonconductive.

OPEN SOURCE. A term used to describe a computer program for which source code is freely and publicly available to use, examine, and modify. By sharing open-source software, programmers can collaborate, extend, and expand on each other's work. The Arduino software is open source, and the designs of the LilyPad boards are also open source.

OUTPUT. An electrical component that takes action — does something — in the world. Actions include lighting up, moving, and making sound. Output devices are things like lights, motors, and speakers. In the Arduino language, outputs are controlled by digitalWrite statements.

PARALLEL CIRCUIT. A type of circuit where current can pass through multiple paths. This is different from a circuit in series, where there is only one path through all the components. Components in a parallel circuit have all of their (+) sides connected together and all of their (–) sides connected together. This configuration allows for all the components to receive the same voltage.

PIN. Part of a microcontroller that can attach to and control an input device (such as a switch) or output device (such as an LED). The pins on a microcontroller look like tiny legs coming out of the controller's black, square body. On the LilyPad Arduino SimpleSnap boards, microcontroller pins are connected to sewable tabs and snaps. When an input or output device is connected to a tab or snap, the microcontroller can control that component. These lessons use the terms pin and tab interchangeably.

POWER (+). The positive terminal of a battery or other power supply in a circuit. A circuit's power is generally its highest possible voltage. Circuit diagrams and drawings use the color red to denote power. Note: Different circuits may have different power (+) voltages. For example, in a circuit that uses a 3-volt battery, power is +3 volts. In a circuit that uses a 3.7 volt battery, power is +3.7 volts. In Arduino code, power is referred to as `HIGH`. See also: **HIGH** and **GROUND (–)**.

PROCEDURE. A block of code that is given a unique name. A procedure may have one or more inputs and may return a value. When a procedure is called, the program jumps to the place in the program where the procedure is defined, executes a block of code that makes up the body of the procedure, and then jumps back to the point right after the procedure in the program. The Arduino language uses procedures such as delay, digitalWrite and analogRead.

PROGRAM. A sequence of instructions written to make a computer perform a specified task. Also known as a piece of code.

PROGRAMMING. The action or process of writing computer programs. Programming is also called "coding," since we write in an artificial language, or "code," that the computer understands.

PROGRAMMING LANGUAGE (or code)**.** An artificial language used to write instructions that can be translated into machine language and then executed by a computer. Python, Ruby, C++, Java, and Basic are just a few programming languages. Arduino is based on a simplified version of C++.

READ (from a pin). The act of gathering information from an input device. Information is collected from an input device such as a switch or sensor that is attached to a pin on the LilyPad or other microcontroller. In Arduino, information is read from a pin with the statements `digitalRead` and `analogRead`. `digitalRead(pin);` tells whether the pin is `HIGH` or `LOW`. `analogRead(pin);` measures the voltage level of the pin and gives a number between 0 and 1023 that corresponds to the voltage. See also: **WRITE** and **INPUT (DEVICE)**.

RUNNING STITCH. The most basic stitch in hand sewing, also called a straight stitch. This stitch is created by passing a needle and thread up and down through a piece of fabric along a line. A good running stitch consists of neat, even stitches of about 1/4" (6 mm) in length.

SENSOR. An electrical component that gathers information from the world — for example, pressure, temperature, or sound. Sensors include touch sensors, thermometers, cameras, and microphones. All sensors are input devices. In Arduino, the `analogRead` statement collects information from sensors.

SERIAL PORT. The communication channel through which the computer communicates with a LilyPad and vice versa. The serial port connection — the USB attachment between the LilyPad and the computer — allows Arduino to upload programs to the LilyPad and allows the LilyPad to send information back to the computer through `Serial.println` and `Serial.print` statements.

SHORT CIRCUIT. When current travels along an unintended path, such as when the (+) side of the power supply is directly attached to the (–) side of the power supply. This can lead to excessive current in the circuit, which drains battery life and can result in overheating and damage to the circuit.

SIMPLE STATEMENTS. Lines of code written in a programming language that tell a computer what to do. Simple statements in Arduino end with a semicolon, just as English sentences end with a period. The line `digitalWrite(led, HIGH);` is a simple statement. So are the lines `delay(1000);` and `song(2000);`.

SWITCH. A circuit component that is always in one of two states: open (disconnected) or closed (connected). In a simple circuit, such as the one in the bookmark tutorial, the flow of electricity through a circuit is stopped when the switch is open and restored when the switch is closed.

TABS. The silver-rimmed holes in the sewable LilyPad boards. Tabs are connected to microcontroller pins. See also: **PIN**.

TRACE. A conductive-thread connection between components in a circuit, such as between the Arduino speaker and the LilyPad SimpleSnap microprocessor.

UPLOAD. The act of sending code that has been compiled and converted into hex code from a computer to a LilyPad Arduino.

USB PORT. A standardized connection between a computer and an external component. Computers usually have several USB ports in which to plug USB cables.

USB CABLE/USB MINI CABLE. A connector between the computer and an external component. A USB cable is used to connect the Arduino LilyPad SimpleSnap to the computer for programming.

VARIABLE. The name for a place in the computer's memory where data is stored. Variables identify the components that the program will be controlling. By creating a variable, the programmer sets aside a chunk of memory and assigns it a name. This allows flexibility and speed when writing code.

VOLT/VOLTAGE. The unit of charge for electricity or electrical potential across a conductor.

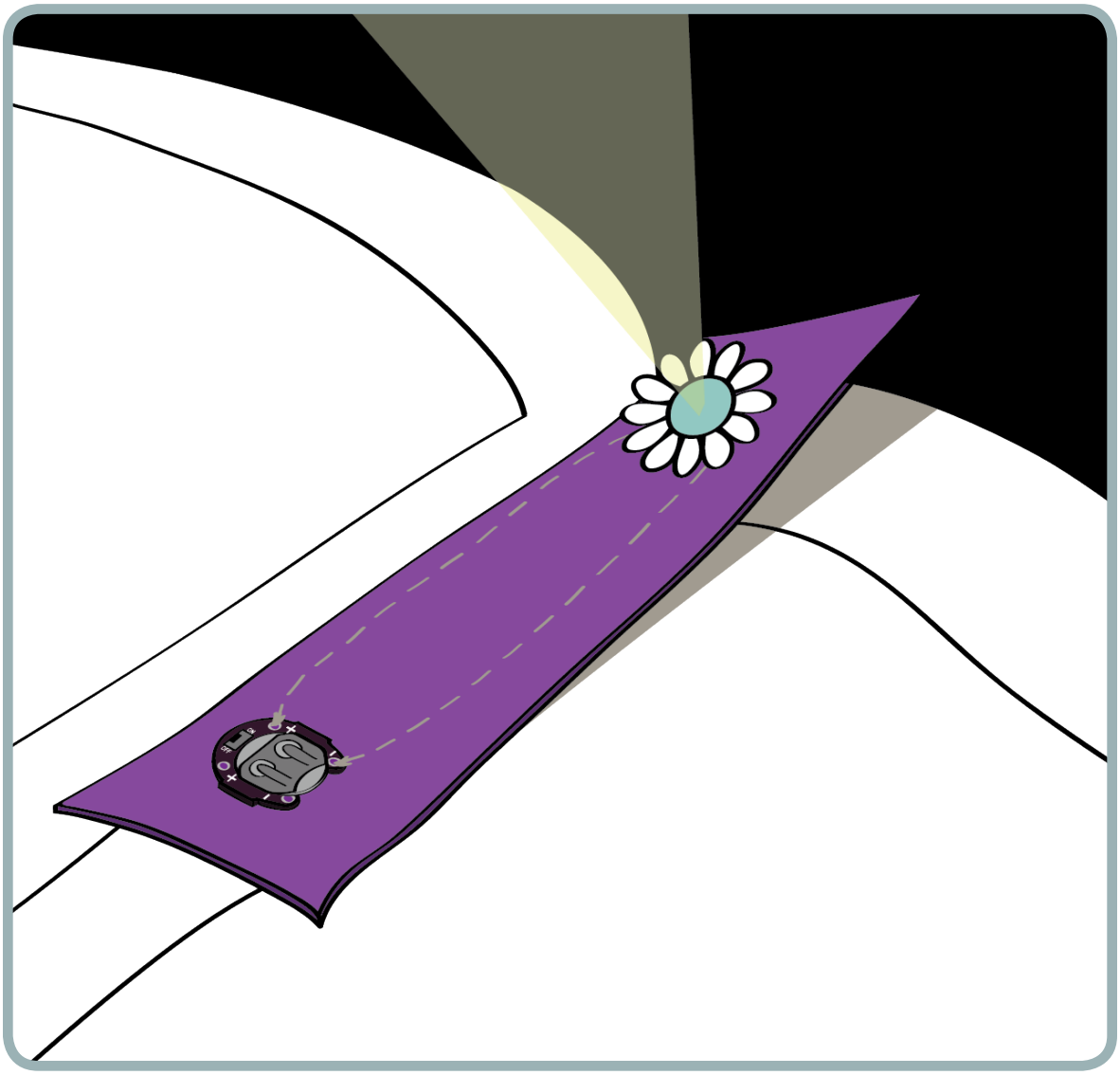
VOLTAGE RATING. The maximum sustained voltage that can safely be applied to an electric device without risking the possibility of electric breakdown. The voltage ratings for copper wire is usually close to 1,000 volts. The wiring for table lamps and other small household appliances is much lower.

WATT-HOURS. A measure of electrical energy equivalent to the power consumption of one watt for one hour. A heater rated at 1000 watts (1 kilowatt), operating for one hour, uses one kilowatt-hour of energy.

WRITE (a pin)**.** The act of controlling an output device attached to a pin on a LilyPad (or other microcontroller) by sending it electrical signals. In Arduino, electrical signals are written to a pin with the statement `digitalWrite`. `digitalWrite(pin, value);` sets the pin to be either `HIGH` (maximum circuit voltage, power [+]) or `LOW` (minimum circuit voltage, 0 volts, ground [–]). See also: **READ** (from a pin) and **OUTPUT** (device).

Bookmark Booklight

Bookmark Book Light: Sewing Simple Circuits



Welcome to the world of electronics and textiles! In this tutorial, you'll explore how to make soft electronic circuits using conductive thread, LEDs, batteries, and fabric. You'll stitch together a fuzzy bookmark that you can use to read in the dark.

TIME REQUIRED: 1–2 hours

Collect Your Tools and Materials

This project uses a basic set of electronic, sewing, and sketching supplies. You can find detailed descriptions of all of these tools and materials in the reference section of the Practical Guide.

SUGGESTED NONESSENTIAL MATERIALS:

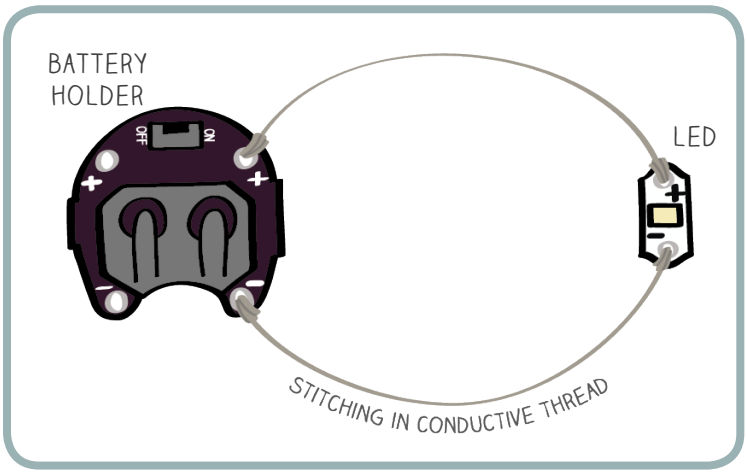
- Fabric or tapestry scissors
- Fabric glue
- Felt or yarn needles
- Toothpicks

Design Your Bookmark

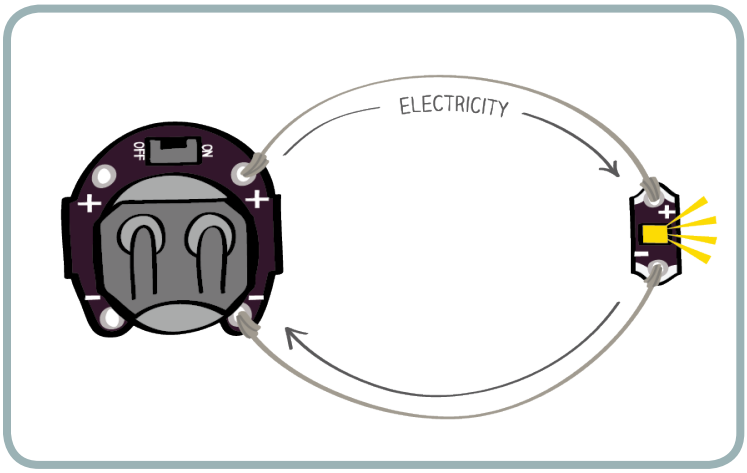
CHECK OUT YOUR ELECTRONIC PIECES

Get out your LED (short for "Light Emitting Diode") and battery holder and look at them closely. The battery holder has four silver-rimmed holes called **tabs** that can be sewn through. Next to each tab is a (+) or (–). The two (+) tabs are connected to the (+) side of the battery that slides into the holder, and the two (–) tabs are connected to the (–) side of this battery. The LED housing also has two tabs labeled (+) and (–). These tabs are connected to the (+) and (–) sides of the LED.

You're going to use these pieces to stitch together a circuit that looks like this.

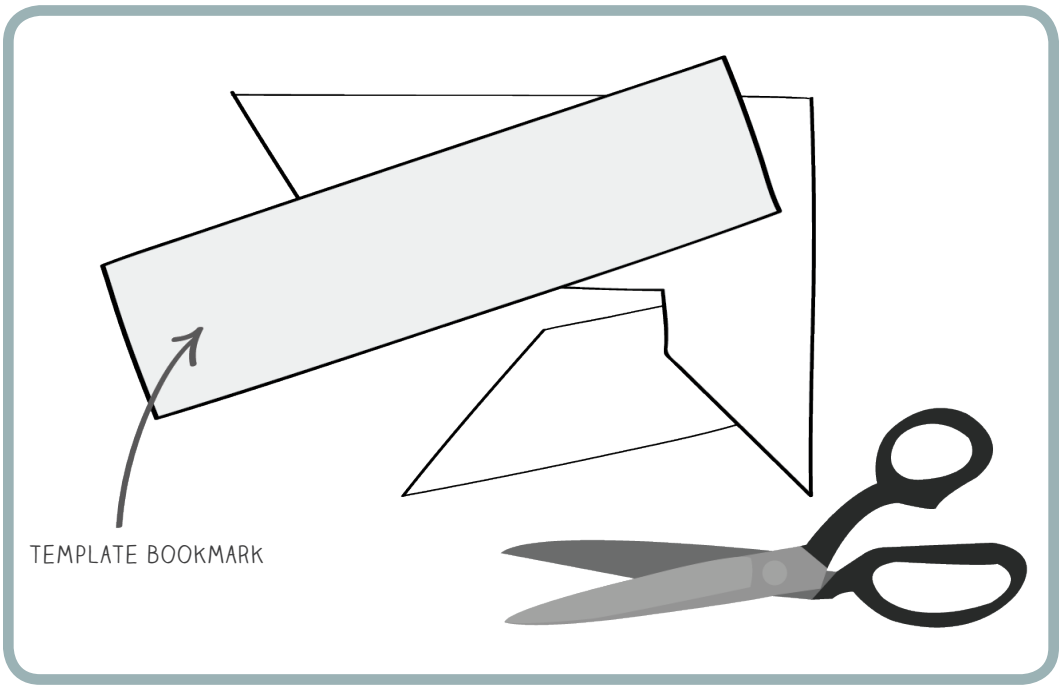


The LED will glow when electricity flows from the (+) side of the battery through the conductive thread, through the LED, and back to the (–) side of the battery.

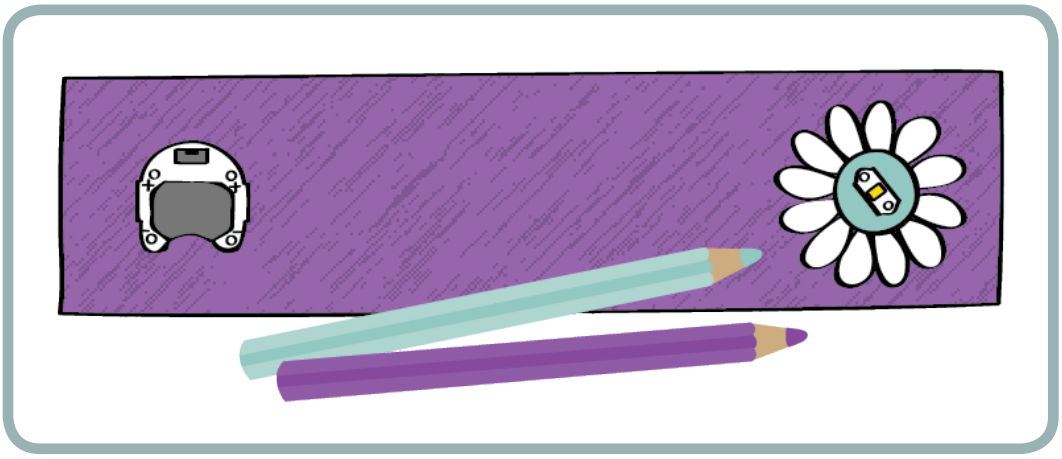


SKETCH YOUR BOOKMARK

Decide on the size and shape for your bookmark. On a piece of stiff paper, draw your bookmark's outline. It should be at least 2 inches wide so that your battery holder fits on it easily. Cut out this outline. You'll use it as a template for cutting out your fabric later.



On a second piece of paper, use your template to draw a new bookmark outline. Put your template aside. Decide on a color scheme and a decorative theme for your bookmark, and add these details to your sketch. Figure out where you want your battery holder and LED to be. Draw these on the sketch too.

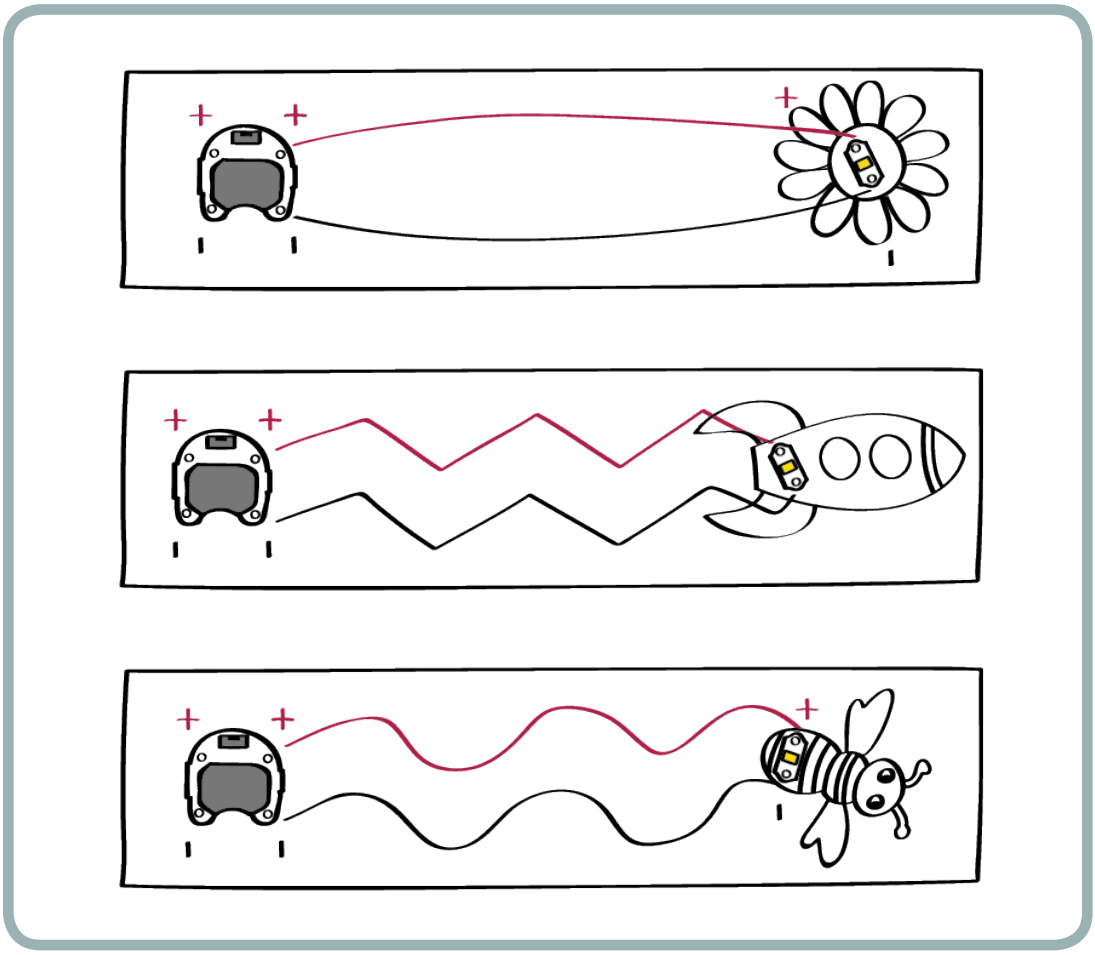


Next, plan out the conductive thread connections between the components. In electronics terminology, each of these connections is called a **trace**. In your circuit, there are two traces: the (+) trace that connects the (+) side of the battery holder to the (+) side of the LED and the (–) trace that connects the (–) sides of the battery holder and LED. The (+) traces are drawn in red, and the (–) traces are drawn in black. These are the traditional colors used to indicate (+) and (–) in electronics.

Here are a few things to keep in mind as you plan your traces and add them to your sketch:

- It's important to connect a (+) tab on the battery holder to the (+) side of the LED and a (–) tab on the battery to the (–) side of the LED. If you connect your LED backwards, it won't light up.
- If the (+) and (–) traces touch each other, it creates what is called a short circuit. Short circuits will drain your battery and prevent your LED from turning on. You want to keep the (+) and (–) traces as far away from each other as possible.
- Traces can form part of your decoration — they can be curvy or square, zigzags, loops, or straight lines — or they can be covered up by decoration.

Here are a few example designs. (The rest of this tutorial will use the first one.) Look at the rocket example. Where should (+) and (–) be marked on the rocket? Check with your neighbor. Were you right?



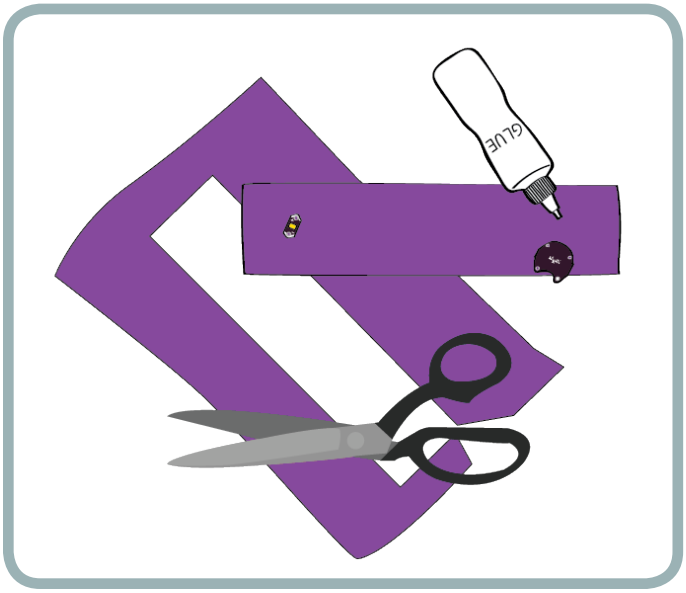
Use your colored pencils to draw your electrical connections on your design sketch.

Build Your Bookmark

BEGIN BUILDING

Use the template you made and a piece of chalk or a pencil or pen to trace the shape of your bookmark on your fabric. Cut out this shape.

Glue the battery holder and LED to the top side of the fabric, using your design drawing as a reference. This will keep them in place as you sew. Be careful not to fill any of the tab openings with glue. (You may want to use a toothpick to apply the glue to the components.) You will sew through these with conductive thread in a few minutes.

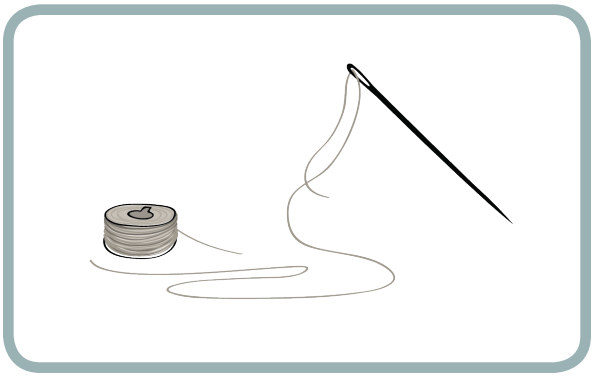


While you are waiting for the glue to dry, use a piece of chalk, a pencil, or a pen to lightly mark on your fabric the traces that go from the battery holder to the LED. This will make it easier for you to sew your connections when you start stitching.

THREAD YOUR NEEDLE

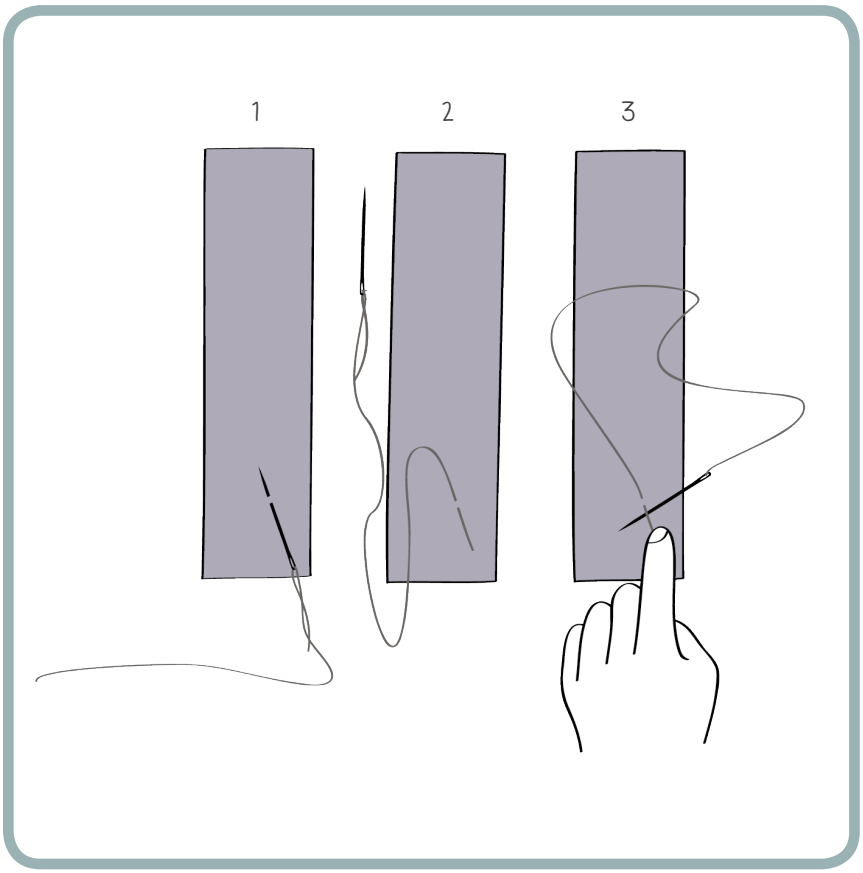
Get out your needle and conductive thread. Cut off 2–3 feet of conductive thread and thread it through the needle. (If it is difficult to thread the conductive thread through the needle, pull the thread taut and cut it at a 45-degree angle. Then gently wet the tip of the conductive thread and press together with your fingers. The thread should more easily fit through the eye.)

Pull the conductive thread partway through the needle so that you have one long tail and one short tail coming out of the needle.

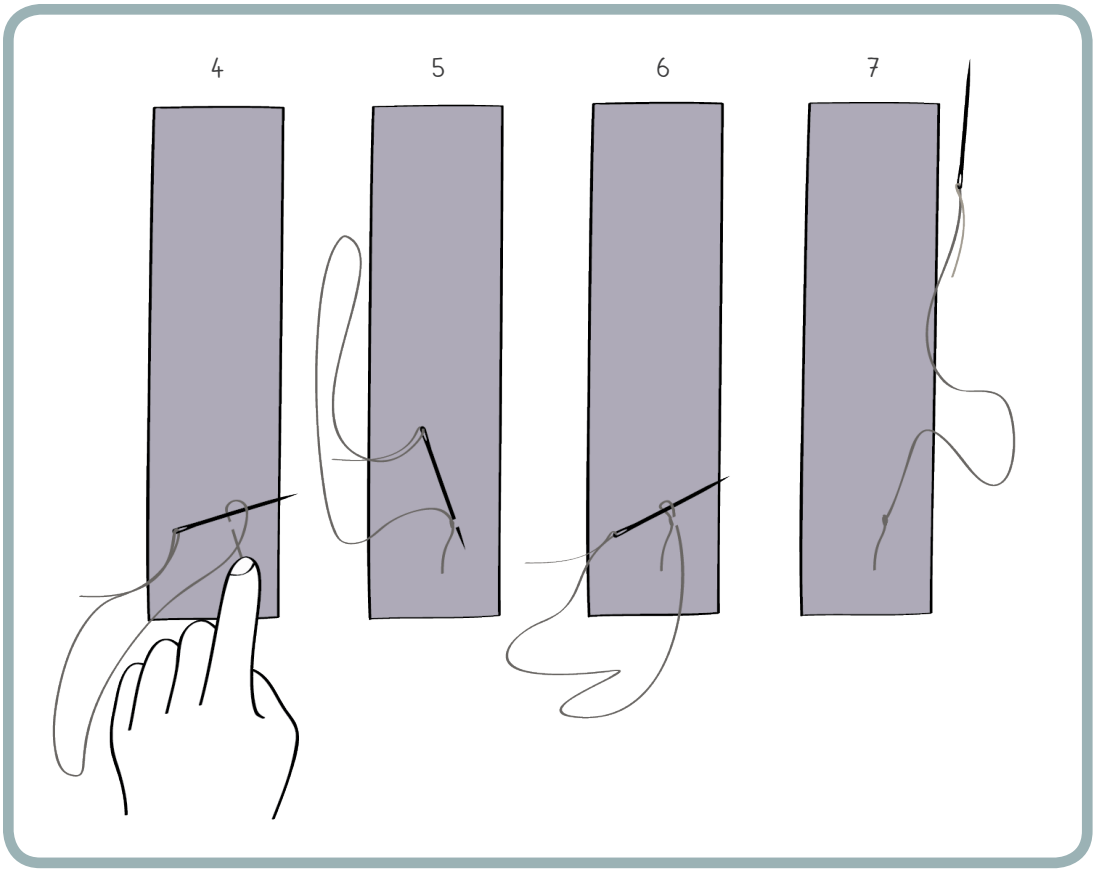


TIE YOUR FIRST KNOT

To ensure that your stitching doesn't come undone, you need to tie your thread securely to your fabric before you start sewing. Follow along with the figures as you go through the steps. (The following is a recommendation for making a knot on your fabric. If you prefer, use your own style of knot to secure the thread.)



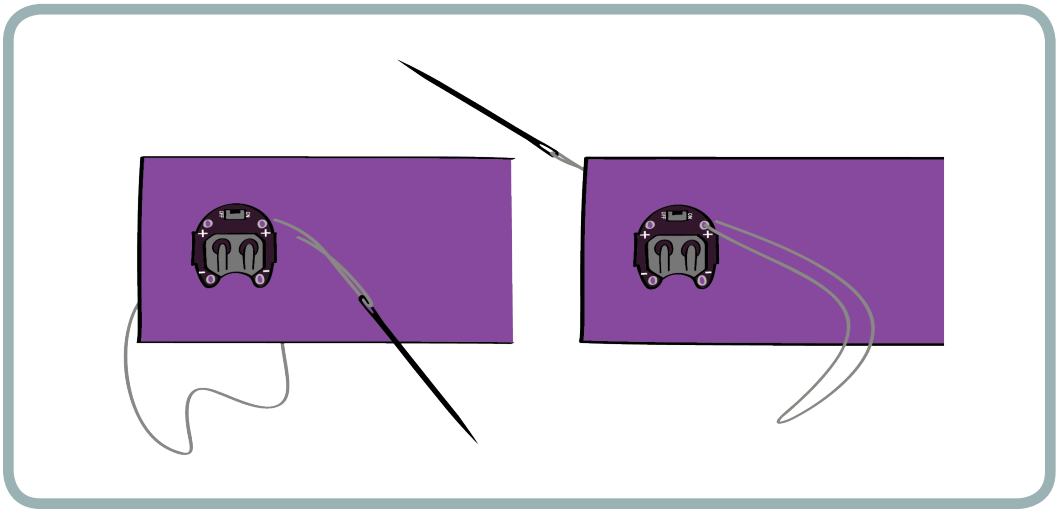
1. On the underside of your fabric, right underneath one of the (+) tabs on your battery holder, sew through a small piece of fabric with your needle. Note: Don't sew through the (+) tab itself, just through the fabric.
2. Pull your needle through the fabric so that a short (approximately 1 inch) tail is left behind.
3. Hold this tail with your left hand, and thread the needle underneath the tail to make a small loop.



4. Guide your needle back through the loop you made in step 3. Pull the needle tightly to produce a snug knot on the surface of your fabric.
5. Guide your needle back through the fabric, creating a new loop.
6. Guide your needle through this loop.
7. Pull it tightly to produce the final knot.

STITCH THE (+) TAB OF THE BATTERY HOLDER

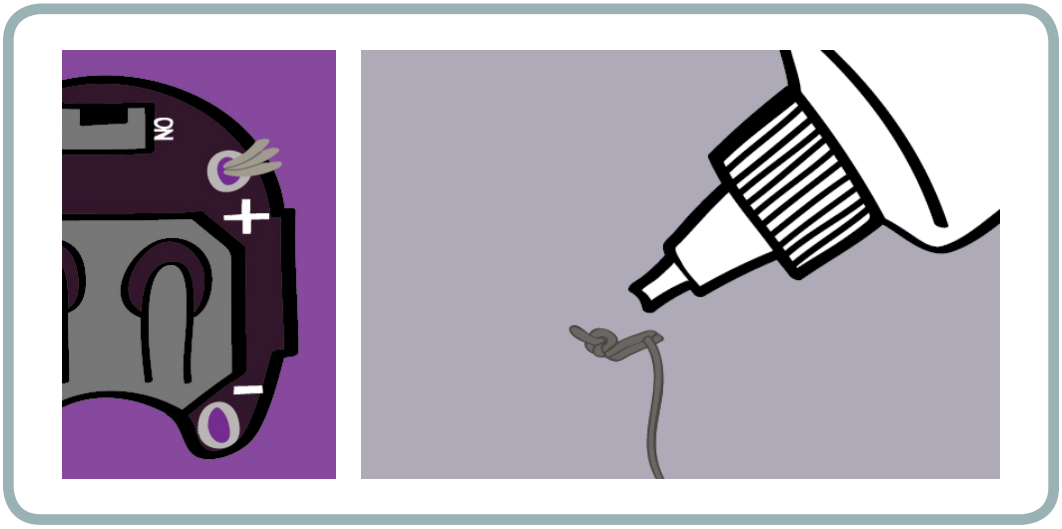
Push your needle up through the fabric right next to the (+) tab of the battery holder. Tug on your needle to make sure you don't leave any excess thread on the underside of your fabric.



Now, poke the needle through the (+) hole to create a loop around the (+) tab, and secure the battery holder to the fabric. Pull the thread through tightly to make a snug connection between the thread and the tab.

Repeat this process at least three times. Make sure you maintain tight contact between the thread and the tab by tugging on the thread after each stitch.

Before you move on to the next step, put a small dab of glue on the knot on the underside of your fabric to make sure it doesn't unravel, and trim the excess thread on your knot to about ¼ inch. Let your bookmark sit for 5 minutes to dry.

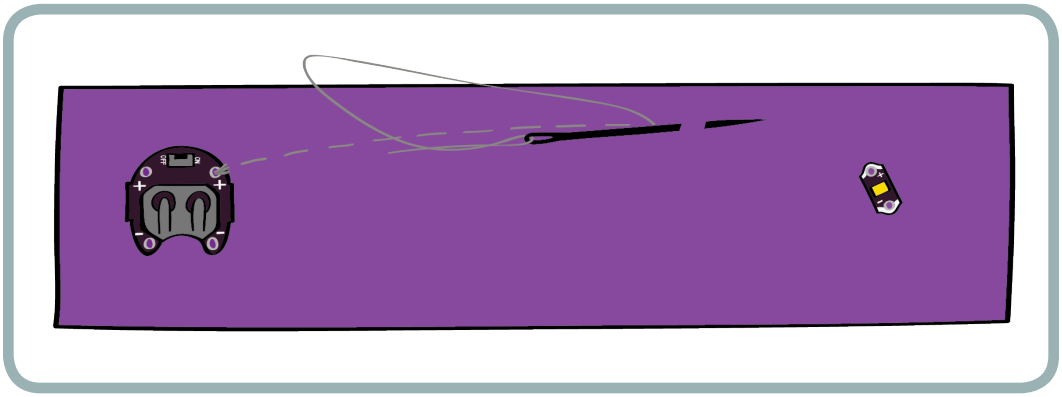


STITCH THE (+) TRACE

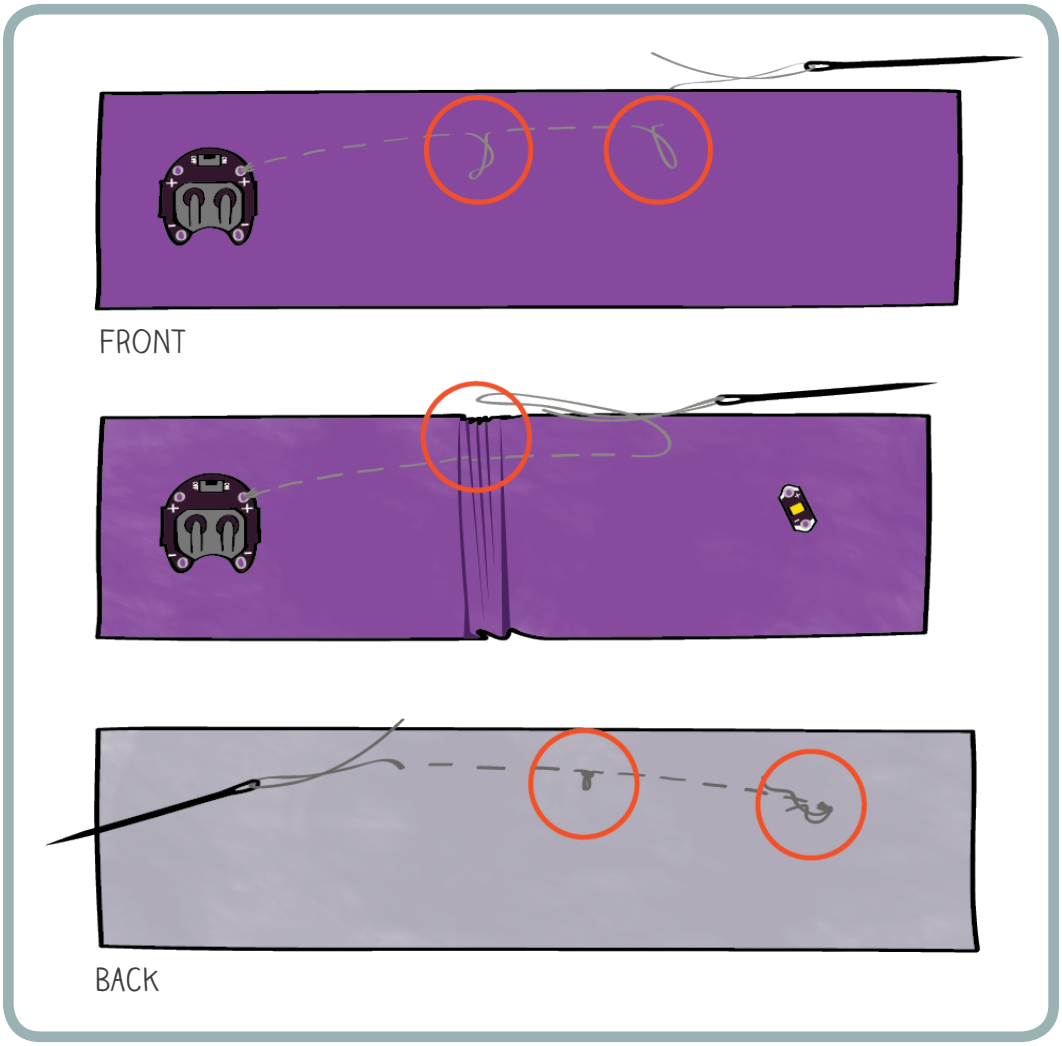
Now you're ready to begin sewing the trace from the (+) tab on the battery holder to the (+) tab on the LED. To do this, you'll use a simple technique called a running stitch that weaves the thread between the front and back sides of the fabric.

Begin with your needle on the underside of the fabric, near the battery holder tab you just sewed. If your needle is on the front side, poke it through to the back side.

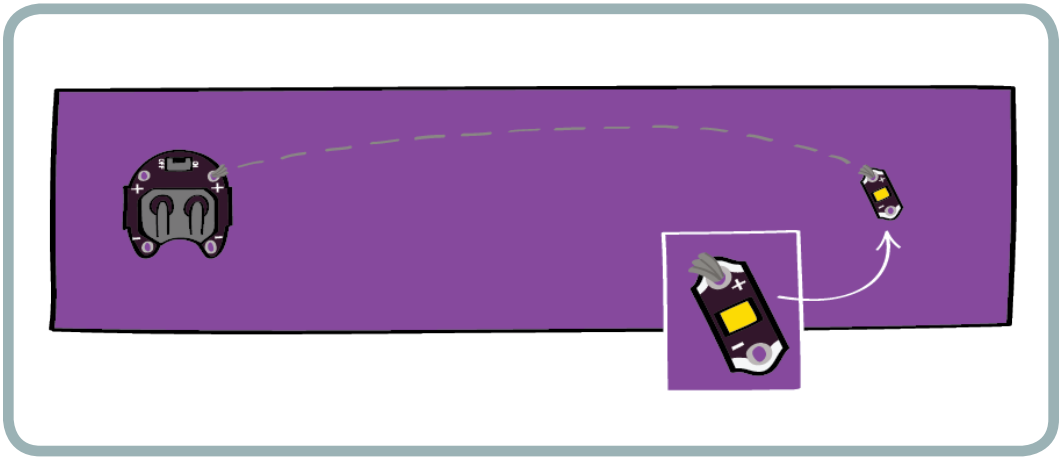
Guide your needle up through the fabric about ¼ inch away from the battery holder to make your first stitch. Then, sew down through the front side of your fabric another ¼ inch toward your LED. Repeat this process until you reach the (+) side of your LED. You should make neat, even stitches that follow the line you drew from your battery holder to your LED.



Pause every few stitches to check for loose or tangled threads. Check both the front and back of the fabric to make sure there aren't any tangles or knots hiding on the side you can't see. Also make sure that you're not gathering and puckering your fabric as you stitch. If your fabric is puckered, take time to flatten it back out before you continue stitching.



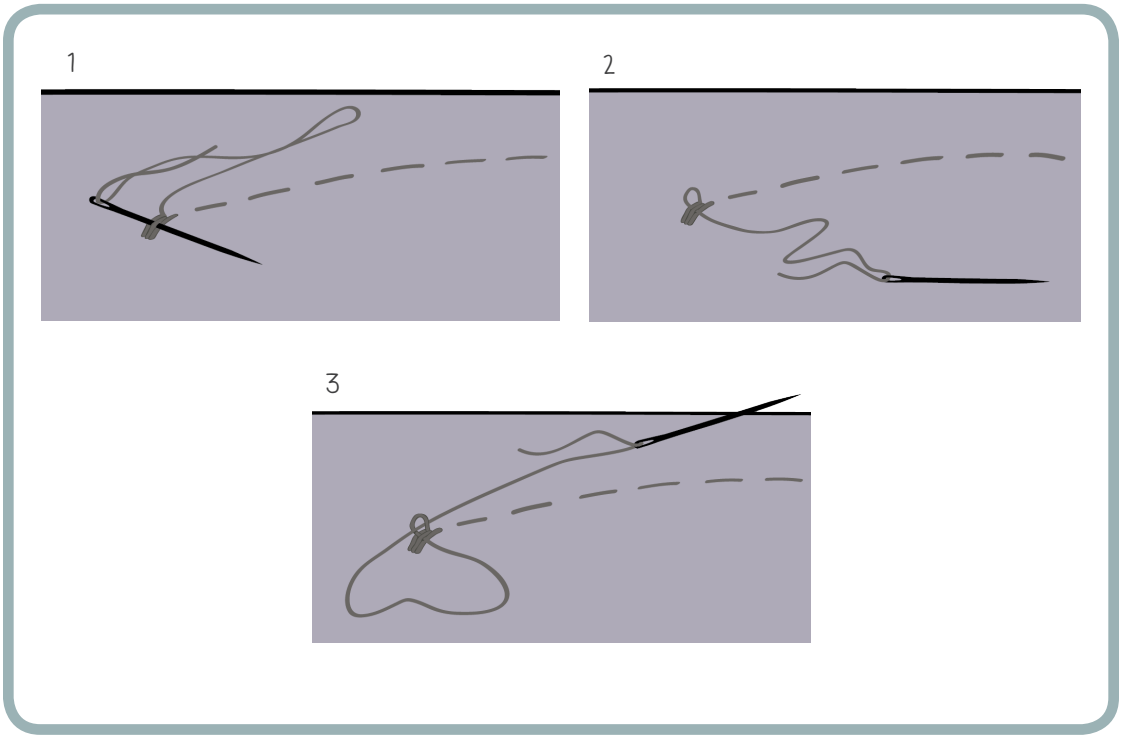
Once you reach the LED, make at least three tight loops around its (+) tab, the same way you attached your battery holder.



TIE A FINISHING KNOT

Now you need to tie a knot to complete this trace. Begin by making sure your needle is on the back side of the fabric. (You want to hide all your knots on the back of your bookmark.)

Now you'll repeat the steps you followed to tie your beginning knot:



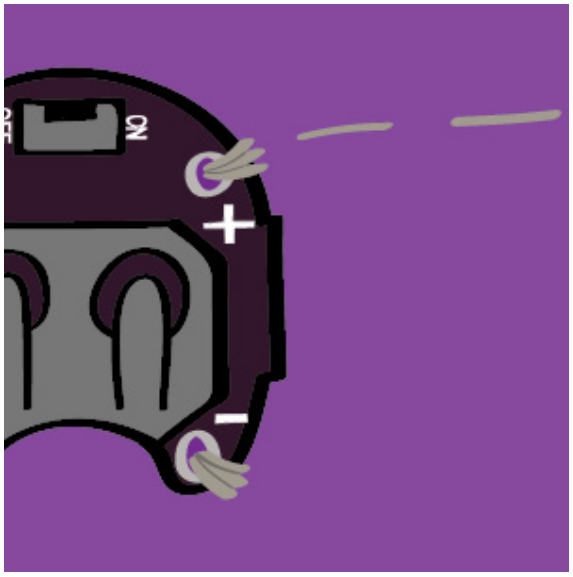
1. Guide your needle through one of your earlier stitches . . .
2. . . . to create a loop of thread.
3. Push your needle through the loop you just made, and pull on it slowly and firmly until the loop has tightened.
4. Repeat this process at least one more time to make a secure knot.

Cut your thread, leaving behind tail that's about ¼-inch long. Put a dab of glue on the tail to make sure it doesn't unravel. Let your bookmark sit for 5 minutes to dry.

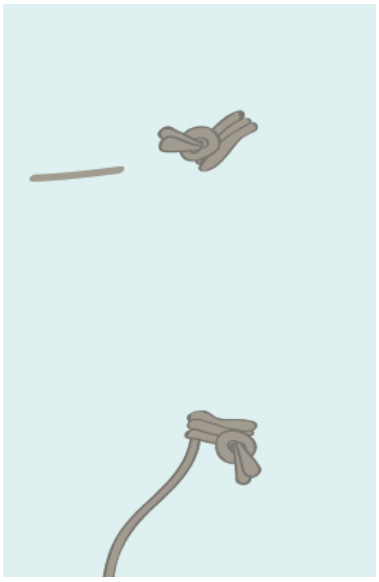
STITCH THE (–) TRACE

Sewing on the (–) trace is just like sewing on the (+) trace except you'll stitch from a (–) tab on the battery holder to the (–) tab on the LED.

Begin by tying a knot in your thread and making at least three tight loops around the (–) tab on the battery holder. (Refer to "Tie Your First Knot," above.) Trim the tail on your starting knot, and put a dab of glue on it to make sure it doesn't come undone. You may want to let your bookmark sit for a few minutes to allow the knot to dry.

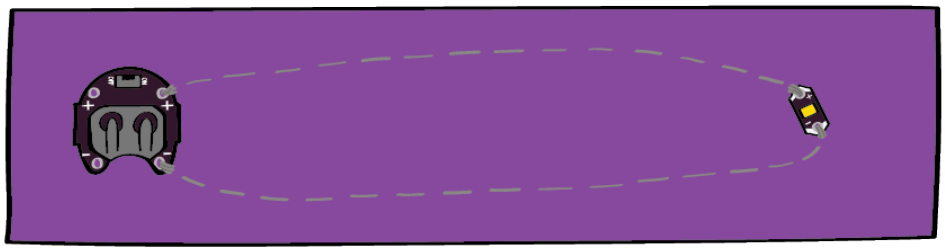


FRONT

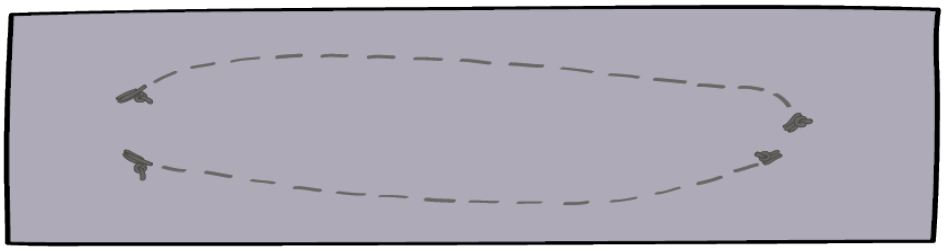


BACK

Then, following the line you drew on your fabric, stitch from this tab to the (–) tab on the LED, checking periodically for tangles, loose threads, and puckering. When you reach the (–) tab on the LED, loop through it snugly at least three times before tying a knot, cutting your thread, and securing your knot with a dollop of glue.

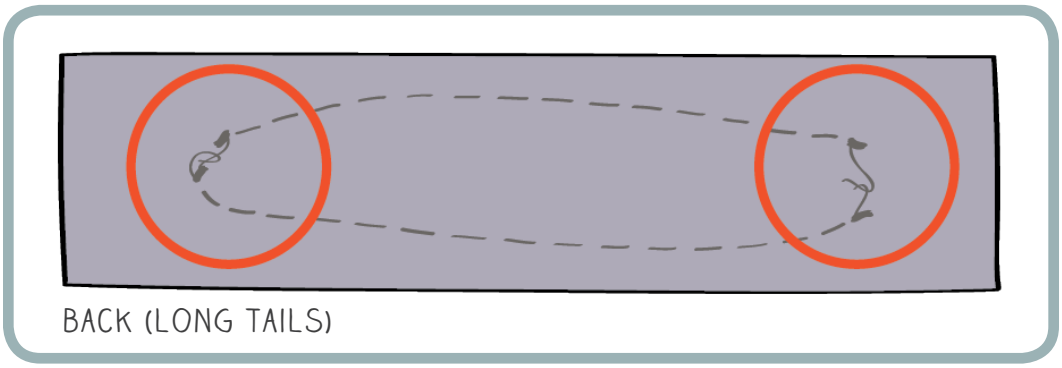


FRONT



BACK

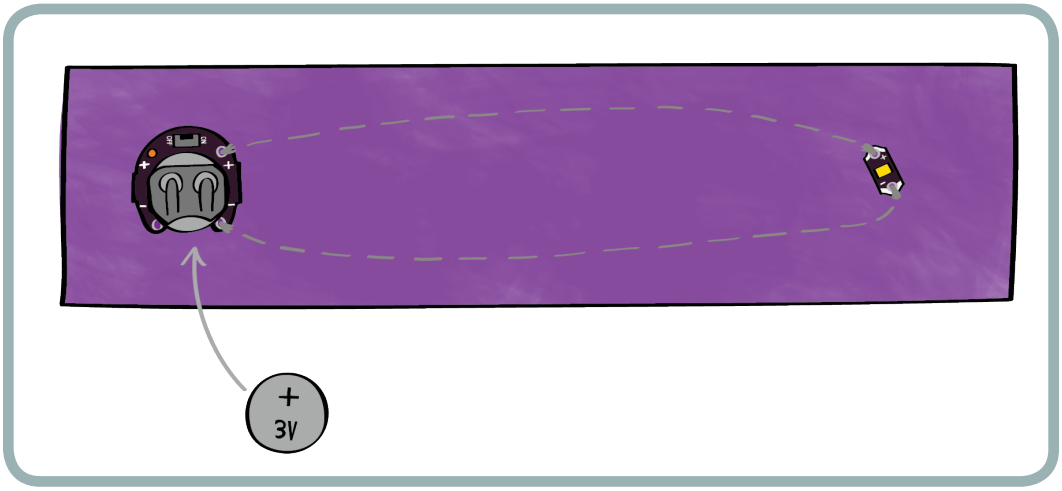
As you're sewing, don't let the (–) trace touch or come close to the (+) trace. This is particularly important in places where there are knot ends or loose threads. When you're finished, double check the front and back of your fabric for loose threads and long tails.



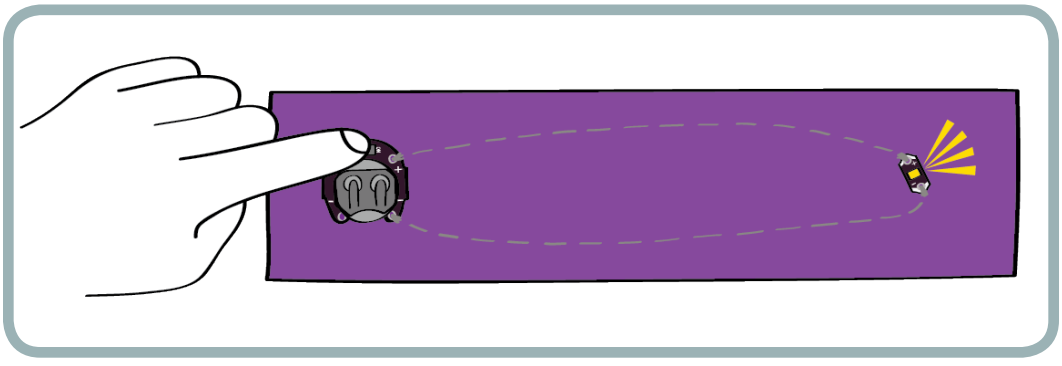
TEST THE CIRCUIT

The circuit is now complete and ready to use! You just have to test it out.

Slide your battery into its holder. The coin cell battery has a flat side with a (+) sign on it that should face up as you put it into the holder.



Flip the switch on your battery holder from "off" to "on" to test out your LED. Your LED should now shine brightly. If it doesn't, see the troubleshooting section.



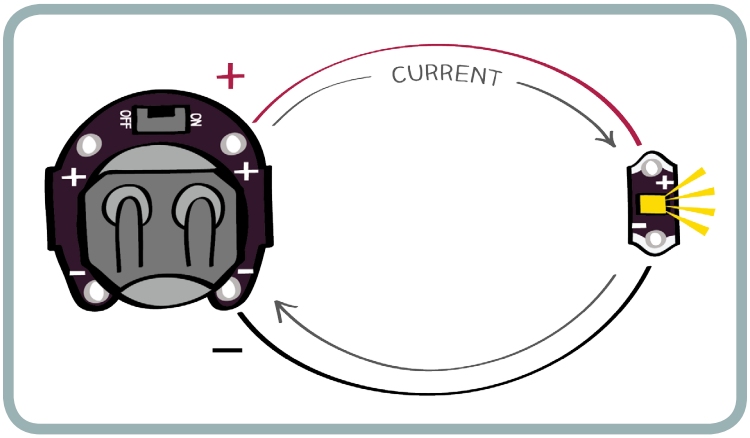
Understanding Your Circuit

Now that you've built a circuit, it's time to explore how it works.

Generally, a circuit consists of a power source, such as a battery that is connected through a conductive material such as copper wire, to other electronics, such as lights, motors, switches, or sensors. In the circuit you built, you used a coin cell battery, an LED light, and a soft conductive thread to make the connections.

CURRENT, VOLTAGE, AND ENERGY

The LED in your circuit turns on when electricity flows through it. This flow is called an electric current. Electric **current** is measured in **amps**. In your circuit, current flows from the (+) side of the battery through conductive thread, then through the LED, then through more conductive thread, and finally back to the (–) side of the battery:



All batteries have two important ratings. They have a **voltage** rating and an **amp-hour** rating. The voltage rating tells you how many volts your battery can supply, and the amp-hour rating tells you how much current it can supply. Together, these two ratings tell you how much **energy** is stored in the battery.

The coin cell battery that you're using for this project is a 3-volt battery with an amp-hour rating of .25 amp-hours. This means that the battery can supply .25 amps of current at 3 volts for 1 hour before it dies. The circuit you built uses about .025 amps of current when it's turned on (about 1/10 of .25 amps). This means that your battery should last about 10 hours. If you used a bigger battery with a higher amp-hour rating — such as a 3-volt camera battery — it would last longer. Here's a table showing how the two compare:

Battery Type	amps Required (amps, A)	amp-hour Rating (amp-hours, Ah)	Battery Will Last (hours, h)
3V coin cell	.025 A	.25 Ah	.25/.025 = 10 h
3V camera battery (Energizer EL1CRBP)	.025 A	.75 Ah	.75/.025 = 30 h

If you multiply the amp-hour rating by the voltage rating, you get a measure of the total amount of **energy** stored in a battery. Energy is measured in watt-hours. The coin cell battery stores .75 **watt-hours** of energy (3 volts * .25 amp hours). In comparison, a typical car battery has a rating of 50–100 amp hours and 12 volts. A car battery can store 1200 watt-hours of energy — 1600 times more than your coin cell! Here are energy ratings for a few different batteries:

Battery Type	Voltage (volts, V)	amp-hour Rating (amp-hours, Ah)	Energy (watt-hours, Wh)
3V coin cell	3.0 V	.25 Ah	3 x .25 = 0.75 Wh
3V camera battery	3.0 V	.75 Ah	3 x .75 = 2.25 Wh
AA battery	1.5 V	2.00 Ah	1.5 x 2 = 3.00 Wh
car battery	12.0 V	100.00 Ah	12 x 100 = 1200.00 Wh

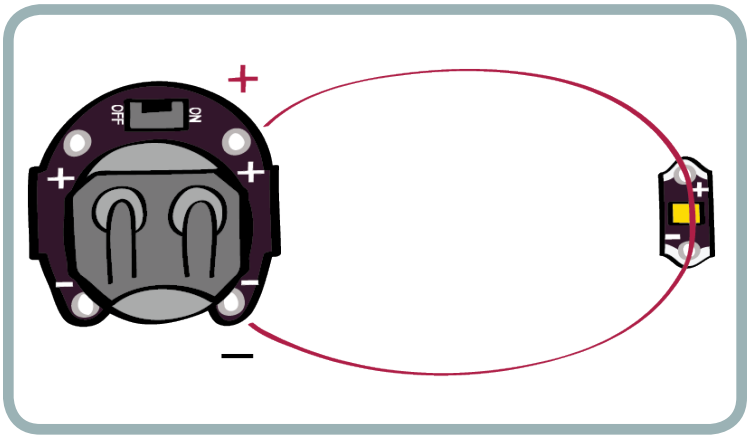
The **voltage** rating of a battery tells you the difference in voltage between its (+) and (–) sides. The (–) side of any battery is always at 0 volts. When a battery is fully charged, the (+) side should be at its rated voltage. So, the (+) side of the coin cell battery should be at 3 volts. The (+) side of a car battery should be at 12 volts, and the (+) side of a AA battery should be at 1.5 volts.

In electronics convention, the (+) side of the battery is called also called "power" or "HIGH." The (–) side is called "ground" or "LOW."

Electrical Values for a 3-Volt Battery		
Symbol	**Voltage**	**Other Name**
+	3 volts	power, HIGH
-	0 volts	ground, LOW

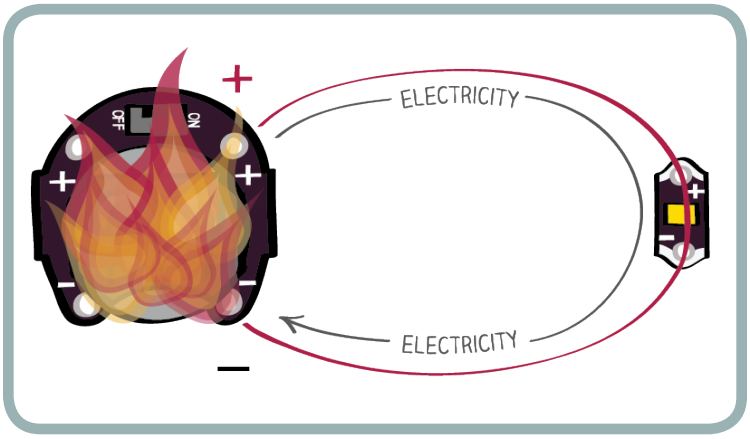
SHORT CIRCUITS

What would happen if you connected the (+) and (–) sides of your battery directly? That is, what if you sewed conductive thread right across the LED's (+) and (–) tabs, like this?



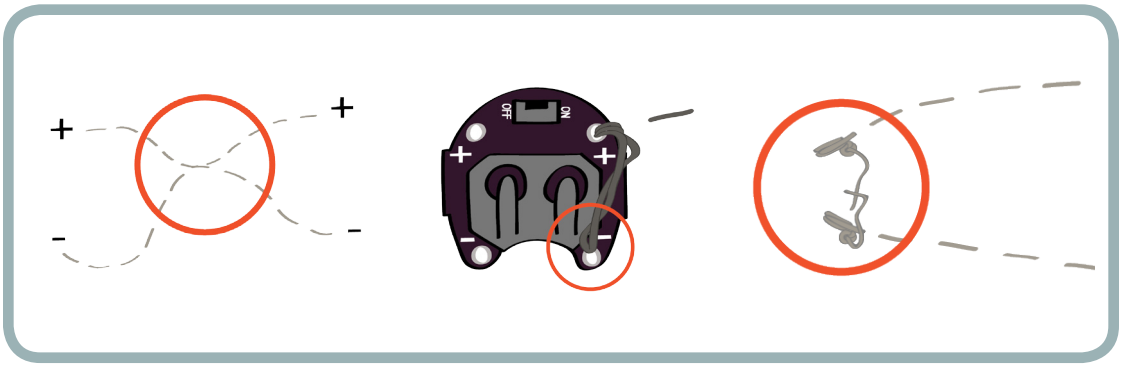
Any time the (+) and (–) sides of any power supply are directly connected, you end up with a **short circuit**, and a tremendous burst of energy is released. This can be dangerous! Imagine connecting the (+) and (–) sides of a car battery together — or even worse, connecting the two slots in an electrical outlet on your wall. Any conductive material — a jumper cable, a bobby pin, a gum wrapper, or a piece of conductive thread — will short out the power supply in a burst of energy. This burst can be very dangerous and destructive. It is likely to ruin your battery or even the electrical outlet and can electrocute or burn you if the power supply is powerful enough (like a car battery or a wall outlet).

For the coin cell battery in your bookmark, if you directly connected (+) to (–), most of the .75 watt-hours of energy stored in the battery would be released all at once.



The battery you're using won't shock or burn you if you create a short circuit, but if you do create one by directly connecting your battery's (+) and (–) sides, your project will not work and you'll quickly ruin your battery.

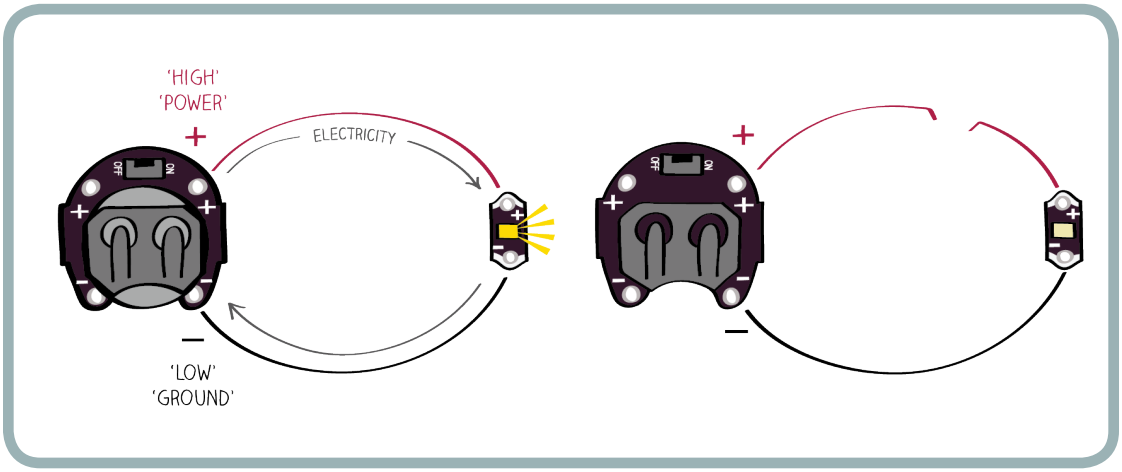
Short circuits are easy to create in fabric circuits, so take care. A loose thread, unraveling knot, messy stitch, or folded piece of fabric can cause your battery's (+) and (–) sides to touch. Here are three examples of problems that may cause short circuits (from left to right): stitches that are too close to one another, a loose loop of thread, and knot ends that are too long.



It's important to think about and check for short circuits as you design, sew, and troubleshoot your projects.

SWITCHES

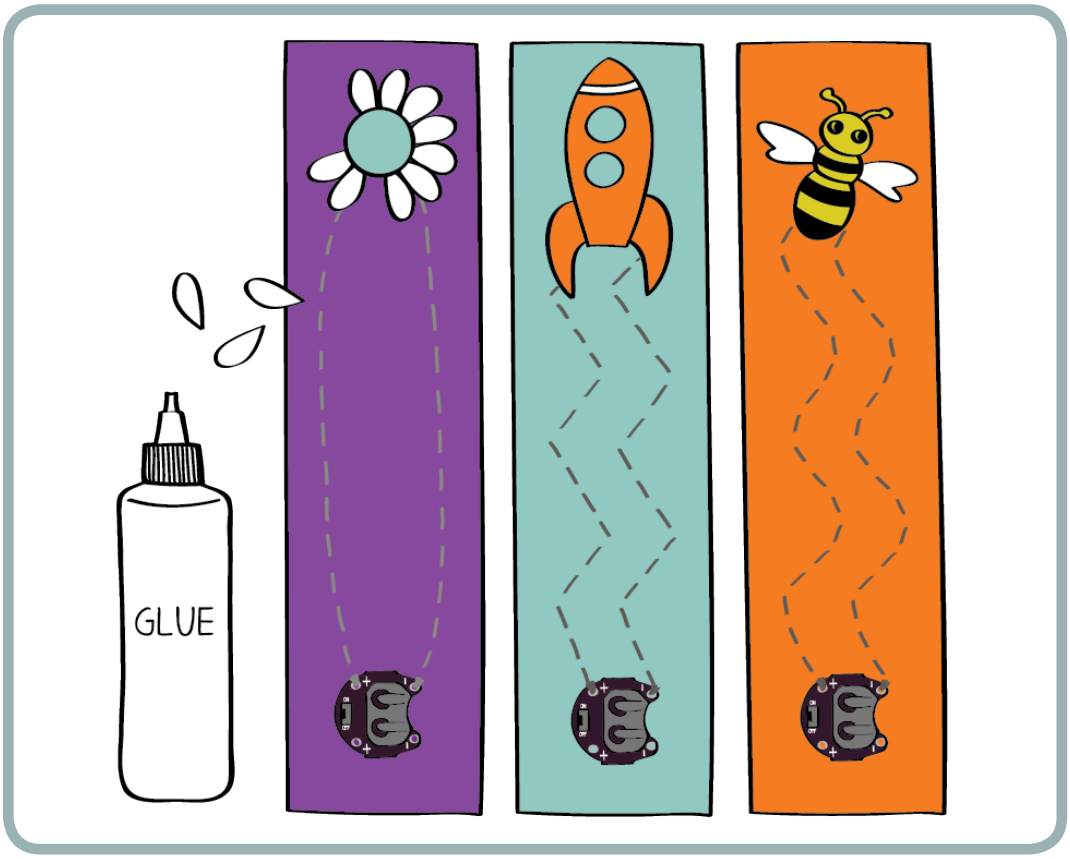
Electric current only flows through a circuit when there is a complete path leading from (+) on the battery through the circuit and back to (–) on the battery — from power to ground. If there is a break in this path, the electricity stops moving. If you patch the break in the path with a conductive material, the electricity is able to flow again.



This is how **switches** work. A switch opens up a break in a circuit and then closes it. When you flip a light switch on your wall, this is what it's doing. This is also what is happening when you flip the switch on your battery holder.

Decorate Your Bookmark

Now that your circuit is complete and you understand how it's working, it's time to decorate your bookmark. Following your original design, glue and sew on additional decorations to make your bookmark beautiful and personal.

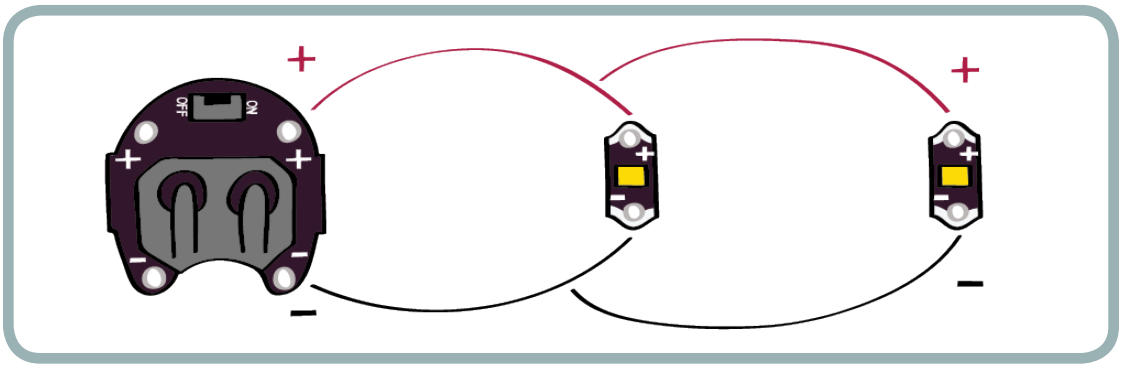


Do some midnight reading! Remember to switch your bookmark off when you're done.

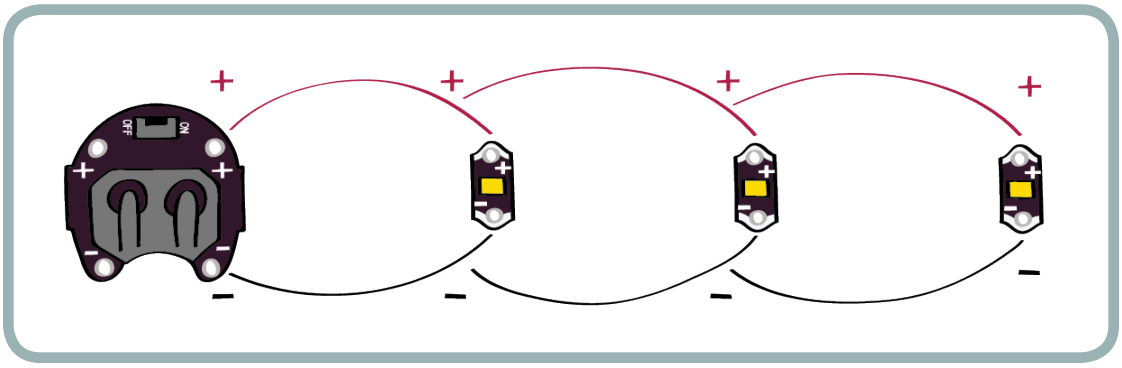
Experiment with Extensions: Add More LEDs

If you want to add extra bling to your bookmark, you can sew in a second LED. (If you're so inclined, you could even sew in a third, a fourth, a fifth, and so on.)

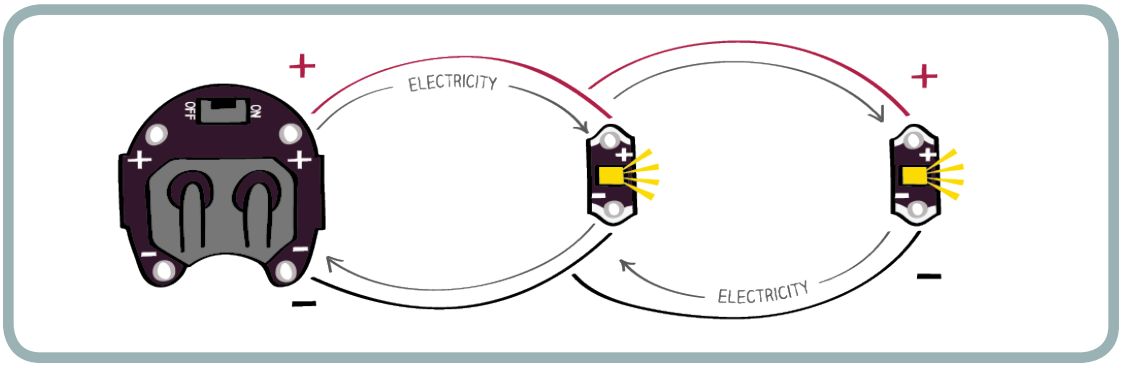
To add additional LEDs to your circuit, you need to stitch them on using the electrical configuration known as a parallel circuit. A **parallel circuit** with two LEDs (sewn on in parallel) looks like this:



A circuit with three LEDs sewn on in parallel looks like this:

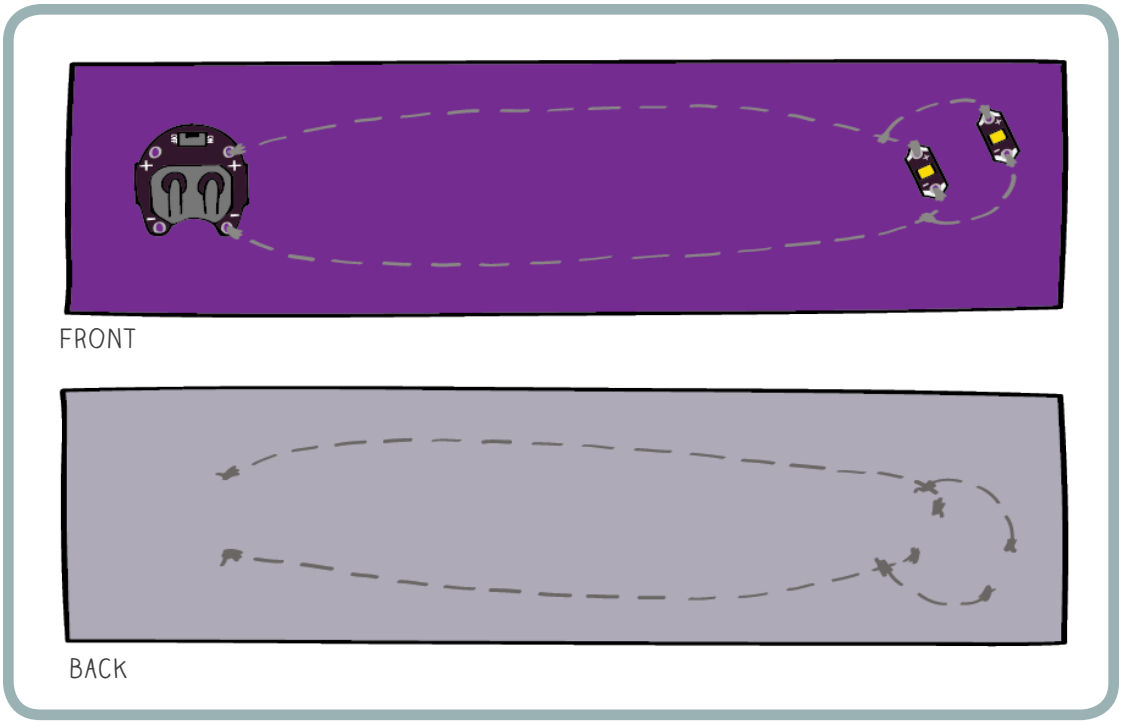


Note that in each of these diagrams, all the (+) tabs on the LEDs are sewn together and all the (–) tabs are sewn together.



To add an LED to your project, stitch the (+) tab of the new LED to the (+) trace of the original circuit and the (–) tab of the new LED to the (–) trace of the original circuit.

Where the new (+) trace intersects the old (+) trace, loop the new trace's thread several times around the old trace's thread before tying a knot, trimming it, and securing it with glue. Do the same at the point where the new (–) trace intersects the old (–) trace.



Experiment with Other Extensions

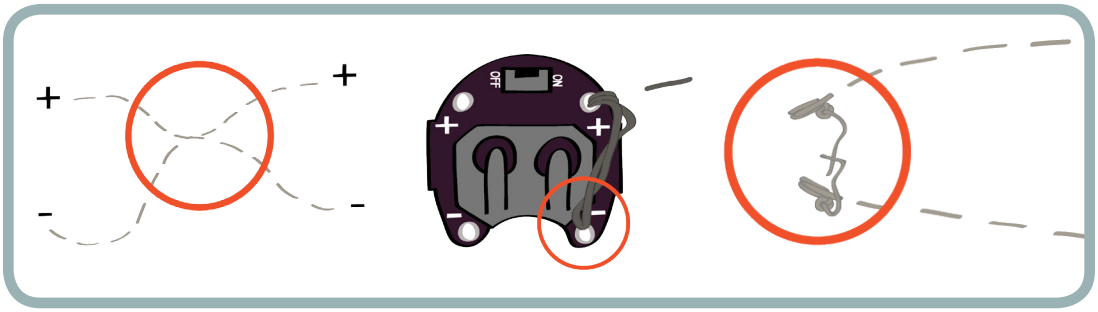
Can you add a switch to your circuit? Or build a new project that includes a switch? What about a switch made out of interesting materials such as metal beads, metal screws, or paper clips? Can you turn your bookmark into a bracelet? Or a hair clip? Or a wallet? Can you make your own flashlight, nightlight, or headlamp?

Now that you know how to sew circuits, you can stitch them almost anywhere — on your family's throw pillows, on your backpack, or on your T-shirts. Explore the possibilities!

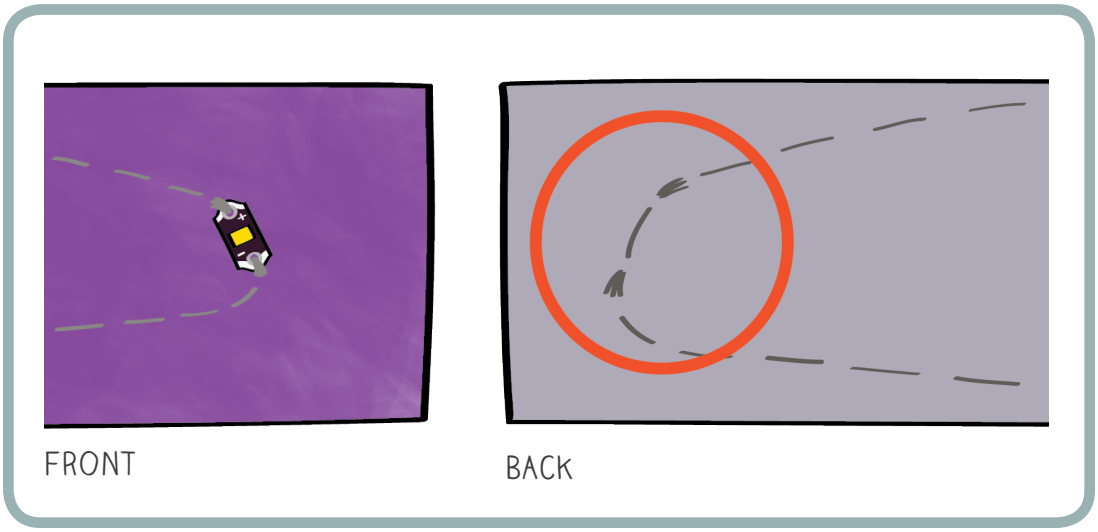
Troubleshooting:

If your LED doesn't light up:

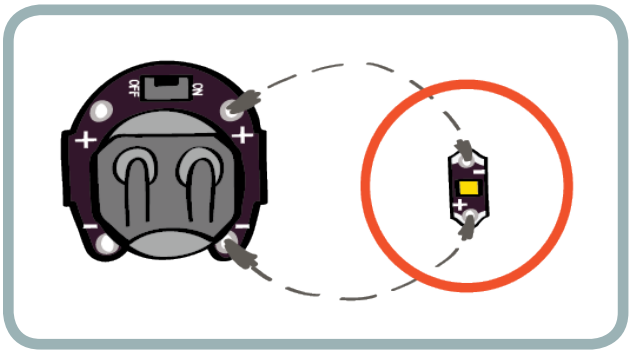
1. Make sure there are no short circuits. That is, make sure that the (+) and (–) traces never touch each other. Here are three examples of problems that may cause short circuits (from left to right): stitches that are too close to one another, a loose loop of thread, and knot ends that are too long.



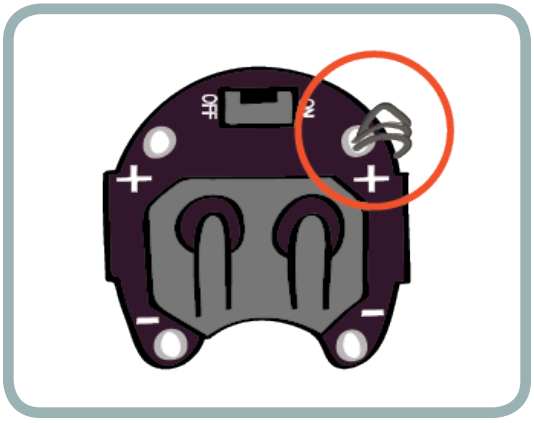
2. Did you use two pieces of thread in this project, or did you sew all the way from (+) to (–), across the LED, with one piece of thread? If you used only one piece of thread, you created a short circuit.



3. Make sure that you've sewn the (+) tab of the battery to the (+) tab of the LED and the (–) tab of the battery to the (–) tab of the LED. If the LED is reversed, it will not turn on. You should take it out and sew it back on in the correct orientation.

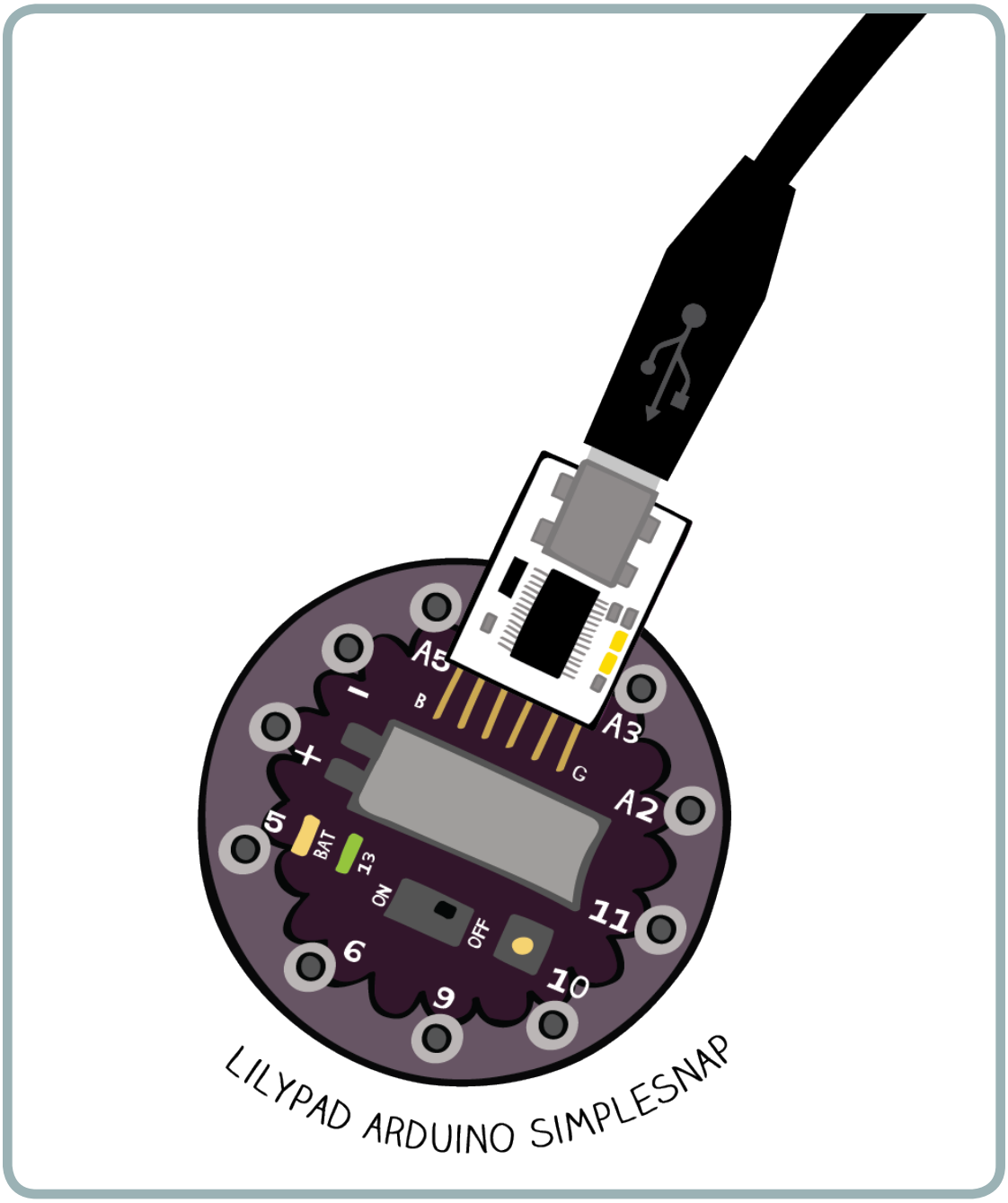


4. Check that the loops around each (+) and (–) tab on the battery holder and LED are snug. If your thread isn't touching one of the (+) or (–) tabs, you'll have a break in your circuit and no electricity will flow.



5. If you've tried everything else on this list and your LED still doesn't work, try a new coin cell battery — yours may be dead.

Introduction to Programming

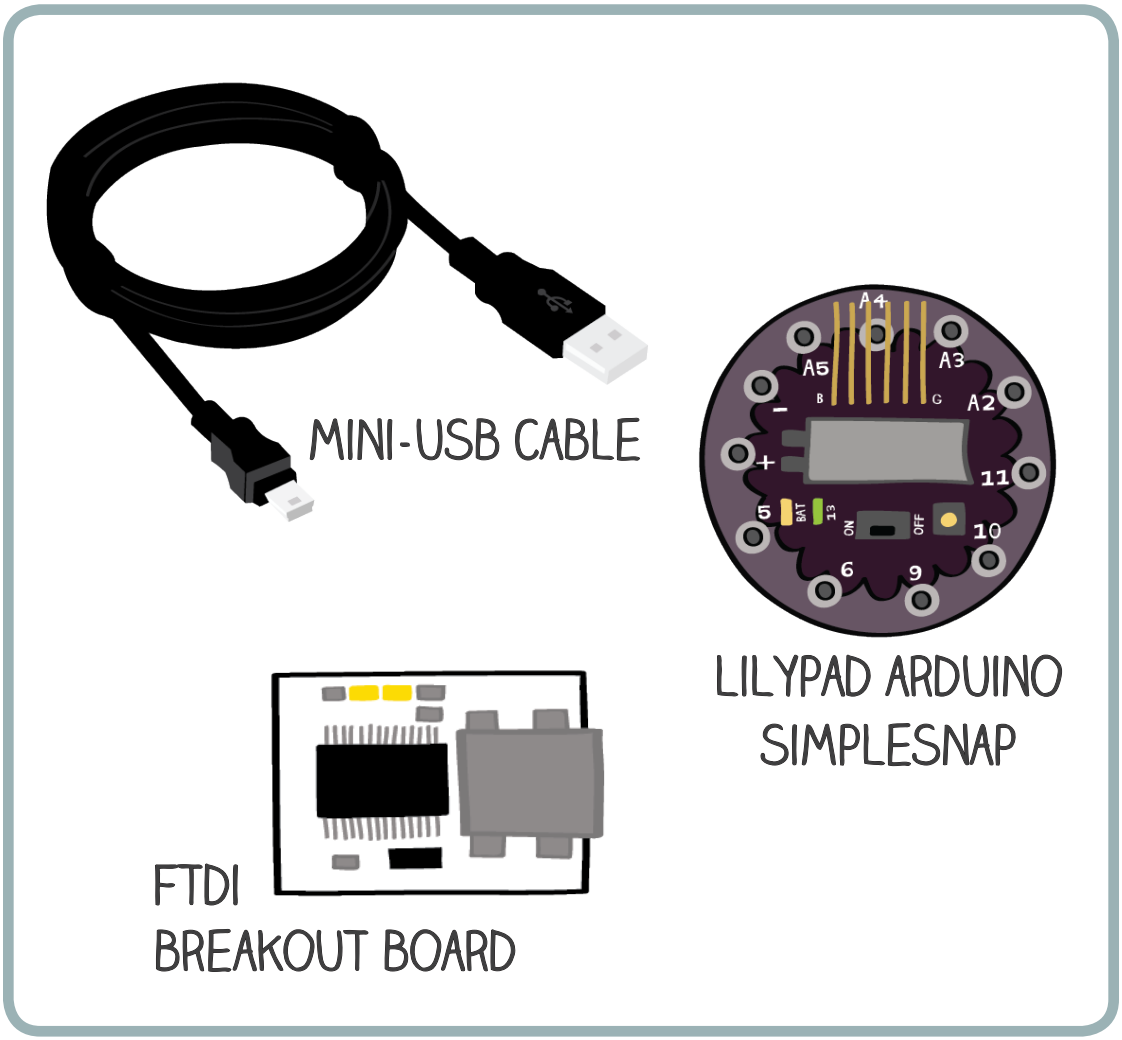


In this tutorial, you're going to learn how to write programs for the LilyPad Arduino SimpleSnap board, which is a small computer. You'll learn how to make an LED blink and flicker exactly how you want it to. You'll also begin to explore the ways that programming can make your e-textile projects more interesting, interactive, and personal.

TIME REQUIRED: 1–3 hours if Arduino software is preinstalled
3–5 hours if you are installing Arduino software

Collect Your Tools and Materials

You will write an Arduino program on a computer, transfer that program to a LilyPad Arduino SimpleSnap board, and run the program on the LilyPad. You'll need a computer, a LilyPad Arduino SimpleSnap board (referred to from now on as "LilyPad" or "LilyPad SimpleSnap"), a USB cable, and an FTDI breakout board to make connections.

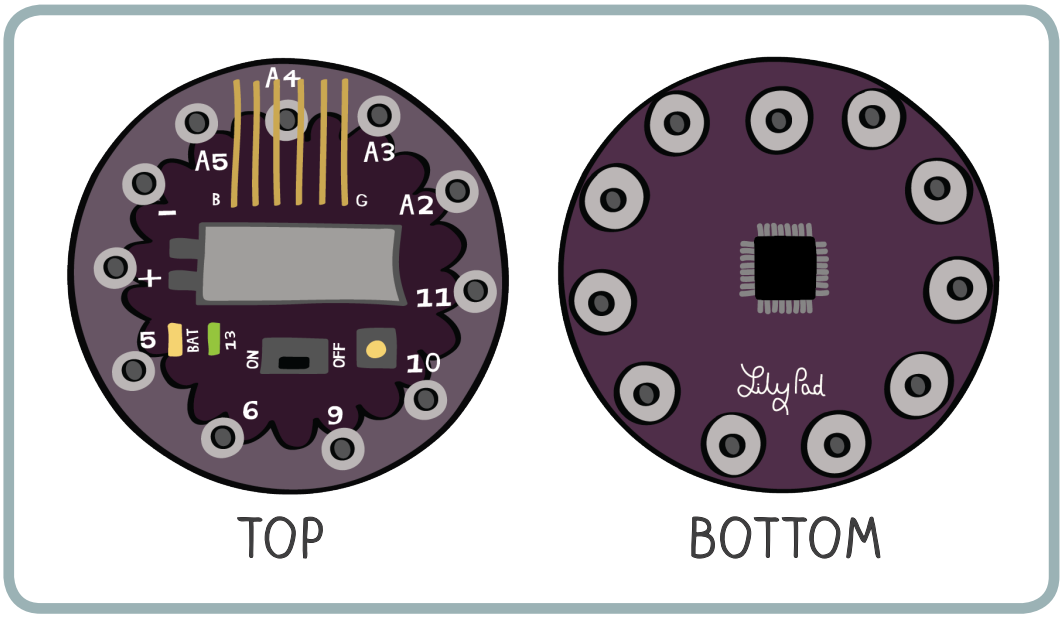


The LilyPad Arduino SimpleSnap

THE BASICS

Before you start programming, it's useful to know a few things about the LilyPad SimpleSnap — a small computer in a snap-on package. Take out your LilyPad and take a close look at it.

On the top side of the Lilypad, you will find an on/off switch, a push-button switch, a connector that will attach to your FTDI board, a battery, and two LED lights. There are also 11 silver circles on the edge of the board. We'll call these **tabs**. Notice that each tab has a label: +, -, 5, 6, 9, etc.



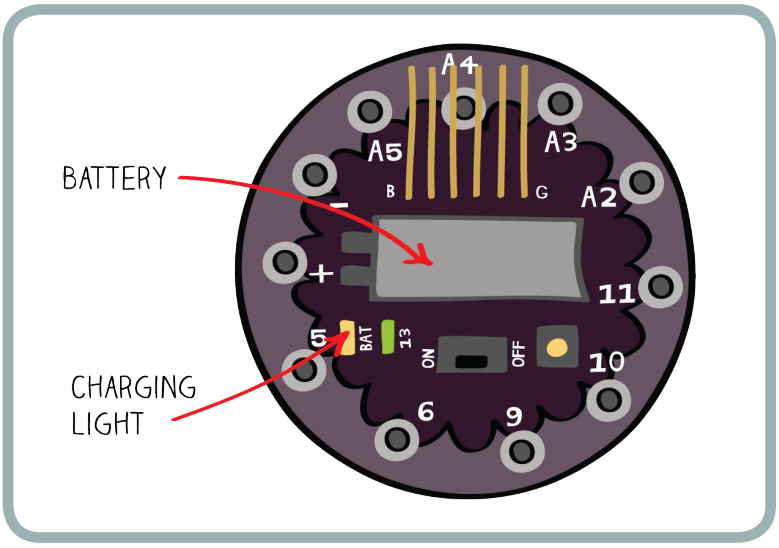
The bottom side of the Lilypad has a ring of snaps surrounding an assortment of electronics. Each snap is connected to one of the tabs on the top side of the Lilypad. The black square in the center of the board is a computer chip called a **microcontroller**. This tiny chip stores programs and controls the electronics that are connected to the LilyPad. Notice that the chip has tiny silver legs coming out of each of its sides. These legs are called **pins**.

Every tab on the LilyPad is connected to one of the pins on the chip. If you look very closely at the LilyPad, you can see some of the connections between the pins on the chip and the snaps on the bottom of the board. Because tabs are always attached to pins, we will refer to pins and tabs interchangeably.

THE BATTERY

The LilyPad SimpleSnap has a built in rechargeable battery. This is the large silver block on the top of the board. This means that you don't have to sew a battery to projects you build with this LilyPad — it's already built in.

When the LilyPad is plugged into your computer, the battery will charge. An orange LED on the LilyPad will shine while the battery is charging. When the battery is fully charged, this light will turn off.



Set Everything Up

This section describes how to set up your PC (Windows) or Apple (Mac OS X) computer and LilyPad SimpleSnap. If you have already done this (or if someone has done it for you), you can skip ahead to the next section, **Programs and Arduino**.

There are several steps to downloading and setting up the software you will use in this programming lesson:

A. Download and install the programming software.

B. Attach your LilyPad to your computer.

C. Select the appropriate serial port.

D. Select the appropriate Arduino board.

We also discuss:

E. Troubleshooting the Arduino software setup.

On the next few pages, you'll see two sets of instructions for downloading and setting up the software, one set for PC (Windows) computers and one set for Apple (Mac OS X) computers. Follow the instructions for your computer to download and install the programming software.

PC (WINDOWS) COMPUTER SOFTWARE DOWNLOAD AND SETUP

A. DOWNLOAD AND INSTALL THE PROGRAMMING SOFTWARE.

1. Go to this Arduino link: arduino.cc/en/Guide/ArduinoLilyPad.
2. Select Windows. You will see "Getting Started w/ LilyPadArduino on Windows."
3. Go to step 2, "Download the Arduino environment."
    a. Select the link for the latest version of Arduino IDE* from the download page.
    b. You will see two options: Windows Installer and Windows (Zip file). Choose Windows Installer.
    c. When you are prompted, download to your computer's Program folder.
    d. Agree to each Windows prompt.
    e. Make sure all the "Select Components to Install" boxes are checked.
    f. Install software.
    g. If you are prompted to Install Driver, click "Install."
    h. When you see "Completed," click "Close."
4. Go to your Windows start icon or program folder and click the Arduino icon to open.

B. ATTACH THE LILYPAD SIMPLESNAP TO YOUR COMPUTER.

1. Attach the FTDI breakout board to your LilyPad. (See the cover of this tutorial for a picture.)
2. Plug the small end of your USB cable into the FTDI board and the other end into your computer. There is a small battery in the processor. An amber light may come on, indicating that the battery is charging. If it does not light up, your battery is fully charged. Either way, proceed with the next step.

C. SELECT THE APPROPRIATE SERIAL PORT.

Now select a **serial port** so that the Arduino software knows which USB port your LilyPad SimpleSnap is attached to. The serial port is the communication channel through which your programs will be sent.

1. To select the correct serial port, in the Arduino menu, open "Tools ➔ Serial Port."
2. On the PC, choose the highest numbered "COM" port. [Note: You can also find your serial port by unplugging your LilyPad SimpleSnap, looking at the menu to note which serial port entries are there, and then plugging the Lilypad back in. The serial port entry that appears when you plug it back in is the one you want to select.]

Having trouble? See E. "Troubleshooting," below.

D. SELECT THE APPROPRIATE ARDUINO BOARD.

Now tell the Arduino software that you are using a LilyPad SimpleSnap and not some other Arduino microprocessor.

1. Under the "Tools" menu, select "Board," and then select "LilyPad Arduino w/ ATmega 328."

At the bottom of the programming environment, in white type, you will see the serial port and Arduino board you selected. You now have the proper setup for programming in Arduino!

E. TROUBLESHOOTING THE ARDUINO SOFTWARE SETUP

If you're having trouble installing or setting up the software, try one of these options:

1. Look through "Getting Started w/ Arduino on Windows," which you can find here: http://arduino.cc/en/Guide/Windows.
2. Look through the Arduino troubleshooting" guide on the Arduino website: http://arduino.cc/en/Guide/Troubleshooting.
3. Seek technical support from someone familiar with operating systems or programming.

APPLE (MAC OS-X) COMPUTER SOFTWARE DOWNLOAD AND SETUP

A. DOWNLOAD AND INSTALL THE PROGRAMMING SOFTWARE.

1. Go to this Arduino link: arduino.cc/en/Guide/ArduinoLilyPad.
2. Select Mac. You will see "Getting Started w/ LilyPadArduino on Mac OS X."
3. Go to step 2, "Download the Arduino environment."
    a. Select the latest version of Arduino IDE for Mac OS X. (Ignore options for beta, nightly builds, etc.) This will download and self-expand from a .zip file.
    b. You may be asked if you will allow an application to be downloaded from the Internet. Allow this by clicking "Open." (Otherwise, open it by double-clicking on the Arduino icon, which is teal green with an infinity sign.) The window that opens will be your programming environment.

B. Attach the LilyPad SimpleSnap to your computer.

1. Attach the FTDI breakout board to your LilyPad. (See the cover of this tutorial for a picture.)
2. Plug the small end of your USB cable into the FTDI board and the other end into your computer. There is a small battery in the processor. An amber light may come on, indicating that the battery is charging. If it does not light up, your battery is fully charged. Either way, proceed with the next step.

C. Select the appropriate serial port

Now select a **serial port** so that the Arduino software knows which USB port your LilyPad SimpleSnap board is attached to. The serial port is the communication channel through which your programs will be sent.

1. To select the correct serial port, in the Arduino menu, open "Tools ➔ Serial Port."
2. You should see an entry that starts with: "/dev/tty.usbserial- " followed by a series of letters and numbers. Select this port. (Note: You can also find your serial port by unplugging your LilyPad SimpleSnap board, looking at the menu to note which serial port entries are there, then plugging the board back in. The serial port entry that appears when you plug it back in is the one you want to select.)

Having trouble? See E. "Troubleshooting," below.

D. SELECT THE APPROPRIATE ARDUINO BOARD.

Now tell the Arduino software that you are using a LilyPad SimpleSnap board and not some other Arduino microprocessor.

1. Under the "Tools" menu, select "Board," and then select "LilyPad Arduino w/ ATmega 328."

At the bottom of the programming environment, in white type, you will see the serial port and Arduino board you selected. You now have the proper setup for programming in Arduino!

E. TROUBLESHOOTING THE ARDUINO SOFTWARE SETUP

If you're having trouble installing or setting up the software try one of these options:

1. Look through "Getting Started w/ Arduino on Mac OS X," which you can find here: http://arduino.cc/en/Guide/MacOSX.
2. Look through the troubleshooting guide on the Arduino website: http://arduino.cc/en/Guide/Troubleshooting.
3. Seek technical support from someone familiar with operating systems or programming.

Programs and Arduino

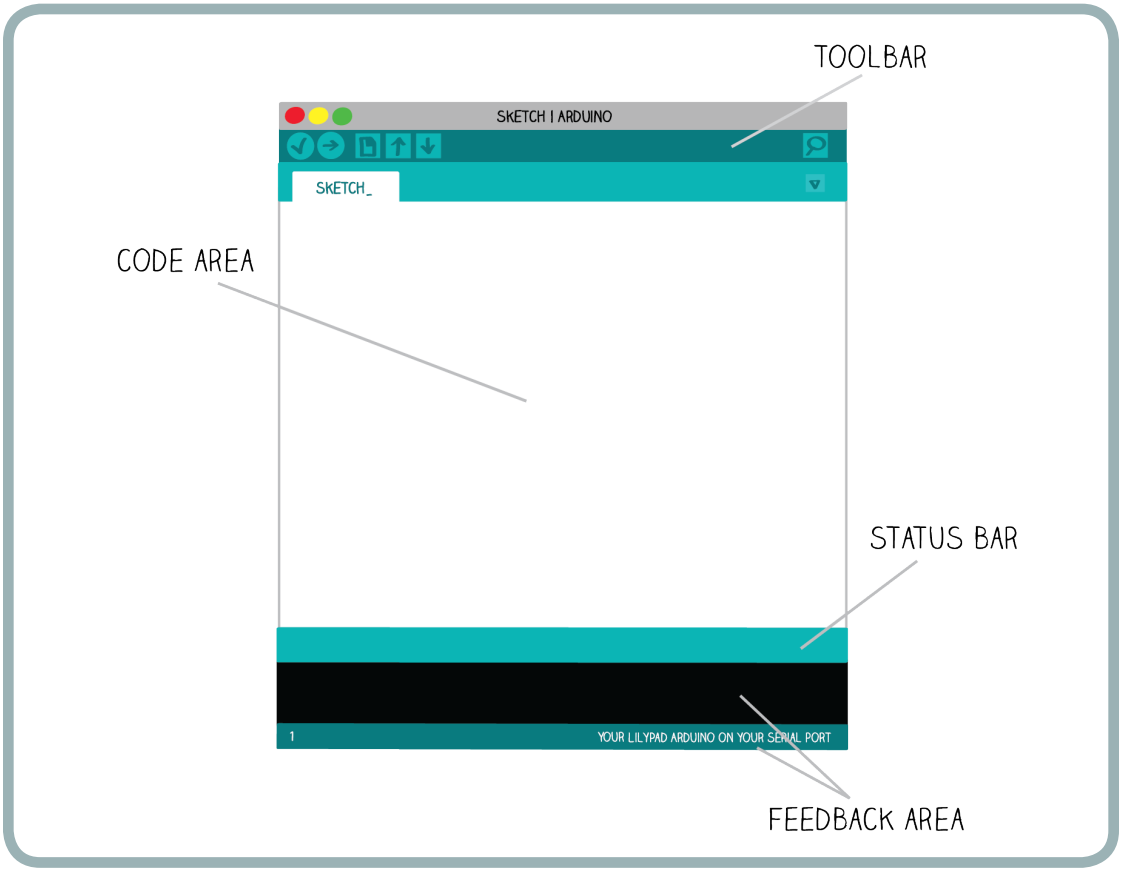
Before you start programming, it's worth taking a moment to reflect on what computer programs are and what they do. Computer programs are everywhere. Your Internet browser and music player are examples of complex computer programs. So are Google, Instagram, and the video games that you play at home and online. Computer programs can also be used to control electronics such as lights, motors, and speakers. You can also find computer programs in microwaves, cars, robots, and hairdryers.

Programmed computers carry out boring and repetitive jobs for us, make it possible for us to perform precise tasks at incredibly high speeds, and enable us to build dazzlingly complex systems. They allow us to communicate instantly with people around the world, and they enable us to express ourselves in compelling new ways.

A **program** (also called a piece of **code**) is a set of instructions written in a **programming language** that follows a very precise format. The program does its work when a computer "runs" or "executes" these instructions by following them in order.

THE ARDUINO DEVELOPMENT ENVIRONMENT

If you haven't done so already, open the Arduino software — also called the Arduino **development environment**. An empty window will pop up. This window is divided into four sections: a Toolbar, a Code Area, a Status Bar, and a Feedback Area.



1. **TOOLBAR.** The Toolbar provides you with a quick way to perform common tasks. Each icon in the Toolbar corresponds to a different action.

   - COMPILE YOUR CODE.
   - COMPILE AND UPLOAD YOUR CODE.
   - CREATE A NEW FILE.
   - OPEN AN EXISTING FILE.
   - SAVE THE FILE YOU'RE WORKING ON.
   - OPEN THE SERIAL PORT_MONITOR.

   Notice that when you hover your mouse over any icon, text pops up that explains what it's for.

2. **CODE AREA.** The Code Area is where you write programs.

3. **STATUS BAR.** The Status Bar gives you information about the status of your program. It tells you when your code is compiling or uploading and lets you know when there's an error that you need to fix.

4. **FEEDBACK AREA.** This is where you get feedback about the compiling and uploading processes. If your program cannot compile or the software can't communicate with your LilyPad SimpleSnap board, you will get a red error message in this box telling you what went wrong.

Basic Programming Steps

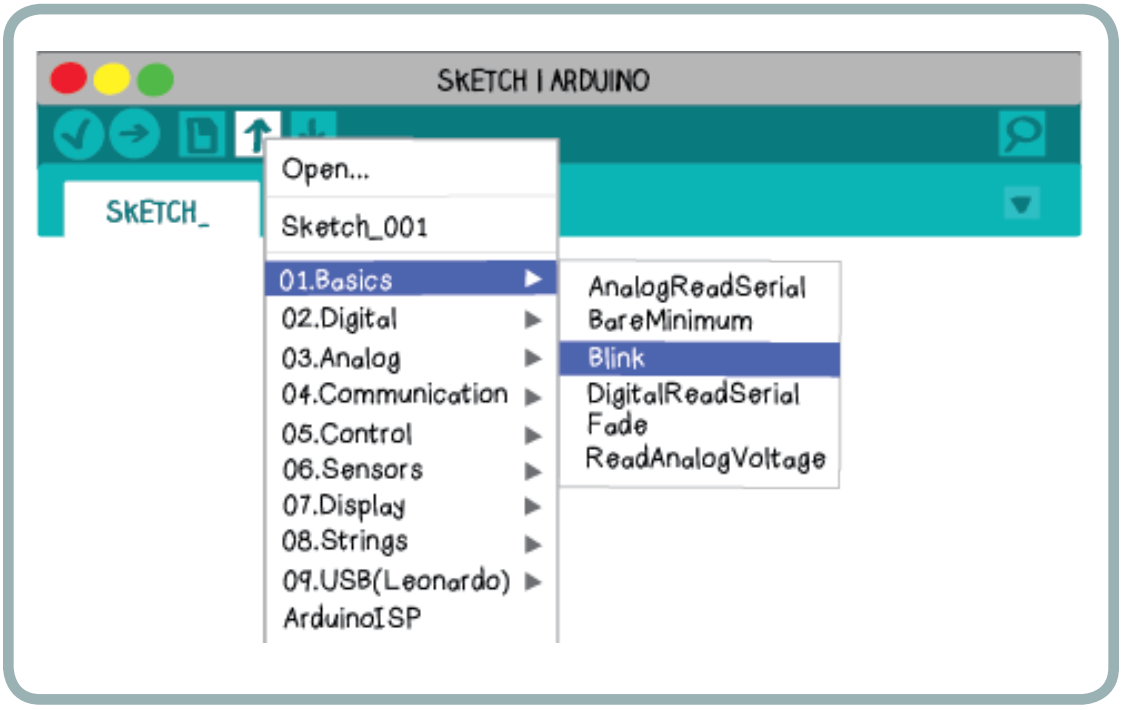
Now you are ready to start programming! There are four steps to writing a program and running it on your LilyPad:

1. You write the program.
2. You **compile** the program.
3. You load the program onto the LilyPad.
4. The program runs on the LilyPad.

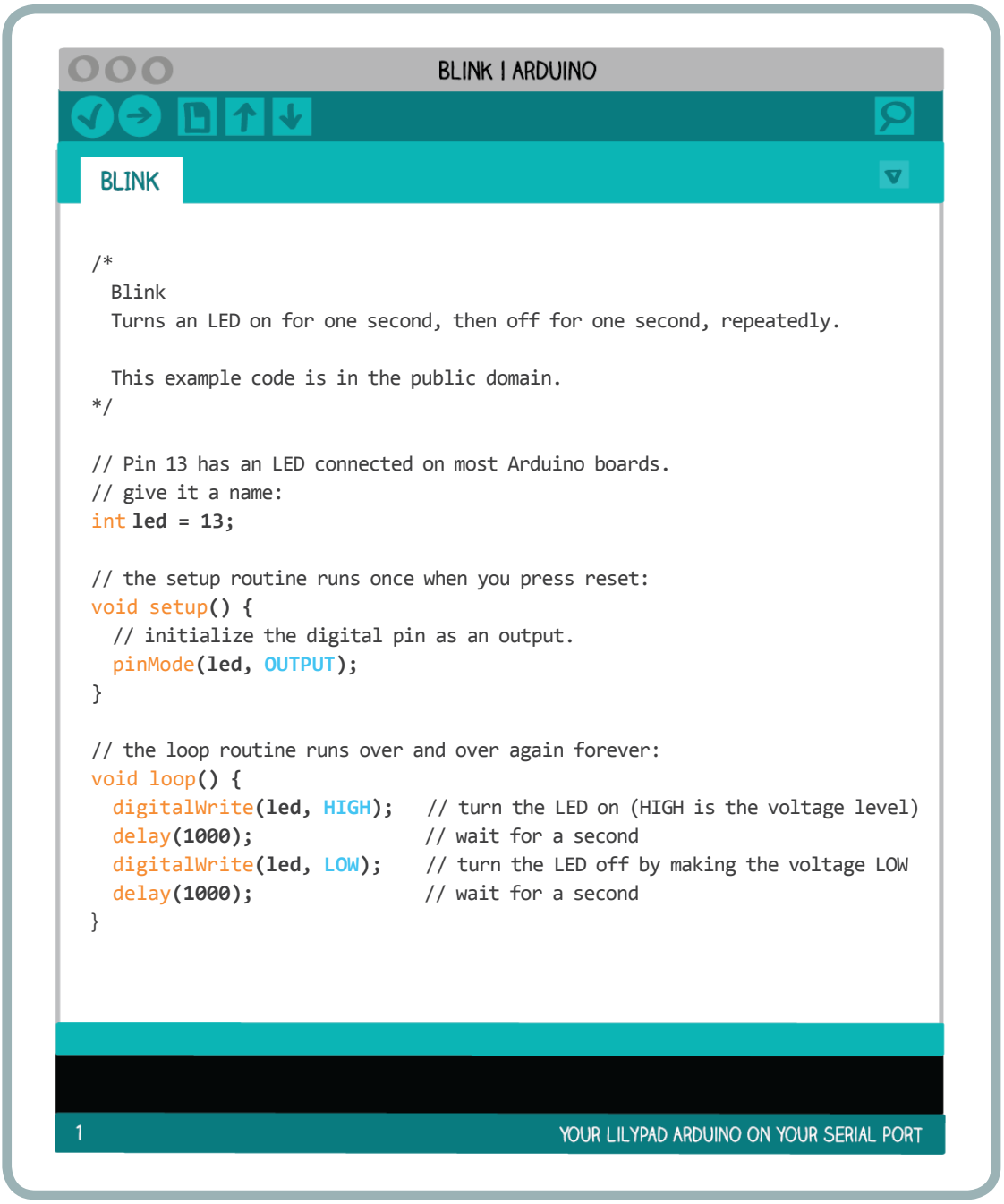
WRITE THE PROGRAM

You'll begin by trying out prewritten example code that is included in the Arduino software. First, you'll practice by using this prewritten code, and then you'll begin to write your own programs.

Click on the upward-pointing arrow in the Toolbar to open Arduino's built in library of examples. Select 01.Basics ➔ Blink. (You can also get a new window with this code by going to the File menu and selecting Examples ➔ 01.Basics ➔ Blink.)



The example program you opened looks like this:



This program is written in a **programming language** called C. The Arduino software only understands programs written in C, so you'll be learning how to write C programs.

For now, we're going to pretend that you have written this program. And once you've written a program, the next step is to **compile** it.

COMPILE THE PROGRAM

When a program is **compiled**, it is translated from the code that you wrote into a new code (called **hex code**) that the LilyPad can understand. The hex code is a very condensed form of the code that you wrote — essentially, it is a long string of numbers. Here's an excerpt of hex code:

```
:1038D00040E350E022533040404040504057FFFACF81
:1038E000962F9F5F692F981728F39093090208 95E8
:1038F000982F8091C00085FFFCCF9093C60008955B
:10390000EF92FF920F931F93EE24FF248701809183
:10391000C00087FD17C00894E11CF11C011D111D9A
:1039200081E0E81682E1F8068AE7080780E01807D8
```

As you can see, hex code is pretty hard to understand and would be even harder to write. This is why you write C code instead of hex code.

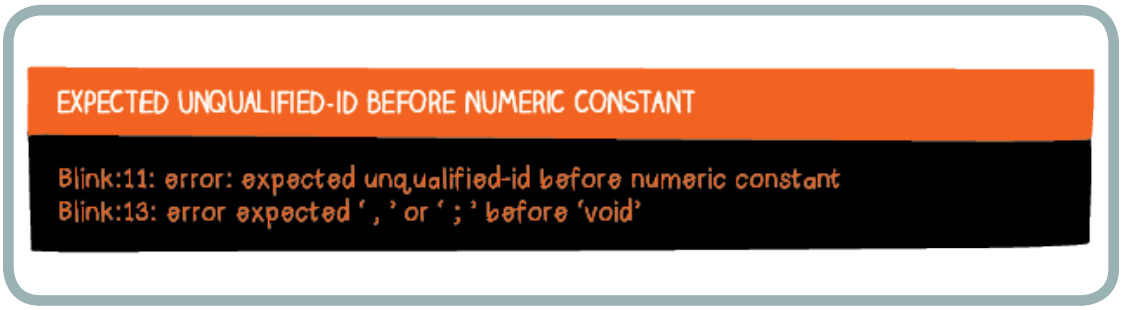
If you make any mistakes in your C code, they will be detected during the compiling phase and you will have to correct them before your code will compile. Only perfectly formatted and "grammatically correct" C code will compile into hex code.

To compile your Blink program, click on the checkmark button in the Toolbar. When you scroll over this button, you should see the word "Verify".

A progress bar that shows how long the compiling process will take will appear in the Status Bar.

If there are no errors in your program, your compilation will be successful and a "Done compiling" message will appear in the Status Bar. The Feedback Area will display a message that tells you the size of your compiled program.

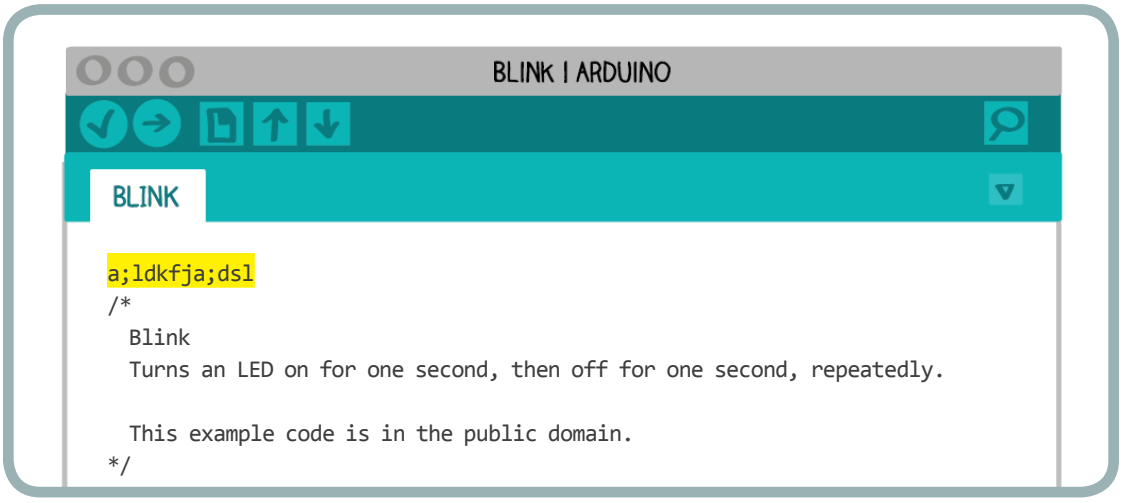
If there are errors in your program — if your C code is formatted incorrectly or if there are any "grammatical" mistakes — your compilation will fail. In this case, the Status Bar will turn orange and a confusing error message will appear. You will also get a more detailed and equally confusing error message in the Feedback Area.



Most of the error messages that appear in the Status Bar and Feedback Area are confusing to new programmers. As you get more familiar with them, they'll become more helpful.

Try editing your progam to introduce an error. Add a new line of random text to the top of the code (see example below). Click the compile button to see what happens. Most likely, the error message that appears will be completely meaningless to you. If you can make sense of it, you are a programming prodigy!

However, Arduino does one very useful thing when it detects an error. It highlights the line where the error was detected or moves the cursor near where the error was detected. This can be a helpful clue about where the error is that you need to correct. Keep this in mind as you write and **debug** your own programs.



Before you move on to the next step, remove the extra line you added, and click on the check mark to recompile your program.

LOAD THE PROGRAM ONTO THE LILYPAD

Once your program is succesfully compiled and hex code has been generated, the next step is to load this hex code onto your LilyPad SimpleSnap board. To upload the compiled code, click on the rightward-pointing arrow button in the Toolbar. When you scroll over this button, you will see the word "Upload."

In the Status Bar, you will see messages telling you that the program is being compiled and then uploaded. If the upload is successful, you will see a "Done uploading" message in the Status Bar. A message in the Feedback Area will tell you the size of your uploaded program and the amount of memory available on the LilyPad.

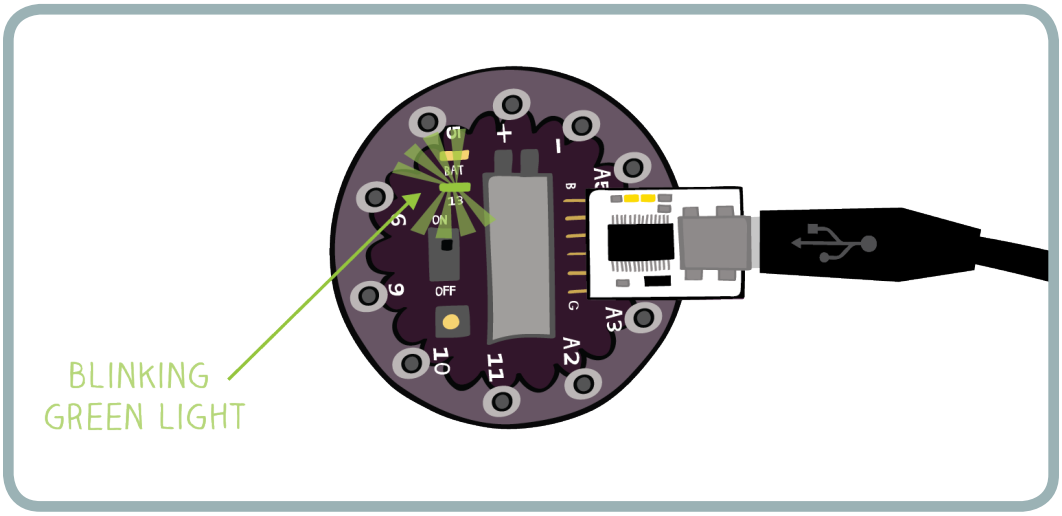
If you have not set up your LilyPad properly, the upload process will fail. The Status Bar will turn orange, and an error message will appear.

If you receive an error like this, return to the Setup section of this tutorial and make sure that you have completed all the necessary setup steps, including selecting the appropriate Serial Port and board in the Arduino software. If you are still having problems, look through the troubleshooting section of the Arduino website: http://arduino.cc/en/Guide/Troubleshooting.

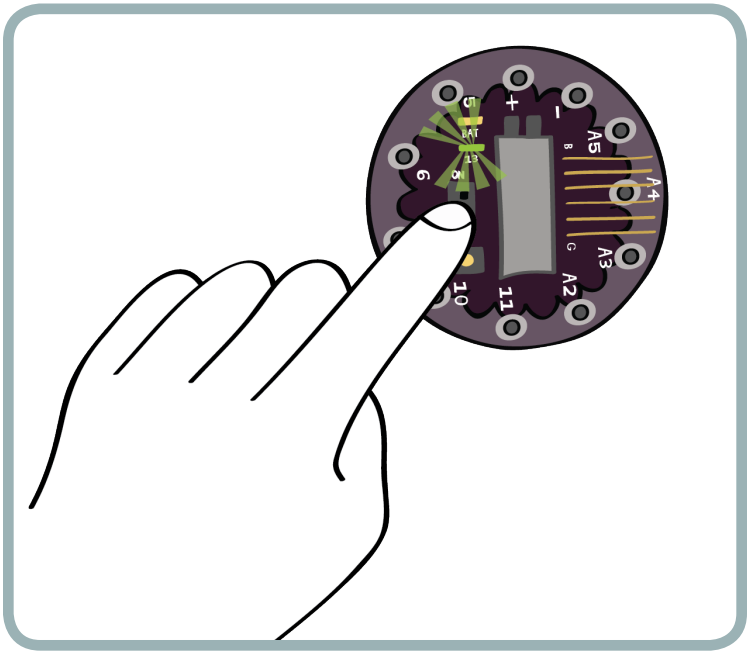
RUN THE PROGRAM ON THE LILYPAD

Look at your LilyPad. It should be blinking!

After the Blink program is uploaded, a green LED on the LilyPad should start to flash on and off. It'll turn on, wait a second, turn off, wait a second, and continue doing this until the LilyPad is turned off or reprogrammed.



Try unplugging the LilyPad from your FTDI board. Flip the on/off swith on the LilyPad to off. The LilyPad should stop blinking. Now, flip the on/off switch on. The LilyPad should begin flashing again.



When you upload a program to the LilyPad, that program is stored in the LilyPad's memory. This means that the LilyPad can run the program independently of the computer. Once you upload a program to the LilyPad, you can unplug the LilyPad and take it with you. It will remember exactly what you told it to do!

Attach your LilyPad to your computer again so that you can keep experimenting.

Make the LED Blink Faster

Take a closer look at the example code. The last six lines of the program look like this:

```
// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off
  delay(1000);               // wait for a second
}
```

The four lines of code between the curly brackets after "void loop()" are the heart of the Blink program.

- The first line, `digitalWrite(led, HIGH);`, is the code that turns the LilyPad's LED on.
- The second line, `delay(1000);`, tells the LilyPad to do nothing for one second (1000 milliseconds).
- The third line, `digitalWrite(led, LOW);`, is the code that turns the LED off.
- The fourth line is the same as the second.

(Notice that comments at the end of each line describe what the code is doing.)

See if you can change this code to get the LED on the LilyPad to blink faster. Hint: To get your LED to blink twice as fast, change each `delay(1000);` line to `delay(500);`.

```
// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on
  delay(500);                // wait for a second
  digitalWrite(led, LOW);    // turn the LED off
  delay(500);                // wait for a second
}
```

Make the changes, compile your code (by clicking on the check mark in the Toolbar), and upload the new code to your LilyPad (by clicking on the right pointing arrow).

Experiment with different blinking speeds. What happens when you try a very short delay? (Note: The shortest delay time you can use is 1. In addition, decimal numbers will not work in this program.) Try gradually increasing the delay. Re-upload the code to your LilyPad for each new delay value. Is there an interesting turning point when the LED's behavior seems to change? What do you think might be happening?

Once you've gotten your LED to blink quickly, see if you can get it to blink very slowly. Can you get it to blink every two seconds or every four seconds? Can you get it to blink in an uneven pattern, like a heartbeat? If you're programming with a friend or in a group, compare your code and LED behavior with theirs.

SAVE YOUR CODE

Once you've created a blinking behavior that you like, click on the downward-pointing arrow in the Toolbar to save your code. When you scroll over this button, you'll see the word "Save."

Click "OK" on the popup window that appears. Choose a good name for your file (such as "JanesFirstBlinking"), and click on the "Save" button to complete the process.

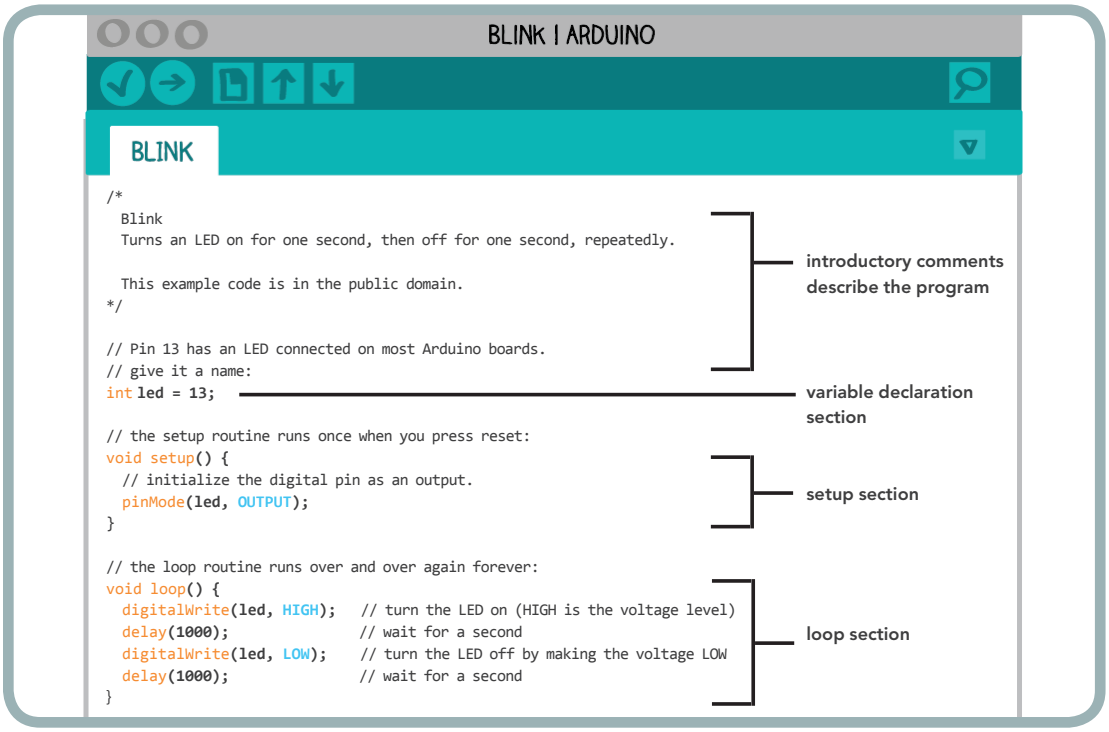
To make sure that the file saved properly, click on the upward pointing arrow in the Toolbar. This is the "Open" icon in Arduino; when you scroll over it you'll see the word "Open."

The file you just saved should appear at the top of the popup menu that appears. All the programs that you save in Arduino will appear in this menu so that you can find and open them easily.

Basic Code Elements

You just did some simple coding. Now, you'll explore some of the basic elements of the C language. Reopen the basic Blink example by clicking on the upward-pointing arrow in the Toolbar and then selecting 01.Basics ➔ Blink.

There are three code elements in the Blink example: comments, variables, and simple statements. Each element has its own place in the programming environment, as shown here.



COMMENTS

The very first part of the Blink example code looks like this:

```
/*
  Blink
  Turns an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
 */
```

This is a **comment**, a piece of text that will be ignored by the computer when the code is compiled. Any block of text between **/*** and ***/** characters is a comment.

You can use comments to make notes — for yourself and others — in your programs. They do not affect the behavior of your code. In the Arduino environment, comments appear in greyish brown text.

Anything written on a line after two slash characters // is also a comment. The two slash characters // can only create comments that are a single line long — not multiple lines. There are several of these single-line comments in your example code. Here's one:

```
// initialize the digital pin as an output.
```

It is easy to add comments to a program. For example, you could add a new comment to the very top of your code:

```
// Mary had a little lamb, little lamb, little lamb...
/*
  Blink
  Turns an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
 */
```

Try adding your own comment to the code and then uploading your new program to the LilyPad. You can add comments anywhere you want in the program, and they won't change the behavior. Arduino will automatically change their color to grey.

Comments can also be useful when you want to temporarily remove a piece of code from your program. For example, if you put // in front of the `digitalWrite(led, LOW);` line in your program, that line is skipped when you compile and upload the code to the LilyPad. This is called "commenting out" pieces of code. Try making this edit and then compiling and uploading the code. What happens? Look at the new code carefully. Can you see why the LED's behavior has changed?

```
void loop() {
  digitalWrite(led, HIGH);     // turn the LED on
  delay(500);                  // wait for a second
  //digitalWrite(led, LOW);    // turn the LED off
  delay(500);                  // wait for a second
}
```

Now, remove the // you put in front of the the `digitalWrite(led, LOW);` line. Compile and upload your code so that your LED blinks again.

VARIABLES

Writing a program is a lot like writing a cooking recipe. When you write a recipe, you follow a basic structure. You list the ingredients at the beginning of the recipe, and you write instructions in the order that they need to be carried out. (You don't, for instance, tell a cook to put a pie in the oven before you explain how to make dough for the crust.)

When you write a program, you also write instructions in the order that they need to be carried out. You list "ingredients" — pieces of code that you'll use in the rest of your program — at the top of your file. A computer, like a good cook, will read and carry out a program in the order it's written.

Variables in a program are like ingredients in a recipe. They are generally listed at the beginning of a program, and they identify the components that the program will be controlling.

After the comment at the top of the program, the next section of the Blink code looks like this:

```
// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;            // set the variable "led" to the value 13
```

The first two lines in this section are comments. The third line, `int led = 13;`, creates a **variable** called **led**. This line identifies the critical ingredient of the Blink program, the LED.

The green LED on the LilyPad board is connected to pin 13 on the LilyPad's computer chip. The line `int led = 13;` is telling the LilyPad (and the programmer) that something called **led** is attached to pin 13.

More generally, when you create a variable, you set aside a chunk of memory in the LilyPad and give it a name. The `int led = 13;` line creates a variable called **led** and stores the value 13 in that variable.

There are different types of variables in the same way that there are different types of computer files. Computer file types are distinguished by suffixes (for example, .docx, .pdf, and .jpg). Variable types are specified when they're created. In the example above, the text **int** describes what type of variable **led** is. **int** stands for "integer," which means a whole number. Almost all the variables you will use in your Arduino programs will be of type int.

The variable line of code can be broken down like this:

Variable Type	Variable Name	Assigned To	Value Stored In Variable
int	led	=	13

Any line of code that creates and assigns a value to a variable has this same basic structure.

When the program is running (after it's compiled and uploaded), whenever the LilyPad finds the variable **led**, it replaces it with the number **13**. What looks like this to you now:

```
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off
  delay(1000);               // wait for a second
}
```

will look like this to the LilyPad when the program is running:

```
void loop() {
  digitalWrite(13, HIGH);    // turn the LED on
  delay(1000);               // wait for a second
  digitalWrite(13, LOW);     // turn the LED off
  delay(1000);               // wait for a second
}
```

To begin to understand why variables are useful, add one to your program. Add a variable called **delayTime** to your program, and set its value to 1000. Add this line right after the `int led = 13;` line.

```
// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;            // set the variable "led" to the value 13
int delayTime = 1000;
```

Now, replace every `delay(1000);` line in your program with `delay(delayTime);`.

```
void loop() {
  digitalWrite(led, HIGH);        // turn the LED on
  delay(delayTime);               // wait for a second
  digitalWrite(led, LOW);         // turn the LED off
  delay(delayTime);               // wait for a second
}
```

Compile this code, and upload it to your LilyPad. The LED should blink every second, just like before. Now try changing the initial value of delay time from 1000 to 500, by editing the `int delayTime = 1000;` line to `int delayTime = 500;`

```
int delayTime = 500;
```

Compile this code, and upload it to your LilyPad. What happens to the behavior of your LED? Notice that you only have to change one line in your program to change the speed of the LED's blinking. Before, when you changed the timing in your code, you had to change at least two lines.

Variables help you keep track of the critical features (ingredients) of your code. When you use a variable, all the important information about it — its name and value — is listed once at the beginning of the program where it can be easily and quickly changed.

If you would like to save the changes you made to your code, click on the downward-pointing arrow in the Toolbar. Click "OK" on the popup window that appears, and choose an appropriate name for your file. Click on the "Save" button to complete the process.

SIMPLE STATEMENTS

Begin this next section with a fresh version of the Blink program. Reopen the Blink example by clicking on the upward-pointing arrow in the Toolbar and then selecting 01.Basics ➔ Blink.

Statements are essentially computer sentences — lines of code that tell the computer to do something. The line of code `digitalWrite(led, HIGH);` is an example of a simple statement. All simple statements end with a semicolon (;), just as all sentences end with a period. Your code is full of simple statements. Here are a few:

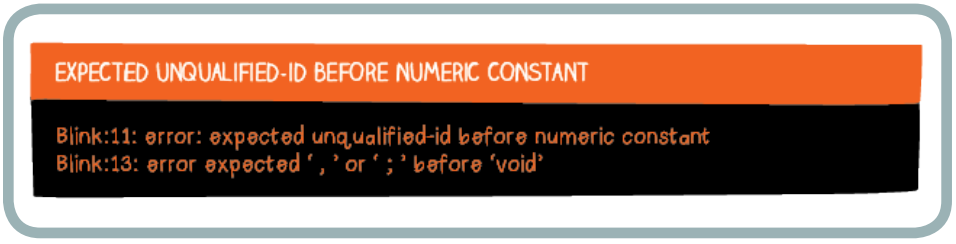
```
int led = 13;
```

```
digitalWrite(led, HIGH);
```

```
delay(1000);
```

Each of these lines tells the computer to do something. Each ends with a semicolon. Note that the comment to the right of each statement tells you what each one does.

Try deleting the semicolon at the end of the `int led = 13;` line. Compile your code. What happens? Do you see an error message?



When you're writing your own code, it's easy to forget to put a semicolon at the end of a statement. The Arduino software will not compile or upload your code until it is perfectly punctuated.

Errors like missing semicolons can be tricky to find and fix, but Arduino does try to help you. Notice that when you compiled the code, the cursor in Arduino jumped to the line immediately following the `int led = 13;` line. This is a clue about where the error is. The other clue is in the cryptic error message that appears in the Feedback Area: `Blink:13: error: expected ',' or ';' before 'void'`. Notice that the message says "`expected ',' or ';'`". It's telling you that the error might be a missing semicolon. You will gradually learn to make more sense of the strange error messages!

Replace the semicolon at the end of the `int led = 13` line, and recompile your code.

To experiment with statements, you are now going to create an uneven blink pattern. Modify the program so that the LED:

- turns on for one second
- turns off for one second
- turns on for half a second
- turns off for half a second

Add four statements to your code to achieve this behavior:

```
void loop() {
  digitalWrite(led, HIGH);        // turn the LED on
  delay(1000);                    // wait for a second
  digitalWrite(led, LOW);         // turn the LED off
  delay(1000);                    // wait for a second
  digitalWrite(led, HIGH);        // turn the LED on
  delay(500);                     // wait for half a second
  digitalWrite(led, LOW);         // turn the LED off
  delay(500);                     // wait for half a second
}
```

Compile and upload this new program to your LilyPad, and see what happens. Notice how the additional statements change the blinking pattern.

Try adding, removing, or editing a few statements of your own to get different patterns. Have you tried copying and pasting lines of code yet? This is a fast method for writing code and also a way to avoid errors.

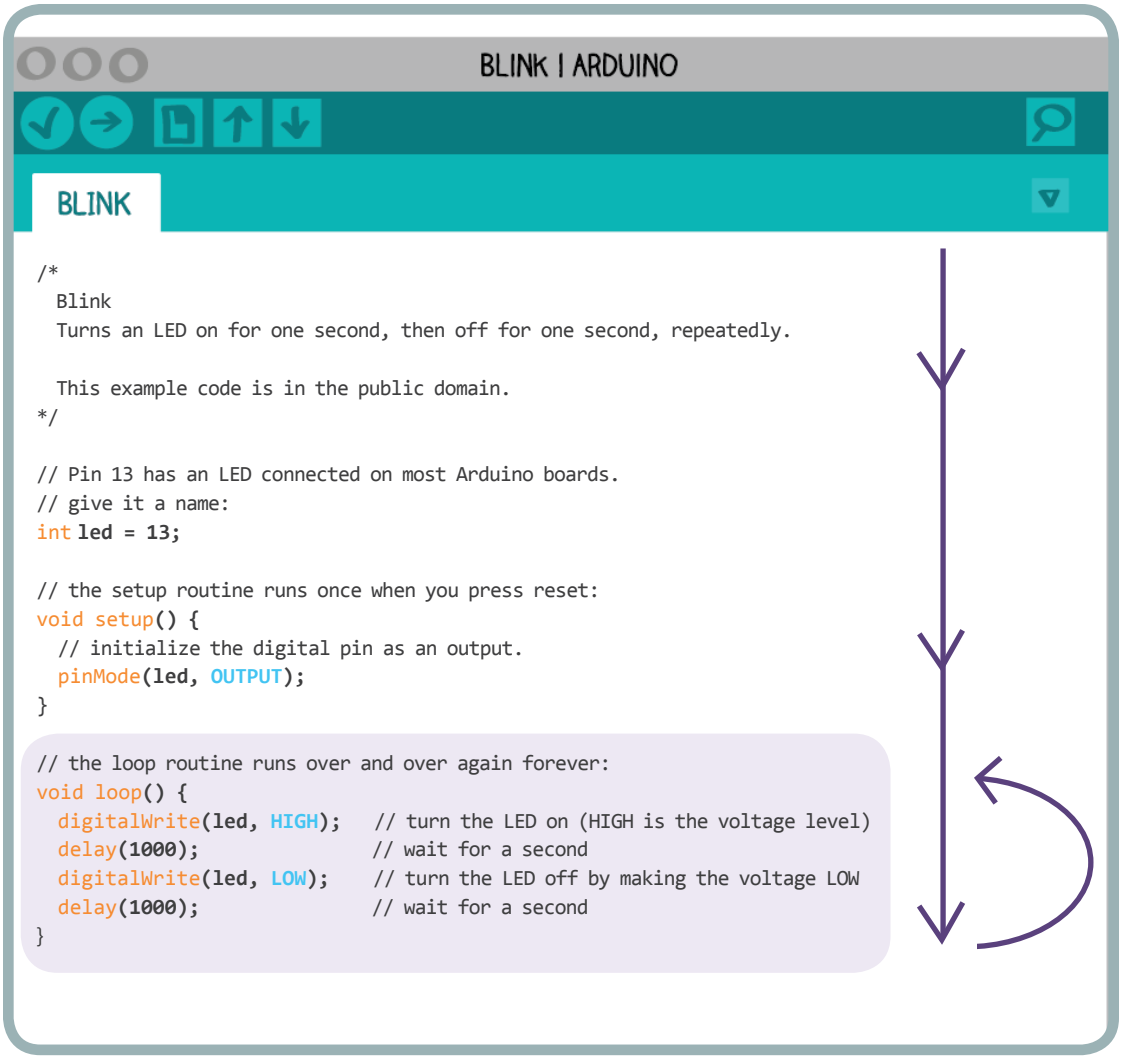
Once you create a blinking pattern you are happy with, click on the downward-pointing arrow in the Toolbar to save your changes. Click "OK" on the popup window that appears, and choose a name for your file. Click on the "Save" button to complete the process.

Arduino Program Structure

The previous section looked at different code elements in the Arduino/C language. Now you will see how they come together into a complete program. Reopen the Blink example by clicking on the upward-pointing arrow in the Toolbar and then selecting 01.Basics ➔ Blink.

Each Arduino program has three main parts: a section where you declare variables, a **setup** section, and a **loop** section. Note: These sections are often preceded by comments that describe what the program is doing. The three main sections are analogous to the ingredient list, preparation steps, and cooking steps in a recipe. Here is a diagram showing where the three parts are located in the code:

When your program runs, it will first define your variables (the ingredients that you need), then execute the setup section once (set everything up to begin cooking), and then execute the loop section over and over (actually do the cooking).



VARIABLE DECLARATION SECTION

Variables are declared at the beginning of a program. All the variables that you are using in your code should be listed at the top of your program, before the setup and loop sections.

SETUP SECTION

The setup section, which runs once when the program begins, follows the variable declaration. Statements that lay the foundation for actions that happen later on in the program should be put in the setup section. Note that all statements in the setup section are placed between an open curly bracket "{" and a closed curly bracket "}". These brackets immediately follow `void setup():`

```
// the setup routine runs once, when the program first starts:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}
```

These brackets tell the Arduino software when the setup section begins and ends. Without them, the software is lost and unable to compile your code. Think of these curly brackets as additional punctuation that your code requires. Your code must be perfectly punctuated to compile.

Try removing the closing bracket of the setup section and compiling your code.

```
// the setup routine runs once, when the program first starts:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
```

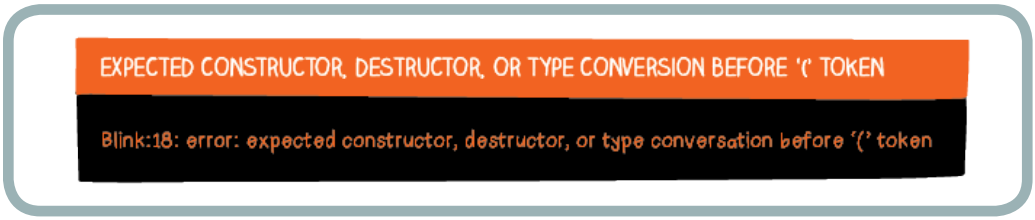
The error message you get in this case is not particularly helpful:

However, it does provide a clue to the problem in the last line: `Blink:24: error: expected '}' at end of input`. This identifies the problem as a missing } somewhere in your program.

Replace the closing curly bracket and recompile your code. Now, move the `pinMode(led, OUTPUT);` statement to the line after the closing curly bracket.

```
// the setup routine runs once, when the program first starts:
void setup() {
  // initialize the digital pin as an output.
}
pinMode(led, OUTPUT);
```

Compile your code. You should get another baffling error message.



However, Arduino also highlights the misplaced `pinMode(led, OUTPUT);` line in yellow — a good clue about the source of your problem.

Return the `pinMode(led, OUTPUT);` line to its proper position in the setup section, and recompile your code.

It's useful to familiarize yourself with these errors, because missed or misplaced curly brackets are another common source of problems. Remember that the setup and loop sections should always begin with an open curly bracket and end with a closing curly bracket. Statements should only ever appear above the setup section (variable declarations appear there), inside the setup section, or inside the loop section.

If you encounter a compile error, the first things to check for are missing semicolons and misplaced or missing curly brackets.

Loop section

After the setup section runs, the loop section runs over and over until the LilyPad is turned off or reprogrammed. The statements that carry out the main action of your program are placed in this section. As with the setup section, statements in the loop section are placed between open and closed curly brackets immediately following `void loop()`:

```
void loop() {
  digitalWrite(led, HIGH);    // turn the LED on
  delay(1000);                // wait for a second
  digitalWrite(led, LOW);     // turn the LED off
  delay(1000);                // wait for a second
}
```

You will receive frustrating compile errors if either of the curly brackets is missing or if any statements are placed after the closing curly bracket or before the opening curly bracket.

Here are examples of loop sections that will not compile. See if you can find the problem in each example.

```
void loop() {
  digitalWrite(led, HIGH);    // turn the LED on
  delay(1000);                // wait for a second
  digitalWrite(led, LOW);     // turn the LED off
}

delay(1000);                  // wait for a second
```

```
void loop()
  digitalWrite(led, HIGH);    // turn the LED on
  delay(1000);                // wait for a second
  digitalWrite(led, LOW);     // turn the LED off
  delay(1000);                // wait for a second
}
```

```
void loop()
  digitalWrite(led, HIGH);    // turn the LED on
{
  delay(1000);                // wait for a second
  digitalWrite(led, LOW);     // turn the LED off
  delay(1000);                // wait for a second
}
```

Experiment

Now that you have a foundational understanding of Arduino programs and Lilypads, experiment with your LED's behavior. Can you get your LED to blink in a heartbeat pattern? A seemingly random pattern? Can you get it to flicker like a candle?

Think about some trickier challenges. How could you get the LED to gradually fade on and off? How could you make it blink more and more slowly over time? Hint 1: Both of these challenges will require additional variables to keep track of blinking speed. Hint 2: You will need to change the values stored in these variables each time loop runs.

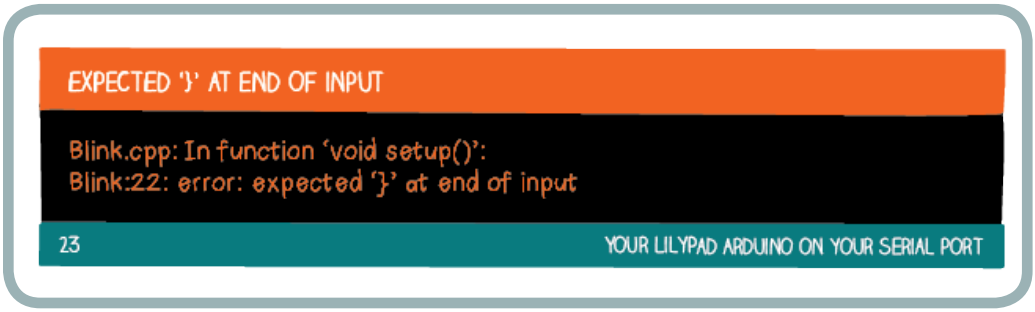
Remember to save any programs you like by clicking on the downward-pointing arrow in the Ardunio software so that you can return to them later. You can give each program a different name.

Troubleshooting:

As you might have discovered already, the programming process involves finding and fixing a lot of errors.

HOW TO READ AN ERROR

Errors show up on an orange background in the Status Bar.



There are two types of errors:

1. An error in your code's "syntax" — its grammar, punctuation, or spelling. These are also called **software errors** since they are about code.
2. An error communicating with the LilyPad board. These are called **hardware errors** since they are usually about the physical connection between your LilyPad and computer.

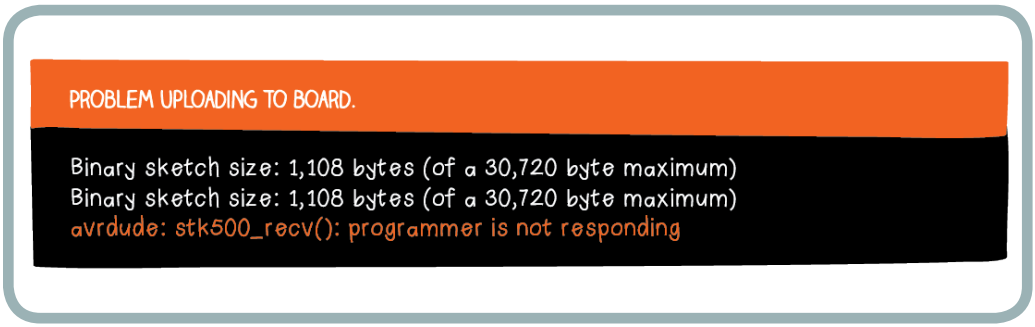
Software errors appear when you attempt to compile your code. You can fix them by changing your code.

Hardware errors appear when you attempt to upload your code. They occur when there is a problem in the connection between the LilyPad board and your computer.

The rest of this section describes a few common hardware and software errors.

HARDWARE ERROR: PROGRAMMER NOT RESPONDING

The most frequent hardware error is the "Programmer not responding" or "Problem uploading to board" error.

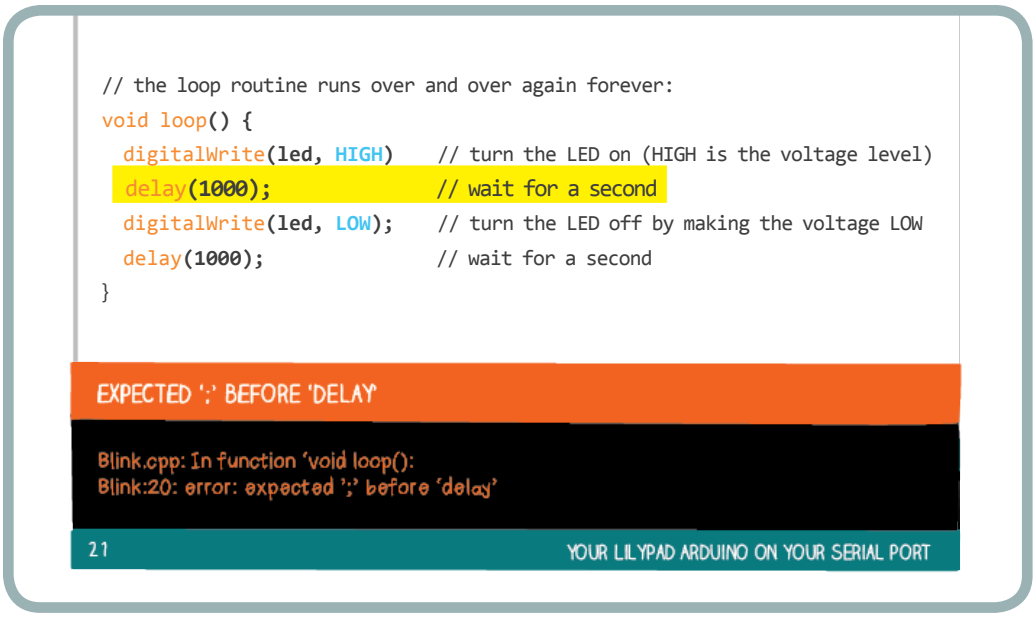


This means that the LilyPad Arduino board is not responding to your computer's attempts to communicate with it. Possible causes of this error are:

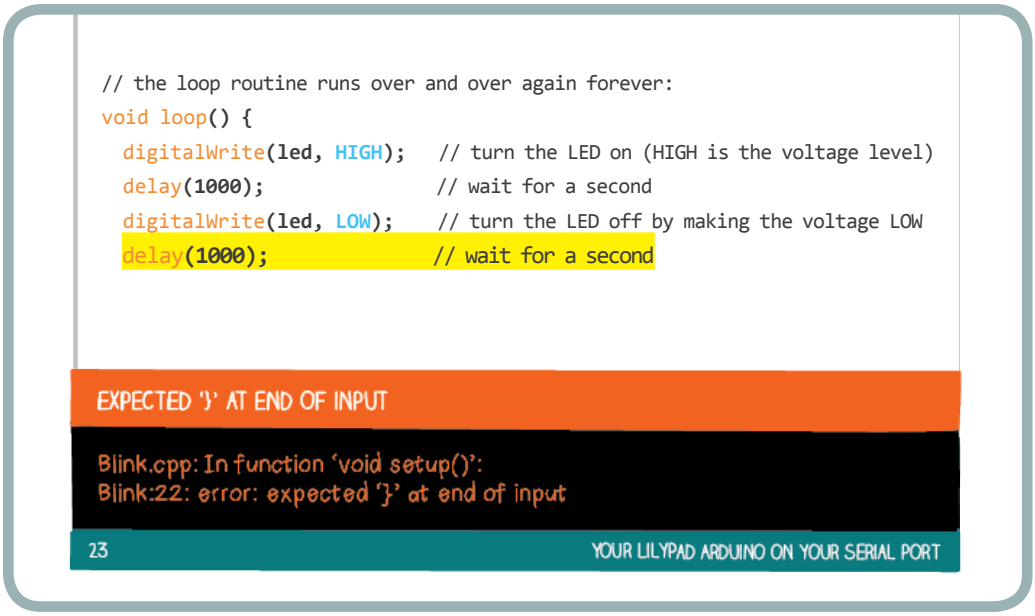
- You did not select the correct serial port or board. Return to the setup section of this tutorial.
- Your drivers were not properly installed. Return to the setup section of this tutorial.
- Parts are loose or disconnected. Make sure that that the LilyPad, FTDI board, USB cable, and computer are all connected.
- Your USB cable is not working. Try another USB cable.

If you are still having trouble, see the troubleshooting section of the Arduino website: http://arduino.cc/en/Guide/Troubleshooting.

SOFTWARE ERROR: MISSING SEMICOLONS OR CURLY BRACKETS

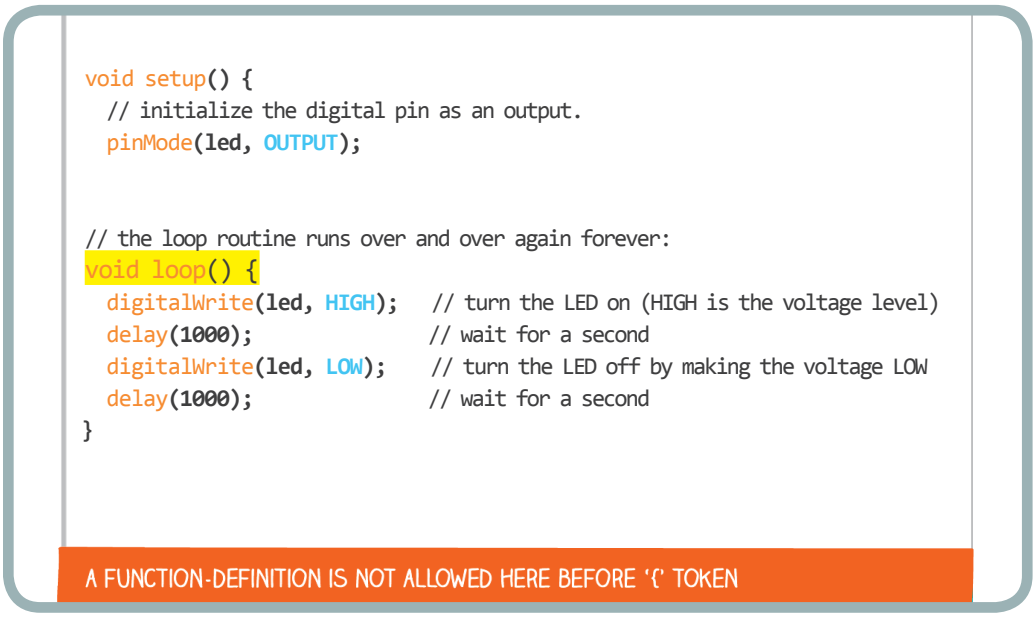


A red message that reads `expected ';' before...` means that you forgot a semicolon at the end of a statement in your code. Arduino tells you that it expects a semicolon, and it also highlights the line directly after the line where your semicolon was forgotten. In the example above, notice the text on the black background that says `Blink:20: error: expected ';' before 'delay'`. This means that the error occurred on line 20, right before the line `delay(1000);`



You will receive a similar error if your program is missing curly brackets or parentheses. In the above example, the closing curly bracket of **loop** is missing. Arduino finds the error on the last line of your program (line 22).

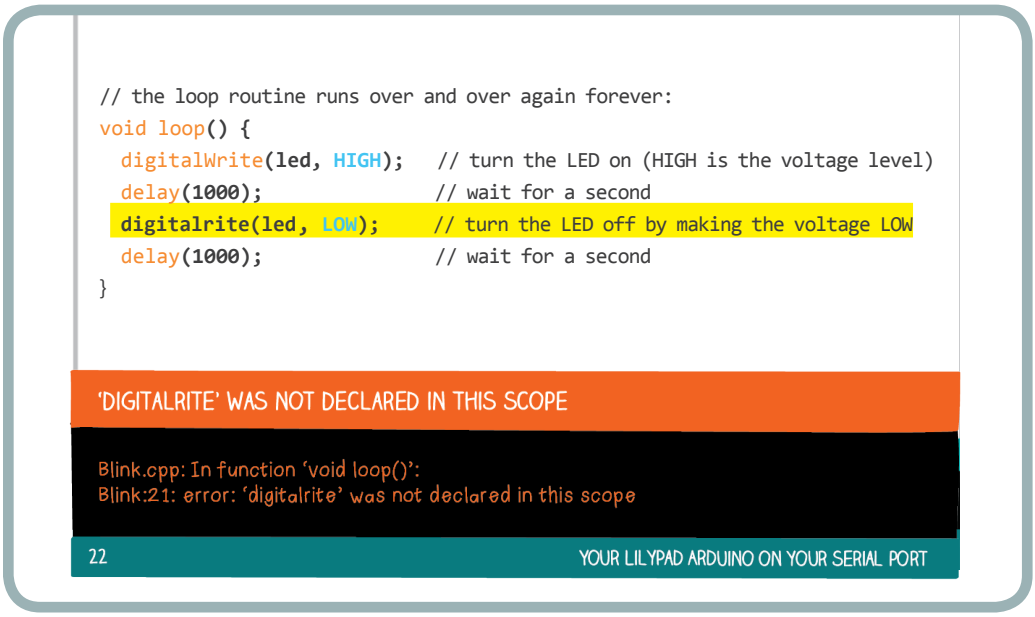
As you have already seen, sometimes error messages for missing brackets or parentheses will be much less helpful.



In the example above, we have left off the closing bracket of the **setup**. The error reads: `a function-definition is not allowed here before '{' token` — not a very understandable or useful message.

The moral of this example is that even if you do not understand what an error message means, the first thing you should check for in your code is missing semicolons, parentheses, and curly brackets.

SOFTWARE ERRORS: MISSPELLINGS AND CAPITALIZATION ERRORS



Sometimes, you may misspell the name of a procedure or variable. Above, we misspelled `digitalWrite` as `digitalrite`. Arduino gives another cryptic and confusing error: `digitalrite was not declared in this scope`. This means that Arduino doesn't know of any command called `digitalrite` — it only knows about the `digitalWrite` command.

Whenever Arduino says that something "was not declared in this scope," it means that you are using a variable or a command that Arduino does not recognize. Make sure that you spell everything correctly.

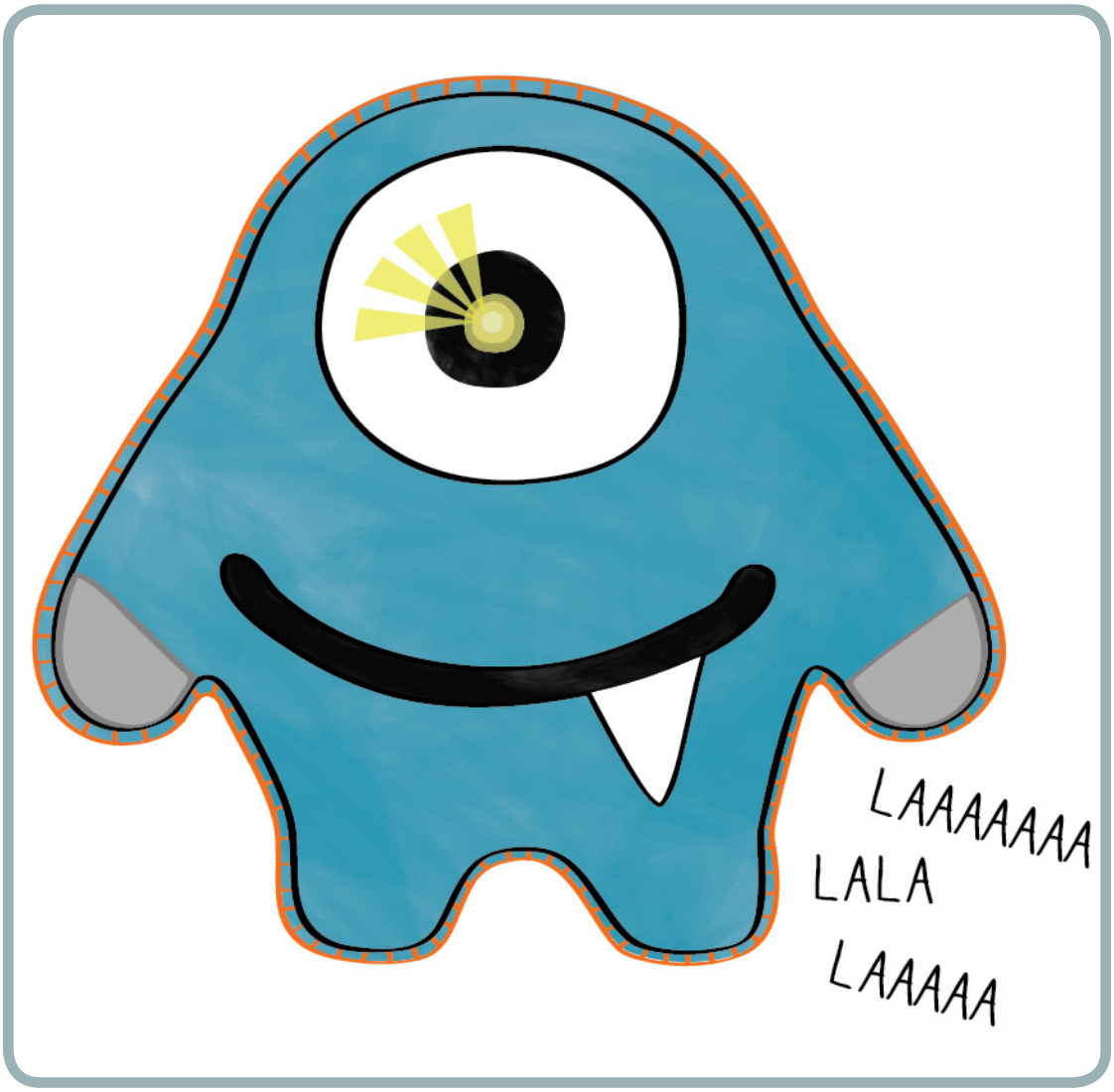
Also note how the correctly spelled `digitalWrite` shows up orange in the Arduino software. Misspelled text will not turn orange. Arduino uses color coding to help you avoid misspellings.

All the code that you are writing is also case sensitive. This means that while Arduino understands `digitalWrite`, it does not understand `digitalwrite`. Similarly, while it understands `LOW`, it does not understand `low`.

ADDITIONAL HELP

If you're having problems that aren't addressed here, check out the troubleshooting section of the Arduino website: http://arduino.cc/en/Guide/Troubleshooting or post a question on the Arduino forum: http://arduino.cc/forum/.

Interactive Stuffed Monster



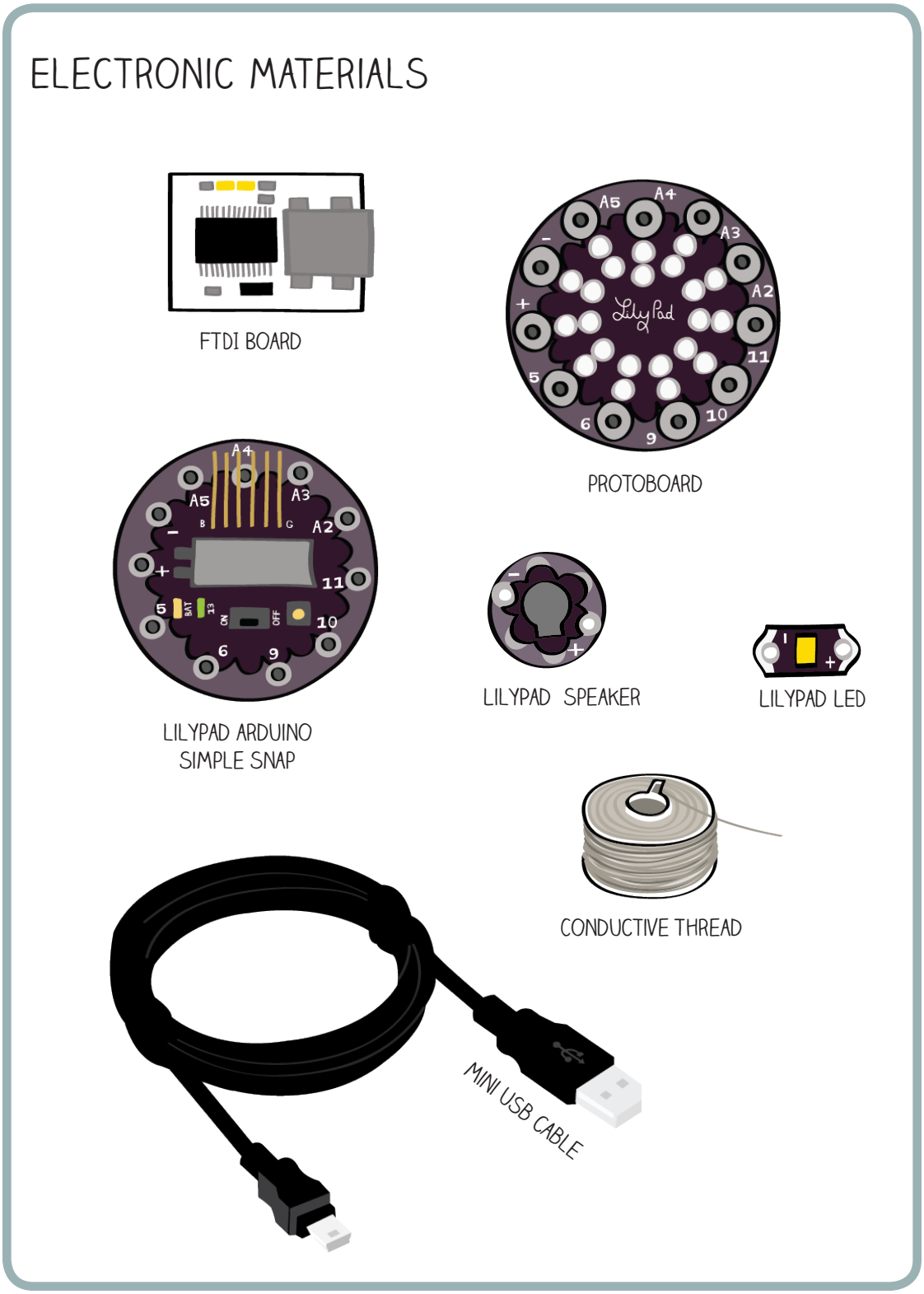
Now that you know how to sew, put circuits together, and program a LilyPad, you can combine those skills to create a soft interactive project. This tutorial will show you how to design and build a singing and glowing stuffed monster that responds to touch.

TIME REQUIRED: 10–15 hours

Collect Your Tools and Materials

This project uses a LilyPad Arduino SimpleSnap, an aluminum foil sensor to control an LED, and a speaker. You'll need fabric and a basic set of sewing and sketching supplies in addition to your electronics.

ELECTRONIC MATERIALS AND TOOLS



CRAFT MATERIALS AND TOOLS

NOTE ABOUT MATERIALS

1. Consider using fabric glue and fading fabric markers, and have toothpicks on hand for spreading glue.
2. Additional materials and tools necessary for the aluminum-foil touch sensor (an optional part of this project) include:
    - Aluminum foil (not the nonstick variety)
    - At least 1 yard of Heat-n-Bond ™ ultra hold iron-on adhesive sheet (*not* tape)
    - Iron and ironing board

Design Your Monster

YOUR ELECTRONICS

Before you start working on your project, it's important to know a bit more about the electronics you will be working with. After working through the other lessons, you should be familiar with the LilyPad Arduino SimpleSnap (or "LilyPad") and the pieces you need to program it. You've also worked with LEDs before, but the LilyPad SimpleSnap Protoboard ("protoboard") and the LilyPad Speaker are new. Refer to the Practical Guide to see all the components.

The protoboard is the board with a ring of "male" snaps around its outer edge and holes in its center. The protoboard snaps onto the LilyPad Arduino SimpleSnap. Try snapping the two boards together. Notice that they'll only snap together in one orientation. Try taking them apart — this can be tricky! If you're having trouble, wedge something stiff, like a pair of scissors or a dull side of a butter knife, in between the two boards and gently pry them apart.

In this project you'll sew the protoboard to your monster and snap the LilyPad to it. This will enable you to reuse your LilyPad in other projects. Notice that the protoboard has labels on each of its snaps that correspond to the labels on the LilyPad.

The LilyPad Speaker, as you've probably guessed, can make sounds when you send it the right kind of signal. It's also called the LilyPad Buzzer. You may see "Buzzer" instead of "Speaker" on your packaging. If so, don't worry — you still have the right part. This tutorial will call the LilyPad Speaker simply "the speaker."

BASIC DESIGN

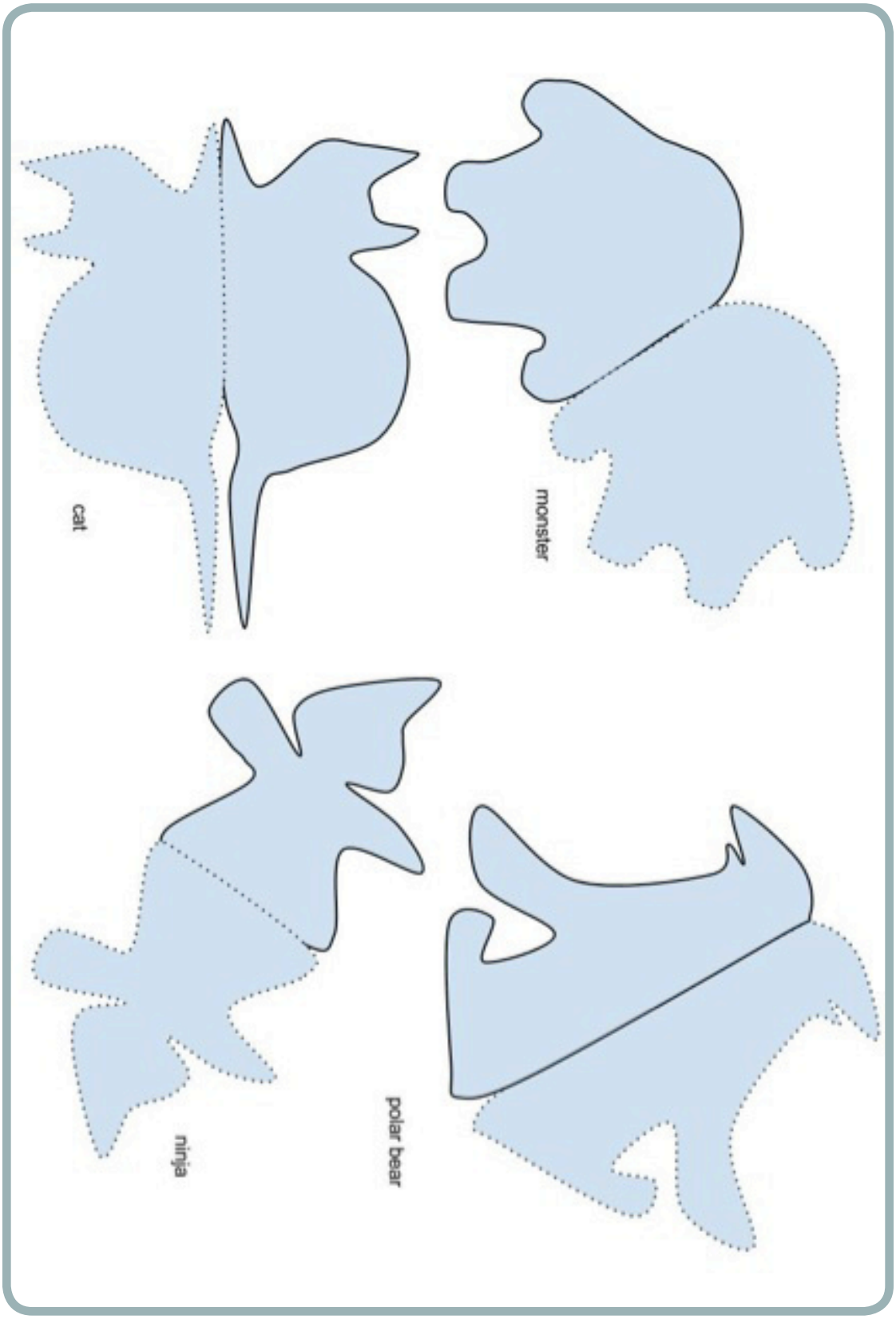
Begin by designing a shape for your monster and drawing it on a piece of stiff paper or thin cardboard. Two things to note:

SIZE. Your design should fit easily on a sheet of 8.5" x 11" letter-sized paper. And, since circuitry needs to fit, make sure that your design is no smaller than half a sheet.

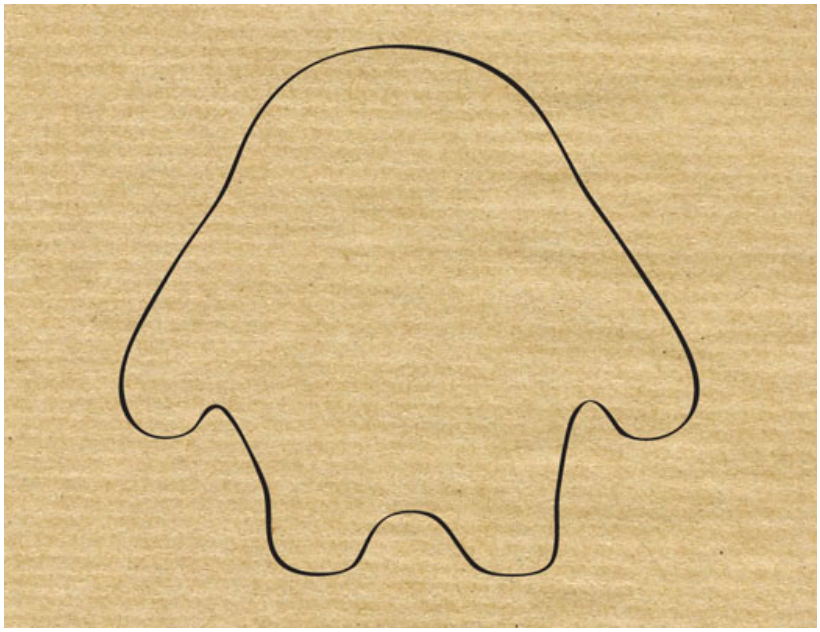
SHAPE. Because the conductive thread needs to follow traces from one side of the monster to the other, you need one fairly wide, continuous edge for the front and back pieces.

The patterns on the next page are good basic shapes for your first monster. Consider making a bigger version of one of these.

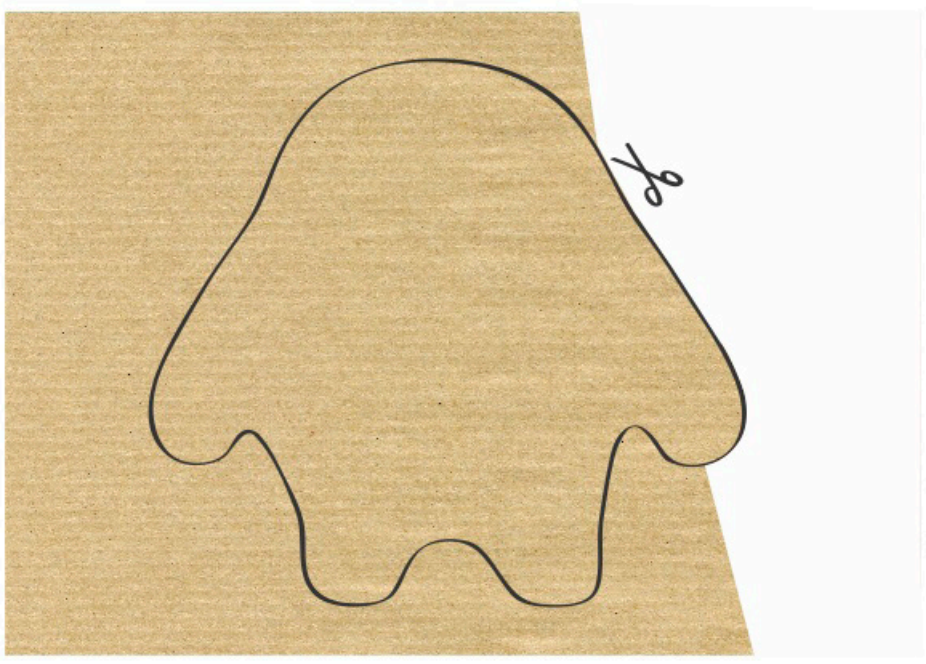
PATTERNS



Draw your design on stiff paper.

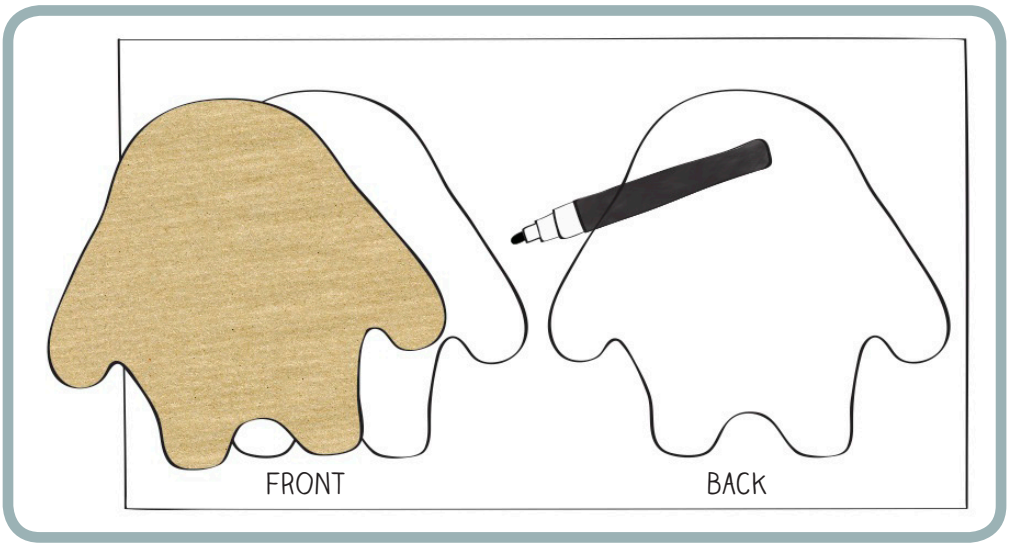


Cut out your shape to use as a template.

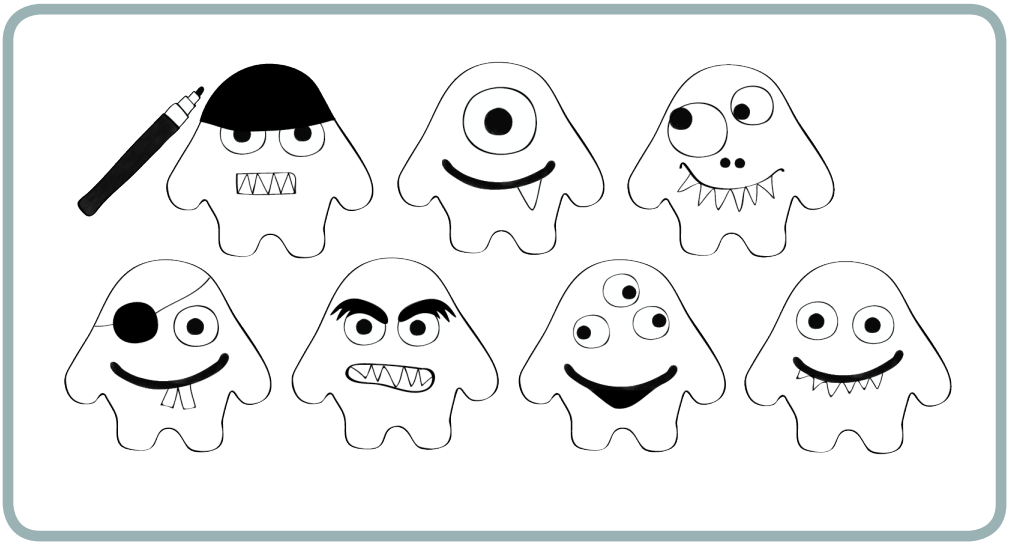


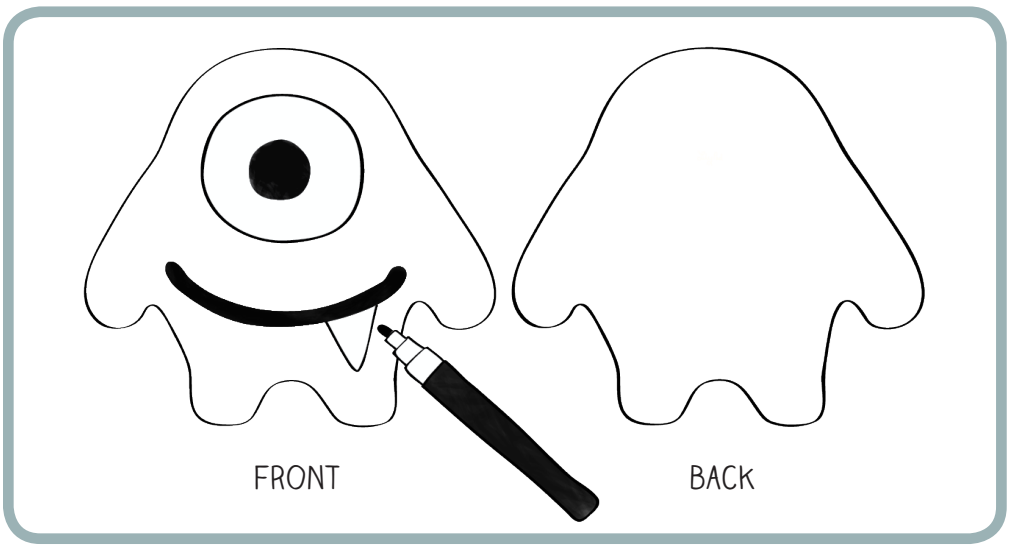
On blank paper (ideally, size 11" x 14"), use the template to trace out two copies of your monster — one for the front and a "mirror" copy for the back. You should have one longer edge that lines up when the pieces are side-by-side, as shown in the earlier picture. Label one as the front of the monster and one as the back.

NOTE: As you design and build your monster, keep careful track of the front and back pieces, as well as the inside and outside of your monster. All the design drawings here show the *outside* of the monster.



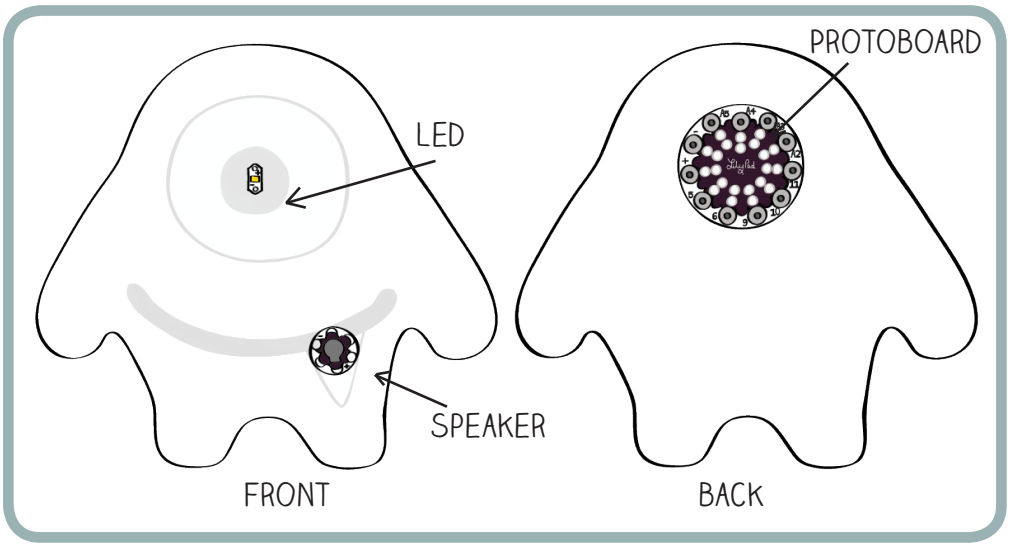
Give your monster some personality. What will its eyes and mouth look like? Will it have claws? Toenails? With pencil, sketch these details on your design.



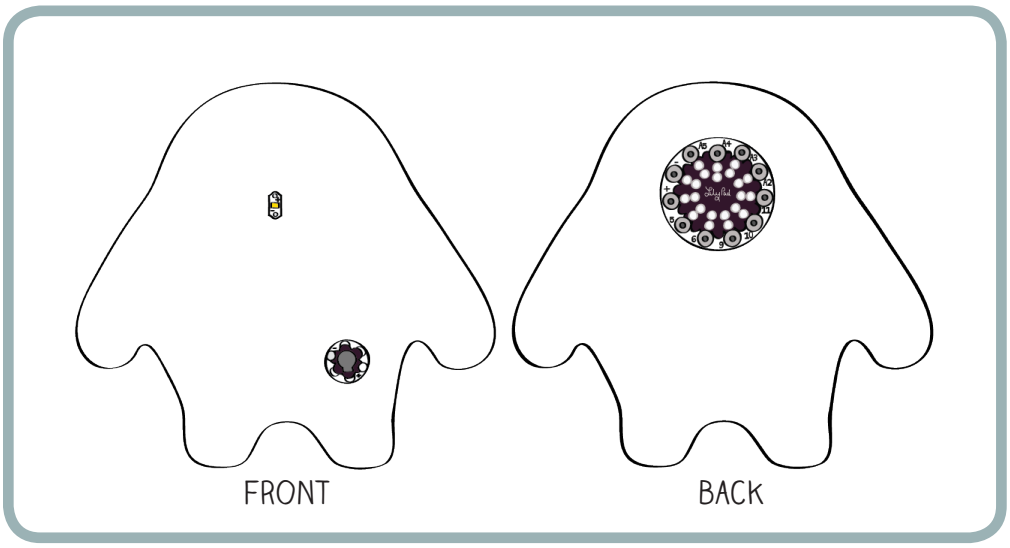


CIRCUIT DESIGN

Once you've created your monster character, decide where you want to put your protoboard (the LilyPad will snap onto this board), LED, and speaker. Now, add these components to your design, keeping all these pieces on the outside of your monster. Here, the protoboard is on the back of the monster and the LED and speaker are on the front of the monster. Nothing should be inside the monster.

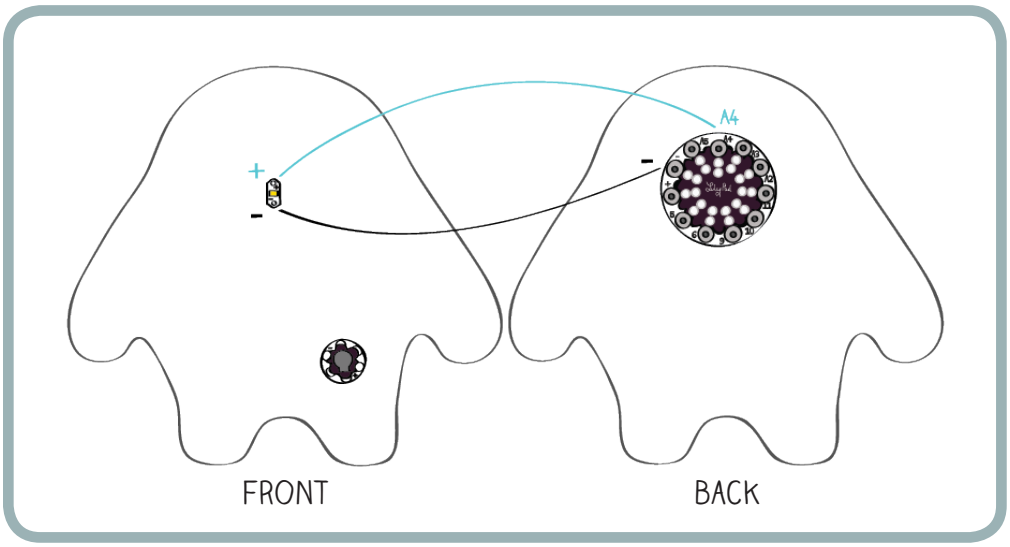


It's now useful to remove the character elements from your sketch so you can focus on the circuit design. You can make your **circuit diagram** on your character sketch or make a new drawing for the electrical elements. Do whatever you think will be most helpful.

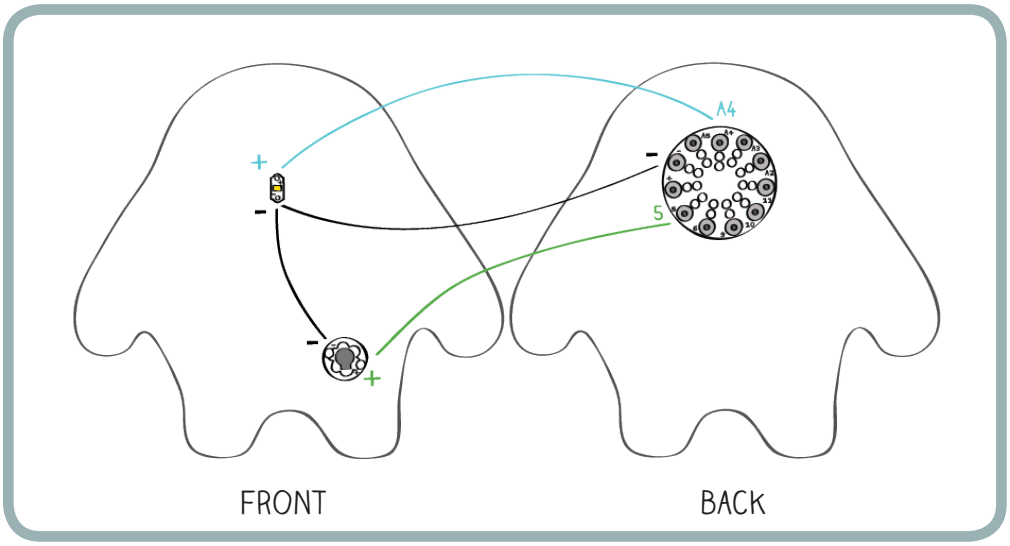


Now that you know where the components will go, you need to figure out how to connect them. Start with the LED. The (–) tab on the LED should attach to the (–) tab on the protoboard. The (+) tab on the LED should attach to one of the numbered tabs on the protoboard tab (A4 for the monster shown here.) Remember, each of these connections is called a "trace" in the language of electronics.

Sketch the (–) trace in black and the other trace in another color (such as blue, as shown here).



Next, plan the speaker connections. The (–) tab on the speaker needs to be attached to the (–) tab on the LED (shown in black, below). The (+) tab on the speaker should be attached to another tab on the protoboard. Any tab except (–) or (+) will do. Here, the green trace shows the (+) tab of the speaker attached to pin 5.



When planning a circuit, keep these general guidelines in mind:

- Don't let traces from different pins or tabs cross each other or get too close to each other. It's OK for two of the same electrical traces to touch each other — like the (–) traces for the LED and speaker. But, as described in the bookmark tutorial, two different traces (+) and (–) touching each other creates a short circuit. In the monster, short circuits will cause the LED and speaker to malfunction and, in the worst case, may damage the LilyPad or its battery.
- Make sure you always plan out your circuit before you start building. Without careful planning, you will end up creating short circuits, sewing unnecessarily long traces, and having to take out and resew stitches during construction.

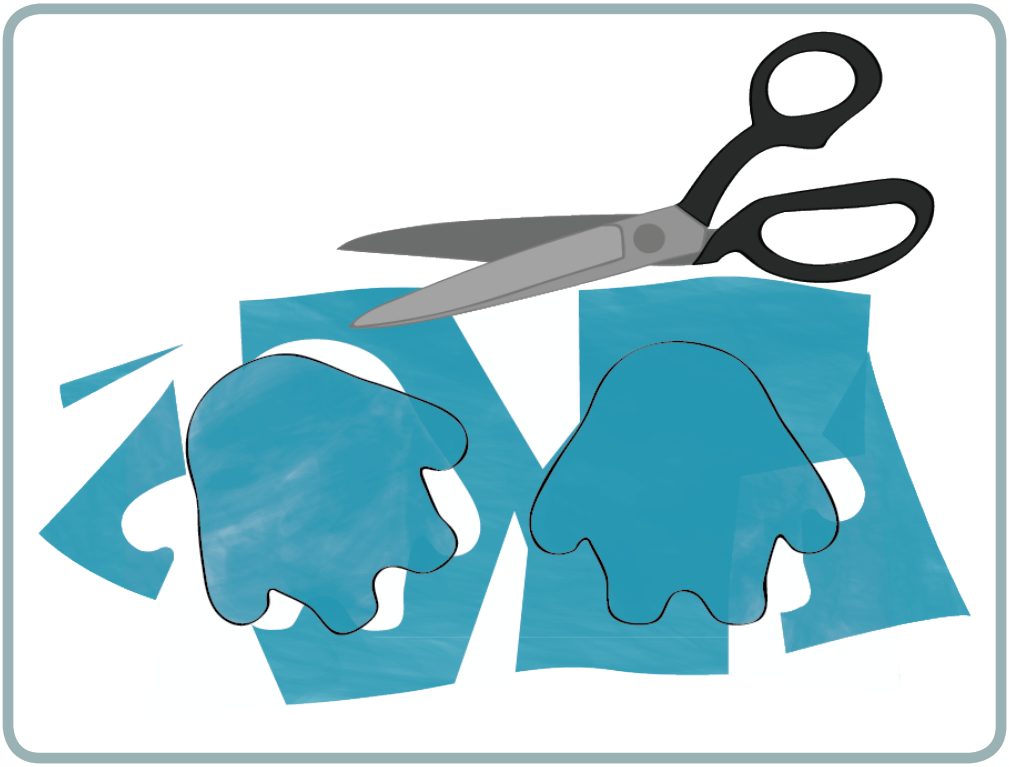
NOTE: You may want to add additional electronic elements (such as sensors) to your monster later on. (See: "Add a sensor to your monster" on page 105.) In your design, make sure to leave room for any additions you might like to add later.

Now, use your colored pencils or markers to draw the traces on your circuit diagram, using black for (–) connections, and different colors for the (+) connections for your LED and speaker.

Begin Building

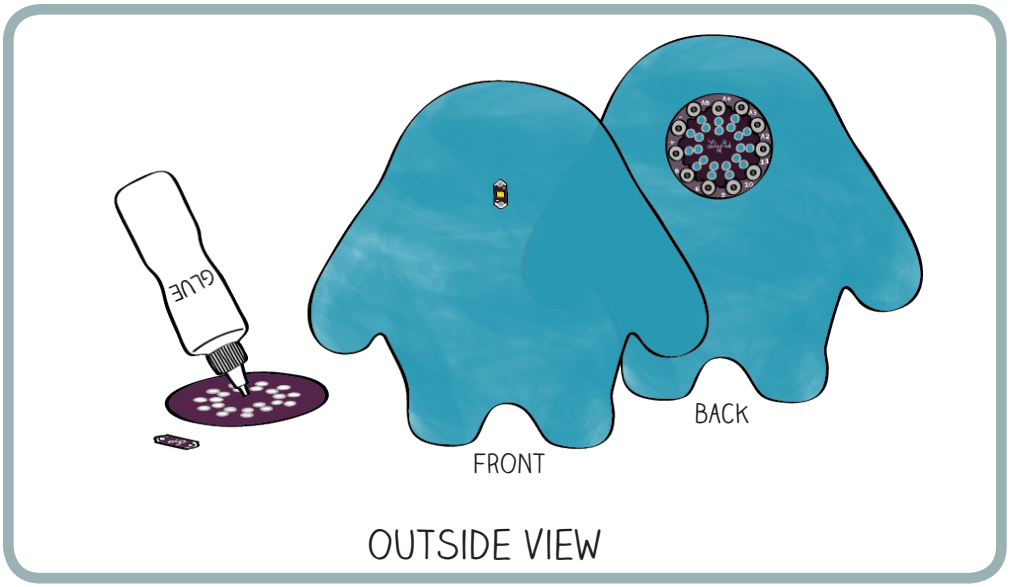
CUT OUT YOUR FABRIC

Trace your template onto fabric with a piece of chalk or marker. Make two copies of the monster shape, and cut out both pieces — one for the front and one for the back of the monster.



ATTACH YOUR COMPONENTS

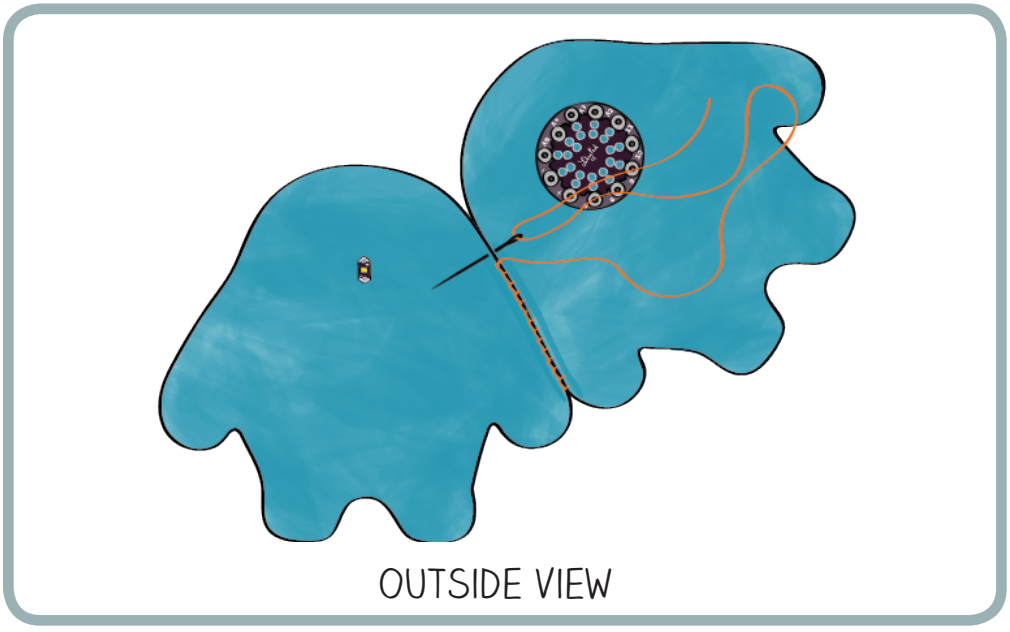
Using your design sketch as a guide, glue the protoboard and LED onto the fabric by putting a dab of glue on the back side of each component. Use only as much glue as you need to attach the pieces, and be careful not to fill in any holes, since you'll need to stitch through these soon. A toothpick may make the job easier. Let the glue set for at least 5 minutes before proceeding to the next step.



SEW ONE SIDE OF YOUR MONSTER TOGETHER

Using (nonconductive) embroidery thread, sew one side of your monster together. This will make it easy for you to stitch from the front to the back of the monster with the conductive thread. You can use a running stitch to sew the two sides together. Alternately, you can use what's called a **whip stitch** — looping your thread around the outside of the two pieces of fabric. The images here show a whip stitch edge. These stitches will appear on the outside of the monster as part of the monster's decoration, so select a fun color.

Make sure that you keep all components on the outside of the monster during this step. Your monster should look something like this when you are done:

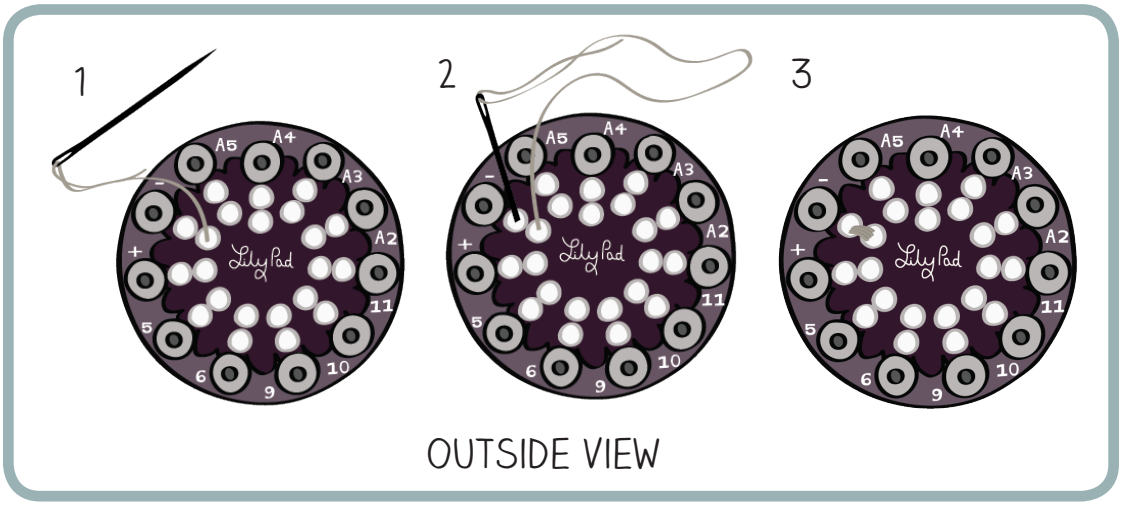


DRAW CONNECTIONS

Using chalk or a pencil, draw the electrical connections between the LED and protoboard on the outside of the monster. These will cross the seam you just sewed. Note: If you wish to hide the circuitry, draw these connections on the inside of your monster, too.

SEW YOUR LED

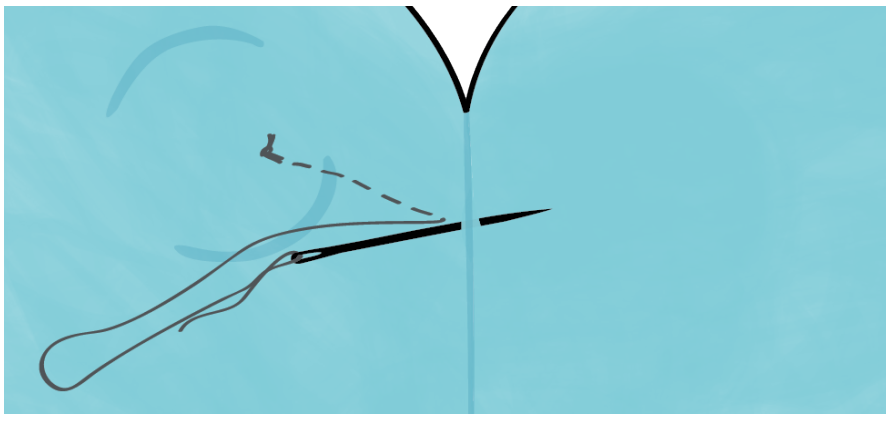
Now you're ready to begin stitching the LED to the protoboard. Measure out 2–3 feet of conductive thread, and thread your needle. Tie a knot on the underside of your fabric near the (–) holes on the protoboard using the knot-tying technique described in the bookmark tutorial. Sew tightly through the pair of (–) holes at least three times to attach the protoboard to the fabric. This will make a solid electrical connection between the protoboard and your thread. Don't let the conductive thread touch any of the other holes on the protoboard. This can create a short circuit later.



OUTSIDE VIEW

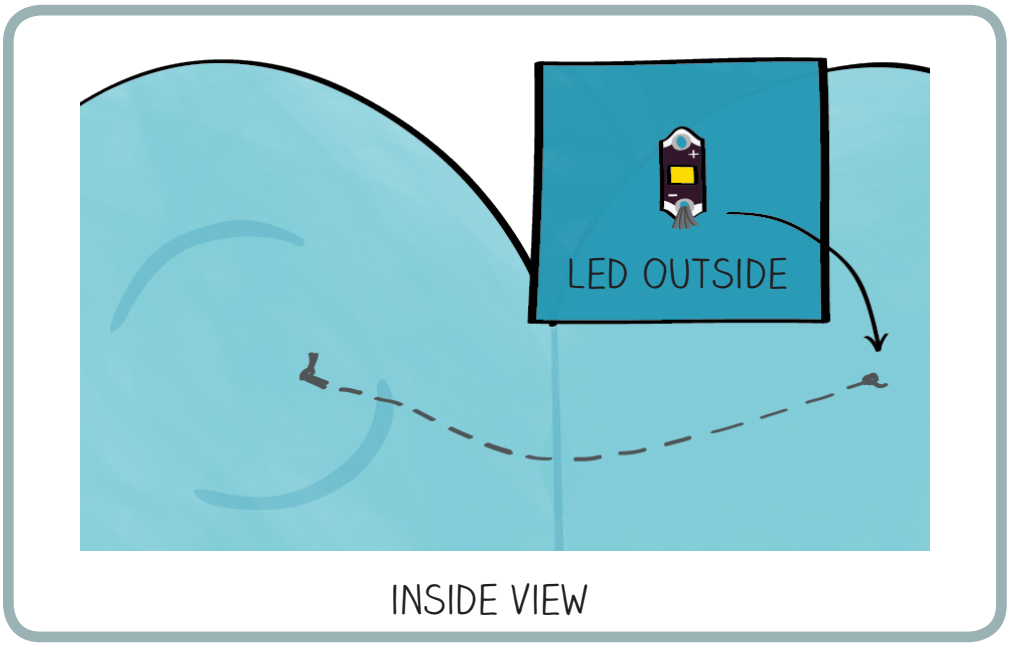
Following the connections you drew on the fabric, sew a running stitch from the protoboard (and across the embroidery thread seam you just stitched) to the (–) side of the LED. The running stitch is described in the bookmark tutorial.

If you want, you can shallowly sew only on the inside of the monster so stitches do not show up on the monster's outside. The drawings in this tutorial show all conductive stitches stitched in this way.

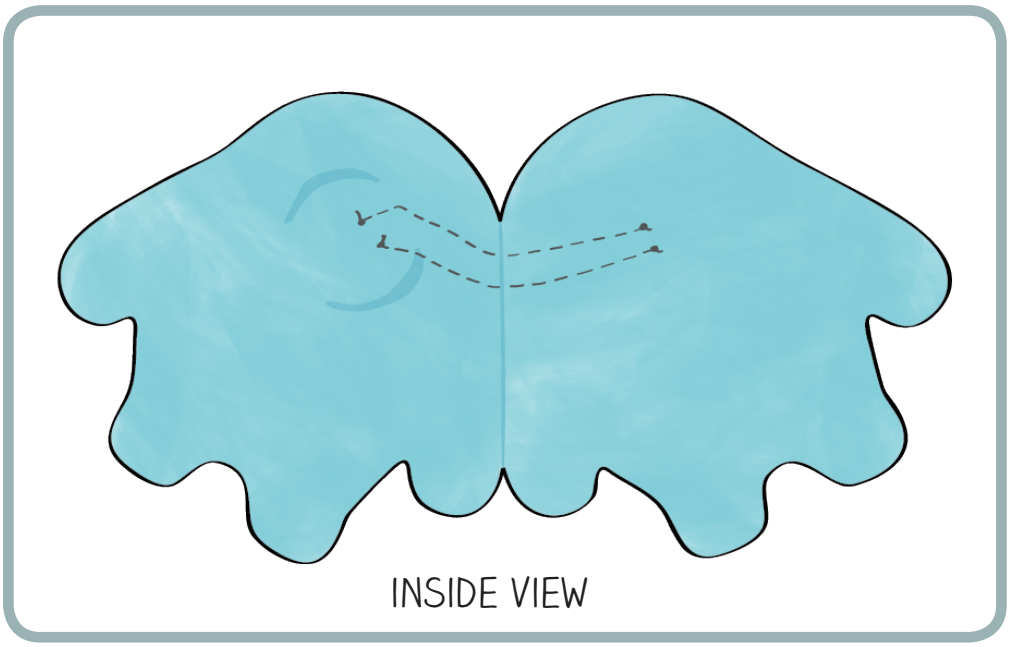


INSIDE VIEW

When you reach the LED, sew at least three loops through the hole on the (–) side of the LED. Then, pull the thread tight and tie a knot on the inside of the monster. Cut the tails of the knots on both ends of your trace to about ¼ inch (approximately 6 mm) long. Put a dab of glue on both to keep them from unraveling.

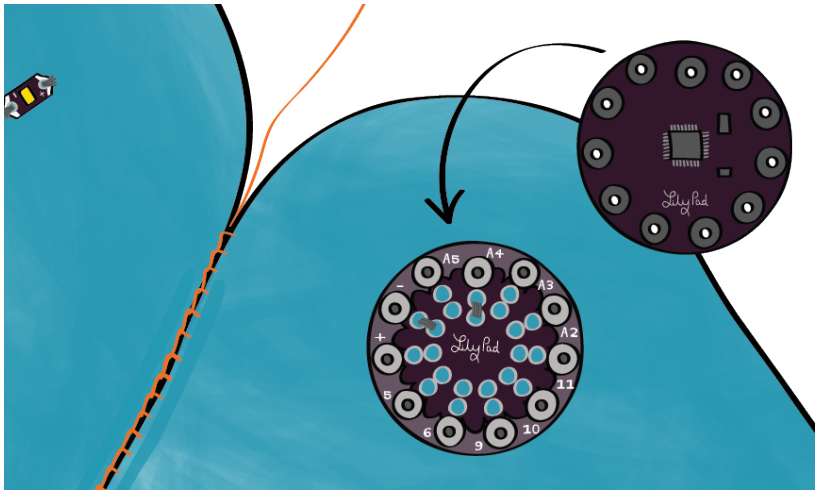


Now sew the (+) side of your LED to the correct pin on the protoboard (A4, or the pin that you chose). Here's an inside view of the monster with both the (+) and (–) LED connections stitched in:



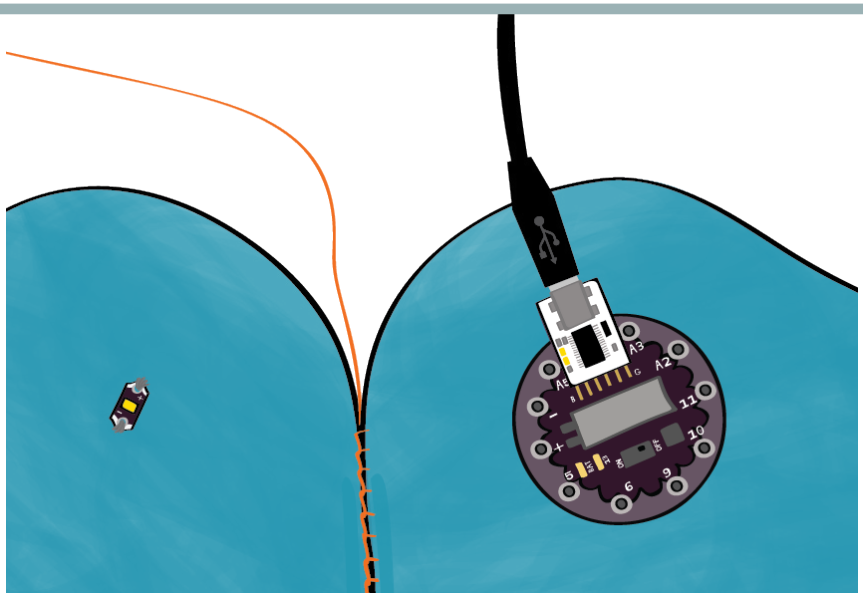
Make Your Monster Blink

Snap the LilyPad onto the protoboard on your monster.



OUTSIDE VIEW

Get out your computer, open up the Arduino software, and attach your LilyPad to your computer via the USB cable and the FTDI board.

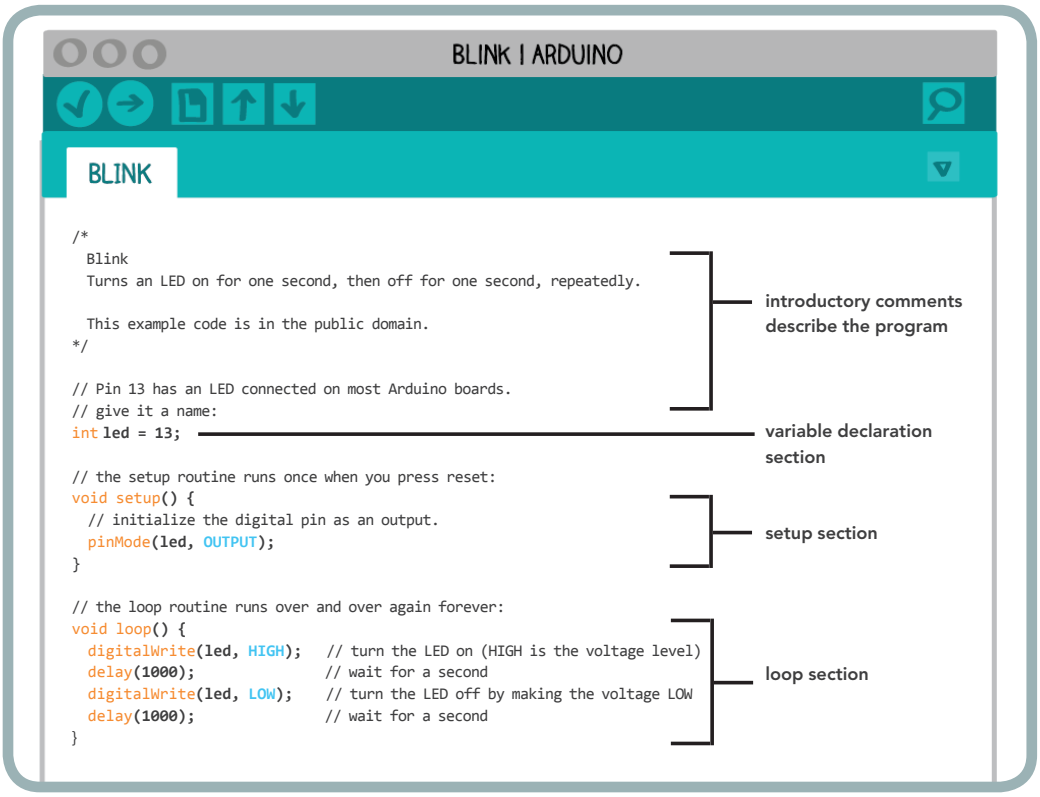


In the Arduino software, open up the "Blink" example by clicking on the upward-pointing arrow in the Toolbar and selecting 01.Basics ➔ Blink.

This is a good time to recall the main elements of programming in "C."

In the Arduino Development Environment (ADE) you find a commenting area, then three main parts: a "variable declaration" section, a "setup" section, and a "loop" section. When your program executes, it will first define your variables, then execute the setup section once, and then execute the loop section over and over.

Below is an example program in the ADE with each section annotated. Refer back to it when you need a reminder about where to write code.



Now, to make it easier to follow along with the examples that follow, delete all the comments at the beginning of your program — all before the `int led = 13;` statement — so that your program looks like this:

```
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
pinMode(led, OUTPUT);
}

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

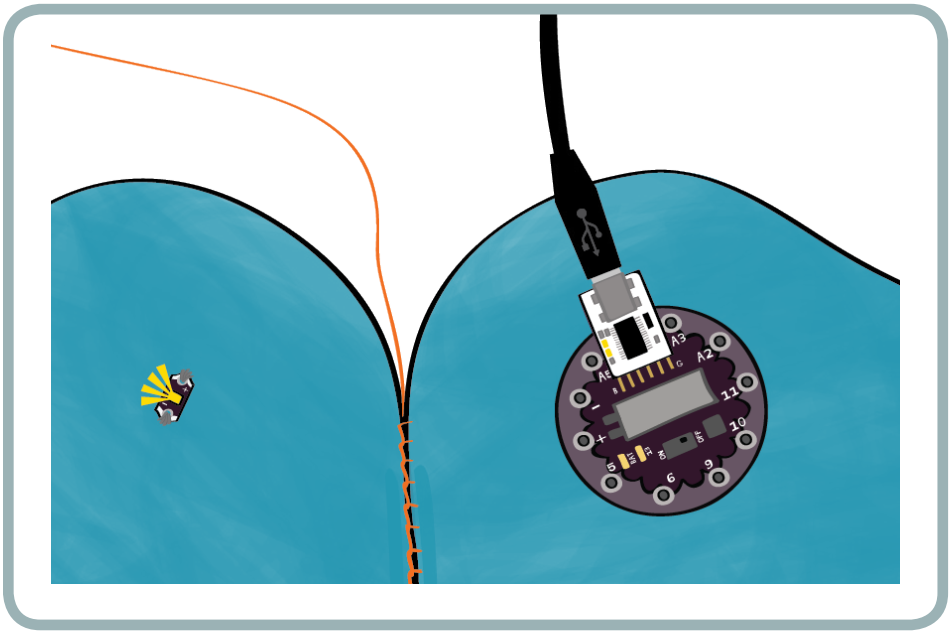
Compile and upload this example to your LilyPad. A green LED on the LilyPad should begin to blink on and off.

You want to use this program to control the LED that you've stitched into your monster, not the LED on the LilyPad. To do this, you'll need to make a small edit. Notice that at the beginning of the code, the variable `led` is set to the value 13 with the line `int led = 13;`. This tells the LilyPad which pin on the LilyPad the LED is attached to.

When the variable `led` is set to 13, the code in the rest of the program will control the green LED that is on the LilyPad board. To get your monster LED to blink, you should change this line of code to match your design. A4 was used there since that was the pin that was sewn to in our design. Check your pin number before changing the code.

```
int led = A4;
```

Make this change to the code, and compile and upload it to your LilyPad. The LED you've sewn to your monster should begin to blink.



MAKING SENSE OF THE CODE: PINMODE

Take a moment to explore the code more closely, beginning with the code in the setup section. (To remind yourself of the different sections, refer back to the annotated view of the Arduino Development Environment (ADE) on page 83.

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

The line `pinMode(led, OUTPUT);` tells the LilyPad that the led pin will be used to control an output. Outputs are things like lights, motors, and speakers. Inputs are things like switches and sensors.

Whenever you add a component to your design, you need to include a `pinMode` statement in the setup part of your program to tell the LilyPad whether the component is an input or an output device. (You'll do this shortly, when you connect your speaker to your protoboard and begin programming it.)

`pinMode` is a **procedure**. The procedure `pinMode` takes two inputs. One specifies the pin number that is being controlled, and one specifies whether that pin will be an input or an output:

Procedure Name	Input 1	Input 2
`pinMode(`	`pin number`	`INPUT or OUTPUT);`

For the monster, the pin number is the variable led (which was set to pin A4 at the beginning of the program). Since you're controlling an LED light with this pin, it should be set to be an output. Your LED won't turn on unless the line `pinMode(led, OUTPUT);` is included in the setup part of your program.

MAKING SENSE OF THE CODE: DIGITALWRITE, HIGH, AND LOW

Examine the statements in the loop section of your program. Remember from the programming tutorial that the loop section is where the main action of your program takes place.

```
void loop() {
digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
delay(1000);               // wait for a second
digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
delay(1000);               // wait for a second
}
```

There are four statements in the loop section. The first one is:

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

`digitalWrite` is another procedure. It is what turns your LED on and off. It takes two inputs, one that specifies the pin that is being controlled and one that tells the LilyPad what to do with the pin. The pin is set to either `HIGH` or `LOW`.

Procedure Name	Input 1	Input 2
`digitalWrite(`	`pin number`	`HIGH or LOW);`

What do `HIGH` and `LOW` mean? These are code words that the Arduino language uses to talk about electricity. When the LilyPad encounters a `digitalWrite(pin, HIGH);` statement, it sets the pin to (+). When the LilyPad encounters the `digitalWrite(pin, LOW);` statement, it sets the pin to (–).

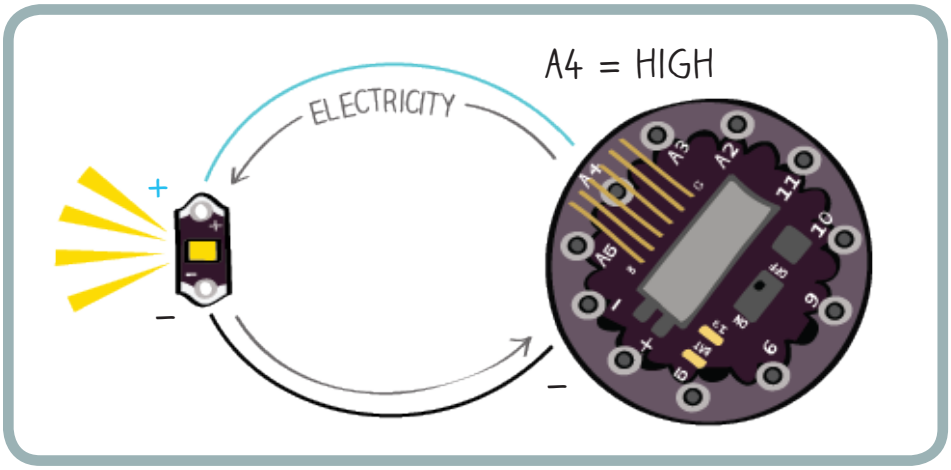
These statements are what allow you to control electrical signals with code. It's worth stopping to think about how powerful this is. The LilyPad and the Arduino turn text that you write on your computer into behavior that happens in the physical world.

Here's a summary of how the code that you write relates to electric power:

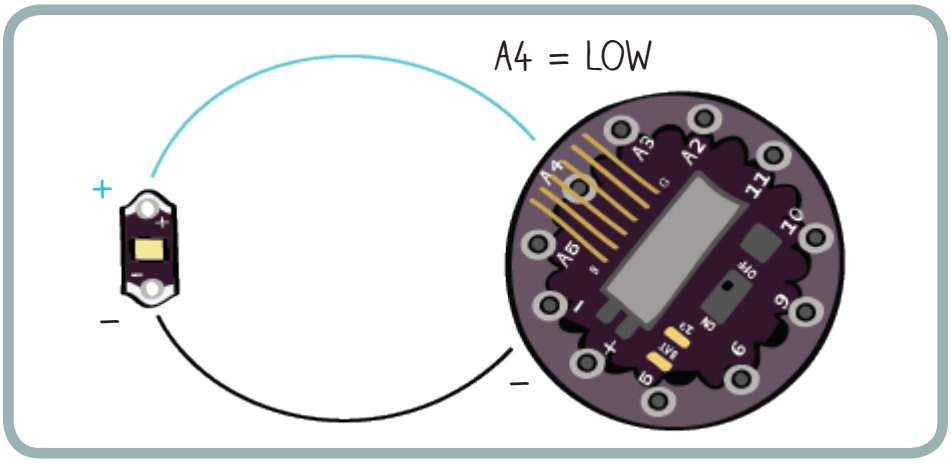
Code Name	Electrical Value		
	Symbol	Voltage	Other Name
`HIGH`	+	3.7 volts[1]	power
`LOW`	–	0 volts	ground

To understand why the `digitalWrite(pin, HIGH);` statement turns the LED on the monster on and the `digitalWrite(pin, LOW);` statement turns the LED off, look at your circuit design sketch.

When you set the led pin (A4) `HIGH` with the statement `digitalWrite(led, HIGH);`, pin A4 gets set to (+). Since electricity flows from (+) to (–), current runs through the LED, lighting it up:



When you set the led pin (A4) `LOW` with the statement `digitalWrite(led, LOW);`, pin A4 gets set to (–). Since electricity does not flow from (–) to (–), the flow of electricity stops:



[1] These voltages assume that the LilyPad Simple Snap is being powered from its 3.7 volt battery.

MAKING SENSE OF THE CODE: DELAY

There's one statement in your program that you haven't yet explored completely, though you experimented with it in the programming tutorial: `delay(1000);`. `delay` is also a procedure. It's a simple one that takes only one input: an amount of time in milliseconds (1/1000 second).

Procedure Name	Input 1
`delay(`	`time in milliseconds);`

The delay statement tells the LilyPad to do nothing for the specified amount of time.

EXPERIMENT

Now that you're more familiar with the code, change the program to get a blinking behavior that you like for your monster. Can you get your LED to flicker like a candle? Or thump like a heartbeat? You might also want to try using one of the LED behaviors you explored in the programming tutorial.

When you're finished, your Arduino window should look something like this:

CREATE YOUR OWN PROCEDURE

So far, you've used built-in procedures such as `digitalWrite` and `delay`. Now, you're going to write your own procedure. This new procedure will make your monster blink in the custom pattern you just created. A procedure is a chunk of code that is given its own special name. You're going to create a procedure called `blinkPattern` that will store the new code you just wrote.

Tip: If steps in different sections in the environment confuse you, look ahead to a complete example program on page 89.

To create a procedure called `blinkPattern`, add the following code to the very end of your program (after the closing bracket of the `loop` section):

```
void loop() {
digitalWrite(led, HIGH);
delay(100);
digitalWrite(led, LOW);
delay(100);
digitalWrite(led, HIGH);
delay(500);
digitalWrite(led, LOW);
delay(500);
}

void blinkPattern() {

}
```

NOTE: The name of the procedure doesn't matter to the compiler, so if you'd like to pick a different name for your procedure, you can. Just make sure that from here on out, you replace `blinkPattern` with the name of your procedure everywhere that `blinkPattern` appears in the example code.

This is the basic template for a procedure definition. The definition begins with the word `void` followed by the name of the procedure. Any inputs taken by the procedure are listed in the parentheses to the right of the procedure's name. Since `blinkPattern` doesn't have any inputs, the parentheses are blank. Two curly brackets surround the "body" — the main part — of the procedure.

Now, cut the code from the loop that defines your blink pattern and paste it into the body of your new `blinkPattern` procedure:

```
void blinkPattern () {
  digitalWrite(led, HIGH);
  delay(100);
  digitalWrite(led, LOW);
  delay(100);
  digitalWrite(led, HIGH);
  delay(500);
  digitalWrite(led, LOW);
  delay(500);
}
```

YOUR BLINK CODE HERE

Try compiling and uploading this code to make sure you haven't introduced any errors. Your LED should stop blinking.

To actually use the procedure, you need to use its name, `blinkPattern`, somewhere else in the program — in programming slang, you need to "call" it. Edit your code so that the `loop` section looks like this:

```
void loop() {
blinkPattern();
}
```

That is, replace all your blink code with one "call" to `blinkPattern`. Compile and upload this code. Now that you've called the procedure in `loop`, your LED should start blinking again!

Your entire program should now look something like this, with your own custom code in the body of the `blinkPattern` procedure:

```
Int led = A4;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
blinkPattern();
}

void blinkPattern() {
digitalWrite(led, HIGH);
delay(100);
digitalWrite(led, LOW);
delay(100);
digitalWrite(led, HIGH);
delay(500);
digitalWrite(led, LOW);
delay(500);
}
```

Notice that you use the same format to call the procedure `blinkPattern` that you've used to call other procedures (for example, `delay` and `digitalWrite`). Since `blinkPattern` doesn't have any inputs, the parentheses after its name are empty:

Procedure Name	No Input
`blinkPattern(`	`);`

You'll experience how useful your procedure is in a moment, when your code starts to get more complex. For now, can you think of reasons why the ability to write procedures might be powerful? How might you use procedures to avoid repeating lines of code? Why might they make your programs easier to read? Why might they make your programs shorter?

SAVE YOUR CODE

Save your code by clicking on the downward pointing arrow in the Toolbar. Click "OK" on the popup window that may appear, and choose a good name (like "monster") for your file. To finish the process, click on the "Save" button in the next window that appears.

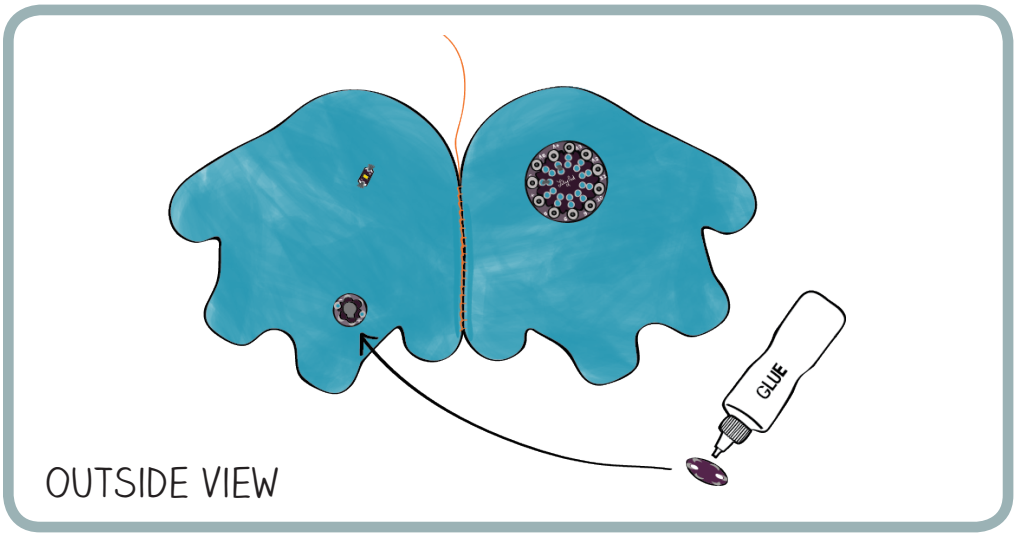
TROUBLESHOOTING

If your LED doesn't turn on:

1. When you uploaded the program, did it compile and upload successfully, or were there red error messages? If you received error messages, you need to correct a problem in your code. Carefully compare your code to the example, looking especially for missing parentheses, curly brackets, and semicolons. Fix any problems you see, and try uploading again.
2. Check your code. Did you set the `led` variable to the correct pin number? The correct pin number is the one that you sewed the (+) side of your LED to with the conductive thread.
3. Did you insert your `blinkPattern` procedure in the right place in your code? Is it right after the closing bracket of the `loop` section?
4. If you are doing this project with others, discuss the challenges you are having. Don't just copy their process, though. Discuss why the problems occurred, and relate this back to the programming concepts you are learning.
5. Look at where you tied your conductive thread knots. Are any stray knot ends touching each other where they shouldn't be? Make sure that all knots are neatly trimmed and sealed with glue or nail polish.
6. Check the two traces of conductive thread you sewed. Do they cross or touch where they shouldn't? Do they go all the way from the protoboard to the LED? Are there tight loops of thread at every connection on the protoboard and LED?

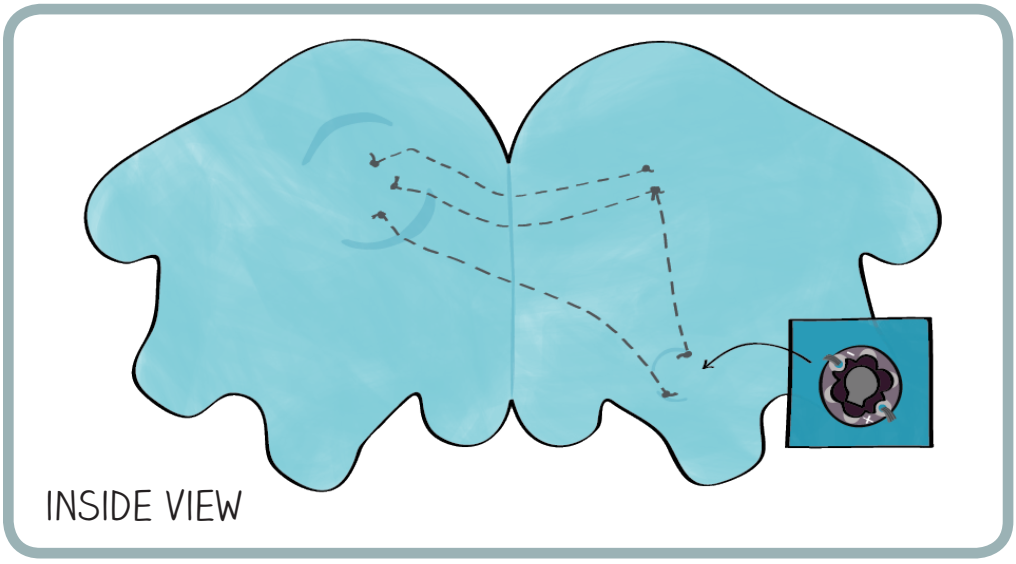
Build Your Monster: Attach Your Speaker

Remove the LilyPad Arduino SimpleSnap from your protoboard. Glue your speaker onto your monster. Remember not to fill its holes with glue.



Using your chalk or pencil, draw the electrical connections between the speaker and protoboard. Again, draw these connections on both the outside and inside of your monster.

Sew the traces for your speaker. Stitch the (–) tab of the speaker to the (–) tab of the LED or anywhere along the (–) trace you've already sewn. Stitch the (+) tab of the speaker to the appropriate pin on the Protoboard. Here, the (+) side of the speaker is attached to pin 5.

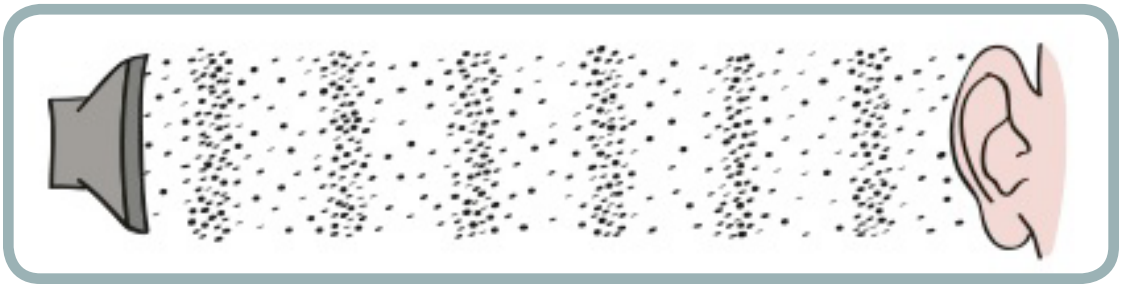


Make Your Monster Sing

UNDERSTANDING SOUND

Sound is created by vibrations of molecules in the air. When these vibrations hit your eardrums, your eardrums vibrate, and you hear a note. When air molecules vibrate very quickly, you hear a high note; when they vibrate more slowly, you hear a low note.

A speaker makes sound by vibrating — and making the air vibrate — at a particular speed, called a **frequency**.



Inside the LilyPad speaker, there is a material that moves in response to electricity. When this material moves, it creates air vibrations and sound.

THE TONE PROCEDURE

You're going to use Arduino's built-in `tone` procedure to create sound. This is what a line of code using the `tone` procedure looks like:

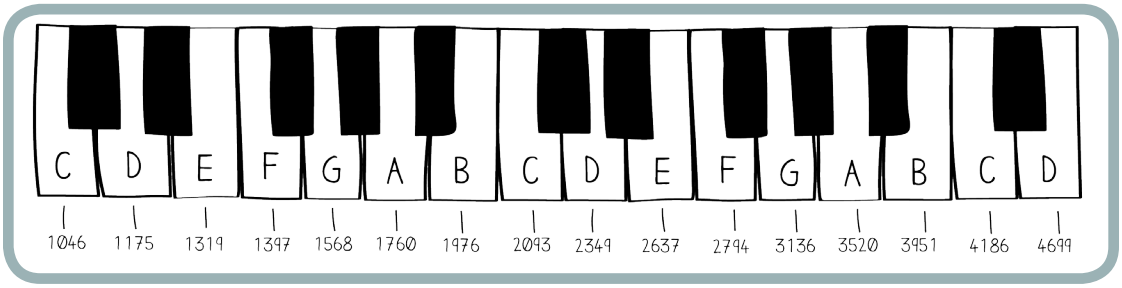
```
tone(5, 1760);     // play a note
```

The two numbers in parentheses after `tone` are the inputs to the procedure. Each of the two numbers changes the behavior of the procedure:

Procedure Name	Input 1	Input 2
`tone(`	`pin number`	`frequency);`

- **pin number:** This input is the number of the pin that the (+) end of your speaker is attached to. (In this case, we are using pin number 5.)
- **frequency:** This input is the frequency (or pitch) of the sound you want to play; the frequency is measured in Hertz (pulses per second). See the figure below for a chart of frequencies for one scale, and for more information, see: http://sewelectric.org/misc/musical-notes.

Here are some common notes and their frequencies:



Here is how you would use the `tone` procedure to play the note **A**:

```
tone(5, 1760);
```

With this line, you're telling the `tone` procedure that your speaker is connected to pin **5** and that you want to play a sound at a frequency of **1760** Hertz.

The `tone` procedure will play a note until you tell it to stop. The `noTone` procedure tells the LilyPad to stop playing a sound. The `noTone` procedure takes one input, the pin number that the (+) side of your speaker is attached to:

Procedure Name	Input 1
`noTone(`	`pin number);`

Here's how you would play the note **A** for 1 second (1000 milliseconds) and then pause for 1 second:

```
tone(5, 1760);      //note A begins playing
delay(1000);        //note plays for 1 second
noTone(5);          //note stops playing
delay(1000);        //silence for 1 second
```

MAKE A SOUND

Now you're going to use the `tone` and `noTone` procedures to make your monster beep and sing. Attach your LilyPad to your computer using the FTDI board and the mini USB cable, and snap it onto your protoboard. Open the Arduino software.

Open the code you wrote in the last section by clicking on the upward-pointing arrow on the Arduino Toolbar and selecting the file that you saved earlier. When it's open, it should look something like this:

```
int led = A4;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
blinkPattern();
}

void blinkPattern() {
digitalWrite(led, HIGH);
delay(100);
digitalWrite(led, LOW);
delay(100);
digitalWrite(led, HIGH);
delay(500);
digitalWrite(led, LOW);
delay(500);
}
```

Add a variable called `speaker` to the code to tell the LilyPad what pin number your speaker is attached to. This line should go right after the `int led = A4;` line. Since the (+) side of your speaker is attached to pin 5, you should set the `speaker` variable to 5:

```
int led = A4;
int speaker = 5;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
blinkPattern();
}
```

Since you are not going to edit the `blinkPattern` procedure now, the program snapshots in the rest of this section do not include it. However, you should still keep the `blinkPattern` procedure at the end of your code.

Next, set the `pinMode` for the speaker in the `setup` section. A speaker, like an LED, is an output — you're sending sound out into the world — so you need to set the speaker pin to be an output. Add the line `pinMode(speaker, OUTPUT);` to your program, right after the line `pinMode(led, OUTPUT);`.

Also, so that you can focus on your sound-generation code and not worry about your blink code, comment out the call to `blinkPattern` by adding `//` to the beginning of that line. Your code should now look like this:

```
int led = A4;
int speaker = 5;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
pinMode(led, OUTPUT);
pinMode(speaker, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
//blinkPattern();
}
```

Upload this code to your LilyPad. Your LED should stop blinking, but your speaker won't produce any sounds yet.

At the end of the `loop` section of your code, add a "call" to the `tone` procedure to get it to play the note A:

```
int led = A4;
int speaker = 5;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
pinMode(led, OUTPUT);
pinMode(speaker, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
//blinkPattern();
tone(speaker, 1760);
}
```

Compile and upload this new code. You should hear a continuous tone from your speaker. If no sound comes out of your speaker, see the troubleshooting section at the end of this lesson. If the sound starts to drive you crazy, unplug the USB cable from your computer, and turn off your LilyPad!

To give yourself a break from the continuous beeping, make the following adjustments to the `loop` section:

```
// the loop routine runs over and over again forever:
void loop() {
//blinkPattern();
tone(speaker, 1760);
delay(500);
noTone(speaker);
delay(1000);
}
```

Compile and upload the new code.

You may have noticed that it's not very intuitive to use numbers (frequencies) for musical notes. You can use variables to make working with `tone` easier and more intuitive. Add a variable for each note in the figure above to the top of your program right before the `setup` section:

```
int led = A4;
int speaker = 5;

int C = 1046;
int D = 1175;
int E = 1319;
int F = 1397;
int G = 1598;
int A = 1760;
int B = 1976;
int C1 = 2093;
```

Now you can use the names of notes instead of their frequencies when you're writing your code! You'll almost certainly find that the notes are much easier to remember. Edit the `loop` section of your code, replacing frequencies with notes:

```
// the loop routine runs over and over again forever:
void loop() {
//blinkPattern();
tone(speaker, A);
delay(500);
noTone(speaker);
delay(1000);
}
```

Now, experiment with the `tone` procedure. To change the note that it plays, call the procedure with a different note input. To change how long the note plays, adjust the `delay` after `tone`. For instance, here's how you would play the note C for 1 second:

```
// the loop routine runs over and over again forever:
void loop() {
//blinkPattern();
tone(speaker, C);
delay(1000);
noTone(speaker);
delay(1000);
}
```

Each time you change your code, compile and upload it to try out your edits. Save your code by clicking on the downward-pointing arrow in the Toolbar.

Tip: If steps in different sections in the environment confuse you, look ahead to a complete example song program on page 103.

PLAY A SONG

You can make the monster sing the first two verses of a song called "Hot Cross Buns" by editing the `loop` section like this:

```
void loop() {
//blinkPattern();
tone(speaker, E);
delay(2000);
tone(speaker, D);
delay(2000);
tone(speaker, C);
delay(2000);
noTone(speaker);
delay(2000);

tone(speaker, E);
delay(2000);
tone(speaker, D);
delay(2000);
tone(speaker, C);
delay(2000);
noTone(speaker);
delay(2000);

delay(5000);              //rest
}
```

NOTE: To reduce the chance of typing errors, it may be easier to copy and paste batches of code and then modify them.

Try making this change to your program, and then compile and upload the new code to your LilyPad. Can you recognize the song? Notice that the two verses consist of three notes followed by a pause. Also notice that the three-note phrase is repeated twice.

Create a procedure called **song** to store "Hot Cross Buns." Add the following code to your program, just after the closing bracket of the `loop` section but before the `blinkPattern` procedure definition:

```
delay(5000);              //rest

}void song() {

}

void blinkPattern() {
digitalWrite(led, HIGH);
delay(100);
digitalWrite(led, LOW);
delay(100);
digitalWrite(led, HIGH);
delay(500);
```

This creates the basic skeleton for your song procedure, but there's no code in its body yet — inside its curly brackets. To give the song procedure some behavior, cut and paste the first verse of "Hot Cross Buns" to its body (and delete the second verse):

```
void song() {
tone(speaker, E);
delay(2000);
tone(speaker, D);
delay(2000);
tone(speaker, C);
delay(2000);
noTone(speaker);
delay(2000);
}
```

Try compiling and uploading your code with this new addition to make sure you haven't made any errors.

To make use of your new procedure, you need to call it in the `loop` section of your code. Edit your `loop` section so that it looks like this:

```
void loop() {
//blinkPattern();
song();
song();

delay(5000);        //rest
}
```

Compile and upload the code. The behavior of your monster should be the same as it was before. Notice that you've replaced sixteen lines of code with two! Two calls to the `song` procedure replaced sixteen lines of code.

This example illustrates one way that procedures are powerful: They enable you to give a name to a piece of code that is repeated. Instead of typing out the repeated code over and over again, you can just call the procedure.

Procedures are powerful in other ways too. What if you wanted to make your song play faster? Say, hold each note for 1 second (1000 milliseconds) instead of 2 seconds (2000 milliseconds)?

You could write a new procedure called `fast_song` that looks like this:

(**NOTE:** Don't add this code to your program — just read along for a moment. Compare it to the code in your Arduino window.)

```
void fast_song() {
tone(speaker, E);
delay(1000);
tone(speaker, D);
delay(1000);
tone(speaker, C);
delay(1000);
noTone(speaker);
delay(1000);
}
```

What if you wanted to make the song even faster, holding each note for half a second (500 milliseconds) instead of one second? Then you'd need a third procedure. And if you wanted to go faster still, you'd need a fourth! This seems pretty crazy — especially since the basic behavior, the three notes played in a row, stays basically the same. All you want to do is adjust how quickly the notes are played.

You can use another powerful feature of code to capture this range of possible speeds in a single procedure. You can add an **input** variable to your procedure that controls how long the notes are played.

Change your song procedure to take an input called `duration`:

```
void song(int duration) {
tone(speaker, E);
delay(duration);
tone(speaker, D);
delay(duration);
tone(speaker, C);
delay(duration);
noTone(speaker);
delay(duration);
}
```

The `(int duration)` addition in the first line tells Arduino that the procedure song requires an input called `duration` that's an `int` (short for an integer, a whole number). When you call song, you now need to include a number for `duration`. Here's how you'd get your original slow song to play:

```
void loop() {
//blinkPattern();
song(2000);
song(2000);

delay(5000);              //rest
}
```

Make these edits to your code and then compile and upload it. You should hear the same song as before. When `song` runs, every instance of duration is replaced by the number you put in parentheses after `song`. In the case above, every `duration` in the `song` procedure would be replaced by **2000**.

Now, edit the loop section to take advantage of the new duration `input` to `song`:

```
void loop() {
//blinkPattern();
song(2000);               // play the song slowly
song(1000);               // play the song faster
song(500);                // and even faster

delay(5000);                    //rest
}
```

Compile and upload this new version of the code. Does the song sound different? Can you hear the speed changes?

Save your code by clicking on the downward-pointing arrow in the Toolbar.

EXPERIMENT

Try editing the body of the song procedure so that it plays a different tune. Can you get your monster to play "Mary Had a Little Lamb," "Happy Birthday," Beethoven's "Symphony No. 9," or your own composition?

Hint 1: Here's a version of the song procedure that plays a scale:

```
void song(int duration) {
tone(speaker, C);
delay(duration);
tone(speaker, D);
delay(duration);
tone(speaker, E);
delay(duration);
tone(speaker, F);
delay(duration);
tone(speaker, G);
delay(duration);
tone(speaker, A);
delay(duration);
tone(speaker, B);
delay(duration);
tone(speaker, C1);
delay(duration);
noTone(speaker);
delay(duration);
}
```

Hint 2: You might want to play different notes for different amounts of time. There are a few different ways to approach this (each approach is described in a comment):

```
void song(int duration) {
tone(speaker, C);
delay(100);        //duration is permanently set at 100ms
tone(speaker, D);
delay(duration*2);//duration is twice as long as input
tone(speaker, E);
delay(duration/2);//duration is half as long as input
}
```

If you're programming with a friend or in a group, compare your code and your song with theirs as you experiment.

Once you've created a melody that you like, you probably want to add the blinking behavior you created in the last section back into your program. Do this by removing the `//` characters that are in front of the `blinkPattern` procedure call in `loop`. You may also want to adjust and delete the final `delay` in `loop`. Compile and upload your new code.

```
void loop() {
blinkPattern();
song(2000);
delay(5000);              //rest
}
```

Note: Your code may look different than the example above. You may have more calls to the song procedure, or you may be using a different `duration` input to the `song` procedure. If so, that's fine.

See how your blinking and singing monster behaves. At this point, you may also want to adjust your `blinkPattern` or song procedures so that they work well together.

SAVE YOUR CODE

Once you are happy with your program, save it by clicking on the downward-pointing arrow in the Toolbar.

Your entire program should now look something like the following. Note: The bodies of the song and `blinkPattern` procedures may be different in your code. The `loop` section may be slightly different, as well.

```
int led = A4;
int speaker = 5;      // speaker is attached to pin 5

int C = 1046;
int D = 1175;
int E = 1319;
int F = 1397;
int G = 1598;
int A = 1760;
int B = 1976;
int C1 = 2093;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
pinMode(led, OUTPUT);
pinMode(speaker, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
blinkPattern();
song(2000);
delay(5000);              //rest
}

void song(int duration) {
tone(speaker, C);
delay(duration);
tone(speaker, D);
delay(duration);
tone(speaker, E);
delay(duration);
tone(speaker, F);
delay(duration);
tone(speaker, G);
delay(duration);
tone(speaker, A);
delay(duration);
tone(speaker, B);
delay(duration);
tone(speaker, C1);
delay(duration);
noTone(speaker);
delay(duration);
}

void blinkPattern() {
digitalWrite(led, HIGH);
delay(100);
digitalWrite(led, LOW);
delay(100);
digitalWrite(led, HIGH);
delay(500);
digitalWrite(led, LOW);
delay(500);
}
```

TROUBLESHOOTING

If your speaker doesn't work as expected:

1. When you upload your program, does it compile and upload successfully, or are there red error messages? If you receive error messages, you need to correct a problem in your code. Carefully compare your code to the example, looking especially for missing parentheses and semicolons. Fix any problems, and try uploading again.

2. Check your program code. Did you set the `speaker` variable to the correct pin number? The correct pin number is the one that you sewed the (+) side of your speaker to with the conductive thread.

3. Look at where you tied your conductive thread knots. Are any stray knot ends touching each other where they shouldn't be? Make sure that all knots are neatly trimmed and sealed with glue.

4. Check the traces of conductive thread that you have sewn. Do they cross anywhere they shouldn't? Do they go all the way from the protoboard to the speaker? Are there tight loops of thread on every connection to the protoboard and speaker?

Add a Sensor to Your Monster

This section describes how to add a touch sensor to your monster. This sensor enables your monster to respond to interaction — for example, you can make your monster blink or play songs only when someone holds its paws.

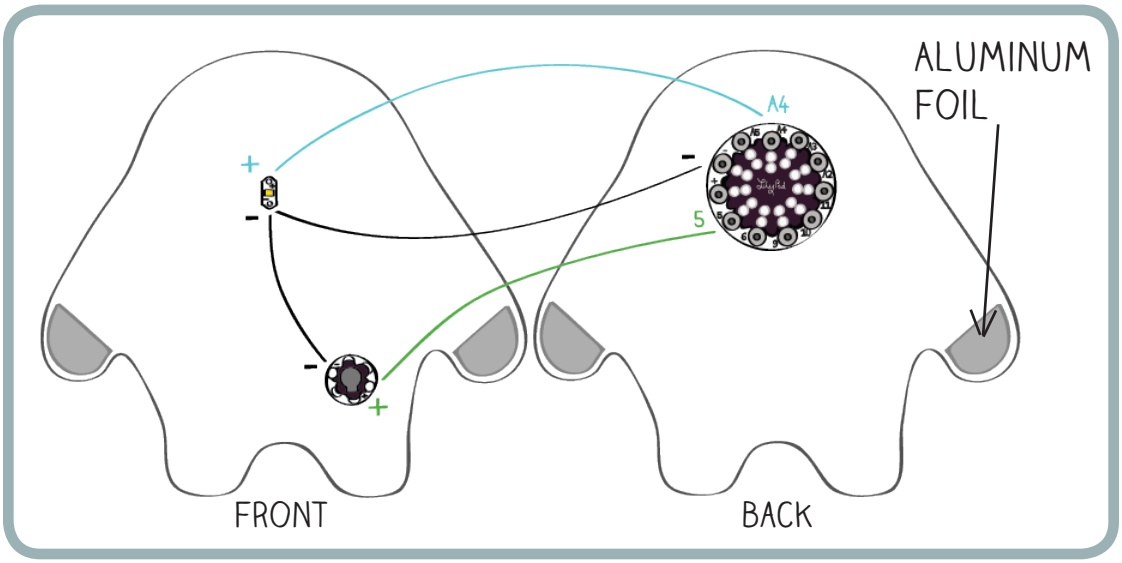
If you're not adding a sensor to your monster, you can jump to the section on sewing and stuffing your monster.

Either way, gently pry apart and set aside the LilyPad Arduino SimpleSnap ("LilyPad") from the protoboard so you can manipulate your monster.

DESIGN YOUR SENSOR

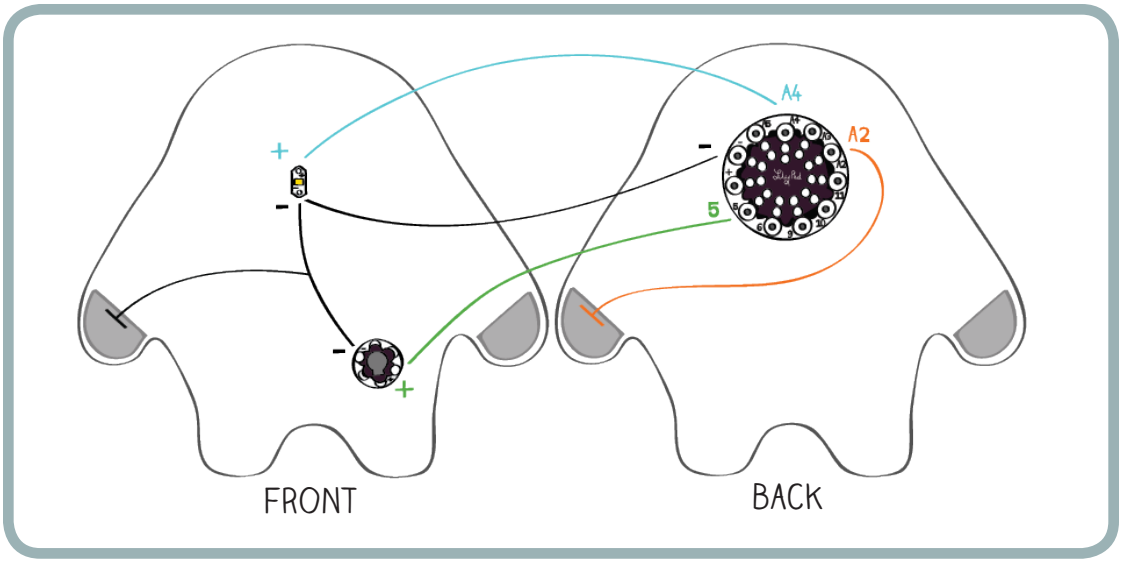
The sensor you're building consists of two aluminum foil patches. These patches will detect when a person touches them. A person has to touch both patches at the same time for the sensor to "feel" the touch.

The patches are attached to the monster's paws so that the monster can detect when someone holds its hands. They need to go on the outside of your monster. Add the patches to your design sketch.



Though it looks like there are four sensors on the monster, note that when you stitch the monster together, you'll only have two: one on each paw, covering both the front and back of the paws. For this reason, when adding your paw sensors, make sure you fold over your monster to confirm that the paws match up.

For the sensor to function electrically, one paw needs to be attached to (–) on the LilyPad and the other needs to be attached to tab A2, A3, A4, or A5. Here, the second paw is attached to tab A2:

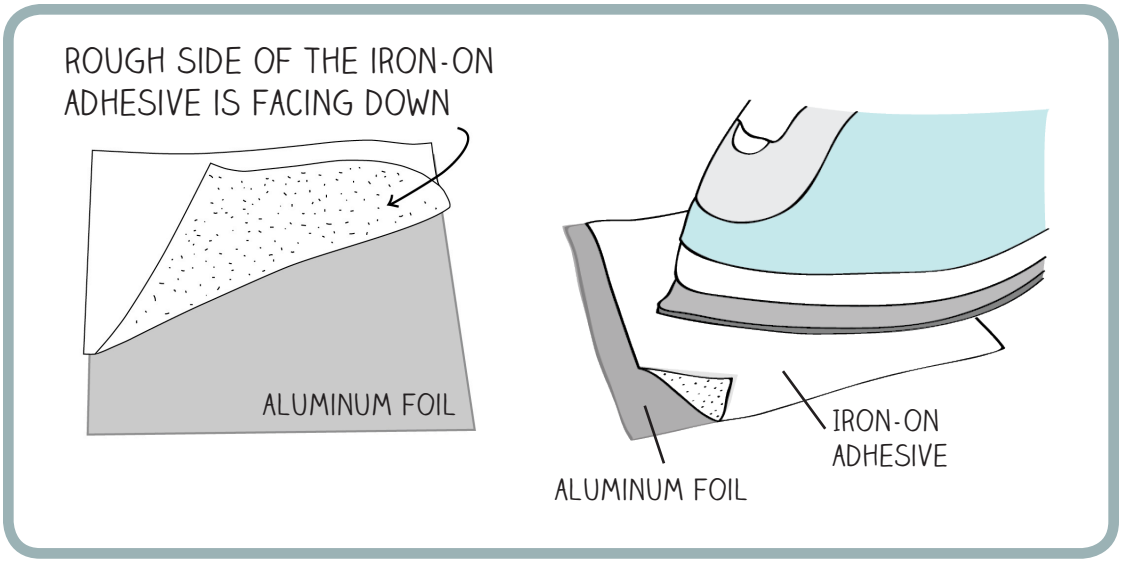


Use colored pencils to add these details to your design sketch. Use black for the (–) paw and a new color for the A2 paw.

MAKE YOUR SENSOR AND ATTACH IT TO YOUR MONSTER

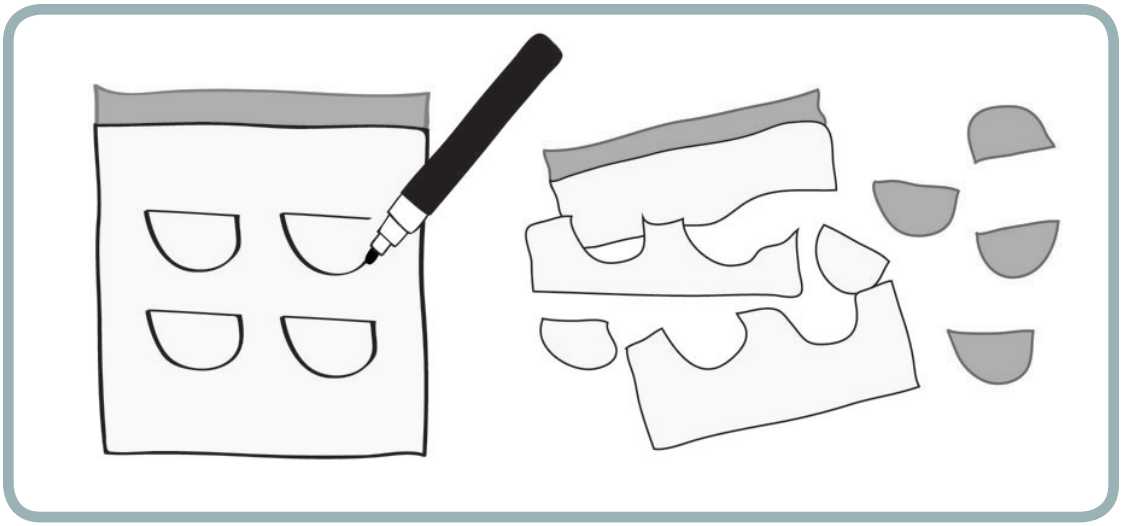
Get out your aluminum foil, iron-on adhesive, and iron. Read through the instructions on the Heat-n-Bond™ packaging to find the correct temperature setting for your iron and to familiarize yourself with the iron-on process.

Place a piece of aluminum foil on your ironing board. Place a matching piece of iron-on adhesive on the aluminum foil. The rough (adhesive) side should be facing down. Iron the adhesive to the foil.

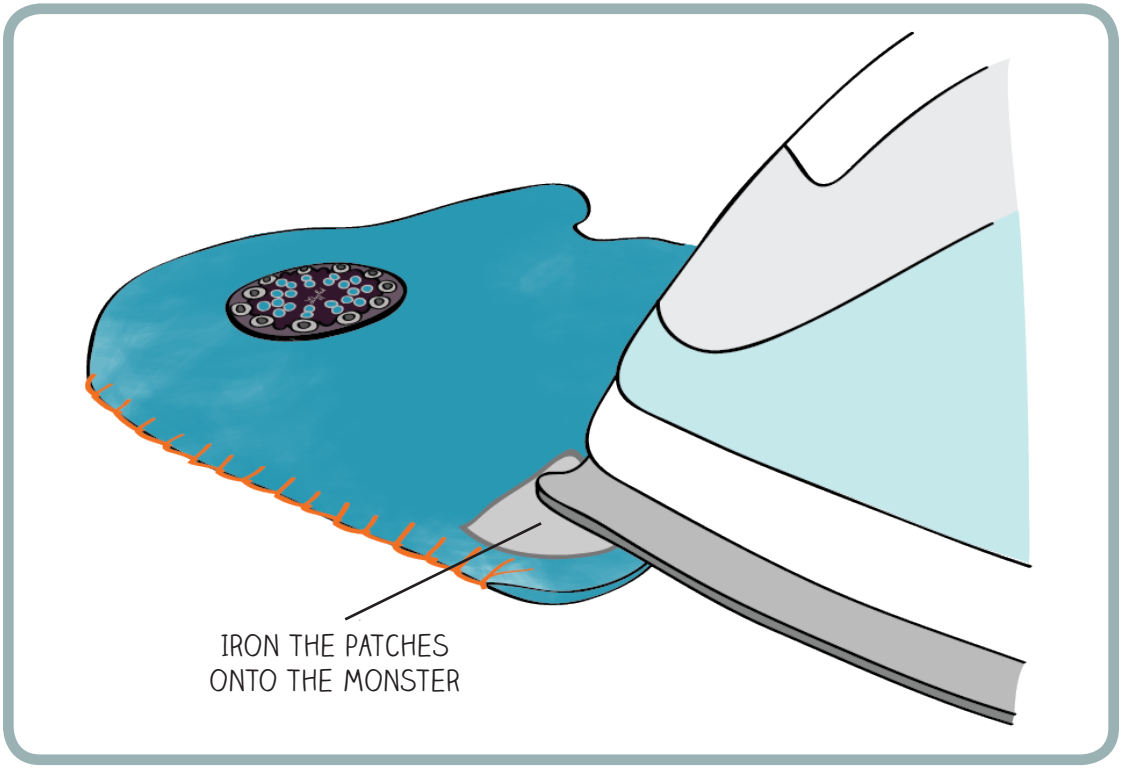


To make sure that the adhesive is firmly attached, peel up a small corner of the paper. It should peel away easily, and you should see a layer of clear adhesive stuck to the aluminum foil. Only peel up enough paper to check on the adhesive. Don't peel away the paper yet.

Now you want to design and cut your sensors, using the iron-on sheet of aluminum foil you just created. On the paper that is attached to the aluminum foil, draw the shapes for your sensor patches. The monster template you made may be useful for this step.



Cut out your sensor patches, peel the paper off them, and iron them onto the outside of your monster. Make sure that the adhesive side of the aluminum foil is facing your fabric. You don't want the glue to stick to your iron. It'll make a stinky sticky mess!

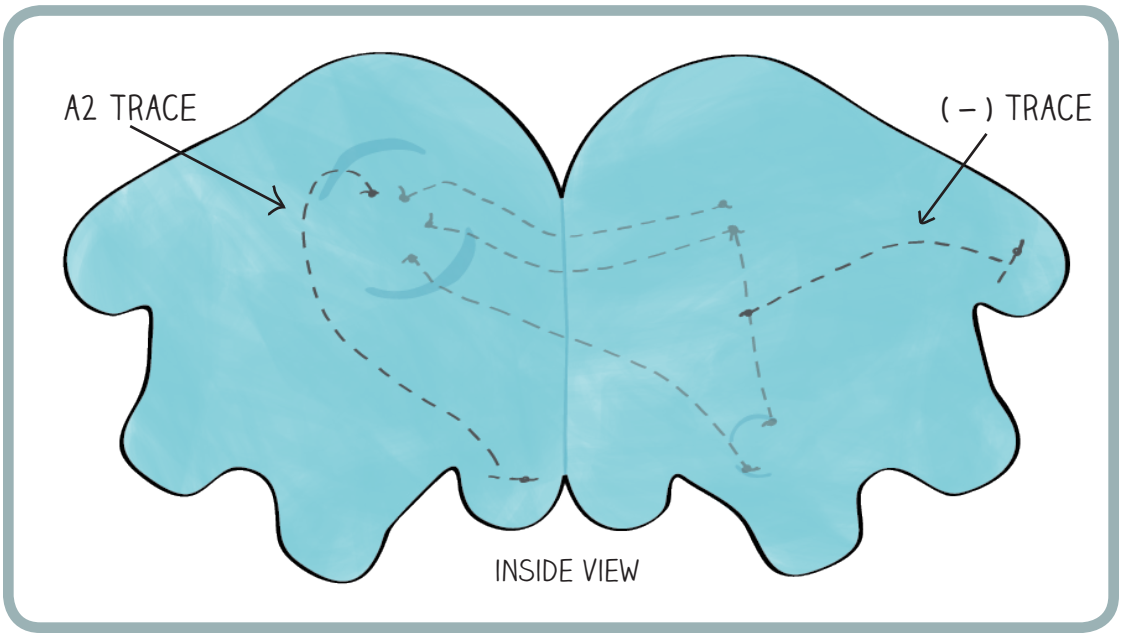


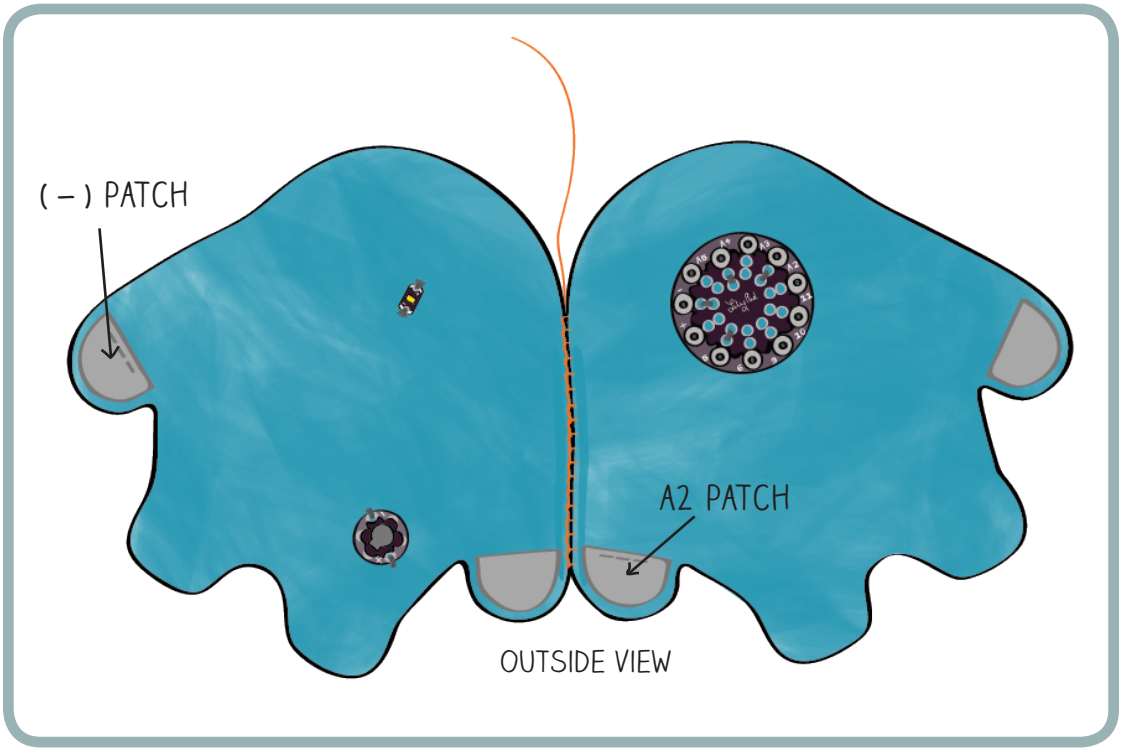
Next, use chalk or pencil to mark the connections between your sensor patches and the rest of your circuit. As you have done before, mark the connections on both the inside and outside of your monster.

Now you're ready to sew the connections between your patches and the rest of your circuit.

On the back piece of the monster, sew from pin A2 on the protoboard to one patch. Make at least three running stitches through the aluminum foil to create a solid electrical connection between the patch and your conductive thread.

On the front piece of the monster, sew from the (–) trace near your LED to the second sensor patch.





Give Your Monster a Sense of Touch

Attach your LilyPad to your computer and snap it onto your Protoboard. Open the Arduino software. Open the code you wrote in the last section by clicking on the upward-pointing arrow on the Arduino Toolbar and selecting the file you saved earlier.

Begin by adding a variable called "aluminumFoil" to the top of your program right below the `intspeaker = 5;` line. This will tell the LilyPad which tab the aluminum foil patch is connected to.

You also need to add some code to the setup section of your program. You need a **`pinMode`** statement like `pinMode(led, OUTPUT);` for your sensor. But a sensor is an input, not an output, so the statement is slightly different: You use `INTPUT` instead of `OUTPUT`. The top half of your program should now look like this:

```
int led = A4;
int speaker = 5;      // speaker is attached to pin 5
int aluminumFoil = A2;

int C = 1046;
int D = 1175;
int E = 1319;
int F = 1397;
int G = 1598;
int A = 1760;
int B = 1976;
int C2 = 2093;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
pinMode(led, OUTPUT);
pinMode(speaker, OUTPUT);
pinMode(aluminumFoil, INPUT);
}
```

Make these changes and compile and upload the updated program to your LilyPad to make sure it doesn't have any errors.

Now you're going to add several lines of code to your program and delete the `delay` line after the call to your `song` procedure. The additions are highlighted below. The tutorial will describe what each line does. For now, edit your code so that it looks like the following:

```
int led = A4;
int speaker = 5;
int aluminumFoil = A2;
int sensorValue;

int C = 1046;
int D = 1175;
int E = 1319;
int F = 1397;
int G = 1598;
int A = 1760;
int B = 1976;
int C2 = 2093;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
pinMode(led, OUTPUT);
pinMode(speaker, OUTPUT);
pinMode(aluminumFoil, INPUT);
digitalWrite(aluminumFoil, HIGH);  // initializes the sensor
Serial.begin(9600);              // initializes the communication
}

// the loop routine runs over and over again forever:
void loop() {
blinkPattern();
song(2000);
sensorValue = analogRead(aluminumFoil);
Serial.println(sensorValue);
delay(100); //delay for 1/10 of a second
}
```

Try uploading this code. If you encounter compile errors, read through your program carefully to make sure the code you added matches the example. Don't confuse letters — for instance, "l" ("el") for "i" ("eye"). Look especially for missing or misplaced parentheses "(" ")", brackets "{" "}", and semicolons ";".

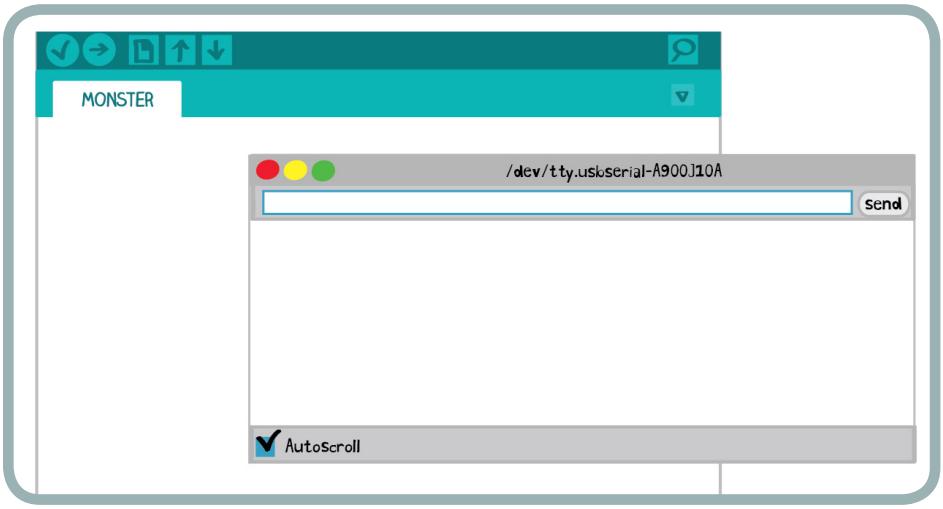
Once you have successfully uploaded this new code to your LilyPad, comment out any `blinkPattern` and `song` procedure calls that are in the `loop` section of your program. This will let you focus on the sensing part of the program:

```
// the loop routine runs over and over again forever:
void loop() {
//blinkPattern();
//song(2000);
sensorValue = analogRead(aluminumFoil);
Serial.println(sensorValue);
delay(100); //delay for 1/10 of a second
}
```

NOTE: You may have more than two lines that need to be commented out of your code. Make sure you comment out all the blinking and song-playing code before you move on to the next step.

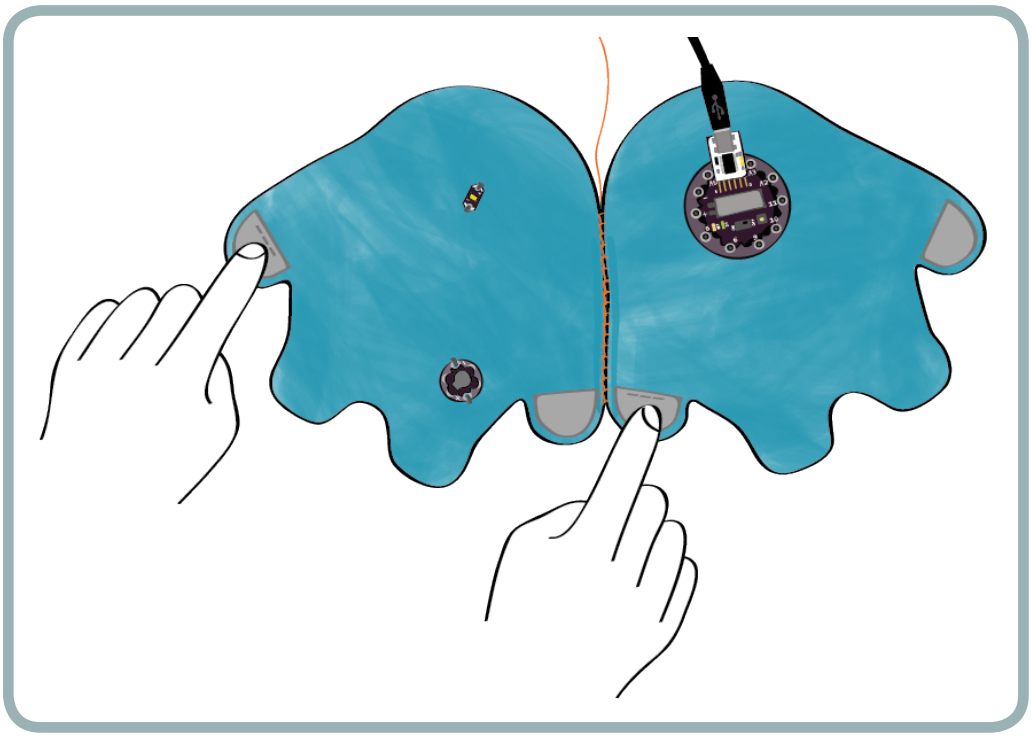
Compile and upload this code. Click on the magnifying-glass icon in the upper-right-hand corner of the Arduino window.

A small window called the "serial monitor" will pop up. Any values your LilyPad's program sends back to the computer while it's running will appear in this window.



You should see a steady stream of numbers:

These values should be close to 1000. Now try touching your aluminum foil patches.



See what happens to the values in the serial monitor. Try pressing your palms firmly against the pads and then lessening your pressure. Notice what happens to the stream of numbers. Touch the two pads to each other, and see what happens to the numbers.

If you don't see any changes, this means that something is wrong with your code, your sensor construction, or the way you're touching the sensors. First, make sure you're touching one hand firmly to each of the sewn patches of your sensor (as in the diagram above). If this still doesn't work, see the troubleshooting section at the end of this tutorial for advice.

MAKING SENSE OF THE CODE: ANALOGREAD

Look at the heart of the code, the loop section:

```
// the loop routine runs over and over again forever:
void loop() {
//blinkPattern();
  //song(2000);
sensorValue = analogRead(aluminumFoil);
Serial.println(sensorValue);
delay(100); //delay for 1/10 of a second
}
```

The first line after the commented-out code is:

```
sensorValue = analogRead(aluminumFoil);
```

This line **reads** information from your aluminum foil sensor. The information or "value" read from the sensor is stored into the variable called `sensorValue`.

The `analogRead` procedure reads sensor data from a pin on the LilyPad. It takes one input, the pin number that your sensor is attached to, and gives back or "returns" a value that corresponds to the sensor reading.

Procedure Returns	Procedure Name	Input 1
`sensor value`	`analogRead(`	`pin number);`

You want to read information from your aluminum foil sensor, which is why you use the statement `analogRead(aluminumFoil);` in your program.

You may be wondering what kind of reading the `analogRead` procedure takes from the sensor and what kind of values it returns. The `analogRead` procedure measures the voltage level on a tab and returns a number between 0 and 1023 that corresponds to the voltage. This table shows how a few of the numbers it returns correspond to voltages:

Number Returned by Analogread	Electrical Value		
	Symbol	Voltage[2]	Other Names
1023	(+)	3.7 volts	power, HIGH
512		1.85 volts	
0	-	0 volts	ground, LOW

This chart may remind you of a similar one you saw earlier that explained how the terms `HIGH` and `LOW` and the statements `digitalWrite(pin, HIGH);` and `digitalWrite(pin, LOW);` related to voltages. Now, instead of generating (or "writing") voltages with `digitalWrite`, you're using `digitalRead` to sense (or "read") them.

Returning to the sensor and the numbers that you're seeing in the serial monitor: Why do the readings from the sensor change when you touch the sensor? Or, to put it a different way, why do the voltages measured at pin A2 change when you touch the sensor?

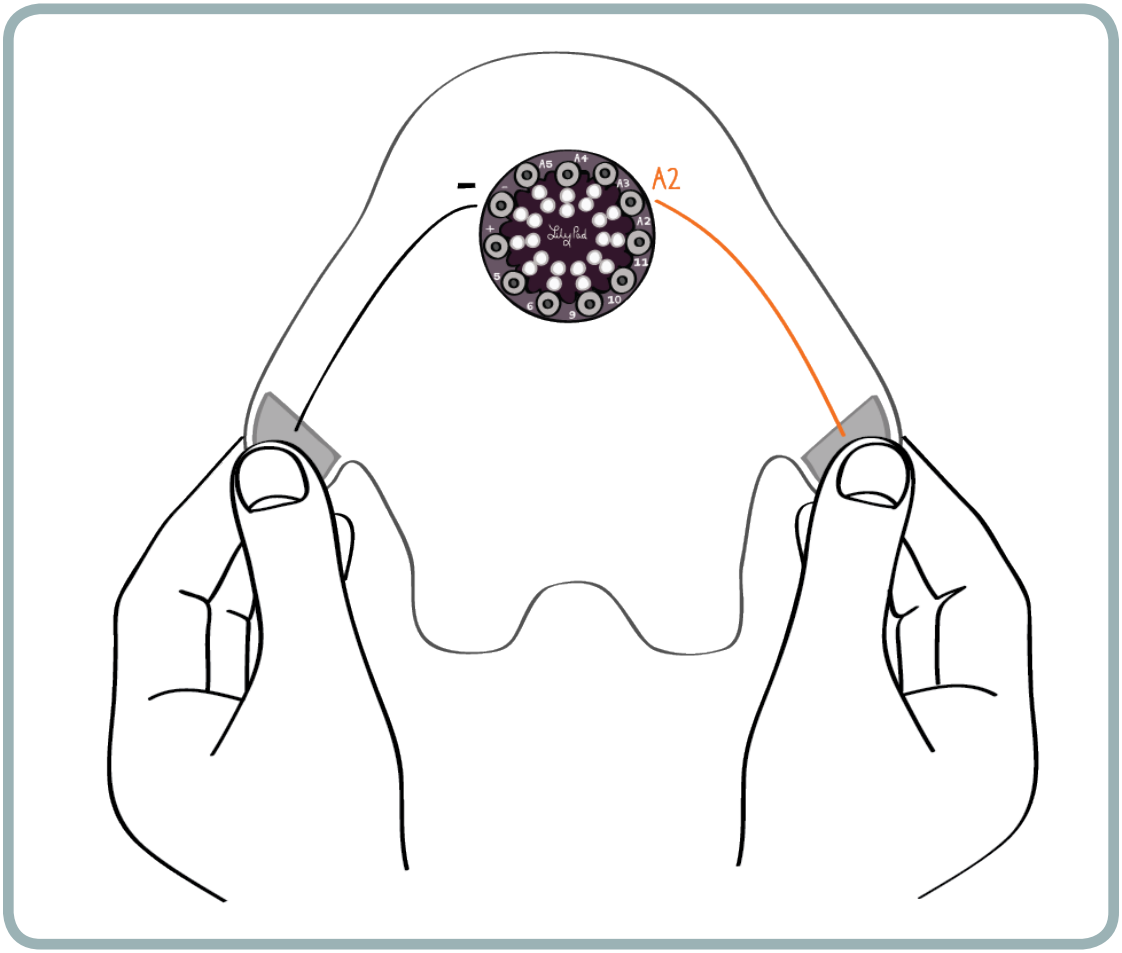
In the setup section of your program, you used this line to set the sensor's value to `HIGH` or 3.7 volts:

```
digitalWrite(aluminumFoil, HIGH);  // initializes the sensor
```

This is why you see very high numbers — numbers close to 1023 (or `HIGH`) — in the serial monitor when you're not touching the sensor. When you're not touching the sensor, it returns its default value (`HIGH`).

[2] These voltages assume that the LilyPad Simple Snap is being powered from its 3.7 volt battery.

When you touch the aluminum foil patches, one of your hands is connected to the (–) pin (which is at 0 volts or `LOW`), and one of your hands disconnected to pin A2 (which is at 3.7 volts or `HIGH`). Your body, which is slightly conductive, creates an electrical connection between (–) and the sensing pin A2. This has the effect of lowering the voltage on pin A2, bringing it closer to 0 volts. If you folded your monster in half and looked at it from the back, this is what it would look like:



This is why the numbers in the serial monitor go down when you touch the sensor. The harder you squeeze on the sensor, the stronger the connection is between your body and the aluminum foil, and the better the electrical connection is between (–) and A2. This is why you see lower numbers as you squeeze harder. This is also why you see very low numbers (close to 0) when you touch the two paws directly together.

MAKING SENSE OF THE CODE: SERIAL.PRINTLN

The code you just wrote is doing another important thing that you haven't examined yet. It is sending information from the LilyPad back to the computer, where it is displayed in the serial monitor. The lines of your program that take care of this important communication function are in the `setup` and `loop` sections. Each line involved in the communication is highlighted below:

```
// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
pinMode(led, OUTPUT);
pinMode(speaker, OUTPUT);
pinMode(aluminumFoil, INPUT);
digitalWrite(aluminumFoil, HIGH);  // initializes the sensor
Serial.begin(9600);              // initializes the communication
}

// the loop routine runs over and over again forever:
void loop() {
//blinkPattern();
  //song(2000);
sensorValue = analogRead(aluminumFoil);
Serial.println(sensorValue);
delay(100); //delay for 1/10 of a second
}
```

There are two important pieces to the communication code. First, in the setup section, you need to tell the LilyPad that it will be communicating back to the computer. You also need to tell the LilyPad how fast it should talk to the computer. (The LilyPad and the computer can "talk" at different speeds, and they need to be talking at the same speed to understand each other.) This is accomplished with the `Serial.begin` procedure:

```
Serial.begin(9600);
```

Procedure Name	Input
`Serial.begin(`	`communication speed);`

The number in parentheses tells the LilyPad what speed to use to talk to the computer. You are using speed 9600 because it is the standard Arduino communication speed. The entire statement basically tells the LilyPad to be prepared to talk to the computer.

The second piece of communication code is in the loop section. It's the code in the loop that actually sends information from the LilyPad to the computer. In this program, you're sending the readings you took from your sensor back to the computer. The lines that do this are:

```
Serial.println(sensorValue);
delay(100); //delay for 1/10 of a second
```

The `Serial.println` procedure tells the LilyPad to send information back to the computer through the USB cable. The value in the parentheses is what gets sent back to the computer. In your code, the sensor readings stored in the variable `sensorValue` get sent back to the computer.

Procedure Name	Input
`Serial.println(`	`value to send);`

There is one last important piece of the communication code: the statement `delay(100);`. This short delay gives the computer time to process and display the information that the LilyPad is sending. Without this delay, the computer will become overloaded with information from the LilyPad and crash. Whenever you have a `Serial.println` statement in your code, it should be followed by a `delay` statement.

USING THE SENSOR TO CONTROL YOUR MONSTER: IF ELSE

Now that you know how to read information from a sensor, you can use it to control your monster's behavior. Say you want the LED to light up if your hands are touching the aluminum foil sensor and turn off if you're not touching the sensor.

The way you describe this situation in code is similar to the way you'd say it in a sentence. You use what's called a **conditional statement** to say:

- **If** I am touching the aluminum foil sensor, the LED should turn on.
- **Otherwise** the LED should turn off.

In your program, you replace the word "otherwise" with the word "else." Here's what the **if else** conditional statement looks like (half in code and half in written English):

```
if(condition)
{
  //turn on
}
else
{
//turn off
}
```

Notice that the body of the if and else statements are enclosed in curly brackets. Also notice that the **condition** of the if statement is inside a pair of parentheses. Now you'll work on writing an actual conditional statement to see how the code's punctuation, structure, and behavior all come together.

You saw earlier, from the numbers in the serial monitor, that you get sensor values around 1023 when you aren't touching the monster, and that these values decrease when you squeeze the two paws. To use this behavior in an `if else` statement, add a few lines to your program's `loop` section:

```
// the loop routine runs over and over again forever:
void loop() {
//blinkPattern();
  //song(2000);
sensorValue = analogRead(aluminumFoil);
Serial.println(sensorValue);
delay(100); //delay for 1/10 of a second

if(sensorValue< 1000)  // if you are touching sensor
{
digitalWrite(led, HIGH);    // turn the LED on
}

}
```

Make this adjustment to your code, and compile and upload it to your LilyPad. What happens to your LED when you touch your sensors now? What happens to the LED when you let go of the sensors?

`sensorValue< 1000` is the **condition** that determines whether the LilyPad runs the code that is inside the `if` statement's curly brackets—whether the LED turns on. A condition is a statement that is either true or false. The statement `sensorValue< 1000` is **true** when sensorValue is less than (<) 1000 — for example, when it's 800. The statement `sensorValue< 1000` is **false** when `sensorValue` is greater than 1000 — for example, when it's 1010, 1001, or 1023.

Make the behavior a little more interesting by finishing your **if else** statement:

```
// the loop routine runs over and over again forever:
void loop() {
//blinkPattern();
  //song(2000);
Serial.println(sensorValue);
delay(100); //delay for 1/10 of a second

if(sensorValue< 1000)      // if touching sensor
  {
digitalWrite(led, HIGH);    // turn the LED on
  }
else        // if you are NOT touching sensor
{
digitalWrite(led, LOW);    // turn the LED off
}
}
```

Try compiling and uploading this new code to your LilyPad. What happens now when you squeeze and release the touch sensors?

Adjust your threshold (yours might be higher or lower than 1000) until your monster can clearly distinguish between touching and not touching. Now you should be able to reliably control your monster's LED by touching the aluminum foil pads!

This is a good time to save your program.

PUTTING IT ALL TOGETHER: CONTROL THE LED AND SPEAKER

Instead of turning the LED on and off when you touch your sensors, you can have it blink or sing using the custom procedures you wrote earlier. Edit the code so that your monster blinks and sings when its paws are squeezed. Delete the `digitalWrite` statements that turn your LED on and off, move the calls to `blinkPattern` and song into your `if` statement, and remove the `//` characters on each of their lines:

```
// the loop routine runs over and over again forever:
void loop() {
Serial.println(sensorValue);
delay(100); //delay for 1/10 of a second

if(sensorValue< 1000)       // if you are touching sensor
  {
blinkPattern();
song(2000);
  }
else          // if you are NOT touching sensor
  {

  }
}
```

EXPERIMENT

Experiment with your code to try out different behaviors. Can you edit the code so that your monster blinks until someone touches it and then it plays a song? Can you have your monster play a song slowly when you're touching the paws lightly and quickly when you're touching the paws firmly?

If you're programming with a friend or in a group, see how your monster's behavior and code compares to theirs. Are they using different strategies or techniques that you could borrow? Can you show them something that they haven't discovered yet?

SAVE YOUR CODE

When you're happy with your program, save it by clicking on the downward-pointing arrow in the Toolbar.

Your entire program should now look something like this. Note that the bodies of the song and `blinkPattern` procedures will be different in your code, and the `loop` section may be slightly different as well.

```
int led = A4;
int speaker = 5;
int aluminumFoil = A2;
int sensorValue;

intC = 1046;
intD = 1175;
intE = 1319;
intF = 1397;
intG = 1598;
intA = 1760;
intB = 1976;
intC2 = 2093;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
pinMode(led, OUTPUT);
pinMode(speaker, OUTPUT);
pinMode(aluminumFoil, INPUT);
digitalWrite(aluminumFoil, HIGH);  // initializes the sensor
Serial.begin(9600);              // initializes the communication here
}

// the loop routine runs over and over again forever:
void loop() {
Serial.println(sensorValue);
delay(100); //delay for 1/10 of a second

if(sensorValue< 1000)       // if you are touching sensor
  {
song(2000);
  }
else             // if you are NOT touching sensor
  {
blinkPattern();
  }
}

void song(int duration) {
tone(speaker, C, duration);
tone(speaker, D, duration);
tone(speaker, E, duration);
tone(speaker, F, duration);
tone(speaker, G, duration);
tone(speaker, A, duration);
tone(speaker, B, duration);
tone(speaker, C1, duration);
}

void blinkPattern() {
digitalWrite(led, HIGH);
delay(100);
digitalWrite(led, LOW);
delay(100);
digitalWrite(led, HIGH);
delay(500);
digitalWrite(led, LOW);
delay(500);
}
```

TROUBLESHOOTING

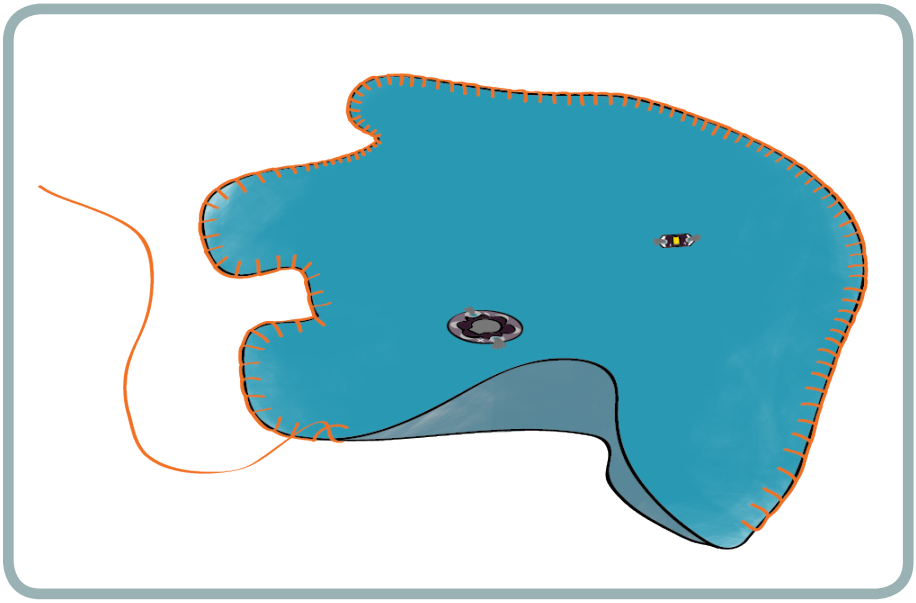
If your sensor doesn't work as expected:

1. When you uploaded the program, did it compile and upload successfully, or were there red error messages? If you received error messages, you need to correct a problem in your code. Carefully compare your code to the example, looking especially for missing parentheses and semicolons. Fix any problems and try uploading again.
2. Check your program code. Did you set the `aluminumFoil` variable to the correct pin number? The correct pin number is the one that you sewed one of your aluminum foil patches to.
3. Is the threshold value you set for your sensor appropriate? You should choose a number that is slightly above the numbers you see in the serial monitor when you touch your sensor pads. This number goes into the line `if(sensorValue< 1000)`.
4. Look at where you tied your conductive thread knots. Are any stray knot ends touching each other where they shouldn't be? Make sure that all knots are neatly trimmed and sealed with glue or nail polish.
5. Check the traces of conductive thread that you have sewn. Do they cross anywhere they shouldn't? Do they go all the way from the protoboard to the aluminum foil patches?
6. Are there tight loops of thread on every connection to the protoboard and aluminum foil patches?

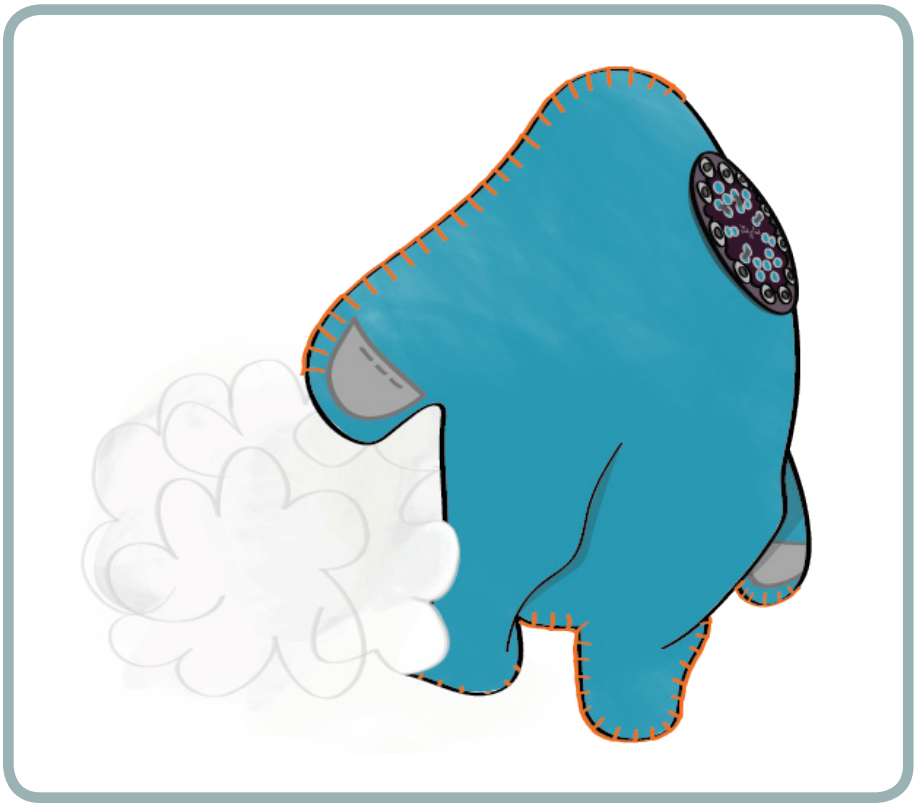
Sew and Stuff Your Monster

Now that all your monster's electronics are sewn on and tested, you can stitch the monster together. Carefully pry apart and remove the LilyPad from the Protoboard before you begin.

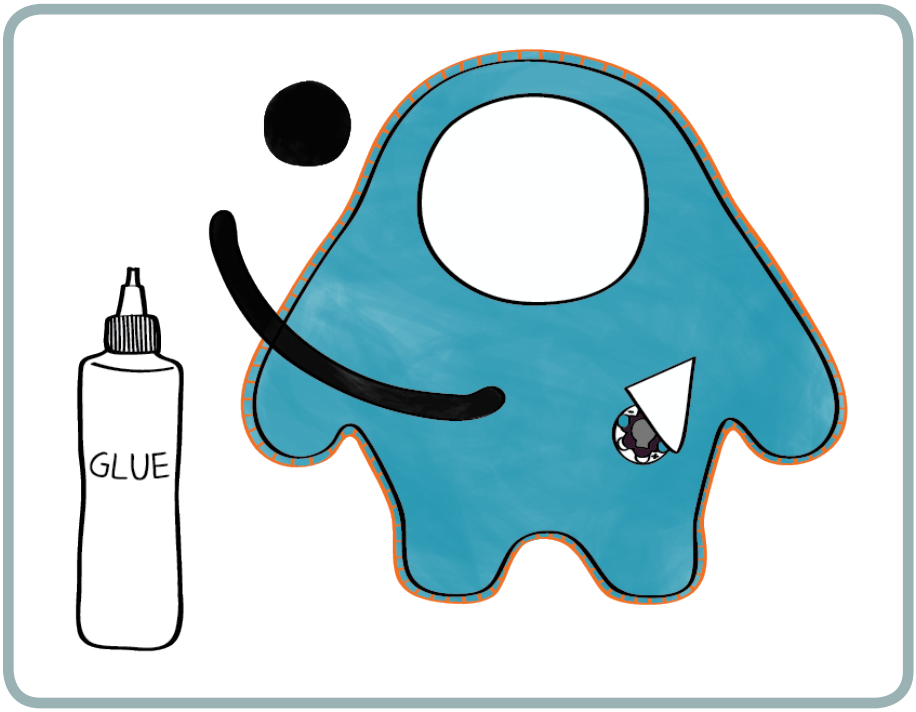
Using the (nonconductive) embroidery thread, continue stitching the monster's two sides together along the outside edge of the monster until about 2 inches of space remains open. Leave your embroidery thread uncut.



Stuff the inside of the monster with polyester filling until it's full. Then stitch up the rest of the seam, so that your monster is completely enclosed. Tie off the embroidery thread, and snip the ends off.



Glue on your monster's eyes, mouth, claws, and other features.



If your LilyPad is not attached to your monster, snap it on now. Unplug the LilyPad from your computer and turn it on.

Squeeze your monster, hold its hands, and listen to it sing! Take it into a dark room to see how bright and blinky it can be. Take it for a walk. Invite your friends over for a slumber party. Show off your new friend!