

Introduction

When we first started teaching computer science, we discovered two important things. We found that existing curriculum for beginners focused mostly on solving math problems or constructing geometric shapes and that there was a certain type of student that signed up for computer science classes and these students were almost always boys. We wondered whether a different approach to teaching the basics of computer programming would be more engaging and also attract a larger variety of different types of students, both boys and girls.

We decided to focus on what knowing how to program allowed you to do and create. Ultimately all programs are created to solve a problem or serve a purpose. The problem may be local or global, the purpose may be anything from helping doctors treat patients to pure entertainment. By starting with interesting problems the students wanted to solve, they were much more engaged in learning to code. They saw coding skills as an important part of building creative solutions.

With this approach, we found that not only did we get more girls taking the course, we also got a more diverse group of boys. Opportunities for collaboration increased, and all the students got to see where their talents and skills meshed with others' interests and experiences, to make a whole that was greater than the sum of its parts.

We are now at the point where a third of the students taking computer science are girls, and more importantly, students are coming out of the course not only with an understanding of code, but also knowing how to read through professionally written code, and take an idea from brainstorming through prototyping to build something that matters.

- *Authors Mary Kiang and Douglas Kiang*

Course Introduction

This is an introduction to coding and computer science by way of making and design, using the revolutionary new micro:bit microcontroller board, and Microsoft's easy and powerful MakeCode block-based coding environment. It is a project-based curriculum with a maker philosophy at its core; the idea is that by making physical objects, students create a context for learning the coding and computer science concepts.



- Micro:bits may be purchased from these resellers: <http://microbit.org/resellers> (you will need 1 micro:bit per student for this course). The "Micro:bit Go Kit" includes a battery pack and USB cable as well.
- Other optional suggested micro:bit accessories include:
 - Alligator/Crocodile clip cables



- Headphone/earbuds (for audio)



- Servo motor (for movement)



- Croc clip to Male connector (for connecting to Servo motor)



- Croc clip to Headphone jack adapter (<http://microbit-accessories.co.uk/>)



- MakeCode for the micro:bit is a free web app: <https://makecode.microbit.org>



When students complete this course they will have a good understanding of computer science concepts that can serve as the foundation for future study. They will develop powerful design skills that they can use in future projects of all types, whether they are designing 3D printed prototypes or creating apps that serve a real world purpose.

This course is targeted to middle school grades 6-8 (ages 11-14 years). It is also written for teachers who may not have a Computer Science background, or may be teaching an "Intro to Computer Science" course for the first time.

This course takes approximately 14 weeks to complete, spending about 1 week on each of the first 11 lessons, and 3 weeks for students to complete the final project at the end. Of course, teachers should feel free to customize the curriculum to meet individual school or district resources and timeframe.

Overall Course Scope & Sequence:

1. Making
2. Algorithms
3. Variables
4. Conditionals
5. Iteration
6. Review/Mini-Project
7. Coordinate Grid System

8. Booleans
9. Bits, Bytes, and Binary
10. Radio
11. Arrays
12. Independent Final Project

Each of the 12 lessons is comprised of the following parts:

- Topic Introduction
- Unplugged Activity (30 min) — An offline game or activity that demonstrates the concept/topic
- Micro:bit Activity (45-60 min) — An activity that everyone makes on their micro:bit that teaches the skills learned in this lesson.
- Project (60-120 min) — A prompt for an original project that each student will create to demonstrate their understanding of the skills and concepts covered in this lesson.
- Project Mods — Examples of additional things students can do to extend the project
- Assessment — A project rubric and guidance for grading the project.
- Standards — A list of CSTA K-12 Computer Science Standards and/or concepts covered by this lesson.

Topic Introduction

The introduction to each lesson will tell you what learning objectives are covered in the lesson, and presents an overview of that lesson's topic. Some lessons have a specific activity that can help introduce the topic to students in a fun way.

Unplugged Activity (30 min)

Each lesson starts with an unplugged activity, which doesn't require a computer or a micro:bit. It's a chance to get students up and moving around, and is designed to be a fun introduction to the computer science concept covered in that lesson. Unplugged activities are an important way to demonstrate new concepts in a tangible, often kinesthetic, way. Since so many computer-based topics are abstract, unplugged activities are very effective at fostering understanding that students will then demonstrate in later activities.

Micro:bit Activity (45–60 min)

Each lesson also contains a micro:bit activity, which we informally refer to as a "birdhouse" activity, after the innumerable wooden birdhouses so many of us made in wood shop as a way to master basic skills. Each lesson's micro:bit activity is an example that walks students step-by-step through building a project that demonstrates that lesson's topic. By the time students finish the activity, they will have written code that they can use in a different project of their own design.

Some students will finish the activity more quickly than others. Those students can then be a helpful resource for their classmates, or they can challenge themselves by modifying, or "modding" the activity to do something different. We have provided examples and suggestions at the end of many of these activities, and feel free to suggest your own (or encourage your students to come up with their own ideas!)

Project (60–120 min)

After presenting the concept in an unplugged fashion, then walking students through a demonstration activity, it is time to challenge students to use those skills to create something

that is creative and original. Students will be working on their projects in a "collaboratively independent" way, which means each student is responsible for turning in his or her own project, but are encouraged to work together and help each other while doing so. Some form of reflection is an important part of documenting the learning that has taken place, and it's a great idea to share out the final projects and reflections, either at an event or on a blog.

There are also a series of Project Mods that students can do to extend the project they have created. These are useful for students who already have some experience with coding or who want an extra challenge.

Assessment

A rubric is provided for each project that can be customized according to what students are being asked to demonstrate. For the Activities we just expect students to do them, so those are fairly simple to check off. For the Projects, however, there is often a range of grades based on how closely the project meets the specifications of the assignment.

Standards

Where applicable, we have mapped each of the lessons to the Computer Science Teachers Association (CSTA) K-12 Standards, which are US nationally recognized standards for computer science education.

References



We have included some additional reference books and materials if you are interested in learning more about Maker Education, Physical Computing or Design Thinking in the classroom.

- [Invent To Learn](#)
Making, Tinkering, and Engineering in the Classroom
By Sylvia Libow Martinez & Gary Stager
- [Launch](#)
Using Design Thinking to Boost Creativity and Bring Out the Maker in Every Student
by John Spencer and AJ Juliani
- [The Innovator's Mindset](#)
Empower Learning, Unleash Talent, and Lead a Culture of Creativity
by George Couros
- [The Big Book of Makerspace Projects](#)
Inspiring Makers to Experiment, Create, and Learn
by Colleen Graves

If you have feedback for the Microsoft MakeCode team, you can fill our their survey form here: <https://aka.ms/microbitfeedback>

The support site for the micro:bit is located here: <https://support.microbit.org/>

About The Authors



Douglas Kiang is a speaker, teacher, and workshop presenter with twenty-seven years of teaching experience in independent schools at every grade level. He currently teaches high school computer science at Punahou School in Honolulu, Hawaii. Douglas holds a master's degree in Technology, Innovation, and Education from Harvard and is a Microsoft Innovative Educator.

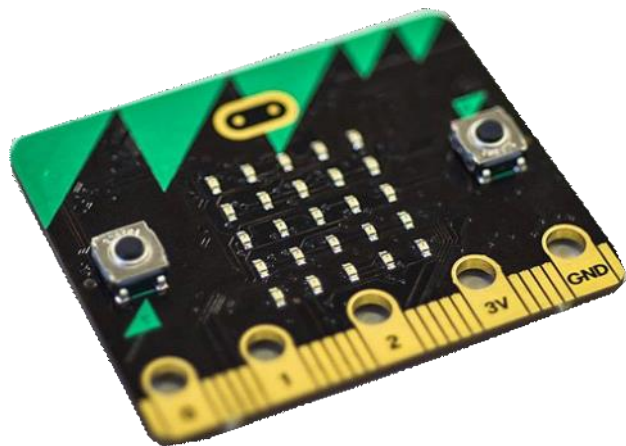
You can follow him on Twitter at [@dkiang](https://twitter.com/dkiang).



Mary Kiang has been teaching for over twenty-five years at elementary, middle, and high school levels. She also developed curriculum in the Education Department of the Museum of Science in Boston. She currently teaches 6th grade Math/Science at Punahou School. Mary is a former programmer for Houghton Mifflin and Dun & Bradstreet and holds a Master's degree in Elementary Education from Simmons College. Mary is the founder of GO Code!, an organization that supports girls and young women in exploring coding and STEM.

Making with Micro:bit

This Lesson introduces the Micro:bit as a piece of hardware that has a specific size and weight, and generally must be supported and incorporated as an essential component of a tangible artifact. Focus on incorporating the physical Micro:bit into a basic making activity.



Lesson Objectives

Students will...

- Exercise creativity and resourcefulness by coming up with ideas for using simple household materials to accommodate the micro:bit's size and weight in many different ways.
- Test and iterate using different materials and sizes in order to create an optimal design to house the micro:bit and battery pack
- Learn how to download programs and move them to the Micro:bit file to run on the Micro:bit.
- Use the design thinking process to develop an understanding for a problem or user need.
- Apply their understanding in a creative way by making a "micro:pet" creature.

Lesson Plan Structure

- Introduction: The Micro:bit is for making
- Unplugged Activity: Design Thinking
- Micro:bit Activity: MakeCode download
- Project: Micro:pet
- Project Mods
- Assessment: Rubric
- Standards: listed

Introduction

The Micro:bit is a great way to teach the basics of programming and computer science. The Microsoft MakeCode block-based coding environment is a powerful and intuitive way to make the Micro:bit react to all sorts of input, and you can introduce fundamental concepts such as iteration, conditional statements, and variables using MakeCode.

Students often focus primarily on the 5x5 LED screen for providing output. Although this is the most directly accessible way to see a reaction to some kind of input, there are many more creative possibilities when you encourage your students to see the micro:bit as a "brain" that can control physical, tangible creations.

These creations don't have to be complex or highly technical. It's great to have students building with common household supplies. Because the micro:bit is so lightweight, and supports so many sensors, it can be incorporated easily into a physical design as long as students plan ahead for its size and weight. One of the first questions you might ask students is "Where does the micro:bit fit in your creation?"

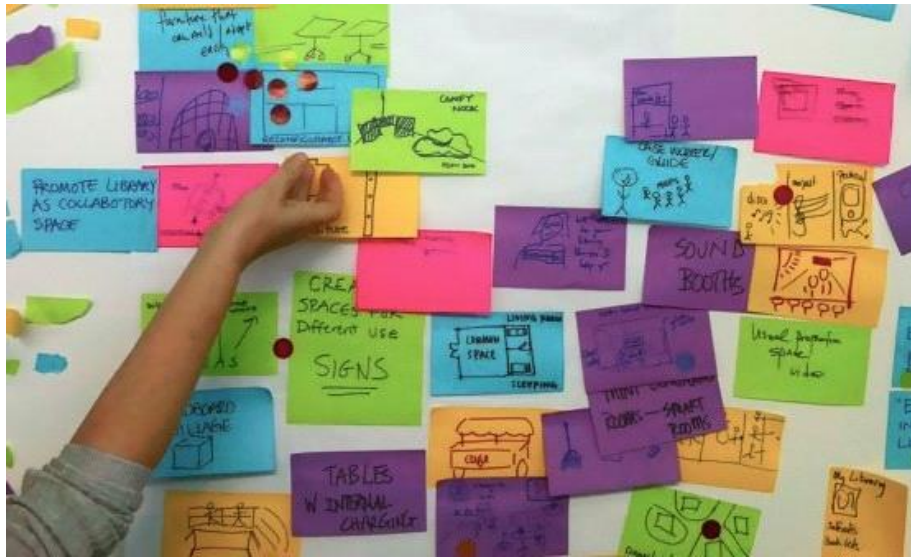
In this first lesson's project, we focus on making something creative that features the micro:bit as its "face". We purposely start this course with a lesson on Making and the physical nature of the micro:bit, because it is important to set the tone for the whole course that this is a class about making, building, crafting and construction. It helps if you have an art room available where kids can work, or arts and crafts supplies in your classroom that kids can use to build.

Some common making supplies to gather:

- pizza boxes
- scrap cardboard
- colored construction paper
- colored duct tape
- scissors
- pipe cleaners
- stickers
- feathers
- string
- markers



Unplugged: Design Thinking



Objective: To introduce a process of design that starts with talking to one another. Whatever you build with code should serve a purpose or fill a need. Sometimes what you build will make the world more beautiful, or help somebody else. Our design process, based on a process called *design thinking*, can give students a specific framework for thinking purposefully about design.

Overview: In this activity, students will interview each other about their ideal pet. They should take notes.

The first step in coding by design involves understanding someone else's need. Then, you can create prototypes that get you closer and closer to the best solution.

Materials:

Pairs of students, something to take notes on

Getting started:

Pair students up with each other. One is Student A, the other is Student B. The goal of this activity is to gather information from their partner that will help them to design a Micro:bit pet for their partner.

5 minutes: Student A interviews Student B. The goal is to find out what Student B considers to be their ideal pet. Student A should mostly listen, and ask questions to keep Student B talking for the entire time. Here are some questions to start with:

- Do you have a pet? What is it?
- What do you like about your pet? What do you dislike?
- Is there anything you wish your pet could do? Why?
- Tell me about your ideal pet.

5 minutes: Student B interviews Student A, as above.

The goal is to find out more about your partner by asking questions. Try to ask "Why?" as much as possible. Your partner will tell you about his or her ideal pet, but you are really finding out more about your partner's likes and dislikes. When we design, we create real things for real people. So we need to start with understanding them first.

5 minutes: Student A and Student B review their notes, and circle anything that seems as if it will be important to understanding how to create the ideal pet for their partner. Circle ideas, advice, anything that could be helpful when they start building. Then, they should use what they have discovered about their partner to fill in the blanks:

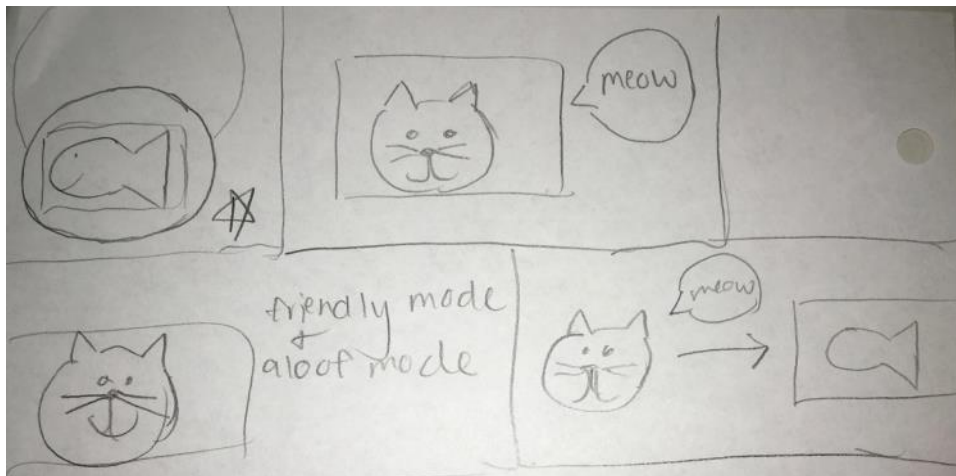
"My partner needs a _____ because _____."

This definition statement should draw some conclusions about their partner's need based on the conversation they have had with that person.

5 minutes: Student A and Student B sketch at least 5 ideas of pets that would meet their partner's needs. Stick figures and diagrams are okay. At this point, quantity is more important than quality. Students shouldn't limit themselves to real animals; unicorns and mashups are totally fine!

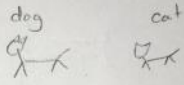
Make sure students keep their notes and sketches! They will use them in the project for this lesson.

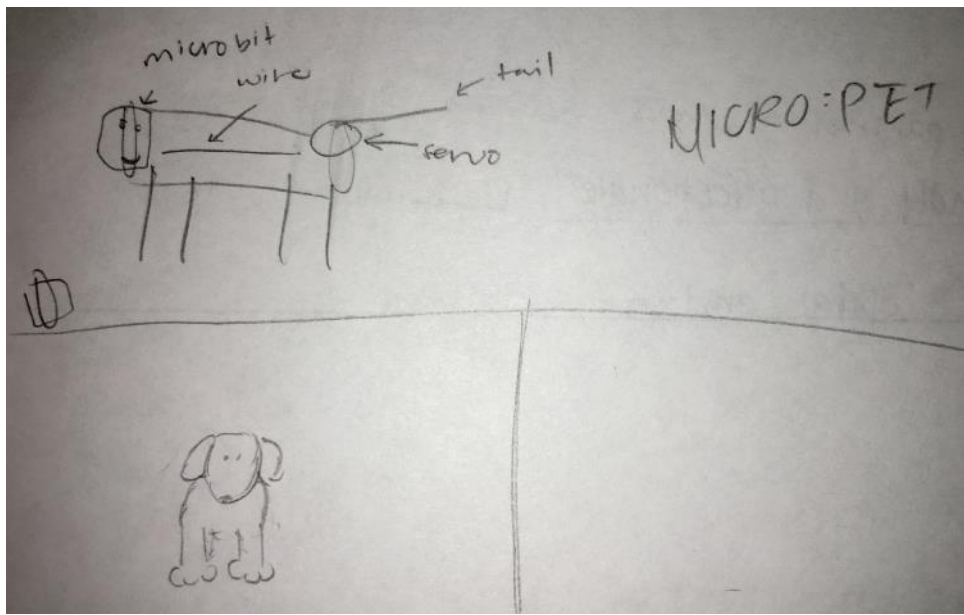
Examples

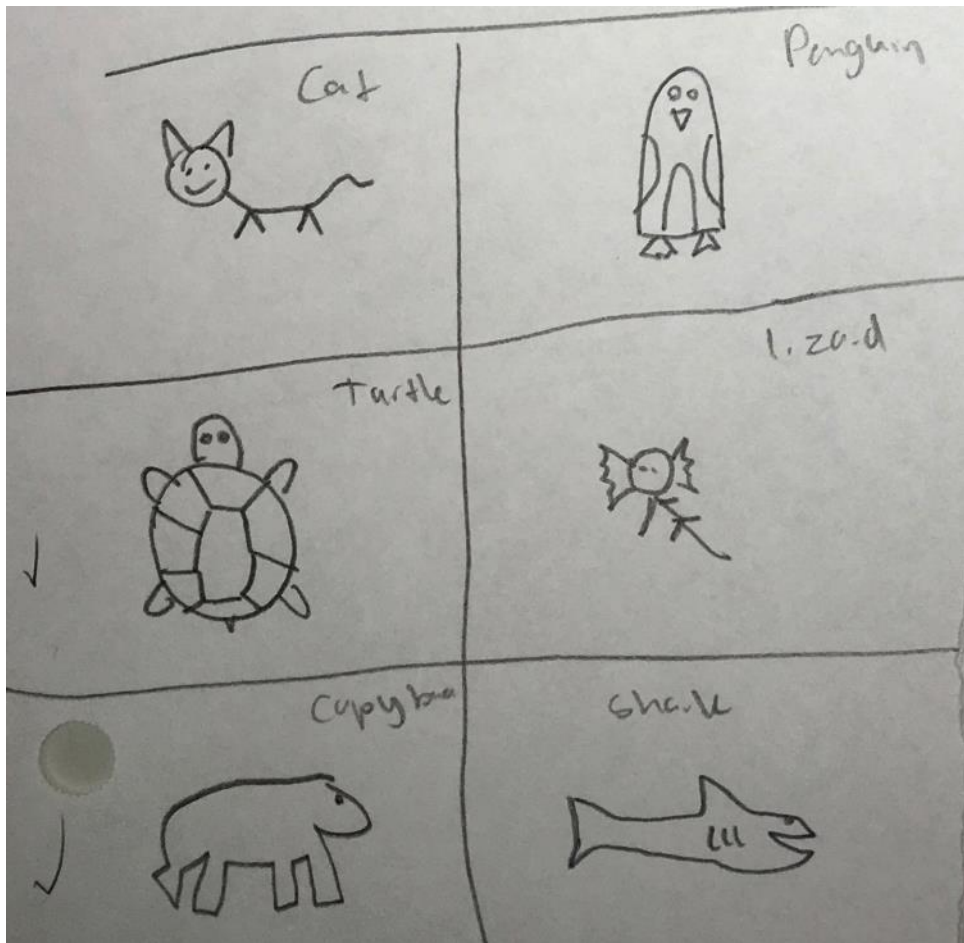


- Silky Terrier
 Likes { - Playful and affectionate
 - Can interact with it
 - Trainable
 Dislikes { - Has anxiety
 - Is needy
 - Wish it could talk
 - Wish it didn't have anxiety
 Ideal { - German Shepherd
 - Bigger dog that can keep up
 - Multifunctional
 - Affectionate
 - Guard Dog

My partner needs a pet that is well rounded because she values affection and safety.

dog cat






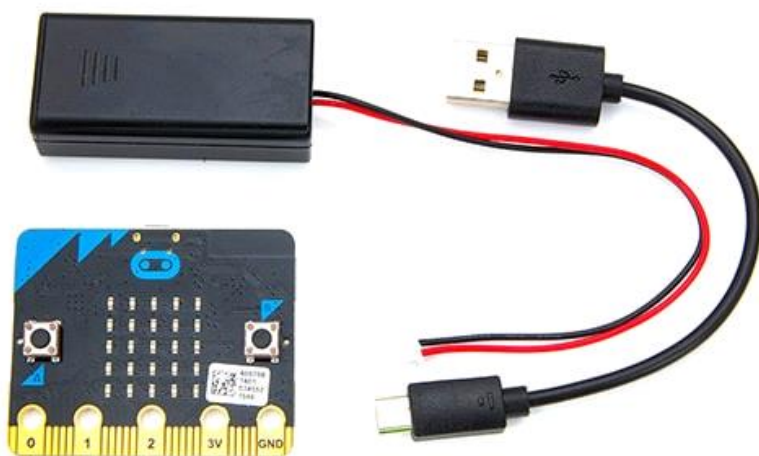
Activity: Installing a Program

Micro:bit Activity: Installing a Microsoft MakeCode Program on the Micro:bit

Objective: Learn how to download programs from the MakeCode tool.

Overview: Students will create a simple program in Microsoft MakeCode and download it to their Micro:bit using a USB cable.

For this activity, students will each need a Micro:bit, a micro-USB cable, a computer, and a battery pack.

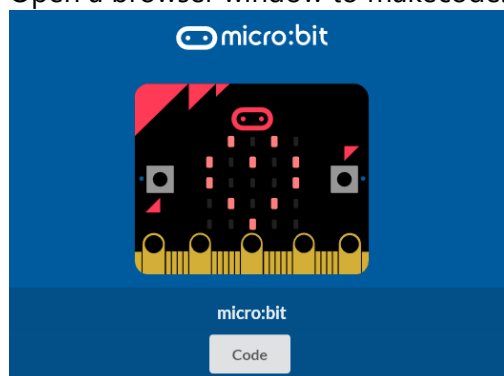


Download this file to your computer (right-click, Save As):



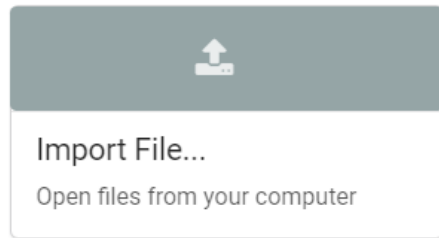
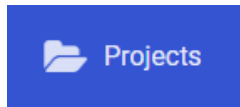
microbit-MicroPet

Open a browser window to makecode.com, and select the micro:bit code editor



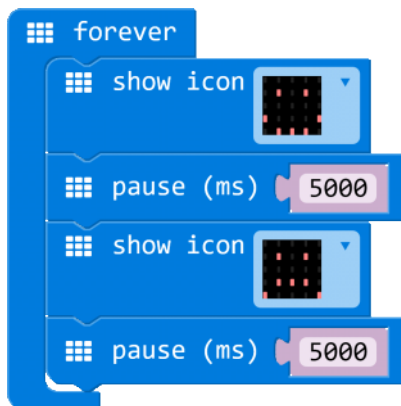
From the top left corner of the screen, select the **Projects** Menu, and click on **Import File**.

Select the file that you saved on your computer in the previous step.



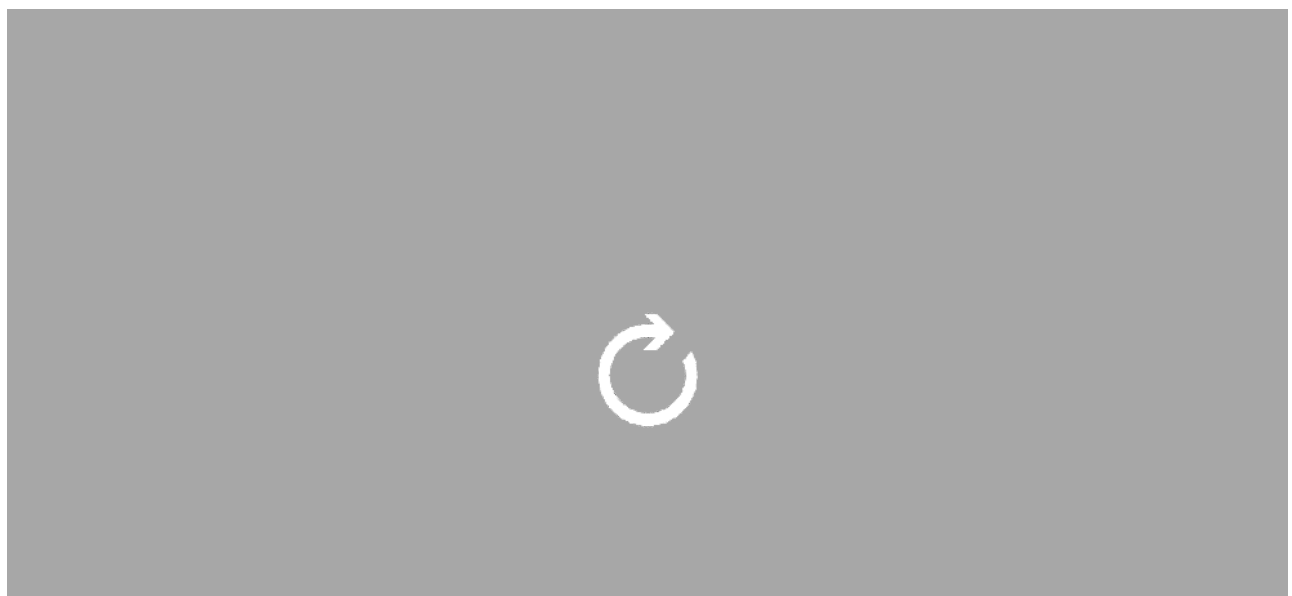
The program should look like the following in MakeCode.

It shows a repeating series of faces:



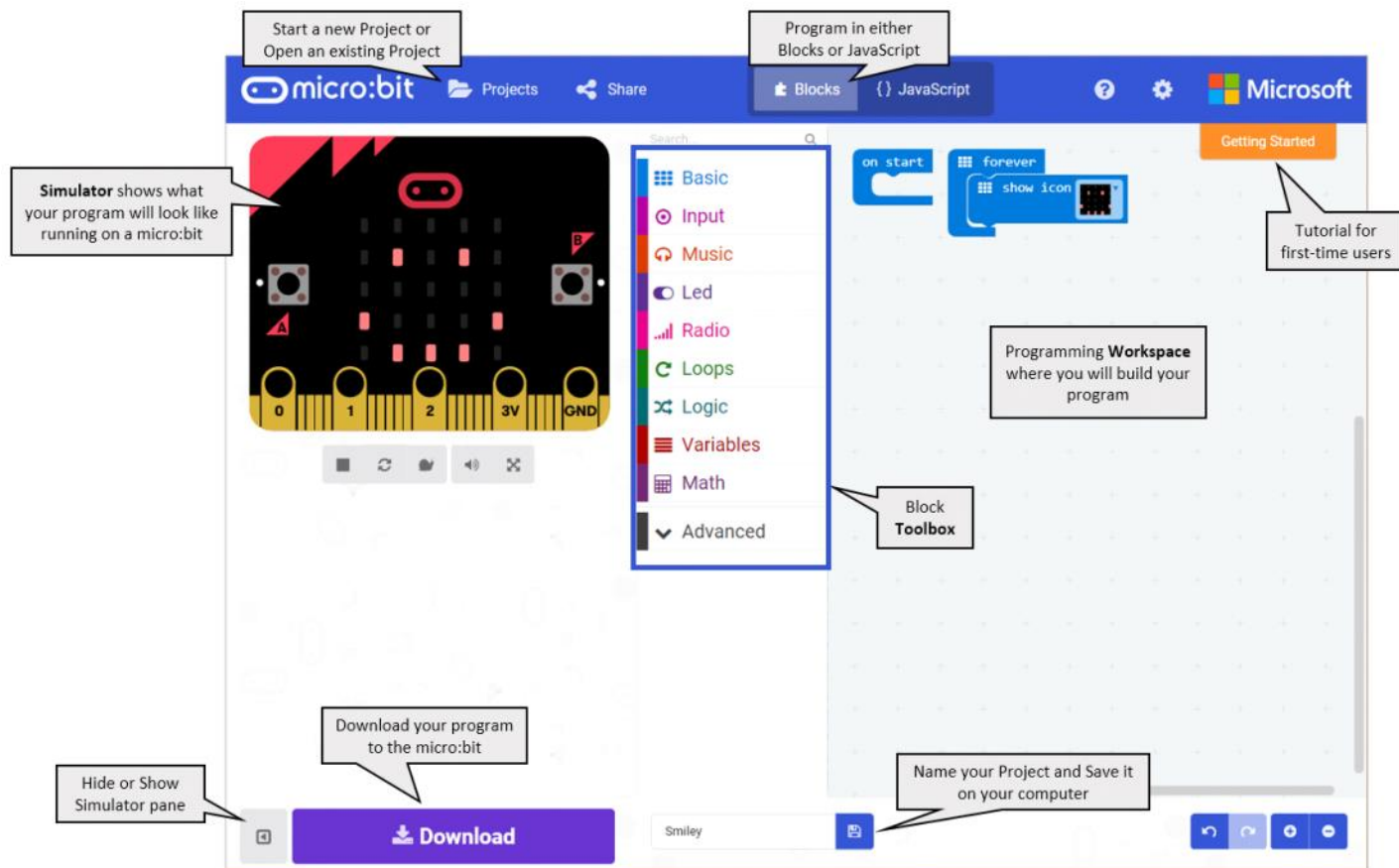
```
basic.forever(() => {  
  basic.showIcon(IconNames.Happy)  
  basic.pause(5000)  
  basic.showIcon(IconNames.Sad)  
  basic.pause(5000)  
})
```

[MicroPet](#)



Tour of Microsoft MakeCode

- Simulator - on the left side of the screen, you will see a virtual micro:bit that will show what your program will look like running on a micro:bit. This is helpful for debugging, and instant feedback on program execution.
- Toolbox - in the middle of the screen, there are a number of different categories, each containing a number of blocks that can be dragged into the programming workspace on the right.
- Workspace - on the right side of the screen is the Programming Workspace where you will create your program. Programs are constructed by snapping blocks together in this area.



The color of the blocks identifies their category. All of the blocks that make up the program above come from the **Basic** Toolbox category, which is light blue.

Downloading a MakeCode Program to the micro:bit

To download the file to your micro:bit you must connect it to your computer's USB port using a micro-USB cable. The micro:bit will draw power from your computer through the USB connection, or you can connect an optional battery pack so it can function even after it is unplugged from the computer. Once plugged in, the micro:bit shows up on your computer like

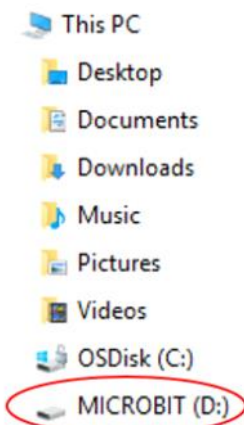
a USB flash drive.



Click the purple Download button in the lower left of the MakeCode screen. This will download the file to your computer, to the location where your browser is set to save downloads.

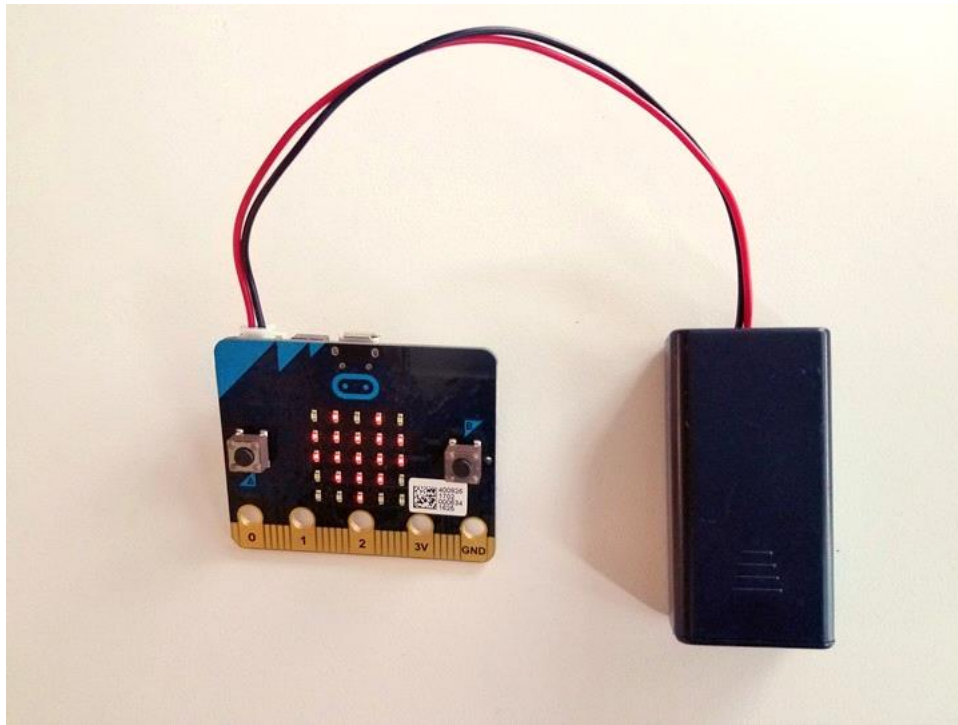


To move the program to your micro:bit, drag the downloaded "microbit-xxx.hex" file to the MICROBIT drive, as if you were copying a file to a flash drive. The program will copy over, and it will begin running on the micro:bit immediately.



The micro:bit will hold one program at a time. It is not necessary to delete files off the micro:bit before you copy another onto the micro:bit; a new file will just replace the old one.

For the next project, your students should attach the battery pack (it takes 2 AAA batteries) to the micro:bit using the white connector. That way they can build it into their design without having to connect it to the computer.



Project: Micro:pet

Project

This project is an opportunity for students to create a micro:pet for the partner they interviewed in the Unplugged activity. They should review their notes and try to summarize what their partner finds appealing in a pet. Then, they should use whatever materials are available to create a prototype of a pet their partner would like.

We often ask students to sketch a few designs on paper first, then consult with their partner to see which aspects of those designs they find most appealing. The purpose of prototyping is to gather more feedback to help you in your final design (“I like this part from Idea A, and I like this part from Idea B...”)

Build a micro:pet that:

- Matches your partner’s needs
- Supports the micro:bit and its battery pack
- Allows you to easily access the micro:bit to turn it on and off

Your design should use whatever materials are available to support the micro:bit so that its face is showing. You can be creative and decide how to mount the board, and how to decorate your critter.

Think about the following questions when you construct it:

- Will it be an animal? A plant? A robot? A bug?
- Will it have any moving parts?
- If it moves, how can you hold the micro:bit securely?

Some photos of sample micro:pets below!

Ideas for Mods

- Find a way to make part of the animal move.
- Give your animal a natural habitat.
- Create a way to carry your animal.
- Create an animal that reacts when you pet it or move it (find a way to detect when the micro:bit is moved or when its position changes in a certain way.)

Reflection

Have students write a reflection of about 150–300 words, addressing the following points:

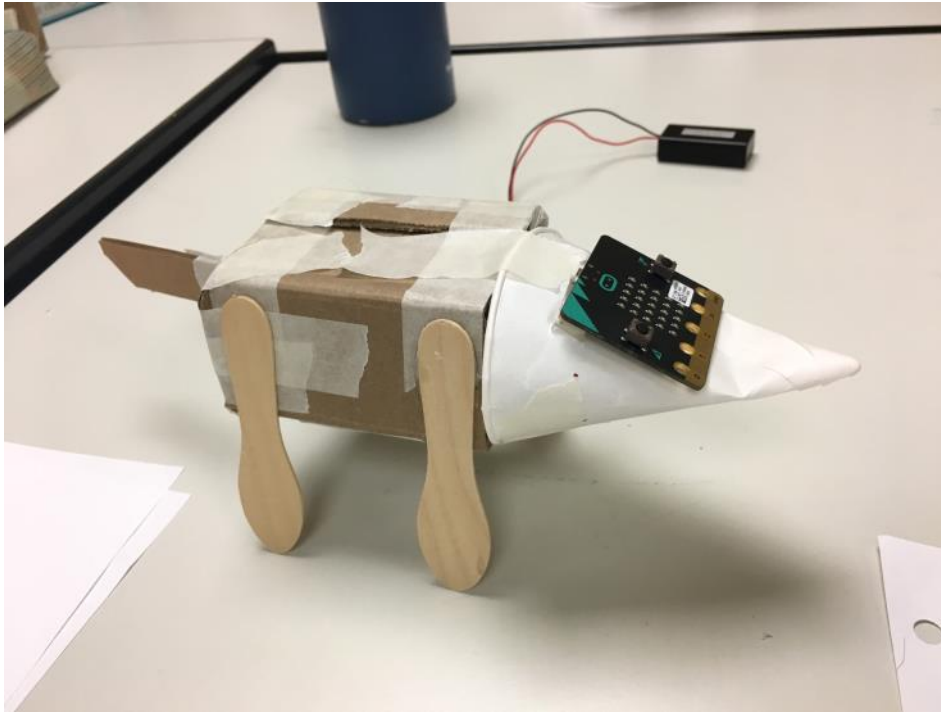
- Summarize the feedback you got from your partner on your idea. How would you revise your design, if you were to go back and create another version?
- What was it like to have someone designing a pet for you? Was it a pet you would have enjoyed? Why or why not? What advice did you give them that might help them redesign?
- What was it like to interview your partner? What was it like to be listened to?
- What was something that was surprising to you about the process of designing the micro:pet?
- Describe a difficult point in the process of designing the micro:pet, and explain how you resolved it.

Rubric

For creative projects such as these, we normally don’t use a qualitative rubric to grade the creativity or the match with their partner’s needs. We just check to make sure that the micro:pet meets the required specifications:

- Program properly downloaded to micro:bit
- Micro:bit supported so the face is showing
- Micro:bit can be turned on and off without taking critter apart
- Turned in notes on interview process
- Written reflection (prompt is above)

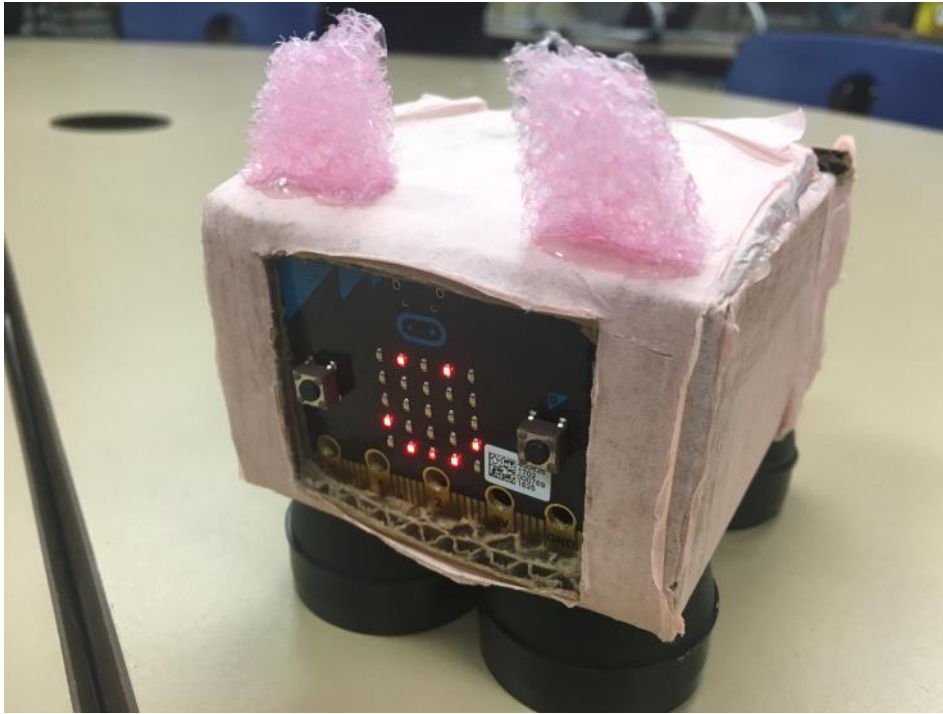
Micro:Pet Examples



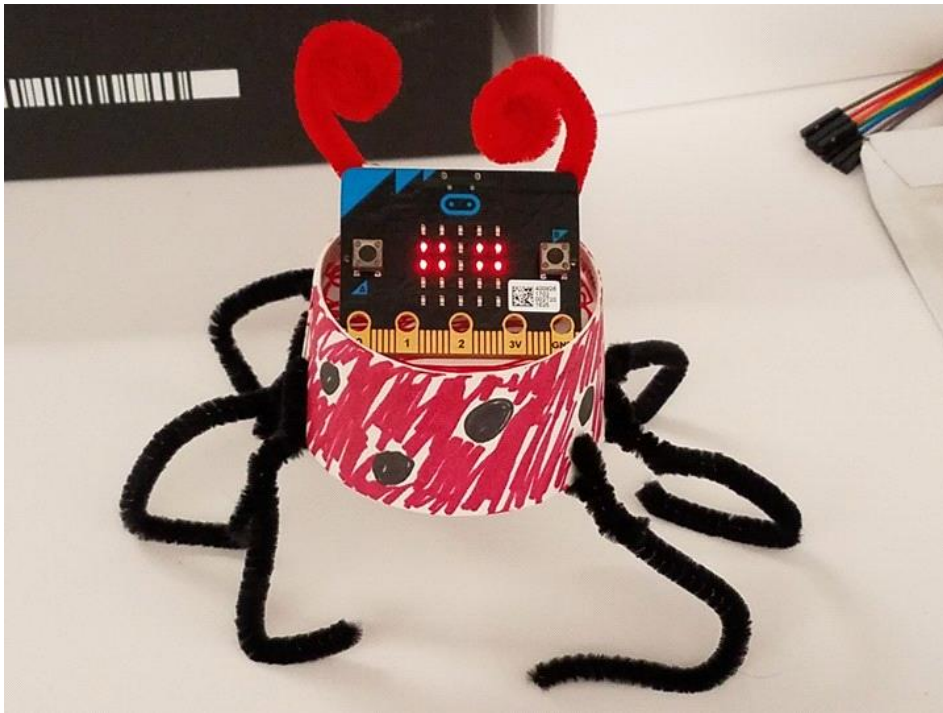
Dog

[micro:pet Fish Tank](#)

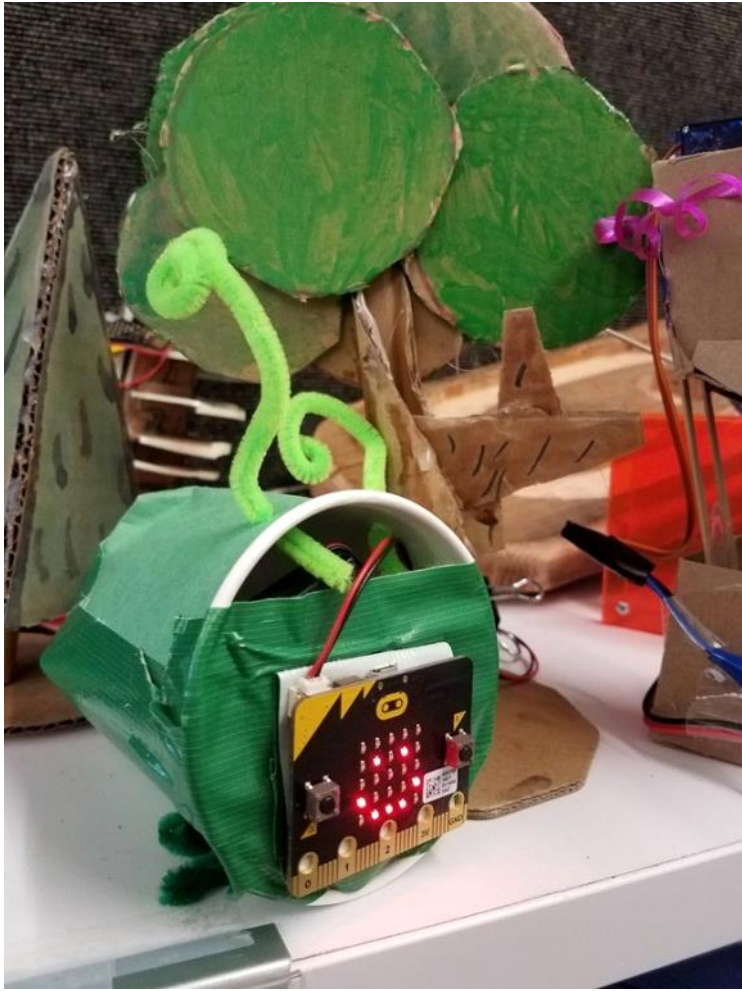




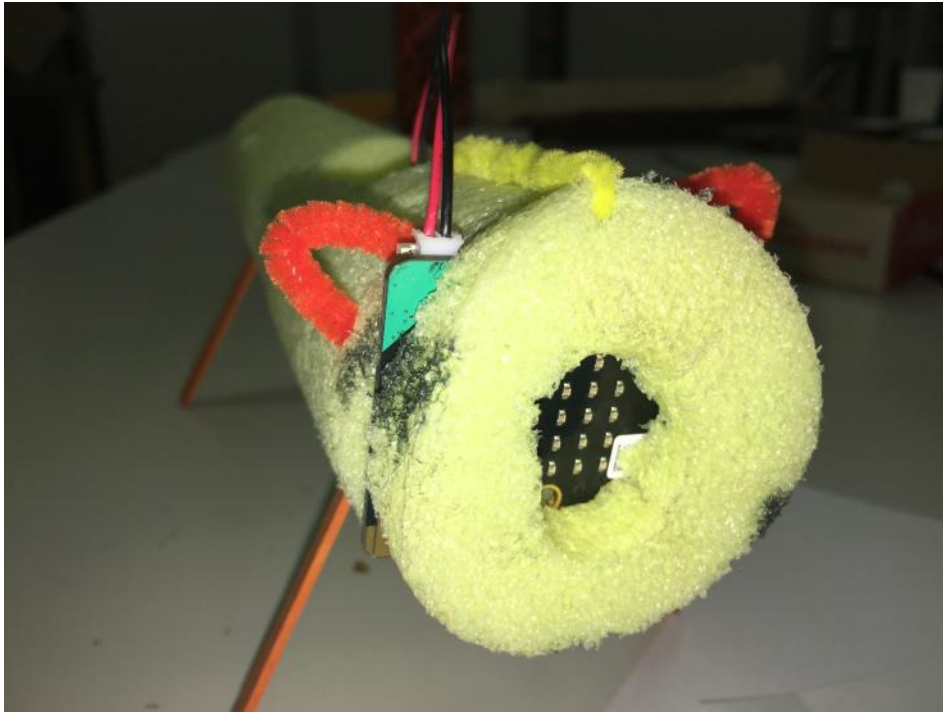
Pink Piggy



Ladybug



Caterpillar



Fox



Robot

Standards

CSTA K-12 Computer Science Standards

- 2-A-2-1 Solicit and integrate peer feedback as appropriate to develop or refine a program
- 2-A-6-10 Use an iterative design process (e.g., define the problem, generate ideas, build, test, and improve solutions) to solve problems, both independently and collaboratively.

Algorithms

This lesson introduces a conceptual framework for thinking of a computing device as something that uses code to process one or more inputs and send them to an output(s).

Lesson Objectives

Students will...

- Understand the four components that make up a computer and their functions.
- Understand that the micro:bit takes input, and after processing the input, produces output.
- Learn the variety of different types of information the micro:bit takes in as input.
- Apply this knowledge by creating a micro:bit program that takes input and produces an output.

Introduction

What is a micro:bit?

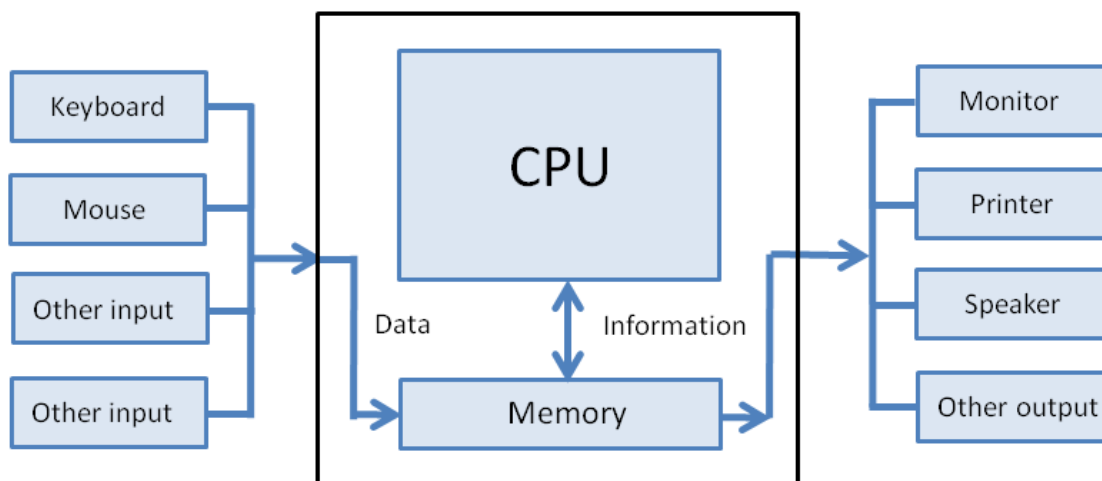
The micro:bit was created in 2015 in the UK by the BBC to teach computer science to students. The BBC gave away a micro:bit to every Year 7 student in the UK. You can think of a micro:bit as a mini computer.

<http://microbit.org>



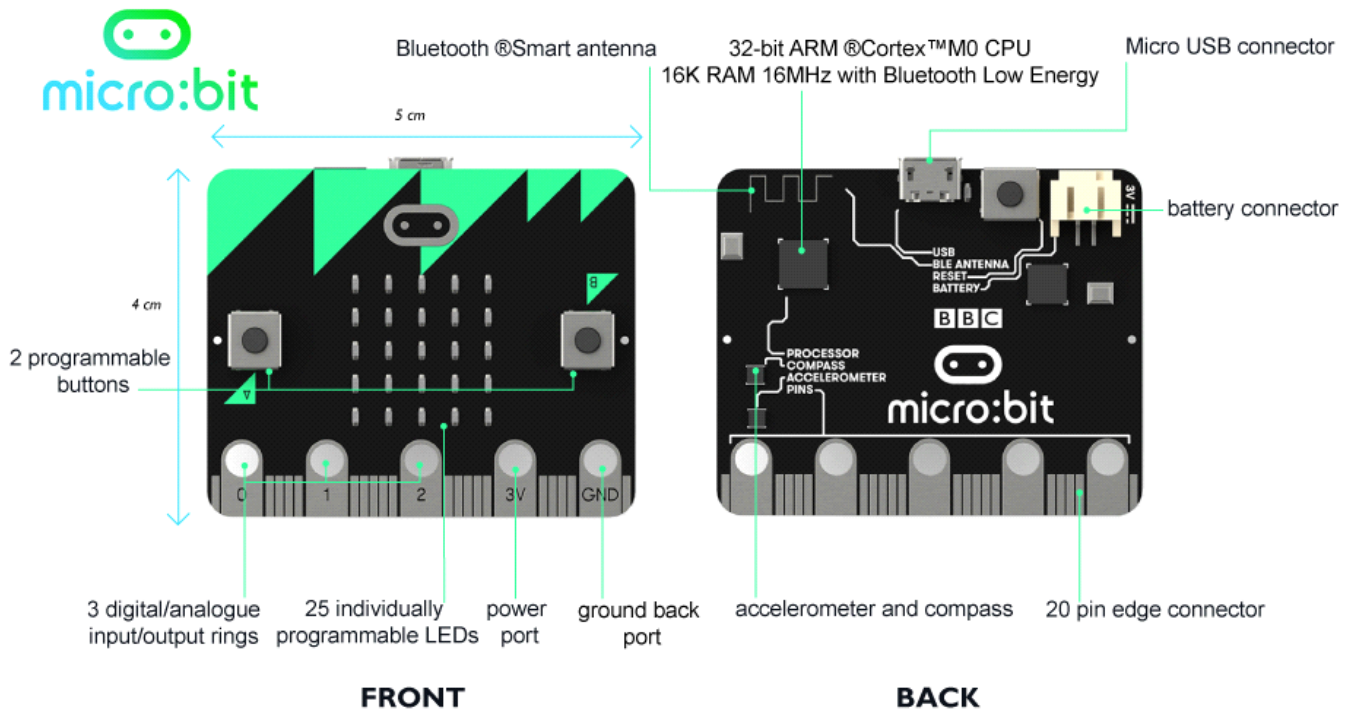
What is a computer?

There are 4 main components that make up any computer:



1. The Processor – this is usually a small chip inside the computer, and it's how the computer processes and transforms information. Has anyone heard of the term "CPU"? CPU stands for Central Processing Unit. You can think of the processor as the Brains of the computer - the faster the processor, the more quickly the computer can think.
2. The Memory – this is how the computer remembers things. There are two types of memory:
 - o RAM (random access memory) - you can think of this as the computer's short-term memory
 - o Storage (also referred to as the "hard drive") - this is the computer's long-term memory, where it can store information even when power is turned off
3. Inputs – this is how a computer takes in information from the world. On humans, our input comes in through our senses, such as our ears and eyes. What are some Computer Inputs? *Keyboard, Mouse, Touchscreen, Camera, Microphone, Game Controller, Scanner*
4. Outputs – this is how a computer displays or communicates information. On humans, we communicate information by using our mouths when we talk. What are some examples of communication that don't involve talking? *Blushing, sign language.* What are some examples of Computer outputs? *Monitor/Screen, Headphones/Speakers, Printer*

Now, let's look at our micro:bit:



- Use the diagram here as a visual aid: <http://microbit.org/hardware/>
- Can you find the Processor?
- How much memory does the micro:bit have? *16K, which is smaller than many files on your computer!*
- Can you locate the following Inputs? *Buttons (on board), Pins (at base), Accelerometer / Compass*
 - o *Though not pictured, the Light Sensor is located on the LED lights*
- Where are the Outputs? *LED lights, Pins*

All computers need electricity to power them. There are 3 ways to power your micro:bit:

- Through the USB port at the top

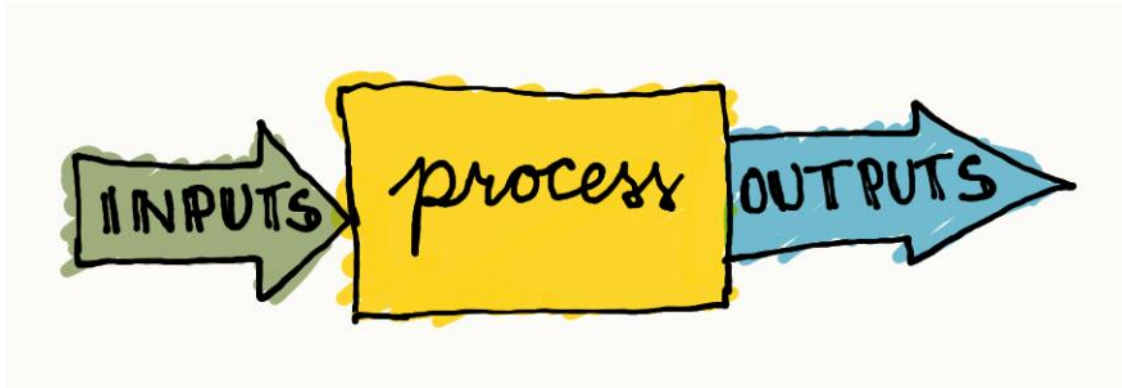
- By connecting a battery pack to the battery connector
- Through the 3V Pin at the bottom (not the recommended way to power your micro:bit)

On the top left corner you may notice that your micro:bit has a Bluetooth antenna. This means your micro:bit can communicate and send information to other micro:bits. We will learn more about this feature in the Radio Lesson.

Unplugged: What's Your Function & Crazy Conditionals

Materials

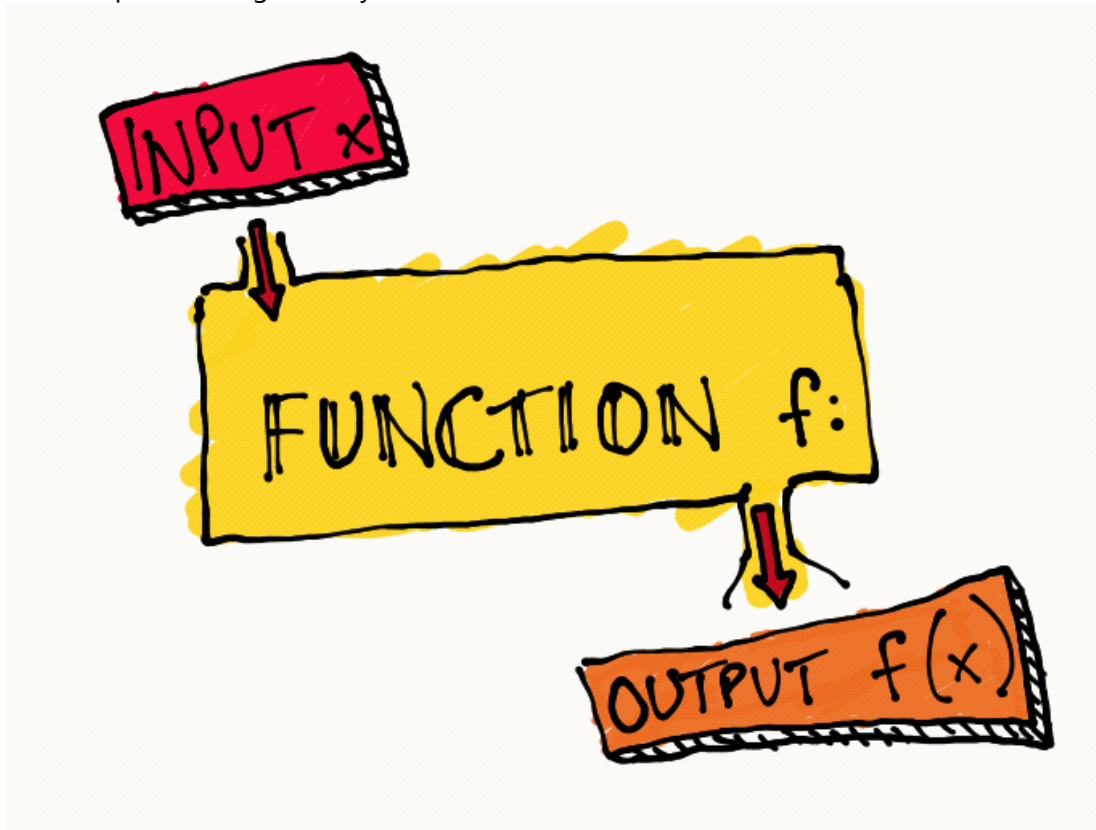
- Pencils
- Paper (or index cards)



In computer programming, algorithms are sets of instructions.

Algorithms 'tell' the computer how to process input and what, if any, output to produce.

An example of an algorithm you have seen in math class is the 'function machine'.



A function machine takes an input, processes the input, and then delivers an output.

The inputs and their outputs are usually recorded in an input output table, where the value of x represents the

input and the value of **y** represents the output. *See example.*

Input (x)	Output (y)
1	2
2	4
3	6
4	8

A common math problem is to determine what processing is happening to the input that results in the given output. In the example above, each input is being doubled (multiplied by 2) to produce the corresponding output.

Input (x)	Processing =>	Output (y)
1	* 2	2
2	* 2	4
3	* 2	6
4	* 2	8

Unplugged: What's Your Function?

For this activity, the students can work in pairs, Player A and Player B. The pairs will take turns being the function machine for their partner who will be providing input to be processed.

Direct the students how you would like them to record their work.

They can use pencil and paper or index cards.

On paper, they can keep track of inputs and outputs in a table (see example above).

With index cards, Player A can write each input on one side of an index card, hand the card to Player B, who then writes the corresponding output on the other side of the card.

To begin:

- Player B decides on a mathematical function or bit of processing* that will be done on whatever input she receives from Player A.
- Player B should write down the function or bit of processing and set it aside, out of sight of Player A.
- Player A then gives Player B a number to process.
- Player B processes the number and returns an output to Player A.
- Player A can then state what function or bit of processing she thinks Player B is using on the input to produce the given output. One try per round of input/output.
- If Player A states the correct function, Player B confirms that it is correct by showing the previously hidden function and the players switch roles and start the game over.
- If Player A does not guess correctly, Player A provides another input that Player B processes and provides an output for.
- The goal is for Player A to figure out what function or bit of processing Player B is using in the fewest number of rounds of input/output possible.
- After each student has had at least one chance to be the function machine, play more rounds as time permits.

Notes:

- The difficulty level of the possible functions should be determined by the teacher and shared with the students ahead of playing. Alternately, the teacher can provide function cards that are handed out at random to be used by the players, rather than the players creating their own.

- The player providing the input should not just guess what the function is. She should be able to explain why she thinks her input resulted in the given output.
- Examples of 'easier' functions:
 - Add 8
 - Subtract 6
 - Multiply by 3
 - Divide by 2
- Examples of more difficult functions:
 - Multiply by 2 and then subtract 1
 - Square the input
 - Return 20% of the input

Unplugged: Crazy Conditionals

This is a fun, interactive exercise to introduce conditionals and event handlers as computer processing. Read through the entire activity and adjust as needed for your class and classroom.

Preparation:

- Print & cut into strips with one conditional on each strip
- Note that some of the same conditionals can be given to multiple students, while other conditionals are to be given to just one student.
- Except for the first 'BEGIN' conditional, hand out the conditionals PRINT SIDE DOWN.
- Besides the 'BEGIN' and 'STOP' conditional, give at least two other conditionals to each student. A lesson from this is that it is challenging for a student to keep track of a lot of different conditionals, though not so for a computer! :)

Notes:

- Some of the same conditionals can be given to multiple students, while other conditionals are to be given to just one student.
- Technically these conditionals are all event handlers because the students are simply waiting for a specific event to trigger them into action.
- Unless instructed otherwise, students do not speak or make noise during this activity.

Extensions/Variations:

- Add AND, OR, AND/OR statements to the conditionals.
- Create nested IF's
- Let students create the IF's
- Relate this activity to a system and have the students create the conditionals that would end in a product of some kind or the completion of some task.

Give these 2 conditionals to all students.

- These 2 conditionals will be triggered only once.
- These conditionals start and stop this activity.
- Give the first 'BEGIN' conditional to the students PRINT SIDE UP.

IF the teacher writes the word '**BEGIN**' on the whiteboard,
THEN flip over the conditionals in front of you and follow the directions.

IF you see the word '**STOP**' on the whiteboard,
THEN sit back, cross your arms, and look at the teacher (smile!).

=====

Give these 6 conditionals to multiple students.

- These 6 conditionals may be triggered more than once.
- Walk around the classroom during the activity to trigger some of these conditionals.

IF the teacher says the word 'popcorn',
THEN stand up and say 'Pop!' once and sit back down.

IF any student stands up for any reason,
THEN clap 3 times.

IF anyone writes on the whiteboard with a GREEN marker,
THEN get up and touch something GREEN in the room and sit back down.

IF anyone walks past you while you are seated,
THEN snap your fingers 3 times.

IF someone snaps their fingers **AND** you have the letter 'e' in your first name,
THEN select a book from the bookcase and sit back down.

IF anyone writes anything on the whiteboard,
THEN get up and turn around in place one full turn and sit back down.

=====

Give one student each of the following 7 conditionals.

- These 7 conditionals will be triggered only once and set in motion the spelling of STOP on the whiteboard.

IF the teacher picks up a book,
THEN get up and write the letter S on the whiteboard and sit back down.

IF someone writes the letter S on the whiteboard,
THEN go open and close the classroom door and sit back down.

IF someone opens and closes the classroom door,
THEN get up and write the letter T (after the letter S) on the whiteboard.

IF someone writes the letter T on the whiteboard,
THEN get up and turn the lights on and off and sit back down.

IF someone turns on and off the lights,
THEN get up and write the letter O (after the letter T) on the whiteboard.

IF someone writes the letter O on the whiteboard,
THEN get up and sharpen a pencil.

IF someone sharpens a pencil,
THEN get up and write the letter P (after the letter O) on the whiteboard.

Activity: Happy Face, Sad Face

The micro:bit itself is considered hardware. It is a physical piece of technology. In order to make use of hardware, we need to write software (otherwise known as "code" or computer programs). The software "tells" the hardware what to do, and in what order to do it using algorithms. Algorithms are sets of computer instructions.

In this activity, we will discover how to use the micro:bit buttons as input devices, and write code that will make something happen on the screen as output. We will also learn about pseudocode, the MakeCode tool, event handlers, and commenting code.

Pseudocode

What do you want your program to do?

The first step in writing a computer program is to create a plan for what you want your program to do. Write out a detailed step-by-step plan for your program. Your plan should include what type of information your program will receive, how this input will be processed, what output your program will create and how the output will be recorded or presented. Your writing does not need to be written in complete sentences, nor include actual code. This kind of detailed writing is known as *pseudocode*. Pseudocode is like a detailed outline or rough draft of your program. Pseudocode is a mix of natural language and code.

For the program we will write, the pseudocode might look like this:

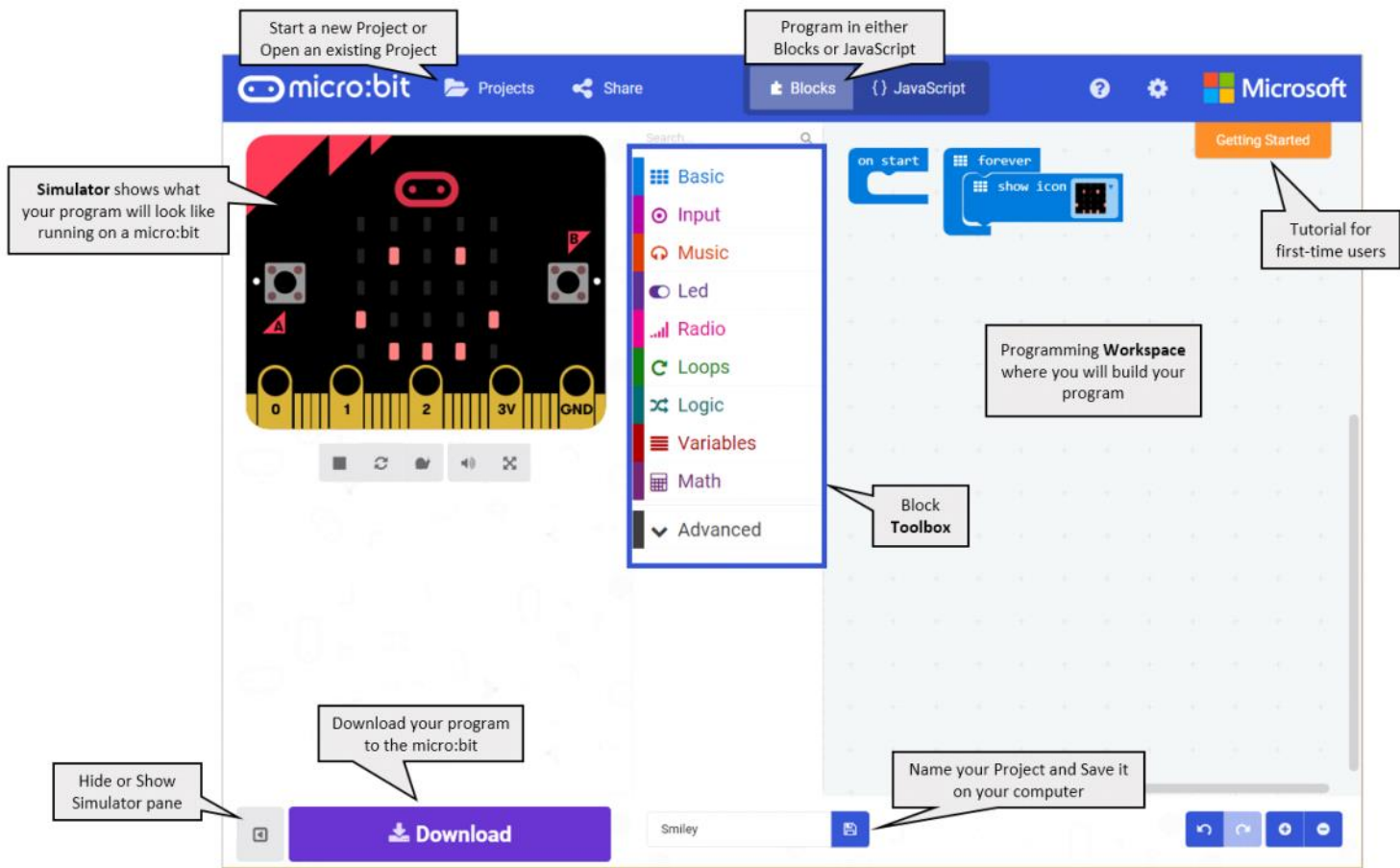
- Start with a blank screen
- Whenever the user presses button A, display a happy face.
- Whenever the user presses button B, display a sad face.

Microsoft MakeCode

Now that you have a plan for your program, in the form of pseudocode, let's start creating the real program. In a browser window, open the Microsoft MakeCode for micro:bit tool (<https://makecode.microbit.org>). The MakeCode tool is called an *IDE* (Integrated Development Environment), and is a software application that contains everything a programmer needs to create, compile, run, test, and even debug a program.

Tour of Microsoft MakeCode

- Simulator - on the left side of the screen, you will see a virtual micro:bit that will show what your program will look like running on a micro:bit. This is helpful for debugging, and instant feedback on program execution.
- Toolbox - in the middle of the screen, there are a number of different categories, each containing a number of blocks that can be dragged into the programming workspace on the right.
- Workspace - on the right side of the screen is the Programming Workspace where you will create your program. Programs are constructed by snapping blocks together in this area.



Event handlers

When you start a new project, there will be two blue blocks, 'on start' and 'forever' already in the coding workspace. These two blocks are *event handlers*.

In programming, an *event* is an action done by the user, such as pressing a key or clicking a mouse button. An *event handler* is a routine that responds to an event. A programmer can write code telling the computer what to do when an event occurs.

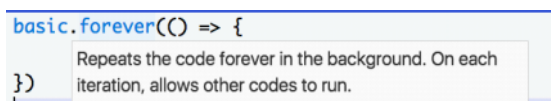
One fun unplugged activity you can do with kids to reinforce the idea of an action that waits for an event is the Crazy Conditionals activity.

Notes:

- Tooltips - Hover over any block until a hand icon appears and a small text box will pop up telling you what that block does. You can try this now with the 'on start' and 'forever' blocks. Notice that it also shows you the equivalent code in JavaScript.

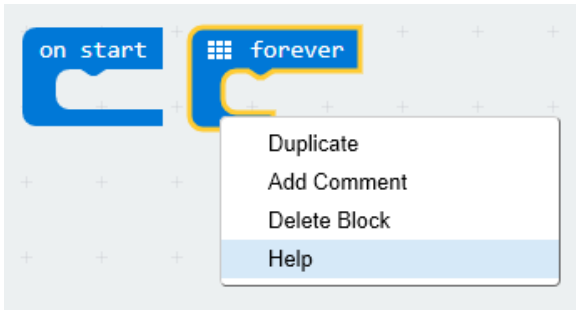


Hovering over the code in JavaScript has the same effect.

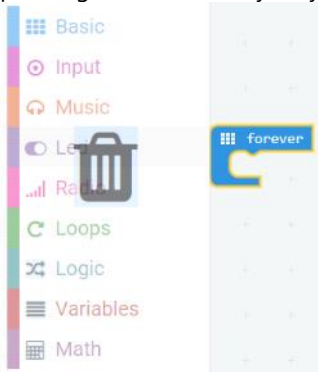


- Help/Documentation - You can also right-click on any block and select Help to open the

reference documentation.

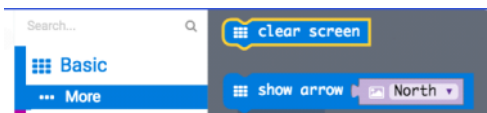


- Deleting blocks - Click on the 'forever' block and drag it left to the Toolbox area. You should see a garbage can icon appear. Let go of the block and it should disappear. You can drag any block back to the Toolbox area to delete it from the coding workspace. You can also remove a block from your coding window by selecting the block and then pressing the "delete" key on your keyboard (or command-X on a mac).

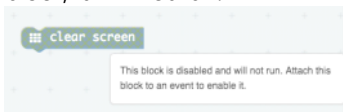


Looking at our pseudocode, we want to make sure to start a program with a clear screen.

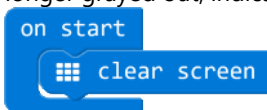
- We can do this by going to the Basic menu -> More and choosing a 'clear screen' block.



- Drag the 'clear screen' block to the coding Workspace. Notice that the block is 'grayed' out. If you hover over the 'grayed out' block, a pop up text box will appear letting you know that since this block is not attached to an event handler block, it will not run.



- Go ahead and drag the 'clear screen' block into the 'on start' block. Now the block is no longer grayed out, indicating that it will run when the event, the program starts, occurs.



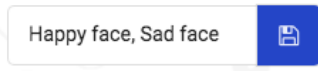
```
basic.clearScreen()
```

Save early, save often!

We now have a working program running on the micro:bit simulator!

As you write your program, MakeCode will automatically compile and run your code on the simulator. The program doesn't do much at this point, but before we make it more interesting, we should name our program and save it.

On the bottom left of the application window, to the right of the Download button, is a text box in which you can name your program. After naming your program, press the save button to save it.



Important: Whenever you write a significant piece of code or just every few minutes, you should save your code. Giving your code a meaningful name will help you find it faster from a list of programs and will let others know what your program does.

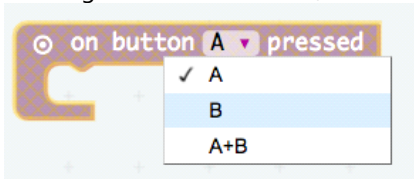
More event handlers

Now to make our program a bit more interesting by adding two more event handlers.

- From the Input menu, drag two 'on button A pressed' blocks to the coding window. Notice that the second block is grayed out. This is because, right now, they are the same block, both 'listening' for the same event 'on button A pressed'.



- Leave the first block alone for now, and using the drop-down menu within the second block, change the 'A' to 'B'. Now this block will no longer be grayed out, as it is now listening for a different event, 'on button B pressed'.

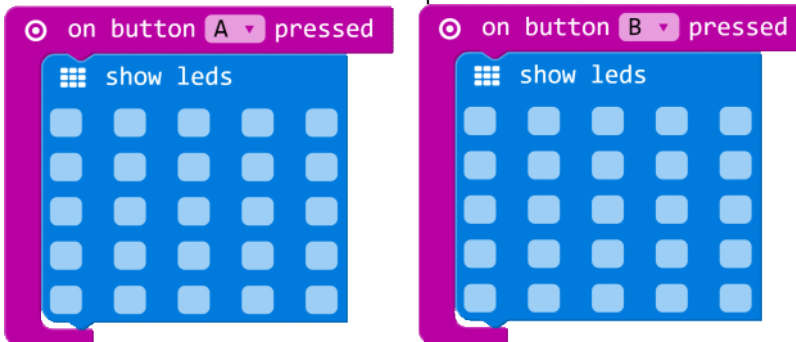


```
input.onButtonPressed(Button.A, () => {
})
input.onButtonPressed(Button.B, () => {
})
```

Show LEDs

Now we can use our LED lights to display different images depending on what button the user presses.

- From the Basic menu, drag two 'show leds' blocks to the coding workspace
- Place one 'show leds' block into the 'on button A pressed' event handler and the second 'show leds' block into the 'on button B pressed' event handler.



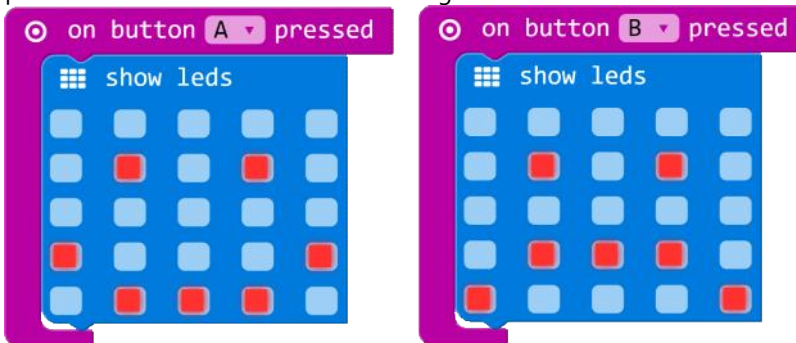
```
input.onButtonPressed(Button.A, () => {
  basic.showLeds(`
    . . . . .
    . . . . .
    . . . . .
    . . . . .
    . . . . .
  `)
})
```

```

        . . . . .
        . . . . .
        . . . . .
        . . . . .
        ` )
    })
    input.onButtonPressed(Button.B, () => {
        basic.showLeds(`
            . . . . .
            . . . . .
            . . . . .
            . . . . .
            `)
    })
}

```

- Click on the individual little boxes in the 'show leds' block that is in the 'on button A pressed' event handler to create the image of a happy face.
- Click on the individual little boxes in the 'show leds' block that is in the 'on button B pressed' event handler to create the image of a sad face.



```

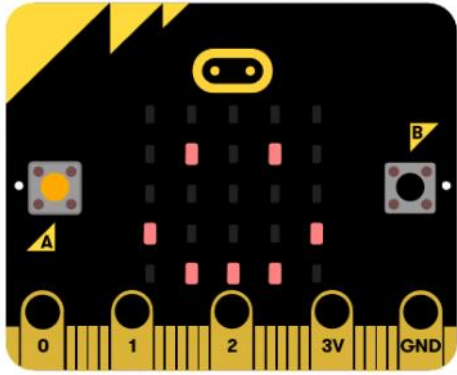
input.onButtonPressed(Button.A, () => {
    basic.showLeds(`
        . . . . .
        . # . # .
        . . . . .
        # . . . #
        . # # # .
        `)
})
input.onButtonPressed(Button.B, () => {
    basic.showLeds(`
        . . . . .
        . # . # .
        . . . . .
        . # # # .
        # . . . #
        `)
})

```

Test your program!

Remember, MakeCode automatically compiles and runs your program, so all you need to do now is press button A and then button B in the simulator to see the output produced by your code.

- Feel free to play around with turning LEDs on or off in the 'show leds' blocks until you get the images you want.
- Remember to save your code.

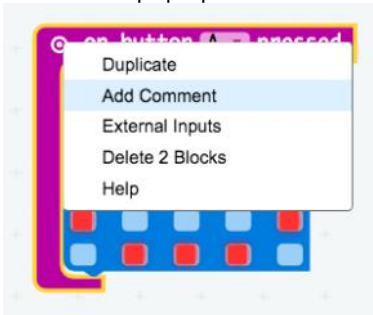


Commenting your code

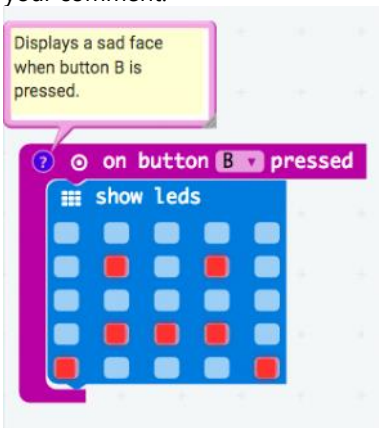
It is good practice to add comments to your code. Comments can be useful in a number of ways. Comments can help you remember what a certain block of code does and/or why you chose to program something the way you did. Comments also help others reading your code to understand these same things.

To comment a block of code:

- Right-click on the icon that appears before the words on a block.
- A menu will pop up. Select 'Add Comment'.



- This will cause a question mark icon to appear to the left of the previous icon.
- Click on the question mark and a small yellow box will appear into which you can write your comment.



- Click on the question mark icon again to close the comment box when you are done.
- Click on the question mark icon whenever you want to see your comment again or to edit it.

Notes

- When you right-click on the icon that appears before the words on a block, notice that there are other options available to you that allow you to duplicate and delete blocks, as

well as get help. Feel free to explore and use these as you code.

- In JavaScript, you can add a comment by using two forward slashes, then typing your comment. The two forward slashes tell JavaScript that the following text (on that same line) is a comment.

```
// Display a happy face when button A is pressed.
```

Cleaning up!

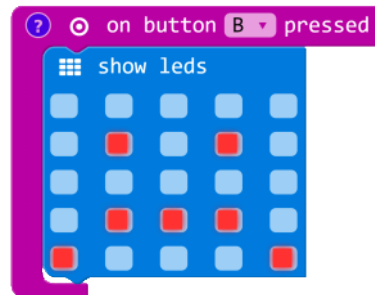
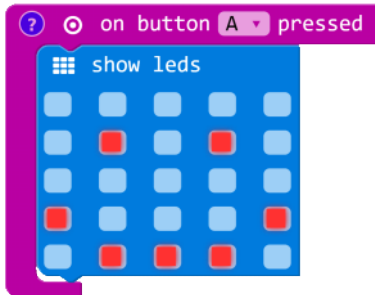
Clean up your coding workspace before you do a final save! What does this mean?

- It means that only the code and blocks that you are using in your program are still in the workspace.
- Remove (delete) any other blocks that you may have dragged into the coding workspace as you were experimenting and building your program.

Save and Download

Now that your code is running just fine in the simulator, is commented, and your coding window is 'clean', save your program, download it to your micro:bit, and enjoy!

Here is the complete program:



```
// Display a happy face when button A is pressed.
input.onButtonPressed(Button.A, () => {
  basic.showLeds(`
    . . . . .
    . # . # .
    . . . . .
    # . . . #
    . # # # .
    `)
})
// Display a sad face when button B is pressed.
input.onButtonPressed(Button.B, () => {
  basic.showLeds(`
    . . . . .
    . # . # .
    . # # # .
    # . . . #
    `)
})
basic.clearScreen()
```

[HappySadFace](#)



Project: Fidget Cube

A fidget cube is a little cube with something different that you can manipulate on each surface. There are buttons, switches, and dials, and people who like to “fidget” find it relaxing to push, pull, press, and play with it. In this project, students are challenged to turn the micro:bit into their very own “fidget cube”.

Show students some examples of fidget cubes:

- Original Kickstarter Fidget Cube - <https://www.kickstarter.com/projects/antsylabs/fidget-cube-a-vinyl-desk-toy> (there is a funny video showing the fidget cube in action).

Discussion questions

- Do any of your students fidget?
- What kinds of things do they fidget with? *Spinning pens, fidget spinners, rings, coins?*
- There are many different versions of fidget cubes available now. Do any students have any?
- Have they seen them before?
- What are the types of fidget activities?
- If students could add or modify features of the fidget cube, what would they choose to do?
- What would make the ultimate fidget cube?

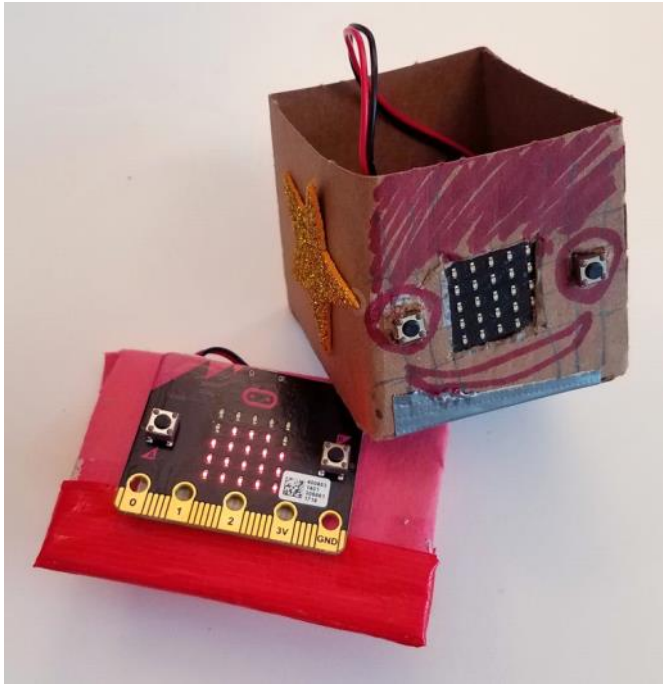
Remind students that a computing device has a number of inputs, and a number of outputs. The code that we write processes input by telling the micro:bit what to do when various events occur.

Project

Make a fidget cube out of the micro:bit, create a unique output for each of the following inputs:

- on button A pressed
- on button B pressed
- on button A+B pressed
- on shake

See if you can combine a maker element similar to what you created in Lesson 1 by providing a holder for the micro:bit that holds it securely when you press one of the buttons.



Sample Fidget Cube designs

Project Mod

- Add more inputs and more outputs - use more than 4 different types of input. Try to use other types of output (other than LEDs) such as sound!

Assessment

	4	3	2	1
Inputs	At least 4 different inputs are successfully implemented	At least 3 different inputs are successfully implemented	At least 2 different inputs are successfully implemented	Fewer than 2 different inputs are successfully implemented
Outputs	At least 4 different outputs are successfully implemented	At least 3 different outputs are successfully implemented	At least 2 different outputs are successfully implemented	Fewer than 2 different outputs are successfully implemented
Micro:bit program	Micro:bit program: <ul style="list-style-type: none"> • uses event handlers in a way that is integral to the program • compiles and runs as intended, • includes meaningful comments 	Micro:bit program lacks 1 of the required elements	Micro:bit program lacks 2 of the required elements	Micro:bit program lacks all or of the required elements
Collaboration reflection	Reflection piece includes: <ul style="list-style-type: none"> • brainstorming 	Reflection piece lacks 1 of the required elements	Reflection piece lacks 2 of the required elements	Reflection piece lacks 3 of the required elements

- | | | | |
|----------------|--|--|--|
| ideas | | | |
| • construction | | | |
| • programming | | | |
| • beta testing | | | |

Standards

CSTA K-12 Computer Science Standards

- CT.L2-03 Define an algorithm as a sequence of instructions that can be processed by a computer.
- CD.L2-01 Recognize that computers are devices that execute programs.
- CD.L2-02 Identify a variety of electronic devices that contain computational processors.
- CD.L2-03 Demonstrate an understanding of the relationship between hardware and software.
- CD.L3A-04 Compare various forms of input and output.

Variables

This lesson introduces the use of variables to store data or the results of mathematical operations. Students will practice giving variables unique and meaningful names. And we will introduce the basic mathematical operations for adding subtracting, multiplying, and dividing variables.



Lesson Objectives

Students will...

- Understand what variables are and why and when to use them in a program.
- Learn how to create a variable, set the variable to an initial value, and change the value of the variable within a micro:bit program.
- Learn how to create meaningful and understandable variable names.
- Understand that a variable holds one value at a time.
- Understand that when you update or change the value held by a variable, the new value *replaces* the previous value.
- Learn how to use the basic mathematical blocks for adding, subtracting, multiplying, and dividing variables.
- Apply the above knowledge and skills to create a unique program that uses variables as an integral part of the program.

Lesson Plan Structure

- Introduction: Variables in daily life
- Unplugged Activity: Rock Paper Scissors scorekeeping activity
- Micro:bit Activity: Make a game scorekeeper
- Project: Make a scorekeeper
- Project Mods
- Assessment: Rubric
- Standards: Listed

Introduction

Computer programs process information. Some of the information that is input, stored, and used in a computer program has a value that is **constant**, meaning it does not change throughout the course of the program. An example of a **constant** in math is 'pi' because 'pi' has one value that never changes. Other pieces of information have values that **vary** or change during the running of a program. Programmers create **variables** to hold the value of information that may change. In a game program, a variable may be created to hold the player's current score, since that value would change (hopefully!) during the course of the game.

Ask the students to think of some pieces of information in their daily life that are **constants** and others that are **variables**.

- What pieces of information have values that don't change during the course of a single day (constants)?
- What pieces of information have values that do change during the course of a single day (variables)

Constants and variables can be numbers and/or text.

Examples

In one school day...

- Constants: The day of the week, the year, student's name, the school's address
- Variables: The temperature/weather, the current time, the current class, whether they are standing or sitting...

Variables hold a specific type of information. The micro:bit's variables can keep track of numbers, strings, booleans, and sprites. The first time you use a variable, its type is assigned to match whatever it is holding. From that point forward, you can only change the value of that variable to another value of that same type.

- A number variable could hold numerical data such as the year, the temperature, or the degree of acceleration.
- A string variable holds a string of alphanumeric characters such as a person's name, a password, or the day of the week.
- A boolean variable has only two values: true or false. You might have certain things that happen only when the variable called `gameOver` is false, for example.
- A sprite is a special variable that represents a single dot on the screen and holds two separate values for the row and column the dot is currently in.

Unplugged: Keeping Score

To experience creating and working with variables, have students pair up and play Rock Paper Scissors.



Ask students to keep track of their scores on paper.

You can also have students play in groups of three with the third student acting as the scorekeeper.

Students will keep track of how many times each player wins as well as the number of times the players tie.

Play: Have students play Rock Paper Scissors for about a minute. When done, ask the students to add up their scores and how many 'rounds' they played.

Play again: Tell students they will now start over and play again for another minute. When done, ask the students to add up their scores and how many 'rounds' they played.

Ask some students to share how they kept track of player scores.

There may be some variety, but most will have written down the players' names and then beside or below the names, marks representing the 'wins' of each player. And they may have made a separate place for recording ties.

Game 1		
	W	T
Mary		—
Sam		

Game 2		
	W	T
Mary		—
Sam		

Sample Score-keeping sheet

Ask the students what parts of the score sheet represent **constants**, values that do not change through the course of a gaming session. Example: The players' names are constants.

Ask the students what parts of the score sheet represent **variables**, values that do change through the course of a gaming session. Example: The players' number of wins are variables.

Activity: Scorekeeper

This Micro:bit activity guides the students to create a program with three variables that will keep score for their Rock Paper Scissors game.

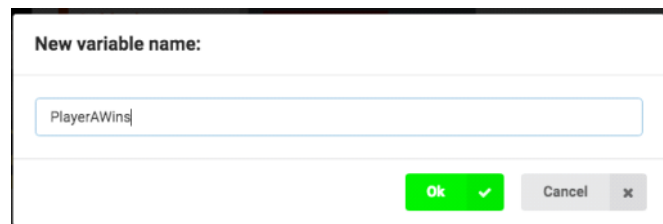
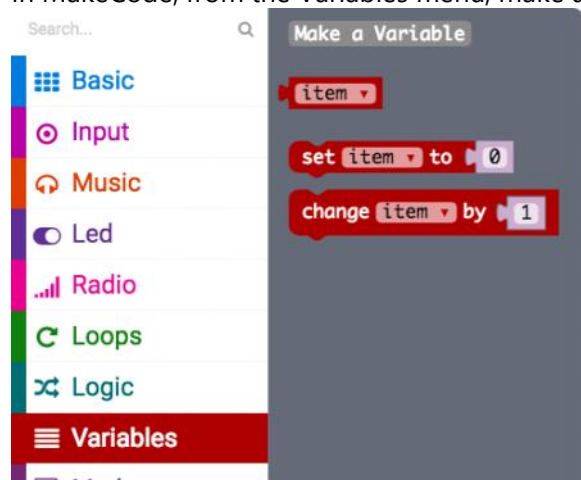
Tell the students that they will be creating a program that will act as a scorekeeper for their next Rock Paper Scissors game. They will need to create variables for the parts of scorekeeping that change over the course of a gaming session. What are those variables?

- The number of times the first player wins
- The number of times the second player wins
- the number of times the players tie

Creating and naming variables: Lead the students to create meaningful names for their variables.

- What would be a unique and clear name for the variable that will keep track of the number of times Player A wins?
- Student suggestions may be: 'PAW', 'PlayerA', 'AButtonPress', 'AButtonCount', 'PlayerAWins'...
- Discuss why (or why not) different suggestions make clear what value the variable will hold. *In general, variable names should clearly describe what type of information they hold.*

In MakeCode, from the Variables menu, make and name these three variables: PlayerAWins, PlayerBWins, PlayersTie



Initializing the variable value

It is important to give your variables an initial value. The initial value is the value the variable will hold each time the program starts. For our counter program, we will give each variable the value 0 (zero) at the start of the program.



```
let PlayerAWins = 0
let PlayerBWins = 0
let PlayersTie = 0
```

Updating the variable value

In our program, we want to keep track of the number of times each player wins and the number of times they tie. We can use the buttons A and B to do this.

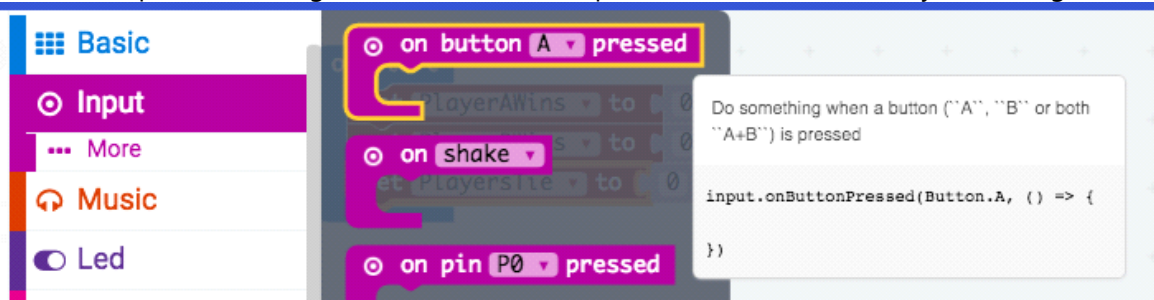
Pseudocode:

- Press button A to record a win for player A
- Press button B to record a win for player B
- Press both button A and button B together to record a tie

We already initialized these variables and now need to code to update the values at each round of the game.

- Each time the scorekeeper presses button A to record a win for Player A, we want to add 1 to the current value of the variable 'PlayerAWins'.
- Each time the scorekeeper presses button B, to record a win for Player B, we want to add 1 to the current value of the variable 'PlayerBWins'.
- Each time the scorekeeper presses both button A and button B at the same time to record a tie, we want to add 1 to the current value of the variable PlayersTie

From the Input menu, drag 3 of the 'on button A pressed' event handlers to your coding window.



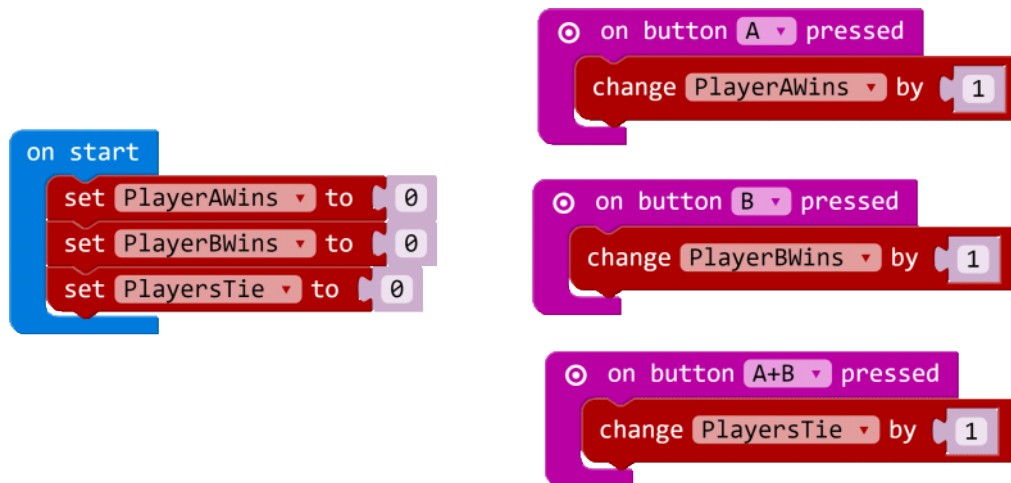
Leave one block with 'A'. Use the drop-down menu in the block to choose 'B' for the second block and 'A+B' for the third block.

From the Variables menu, drag 3 of the 'change item by 1' blocks to your coding window.



Place one change block into each of the Button Pressed blocks.

Choose the appropriate variable from the pull down menus in the change blocks.



```

input.onButtonPressed(Button.A, () => {
  PlayerAWins += 1
})
input.onButtonPressed(Button.B, () => {
  PlayerBWins += 1
})
input.onButtonPressed(Button.AB, () => {
  PlayersTie += 1
})

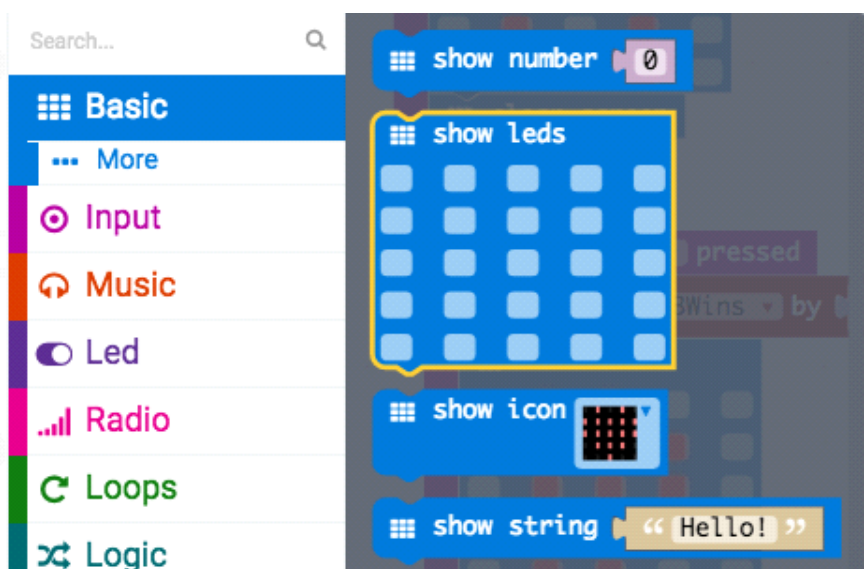
```

User feedback

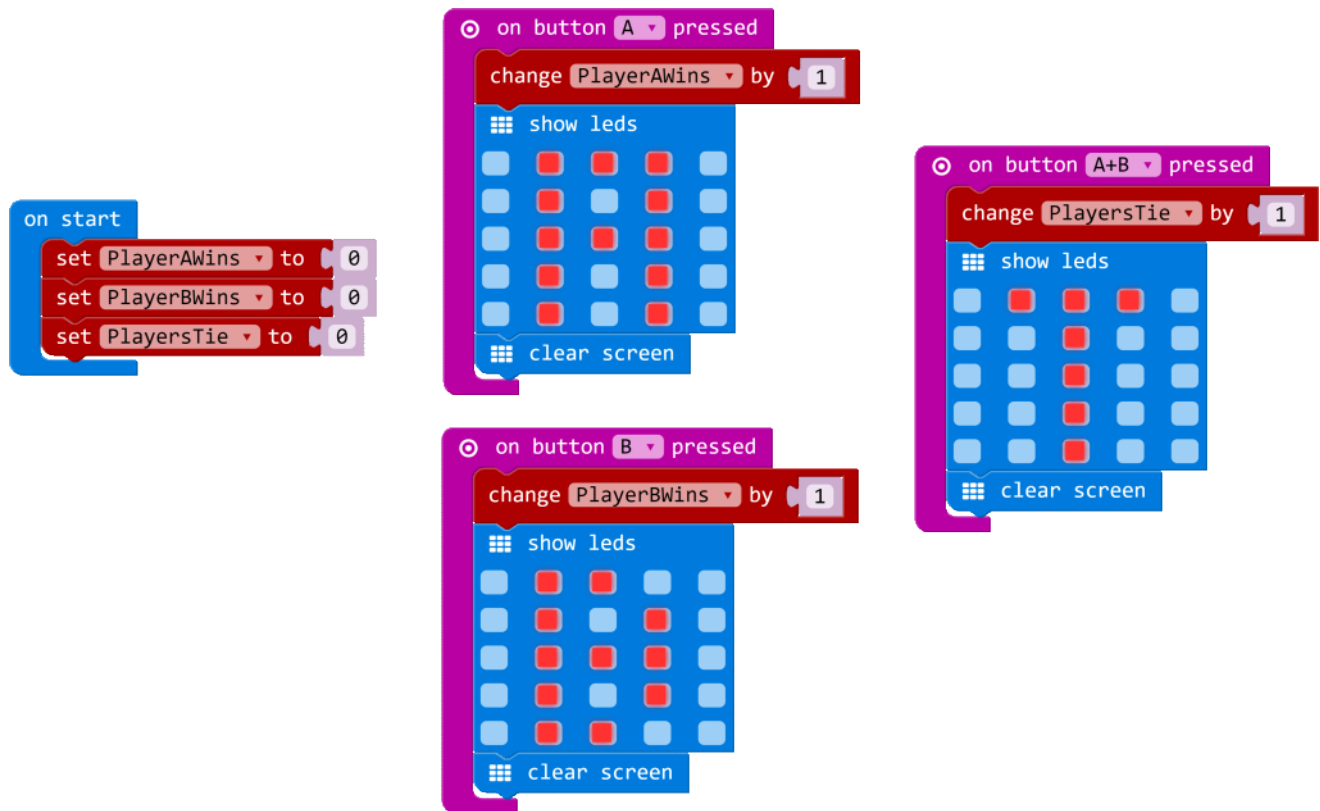
Whenever the scorekeeper presses button A, button B, or both buttons together, we will give the user visual feedback acknowledging that the user pressed a button. We can do this by coding our program to display:

- an 'A' each time the user presses button A to record a win for Player A,
- a 'B' for each time the user presses button 'B' to record a win for Player B,
- a 'T' for each time the user presses both button A and button B together to record a tie.

We can display an 'A', 'B', or 'T' using either the 'show leds' block or the 'show string' block.



In this example, we have used the 'show leds' block.



```

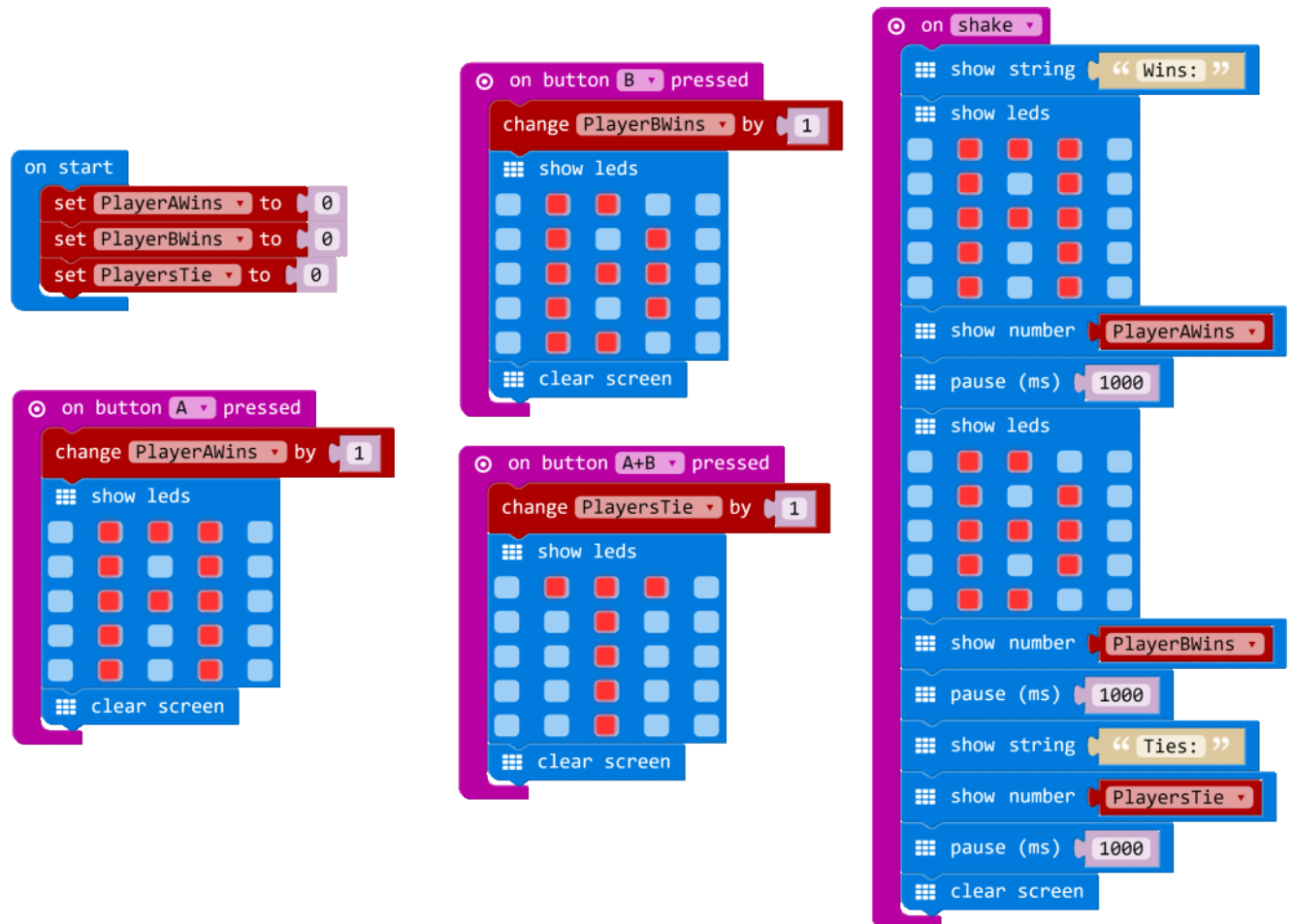
input.onButtonPressed(Button.A, () => {
  PlayerAWins += 1
  basic.showLeds(`
    . # # # .
    . # . # .
    . # # # .
    . # . # .
    . # . # .
  `)
  basic.clearScreen()
})
input.onButtonPressed(Button.B, () => {
  PlayerBWins += 1
  basic.showLeds(`
    . # # . .
    . # . # .
    . # # # .
    . # . # .
    . # # . .
  `)
  basic.clearScreen()
})
input.onButtonPressed(Button.AB, () => {
  PlayersTie += 1
  basic.showLeds(`
    . # # # .
    . . # . .
    . . # . .
    . . # . .
    . . # . .
  `)
  basic.clearScreen()
})

```

Notice that we added a 'clear screen' block after showing 'A', 'B', or 'T'. What do you think would happen if we did not clear the screen? Try it.

Showing the final values of the variables

To finish our program, we can add code that tells the Micro:bit to display the final values of our variables. Since we have already used buttons A and B, we can use the 'on shake' event handler block to trigger this event. We can use the 'show string', 'show leds', 'pause', and 'show number' blocks to display these final values in a clear way. Here is the complete program.



```

let PlayersTie = 0
let PlayerBWins = 0
let PlayerAWins = 0
input.onButtonPressed(Button.A, () => {
  PlayerAWins += 1
  basic.showLeds(`
    . # # # .
    . # . # .
    . # # # .
    . # . # .
    . # . # .
  `)
  basic.clearScreen()
})
input.onButtonPressed(Button.B, () => {
  PlayerBWins += 1
  basic.showLeds(`
    . # # . .

```

```

        . # . # .
        . # # # .
        . # . # .
        . # # . .
        `)
    basic.clearScreen()
})
input.onButtonPressed(Button.AB, () => {
    PlayersTie += 1
    basic.showLeds(`
        . # # # .
        . . # . .
        . . # . .
        . . # . .
        . . # . .
        `)
    basic.clearScreen()
})
input.onGesture(Gesture.Shake, () => {
    basic.showString("Wins:")
    basic.showLeds(`
        . # # # .
        . # . # .
        . # # # .
        . # . # .
        . # . # .
        `)
    basic.showNumber(PlayerAWins)
    basic.pause(1000)
    basic.showLeds(`
        . # # . .
        . # . # .
        . # # # .
        . # . # .
        . # # . .
        `)
    basic.showNumber(PlayerBWins)
    basic.pause(1000)
    basic.showString("Ties:")
    basic.showNumber(PlayersTie)
    basic.pause(1000)
    basic.clearScreen()
})
PlayerAWins = 0
PlayerBWins = 0
PlayersTie = 0

```

Scorekeeper



Try it out!

Download the Scorekeeper program to the micro:bit, and have the students play one last round of Rock Paper Scissors using their micro:bits to act as the Scorekeeper!

'Adding' on with mathematical operations

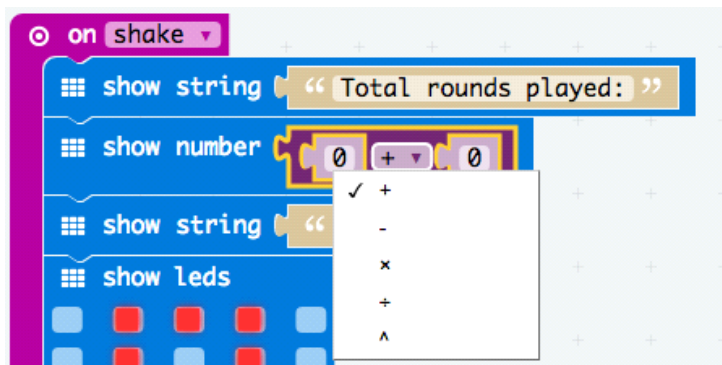
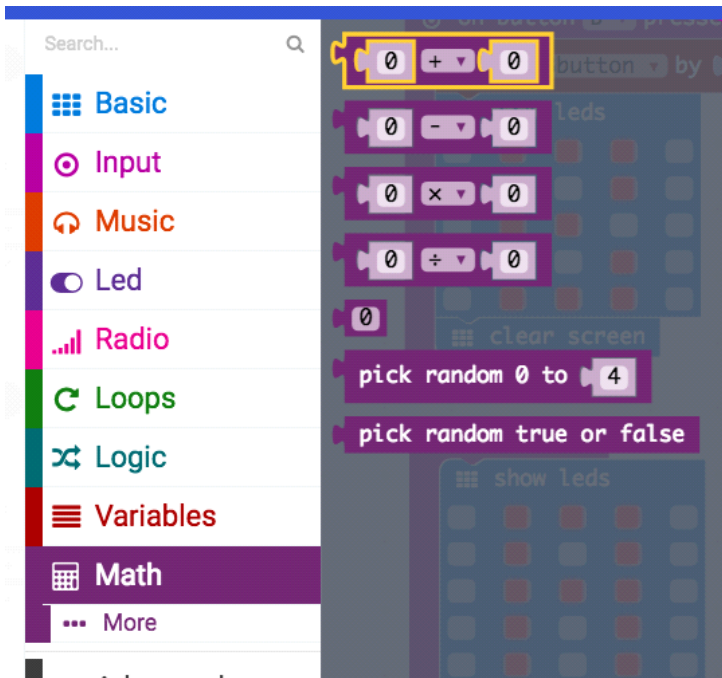
There is more we can do with the input we received using this program. We can use mathematical operations on our variables.

Example: Perhaps you'd like to keep track of, and show the player the total number of 'rounds' that were played. To do this, we can add the values stored in the variables we created to keep track of how many times each player won and how many times they tied.

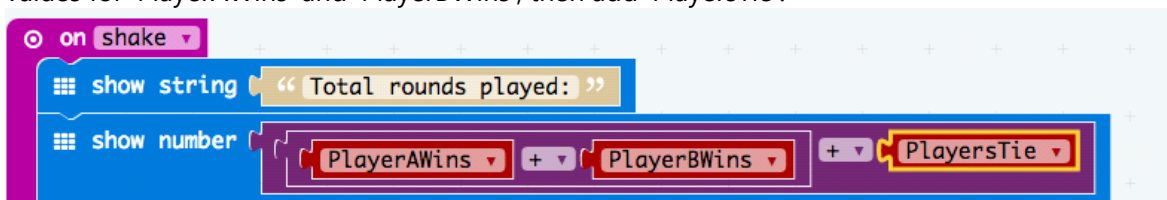
In order to do this, we can add the code to our program under the 'on shake' event handler .

- First, display a string to show the player that the following sum represents the total number of rounds played.
- Our program will add the values stored in the variables 'PlayerAWins', 'PlayerBWins', and 'PlayersTie' and then display the sum of this mathematical operation.
- The blocks for the mathematical operations adding, subtracting, multiplying, and dividing are listed in the Math section of the Toolbox.

Note: Even though there are 4 blocks shown for these 4 operations, you can access any of the four operations from any of the four blocks, and you can also access the exponent operation from these blocks.



- Replace the default values of zero with the names of the variables we want to add together. Notice that because we are adding three variables together we need a second math block. First we add the values for 'PlayerAWins' and 'PlayerBWins', then add 'PlayersTie'.



```
input.onGesture(Gesture.Shake, () => {
  basic.showString("Total rounds played:")
  basic.showNumber(PlayerAWins + PlayerBWins + PlayersTie)
})
```

- Save, download, and try the program again to make sure that it runs correctly and displays the correct numbers for each variable.

Remember that the micro:bit is a device that processes input and displays it as output in some way. By storing values in variables, you can perform mathematical operations on that data that provides you with useful information.

What other math operations could provide valuable information from the values stored in these variables?

Examples:

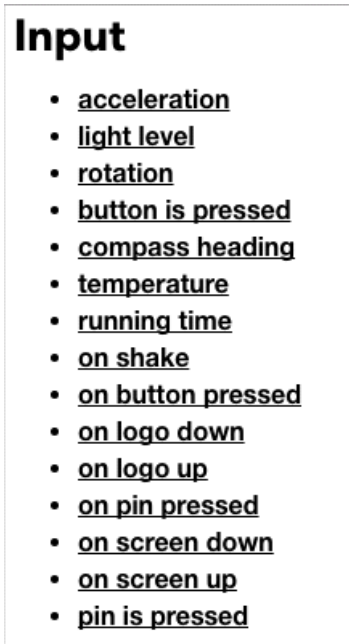
- Calculate and display a player's wins and/or losses as a percentage of all rounds played.
- Calculate and display the number of tied games as a percentage of all rounds played.

Project: Everything Counts

This is an assignment for students to come up with a Micro:bit program that counts something. Their program should keep track of **input** by storing values in variables, and provide **output** in some visual and useful way. Students should also perform mathematical operations on the variables to give useful output.

Input

Remind the students of all the different inputs available to them through the Micro:bit.



Project Ideas

Duct tape wallet

You can find instructions on the web for creating a durable, fashionable wallet or purse out of duct tape (<https://pxt.microbit.org/projects/wallet>). Create a place for the micro:bit to fit securely. Use Button A to add dollars to the wallet, and Button B to subtract dollars from the wallet.

Extra Mod: Use other inputs to handle cents, and provide a way to display how much money is in the wallet in dollars and cents.

Umpire's baseball counter (pitches and strikes)

In baseball during an at-bat, umpires must keep track of how many pitches have been thrown to each batter. Use Button A to record the number of balls (up to 4) and the number of strikes (up to 3).

Extra Mod: Create a way to reset both variables to zero, create a way to see the number of balls and strikes on the screen at the same time.

Shake counter

Using the 'On Shake' block, you can detect when the micro:bit has been shaken and increment a variable accordingly. Try attaching the micro:bit to a lacrosse stick and keep track of how many times you have successfully thrown the ball up in the air and caught it.

Extra Mod: Make the micro:bit create a sound of increasing pitch every time you successfully catch the ball.

Pedometer

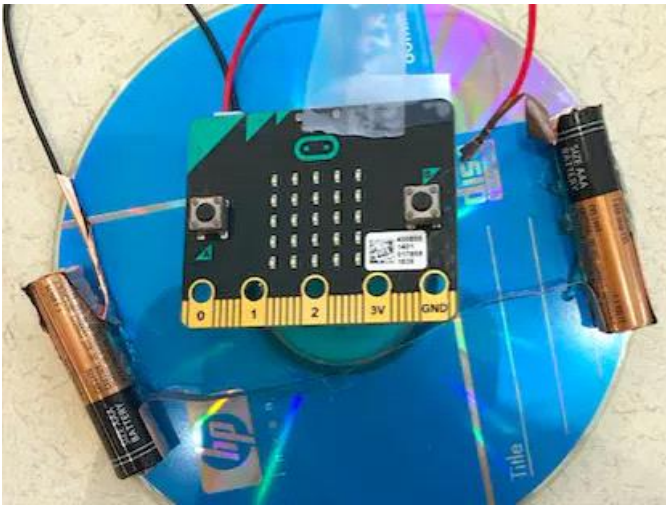
See if you can count your steps while running or doing other physical activities carrying the micro:bit. Where is it best mounted?

Extra Mod: Design a wearable band or holder that can carry the Micro:bit securely so it doesn't slip out during exercise.

Calculator

Create an adding machine. Use Button A to increment the first number, and Button B to increment the second number. Then, use Shake or Buttons A + B to add the two numbers and display their sum.

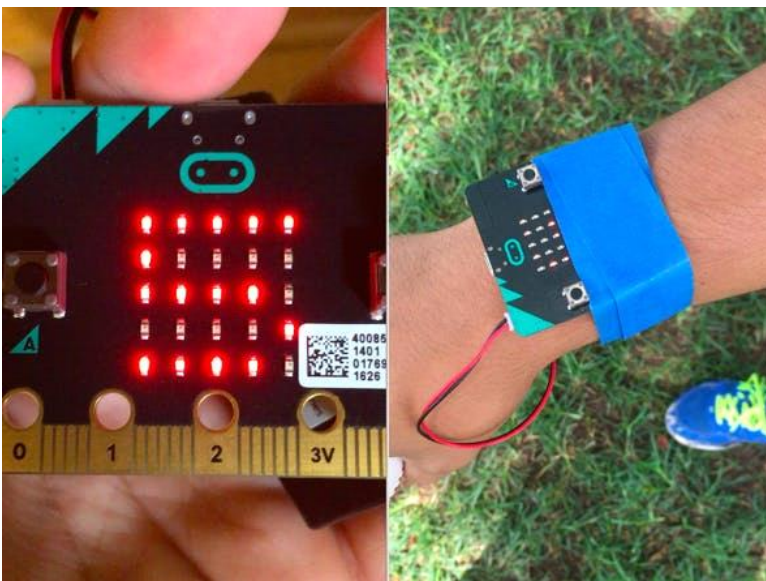
Extra Mod: Find a way to select and perform other math operations.



Homemade Top with Micro:bit Revolution Counter



Duct Tape Wallet with Micro:bit Display



Baseball Pitch Counter

Process

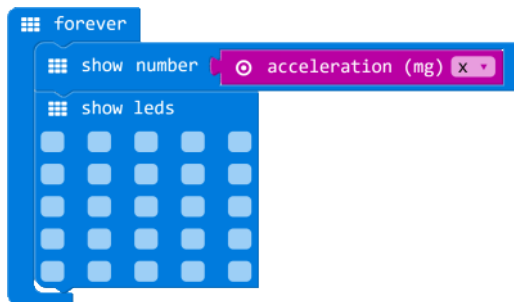
In any design project, it's important to start by understanding the problem. You can begin this activity by interviewing people around you

who might have encountered the problem you are trying to solve. For example, if you are designing a wallet, ask your friends how they store their money, credit cards, identification, etc. What are some challenges with their current system? What do they like about it? What else do they use their wallets for?

If you are designing something else, think about how you might find out more information about your problem through interviewing or observing people using current solutions.

Then start brainstorming. Sketch out a variety of different ideas. Remember that it's okay if the ideas seem far-out or impractical. Some of the best products come out of seemingly crazy ideas that can ultimately be worked into the design of something useful. What kind of holder can you design to hold the Micro:bit securely? How will it be used in the real world, as part of a physical design?

Use the simulator to do your programming, and test out a number of different ideas. What is the easiest way to keep track of data? If you are designing for the accelerometer, try to see what different values are generated through different actions (you can display the value the accelerometer is currently reading using the 'Show Number' block; clear the screen afterward so you can see the reading).



```
basic.forever(() => {
  basic.showNumber(input.acceleration(Dimension.X))
  basic.showLeds(`
    . . . . .
    . . . . .
    . . . . .
    . . . . .
    . . . . .
  `)
})
```

Reflection

Have students write a reflection of about 150–300 words, addressing the following points:

- What was the problem you were trying to solve with this project?
- What were the Variables that you used to keep track of information?
- What mathematical operations did you perform on your variables? What information did you provide?
- Describe what the physical component of your Micro:bit project was (e.g., an armband, a wallet, a holder, etc.)
- How well did your prototype work? What were you happy with? What would you change?
- What was something that was surprising to you about the process of creating this project?
- Describe a difficult point in the process of designing this project, and explain how you resolved it.

Assessment

	4	3	2	1
Variables	At least 3 different variables are implemented in a meaningful way	At least 2 variables are implemented in a meaningful way	At least 1 variable is implemented in a meaningful way	No variables are implemented.
Variable names	All variable names are unique and clearly describe what information values the variables hold.	The majority of variable names are unique and clearly describe what information values the variables hold.	A minority of variable names are unique and clearly describe what information values the variables hold.	None of the variable names clearly describe what information values the variables hold.
Mathematical operations	Uses a mathematical operation on at least two variables in a way that is integral to the program	Uses a mathematical operation on at least one variable in a way that is integral to the program	Uses a mathematical operation incorrectly or not in a way that is integral to the program	No mathematical operations are used.

Micro:bit program	<p>Micro:bit program:</p> <ul style="list-style-type: none"> • Uses variables in a way that is integral to the program, • Uses mathematical operations to add, subtract, multiply, and/or divide variables, • Compiles and runs as intended, • Meaningful comments in code 	Micro:bit program lacks 1 of the required elements	Micro:bit program lacks 2 of the required elements	Micro:bit program lacks 3 or more of the required elements
Collaboration reflection	Reflection piece addresses all prompts.	Reflection piece lacks 1 of the required elements.	Reflection piece lacks 2 of the required elements.	Reflection piece lacks 3 of the required elements.

Standards

CSTA K-12 Computer Science Standards

- CL.L2-03 Collaborate with peers, experts, and others using collaborative practices such as pair programming, working in project teams, and participating in group active learning activities
- CT.L1:6-01 Understand and use the basic steps in algorithmic problem-solving
- CT.L1:6-02 Develop a simple understanding of an algorithm using computer-free exercises
- CPP.L1:6-05 Construct a program as a set of step-by-step instructions to be acted out
- 2-A-5-7 Create variables that represent different types of data and manipulate their values.

Conditional Statements

This lesson introduces the Logic blocks such as 'If...then' and 'If...then...else'. Students practice skills of creativity, problem-solving, and collaboration.

Lesson Objectives

Students will...

- Understand what conditional statements are, and why and when to use them in a program.
- Learn how to use the Logic blocks 'If...then' and 'If...then...else'.
- Practice using the Logic blocks so different conditions yield specified outcomes.
- Demonstrate understanding and apply skill by collaborating with classmates to create a game that uses a micro:bit and a program that correctly and effectively uses conditionals.

Lesson Plan Structure

- Introduction: Conditionals in daily life
- Unplugged Activity: Red if, Green then
- Micro:bit Activity: Rock Paper Scissors
- Project: Board Game
- Assessment: Rubric for board game project
- Standards: Listed

Introduction

Computer programs are instructions telling the computer how to process input and deliver output.

An important part of programming is telling the computer WHEN to perform a certain task. For this, we use something called 'conditionals'. Conditionals get their name because a certain Condition or Rule has to be met.

Students are all already familiar with the concept of conditionals in their daily lives!

Have they ever had their parents say..?

- "If you clean your room, you can go out with your friends."
- "If your homework is done, you can play video games."
- "If you do your chores all week, you get your allowance, else you are grounded."

These are all conditionals! Conditionals follow the format of IF this, THEN that.

IF (condition is met), **THEN** (action performed)

Have the students share a few conditionals from their own lives with the class or within small groups.

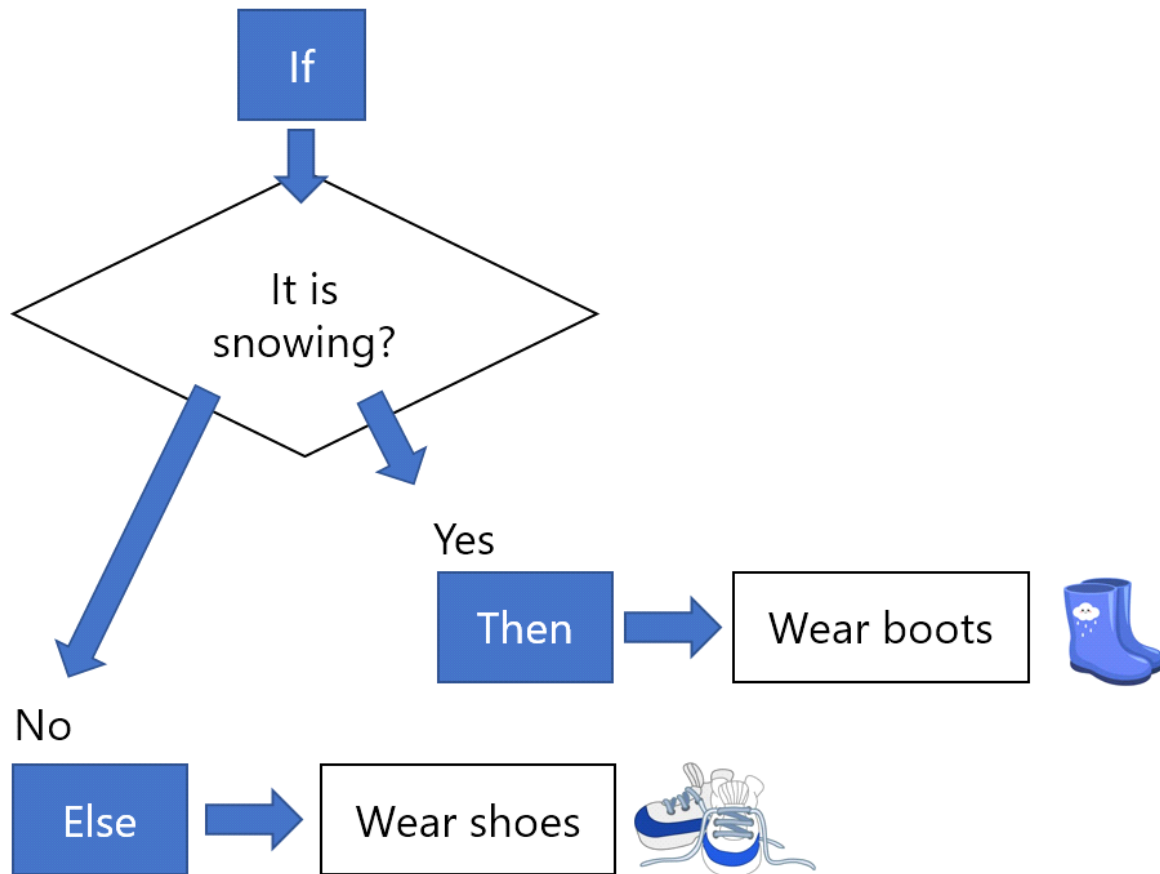
Note: For older students, you can have them add the ELSE portion of a conditional.

IF (condition is met), **THEN** (action performed), **ELSE** (different action performed)

Example:

- IF it is snowing, THEN wear boots, ELSE wear shoes.

The ELSE portion makes sure that a different action is performed in either case. Without the ELSE action, your students might be barefoot!



Tell the students that they will be acting out some conditionals as though the whole class is a computer program for a game. Each student will perform a described action if the indicated condition is met.

Note: This activity can be done as a whole class or in smaller groups or as a pencil and paper activity.

Unplugged: Red Light Green Light



Objective

To reinforce the programming of basic conditionals by having students experience conditionals through acting them out in real life.

Activity Overview

Students will line up at one end of the classroom with the goal of reaching the other side of the classroom. The teacher, and then the students themselves will call out conditionals and all the students will advance or not depending on the specific conditional statement.

Note: As the teacher you will need to keep an eye out for any 'errors' that occur during the running of the program.

Materials

- Pencils and lined paper (if doing this activity seated). Students can advance across the paper instead of the room with one inch line equal to one step.

Process

- Have the students line up at one side of the room.
- Explain the rules:
 - The object of the game is to get across the room first.
 - For 'If...then' conditionals: If the condition called out is 'true' for you, then perform the action described in the 'then'. If the condition called out is 'false' for you, then do nothing.
 - For 'If...then...else' conditionals, listen carefully to the whole condition, as the 'else' may apply to you.

Example conditional statements

- If you are wearing something green, then take a step forward.
- If you have the letter 'e' in your first name, then take two giant steps forward.
- If you are wearing sneakers, then take a step forward, else take 2 steps forward.
- If your birthday is this month, then take a giant hop forward.

The conditionals you use will depend on your individual class.

After the students get the idea of the game, allow them to make up and call out conditionals (that meet teacher approval).

They will need to be observant, as a conditional that moves them forward, will also move their

competition forward!

Tips

- SAFETY FIRST! Students, especially younger ones, can get quite silly with this and while it is meant to be fun and even funny, safety first!
- Student conditionals need to apply to at least two people in the class.

Reflections

How did they do? Were there any 'run-time errors'? Did a student miss a conditional being met or fail to correctly carry out the THEN or ELSE action? Were there some conditions that could be evaluated as something other than True or False (maybe, sometimes)?

Extensions/Variations

- Add AND, OR, AND/OR statements to the conditionals.
Example: If you have brown hair AND brown eyes, then...
- Create nested IF's
Example: If you are wearing sneakers, then... if you are also wearing white socks, take three steps forward.
- Let students create their own conditionals for future program runs with the class. (A very popular activity, though all conditionals should be run by the teacher first for approval.)
- Relate this activity to a system and have the students create the conditionals that would end in a product of some kind or the completion of some task, like writing a sentence or setting a table or constructing a simple structure.

Activity: Rock Paper Scissors

For this activity, each student will need a micro:bit.
Everyone will create the same program, the classic rock paper scissor game.



Introduce activity

- Have students recall the classic rock paper scissors game.
- What are the rules of the game? What are the conditionals?
- Example:
 - If Player A gets rock, and Player B gets scissors, Then Player A wins.
- Have students write the pseudocode for how to play the game on the micro:bit.
Example pseudocode:
 - On button A press: choose random number from 0-2
 - If random number = 0, then display rock icon,
 - Else if random number = 1, then display paper icon,
 - Else display scissors icon.
- Point out that because there are only three possibilities, we don't need to do a separate check to see if random number = 2. So we just use an else.

Micro:bit

- Working from the specifications, have students work in pairs to try to code a Rock Paper Scissors game on their own.
- If students get stuck, there is a tutorial at <https://pxt.microbit.org/projects/rock-paper-scissors> (steps 1 through 4), that leads students step-by-step through the process of coding a working rock paper scissor game for their micro:bit.
- Let them play the game against their program.

Ideas for Mods

- Add a way to keep score: Steps 5 through 7 in the tutorial
- Mod the game to use different images or to add more options like 'Rock Paper Scissors Lizard Spock', Step 8 in the tutorial

[rock-paper-scissors](https://pxt.microbit.org/projects/rock-paper-scissors)



Project: Board Game

This is an assignment for students to create a board game. It should take two to three class periods. If your school has a makerspace or an art classroom where students can access materials such as cardboard, poster paints, or markers, you might schedule your classes to work there.

Once students have finished the first version of their games, schedule time for students to play each other's games. Ideally, give them some time to give and gather feedback, then revise their games accordingly.

Introduction

Many board games use an electronic toy to signal moves, or provide clues. There are some funny examples online if you search for "electronic board game". Here are some examples:

Dark Tower (featuring Orson Welles): This is an example of a circular board game in which the pieces start on the edges and move in toward the middle.

[Electronic Dream Phone Board Game Commercial - 1992](#): This board game is really a logic puzzle. There are printed clues that illustrate relationships and the phone provides clues that help you to narrow down possibilities by a process of elimination.

[Stop Thief Electronic Board Game commercial 1979](#): This board game uses a device to give audio clues that help you to figure out what to do on the game board. It's a good example of how you might use sound as a clue.

Assignment

Students should work in pairs to create an original board game project in which micro:bit is a central feature, and the rules of their board game should use Conditionals.

Students will need to work together to come up with:

- A set of written rules (how to play)
- A game board
- A program for the micro:bit
- Photo documentation of the different game pieces, cards, or other components of the game with the micro:bit included as well as a screenshot of your micro:bit code. Each photo must have a caption that describes what the photo is documenting.
- Reflection: A text entry describing your team's game making process and each teammate's part in the creation of the game from brainstorming ideas, through construction, programming, and beta testing.

The micro:bit needs to work in conjunction with the game board and/or game pieces and should be a central feature of the game. Ideally, it should be more than a simple substitute for a six-sided die.

The Micro:bit might:

- Simulate the results of a battle between two pieces
- Randomly point in a different direction of travel
- Generate a result based on its current incline
- Point randomly at players and kill them
- Display a dynamic score
- ... let your imaginations run wild!

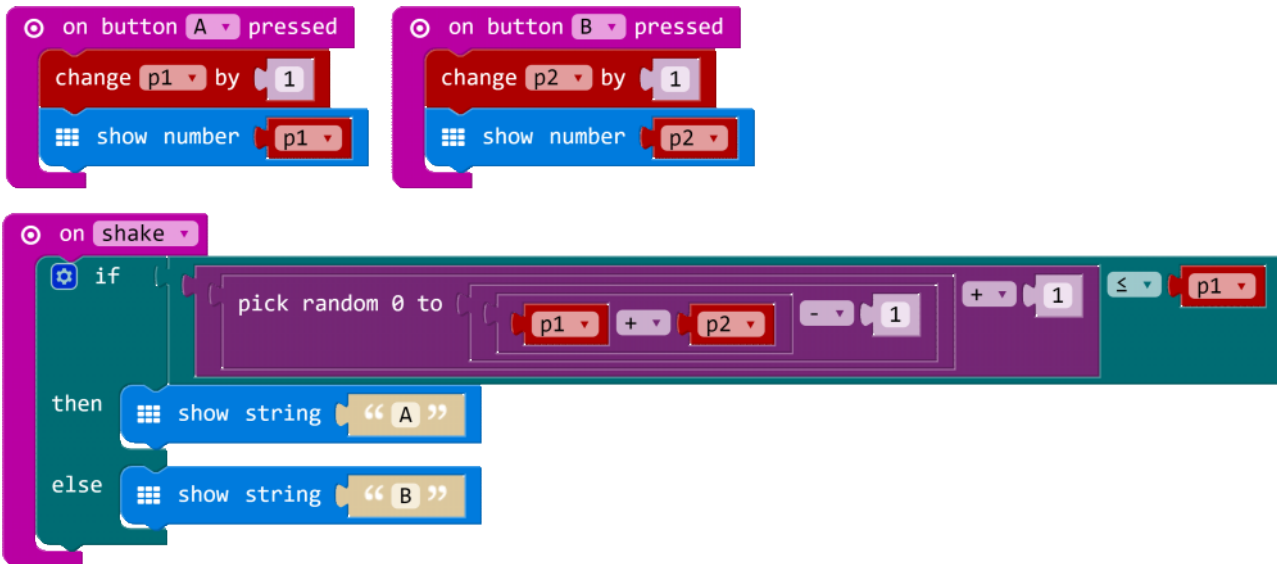
Ideally, students should be writing their own versions of Micro:bit programs to do something original.

Here is one simple program to discuss and use as an example:

Battle Game Pieces

In this example, pieces start out at full strength and lose points based on random events on the board. When two pieces meet on the same space, they battle.

- Press A to enter the strength of piece A.
- Then press B to enter the strength of piece B.
- Shake the Micro:bit to determine the winner of the battle, which is proportionately random to the strength of each piece.



```

let p2 = 0
let p1 = 0
input.onButtonPressed(Button.A, () => {
  p1 += 1
  basic.showNumber(p1)
})
input.onButtonPressed(Button.B, () => {
  p2 += 1
  basic.showNumber(p2)
})
input.onGesture(Gesture.Shake, () => {
  if (Math.random(p1 + p2 - 1 + 1) + 1 <= p1) {
    basic.showString("A")
  } else {
    basic.showString("B")
  }
})

```

[BattleGame](#)



Beta Testing

Give students a chance to play each other's games. The following process works well:

- Have each pair of students set up their own project at their table.
- Leave a clipboard or a laptop on the table for taking notes.
- Rotate the students through each project, moving clockwise around the room:
 - Play the game (5 min)
 - Fill out a survey form (5 min)

Sample Survey questions

- How easy was it to figure out what to do?
- What is something about this project that works really well?
- What is something that would make this project even better?
- Any other comments or suggestions?

Many online survey tools will allow you to sort the comments by project and share them with project creators so they can make improvements based on that feedback.

Reflection

Have students write a reflection of about 150–300 words, addressing the following points:

- Explain how you decided, as a pair, on your particular board game idea.
- What was something that was surprising to you about the process of creating this game?
- Describe a difficult point in the process of designing this game, and explain how you resolved it.
- What feedback did your beta testers give you? How did that help you improve your game? What were the Conditionals that you used as part of your game rules?

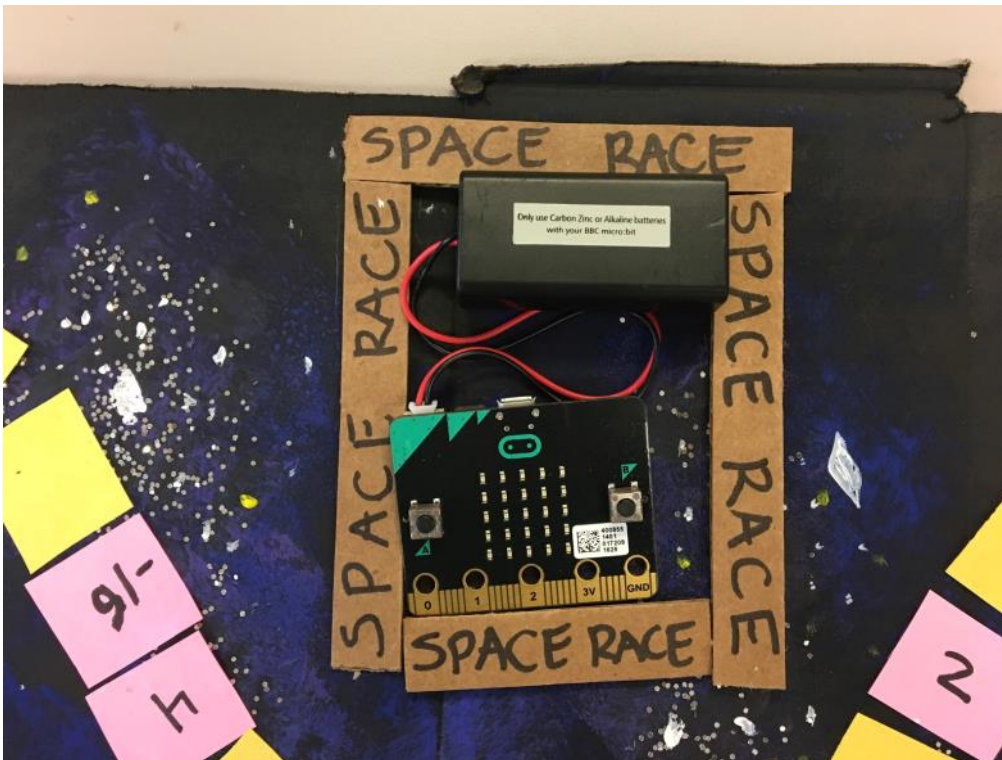
Board Game Example

Space Race by K. and S.

- How to win: Starting from Earth, your goal is to progress to Mars. The first person to reach Mars is the winner.
- Rules:
 - Shake the microbit to randomize how far you get to advance
 - If you land on a pink square, press "B" on the Micro:bit until your previous roll number appears. Then press A and B at the same time to see whether or not you move based upon the number on the square
 - Up to four players



Finished Game



Micro:bit Holder



Game Pieces

```

on start
  show string "SPACE RACE"
  set previous_roll to 0

on shake
  set current_roll to pick random 0 to 5
  show number current_roll + 1
  pause (ms) 5000
  clear screen

on button B pressed
  change previous_roll by 1
  show number previous_roll

on button A pressed
  change previous_roll by -1
  show number previous_roll

on button A+B pressed
  set previous_roll to 0
  if 4 <= previous_roll
    then set yes_or_no to pick random 0 to 7
  if 4 > previous_roll
    then set yes_or_no to pick random 0 to 4
  if 2 < yes_or_no
    then show string "YES"
    clear screen
  else show string "NO"
    clear screen
  
```

```

let yes_or_no = 0
let current_roll = 0
let previous_roll = 0
input.onButtonPressed(Button.AB, () => {
  previous_roll = 0
  if (4 <= previous_roll) {
    yes_or_no = Math.random(8)
  }
})
  
```

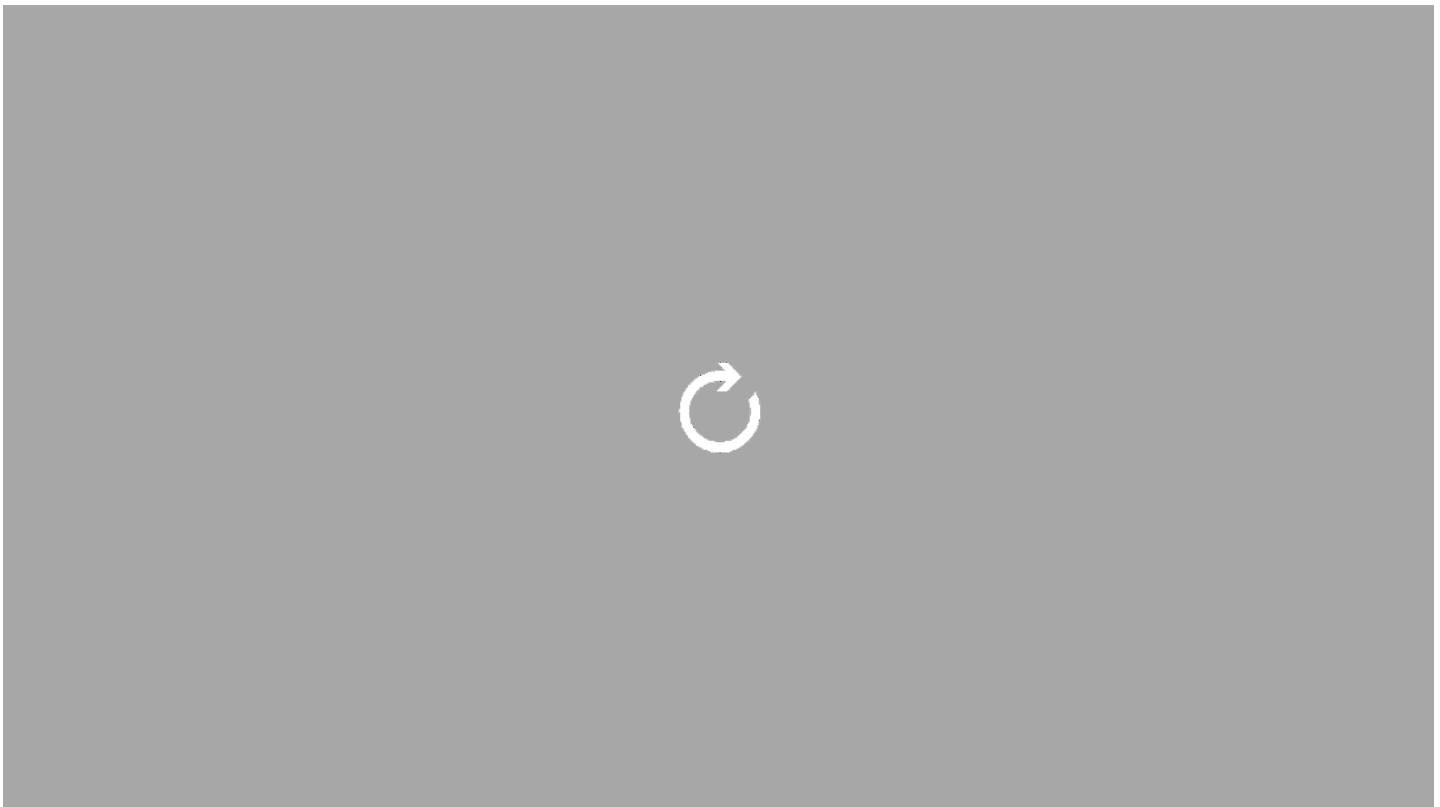


```

    if (4 > previous_roll) {
        yes_or_no = Math.random(5)
    }
    if (2 < yes_or_no) {
        basic.showString("YES")
        basic.clearScreen()
    } else {
        basic.showString("NO")
        basic.clearScreen()
    }
})
input.onGesture(Gesture.Shake, () => {
    current_roll = Math.random(6)
    basic.showNumber(current_roll + 1)
    basic.pause(5000)
    basic.clearScreen()
})
input.onButtonPressed(Button.B, () => {
    previous_roll += 1
    basic.showNumber(previous_roll)
})
input.onButtonPressed(Button.A, () => {
    previous_roll += -1
    basic.showNumber(previous_roll)
})
basic.showString("SPACE RACE")
previous_roll = 0

```

SpaceRace



Assessment

	4	3	2	1
Rules	All game rules are clear and complete	A game rule is missing or not complete or not clear	More than one game rule is missing or not complete or not clear	Most of the game rules are missing or it is not clear what the rules are.

Game board	<p>Game board is:</p> <ul style="list-style-type: none"> • Complete • Neat • Fits with the theme of the game • Micro:bit is a central part of the game 	Game board meets only 3 of the conditions listed for a score of 4.	Game board meets only 2 of the conditions listed for a score of 4.	Game board meets only 1 of the conditions listed for a score of 4.
Micro:bit program	<p>Micro:bit program:</p> <ul style="list-style-type: none"> • Uses the Micro:bit in a way that is integral to the game • Uses conditionals correctly • Compiles and runs as intended • JavaScript includes comments in code 	Micro:bit program lacks 1 of the required elements	Micro:bit program lacks 2 of the required elements	Micro:bit program lacks 3 of the required elements
Photo documentation	Complete photo documentation that includes photos of game board and code and captions.	A photo is missing or of poor quality or a caption is missing.	Multiple photos and/or captions missing or of poor quality.	Most photos and/or captions missing or of poor quality.
Collaboration reflection	<p>Reflection piece includes:</p> <ul style="list-style-type: none"> • Brainstorming ideas • Construction • Programming • Beta testing 	Reflection piece lacks 1 of the required elements.	Reflection piece lacks 2 of the required elements.	Reflection piece lacks 3 of the required elements.

Standards

CSTA K-12 Computer Science Standards

- CL.L2-03 Collaborate with peers, experts, and others using collaborative practices such as pair programming, working in project teams, and participating in group active learning activities.
- CL.L2-04 Exhibit dispositions necessary for collaboration: providing useful feedback, integrating feedback, understanding and accepting multiple perspectives, socialization.
- CL.L3A-01 Work in a team to design and develop a software artifact.
- K-12 Computer Science Framework Core concept: Control Structures

Iteration

Iteration & Looping

This lesson introduces the concept of looping and iteration. Presents the 'While' block as a combination of an iteration and a conditional statement.

Lesson Objectives

Students will...

- Understand the value of iteration in programming
- Understand looping as a form of iteration
- Learn how and when to use the Looping blocks 'repeat', 'while', and 'for'
- Apply the above knowledge and skills to create a unique program that uses iteration and looping as an integral part of the program

Lesson Plan Structure

- Introduction: Lather. Rinse. Repeat.
- Unplugged Activity: Walk a Square pseudocode
- Micro:bit Activities: Code a Sprite to Walk a Square, Travelling Light, Micro:bit Alarm!
- Project: Get Loopy!
- Project Mods: Use servo motors to add a motion element to the project
- Assessment: Rubric
- Standards: Listed

Introduction

In computer programming, iteration is the repetition of a sequence of code. A loop is a form of iteration. A loop repeats code until a certain condition is met.

Questions for the students:

- Do you use shampoo to wash your hair? *Most will say 'Yes'.*
- Have you ever read the instructions on a bottle of shampoo? *Most will say 'No'.*

Most of us have never read the instructions on a bottle of shampoo, because we already know how to use shampoo.

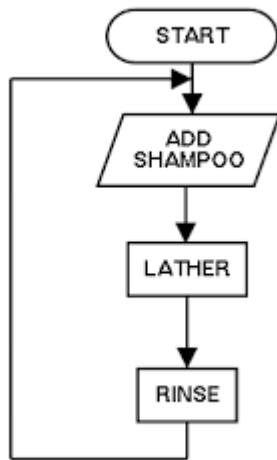
What algorithm could you write for shampooing your hair?

Example:

- 1) *Wet hair.*
- 2) *Apply shampoo to wet hair*
- 3) *Scrub shampoo into hair*
- 4) *Rinse shampoo out of hair*

If you did read the instructions on a bottle of shampoo, you may read similar instructions as the ones you just wrote with one added step at the end. That step is 'Repeat.'

How does this one extra step affect the algorithm?



.|

In computer programming, this is known as the 'shampoo algorithm' and is an example of a loop. It is also an example of an 'infinite' or 'endless' loop as the algorithm keeps repeating with no condition that ends the looping.



DBwebsolutions.com

'Rinse. Repeat.' has even become a meme and made its way into modern song lyrics.

What other common activities involve repetitive actions? *Examples: Singing (choruses repeat), dancing, school cheers, walking and running, exercise routines...*

Optional

Share with your students the history of 'Lather, Rinse, Repeat.'

Lather, Rinse, Repeat: Hygiene Tip or Marketing Ploy

By Lauren Goldstein

October 11, 1999

http://archive.fortune.com/magazines/fortune/fortune_archive/1999/10/11/267035/index.htm

(FORTUNE Magazine) – In Benjamin Cheever's novel The Plagiarist, a marketing executive

becomes an industry legend by adding one word to shampoo bottles: REPEAT. He doubles shampoo sales overnight.

This bit of fiction reflects a small yet significant eddy of U.S. consumer angst: If we REPEAT, are we or are we not playing into the hands of some marketing scheme? It turns out that in real life there's a reason you should repeat, or at least there used to be. In the 1950s, when shampoos began to be mass-marketed, we didn't wash our hair all that often--once or twice a week, as opposed to five times a week as most of us do now. Also, we used a lot more goop in our hair. It was the age of Brylcream and antimacassars, remember. Paul Wallace, the director of hair-care research and development for Clairol, says that when cleaning agents in shampoo came up against that amount of oil and goop, "it depressed the lather." A second application was needed to get the suds that consumers expected. Lots of suds mean that hair is already clean. Maybe too clean (there's no oil to break through), but consumers like it.

FORTUNE asked Frederic Fekkai, the noted and notably expensive New York City hairdresser, what he thought about the double lather. He says, "Yesterday I put oil on my hair for a different look and went to a restaurant where the smoke was horrible. This morning I realized I had to do two shampoos."

At any rate, Wallace says advances in shampoo technology mean that only one application of, for instance, Clairol's Herbal Essences is sufficient to break through the oiliest hair. The company has stricken the use of both REPEAT and REPEAT IF DESIRED from all Clairol products. Yet a lot of brands, like Suave by Unilever and L'Oreal, still say REPEAT. Others, like Unilever's Finesse and Revlon's Flex, opt for the less imperative REPEAT IF DESIRED. Procter & Gamble uses REPEAT IF NECESSARY on Pantene.

Getting consumers to wash twice can, of course, increase sales--in ways one might not imagine. Double sudsing leads to dry hair, Fekkai points out, and that means more beauty products! "When you do two shampoos, even if you don't usually use a conditioner, you have to use a little," he says. "The conditioner becomes very important." REPEAT. FOLLOW WITH CONDITIONER. Words Cheever's marketer could have retired on.
--Lauren Goldstein



From Wikipedia (https://en.wikipedia.org/wiki/Lather,_rinse,_repeat):

Lather, rinse, repeat (sometimes wash, rinse, repeat) is an idiom roughly quoting the instructions found on many brands of shampoo. It is also used as a humorous way of pointing out that such instructions if taken literally would result in an endless loop of repeating the same steps, at least until one runs out of shampoo. It is also a sarcastic metaphor for following instructions or procedures slavishly without critical thought.

Unplugged: Walk a Square

Objective

To reinforce the concept of iteration by having students act out the repeated steps of an algorithm in real life.

Overview

Students will give the teacher instructions to do a simple activity, then look for places where using iteration could shorten their code and make it more efficient.

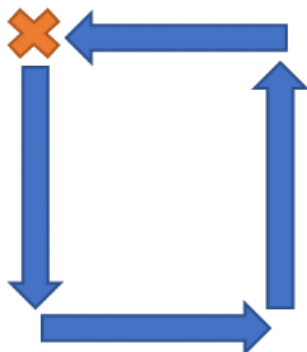


Process

- Place a chair in the front of the room.
- Stand at the back right side of the chair facing the students.
- Ask the students what instructions they could give you that when followed would lead you to walk around the chair, ending up just as you started. You may want to demonstrate what this would look like by walking around the chair.
- Tell the students you can only process one instruction at a time, so their algorithm needs to be step-by-step.
- As students suggest instructions write them on the board or wherever everyone can see them.

Their pseudocode will probably end up looking something like this:

- 1) Step forward
- 2) Turn left
- 3) Step forward
- 4) Turn left
- 5) Step forward
- 6) Turn left
- 7) Step forward
- 8) Turn left



- Go ahead and follow their algorithm to prove that it works.

But that's eight lines of code! Tell students that the same instructions can be written using just three lines of code. If they have not noticed already, have students look for places where the code repeats.

- Tell them that whenever you have code that repeats, you have an opportunity to use a loop to simplify your code.
- Prompts:
 - What lines are repeated? *1) Step forward. 2) Turn left.*
 - How many times are they repeated? *Four*
 - So how could we rewrite this code? Students will suggest a version of the following:
Repeat 4 times:
 - 1) Step forward
 - 2) Turn left
- Go ahead and follow their revised algorithm to prove that it works.

There! They have just rewritten eight lines of code as three lines of code, by using a loop. The 'repeat' command creates a loop. The code within the loop gets repeated a certain number of times until a condition is met. The condition in this algorithm is that the code in the loop is repeated 4 times. Once this condition is met, the program exits the loop.

This is a great opportunity to have the students think of the benefits of having fewer lines of code. *Some possible reasons: Less typing, saves time, fewer chances of making a mistake, easier to read the code, fewer lines of code to debug...*

Notes

- Depending on the particular class, you can make this exercise more challenging, by requiring the students to be more specific in their instructions.
For example: Step forward 14 inches (you can have students actually measure the exact distance), turn left 90 degrees...

Activity: Loops Demos

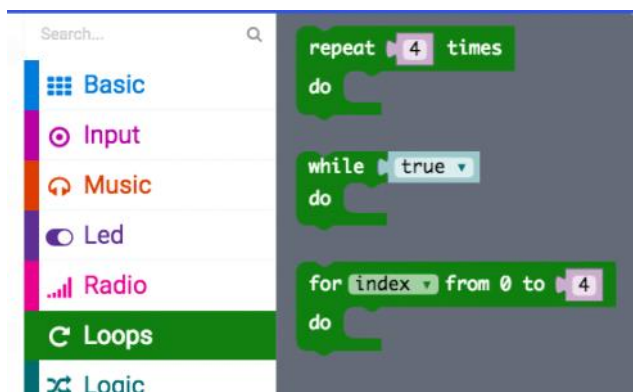
Microsoft MakeCode has three different loop blocks:

- 'Repeat' block
- 'While' block
- 'For' block

To start, the students can code the same algorithm they created in the unplugged activity using a loop.

'Repeat' block

Code a Sprite to walk a square. Have students click on the Loops category in the Toolbox, and look at the three choices available.



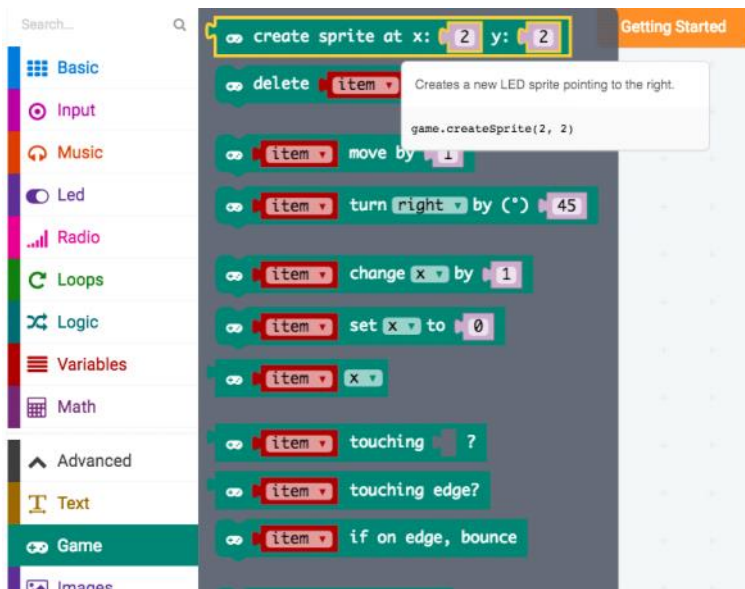
The very first one is the 'repeat' block! Have students drag the repeat block to the coding Workspace. They'll notice that this block takes a **parameter**.

A **parameter** is a type of variable used as input to a function or routine. In this case, the parameter tells the repeat block how many times we want the code within the block to repeat.

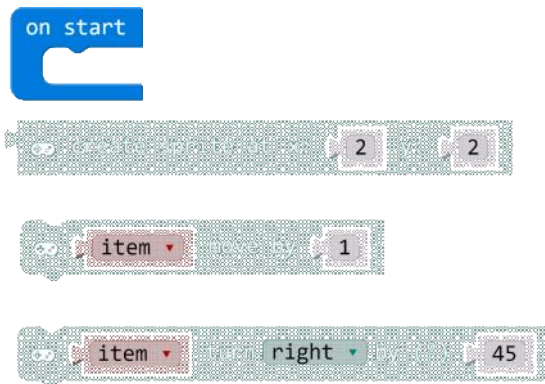
For now, we'll leave the parameter at 4.

To create a **sprite** that will walk a square:

- Click on the Advanced category in the Toolbox. This will open up a more advanced menu of blocks.
- Click on Game category, and drag a 'create sprite' block to the coding workspace.



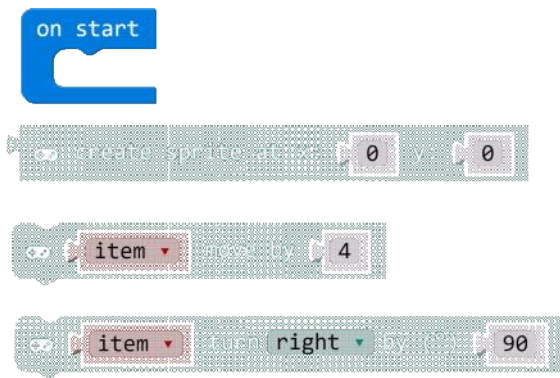
- We'll need two more blocks from the Game menu. Referring to their 'Walk a Square' pseudocode, see if the students can find the blocks they need for moving their sprite and turning their sprite.
- Drag out a 'move by' block and a 'turn right by' block.
- They now have these blocks in their coding workspace.
- For this project, they can delete the default 'forever' block.



Time to fix those default parameter values!

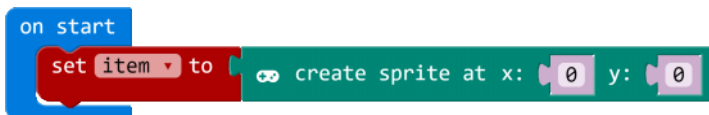
- We want our sprite to start in the top left corner of the micro:bit screen, so change the parameters for both **x** and **y** to zero.
- To make the sprite move from one side of the screen to the other (as though walking around a chair), change the move by parameter to **4**.
- To make the sprite turn to walk a square, change the 'turn right by' degrees to **90**. For now, it's OK to leave the sprite turning right instead of left as we did in our pseudocode.

Your blocks now look like this:



Notice that the blocks are all grayed out. That's because we have not yet attached them to any event handlers.

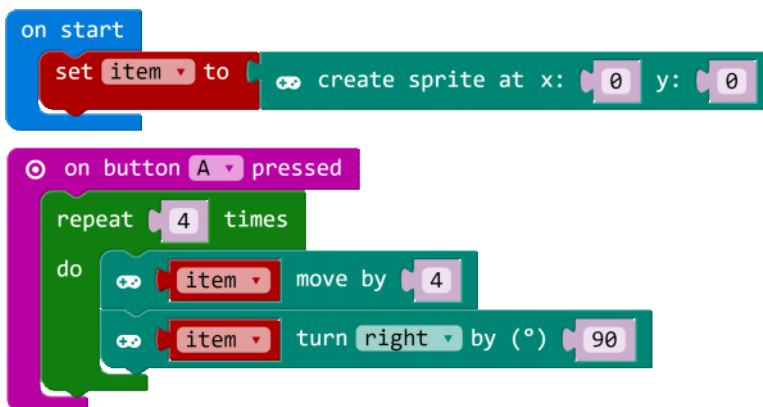
- On start, we want the sprite to appear. To make this happen, go to the Variables menu and drag a 'set item to' block to the coding window.
- Place the 'set item block' into the 'on start' block.
- Attach the 'create sprite' block to the 'set item' block



```
let item: game.LedSprite = null
item = game.createSprite(0, 0)
```

You should now see the sprite appear in the top left of the microbit simulator.

- To add more control for when our sprite moves, drag a 'on button A pressed' block from the Input menu.
- Place the 'repeat' block into the 'on button A pressed' block
- Place the 'move by' block into the 'repeat' block
- Place the 'turn right by' block into the 'repeat' block just under the 'move by' block.



```
input.onButtonPressed(Button.A, () => {
  for (let i = 0; i < 4; i++) {
    item.move(4)
    item.turn(Direction.Right, 90)
  }
})
```

```
}  
})
```

Go ahead and run the program. Make the sprite move by pressing button A.

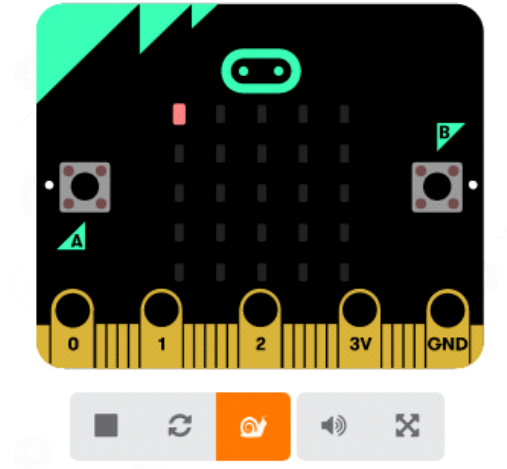
What happened? Did you see the sprite move? No?

Slo-Mo

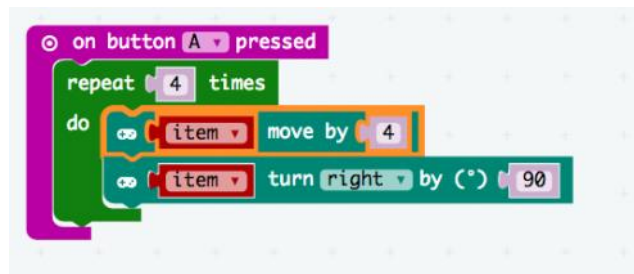
A helpful feature of Microsoft MakeCode is "Slo-Mo", or slow-motion mode.

- Click on the snail icon under the micro:bit simulator.

This will slow down the execution (running) of the program, and highlight parts of your code so you can see step-by-step, which line of code is being processed.



Now run your program several more times. Do you see the different lines of your code highlighted as the program runs? Do you see the sprite move?



Slo-Mo in Blocks

```
let item: game.LedSprite = null  
input.onButtonPressed(Button.A, () => {  
  for (let i = 0; i < 4; i++) {  
    item.move(4)  
    item.turn(Direction.Right, 90)  
  }  
})  
item = game.createSprite(0, 0)
```

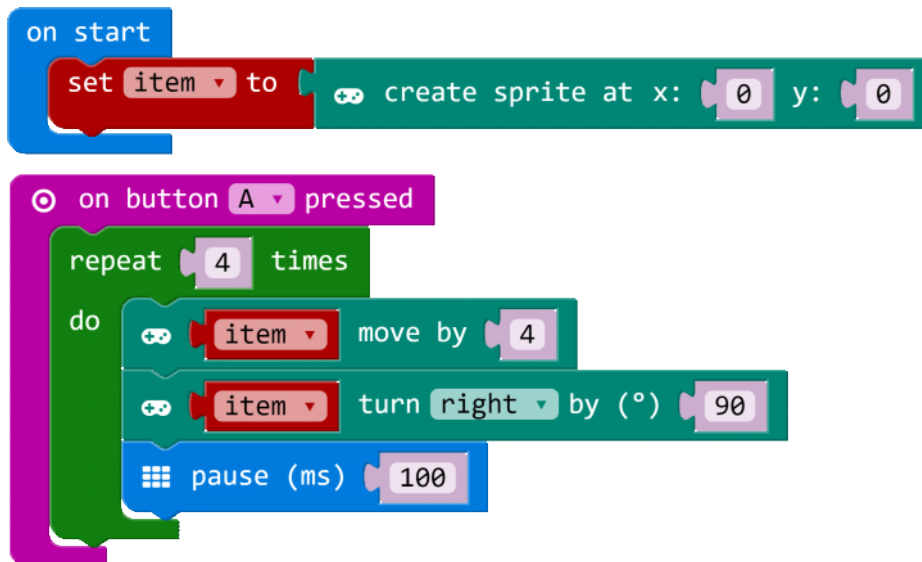
Slo-Mo in JavaScript

So, the code is running and the sprite is moving! Sometimes we forget just how fast computers are. So that we can see the sprite move even in 'regular' mode, lets add a pause to our program

right after each time the sprite moves. This will give our human eyes a chance to see it move.

- Click the snail icon again to turn off Slo-Mo.
- From the Basic Toolbox category, drag a 'pause' block to the coding window and add it to our 'repeat' block right after the 'turn right by' block.

Your final program should look like this:



```
let item: game.LedSprite = null
input.onButtonPressed(Button.A, () => {
  for (let i = 0; i < 4; i++) {
    item.move(4)
    item.turn(Direction.Right, 90)
    basic.pause(100)
  }
})
item = game.createSprite(0, 0)
```

[WalkaSquare](#)



Run your program again. Now we can see the sprite move. It still moves pretty quickly, but at least we can see it move.

If there is time, let the students experiment with changing the parameters to see how these changes affect their program.

We just used the first of the 3 different types of Loop blocks available to us. What about the other 2 loop blocks, 'while' and 'for'?

'For' block: Traveling Light

The 'for' block is useful when you have a variable in your loop that you want to change by a fixed amount within a specific range each time through a loop. What does this mean? Let's look at an example.

Let's make an led light move across the entire display from left to right, top row to bottom row.

Our pseudocode for the first row might look like this:

```
Turn led x:0, y:0 on
Pause
Turn led x:0, y:0 off
Pause
Turn led x:1, y:0 on
Pause
Turn led x:1, y:0 off
Pause
Turn led x:2, y:0 on
Pause
```

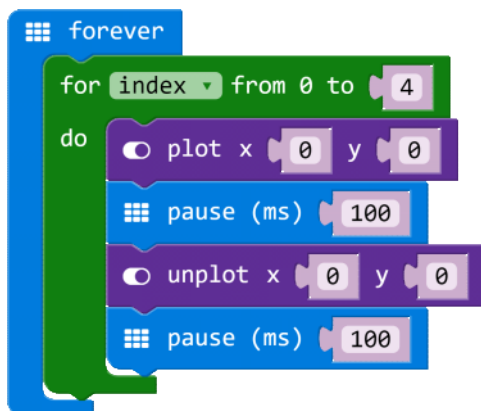
Turn led x:2, y:0 off
Pause
Turn led x:3, y:0 on
Pause
Turn led x:3, y:0 off
Pause
Turn led x:4, y:0 on
Pause
Turn led x:4, y:0 off

That's a lot of code, most of it repeated. Perfect for a loop.

- What is the only variable that is changing in this pseudocode? *The value of the x coordinate.*
- How much is the value of the x coordinate changing each time? *The value of the x coordinate is changing by 1 each time.*
- What is the range of values for the x coordinate? *The range of values for the x coordinate is 0 through 4.*

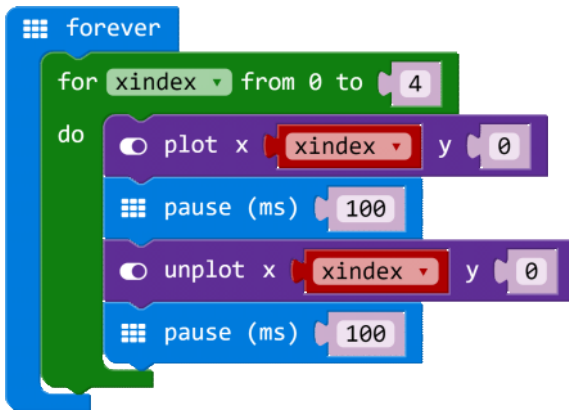
Now let's code!

- From the Loops Toolbox drawer, drag a 'for' block to the coding workspace.
- Since we'll be changing the value of the x coordinate, make a new variable, named **xindex**.
- We'll plot and unplot the leds to turn them on and off. From the Led Toolbox drawer, drag a 'plot' block and an 'unplot' block to the coding workspace.
- From the Basic Toolbox drawer, drag two 'pause' blocks to the coding workspace.
- Place the following blocks into the 'for' block: the 'plot' block, a 'pause' block, the 'unplot' block, the second 'pause' block.
- Place the 'for' block inside a forever block.



Let's look at the parameters.

- Change the 'index' in the 'for' block to the 'xindex' variable we made.
- Change the value of the x coordinates in the plot and unplot blocks to this same variable.



```

let index = 0
basic.forever(() => {
  for (let xindex = 0; xindex <= 4; xindex++) {
    led.plot(xindex, 0)
    basic.pause(100)
    led.unplot(xindex, 0)
    basic.pause(100)
  }
})

```

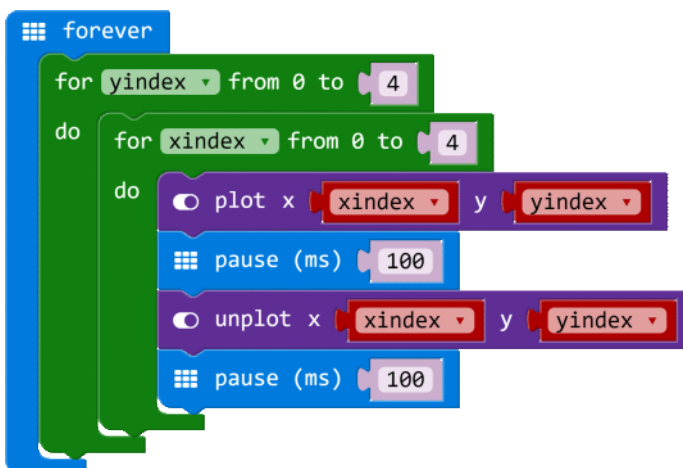
We can use the default values for the rest of the parameters.

You should now see a light moving from left to right along the top row of the micro:bit simulator.

To make our pattern continue through all the leds, we can change the value of the y coordinate as well.

To do this efficiently, using the fewest lines of code, we can even put a loop inside a loop. Loops inside other loops are known as **nested loops**.

- So that we can change the value of the y coordinate, make a new variable, named **yindex**.
- Drag out another 'for' block from the Loops Toolbox drawer.
- Place this new 'for' block around our original 'for' block, all within the forever block.
- Change the 'index' in the outer 'for' block to the 'yindex' variable we made.
- Change the value of the y coordinates in the plot and unplot blocks to this same variable.




```

let index = 0
let yindex = 0
basic.forever(() => {
  for (let yindex = 0; yindex <= 4; yindex++) {
    for (let xindex = 0; xindex <= 4; xindex++) {
      led.plot(xindex, yindex)
      basic.pause(100)
      led.unplot(xindex, yindex)
      basic.pause(100)
    }
  }
})

```

There! With only a half dozen or so lines of code, we have made our light travel through all the coordinates on the micro:bit screen.

- Check: Make sure the students can read this code.

Here is what is happening to the values of the x & y coordinates as the program steps through each line and loop inside the forever block:

1. In the outer of the two for loops, the value of the y-coordinate is set to 0.
2. The nested inner loop then sets the value of the x-coordinate to zero.
3. The corresponding led (x:0, y:0) is plotted and then unplotted.
4. Then the value of the x-coordinate is increased by 1 and step #3 runs again with the coordinates now (x:1, y:0).
5. Then the value of the x-coordinate is increased by 1 again and step #3 runs again with the coordinates now (x:2, y:0).
6. The inner loop keeps running like this until it has completed its loop with the value of the x coordinate now 4.
7. With the inner loop complete, the program now runs the second iteration of the outer loop, increasing the value of the y-coordinate by 1, then back to the inner loop which runs 4 more times stepping through values for x from 0 through 4.

Have the students use the Slo-Mo mode to watch the program step through the loops.

- By the end of the program run, how many times has the inner loop executed? 25
- Other than knowing that there are 25 LEDs and each is lit up once, how can you figure this out? *The outer loop loops 5 times altogether, once for every value of the y coordinate from 0 through 4. Each time the outer loop runs, the inner loop runs 5 times, once for every value of the x coordinate from 0 through 4. 5 runs of the outer loop x 5 runs of the inner loop = 25 times the inner loop executes.*

Mods

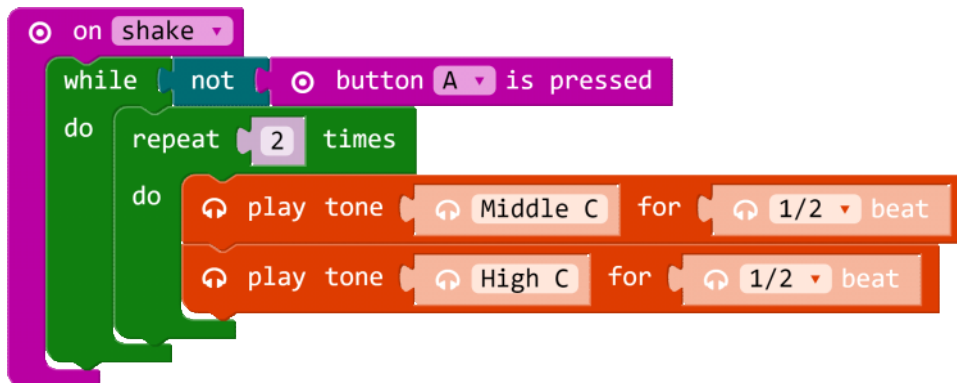
- If there is time, let the students experiment with changing the parameters to see how these changes affect their program.
- What happens if you switch the positions of the nested loops, so the outer loop loops through the xindex values and the inner loop loops through the yindex values?
- What happens if you remove the 'unplot' block and the 'pause' block below it?

'While' block: Micro:bit Alarm!

The while block is useful when you want your program to loop until a certain event happens or a different condition is met.

For example, maybe you want an alarm to sound if someone shakes your micro:bit! In order to turn the alarm off, you press the button A. Until you press the button, the alarm should continue to sound!

You can use a 'while' block with a nested 'repeat' block like this:



```
input.onGesture(Gesture.Shake, () => {
  while (!(input.buttonIsPressed(Button.A))) {
    for (let i = 0; i < 2; i++) {
      music.playTone(262, music.beat(BeatFraction.Half))
      music.playTone(523, music.beat(BeatFraction.Half))
    }
  }
})
```

[Alarm](#)



- Can you read what this code does?
- Can you write out pseudocode that describes what this code does?

Example Pseudocode:

*When someone shakes the microbit, while button A is not pressed, play the two tone alarm twice.
Keep playing the alarm tones until the user presses the A button.*

To use sound with your micro:bit, you will need to connect it to some speakers or headphones.
See how to do this here: <https://pxt.microbit.org/projects/hack-your-headphones>

Project: Get Loopy!

There are many different ways to use the three types of loop blocks.

Recall the different common repetitive actions you thought of back at the beginning of this lesson.

- How will you use loops to create something useful, entertaining, or interesting?
- What might you make?

Here are some suggestions:

- Create an animated gif (looping image that changes) and add music that matches.
- Create animation that repeats for one of the melodies included in Make Code (like Happy Birthday).
- Create different animations that run when different buttons are pressed.
- Create an alarm that includes sound and images. What will set the alarm off? What will make the alarm stop sounding?
- Use servo motors to create a creature that dances and changes its expression while a song plays.

Example



Hat Man Project

Hat Man Videos
[micro:bit Hat Man](#)



[micro:bit Hat Man - inside](#)



This project uses the micro:bit light sensor to display a happy face when it is sunny, and a frowning face when it is dark. The micro:bit is connected to a servo mounted on the inside of the container, and the smile and frown are attached to plastic coffee stirrers with tape and hot glue.

Reflection

Have students write a reflection of about 150–300 words, addressing the following points:

- Explain how you decided on your particular "loopy" idea. What brainstorming ideas did you come up with?
- What type of loop did you use? *For*, *While*, or *Repeat*
- What was something that was surprising to you about the process of creating this program?
- Describe a difficult point in the process of designing this program, and explain how you resolved it.
- What feedback did your beta testers give you? How did that help you improve your loop demo?

Assessment

	4	3	2	1
Loops	At least 3 different loops are implemented in a meaningful way	At least 2 loops are implemented in a meaningful way	At least 1 loop is implemented in a meaningful way	No variables are implemented.
Variables (parameters)	All variable names are unique and clearly describe what information values the variables hold.	The majority of variable names are unique and clearly describe what information values the variables hold.	A minority of variable names are unique and clearly describe what information values the variables hold.	None of the variable names clearly describe what information values the variables hold.
Sound, display, and motion	Uses sound, display, and motion in a way that is integral to the program	Uses a only two of the required element in a way that is integral to the program	Uses a only one of the required element in a way that is integral to the program	None of the required elements are used.
Micro:bit program	Micro:bit program: <ul style="list-style-type: none"> • Uses loops in a way that is integral to the program • Compiles and runs as intended • Meaningful comments in code 	Micro:bit program lacks 1 of the required elements	Micro:bit program lacks 2 of the required elements	Micro:bit program lacks 3 or more of the required elements
Collaboration reflection	Reflection piece includes: <ul style="list-style-type: none"> • Brainstorming ideas • Construction • Programming • Beta testing 	Reflection piece lacks 1 of the required elements.	Reflection piece lacks 2 of the required elements.	Reflection piece lacks 3 of the required elements.

Standards

CSTA K-12 Computer Science Standards

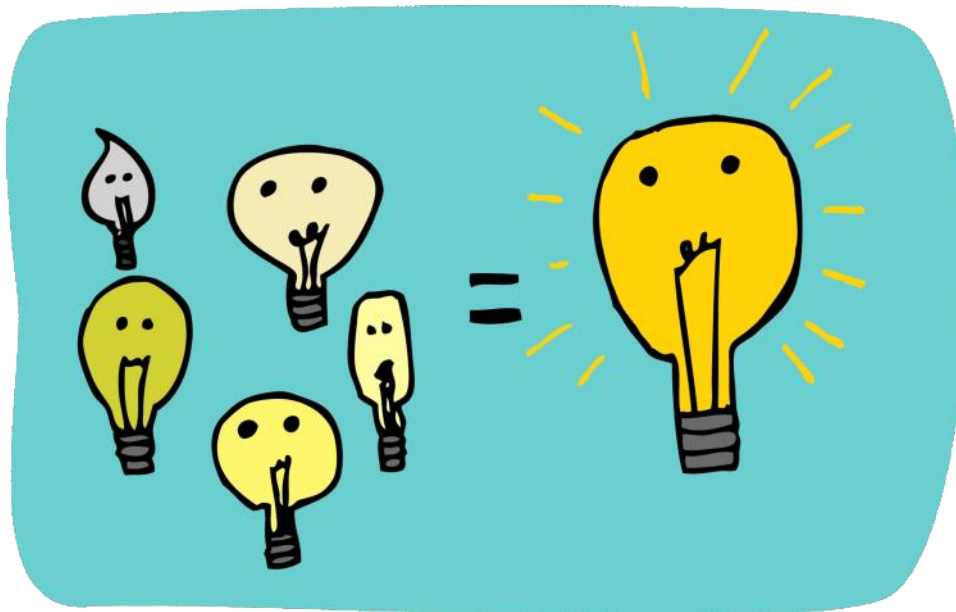
- CL.L2-05 Implement problem solutions using a programming language, including: looping behavior, conditional statements, logic, expressions, variables, and functions.
- CL.L3A-03 Explain how sequence, selection, iteration, and recursion are building blocks of algorithms.

Introduction

In this unit, we will be reviewing the concepts we covered in the previous weeks, and providing some ideas for an independent “mini-project” students can focus on in the next several classes. We will also introduce a framework for keeping students accountable to the work they are doing individually and in groups, and providing a rubric for assessment of the development process, as well as the finished product.

It is important to allow students to practice accounting for the work they are doing on a short “mini-project” like this, so that when they move on to an independent project spanning multiple weeks, it will be easier for you to keep track of what everybody is doing.

It also reinforces the important idea that how you solve problems is at least as important to learning as whether you solved them at all (or even got the right answer). Programming is a process of patient problem-solving, and finding ways to value, acknowledge, and reward the problem-solving process is an important part of assessment.



Review



Take this time to review the concepts we have covered so far.

Making

The micro:bit is very effective at bringing real things to life. It can be supported in a cardboard holder, attached to a wand, or even sewn into fabric. The design thinking process is a helpful way to gather more information about the person who will be using whatever you are designing.

Processing and Algorithms

The code you write for the micro:bit processes data from its inputs, and outputs it in some way. An algorithm is a series of specific instructions, or steps, that solve a problem or accomplish a task.

Variables

Variables store information so that it can be accessed or referenced later. Some variables hold information that changes, and some hold information that stays constant. It is important to name your variables with something that explains what type of information it holds. Using variables in your code allows you to create algorithms that use mathematical operations to perform the same calculations every time, even when the values of your variables are different.

Conditionals

Conditional statements tell the computer when to do something. They are used to create branches, or decision points, where a program can choose one path or the other based on the values of certain variables, or based on data from the micro:bit's inputs. Conditional statements can be nested inside one another so that both conditions must be true in order for the enclosed statements to run.

Iteration and Looping

Portions of your code can be made to run over and over by using a Repeat or a For block loop. This allows you to iterate over several different variables, or items in a group, and do something to each of them. You can also combine a conditional statement and a loop by using a While block, which will repeat until a certain condition becomes true.

Project: Mini-Project

This project takes approximately a week to complete. Most of that time is spent working on the project in a makerspace or art classroom.

The mini-project is an opportunity for students to design a project that serves a purpose by solving a problem or filling a need. It is also an opportunity to do two things:

- Show what you know
- Learn something new

Ideally, there should be a maker component to this project. This is a real world component that works with the code on the micro:bit to do something unique.

Students are asked to each propose an original independent project. Students are allowed to work on the same idea, but they cannot turn in the same code. They can, and should work collaboratively, solving the same kinds of problems together, but the projects they turn in should be unique and original.

Showcasing Student Work

Students will be showing their work regularly to each other in informal ways. Think about also organizing a day or an evening when parents, administrators, or others from the community are invited to come and view the students' projects.

We find that a "science fair" type of setup works well here, with students stationed at their own tables, showing off and demonstrating their project. An event like this works well for these reasons:

- A real world audience for the work students have done can be very motivating
- It is a chance for people who are not familiar with the micro:bit to appreciate the finished product
- It provides good feedback to students about how someone interacts with their product
- It is a chance to have real conversations with the people behind the product, rather than just viewing the product on display by itself
- Finally, and most importantly, it is a chance to bring the community together to celebrate the great work all of your students have done!

Assignment

- Create an original project using the micro:bit.
- Incorporate a physical component to the project.
- Demonstrate the use of one of the following concepts:
 - Input / Processing / Output
 - Variables
 - Simple Circuits
 - Iteration/Loops
 - Conditional Statements

Project Ideas

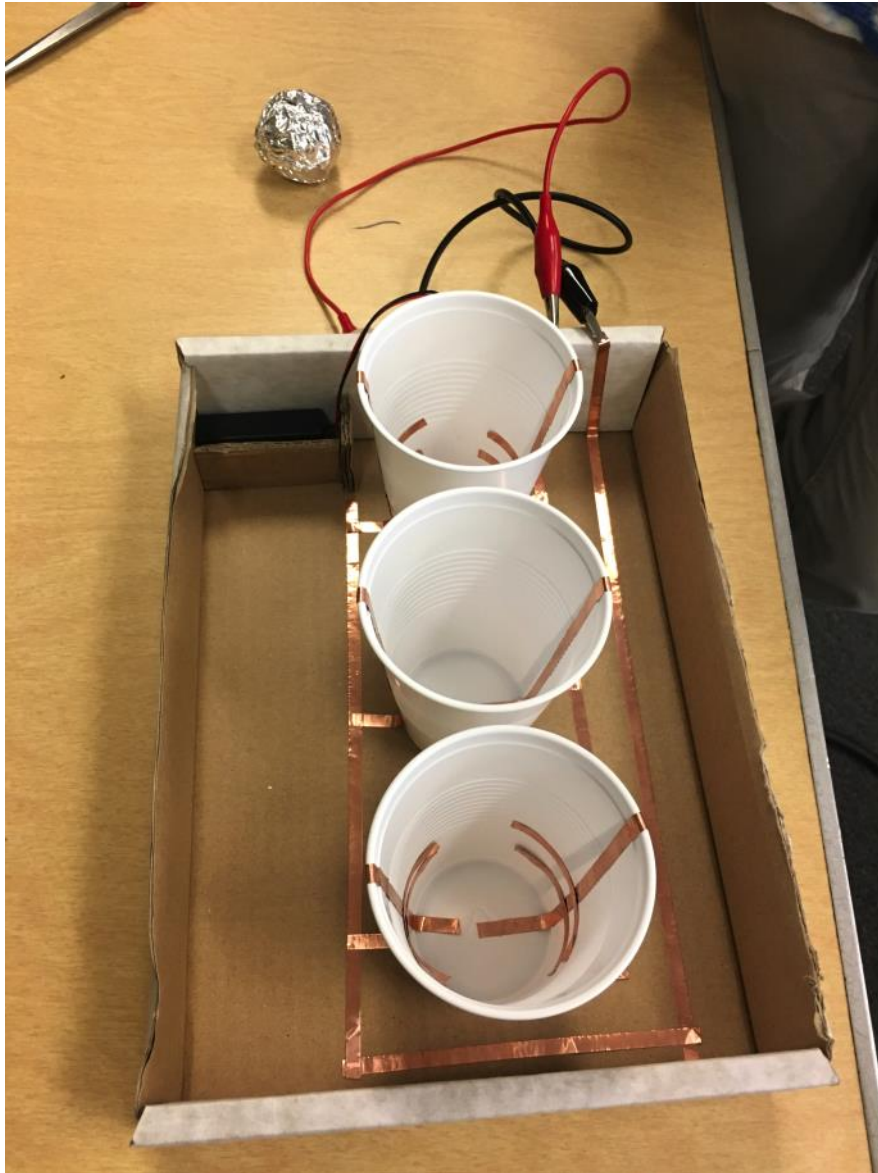
- Make a "New and Improved" Fidget Cube
- Make a Moving Monster
- Make a musical instrument
- Fishing Game
- Make an Air Guitar (uses while loop to do tempo and pitch)
- Screensaver
- Screensaver that uses other inputs to draw

- Interactive book
- Binary Clock or some other way to represent numbers visually

View projects at the following sites for inspiration:

- <http://make.techwillsaveus.com/bbc-microbit>
- <http://microbit.org/ideas/>
- <https://twitter.com/MicroMonstersUK>
- <https://pxt.microbit.org/projects>

Examples



Toss the Ball

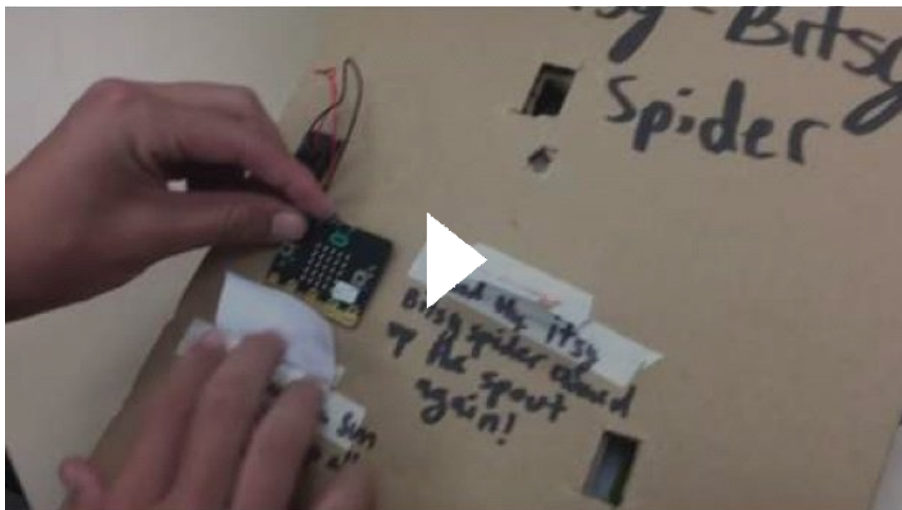
This is a skill game in which an aluminum foil ball is thrown into a plastic cup. Copper tape lining the sides and bottom of the cup completes the circuit when the ball touches it.

[micro:bit Bullseye Project](#)



This is a skill game in which tennis balls are thrown underhand at one of the three rings, which are lined with aluminum foil so they complete a circuit underneath when the ball makes contact with the ring.

[micro:bit Storybook](#)



This is a prototype of a storybook that could use the micro:bit to display animations for part of the story. Copper tape is used on the underside of the paper flaps to make contact between the GND pin and each of the other pins in sequence.

Work Logs

Because students are working on the projects in class, and much of the benefit comes from working together to solve problems, they should account for the work they are doing by writing a work log.

A work log is a short, bullet point list of what they worked on, and how long it took. Stick to the facts. It shouldn't take more than thirty seconds or so to write up a work log. Students should do one for every class. A shared Microsoft OneNote notebook is a great way to keep a work log that students can update regularly. Alternately, you might use a collaborative shared document, or your classroom management system, or even e-mail.

Sample Work Log

April 11

20 min. Created code that reacts when pins P0 and P1 are pressed.

10 min. Talked with Mr. Kiang about how to attach wires so they won't fall off

20 min. Put target back together with pins

10 min. Helped Cody with attaching his scoreboard

Reflection

At the end of the week, students should compose a final reflection that summarizes the process of their learning over the course of the week. They should go back through their work logs and talk about the following:

- Talk about one challenge you faced in creating this project, either a challenge in coding or in making the artifact. How did you overcome this challenge?
- What did you demonstrate that you already knew?
- What was the new thing you learned in order to make this? How did you learn about it?
- Who in the class provided help to you along the way? How?
- Describe one specific thing you are proud of in this project.
- What would you do differently next time?
- If you had another week to work on this project, what might you add or improve?

Sample Reflection (excerpt)

"I spent this week finishing up little details with my program, making it work better and more user friendly. The part that surprised me the most was the little things that kept popping into my head, little suggestions that could potentially be good to add, but might not be necessary or even useful. At the beginning of the assignment, I just added them as quickly as I thought of them, but as the project neared the midpoint and conclusion, I find myself considering if I actually need them (as previous additions have been since quickly deleted). Another thing that I find interesting about this is that it is a rather specialized project. Not many people would use it except for me. However, this is supposed to be easily used by other people, so I have to take them into consideration as I design the project. I also realized that I had, at some point, broken part of my code without realizing it, so I now have to fix part of it. The reason that it is a problem is because I added a lot of code at once without deleting it, which is unfortunate. Next time I will add small amounts of code and test it first."

Assessment

	4	3	2	1
Code - Show what you know	Code very effectively demonstrates the use of previous concept(s). Variable names are unique and clearly describe what information values the variables hold. Code is highly efficient.	Code only partially demonstrates previous concepts, and/or is not efficient.	Code only partially demonstrates previous concepts, and/or is not efficient, variable names not clear.	Code does not demonstrate previous concepts, is not efficient, variable names not clear.
Code - Show something new	Code very effectively demonstrates the use of new concept(s). Variable names are unique and clearly describe what information values the variables hold. Code is highly efficient.	Code only minimally demonstrates new concepts, and/or is not efficient.	Code only minimally demonstrates new concepts, and/or is not efficient, variable names not clear.	Code does not demonstrate new concepts, is not efficient, variable names not clear.
Maker component	Tangible component is tightly integrated with the micro:bit and each relies heavily on the other to make the project complete.	Tangible component is somewhat integrated with the micro:bit but is not essential.	Tangible component does not add to the functionality of the program.	No tangible component
Work Logs	All work logs submitted on time,	One late or missing	Two late or missing	More than two late

	and accurate	work log and/or work logs not accurate nor sufficiently detailed.	work logs and/or work logs not accurate nor sufficiently detailed.	or missing work logs and/or not accurate nor sufficiently detailed.
Reflection	<p>Reflection piece describes:</p> <ul style="list-style-type: none"> • Development Process • Something new • Something proud of • Future mods 	Reflection piece lacks 1 of the required elements.	Reflection piece lacks 2 of the required elements.	Reflection piece lacks 3 of the required elements.

Activity: Collaboratively Independent

Teachers want their students to collaborate on projects but they also want to be able to hold them accountable for getting their work done. Many teachers struggle with assessing exactly how much each individual contributed to a group project, as well as making sure that everyone does his or her “fair share”.

The Mini-Project (and the Final Project) are not group projects. Students are asked to propose their own independent project and are expected to get it done. But they are not on their own in this process! We build in frequent opportunities for students to collaborate and share the collective knowledge of the class as they go. We ask them to be “collaboratively independent.”

Here is how we structure our classes:

Beginning of class

For groups of 15 or so, have students each briefly (no more than 30 seconds or so) report on their progress in front of the group:

- One-line description of project
- Their progress so far
- Something they are going to work on figuring out today

It is important that everyone else is listening to each project and volunteering their help or solutions if they are figuring out the same thing or if they have solved that problem in a previous class.

Example:

I'm working on a pinball machine. So far I have done the board and the ramp. Today I am going to be working on wiring the bumpers so that when the ball hits the bumper, the micro:bit detects it and displays the score.

Sample response from a classmate:

Yesterday I wired up my targets so that when you throw a ball it keeps score. I can show you how I did it.

Ideally students who are working on projects should be aware of what other students are working on and what they are figuring out. It creates more opportunities for collaboration in the classroom and can encourage students to seek help from each other rather than all waiting in line to talk to you.

It's important to hear from everybody but it shouldn't take more than five or ten minutes. For groups larger than 15 or 20 students, you may want to split them into two or three larger groups and have them report out to each other.



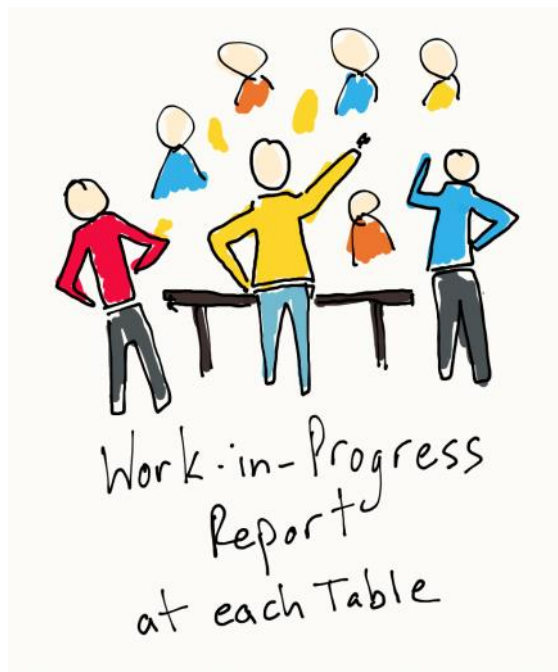
During Class

This is a time to circulate and check in with students individually, starting with those students who seem to still be stuck from last time. For the most part, students should be working on their projects in small groups, helping each other wherever and whenever possible.

End of Class

About ten minutes before the end of class, you can have students do a “work-in-progress” report. Gather the students together and have them move from table to table while each student presents one thing that he or she figured out during the class. This is really an informal presentation, and it is understood that it is not finished at all; it is still a “work in progress.” But everyone needs to show something, and the entire group needs to move as one throughout the classroom, almost like physicians making rounds in a hospital. This is an important way to spread ideas throughout the classroom, and to “cross-pollinate” with helpful tips and techniques.

Work-in-progress reports should be short, no more than twenty or thirty seconds. If you have a large class, you might divide the class into several large groups and have them present to each other.



Standards

CSTA K-12 Computer Science Standards

- CL.L2-03 Collaborate with peers, experts, and others using collaborative practices such as pair programming, working in project teams, and participating in group active learning activities.
- CL.L2-04 Exhibit dispositions necessary for collaboration: providing useful feedback, integrating feedback, understanding and accepting multiple perspectives, socialization.
- CL.L2-05 Implement problem solutions using a programming language, including: looping behavior, conditional statements, logic, expressions, variables, and functions.

Coordinate Grid and LEDs

This lesson introduces the use of coordinates to store data or the results of mathematical operations. It gives students practice programming for the LEDs of the micro:bit screen using coordinates. And introduces the basic game blocks of MakeCode.

Lesson Objectives

Students will...

- Understand that the 5 x 5 grid of LEDs on the micro:bit represent a coordinate grid with the origin (0,0) in the top left corner.
- Understand that the values of the x coordinates range from 0 through four and increase from left to right.
- Understand that the values of the y coordinates range from 0 through four and increase from top to bottom.
- Learn how to refer to an individual LED by its x & y coordinates.
- Learn how to plot (turn on) and unplot (turn off) individual LEDs and how to toggle between these two states.
- Learn how to check the current on or off status of an individual LED as well as check and set the brightness level.
- Apply the above knowledge and skills to create a unique program that uses coordinates as an integral part of the program.

Lesson Plan Structure

- Introduction: Coordinate Grid
- Unplugged Activity: Battleship
- Micro:bit Activities: Animation and Patterns
- Project: Screensaver or Game
- Assessment: Rubric
- Standards: Listed

Introduction

Through math class, most middle school students are already familiar with coordinate grids and mapping x and y coordinates on a plane. To review some terms:

Axes

- The basic coordinate grid a student learns has two axes,
 - an x-axis which runs horizontally and
 - a y-axis which runs vertically.

Origin

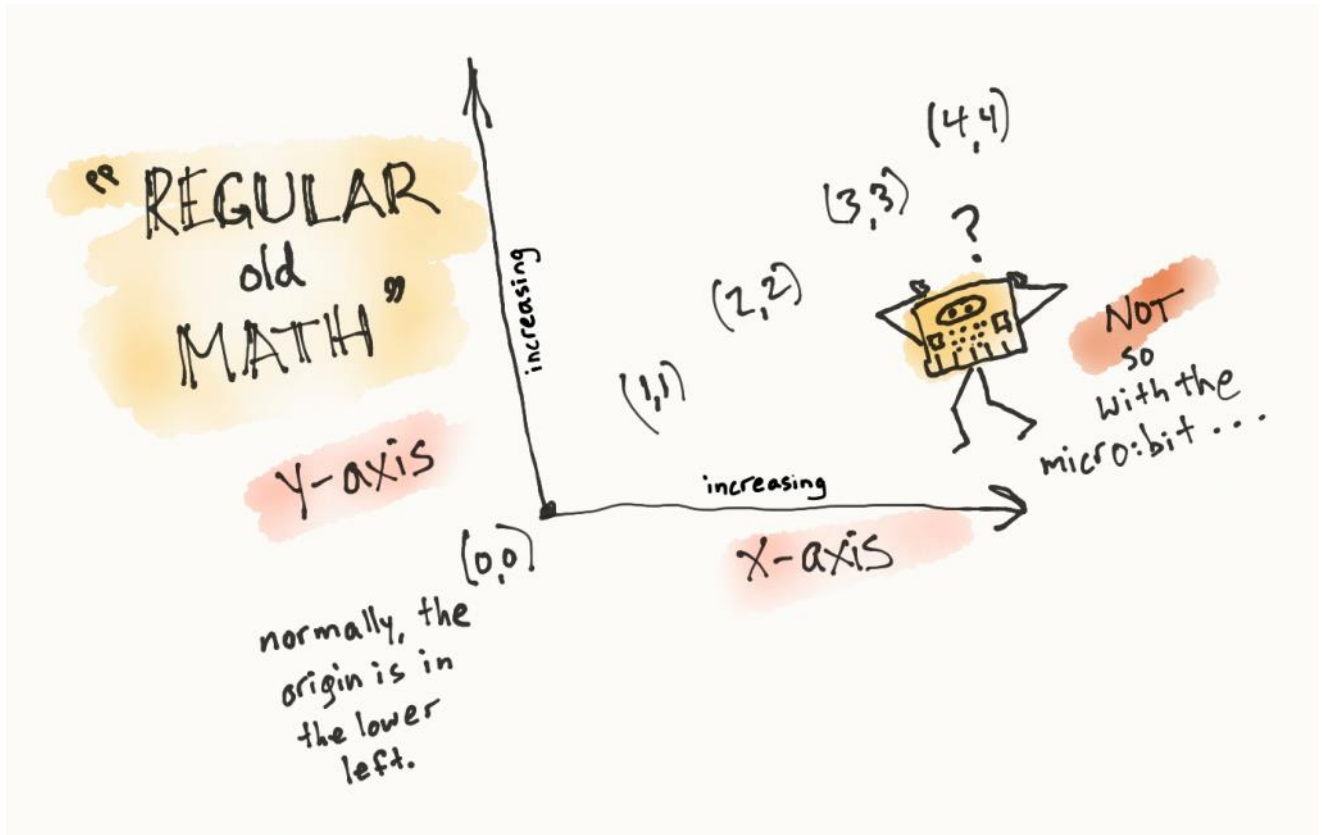
- These two axes meet at a point called the origin where both the x and the y values are zero.
- On this basic coordinate grid, the origin is in the lower left corner of the grid and has the coordinates (0,0).

Coordinate pair

- The first value in a coordinate pair is the x value and the second value in a coordinate pair is the y value.
- A simple way to remember which value comes first is to remember their order in the alphabet. The letter x comes before the letter y in the alphabet and the x coordinate comes before the y coordinate in a coordinate pair.

Coordinate value changes

- On a basic coordinate grid,
 - the value of the x coordinate increases left to right and is a measure of how many units a point is horizontally from the origin
 - the value of the y coordinate increases bottom to top and is a measure of how many units a point is vertically from the origin



Coordinate grid and JavaScript and the micro:bit

The 5 x 5 grid of LEDs on the micro:bit represent a coordinate grid with a horizontal x-axis and a vertical y-axis. It has an origin and you can refer to the position of the LEDs with coordinate pairs.

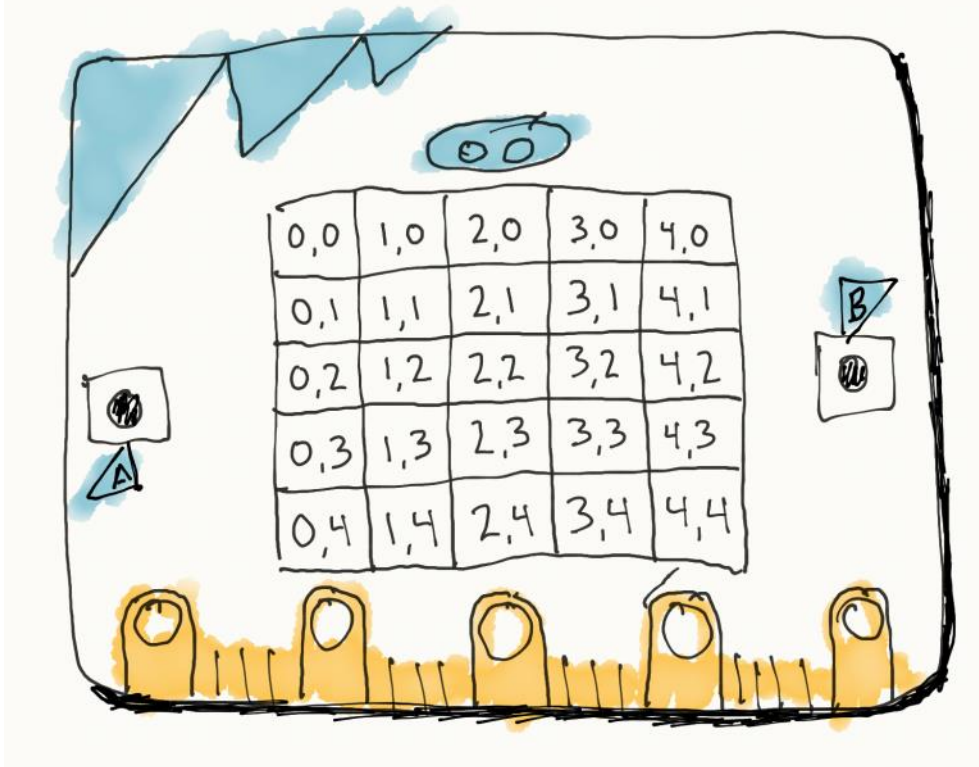
It is important however that the students understand the two major differences between the micro:bit LED grid and the coordinate grid that they are used to using in math class:

- the origin (0,0) is in the top left corner.
- the values of the y coordinates range from 0 through four and increase from top to bottom.

Note:

- The values of the x coordinates range from 0 through four and increase from left to right just as they do in the coordinate grids used in math class.

MICRO:BIT LED grid (x,y) coordinates



Sidebar material



(image credit: Wikipedia Commons)

René Descartes (1596-1650), was a French philosopher and mathematician who developed the coordinate system we use today. A story goes that while lying in bed, he noticed a fly on the ceiling. In wondering how he could describe the fly's exact position on the ceiling, he decided to use a corner of the ceiling as a reference point and then describe the fly's position as a measure of how far away from the reference point one would need to travel horizontally and then vertically to reach the fly. His coordinate system proved useful in many ways including creating an important link between the studies of algebra and geometry. Geometric shapes could now be described by points on a coordinate plane.

Unplugged: Battleship

The game Battleship is perhaps the most fun a student can have practicing using a coordinate grid. The original Battleship game is a 10x10 grid with numbers on one axis and letters on the other.

To help us practice using the correct coordinates for the grid of micro:bit LEDs, let the students play a smaller 5x5 version of Battleship using x and y coordinates instead of letters and numbers.

Have students make their own sets of 5x5 grids to reinforce the layout of the micro:bit grid.

Each student should make two grids. One grid is for placing their own ships and keeping track of their opponent's hits and misses and the other grid is for keeping track of their own hits and misses while trying to determine the location of their opponent's ships.

Player's grid: Mark where your ships are and keep track of your opponent's hits and misses.

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)

Opponent's grid: Keep track of your hits and misses while trying to locate your opponent's ships.

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)

Then pair the students to play against a partner. Each student's ships are hidden somewhere on their 5x5 grid. Students should be taking turns calling their shots using x and y coordinates, in the proper order. Their opponent will use those coordinates to plot the location of their shots.

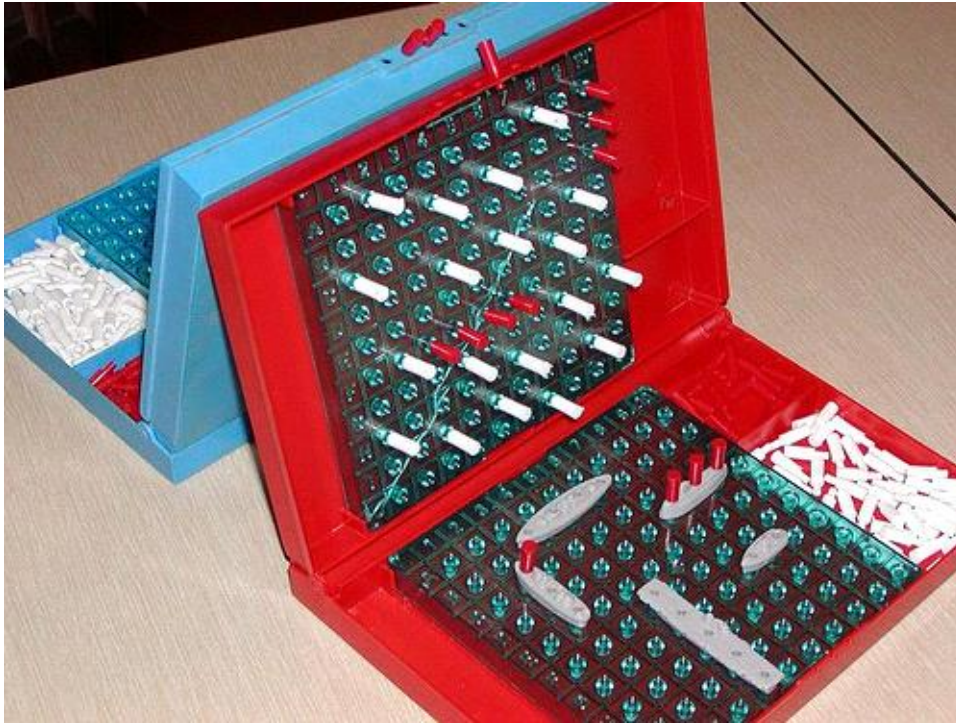
If a hit is recorded on a ship, then you say, "Hit". If the shot misses, you say, "Miss". If the entire length of a ship is hit, it is sunk and removed from play. Tradition dictates that the player announces, "You sank my battleship!"

Since their grid is only one quarter the size of the original Battleship grid, students can use fewer and smaller ships. For example, they could play with 3 ships, one each of size 3, 2, and 1.

The game can be played with just paper and pencils or you could use small tokens and markers, like coins, buttons, or paper clips to represent the ships.

Notes

- Place students' grids in sheet protectors or laminate them so they can be used again and again with white board (dry erase) markers.
- The official rules of Battleship are easily found on the internet. Modify them as needed for your particular class.



The original Battleship Board Game

Activity: Animation and Patterns

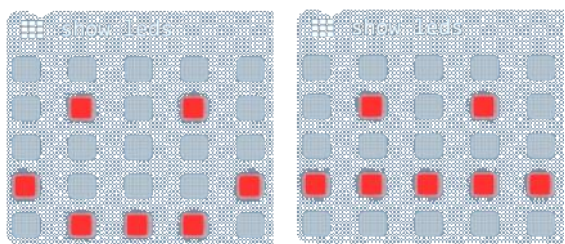
Guide the students to create programs using coordinates and LEDs. Each of these short exercises demonstrates how to use coordinates to control the LEDs. These programs can then be modified and used in the students' more complex projects.

- Smile animation - A short exercise in plotting and toggling LEDs to create a simple animation.
- Random Patterns generator - A short exercise using a loop to generate random LED patterns and then checking the status of a specific LED.
- Brightness - A short exercise in using the brightness settings for the micro:bit LEDs.

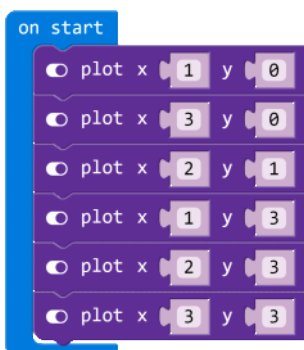
Smile Animation

A short exercise in plotting and toggling LEDs to create a simple animation.

- Though students can use the 'show leds' block for images and animation, there is another way to tell the micro:bit what LEDs to turn on and off using coordinates.
- We can still use the 'show leds' block to plan which LED coordinates to turn on
- Drag out a couple 'show leds' blocks from the Basic Toolbox drawer.
- Create a smiling face and a non-smiling face.



- From the LED Toolbox drawer, drag out 6 'plot x y' blocks.
 - Tip: you can also right-click on a block and select Duplicate to copy blocks
- Have the students compare the two face images and determine which LEDs are on in both images.
- Plot these LEDs using the correct (x,y) coordinates.
- When done, place these 'plot x y' blocks inside an 'on start' block.

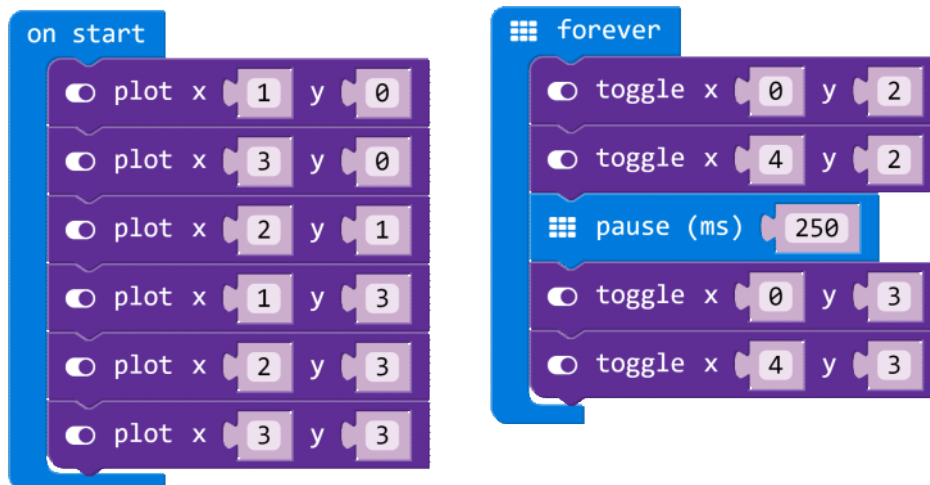


```
led.plot(1, 0)
led.plot(3, 0)
led.plot(2, 1)
led.plot(1, 3)
led.plot(2, 3)
led.plot(3, 3)
```

Now we can code for the 4 LEDs that change back and forth, on and off, as we switch from one face to the other and back again over and over.

- From the LED Toolbox drawer, drag out 4 'toggle x y' blocks.
- Replace the default values with the correct (x,y) coordinates.
The 'toggle x y' block will change the status of an LED from on to off or off to on.
- Place these 4 'toggle x y' blocks in a 'forever' block.
- Place the two 'toggle x y' blocks that create the smile first, followed by the two 'toggle x y' blocks for the non-smile.
- You may notice that the toggling happens too quickly. Let's slow it down a bit by placing a 'pause' block between the two pairs of 'toggle x y' blocks. Set the pause value to 250 milliseconds.

Here is the full program:



```
basic.forever(() => {
  led.toggle(0, 2)
  led.toggle(4, 2)
  basic.pause(250)
  led.toggle(0, 3)
  led.toggle(4, 3)
})
led.plot(1, 0)
led.plot(3, 0)
led.plot(2, 1)
led.plot(1, 3)
led.plot(2, 3)
led.plot(3, 3)
```

[SmileAnimation](#)



Mod this!

- Add a third image to the animation, perhaps a frown face.
- Make your own custom animation! What LEDs stay the same and which need to be toggled?

Random Patterns generator

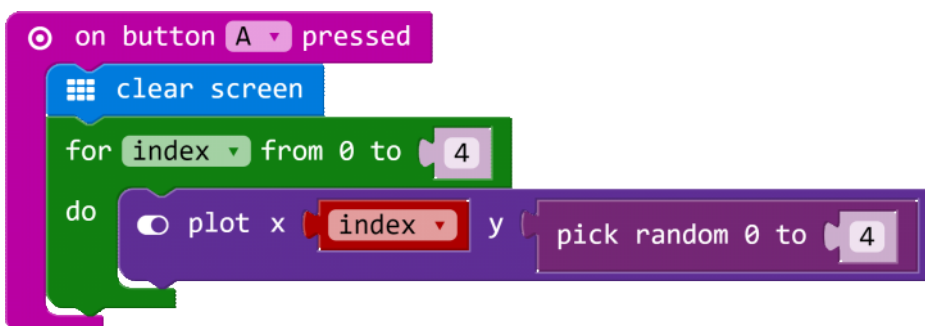
A short exercise using a loop to generate random LED patterns and then checking the status of a specific LED.

Pseudocode:

- On button A pressed we'll use a loop to turn on a random set of LED lights on our micro:bit.
- Our display will have one LED lit for each column or x coordinate value from 0 through 4.

Steps:

- From the Input Toolbox drawer, select the 'on button pressed' block
- From the Basic - More Toolbox drawer, drop in a 'clear screen' block
- From the Loops Toolbox drawer, drop in a 'for' block
- From the LED Toolbox drawer, drop a 'plot x y' block
- Use the variable 'index' for the x value
- From the Math Toolbox drawer, drop a 'pick random' block into the y value



```
input.onButtonPressed(Button.A, () => {  
  basic.clearScreen()  
})
```

```

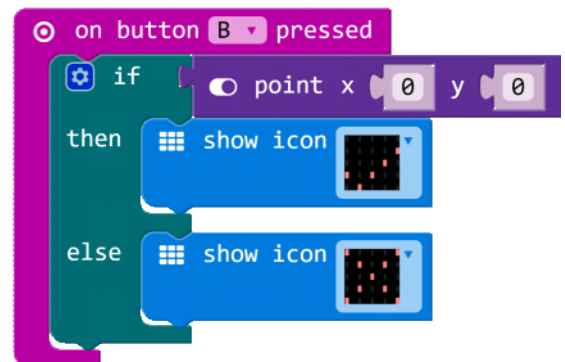
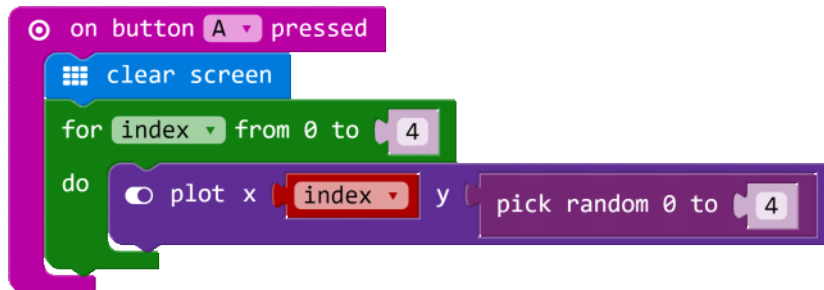
    for (let index = 0; index <= 4; index++) {
        led.plot(index, Math.random(5))
    }
})

```

Check the on/off state of an LED

- On button B pressed we'll use an 'if...then...else' block from the Logic Toolbox drawer
- From the LED Toolbox drawer, drop a 'point x y' block into the 'if' condition to check the current on/off state of a specific LED.
 - If the LED is currently on, the point x y block will return true.
 - If the LED is currently off, the point x y block will return false.
- For this exercise, we'll use the two Yes/No built in icons to display the LED's current status. From the Basic Toolbox drawer, drag 2 'show icon' blocks into each of the 'then' and 'else' clauses. Select the check mark for Yes, and the X icon for No.
- For now, we'll leave the default coordinate values (0,0). But you can challenge your students to add a loop to test for all coordinates on the micro:bit

Here is the complete program:



```

input.onButtonPressed(Button.A, () => {
    basic.clearScreen()
    for (let index = 0; index <= 4; index++) {
        led.plot(index, Math.random(5))
    }
})
input.onButtonPressed(Button.B, () => {
    if (led.point(0, 0)) {
        basic.showIcon(IconNames.Yes)
    } else {
        basic.showIcon(IconNames.No)
    }
})

```

[RandomPatterns](#)



Try it out!

- Download the program to your micro:bit
- Press button A to create a random pattern
- Press button B to check and display the status of the specific LED

Brightness

A short exercise in using the brightness settings for the micro:bit LEDs. Important to note - the brightness level of the micro:bit simulator LEDs will NOT appear to change! You must run your program on the actual micro:bit to see the different brightness levels.

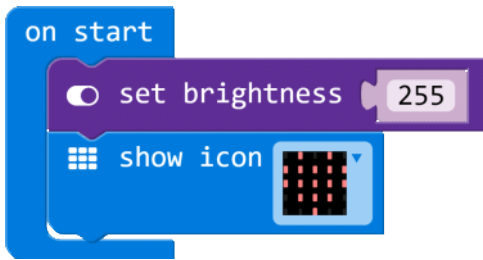
We will check on, and numerically display the brightness level with our program, so we can verify with the simulator that it is working.

Pseudocode:

We'll set the brightness level for the LEDs to the highest level on start and then use on button A pressed to decrease the brightness level and on button B pressed to increase the brightness level. We'll use on button A+B pressed to check and display numerically the current brightness level.

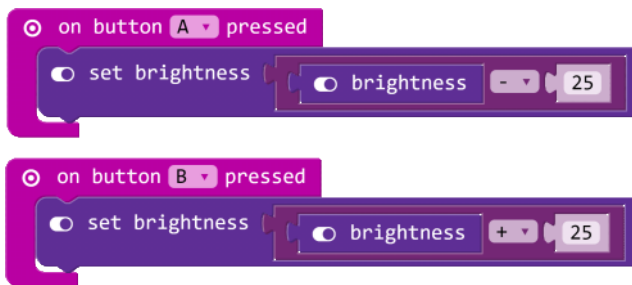
Steps:

- Drag 3 'set brightness' blocks and 3 'brightness' blocks from the Led - More Toolbox drawer onto your coding workspace
- Place one 'set brightness' block in the 'on start' block
- Add a 'show icon' block after the 'set brightness' block so we will have an image to look at



```
led.setBrightness(255)
basic.showIcon(IconNames.Heart)
```

- From the Input Toolbox drawer, drag out 3 'on button pressed' blocks onto your coding workspace
- Leave one 'on button pressed' block with the default setting of A and change the second one to B and the third one to A+B
- Place one 'set brightness' block in the 'on button A' pressed block, and the other 'set brightness' block in the 'on button B' pressed block
- From the Math Toolbox drawer, drag out an addition block and a subtraction block
- Place the addition block within the 'set brightness' block in the 'on button B pressed' block
- Place the subtraction block within the 'set brightness' block in the 'on button A pressed' block
- Place a 'brightness' block on the left side of each math expression
- Change the default value of 0 on the right side of each math expression to 25

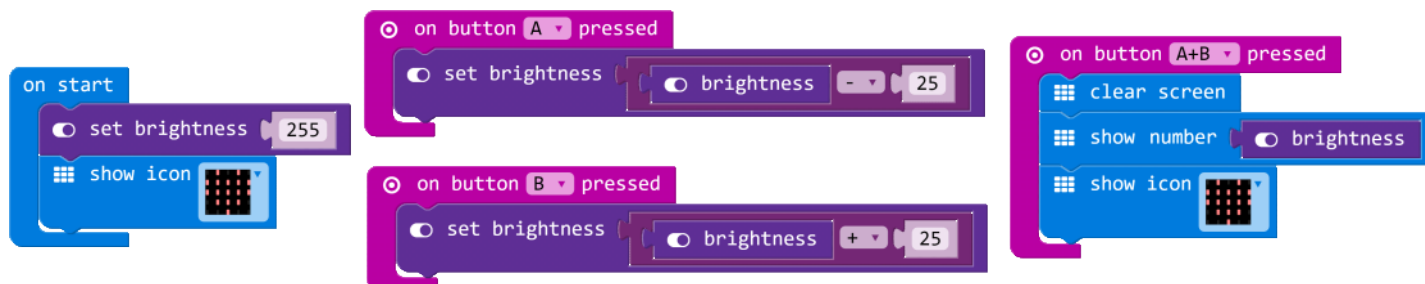


```
input.onButtonPressed(Button.A, () => {
  led.setBrightness(led.brightness() - 25)
})
input.onButtonPressed(Button.B, () => {
  led.setBrightness(led.brightness() + 25)
})
```

Since we cannot see if our program is working in the simulator, let's add a check into our code.

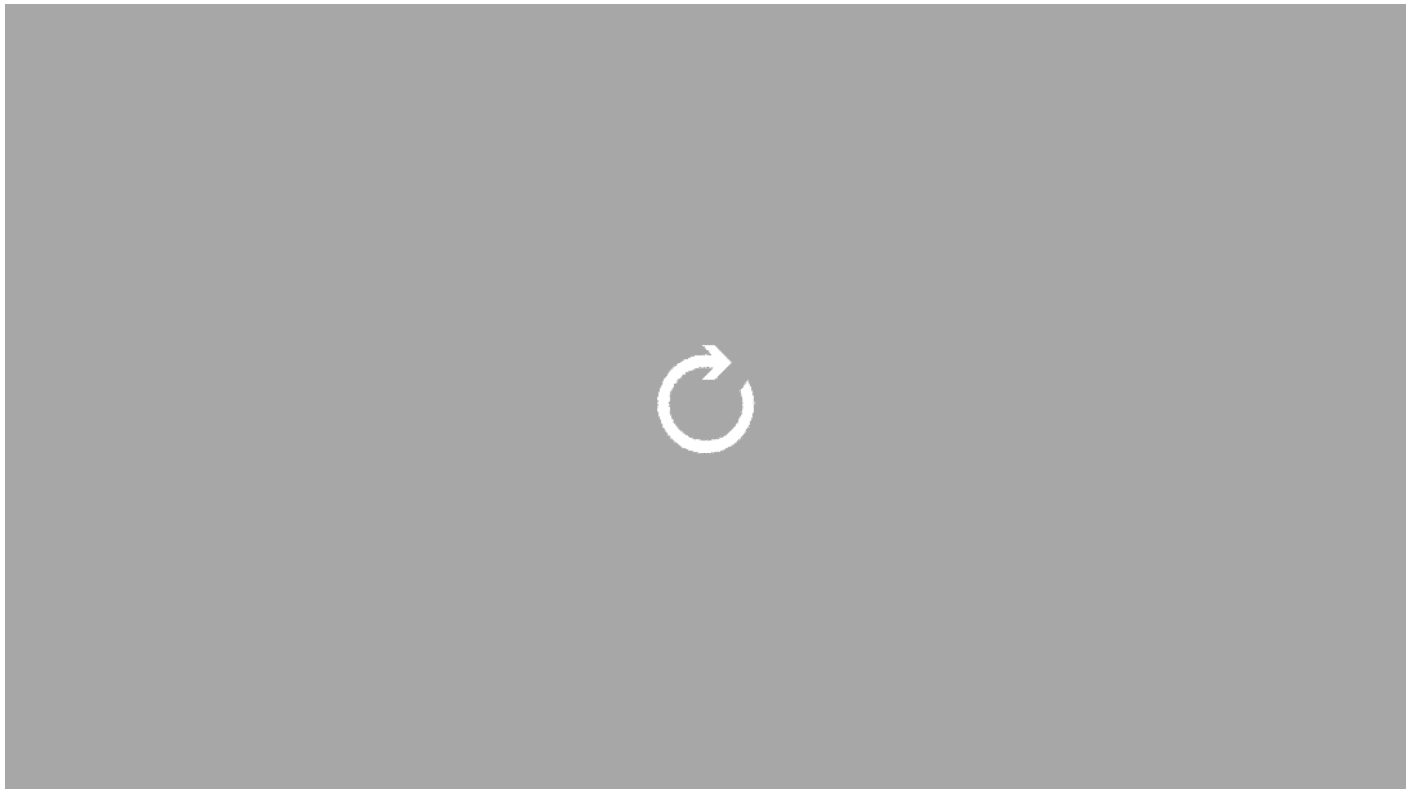
- On button A+B pressed, we'll clear the screen and get and display the current brightness level as a number.
- Then we'll re-display the image we used on start.

Here is the complete program:



```
input.onButtonPressed(Button.A, () => {
  led.setBrightness(led.brightness() - 25)
})
input.onButtonPressed(Button.B, () => {
  led.setBrightness(led.brightness() + 25)
})
input.onButtonPressed(Button.AB, () => {
  basic.clearScreen()
  basic.showNumber(led.brightness())
  basic.showIcon(IconNames.Heart)
})
led.setBrightness(255)
basic.showIcon(IconNames.Heart)
```

Brightness



Try it out!

- What happens if adding 25 or subtracting 25 from the current brightness level would result in a sum or difference outside of the 0 to 255 brightness range?

Project: Screensaver or Game

Use what you now know about LEDs, coordinates, and brightness to create your own project: a screensaver, or a game. You should find a way to use coordinates in your program. Even better, use variables to store and update your coordinates.

Screensavers

One type of project is a screensaver. A long time ago, computers and televisions used cathode ray tube (CRT) screens for displays. The glass screen of the display was coated on the back with phosphor, a substance that glows when painted with electrons from an electron gun at the other end of the tube. When the same area of the screen was painted (excited) over and over again by the stream of electrons, that part of the screen would sometimes "freeze" with the same image, burned into the phosphor for good. This was called "burn-in".

Normally, if a show was running, or if someone was actively using the computer, the display changed often enough that burn-in wasn't a problem. Programmers learned to create a demo screen with an animation that would run whenever the screen was idle. Today, nearly all computers and television sets use LCD displays, which are not affected by burn-in. But you can still find a screen saver in nearly every computer's Settings panel, as an opportunity to show off some neat graphics or animation.

Your task is to create:

- 1) A "screen saver" animation using the plot/unplot blocks. You can fill the screen line by line, pausing between each one, or fill it with a random constellation of stars.

OR

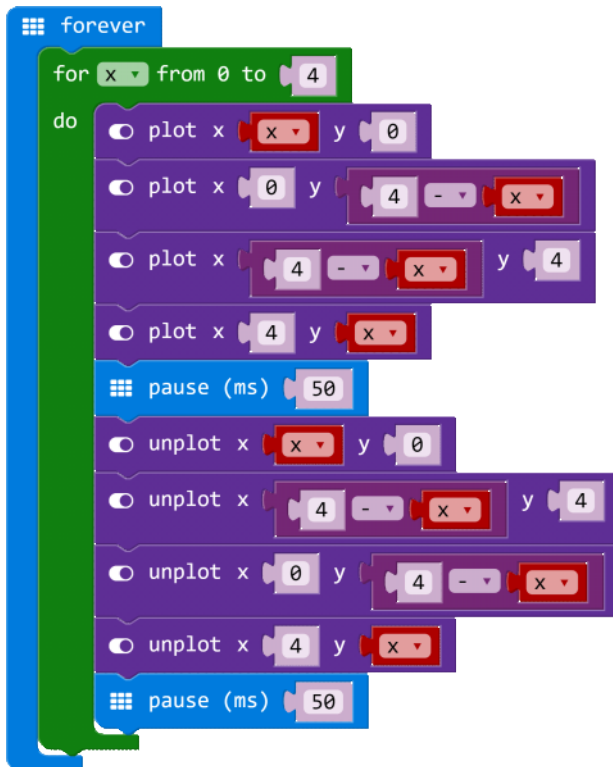
- 1) A game that uses sprites to manage the x and y coordinate values of the different objects.

Your project might use variables to store the values of sprites, which are special structures that contain an x and a y coordinate together that describe the sprite's location as one LED on the screen.

Project Ideas

Firework Screensaver

This project uses a for loop with the plot/unplot blocks to create a symmetrical design on the screen. This student used a subtraction operation to get a variable that decreases as the index variable in the loop increases.



```

basic.forever(() => {
  for (let x = 0; x <= 4; x++) {
    led.plot(x, 0)
    led.plot(0, 4 - x)
    led.plot(4 - x, 4)
    led.plot(4, x)
    basic.pause(50)
    led.unplot(x, 0)
    led.unplot(4 - x, 4)
    led.unplot(0, 4 - x)
    led.unplot(4, x)
    basic.pause(50)
  }
})

```

[Firework](#)

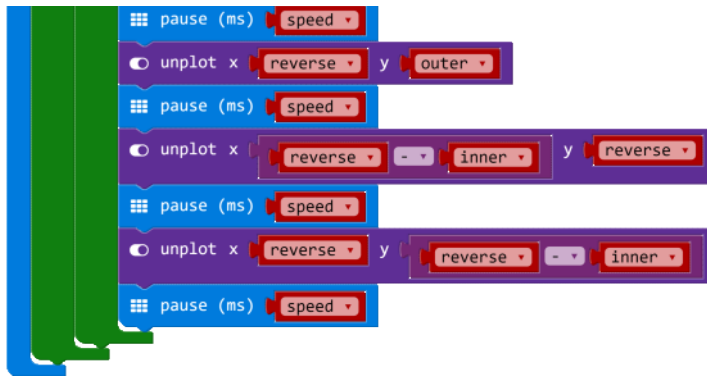


Cascade Screensaver

This example creates a diagonal cascading effect across the screen. Note the use of a variable (speed) to allow you to easily change the speed of the animation by changing just one number value.

```
on start
  set speed to 10

forever
  for outer from 0 to 4
  do
    set reverse to 4 - outer
    for inner from 0 to 4
    do
      plot x outer y reverse
      pause (ms) speed
      plot x reverse y outer
      pause (ms) speed
      plot x reverse - inner y reverse
      pause (ms) speed
      plot x reverse y reverse - inner
      pause (ms) speed
    do
      for outer from 0 to 4
      do
        set reverse to 4 - outer
        for inner from 0 to 4
        do
          unplot x outer y reverse
          pause (ms) speed
          unplot x reverse y outer
```

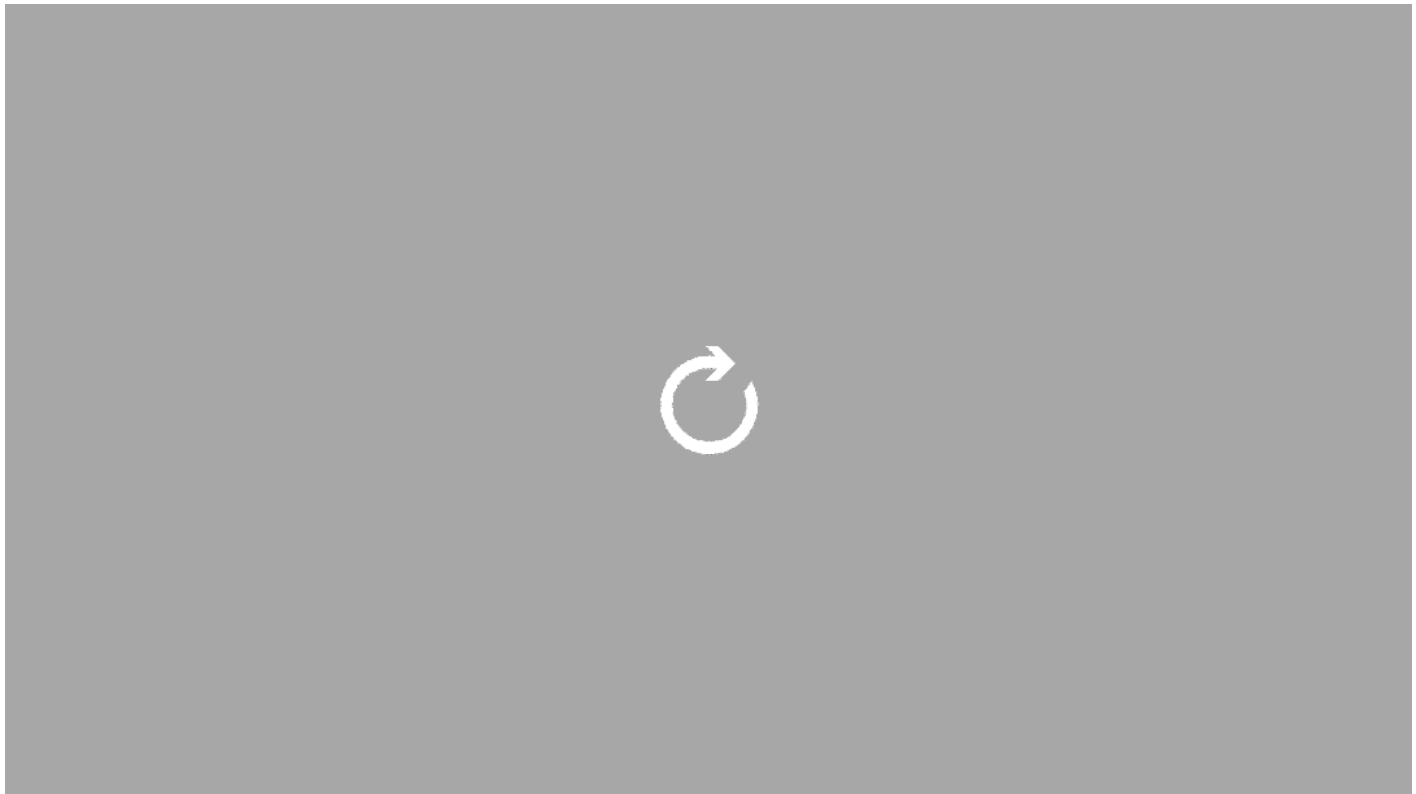



```

let reverse = 0
let speed = 0
let inner = 0
let outer = 0
basic.forever(() => {
  for (let outer = 0; outer <= 4; outer++) {
    reverse = 4 - outer
    for (let inner = 0; inner <= 4; inner++) {
      led.plot(outer, reverse)
      basic.pause(speed)
      led.plot(reverse, outer)
      basic.pause(speed)
      led.plot(reverse - inner, reverse)
      basic.pause(speed)
      led.plot(reverse, reverse - inner)
      basic.pause(speed)
    }
  }
  for (let outer = 0; outer <= 4; outer++) {
    reverse = 4 - outer
    for (let inner = 0; inner <= 4; inner++) {
      led.unplot(outer, reverse)
      basic.pause(speed)
      led.unplot(reverse, outer)
      basic.pause(speed)
      led.unplot(reverse - inner, reverse)
      basic.pause(speed)
      led.unplot(reverse, reverse - inner)
      basic.pause(speed)
    }
  }
})
speed = 10

```

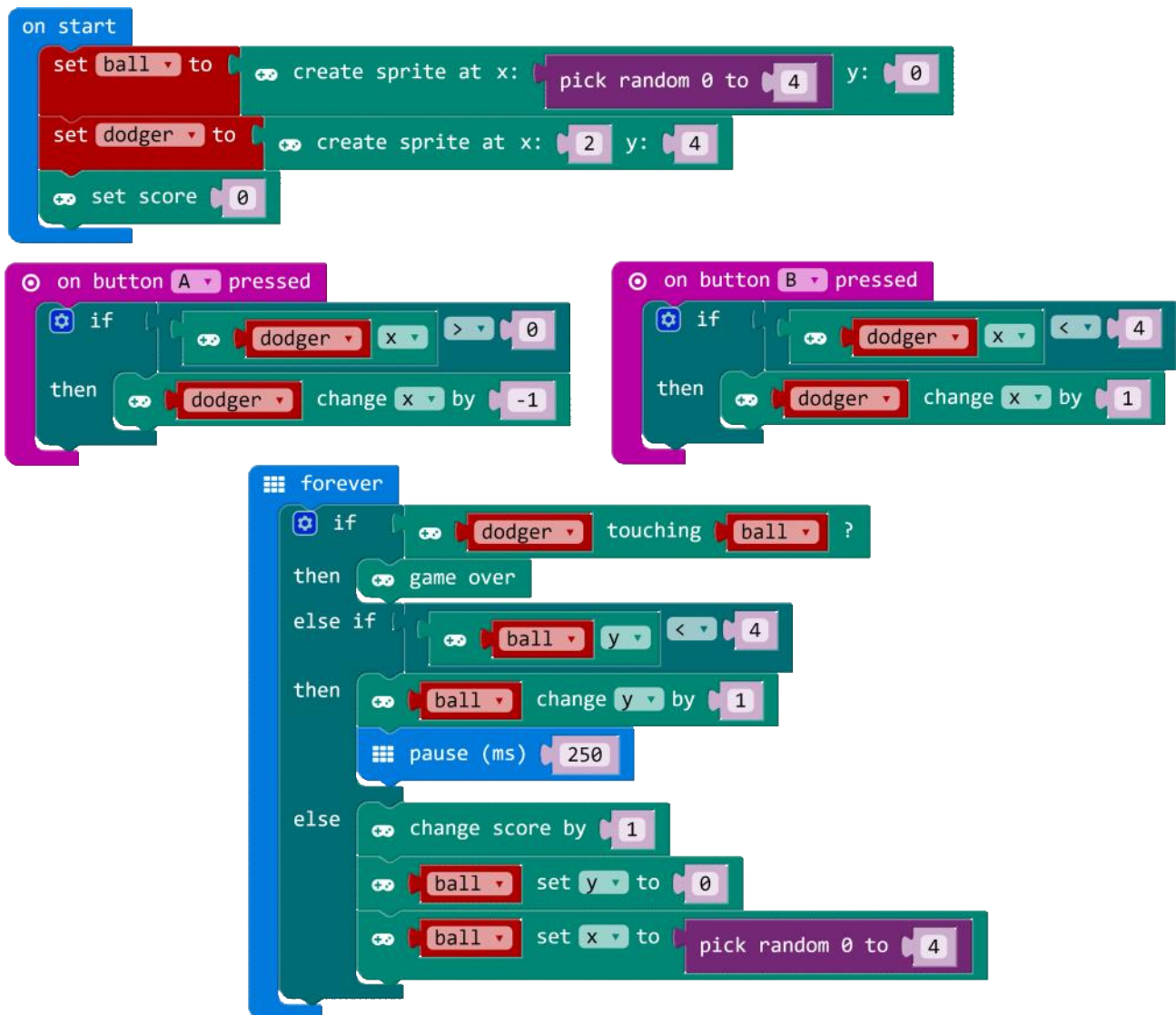
[Cascade](#)



Dodge Ball Game

This is a Dodge Ball game that uses one sprite (dodger) to try to avoid another sprite (ball). You use the A and B buttons to move the dodger to avoid the balls that are falling from the top of the screen.

Here is the complete Dodge Ball program.



```

let dodger: game.LedSprite = null
let ball: game.LedSprite = null
basic.forever(() => {
  if (dodger.isTouching(ball)) {
    game.gameOver()
  } else if (ball.get(LedSpriteProperty.Y) < 4) {
    ball.change(LedSpriteProperty.Y, 1)
    basic.pause(250)
  } else {
    game.addScore(1)
    ball.set(LedSpriteProperty.Y, 0)
    ball.set(LedSpriteProperty.X, Math.random(5))
  }
})
input.onButtonPressed(Button.A, () => {
  if (dodger.get(LedSpriteProperty.X) > 0) {
    dodger.change(LedSpriteProperty.X, -1)
  }
})
input.onButtonPressed(Button.B, () => {
  if (dodger.get(LedSpriteProperty.X) < 4) {

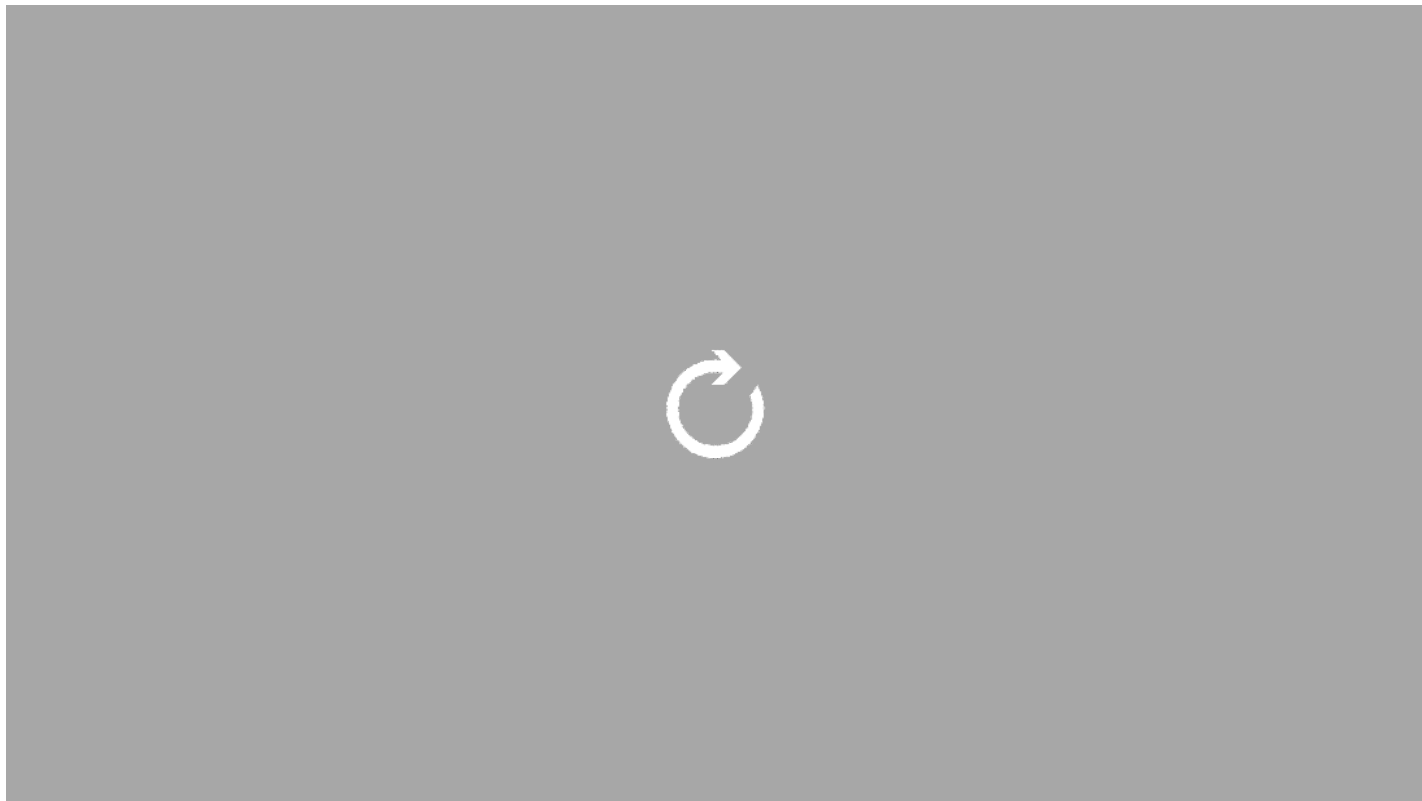
```

```

        dodger.change(LedSpriteProperty.X, 1)
    }
})
ball = game.createSprite(Math.random(5), 0)
dodger = game.createSprite(2, 4)
game.setScore(0)

```

DodgeBall



Reflection

Have students write a reflection of about 150–300 words, addressing the following points:

- Did you do a screensaver? A game? Something different? How did you decide?
- If you did a game, what is the object of the game?
- How does your project use coordinates?
- Describe something in your project that you are proud of.
- Describe a difficult point in the process of designing this program, and explain how you resolved it.
- What feedback did your beta testers give you? How did that help you improve your design?

Assessment

	4	3	2	1
Coordinates and LEDs	Uses at least 3 of the different kinds of plot/unplot/toggle/point x y blocks in a meaningful way. Uses variables to update	At least 2 of the different kinds of plot/unplot/toggle/point x y blocks in a meaningful way	At least 1 of the different kinds of plot/unplot/toggle/point x y blocks in a meaningful way	No plot/unplot/toggle/point x y blocks are implemented.

	coordinates.			
Micro:bit program	<p>Micro:bit program:</p> <ul style="list-style-type: none"> • Uses plotted LEDs in a way that is integral to the program, • Compiles and runs as intended, • Meaningful comments in code 	Micro:bit program lacks 1 of the required elements	Micro:bit program lacks 2 of the required elements	Micro:bit program lacks all of the required elements
Collaboration reflection	<p>Reflection piece includes:</p> <ul style="list-style-type: none"> • Brainstorming ideas • Construction • Programming • Beta testing 	Reflection piece lacks 1 of the required elements.	Reflection piece lacks 2 of the required elements.	Reflection piece lacks 3 of the required elements.

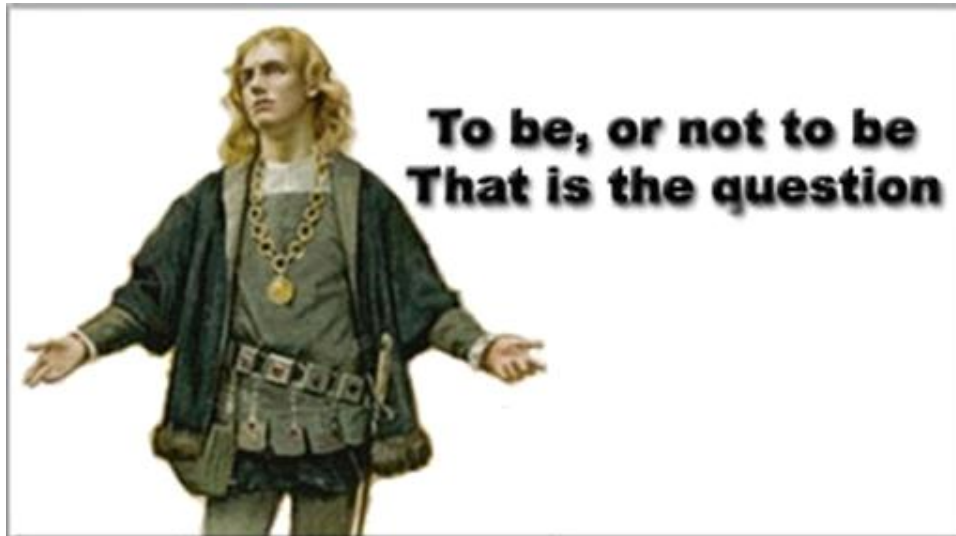
Standards

CSTA K-12 Computer Science Standards

- CL.L2-03 Collaborate with peers, experts, and others using collaborative practices such as pair programming, working in project teams, and participating in group active learning activities
- CT.L1:6-01 Understand and use the basic steps in algorithmic problem-solving
- CT.L1:6-02 Develop a simple understanding of an algorithm using computer-free exercises
- CPP.L1:6-05 Construct a program as a set of step-by-step instructions to be acted out
- 2-A-5-7 Create variables that represent different types of data and manipulate their values.
- CT.L2-14 Examine connections between elements of mathematics and computer science including binary numbers, logic, sets and functions.

Booleans

This lesson introduces the use of the boolean data type to control the flow of a program, keep track of state, and to include or exclude certain conditions.



Shakespeare knew Booleans (quote from Hamlet)

Lesson Objectives

Students will...

- Understand what booleans and boolean operators are, and why and when to use them in a program.
- Learn how to create a boolean, set the boolean to an initial value, and change the value of the boolean within a micro:bit program.
- Learn how to use the random true or false block.
- Apply the above knowledge and skills to create a unique program that uses booleans and boolean operators as an integral part of the program.

Lesson Plan Structure

- Introduction: Booleans in daily life
- Unplugged Activity: Two Heads are Better Than One
- Micro:bit Activity: Double Coin Flipper
- Project: Boolean
- Assessment: Rubric
- Standards: Listed

There are several different data types used in computer programming. We have already used two of these types:

- String (for text)
- Integer (for numbers)

Boolean is another type of data. A boolean data type has only two values: true or false. In true binary fashion, these two values can be represented by the numbers 1 = true, and 0 = false.

Booleans are useful in programming for decision-making, often deciding when certain functions and parts of programs should start or stop running and are also used in database searches.

Ask the students to think of things in daily life that have only two values or states. The status is always one value or the other value.

Examples of Booleans in daily life

- Lights: On or Off
- Time: AM or PM
- You!: Asleep or Awake
- Weather: Raining or Not Raining
- Math: Equal to or Not Equal to
- Game: Truth or Dare
- Soda: Coke or Pepsi
- At the store: Paper or Plastic? Cash or Credit? Chip or Swipe?

Note:

Arguments can be made that some of these can have more than two values.

For example: At the store, you may have brought your own reusable bags or pay by check.

Let the students discuss these to help them hone in on which examples best represent Booleans.

A student might argue that a dimmer switch on a light or the brightness value on the micro:bit LEDs allow the lights to be in a state between on and off. One could respond that you can classify 'on' as the state where any electricity at all is running through the bulb (on) versus no electricity at all (off).

In programming, if you have worked with conditionals or loops, you have already worked with this type of logic:

- If a certain condition is true, do this, otherwise (if condition is false), do something else.
- While a certain condition is true, do this

Boolean Operators: AND, OR, and NOT

To make working with Booleans useful for solving more complex decisions and searches, we can connect two or more Booleans into one decision statement. To do this, we use what are known as Boolean operators. The three most common and the ones we will use with the micro:bit are And, Or, and Not.

These operators can be used in conditionals and loops, like so:

- If condition A is true AND condition B is true
- If condition A is true OR condition B is true
- While event A has NOT happened

Let's look at how each of these work.

AND

(Condition A AND Condition B)

For this expression to evaluate as true, both conditions in the expression need to be true.

So, if both Condition A AND Condition B are true, the expression will evaluate as or return true.

OR

(Condition A OR Condition B)

For this expression to evaluate as true, only one of the conditions in the expression needs to be true.

If Condition A is true, the expression will return true regardless of whether Condition B is true or false.

If Condition B is true, the expression will return true regardless of whether Condition A is true or false.

NOT

NOT can be used when checking that a condition is false (or not true).

For example:

- (NOT Condition A and Condition B) evaluates as true only if Condition A is false and Condition B is true.
- (Condition A and NOT Condition B) evaluates as true only if Condition A is true and Condition B is false.
- (NOT Condition A and NOT Condition B) evaluates as true only if both Condition A and Condition B are true.

NOT is also useful when using a loop. For example, you can use a NOT to check

While button A is NOT pressed, continue to run this code...

Note: 'False' can be thought of as equivalent to 'NOT true'.

Sidebar material



Image credit: Wikimedia Commons

George Boole (/ˈbuːl/; 2 November 1815 – 8 December 1864) was an English mathematician, educator, philosopher and **logician**. He worked in the fields of differential equations and algebraic **logic**, and is best known as the author of *The Laws of Thought* (1854) which contains **Boolean** algebra.

Unplugged: Two Heads are Better Than One

Materials: A penny for each student, paper and pencils

Most students have used a penny to decide something. Ask for some examples.
Who goes first in a game, to break a tie, to decide which activity to do...

A simple penny is the most common binary decision making tool ever!
When you flip a coin to decide something there are only two possible outcomes, heads or tails.
When you flip a coin the outcome is random.

What's a common issue with coin tosses? *Students may bring up issues of trust and fairness. Who gets to flip the coin? Who gets to 'call' it? What if it's a 'faulty' coin?*

Here's a solution... The double coin toss.



In a double coin toss, both people have a coin and they flip the coins at the same time.

Working in pairs, have the students make a table or list of the possible outcomes if each student flipped a coin at the same time.

Example:

Coin A	Coin B
Heads	Heads
Heads	Tails
Tails	Heads
Tails	Tails

There are 4 possible outcomes.

- For 2 outcomes, the result is the same for both coins, both heads or both tails.
- For the other 2 outcomes, the result for each coin is different, heads/tails and tails/heads.

So, if 2 coins are flipped, the chance that the outcomes will be the same (HH/TT) is equal to the chance that the outcomes will be different (HT/TH). Both outcomes, coins the same/coins are different have a 2 in 4 or 50% chance of occurring.

Therefore, if Person A wins each time the outcomes are the same and Person B wins each time

the outcomes are different, both have an equal chance of winning each double coin flip. With this system, no one person's outcome, heads or tails, guarantees a win. If Person A's coin flips to heads, she would win if Person B also flipped heads, but lose if Person B flipped tails. Students will usually see that this is a fair system.

Let the students experiment with this.

Have students flip their coins together, keeping track of the outcomes, perhaps by adding another column to their table.

Example:

Coin A	Coin B	Totals
Heads	Heads	
Heads	Tails	
Tails	Heads	
Tails	Tails	

Just for fun, have them play to a certain total number of rounds.

So, what does this have to do Boolean variables and operators?

Think about how you would code a program a double coin flipper.

How would you represent each of the 4 different possible double coin flip outcomes?

Let's pseudocode it!

We can create a Boolean variable to represent whether an outcome is heads or tails.

We can make

- Heads = True
- Tails = False

Note: Tails = False can also be thought of as Tails = not true.

Have the students copy their Heads/Tails table of possible outcomes, but label the columns "Coin A Heads" and "Coin B Heads" and replace each entry of 'Heads' with 'True' and 'Tails' with 'False'. In the study of logic, this is known as a truth table.

We'll use it to help us pseudocode our program, by adding a third column describing the results of each outcome.

Coin A Heads	Coin B Heads	Results
True	True	If Coin A is true AND Coin B is true, add one to Player A score
True	False	If Coin A is true AND Coin B is false, add one to Player B score
False	True	If Coin A is false AND Coin B is true, add one to Player B score
False	False	If Coin A is false AND Coin B is false, add one to Player A score

Can we make this code more efficient? Can we combine any of these lines?

Try using an OR to combine both conditions in which Player A scores a point.
Do the same for both conditions in which Player B scores a point.

Give the students a chance to work this out on their own.

Combining the conditions in which each player wins, gives us:

- If (Coin A is true AND Coin B is true) OR (Coin A is false AND Coin B is false), add one to Player A score.
- If (Coin A is true AND Coin B is false) OR (Coin A is false AND Coin B is true), add one to Player B score.

Note: Just as you do for math expressions with multiple operators, use parentheses to make it clear how the conditions and statements are grouped together.

The students are by now familiar with the MakeCode blocks. As they think through their algorithms, they may even have started to visualize the blocks they might use. Visualizing the blocks as they pseudocode can help them with the logical steps of their program. It can also help them to visualize and recognize the big picture of their code as well as the details.

Using blocks to start coding these two conditionals as currently written, might look like this:



Then add one to Player A score.

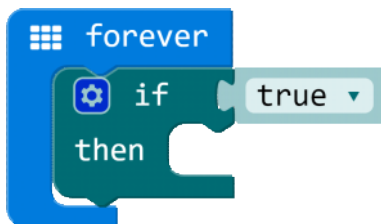


Then add one to Player B score.

Though this code will work as we want it to, it's a lot of code. It is good practice to keep your code as simply as possible. Let's see how we can do just that.

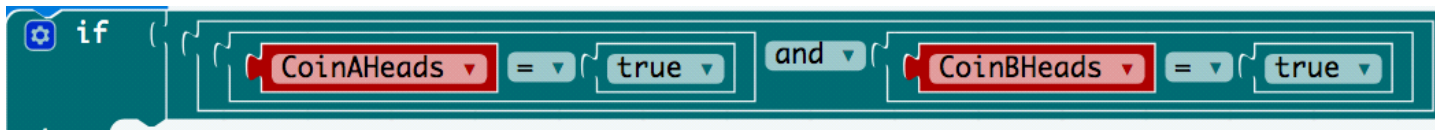
Booleans and Simplifying Code

A boolean can have only one of two values: True or False. Conditionals like 'if...then' check whether a condition is true. Notice that the default condition for the 'if...then' blocks is true. In other words, the 'if...then' blocks will check to see whether whatever condition you place there is true.



This means that 'If CoinAHeads' is the same as 'If CoinAHeads = true'. We can use this to simplify our code.

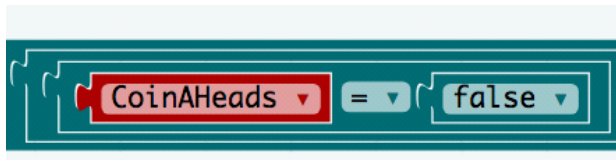
Instead of:



we can code:



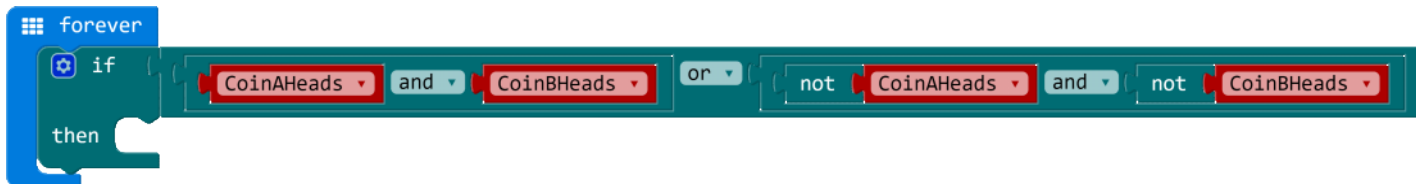
Also, since 'If not CoinAHeads' is the same as 'If CoinHeads = False', instead of



we can code



So now we have



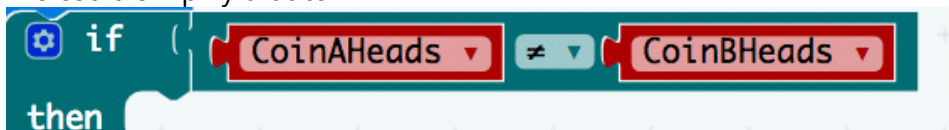
Can we simplify it even more?

For this particular program, since we are checking to see if the conditions CoinAHeads and CoinBHeads are the same, whether both true or both false, we can use a logic equals block to simplify our code to



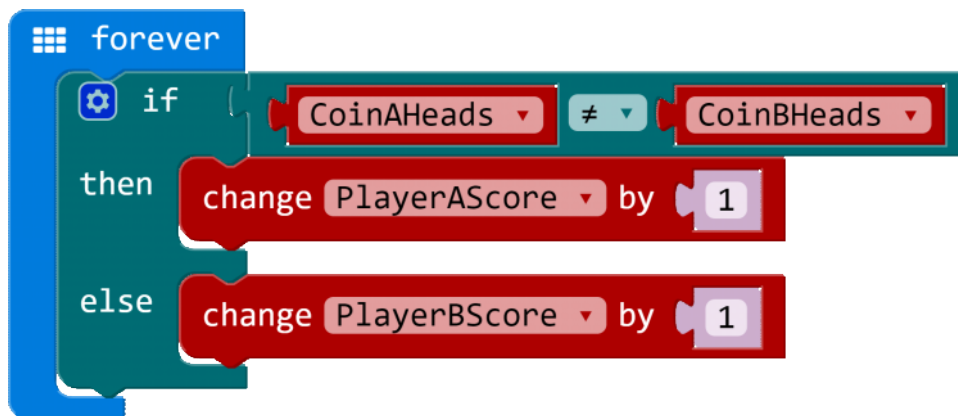
Then add one to Player A score.

What about our other big block of code for the conditions for a Player B win? We could simplify that to



Then add one to Player B score.

We don't need to do this! Since the only other option to being equal is to be not equal, we can simply do this:



Random Functions

We use a coin flip to decide things because the result is random, meaning the result happens without any conscious decision or direction. We use dice and game spinners for the same reason. The results are not predetermined or directed in any way.

So, how do we get a random flip in code? Most computer programming languages have a built in function that will select a random number given a range of values. Microsoft MakeCode has a block for this. And it also has a block for getting a random true or false value.

We will call on this built in function to get a random true or false value for each flip of a coin in the next Activity.

Our basic pseudocode for our Double Coin Flipper could look like this:

1. Use the random function to get a true/false value for Coin A.
2. Use the random function to get a true/false value for Coin B.
3. Compare the current values of Coin A and Coin B.
4. If the current true/false values of Coin A and Coin B are the same, add a point to Player A's score.
5. Otherwise, the current true/false values of Coin A and Coin B must be different, so add a point to Player B's score.
6. When players are done with their double coin flipping, show the final scores for each player.

Activity: Double Coin Flipper

Guide the students to create a program using Boolean variables and operators. We'll use our pseudocode from the previous activity to code a double coin flipper program.

For the first step, let's create our variables.

Make a variable for each of the following:

- CoinAHeads
- CoinBHeads
- PlayerAScore
- PlayerBScore

Now we need to initialize the variable values.

Put a 'set' variable block for each of these 4 variables inside the 'on start' block.

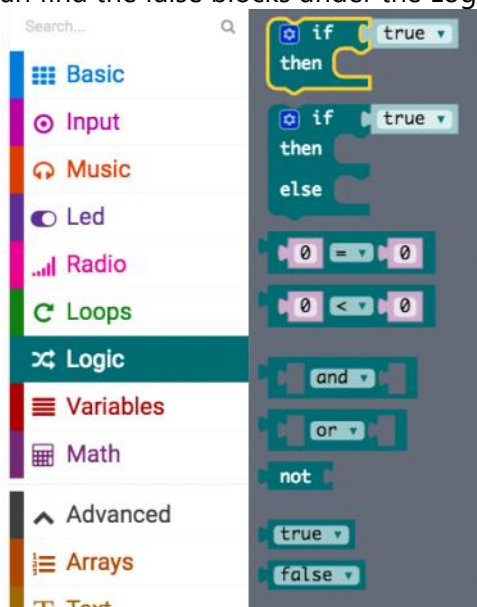
The initial value of a variable is the value the variable will hold each time the program starts.

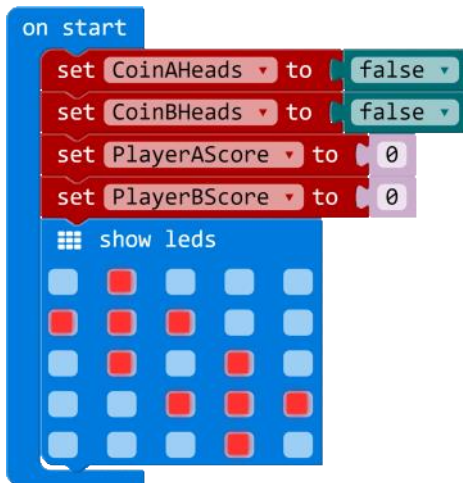
By default:

- a string variable is initialized to an empty string ""
- a number variable is initialized to 0
- a Boolean is initialized to 'false'

Initialize the number variables to zero and the Boolean variables to 'false'.

You can find the false blocks under the Logic menu.





```

let CoinAHeads = false
let CoinBHeads = false
let PlayerAScore = 0
let PlayerBScore = 0
basic.showLeds(`
. # . . .
# # # . .
. # . # .
. . # # #
. . . # .
`)

```

Notice that we also added an image for the start screen, so the user knows the program has started and is ready. Does the image look like two coins?

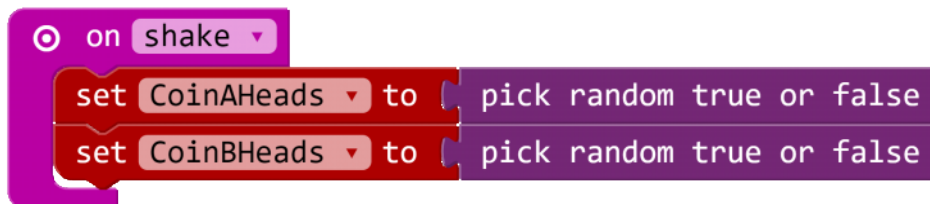
Random coin flips

When the player shakes the micro:bit, we will code the micro:bit to give each of our Boolean variables a random true/false value.

- From the Input Toolbox drawer, drag an 'on shake' block to the coding workspace
- From the Variables Toolbox drawer, drag 2 'set' variable blocks to the coding workspace
- Drag the 2 'set' blocks into the 'on shake' block
- Change the default 'item' to CoinAHeads and CoinBHeads
- From the Math Toolbox drawer, drag 2 'pick random true or false' blocks to the coding workspace
- Hover over this 'pick random' block and note that its pop-up description mentions coin flipping!



- Attach these 'pick random' blocks to the 'set' variable blocks in the 'on shake' block



```
let CoinBHeads = false
let CoinAHeads = false
input.onGesture(Gesture.Shake, () => {
  CoinAHeads = Math.randomBoolean()
  CoinBHeads = Math.randomBoolean()
})
```

Now that the virtual CoinA and CoinB have been virtually flipped, we need to compare the outcomes to see if they are the same or different.

- From the Logic Toolbox drawer, drag an 'if...then...else' block to the coding workspace
- Drag the 'if...then...else' block into the 'on shake' block under the 'set' variable blocks



```
input.onGesture(Gesture.Shake, () => {
  CoinAHeads = Math.randomBoolean()
  CoinBHeads = Math.randomBoolean()
})
```

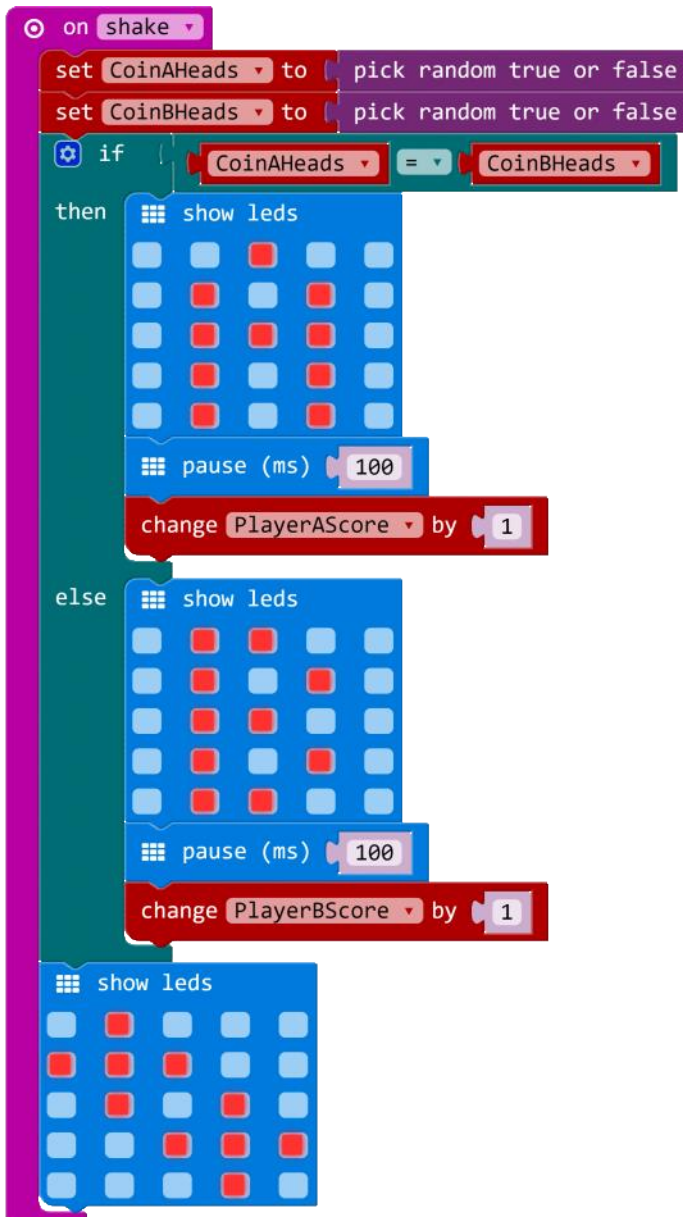
```
if (true) {  
  } else {  
  }  
})
```

Now our logic block is ready for the next steps of our pseudocode.

1. Compare the current values of Coin A and Coin B.
2. If the current true/false values of Coin A and Coin B are the same, add a point to Player A's score.
3. Otherwise, if the current true/false values of Coin A and Coin B are different, add a point to Player B's score.

Because we were able to visualize our blocks as we wrote our pseudocode, we already know what blocks we will use and also know that we have simplified our code as much as possible!

- We can now simply add this to our current code
- And provide user feedback by adding some visuals



To finish our program, we'll display the players' current scores on button A pressed.

Here is the complete program for our Double Coin Flipper.

```

let PlayerBScore = 0
let PlayerAScore = 0
let CoinBHeads = false
let CoinAHeads = false
input.onGesture(Gesture.Shake, () => {
  CoinAHeads = Math.randomBoolean()
  CoinBHeads = Math.randomBoolean()
  if (CoinAHeads == CoinBHeads) {
    basic.showLeds(`
      . . # . .
      . # . # .
    `)
  }
})

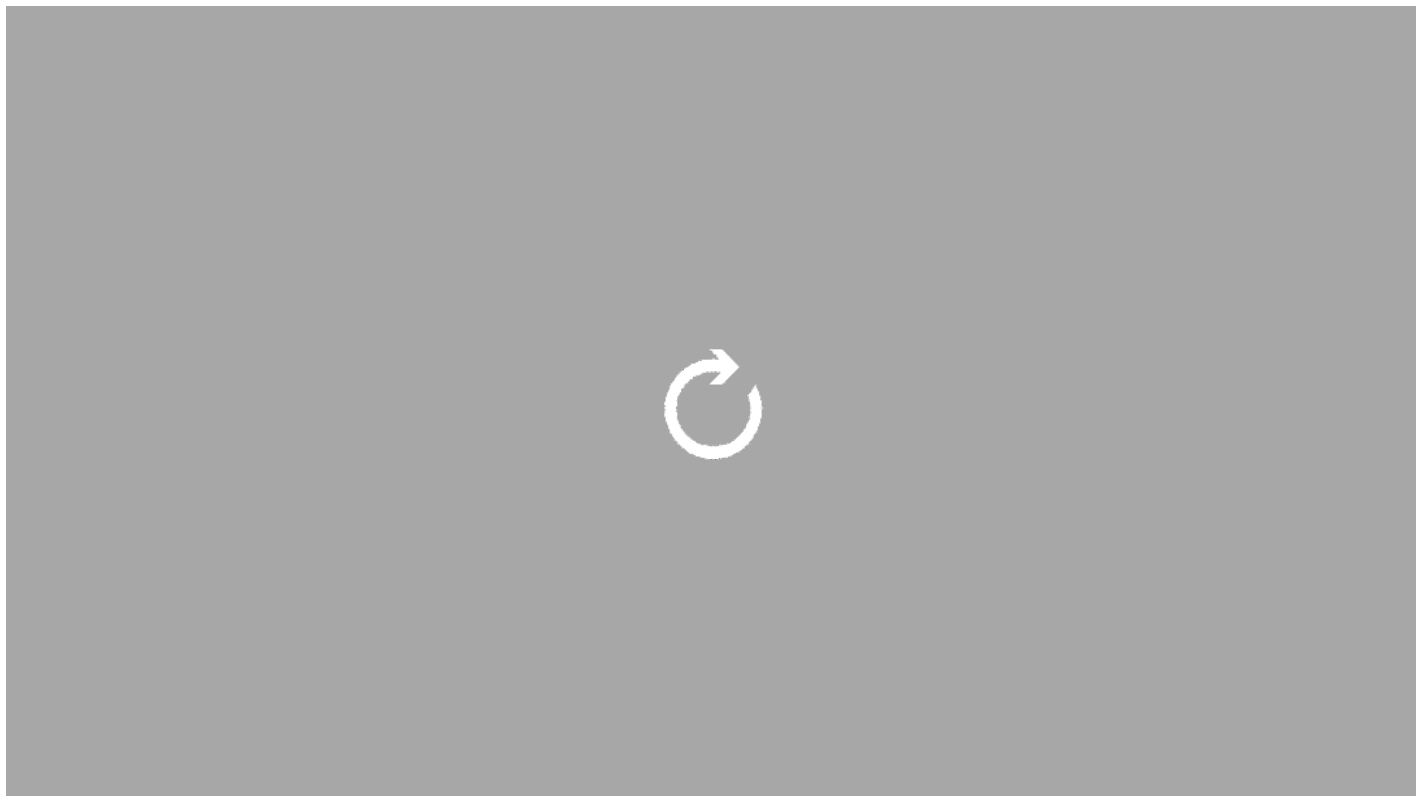
```

```

        . # # # .
        . # . # .
        . # . # .
        `)
        basic.pause(100)
        PlayerAScore += 1
    } else {
        basic.showLeds(`
            . # # . .
            . # . # .
            . # # . .
            . # . # .
            . # # . .
            `)
        basic.pause(100)
        PlayerBScore += 1
    }
    basic.showLeds(`
        . # . . .
        # # # . .
        . # . # .
        . . # # #
        . . . # .
        `)
})
input.onButtonPressed(Button.A, () => {
})
CoinAHeads = false
CoinBHeads = false
PlayerAScore = 0
PlayerBScore = 0
basic.showLeds(`
    . # . . .
    # # # . .
    . # . # .
    . . # # #
    . . . # .
    `)

```

DoubleCoinFlipper



Try it out!

Have the students play a few more rounds of the Double Coin Flip using their new Micro:bit Double Coin Flipper!

Boolean operator NOT in a Loop

```
on shake
  while not button A is pressed
    do
      repeat 2 times
        do
          play tone Middle C for 1/2 beat
          play tone High C for 1/2 beat
```

Do you remember this code from our micro:bit Alarm?

Can you read this code and tell what it does?

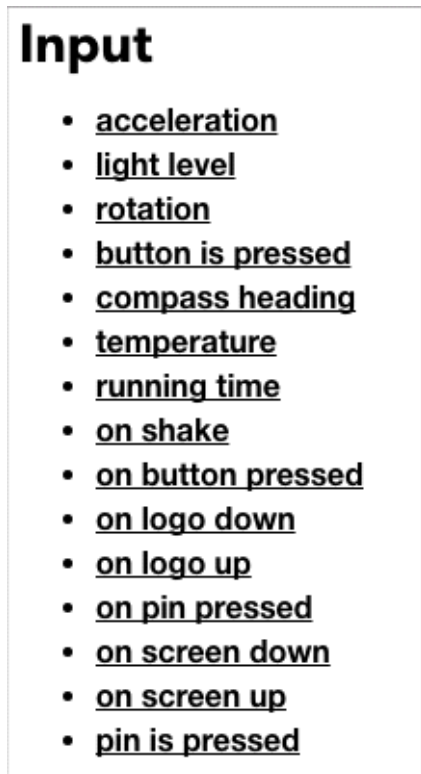
If the micro:bit is shaken, the micro:bit will play two tones twice and keep repeating this action until button A is pressed. So, after shaking, as long as 'is button A pressed?' is false, the two tone alarm will continue to repeat.

Project: Boolean

This is an assignment for students to come up with a micro:bit program that uses Boolean variables, Boolean operators, and possibly the random function.

Input

Remind the students of all the different inputs available to them through the micro:bit.



Project Ideas

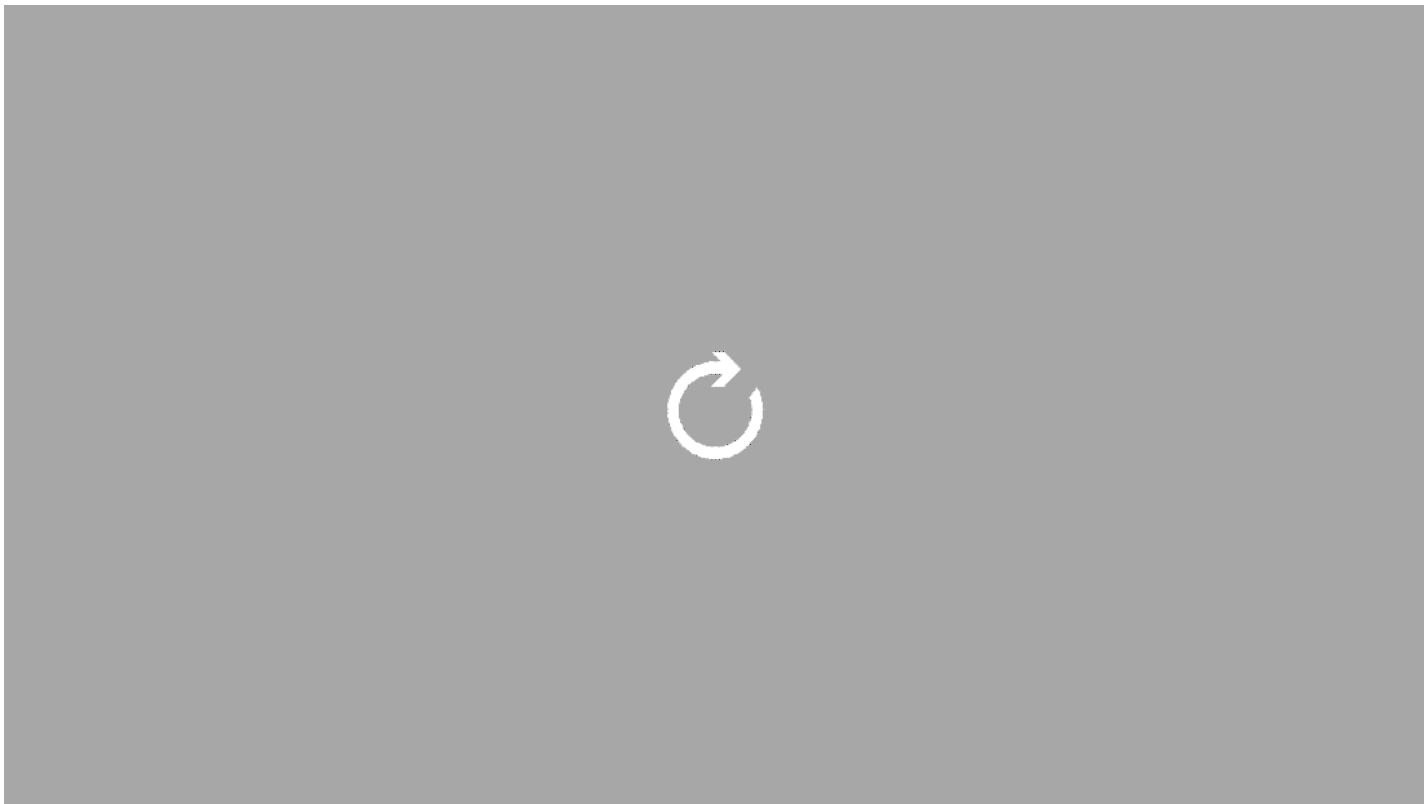
Sunscreen Monitor

When you shake the micro:bit, it reports the current temperature in degrees Fahrenheit. Button B measures the light level and if it is above 70 degrees AND very bright, it will display a sun icon. If it is above 70 degrees and less bright, it will display a cloudy symbol. If it is dark, it will display a nighttime icon.

[micro:bit Sunscreen Monitor](#)



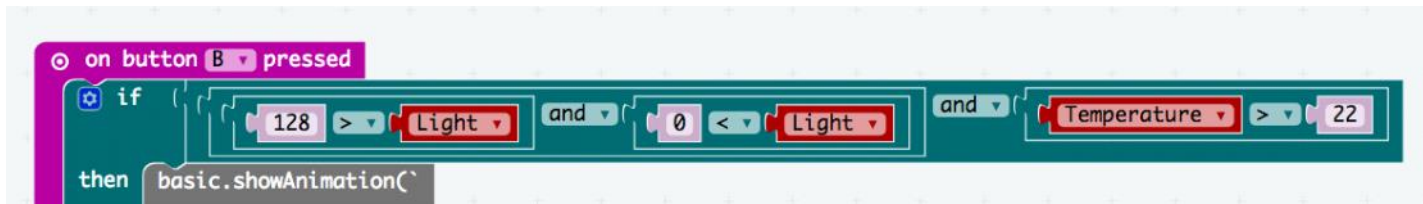
[Sunscreen](#)



Button A displays an animation to tell you whether or not you should use sunscreen (on sunny or cloudy days but not at night or indoors.)

Make a holder that can hold the micro:bit and a bottle of sunscreen.

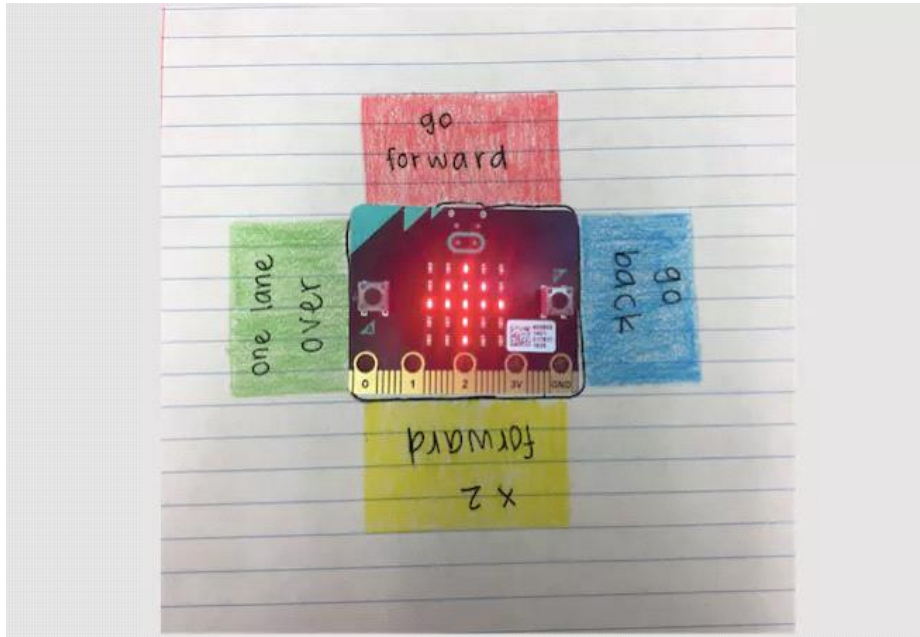
This example uses boolean operations because both light level AND temperature must be high in order to trigger the sun icon:



Two-Player Game

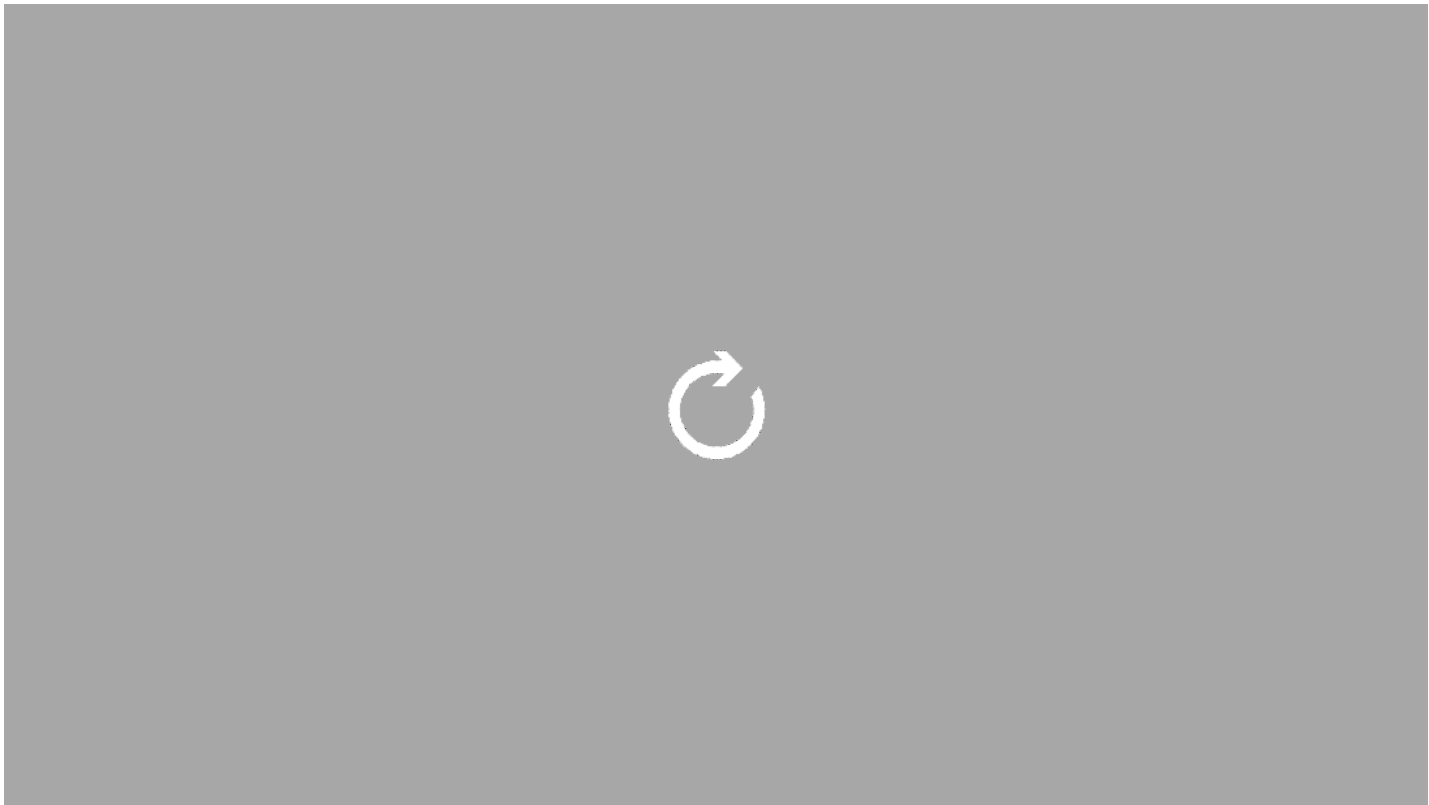
Create a game in which two players take turns on the same micro:bit. You can use a boolean variable called PlayerATurn to keep track of whose turn it is.

Board Game: Use boolean variables and random values as part of a board game (or improve your Board Game from the Variables lesson). Make the board and pieces and a holder for the micro:bit. Try modding a current board game.



Board Game with Arrows

[Board Game Arrow](#)



This is an example of a board game in which the micro:bit displays an arrow pointing in a random direction. The paper legend indicates different actions the player must take.

```

on Shake
  if (player1Turn == true) and (pick random 0 to 3 < 3)
  then
    clear screen
    set delay to 0
    while (delay < 500)
    do
      show leds
  
```

Here is a portion of the board game's code. A boolean variable is used to determine whose turn it is. If player1Turn is false, then it's player 2's turn. A random number is generated to show the arrow seventy-five percent of the time (for values of 0, 1, or 2).

Assessment

	4	3	2	1
Boolean	More than 2 Boolean variables are implemented in a meaningful way	At least 2 Boolean variable is implemented in a meaningful way	At least 1 Boolean variable is implemented in a meaningful way	No Boolean variables are implemented.
Micro:bit program	Micro:bit program: <ul style="list-style-type: none"> • Uses Boolean variables in a way that is integral to the program • Uses a random function in a way that is integral 	Micro:bit program lacks 1 of the required elements	Micro:bit program lacks 2 of the required elements	Micro:bit program lacks 3 or more of the required elements

	<p>to the program</p> <ul style="list-style-type: none"> • Compiles and runs as intended • Meaningful comments in code 			
Collaboration reflection	<p>Reflection piece includes:</p> <ul style="list-style-type: none"> • Brainstorming ideas • Construction • Programming • Beta testing 	Reflection piece lacks 1 of the required elements.	Reflection piece lacks 2 of the required elements.	Reflection piece lacks 3 of the required elements.

Standards

CSTA K-12 Computer Science Standards

- CL.L2-03 Collaborate with peers, experts, and others using collaborative practices such as pair programming, working in project teams, and participating in group active learning activities
- CT.L1:6-01 Understand and use the basic steps in algorithmic problem-solving
- CT.L1:6-02 Develop a simple understanding of an algorithm using computer-free exercises
- CPP.L1:6-05 Construct a program as a set of step-by-step instructions to be acted out
- 2-A-5-7 Create variables that represent different types of data and manipulate their values.

Binary

This lesson presents the concept of binary digits and base-2 notation. Students will learn how data is stored digitally and how it can be read and accessed.

Lesson Objectives

Students will...

- Understand what a bit and byte are and how they relate to computers and the way information is processed and stored.
- Learn to count in Base-2 (binary) and translate numbers from Base-10 (decimal) to binary and decimal.
- Apply the above knowledge and skills to create a unique program that uses binary counting as an integral part of the program.

Lesson Plan Structure

- Introduction: Bits and Bytes
- Unplugged Activity: Binary Vending Machine
- Micro:bit Activity: Binary Transmogriifier
- Project: Make a Binary Cash Register
- Assessment: Rubric
- Standards: Listed

Introduction

Most everyone who uses a computer has heard the terms, kilobyte (kB), Megabyte (MB), Gigabyte (GB) and even Terabyte (TB), usually when referring to the size of computer files and hard drives as well as download speeds. Bandwidth or connection rates are measured in bits/second. But what is a bit and what is a byte and what do they have to do with computers?

Picture a basic room light. The light is either on or it is off. You control the current state of the light by flipping a switch that has only two settings, down (light off) and up (light on). The earliest computers used a series of mechanical switches to control the flow of electricity through their circuits, turning each one on or off. The on/off states of the circuits was used to represent and even store information. The smallest unit of information, representing the state of one switch, is known as a bit.

A bit is a binary digit and has only two possible values, zero or one. The value of the bit represents the current state of a single switch. If the switch is off, then the bit has the value zero. If the switch is on, then the bit has the value one.

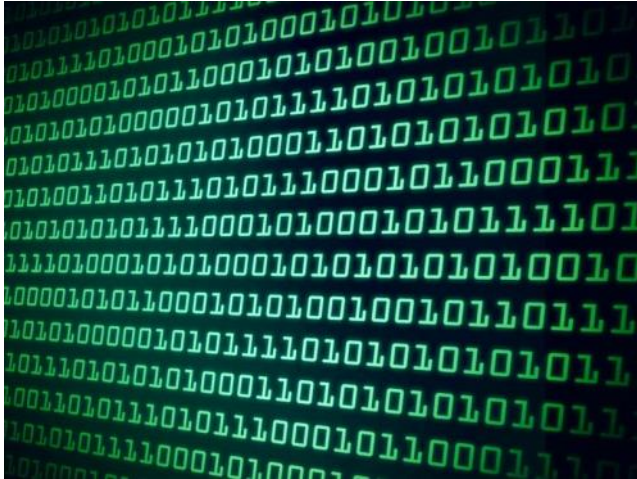
A bit can only represent two different values, zero or one. To represent larger pieces of information, bits are strung together in sequences of 8 called bytes.

A byte is a sequence of binary digits made up of 8 bits.

A byte can represent any value from 00000000 through 11111111, for a total of 256 different possible values. Each digit in a byte can be thought of as representing an individual switch that is either off (zero) or on (one).

Modern computers rely on transistors, which pack millions of tiny switches into a chip smaller than your

thumb, but information is still represented in essentially the same way: as a series of ones and zeros. By using binary, computers can represent information simply and efficiently using a system that is very effectively modeled in digital circuitry.



Binary numbers on a terminal monitor

Review

- A Bit is a binary digit with two possible values, zero or one
- A Byte is a sequence of 8 bits and has 256 possible values from 00000000 through 11111111
- A kilobyte (kB) is 1,024 bytes or 2^{10} bytes
- A Megabyte (MB) is 1,048, 576 bytes or 2^{20} bytes
- A Gigabyte (GB) is 1,073,741,824 bytes or 2^{30} bytes
- A Terabyte (TB) is 1,099,511,627,776 bytes or 2^{40} bytes

Notes

- The ones and zeros of bits and bytes can be used to represent letters, numbers, and even different keys on a computer keyboard.
- A bit can be used to hold a Boolean (true/false) value. A value of zero represents 'false' and a value of one represents 'true'.

Unplugged: Binary Vending machine

In this activity, students will explore the concept of binary numbers by experimenting with a very odd vending machine that only accepts Base-2 coins and doesn't give change! In the process, students will become familiar with an alternate numbering system, in this case binary (Base-2). Students will learn how binary relates to decimal, and will be able to convert between the two systems.

Materials

- Paper
- Pencil
- Set of 'coins' - these could be checkers/chess pieces, cardboard rounds, or even post-it notes
- Vending machine visual (optional)

Pre-activity preparation

Gather or create small counters or 'coins' in the following denominations: 1, 2, 4, 8, 16, 32
Plastic white poker chips work well as coins. You can write the denominations onto one side of the coins with a whiteboard marker. You can also use small index cards or paper squares. Make sure to leave one side of each coin blank.

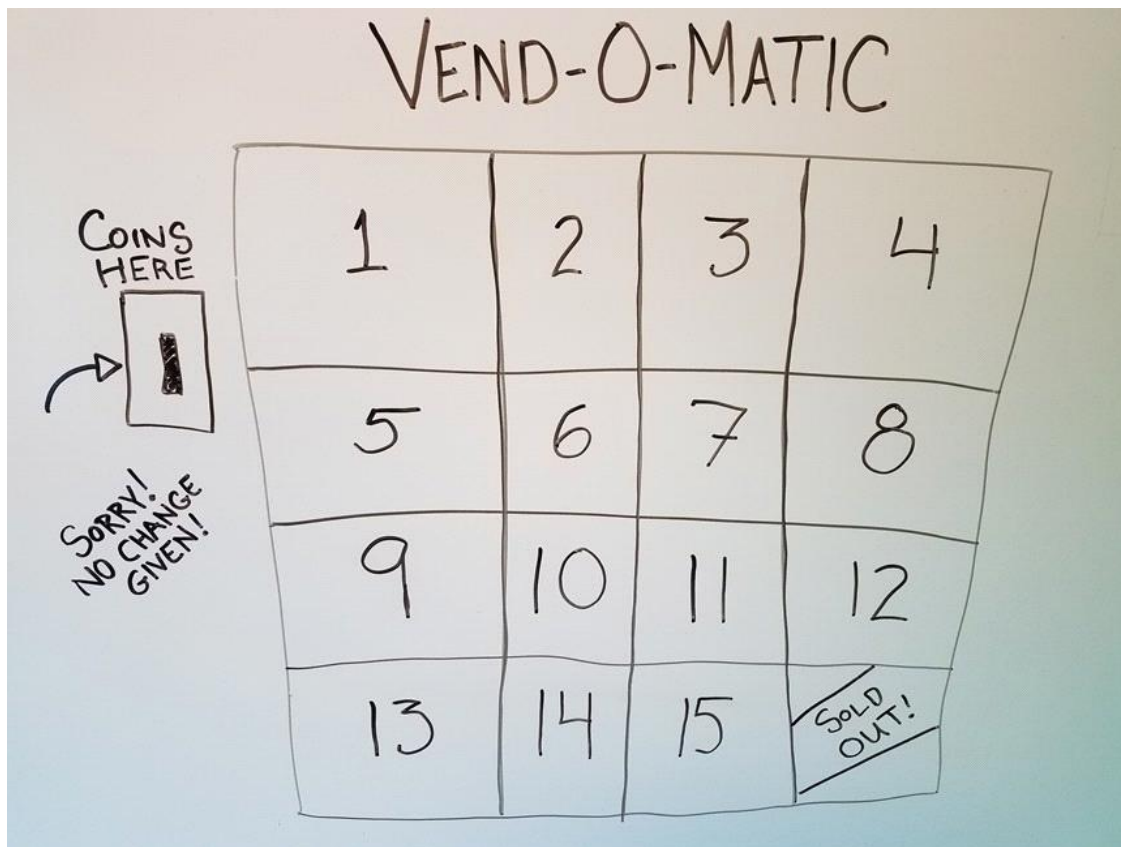
Amount: One set of coins (with one coin of each of the first four denominations in it) for each student or each pair of students.

Hold onto the 16 and 32 unit coins for later.

Tip

If you have time, create on a poster board, on the whiteboard, or on paper as a handout, a big rectangle representing a vending machine. Draw in different items for purchase that would appeal to the students. Have the different items priced differently from 1 unit to 15 units. This is particularly good to have for younger and more visually oriented students.

You can also just make a very simple vending machine diagram like the one below:



Introduction:

Ask the students the following questions to spark discussion:

- Have any of them bought anything in the last 24 hours? *Usually they have bought a snack or perhaps lunch*
- Did any of them use cash?
- What bills or coins did they use?
- What are the core denominations of money in the United States?

Lead the students to realize that our core monetary denominations, like our number system are based on ten.

- 1 penny
- 1 dime = 10 pennies
- 1 one dollar bill = 10 dimes (or 100 pennies)
- 1 ten dollar bill = 10 one dollar bills
- 1 hundred dollar bill = 10 ten dollar bills

Our money system is based on our number system, the decimal system. The *deci-* prefix means 'one tenth'. Each place value in the decimal is one tenth of the place value to its left.

For example: The amount eleven is written in decimal notation as 11.

There is a numeral one in the 'tens place' and a numeral one in the 'ones place'.

The leftmost numeral one in the 'tens place' represents one ten.

The next numeral one represents an amount that is *one tenth* the amount of the place value to its left; in this case, one tenth of ten, or one.

But what is it like to use a different monetary system? A monetary system that has a base other

than ten?

Process

- Give a set of the coins you prepared earlier to each student or pair of students
- Remember to hold onto the 16-unit and 32-unit coins for now
- Present the following scenario:
 - There is a vending machine that sells items of all prices
 - However, the machine cannot give change
 - Therefore, you must pay for everything in exact amounts
 - You have one of each coin: 1, 2, 4, 8.

Questions

- What is the price of the least expensive item you can buy? (1 unit)
- What is the price of the most expensive item you can buy? (15 units)
- What else can you buy? What coin(s) would you use to do this?
- What is the price of something you cannot buy, because you don't have exact change?

Here is where students will start to figure out the different combined sums of different coins. You can also prompt them by saying, for example, "It's impossible to buy something that costs 11 units, isn't it?" Someone will immediately point out that you CAN buy an 11-unit item with $8 + 2 + 1$.

You can now have the students write down how they could pay, what coin(s) could they use to purchase each of the items priced 1 unit through 15 units with the coins they have OR have a whole class discussion with you keeping track of their methods of payment on the whiteboard.

There will soon be a general agreement among the students that:

- You can make every amount between 1 unit and 15 units with the 4 coins in their set
- There is only one way to make each of those amounts.

Have students line up the coins in their set from greatest to least denomination, left to right.

Questions

- What do you notice about the denominations as they increase from right to left? *Each amount is double (or times 2 or twice) the denomination before it (to its right)*
- If we added one more coin to your set of coins that is greater than the 8 unit coin, what is the next logical coin denomination? *16. Why? Because 16 is '2 times' greater than 8*

Hand out the 16 unit coins, one to each student or pair of students.

Questions

- What is the new maximum price you could pay for an item? *31*
- What combinations of coins can you use to pay for an item priced from 16 units to this new maximum price?

Once again, you can now have the students write down how they could pay, what coin(s) could they use to purchase each of the items priced 16 units through the new maximum price with the coins they have, OR have a whole class discussion with you keeping track of their methods of

payment on the whiteboard.

Again, there will soon be a general agreement among the students that:

- You can make every amount between 16 units and the new maximum with the 5 coins now in their set.
- There is only one way to make each of those amounts.

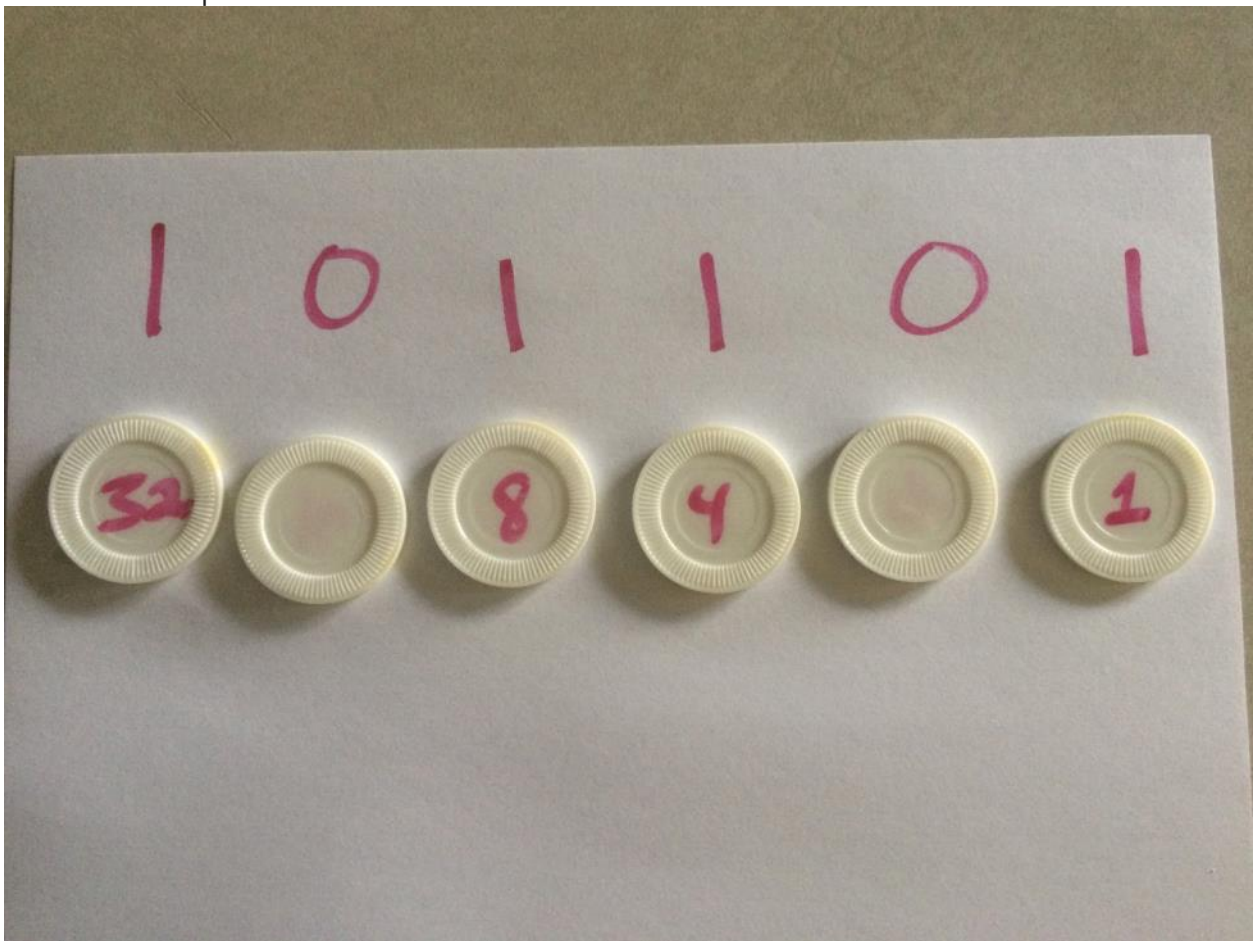
Questions

- If we added one more coin to your set of coins that is greater than the 16 unit coin, what is the next logical coin denomination? 32. Why? *Because 32 is '2 times' greater than 16.*

Hand out the 32 unit coins, one to each student or pair of students.

Questions

- What is the new maximum price you could pay for an item? 63
- What combinations of coins can you use to pay for an item priced from 32 units to this new maximum price?



From coins to binary notation - the number 45

Once students are comfortable making combinations of numbers, encourage them to use ones and zeroes to represent the numbers instead. This number system uses the number 2 as its base (each place is two times the one before it.) It is called the Base-2 system, or binary system. The number system we are normally familiar with is the Base-10 system, or decimal system (each place is ten times the one before it.)

With their coins in a line in descending order from right to left on a piece of paper, ask students to represent a given number by keeping face up the coins they would use to make this amount and flipping over or putting face down the coins not used.

For example: Ask them to represent the number 45. *See image above.*

They should have the 32, 8, 4, and 1 coins face up and the 16 and 2 coins face down. Ask the students to place a numeral 1 above the coins that are face up and a numeral zero over the coins that are face down.

The ones and zeros they just drew are the binary number version of the amount represented by the flipped-up coins. For the example: 45 in Base-10 = 101101 in Base-2

Practice translating numbers from Base-10 to Base-2

The students can now use this same method to translate other numbers from Base-10 to Base-2.

Examples:

22 (1 0 1 1 0)
37 (1 0 0 1 0 1)

Practice translating numbers from Base-2 to Base-10

Next, have the students use the above method in reverse to translate numbers from Base-2 to Base-10.

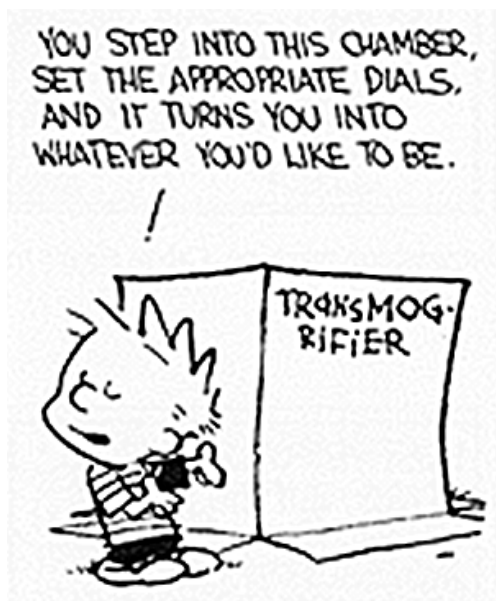
- Start with all the coins face up in a line from greatest to least denomination from left to right.
- Write the ones and zeros representing the binary number being translated above the coins.
- Flip to face down any coin with a zero above it.
- Add up the remaining face up coins.

Examples:

0 1 0 1 0 (10)
1 1 0 1 1 0 (54)

Activity: Binary Transmogriker

Guide the students through building a binary transmogrifier (converter) that converts between binary (base-2) and decimal (base-10) numbers. Let them figure out a pattern that will allow them to do the conversion on the fly.



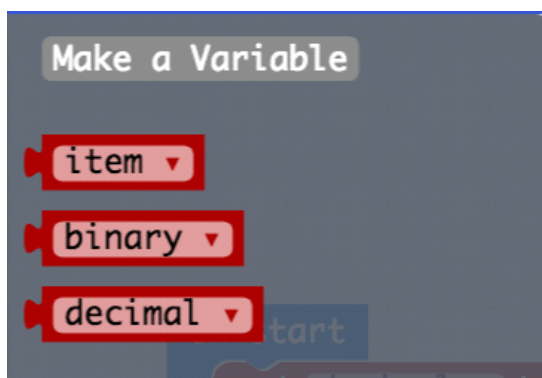
Calvin & Hobbes

Tell the students that they will be building a binary transmogrifier with the micro:bit. The user will be able to use the buttons to enter binary 0s and 1s and will be able to press A+B at any time to display the decimal equivalent of the number that has been entered.

Create the Variables

Students will need to create a number variable to hold the running decimal total. They should also create a string variable to hold the current binary number.

- From the Variables menu, make and name these two variables: decimal, binary.



New variable name:

decimal

Ok ✓

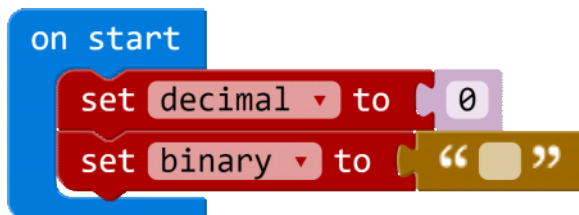
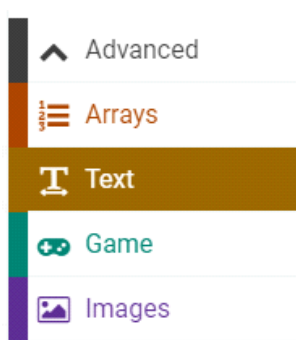
Cancel ✕

Initialize the Variables

When the program starts up, you should initialize your variables to starting values.

- decimal = 0
- binary = "" (empty string)

This also tells the micro:bit what type of variable it is. Use the empty string value found in the Text Toolbox drawer, under the Advanced menu.



```
let binary = ""  
let decimal = 0
```

By setting the binary variable to an initial value of "" you tell the micro:bit that it is a string variable: a literal string of characters. This is important because you will be adding to this string character by character.

Transmogrify Me!

We are ready to start entering numbers. Remember that binary numbers are calculated based on the number of place values ("bits") and as you enter 1s and 0s, the value changes. One way to calculate the decimal value is to wait until the user presses A+B, and then calculate the entire number based on the value of the string.

However, a much simpler method is to calculate the decimal number "on the fly", which is to say, every time the user presses a 1 or a 0, calculate the current decimal value of that string so you only have to deal with one 0 or 1 at a time.

What's the Pattern?

This is a table of the first fourteen binary numbers and their decimal equivalents. Your goal is to use this table to figure out how to calculate a new correct decimal value based on whether a user enters a 0, or a 1 as the next number in the string.

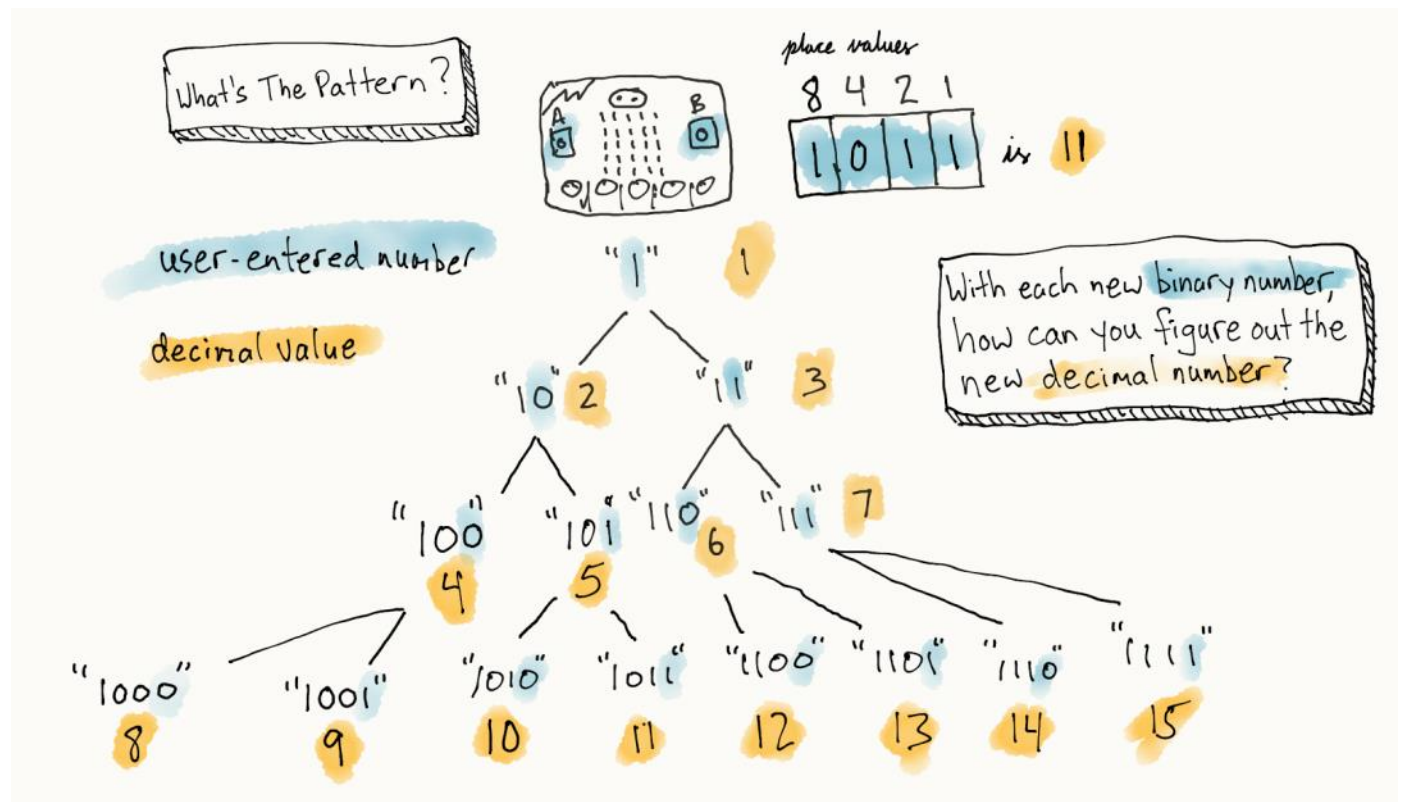
Binary	Decimal	Binary	Decimal
1	1	1000	8
10	2	1001	9
11	3	1010	10
100	4	1011	11
101	5	1100	12
110	6	1101	13
111	7	1110	14

For example, imagine you are the micro:bit. If the first number the human enters is a 1, you automatically know the new decimal value is a 1. If the second number that is entered is a 0, then your decimal value goes from 1 to 2. However, if the second number is also a 1, then your new decimal value goes from 1 to 3.

At that point, you either have a 10, or a 11 in your binary string. Let's take 10 as an example. The decimal value of binary 10 is 2. If the third number entered is a 0, then your new decimal value goes from 2 to 4. If the third number entered is a 1, then your new decimal value goes from 2 to 5.

If, on the other hand, you have 11 in your binary string, then your decimal value is 3. If the third number entered is a 0, then your new decimal value goes from 3 to 6. If the third number entered is a 1, then your new decimal value goes from 3 to 7.

See if you can spot a pattern that will help you figure out, for any given decimal value, what the new decimal value should be if the user enters a 0, or if the user enters a 1.



Pseudocode

Recall from our Algorithms lesson that it is a good idea to write out your algorithm in plain English, before you start coding in MakeCode. This is called pseudocode. The Input for this program will be the buttons. Try to write out what should happen when each of the buttons is pressed.

Here is one possible solution. Your own pseudocode might be different and that's okay.

When Button A is pressed:

1. Add a "1" to the end of the binary string.
2. Show the current value of the binary string.
3. Update the decimal value with the total.

When Button B is pressed:

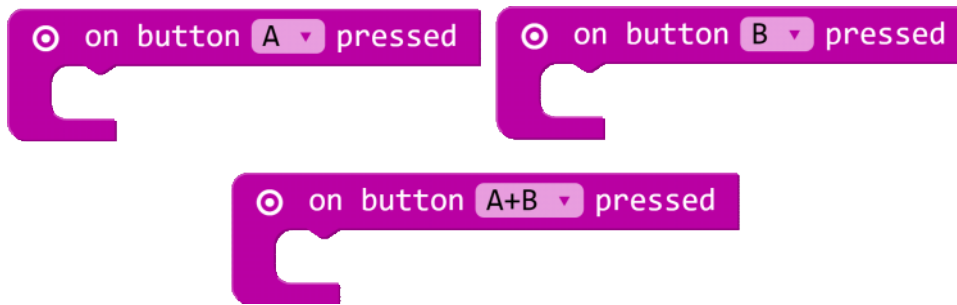
1. Add a "0" to the end of the binary string.
2. Show the current value of the binary string.
3. Update the decimal value with the total.

When Buttons A+B are pressed:

1. Show the current value of the decimal string.

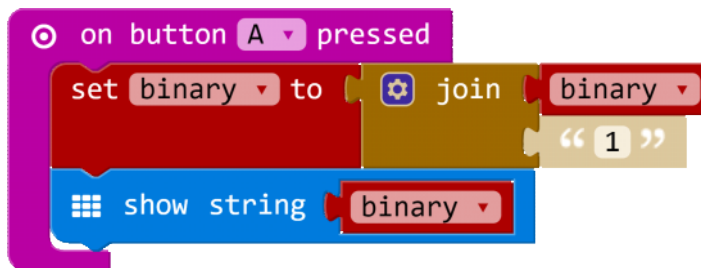
Coding Steps

- From the Input Toolbox drawer, drag 3 of the 'on button A pressed' event handlers to your coding workspace
- Leave one block with button 'A'. Use the drop-down menus in the other 2 blocks to choose button 'B', and button 'A+B'

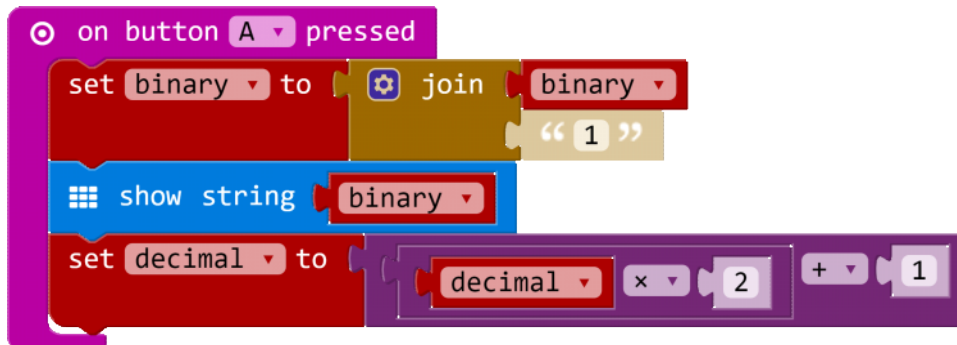


Let's work on what to do when button A is pressed.

- Button A represents a binary "1". Our first task is to join a "1" to the existing string variable called binary.
- From the Text Toolbox drawer (under the Advanced menu), drag the 'join' block to your programming workspace
- Next, use the 'set' variable block to assign the value of the 'binary' variable to the 'join' block
- Join the 'binary' variable and "1" by entering them into the appropriate slots in the 'join' block
- And show the binary value on the screen so that when users press a button they can see the entire binary string

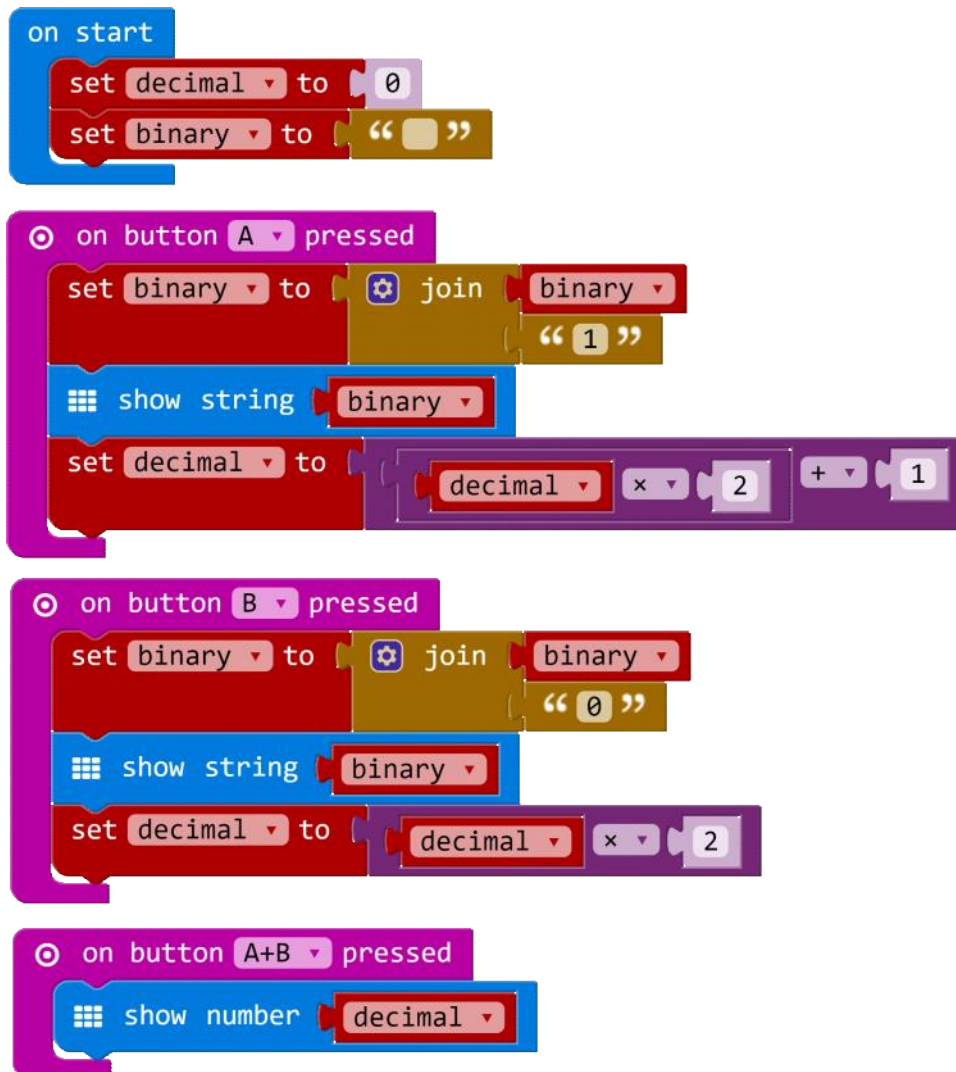


- Finally, you will need to update the current decimal value with the value of the entire binary string. This is pretty straightforward if you have been keeping track of the decimal value every time someone presses a button. The pattern is as follows: *(spoiler alert!)*
 - Whenever someone enters a 0, the new decimal value is twice the previous value.
 - If someone enters a 1, the new decimal value is twice the previous value, plus 1.
- For Button A, you will need to use the multiplication Math block and your binary variable block to create the proper formula. You will need to put that formula inside another Math addition block in order to add one to the result.



- Your Button B algorithm is similar, although you will be joining a "0" to the binary variable and you are just multiplying the decimal variable by 2.
- Your Button A+B algorithm just uses a Show block to show the value of the decimal variable.

Here is the completed program.



```

let binary = ""
let decimal = 0
input.onButtonPressed(Button.A, () => {
  binary = binary + "1"
  basic.showString(binary)
  decimal = decimal * 2 + 1
})
input.onButtonPressed(Button.B, () => {
  binary = binary + "0"
  basic.showString(binary)
  decimal = decimal * 2
})
input.onButtonPressed(Button.AB, () => {
  basic.showNumber(decimal)
})

```



```
decimal = 0  
binary = ""
```

[Transmogriifier](#)



Try it out!

Have someone else try your program out. Then think about how the program might be improved.

Here are some additional modifications you might try:

- Add a way to clear the binary and decimal values so you can start over.
- Add a way to erase the previous value.
- Create a decimal-binary converter that allows you enter a decimal value and see the binary equivalent when you press A+B.

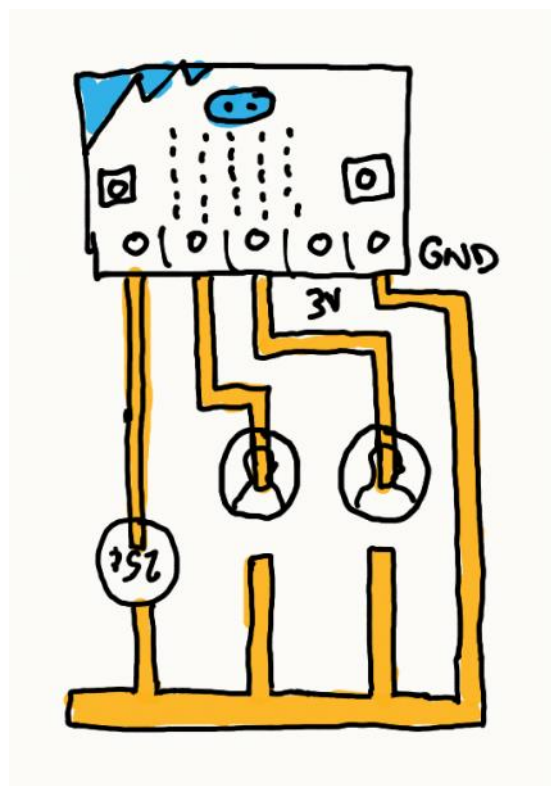
Project: Make a Binary Cash Register

The unplugged activity uses a vending machine as a model for creating different combinations of binary place values. We found that for n coins, there is one and only one way to make every number between 0 and $2^n - 1$.

For this project, students should invent a paper and cardboard version of the binary counter, then program it to display the decimal value of those numbers.

Materials

- Cardboard or heavy paper
- Copper tape - <https://www.adafruit.com/product/1128>, <https://www.sparkfun.com/products/10561>
- 3 quarters or other heavy coins
- Scissors
- Duct tape



Binary micro:bit Cash Register

Tips

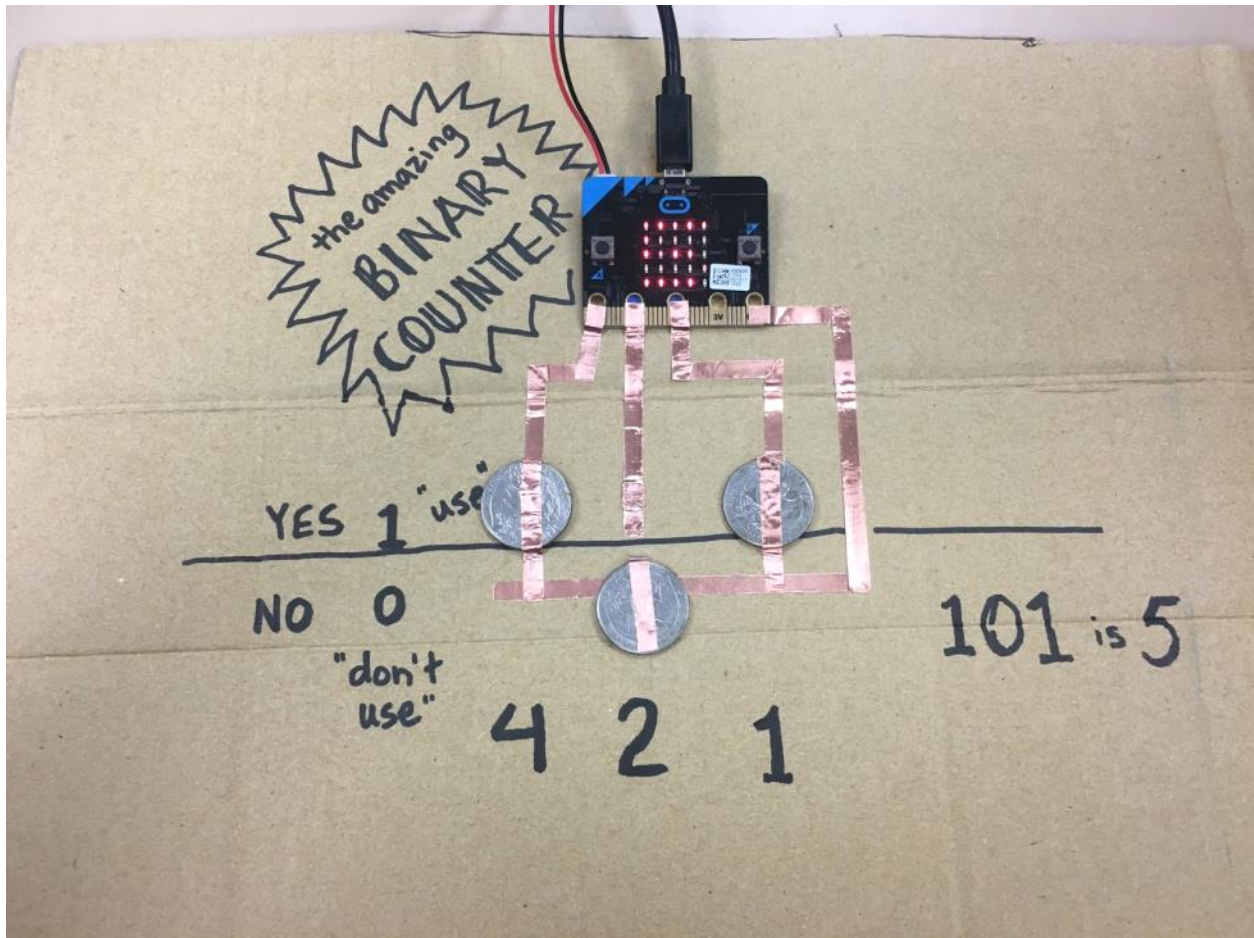
This is one possible design for a binary cash register. We used coins and copper tape on a piece of cardboard. Normally, the coins are flipped up ("off" or 0) and to indicate "on" or 1, the coin is flipped so it lays flat across both pieces of copper tape, completing the circuit so the micro:bit can detect that that pin has been activated, and calculates and displays the decimal value of the binary number that is indicated by the coins.

Copper tape is a thin, flexible strip of copper with an adhesive back. You can sometimes find

copper tape at the hardware, sold as slug tape, to keep slugs out of your garden. Usually, copper tape can conduct electricity even through the sticky side but if you are sticking one piece of copper tape to another, be sure to go over the connection with your fingernail, pressing it down firmly.

Because the micro:bit only has three pins, this binary register is limited to three place values. Students might use variables to represent each of the three place values, or they can simply keep a running total by adding the appropriate amount when each of the three pins is pressed.

You can stick the micro:bit into place using some sticky tape, or you can create an actual holder. The copper tape connections are delicate though, so be careful when plugging and unplugging the power cable from the board.



An implementation of the Binary Cash Register

Extra Mods

- Write some code that will display the number in binary when you press the A button.
- Think of a way to create more place values, perhaps by using a second micro:bit and a Radio connection.

Optional Project: Build a Binary Wristwatch

- Write a program that will display the correct time (once set) on the micro:bit.
- The 3-4 numbers displayed will be in binary (not decimal).
- To make the strap of the wristwatch, put 2 pieces of duct tape back-to-back, and use velcro tabs as the fasteners



Reflection:

Have students write a reflection of about 150–300 words, addressing the following points:

- What were the Variables that you used to keep track of information?
- What mathematical operations did you perform on your variables?
- What information did you provide?
- Describe what the physical component of your micro:bit project was (e.g., an armband, a wallet, a holder, etc.)
- How well did your prototype work? What were you happy with? What would you change?
- What was something that was surprising to you about the process of creating this project?
- Describe a difficult point in the process of designing this project, and explain how you resolved it.

Assessment

	4	3	2	1
Binary display	All binary numerals display correctly	At least 2 binary numerals display correctly	At least 1 binary numeral displays	No binary numerals display correctly.
Micro:bit program	Micro:bit program: <ul style="list-style-type: none"> • Uses binary in a way that is integral to the program • Uses mathematical operations to add, subtract multiply, and/or divide variables • Compiles and runs as intended • Meaningful comments in code 	Micro:bit program lacks 1 of the required elements	Micro:bit program lacks 2 of the required elements	Micro:bit program lacks 3 or more of the required elements
Reflection	Reflection piece	Reflection piece	Reflection piece	Reflection piece

	includes: <ul style="list-style-type: none">• Brainstorming ideas• Construction• Programming• Beta testing	lacks 1 of the required elements.	lacks 2 of the required elements.	lacks 3 of the required elements.
--	---	-----------------------------------	-----------------------------------	-----------------------------------

Additional Questions to Ponder

- How could you use a row of flashlights to represent a number to someone else far away?
- How might you use those flashlights to send a message?

Resources

- <https://www.mathsisfun.com/binary-number-system.html>

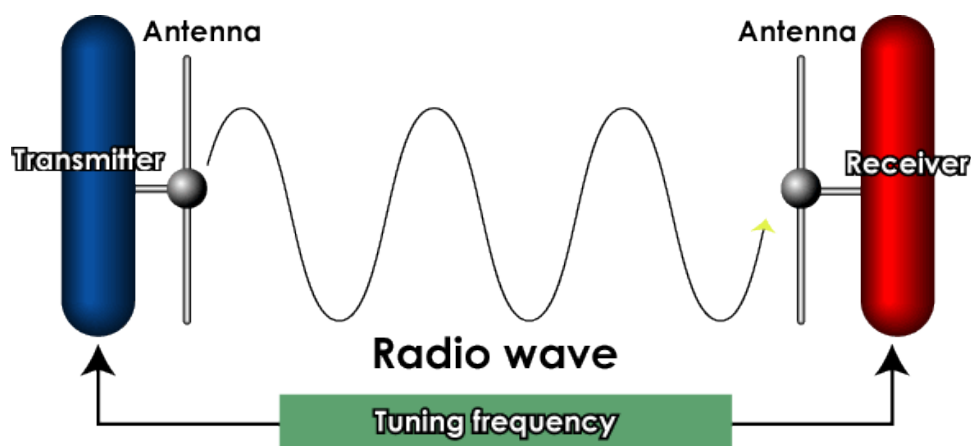
Standards

CSTA K-12 Computer Science Standards

- CT.L1:3-05 Demonstrate how 0s and 1s can be used to represent information.
- CT.L1:6-03 Demonstrate how a string of bits can be used to represent alphanumeric information.
- CT.L2-14 Examine connections between elements of mathematics and computer science, including binary numbers, logic, sets and functions.
- CT.L3B-07 Discuss the interpretation of binary sequences in a variety of forms (e.g., instructions, numbers, text, sound, image).
- CPP.L1:6-06 Implement problem solutions using a block-based visual programming language.

Radio and Communication

This lesson covers the use of more than one micro:bit to share and combine data. Students will explore a complex epidemiological program (Infection) that demonstrates the Radio functionality of the micro:bit. Students will send and receive numbers and strings in a series of guided activities. Finally, students are asked to collaborate so that they can share their micro:bits and create a project together.



Lesson Objectives:

Students will...

- Understand how to use the Radio blocks to send and receive data between micro:bits
- Understand the specific types of data that can be sent over the Radio

Lesson Plan Structure

- Introduction: Radio & Communication
- Unplugged Activity: Infection Simulation
- Micro:bit Activity: Marco Polo & Morse Code
- Project: Radio
- Assessment: Rubric
- Standards: Listed

Introduction

Up to this point, we have been primarily challenging students to collaborate while they create their own projects. This lesson, on communication using the micro:bit radio, is a great opportunity to have students work in pairs on a project. Have kids find a partner to work with for this lesson, and make sure they are seated next to each other.

Note: Many teachers find the concept of “pair programming” to be a valuable way to have students collaborate when programming. Two students share one computer, with one student at the keyboard acting as the driver, and the other student providing directions as the navigator. Students must practice good communication with each other throughout the entire programming process.

The micro:bit allows you to communicate with other micro:bits in the area using the blocks in the Radio category. You can send a number, a string (a word or series of characters) or a string/number combination in a radio packet. You can also give a micro:bit instructions on what to do when it receives a radio packet.

This lesson starts with a “plugged” unplugged activity, in which students use their micro:bits to explore an advanced simulation. The code is quite complex, so students will focus more on how to use the micro:bits to explore aspects of viruses and epidemics, than the intricacies of the code itself.

The project for this lesson will challenge students to work together to send and receive some sort of data to and from each other. There is a wide range of simple and complex projects kids can try, but whatever they choose it is a whole lot of fun to communicate with each other using the micro:bits!

Unplugged: Infection Simulation

For this activity, each student will need a micro:bit and battery pack, as well as the teacher who will be the Master controller.

There are four parts to this unplugged activity:

- Setup: Set up the code on all micro:bits
- Explore: Let students experience the game first
- Discuss: Talk about observations, theories, propose strategies
- Test: Play again, testing different strategies and approaches for containing outbreak

The goals of this activity are:

- Develop a common working vocabulary for talking about disease spread
- Make inferences based on observation
- Propose and test original hypotheses to explain complex behavior
- Explore a professionally developed micro:bit simulation



Setup

This site is the home page for the Infection game:

<https://pxt.microbit.org/projects/infection>

On that page you should be able to copy the JavaScript code, then go to your MakeCode JavaScript editor and paste the JavaScript code into the window.

Then click the Download button to download this program onto your micro:bit. This code should be downloaded onto all of your students' micro:bits as well as your own.

This activity works best in an open area. If it's possible to go outside, that works even better! To set up the game, the teacher should press the A + B buttons on his or her device. This will register all of the student devices and establish the teacher's micro:bit as the Master device.

Explore

In this phase, students should just play the game to get a feel for how it works. The object of the game is to meet as many people as possible without getting sick. If at any time players decide to stop meeting people, they should sit down and cover their micro:bit.

To start the game, students should take their devices and spread out. When everyone is ready to begin, the teacher should press the A + B buttons again. All of the student devices will show a unique player icon.

One of the players is randomly chosen to have a virus that is transmitted when they meet other players. Players can meet each other by going up to another player and placing the two devices next to each other. Players who are healthy, or who are infected but are not showing symptoms yet, will have a smile. Once a player is sick, their

micro:bit will display a frowny face.

After a certain period of time being sick, the player dies and the micro:bit will display a skull icon. That player should sit down and wait for the game to end, when all players are dead or the virus stops spreading.

Discuss

After one round, it is good to have a discussion with the players:

- Did anyone manage to stay healthy? If so, how? If not, why not?
- How do you think the disease spread?
- Who do you think started it?
- What could we do to find out?
- What strategies might we adopt next time, to have a better outcome?

Test

Play the game one more time, or more depending on available time, and attempt to test some of the theories students came up with.

- What strategies worked well?
- Which strategies seemed like a good idea at the time, but in practice, were less effective? Why?
- Are there any real-world situations that this might remind you of?

Vocabulary

As students talk through their theories, they will often talk about a scientific idea without knowing the specific word for it. This presents a nice opportunity, once students have surfaced an idea, to give it a proper name so that you can start to develop a common working vocabulary for talking about the problem.

Here are some common terms that come up in discussion:

- Asymptomatic: Someone who has the virus but is not showing outward symptoms of being sick.
- Carrier: Someone who has the virus and can transmit it to others.
- Immunity: Someone who cannot contract or transmit the virus.
- Incubation: The period of time between when a person contracts the virus and when the person starts to show symptoms of being sick.
- Inoculation: Make someone immune to the virus.
- Patient Zero: The first person to introduce a virus to a community.
- Quarantine: A strategy to isolate those who are suspected of carrying a virus

Reference

This game is a distributed simulation of a viral outbreak. It is modeled after the Thinking Tags participatory simulations developed at MIT Media Lab. Participatory Simulations have been found to enhance student understanding of complex dynamic relationships, inquiry skills, and scientific understanding. (Colella, V. (2000). Participatory Simulations: Building Collaborative Understanding Through Immersive Dynamic Modeling. *Journal of the Learning Sciences*, 9(4), 471–500. http://doi.org/10.1207/S15327809JLS0904_40)

Activity: Marco Polo and Morse Code

Guide the students in creating programs that use the radio communication blocks to send and receive data between two micro:bits.

Notes:

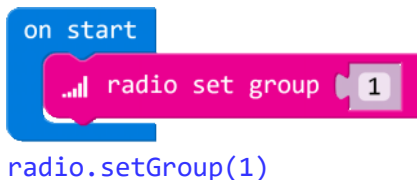
- When using the radio blocks, the micro:bit simulator will show two micro:bits
- In the simulator, a radio transmission icon will appear in the top right corner of the micro:bit. The icon will light up as the micro:bit is transmitting data.
- In the simulator, all the code in the coding workspace runs on both virtual micro:bits. You should include for how to send data as well as what to do when it receives data.

Marco Polo

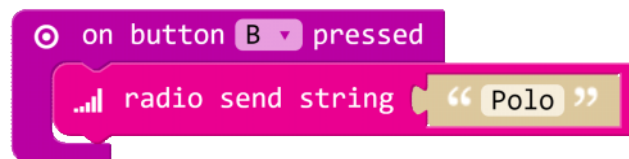
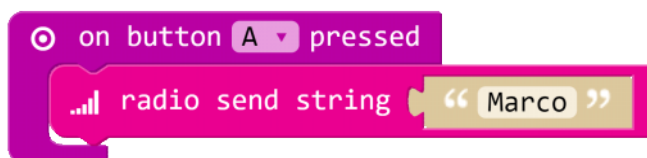
Send and receive strings between micro:bits.

On button A pressed, we will send the string Marco and on button B pressed we will send the string Polo.

- When communicating between micro:bits, it is important that the micro:bits involved are all using the same group ID. So, the first thing we will do is set the group ID number.
- From the Radio menu, drag a 'radio set group' block to the coding workspace and place the block into the on start block.
- In the 'radio set group block', leave the default value of 1 for the group ID



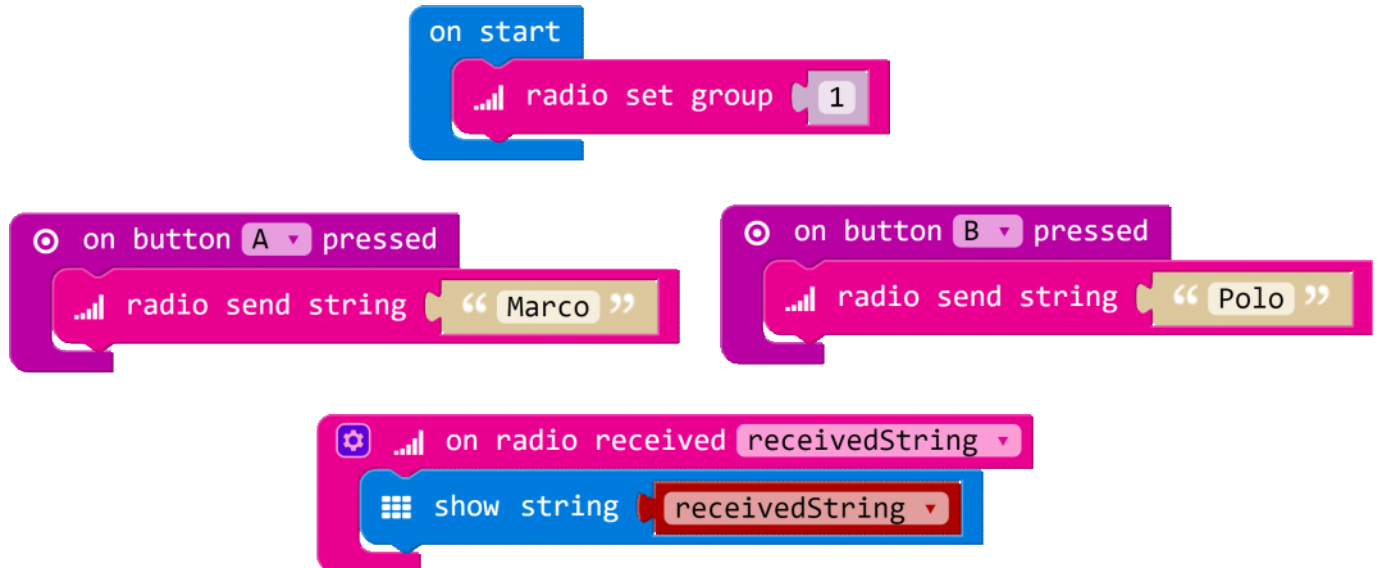
- Drag 2 'on button pressed' blocks to the coding workspace
- Leave one with the default value A and change the other button to B
- From the Radio Toolbox drawer, drag 2 'radio send string' blocks to the coding workspace
- Place one 'radio send string' block into the 'on button A pressed' block, and the other 'radio send string' block into the 'on button B pressed' block
- In the 'on button A pressed' block, change the default empty string value of the 'radio send string' block to the string "Marco"
- In the 'on button B pressed' block, change the default empty string value of the 'radio send string' block to the string "Polo"



```
input.onButtonPressed(Button.A, () => {  
  radio.sendString("Marco")  
})  
input.onButtonPressed(Button.B, () => {  
  radio.sendString("Polo")  
})
```

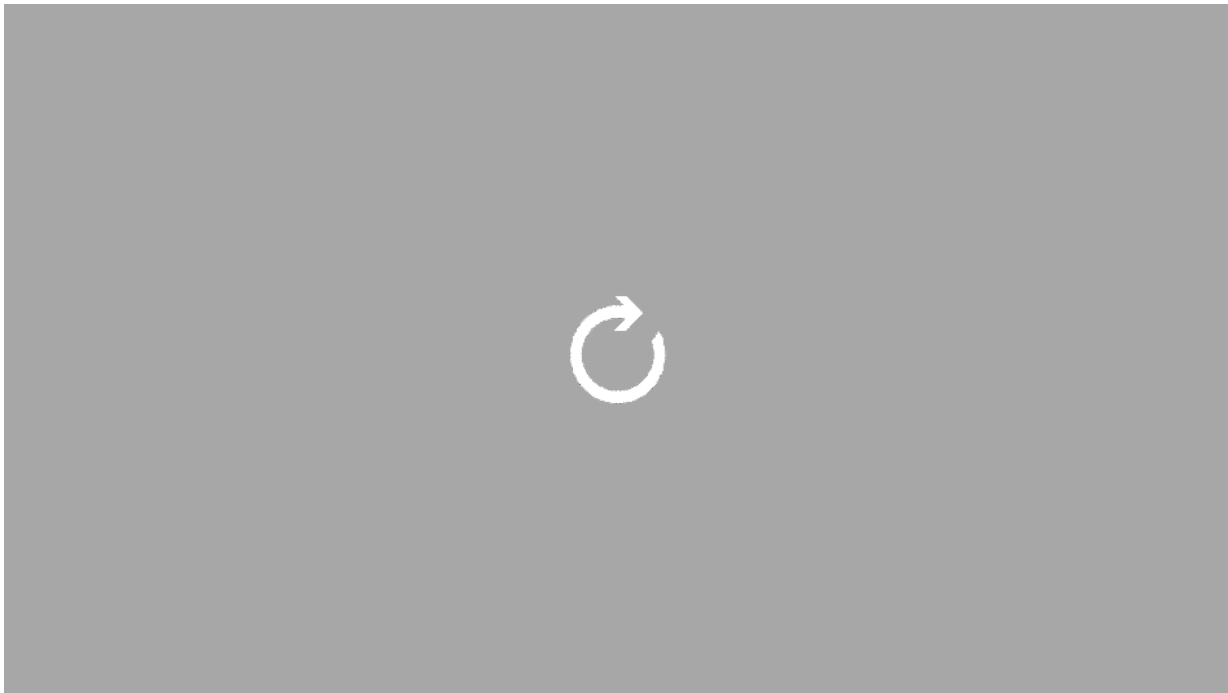
- To display the data sent between the micro:bits, drag an 'on radio received receivedString' block to the coding workspace
- From the Basic Toolbox drawer, drag a 'show string' block into the 'on radio received receivedString' block
- From the Variables Toolbox drawer, drag a 'receivedString' variable block into the default string value of "Hello" in the 'show string' block

Here is the complete Marco Polo program:



```
input.onButtonPressed(Button.A, () => {
  radio.sendString("Marco")
})
radio.onDataPacketReceived(({ receivedString }) => {
  basic.showString(receivedString)
})
input.onButtonPressed(Button.B, () => {
  radio.sendString("Polo")
})
radio.setGroup(1)
```

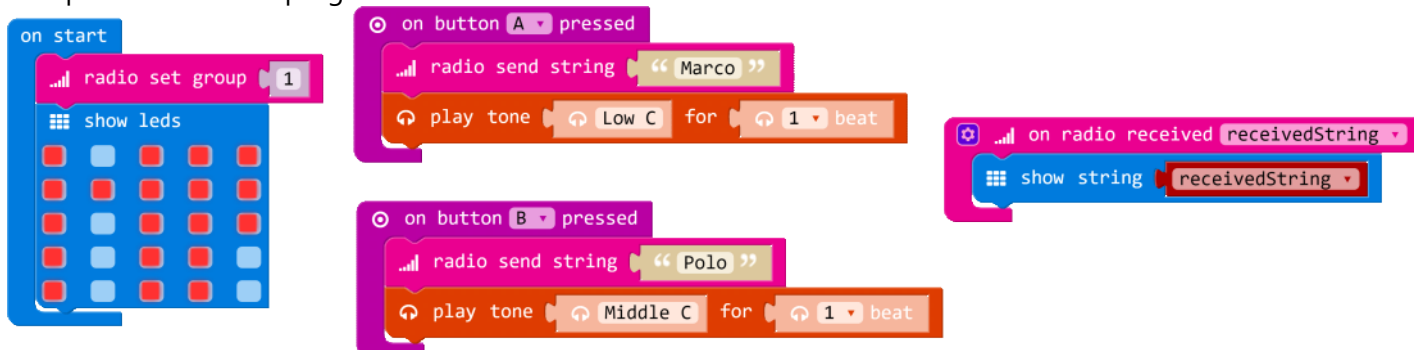
[MarcoPolo](#)



Mods

- Add a 'show leds' block to the 'on start' block. We created an image of the initials MP.
- From the Music Toolbox drawer, drag 2 'play tone' blocks to the coding workspace. See <https://makecode.microbit.org/projects/hack-your-headphones> for how to connect a speaker or headphones to the micro:bit.
- Drag one of the 'play tone' blocks to the 'on button A pressed' block, and the other one to the 'on button B pressed' block.
- Change the default value in the 'play tone' block that is inside the 'on button A pressed' block to the value Low C.

Complete Marco Polo program with mods:

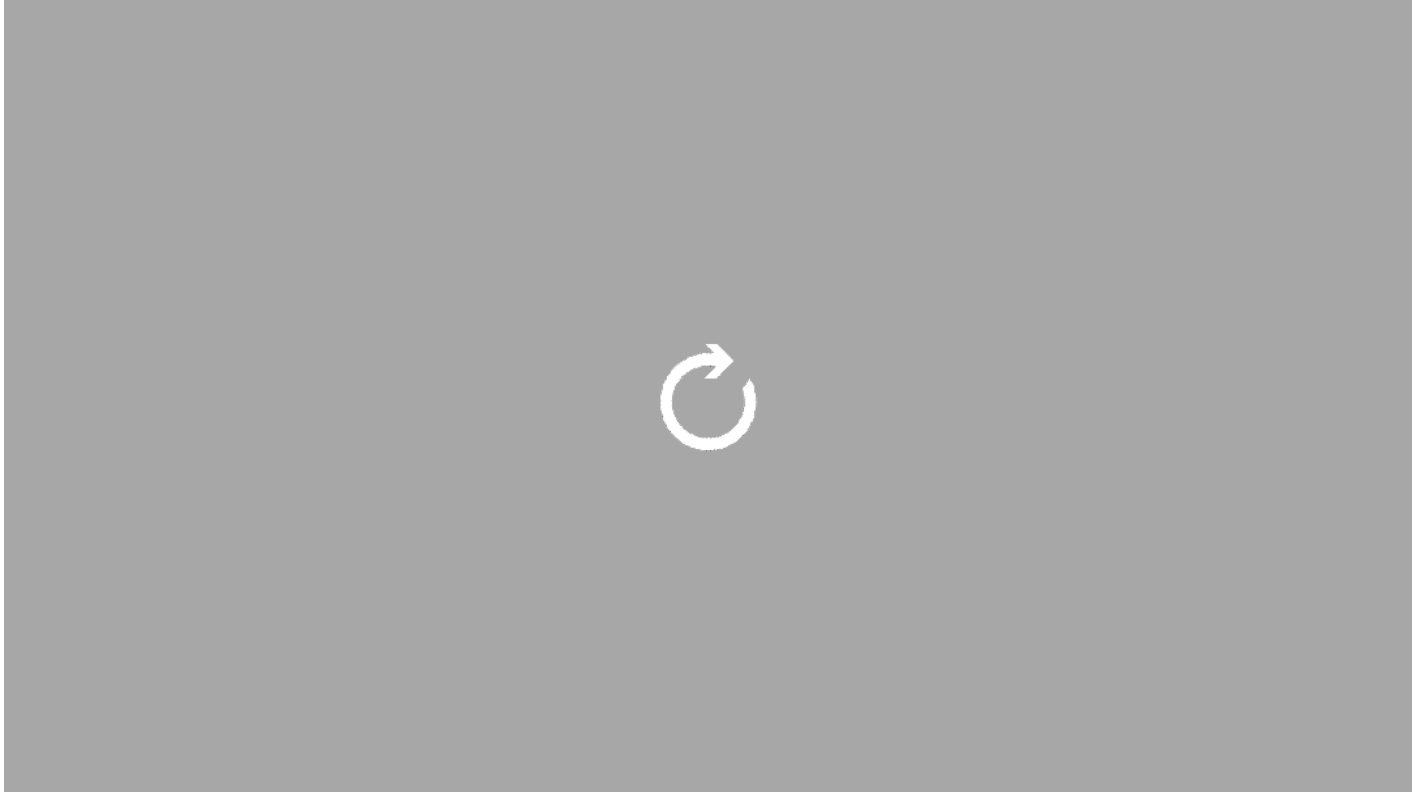


```
input.onButtonPressed(Button.A, () => {
  radio.sendString("Marco")
  music.playTone(131, music.beat(BeatFraction.Whole))
})
radio.onDataPacketReceived( ({ receivedString }) => {
  basic.showString(receivedString)
})
input.onButtonPressed(Button.B, () => {
  radio.sendString("Polo")
  music.playTone(262, music.beat(BeatFraction.Whole))
})
radio.setGroup(1)
basic.showLeds(`
```

. # # #

. # # #
. # # .
. # # .
`)

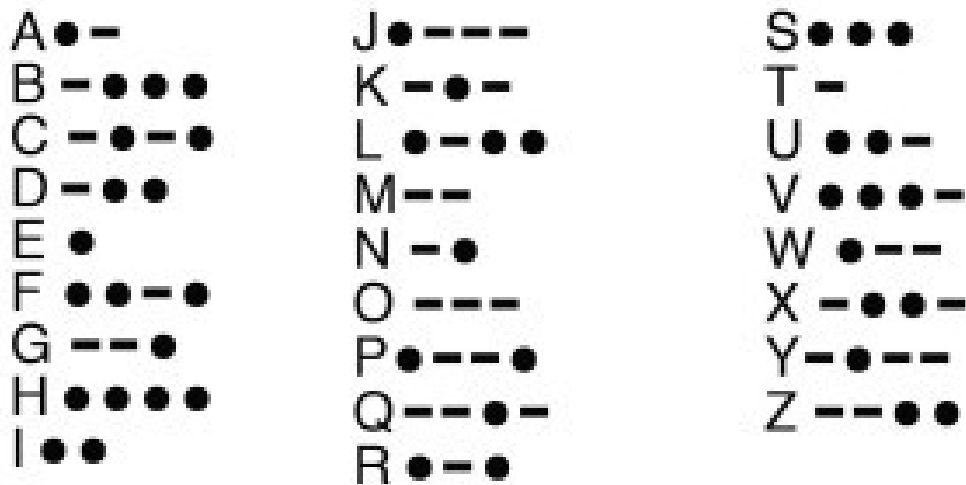
[MarcoPoloMods](#)



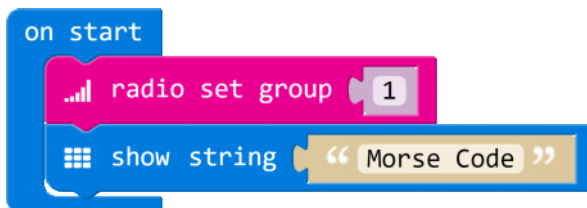
Morse Code

Send and receive numbers between micro:bits.

Depending on the button pressed, send a different number value between micro:bits. On receiving a number, display a different image unique to the number sent. One number will represent a dot, another a dash and another a space or stop.

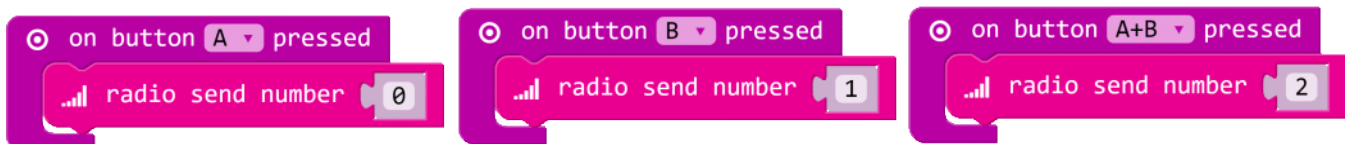


- Set the group ID number.
- Add a 'show string' block to the 'on start' block, to identify the program.
- We choose to change the default string value of "Hello" to the value "Morse Code"



```
radio.setGroup(1)
basic.showString("Morse Code")
```

- Drag 3 'on button pressed' blocks to the coding workspace.
- Leave one with the default value A, change the value in the second block to B, and change the value in the third block to A+B.
- From the Radio Toolbox drawer, drag 3 'radio send number' blocks to the coding workspace.
- Place one radio send number block into each of the 'on button pressed' blocks.
- In the 'on button A pressed' block, leave the default number value of the 'radio send number' block as 0.
- In the 'on button B pressed' block, change the default number value of the 'radio send number' block to the value 1.
- In the 'on button A+B pressed' block, change the default number value of the 'radio send number' block to the value 2.



```
input.onButtonPressed(Button.A, () => {
  radio.sendNumber(0)
})
input.onButtonPressed(Button.B, () => {
  radio.sendNumber(1)
})
input.onButtonPressed(Button.AB, () => {
```

```

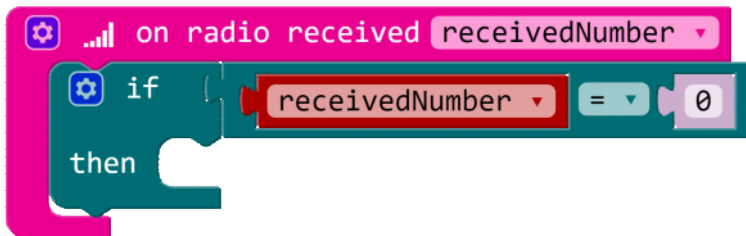
    radio.sendNumber(2)
  })

```

- From the Radio Toolbox drawer, drag an 'on radio received receivedNumber' event handler to the coding workspace.
- Since we will display a different image depending on the number value received, we need a logic block.
- From the Logic Toolbox drawer, drag an 'if...then' block to the coding workspace and place it in the 'on radio received receivedNumber' event handler.

In order to know whether to display a dot, a dash, or a space/stop image, we need to compare the number received to the values 0, 1, and 2.

- From the Logic Toolbox drawer, drag a 0=0 comparison block into the coding workspace.
- Replace the default value 'true' of the 'if...then' block with the comparison block.
- From the Variables Toolbox drawer, drag a 'receivedNumber' variable block into the coding workspace, and drop it into the first slot of the comparison block
- Leave the righthand side default value of zero in the 0=0 block.

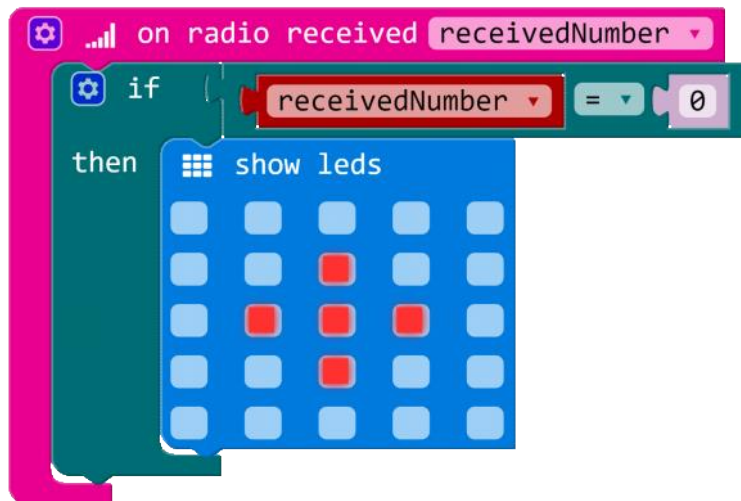


```

radio.onDataPacketReceived( ({ receivedNumber }) => {
  if (receivedNumber == 0) {
  }
})

```

- Place a 'show leds' block in the space after the then of the 'if...then' block.
- Create an image to represent a dot.



```

radio.onDataPacketReceived( ({ receivedNumber }) => {
  if (receivedNumber == 0) {
    basic.showLeds(`

```



```

    . . . . .
    . . # . .
    . # # # .
    . . # . .
    . . . . .
  `)
}
})

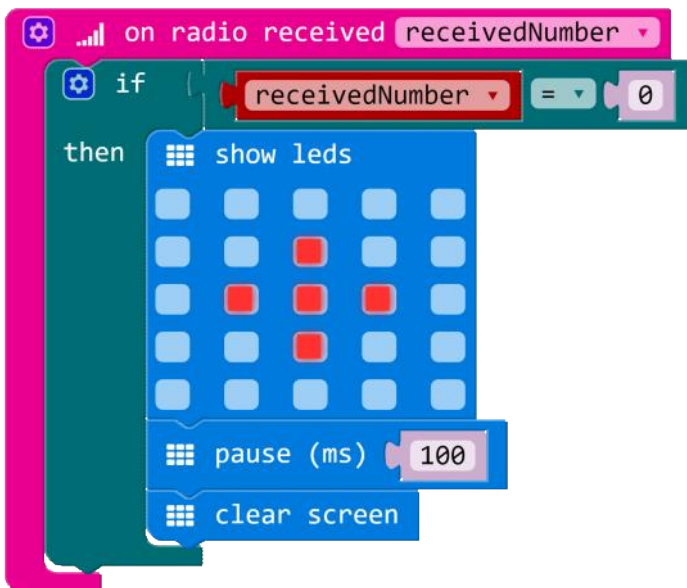
```

Try it!

- Download your program to the micro:bit
- Press button A on the sending micro:bit
- Does this cause a dot to be displayed on the receiving micro:bit?
- However, pressing button A again does not appear to send another dot as the image on the receiving micro:bit does not appear to change.

Challenge question: How can we fix this?

- Add a 'pause' block and a 'clear screen' block after the 'show leds' block



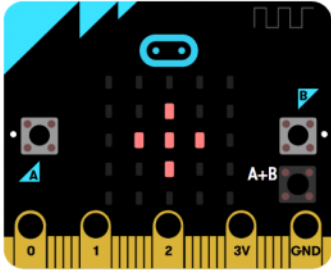
```

radio.onDataPacketReceived( ({ receivedNumber }) => {
  if (receivedNumber == 0) {
    basic.showLeds(`
      . . . . .
      . . # . .
      . # # # .
      . . # . .
      . . . . .
    `)
    basic.pause(1)
    basic.clearScreen()
  }
})

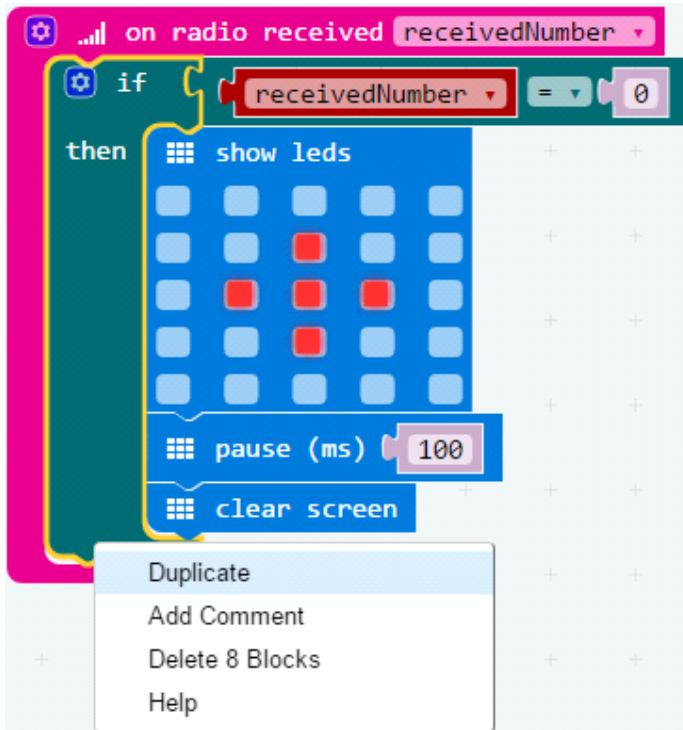
```

Try running the program again.

Now each time the sender presses button A, you see a dot appear.

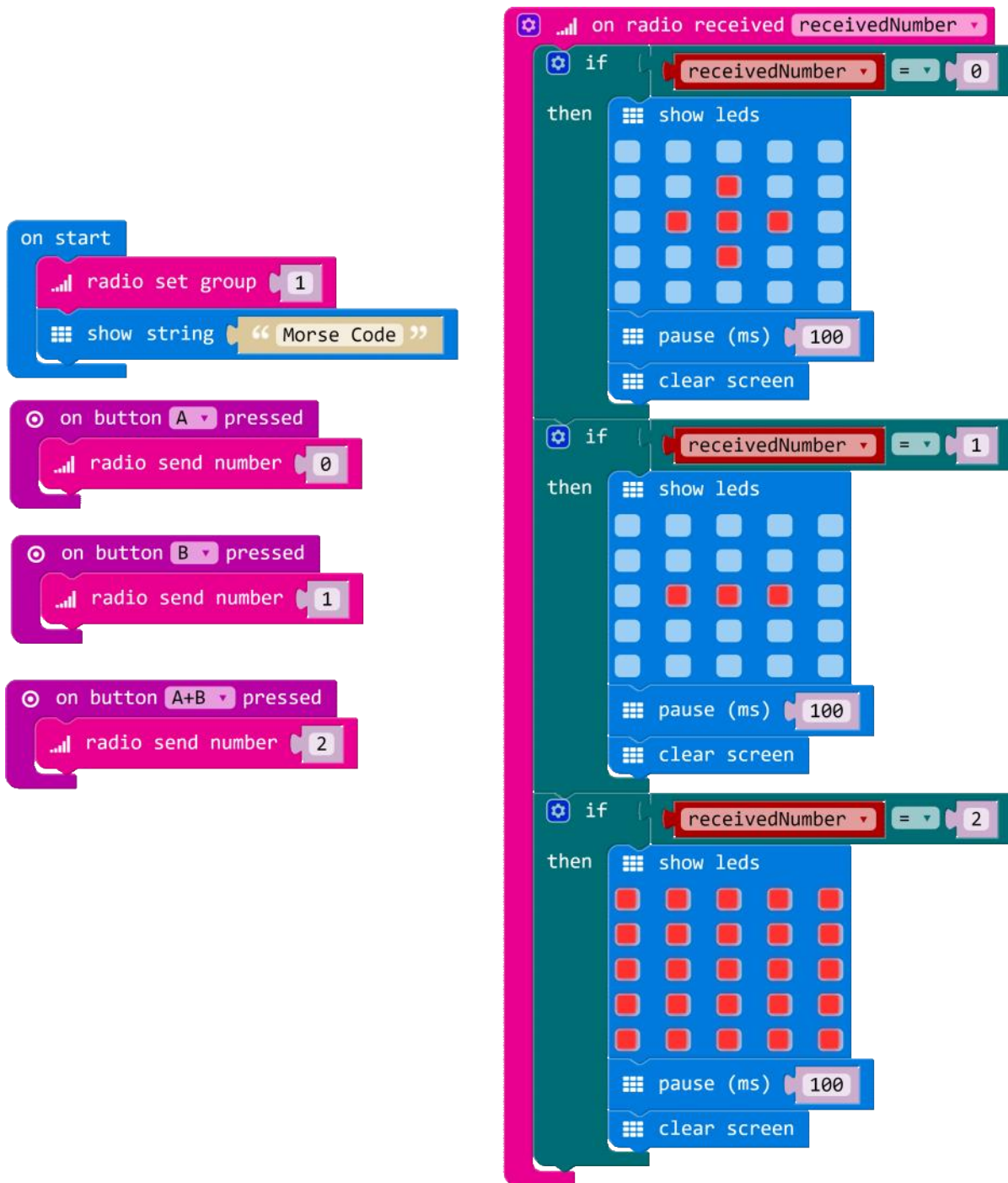


- You can now right-click on the 'if...then' block and select Duplicate to copy that piece of code twice for the other 2 values that a sender may send.



- Change the values on the righthand side of the comparison block to 1, and 2.
- Modify the images displayed to show a dash, and a full screen of lights

Morse Code Program:



```

radio.onDataPacketReceived( ({ receivedNumber }) => {
  if (receivedNumber == 0) {
    basic.showLeds(`
      . . . . .
      . . # . .
      . # # # .
      . . # . .
      . . . . .
    `)
    basic.pause(100)
    basic.clearScreen()
  }
  if (receivedNumber == 1) {
    basic.showLeds(`

```

```

        . . . . .
        . . . . .
        . # # # .
        . . . . .
        . . . . .
        `)
    basic.pause(100)
    basic.clearScreen()
}
if (receivedNumber == 2) {
    basic.showLeds(`
        # # # # #
        # # # # #
        # # # # #
        # # # # #
        # # # # #
        `)
    basic.pause(100)
    basic.clearScreen()
}
})
input.onButtonPressed(Button.A, () => {
    radio.sendNumber(0)
})
input.onButtonPressed(Button.B, () => {
    radio.sendNumber(1)
})
input.onButtonPressed(Button.AB, () => {
    radio.sendNumber(2)
})
radio.setGroup(1)
basic.showString("Morse Code")

```

MorseCode



Try it!

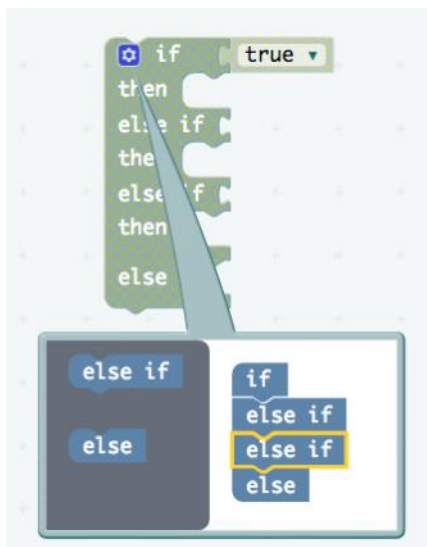
- Download your program to the micro:bit
- Press buttons A, B, and A+B together on the micro:bit

Challenge question: Can our code be made more efficient?

- Whenever you look over a program and see the same lines of code repeated, there is usually a chance to improve the code making it more efficient by reducing the number of lines of code
- What lines are repeated in our program? *If...then, pause, clear screen*
- Can we edit the code to use only one 'if...then' block, one 'pause' block, and one 'clear screen' block? Yes!

Making our code more efficient

Remind students that they can edit the 'if...then' block, adding as many 'else if' conditions as needed. They can do this by clicking on the blue gear-wheel icon in the top left corner of the 'if...then' block.



A final else

In a conditional that might receive a number of different values, it is good coding practice to have a catch-all 'else' clause. In our example, if any number value other than the ones we coded for (0,1, and 2) is received, we can signal the user that an error has occurred by using a 'show icon' block to display an X.

The pause and clear screen

Rather than repeat these lines of code 3 times, we can move the 'pause' block and the 'clear screen' block outside of the edited 'if...then...else' block.

Now our program runs as we designed it to run and is more efficient, too!

Final Morse Code Program

```
on start
  radio set group 1
  show string "Morse Code"

on button A pressed
  radio send number 0

on button B pressed
  radio send number 1

on button A+B pressed
  radio send number 2

on radio received receivedNumber
  if receivedNumber = 0
  then show leds [3-dot pattern]
  else if receivedNumber = 1
  then show leds [2-dot pattern]
  else if receivedNumber = 2
  then show leds [4-dot pattern]
  else show icon [X icon]
  pause (ms) 100
  clear screen
```

```
input.onButtonPressed(Button.A, () => {
  radio.sendNumber(0)
})
```

```

input.onButtonPressed(Button.B, () => {
    radio.sendNumber(1)
})
input.onButtonPressed(Button.AB, () => {
    radio.sendNumber(2)
})
radio.onDataPacketReceived(({ receivedNumber }) => {
    if (receivedNumber == 0) {
        basic.showLeds(`
            . . . . .
            . . # . .
            . # # # .
            . . # . .
            . . . . .
            `)
    } else if (receivedNumber == 1) {
        basic.showLeds(`
            . . . . .
            . . . . .
            . # # # .
            . . . . .
            . . . . .
            `)
    } else if (receivedNumber == 2) {
        basic.showLeds(`
            # # # # #
            # # # # #
            # # # # #
            # # # # #
            # # # # #
            `)
    } else {
        basic.showIcon(IconNames.No)
    }
    basic.pause(100)
    basic.clearScreen()
})
radio.setGroup(1)
basic.showString("Morse Code")

```

[MorseCodeFinal](#)



Project: Radio Project

For this project, students should work in pairs to design a project that incorporates radio communication to send and receive data in some way. Some projects may have two separate programs: One that receives data, and one that sends data. Students might each choose to submit one program in that case.

In other cases, a pair of students might submit one program that has both sending and receiving code in it, and the same code is uploaded to two or more micro:bits.

Project Ideas

Stop, Thief!

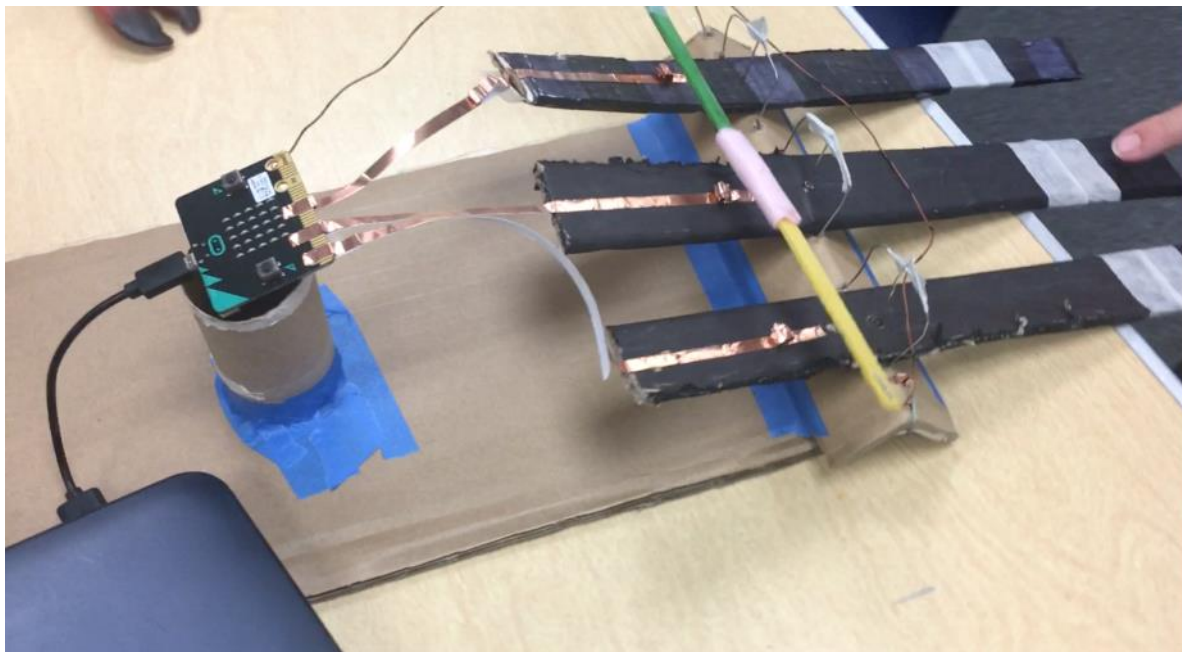
Design an alarm system for your bedroom that alerts you with a screen animation when someone opens your door. You can mount one micro:bit on your door and use the accelerometer to send a signal over the radio when it is being moved.

Interactive Art

Create a piece of interactive artwork that receives something as input over the radio from another micro:bit, and displays something based on that as output.

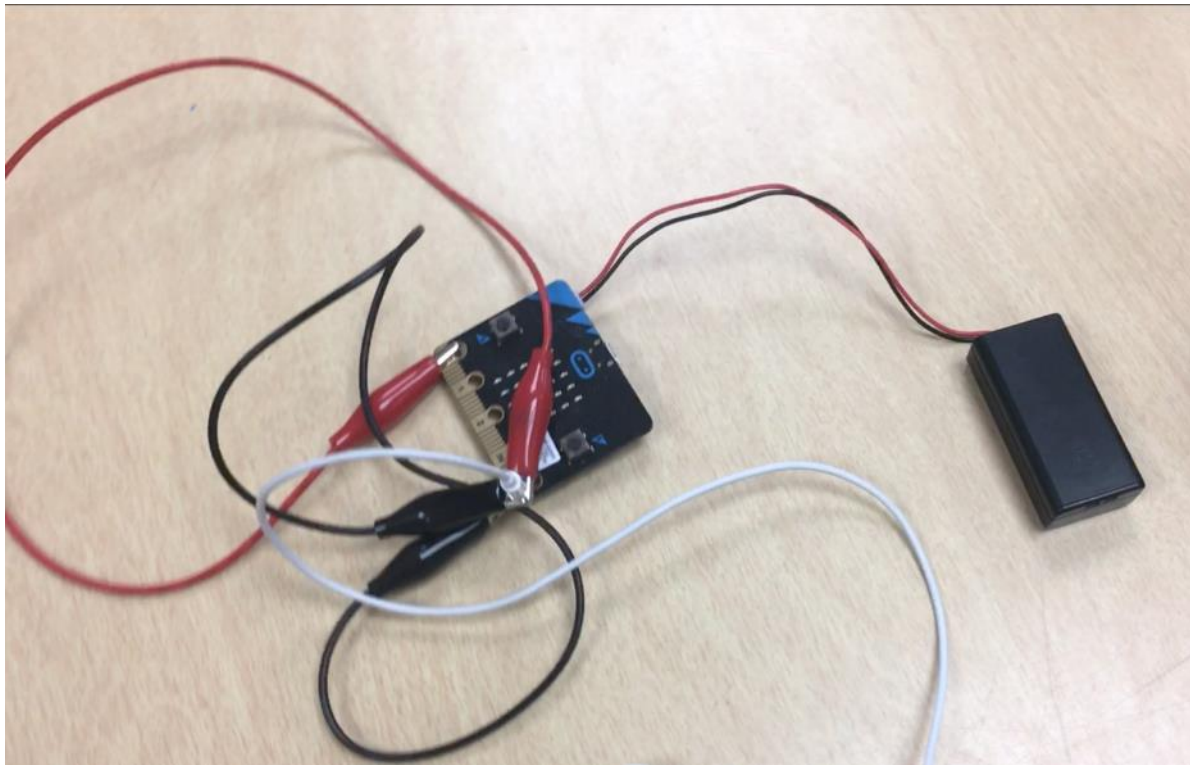
3-Note Keyboard

This is a simple three-note keyboard that uses wooden paint stirrers and copper tape to make a connection to each of the three pins on the micro:bit.



Keyboard with copper tape connections

When a key is pressed, it sends a number over the radio to a second micro:bit that plays the appropriate tone over a set of earbuds. This allows you to use each of the three pins on the first micro:bit to play a different tone.



Second micro:bit that plays the notes

3-Note Keyboard Program:

```

on start
  show leds
  clear screen

on pin P0 pressed
  set sound to 0
  radio send number sound
  show leds
  pause (ms) 500
  clear screen

on pin P1 pressed
  set sound to 1
  radio send number sound
  show leds
  pause (ms) 500
  clear screen

on pin P2 pressed
  set sound to 2
  radio send number sound
  show leds
  pause (ms) 500
  clear screen

on radio received receivedNumber
  if receivedNumber == 0
  then
    set sound to 349
    play tone sound for 1/2 beat
  else if receivedNumber == 1
  then
    set sound to 392
    play tone sound for 1/2 beat
  else if receivedNumber == 2
  then
    set sound to 440
    play tone sound for 1/2 beat
  
```

```

let sound = 0
radio.onDataPacketReceived( ({ receivedNumber }) => {
  if (receivedNumber == 0) {
    sound = 349
  }
}
  
```

```

        music.playTone(sound, music.beat(BeatFraction.Half))
    } else if (receivedNumber == 1) {
        sound = 392
        music.playTone(sound, music.beat(BeatFraction.Half))
    } else if (receivedNumber == 2) {
        sound = 440
        music.playTone(sound, music.beat(BeatFraction.Half))
    }
})
input.onPinPressed(TouchPin.P0, () => {
    sound = 0
    radio.sendNumber(sound)
    basic.showLeds(`
        . . # . .
        . # . # .
        . # . # .
        . # . # .
        . . # . .
        `)
    basic.pause(500)
    basic.clearScreen()
})
input.onPinPressed(TouchPin.P1, () => {
    sound = 1
    radio.sendNumber(sound)
    basic.showLeds(`
        . . # . .
        . # # . .
        . . # . .
        . . # . .
        . # # # .
        `)
    basic.pause(500)
    basic.clearScreen()
})
input.onPinPressed(TouchPin.P2, () => {
    sound = 2
    radio.sendNumber(sound)
    basic.showLeds(`
        . # # # .
        # . . # .
        . . # . .
        . # . . .
        # # # # .
        `)
    basic.pause(500)
    basic.clearScreen()
})
basic.showLeds(`
    # # # # #
    # # # # #
    . . . . .
    . . . . .
    . . . . .
    `)
basic.clearScreen()

```

3NoteKeyboard



Radio Tennis

In this project, the tennis racquets alternate displaying a ball on the micro:bit LED screen. When you swing the racquet, the ball disappears from one micro:bit display and shows up on the other micro:bit's display.



Radio Tennis racquets (made from cardboard)

Reflection

Have students write a reflection of about 150–300 words, addressing the following points:

- What kind of Project did you do? How did you decide what to pick?
- How does your project use radio communication?

- Are there separate programs for the Sender and the Receiver micro:bits? Or 1 program for both?
- Describe something in your project that you are proud of.
- Describe a difficult point in the process of designing this program, and explain how you resolved it.
- What feedback did your beta testers give you? How did that help you improve your design?

Assessment

	4	3	2	1
Radio	Effectively uses the Radio to send and receive data, with meaningful actions and responses for each.	Effectively uses the Radio to send or receive data, with meaningful actions and responses for each.	Use of Radio is incomplete or non-functional and/or tangential to operation of program	No working and/or meaningful use of Radio.
Micro:bit program	Micro:bit program: <ul style="list-style-type: none"> • Uses Radio blocks in a way that is integral to the program • Compiles and runs as intended • Meaningful comments in code 	Micro:bit program lacks 1 of the required elements	Micro:bit program lacks 2 of the required elements	Micro:bit program lacks all of the required elements
Collaboration reflection	Reflection piece includes: <ul style="list-style-type: none"> • Brainstorming ideas • Construction • Programming • Beta testing 	Reflection piece lacks 1 of the required elements.	Reflection piece lacks 2 of the required elements.	Reflection piece lacks 3 of the required elements.

Standards

CSTA K-12 Computer Science Standards

- CL.L2-03 Collaborate with peers, experts, and others using collaborative practices such as pair programming, working in project teams, and participating in group active learning activities.
- CL.L2-04 Exhibit dispositions necessary for collaboration: providing useful feedback, integrating feedback, understanding and accepting multiple perspectives, socialization.
- CL.L2-05 Implement problem solutions using a programming language, including: looping behavior, conditional statements, logic, expressions, variables, and functions.

Arrays

This lesson covers storing and retrieving data in an ordered fashion using Arrays. Introduces JavaScript as an alternate way of creating and modifying code. Uses a melody as a list/array of notes.

Lesson Objectives

Students will...

- Explain the steps they would take to sort a series of numbers
- Recognize three common sorting algorithms
- Learn
- Apply

Lesson Plan Structure

- Introduction: Arrays
- Unplugged Activity: Different Sorts of People
- Micro:bit Activity: Headband Charades, Starry Starry Night
- Project: Make a Musical Instrument
- Assessment: Rubric
- Standards: Listed

Introduction

Any collector of coins, fossils, or baseball cards knows that at some point you need to have a way to organize everything so you can find things. For example, a rock collector might have a tray of specimens numbered like this:



Every rock in the collection needs its own storage space, and a unique address so you can find it later.

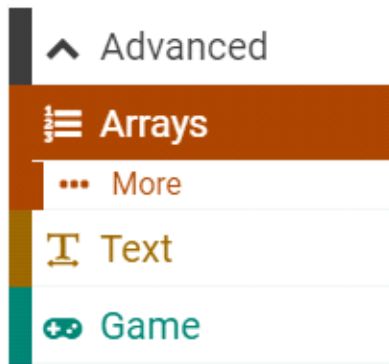
As your MakeCode programs get more and more complicated, and require more variables to keep track of things, you will want to find a way to store and organize all of your data. MakeCode provides a special category for just this purpose, called an Array.

- Arrays can store numbers, strings (words), or sprites. They can also store musical notes.
- Every spot in an array can be identified by its index, which is a number that corresponds to its location in the

array. The first slot in an array is index 0, just like our rock collection above.

- The length of an array refers to the total number of items in the array, and the index of the last element in an array is always one less than its length (because the array numbering starts at zero.)

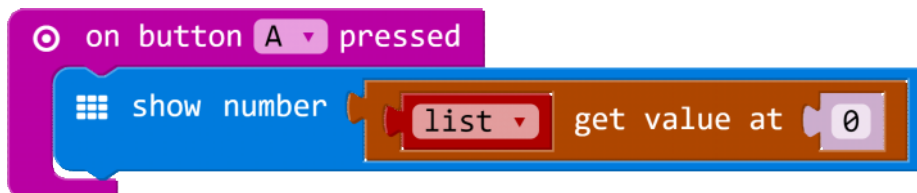
In MakeCode, you can create an array by assigning it to a variable. The Array blocks can be found under the Advanced Toolbox menu.



```
list = []  
list = [4, 2, 5, 1, 3]
```

The code above creates an empty array called list, then fills it with five numbers, indexed from 0 to 4. The index of the first value (4) is 0. The index of the second value (2) is 1. The index of the last value (3) is 4.

You can get items out of the array by specifying its index like this:



```
input.onButtonPressed(Button.A, () => {  
  basic.showNumber(list[0])  
})
```

The code above takes the first element in the array (the value at index 0) and shows it on the screen.

There are lots of other blocks in the Arrays Toolbox drawer. The next few Activities will introduce you to them.

Discussion

- Ask your students if any of them collects anything. What is it? *Comic books, cards, coins, stamps, books, etc.*

- How big is the collection?
- How is it organized?
- Are the items sorted in any way?
- How would you go about finding a particular item in the collection?

In the discussion, see if you can explore the following array vocabulary words in the context of kids' personal collections.

- Length: the total number of items in the collection
- Sort: Items in the collection are ordered by a particular attribute (e.g., date, price, name)
- Index: A unique address or location in the collection
- Type: The type of item being stored in the collection

References

Once you start saving lots of different values in an array, you will probably want to have some way to sort those values. Many languages already implement a sorting algorithm that students can call upon as needed. However, understanding how those different sorting algorithms work is an important part of computer science, and as students go on to further study they will learn other algorithms, as well as their relative efficiency.

There are some good array sorting videos:

- Visually displays a number of different types of sorts: <https://www.youtube.com/watch?v=kPRA0W1kECg>
- Bubble-sort with Hungarian folk dance: <https://youtu.be/lyZQPjUT5B4>
- Select-sort with Gypsy folk dance: <https://youtu.be/Ns4TPTC8whw>
- Insert-sort with Romanian folk dance: <https://youtu.be/ROalU379I3U>



Unplugged: Different Sorts of People

In this activity, you will demonstrate different kinds of sorting methods on your own students. This is an unplugged activity, so your students will be standing at the front of the room. If you or your students are curious to see what these different sorts look like in code, we have included a MakeCode version of each algorithm in this lesson, for you to explore if you choose.

Materials

- Sheets of paper numbered 1–10, one large printed number to a page

Set Up

- Have up to ten students (the *Sortees*) stand up at the front of the classroom. Ask another student to volunteer to be the *Sorter*.
- Mix up the order of the papers and give each student a piece of paper with a number on it. They should hold the paper facing outward so their number is visible. Each of these students is like an element in an array.

Initial Sort

- Ask the *Sorter* to place the students in order by directing them to move, one at a time, to the proper place.
- Once the students are sorted, ask students the following:
 - How did she sort you into the right order?
 - Did you see a pattern?
 - What did she do?

Try to get students to be as precise as possible in explaining their thinking. Sometimes it helps to put the steps on the board, in an algorithm:

- *First, she went to the first student, then put him in the right place.*
- *Then she went to each of the next students and put them in the right place.*

Ask for clarification when necessary: *What does it mean when you say “put them in the right place”?*

To Put Someone in the Right Place:

Bring the person to the front of the line and then compare that person’s number with the first person’s number. If it’s larger, then move that person to the right. Keep doing this as long as the person’s number is larger than the person on the right.

Some Different Types of Sorts

In computer science, there are certain common strategies, or algorithms, for sorting a collection of values. Try acting out each of these different sorts with your students.

Bubble Sort

Compare the first two students. If the student on the right is smaller than the student on the left, they should swap places. Then compare the second and third students. If the student on the right is smaller than the student on the left, they should swap places. When you reach the end, start over at the beginning again. Continue in this way until you make it through the entire row of students without swapping anybody.

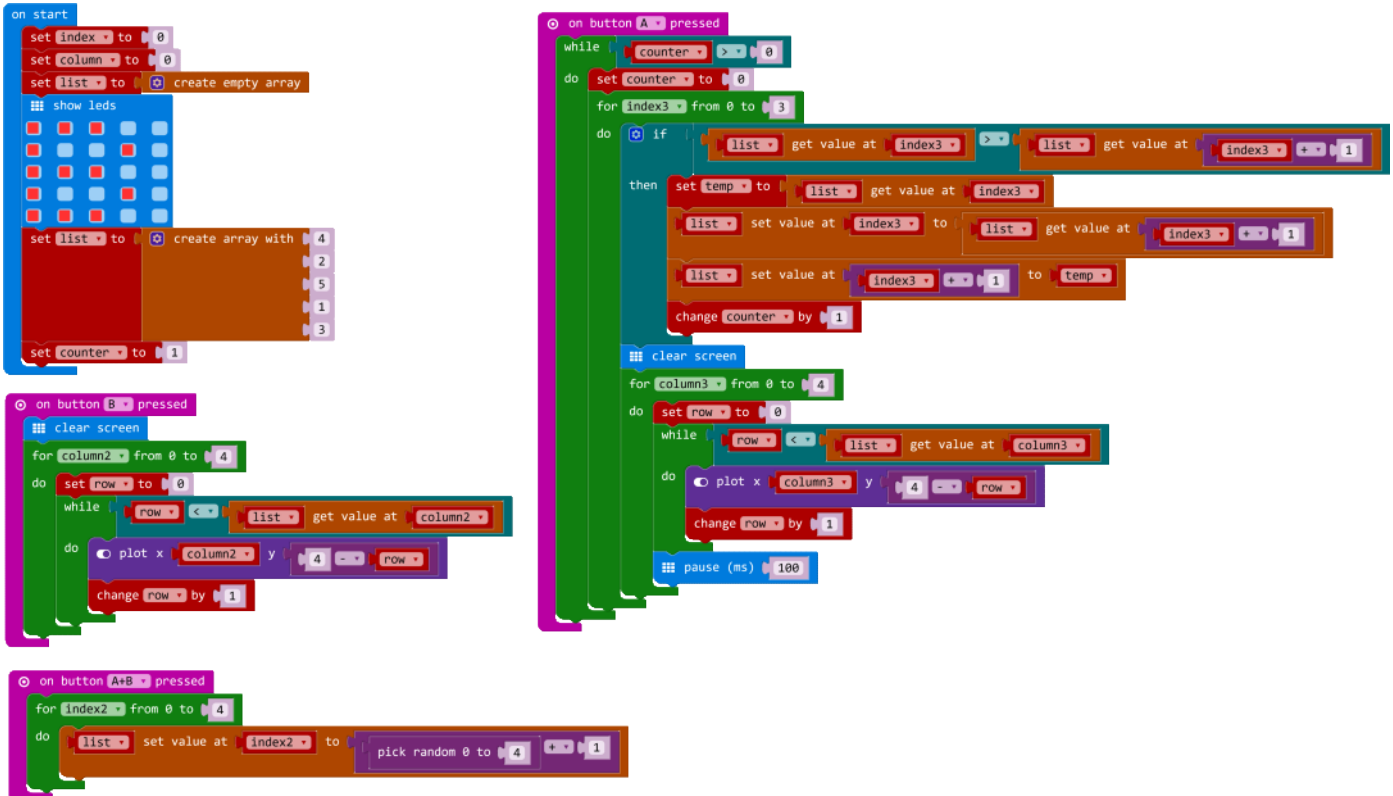
In Pseudocode

1. Create a variable called counter.
2. Set the counter to zero.

3. Go through the entire array.
4. If the value you are considering is greater than the value to its right:
 1. Swap them
 2. Add one to counter
5. Repeat steps 2 through 4 as long as counter is greater than zero.

In MakeCode

Note: Press B to display the array visually. The length of each vertical bar represents each number in the array, from left to right. Press A to sort the array using Bubble Sort. Press A + B to generate new random numbers for the array.



```

let temp = 0
let row = 0
let list: number[] = []
let counter = 0
let column = 0
let index = 0
input.onButtonPressed(Button.AB, () => {
  for (let index = 0; index <= 4; index++) {
    list[index] = Math.random(5) + 1
  }
})
input.onButtonPressed(Button.B, () => {
  basic.clearScreen()
  for (let column = 0; column <= 4; column++) {
    row = 0
    while (row < list[column]) {
      led.plot(column, 4 - row)
      row += 1
    }
  }
})

```

```

})
input.onButtonPressed(Button.A, () => {
  while (counter > 0) {
    counter = 0
    for (let index = 0; index <= 3; index++) {
      if (list[index] > list[index + 1]) {
        temp = list[index]
        list[index] = list[index + 1]
        list[index + 1] = temp
        counter += 1
      }
    }
    basic.clearScreen()
    for (let column = 0; column <= 4; column++) {
      row = 0
      while (row < list[column]) {
        led.plot(column, 4 - row)
        row += 1
      }
      basic.pause(100)
    }
  }
}
})
basic.showLeds(`
  # # # . .
  # . . # .
  # # # . .
  # . . # .
  # # # . .
  `)
list = [4, 2, 5, 1, 3]
counter = 1

```

BubbleSort



Selection Sort

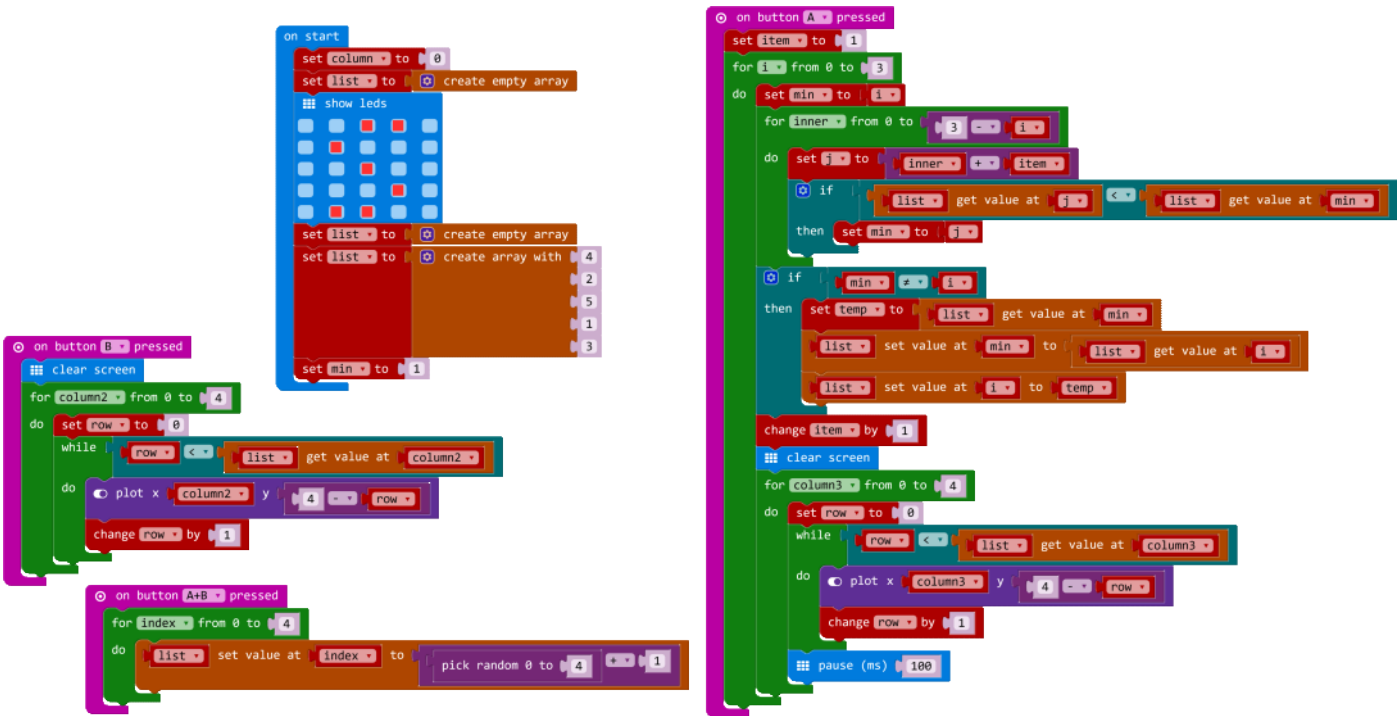
Take the first student on the left and consider that person's number the smallest number you have found so far. If the next person in line has a number that is smaller than that number, then make that person's number your new smallest number and continue in this way until you reach the end of the line of students. Then, move the person with the smallest number all the way to the left. Then start over from the second person in line. Keep going, finding the smallest number each time, and making that person the rightmost person in the sorted line of students.

In Pseudocode

1. Find the smallest unsorted value in the array.
2. Swap that value with the first unsorted value in the array.
3. Repeat steps 1 and 2 while the number of unsorted items is greater than zero.

In MakeCode

Note: The inner loop gets smaller as the sorting algorithm runs because the number of unsorted items decreases as you go. The index that the inner loop starts at needs to change as the number of sorted items increases, which is why we have to use a separate counter (item) and compute j every time through the inner loop.



```

let temp = 0
let j = 0
let min = 0
let row = 0
let list: number[] = []
let item = 0
let column = 0
input.onButtonPressed(Button.B, () => {
  basic.clearScreen()
  for (let column = 0; column <= 4; column++) {
    row = 0
    while (row < list[column]) {
      led.plot(column, 4 - row)
      row += 1
    }
  }
})
input.onButtonPressed(Button.AB, () => {
  for (let index = 0; index <= 4; index++) {
    list[index] = Math.random(5) + 1
  }
})
input.onButtonPressed(Button.A, () => {
  item = 1
  for (let i = 0; i <= 3; i++) {
    min = i
    for (let inner = 0; inner <= 3 - i; inner++) {
      j = inner + item
      if (list[j] < list[min]) {
        min = j
      }
    }
    if (min != i) {
      temp = list[min]

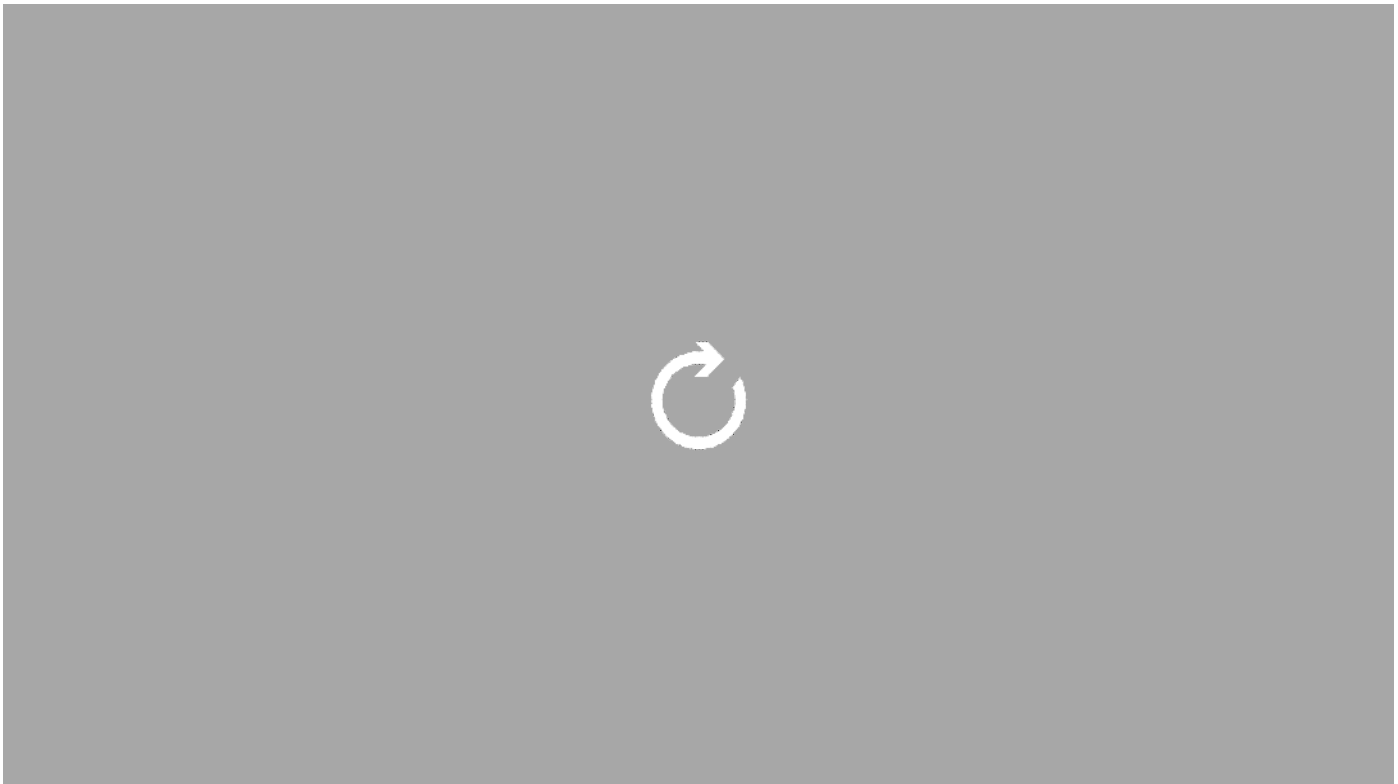
```

```

        list[min] = list[i]
        list[i] = temp
    }
    item += 1
    basic.clearScreen()
    for (let column = 0; column <= 4; column++) {
        row = 0
        while (row < list[column]) {
            led.plot(column, 4 - row)
            row += 1
        }
        basic.pause(100)
    }
}
})
basic.showLeds(`
. . # # .
. # . . .
. . # . .
. . . # .
. # # . .
`)
list = []
list = [4, 2, 5, 1, 3]
min = 1

```

[SelectionSort](#)



Insertion Sort

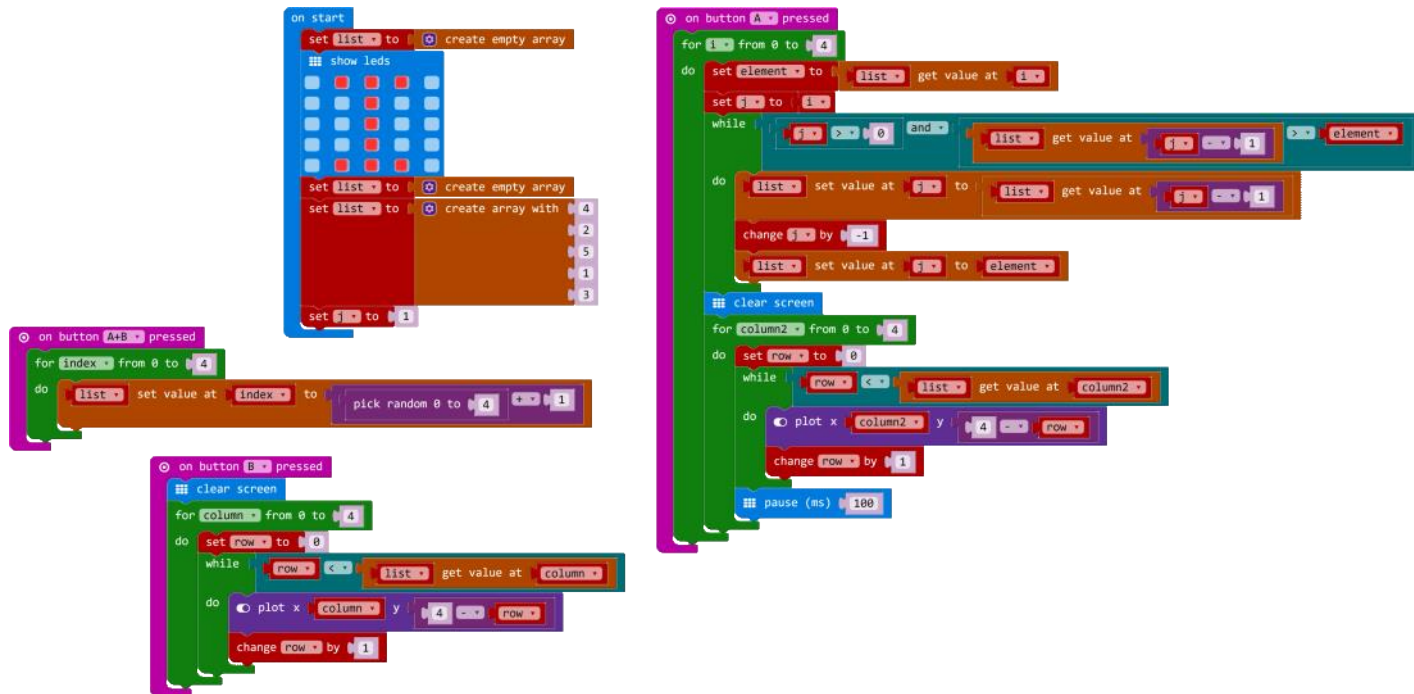
Take the first student on the left and consider that person sorted. Next, take the next student and compare him to the first student in the sorted section. If he is greater than the first student, then place him to the right of the student in the sorted section. Otherwise, place him to the left of the student in the sorted section. Continue

down the line, considering each student in turn and then moving from left to right along the students in the sorted section until you find the proper place for each student to go, shifting the other students to the right to make room.

In Pseudocode:

1. For each element in the unsorted section of the list, compare it against each element in the sorted section of the list until you find its proper place.
2. Shift the other elements in the sorted list to the right to make room.
3. Insert the element into its proper place in the sorted list.

In MakeCode



```

let j = 0
let row = 0
let element = 0
let list: number[] = []
input.onButtonPressed(Button.A, () => {
  for (let i = 0; i <= 4; i++) {
    element = list[i]
    j = i
    while (j > 0 && list[j - 1] > element) {
      list[j] = list[j - 1]
      j += -1
      list[j] = element
    }
  }
  basic.clearScreen()
  for (let column2 = 0; column2 <= 4; column2++) {
    row = 0
    while (row < list[column2]) {
      led.plot(column2, 4 - row)
      row += 1
    }
    basic.pause(100)
  }
}

```



```

    }
  })
  input.onButtonPressed(Button.AB, () => {
    for (let index = 0; index <= 4; index++) {
      list[index] = Math.random(5) + 1
    }
  })
  input.onButtonPressed(Button.B, () => {
    basic.clearScreen()
    for (let column = 0; column <= 4; column++) {
      row = 0
      while (row < list[column]) {
        led.plot(column, 4 - row)
        row += 1
      }
    }
  })
  list = []
  basic.showLeds(`
    . # # # .
    . . # . .
    . . # . .
    . . # . .
    . # # # .
  `)
  list = []
  list = [4, 2, 5, 1, 3]
  j = 1

```

InsertionSort



Sidebar

In 2008, Illinois Senator Barack Obama was interviewed by Google's CEO Eric Schmidt, who asks him a computer science interview question. Watch as the interview doesn't go exactly as planned...



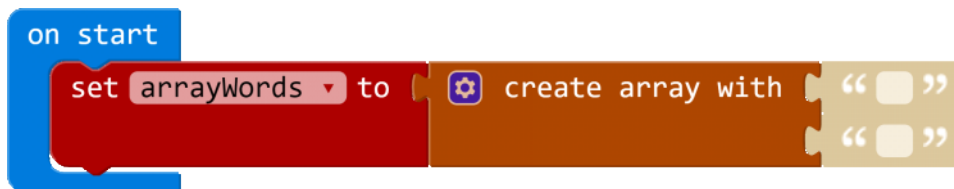
https://www.youtube.com/watch?v=k4RRi_ntQc8

Activity: Headband Charades

Create an array of words that can be used as part of a charades-type game. This activity is based on a very popular phone app invented by Ellen DeGeneres (<https://bits.blogs.nytimes.com/2013/05/03/ellen-degeneres-iphone-game/>).



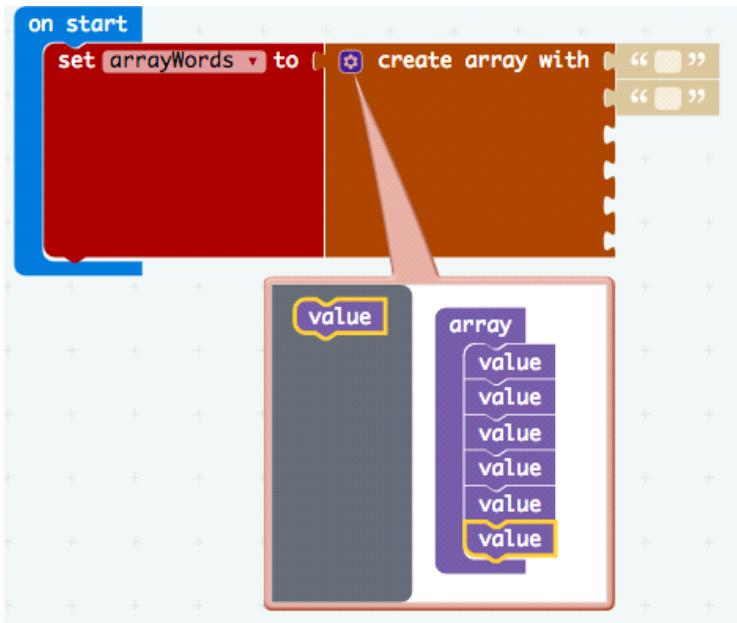
- Create a new variable and give it a name like arrayWords.
- Insert a 'set' variable block into the 'on start' block.
- Change the default variable name to this new variable name.
- From the Array Toolbox drawer, drag a 'create array' block to the coding workspace.
- Attach this array block to the end of the 'set' variable block.



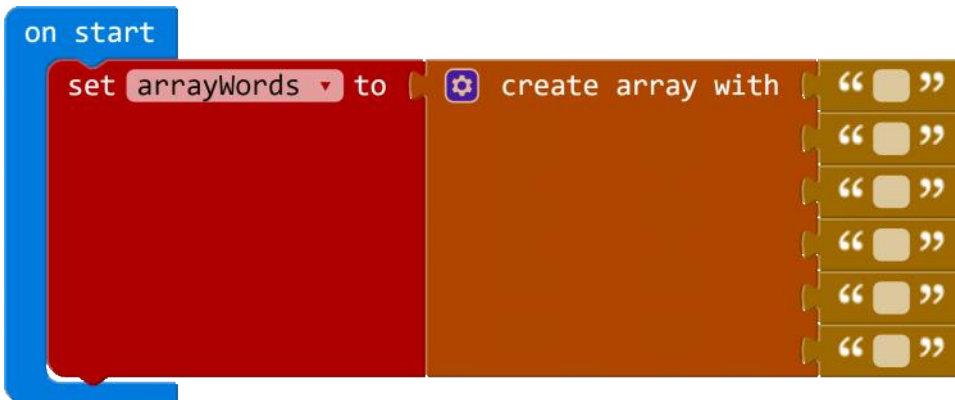
```
let arrayWords: string[] = []  
arrayWords = [ "", "" ]
```

Notice that the array comes with 2 string blocks. We'll want more for our charades game.

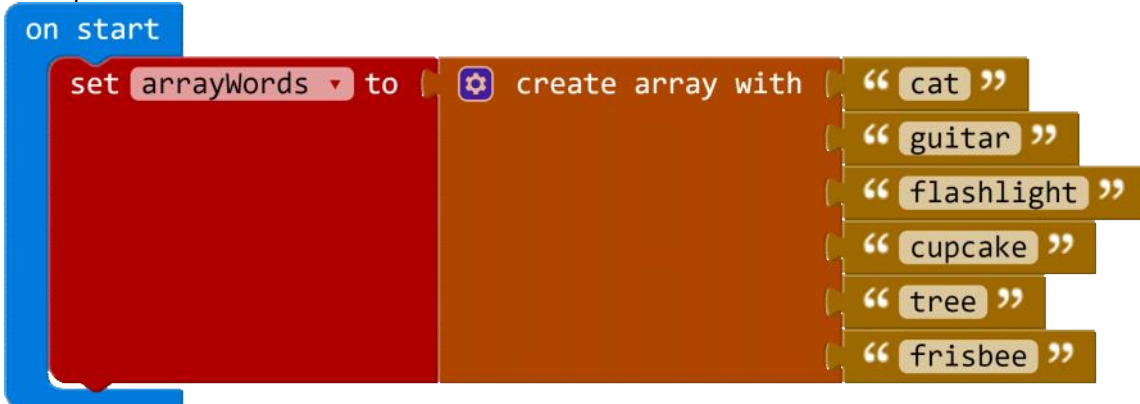
- Click on the blue gear-wheel icon in the top left corner of the 'create array' block.
- From the pop up window, add as many values (elements) as you'd like to the array block by dragging the value block from the left side of the window to the array block on the right side of the window.
- For now, we'll add 4 more values for a total of 6 values.



- Drag 4 string blocks from the Text Toolbox drawer, and place them in the empty array slots.



- Fill each string with one word. Choose words that will be fun for a game of charades.
Example:



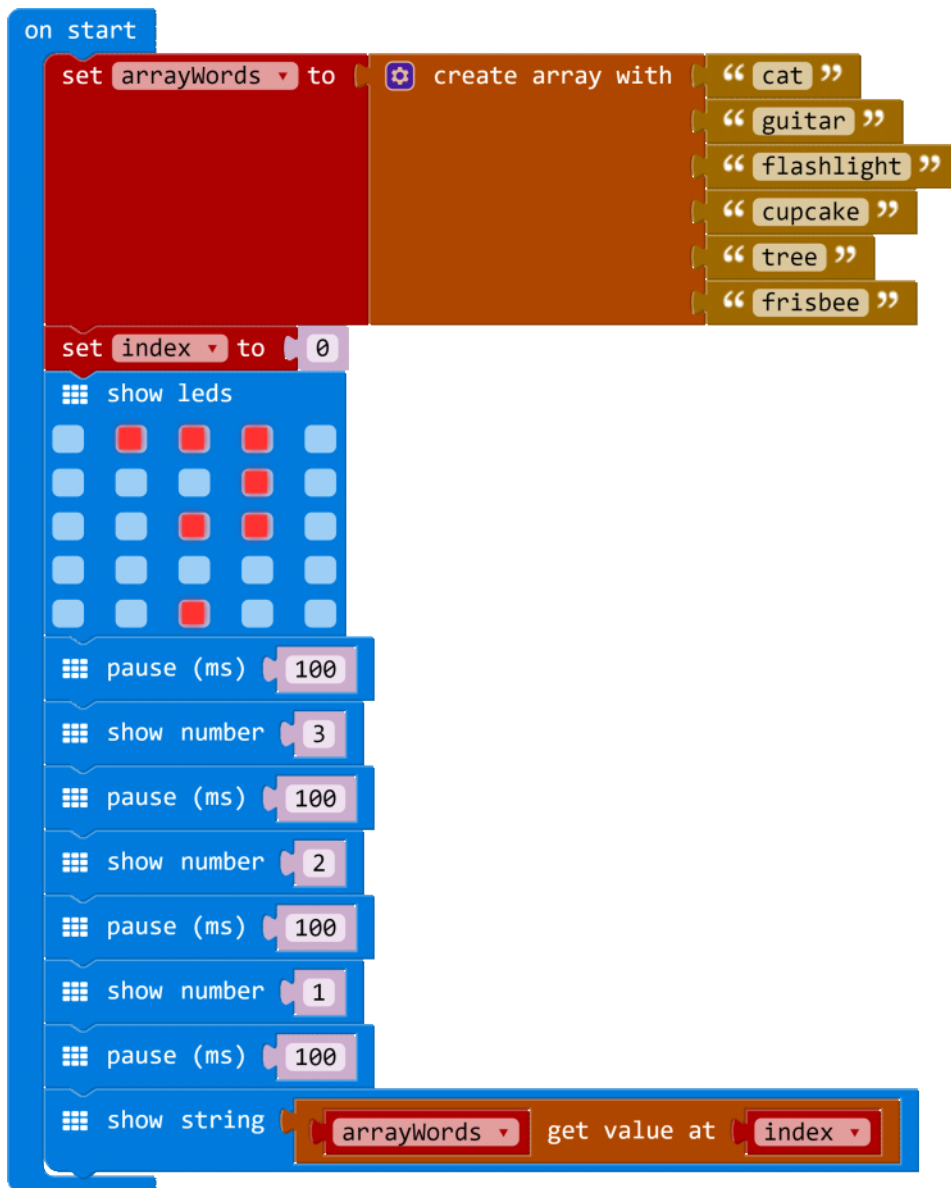
```
let arrayWords: string[] = []
arrayWords = ["cat", "guitar", "flashlight", "cupcake", "tree",
"frisbee"]
```

Now, we need a way to access one word at a time from this array of words.

- We can use the 'show string' block from the Basic Toolbox drawer, and the 'on screen up' event handler from the Input Toolbox drawer (this is a drop-down menu choice of the 'on shake' block) to tell the micro:bit to display a word when we tilt the micro:bit up.
- For this version, we'll display the words one at a time in the order they were first placed into the array.
- We'll use the index of the array to keep track of what word to display at any given time, so you'll need to create an 'index' variable.



- To start the game with the index at zero, add a 'set' variable block to the 'on' start block.
- Next, add the following:
 - an image as a placeholder for when the program has started. Since charades is a guessing game, we made one that looks like a question mark (?),
 - a countdown to the first word using show number blocks and pause blocks
 - And show the first word in the array



```

let index = 0
let arrayWords: string[] = []

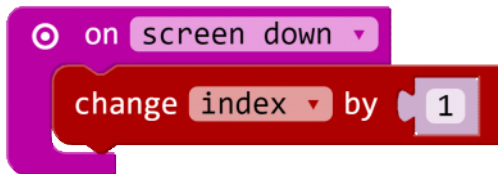
arrayWords = ["cat", "guitar", "flashlight", "cupcake", "tree",
"frisbee"]
index = 0
basic.showLeds(`
. # # # .
. . . # .
. . # # .
. . . . .
. . # . .
`)
basic.pause(100)
basic.showNumber(3)
basic.pause(100)
basic.showNumber(2)
basic.pause(100)
basic.showNumber(1)
basic.showString(arrayWords[index])

```

So far we have a start to our game and a way to display the first word.

Once that word has been guessed (or passed), we need a way to advance to the next word in the array.

- We can do this by changing the index of the array with the 'on screen down' event handler from the Input Toolbox drawer (this is a drop-down menu choice of the 'on shake' block) to advance to the next word when we tilt the micro:bit down



```
input.onGesture(Gesture.ScreenDown, () => {  
  index += 1  
})
```

We have a limited number of elements in our array, so to avoid an error, we need to check and make sure we are not already at the end of the array before we change the index.

- Under the Arrays Toolbox drawer, drag out a 'length of' block. The 'length of' block returns the number of items (elements) in an array. For our array, the length of block will return the value 6.
- But because computer programmers start counting at zero, the index of the final (6th) element is 5.

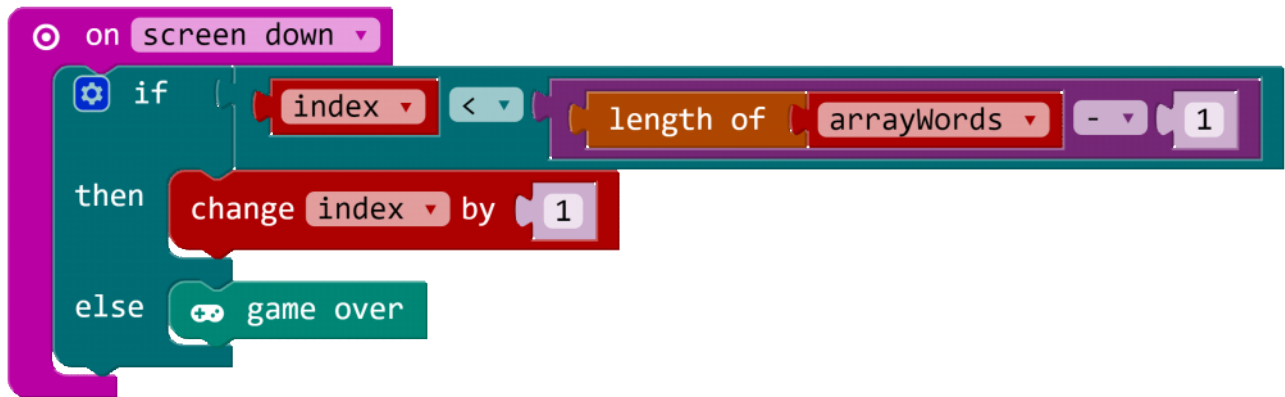
Some pseudocode for our algorithm logic:

- When the player places the micro:bit screen down
 - Check the current value of the index.
 - If the current value of the index is less than the length of the array minus one*,
 - Then change the value of the index by one,
 - Else indicate that it is the end of the game.

*Notes:

- Our array has a length 6, so this will mean that as long as the current value of the index is less than 5, we will change the array by one.
- Using 'less than the length of the array minus one' instead of the actual numbers for our array makes this code more flexible and easier to maintain. We can easily add more elements to our array and not have to worry about changing numbers elsewhere in the code.

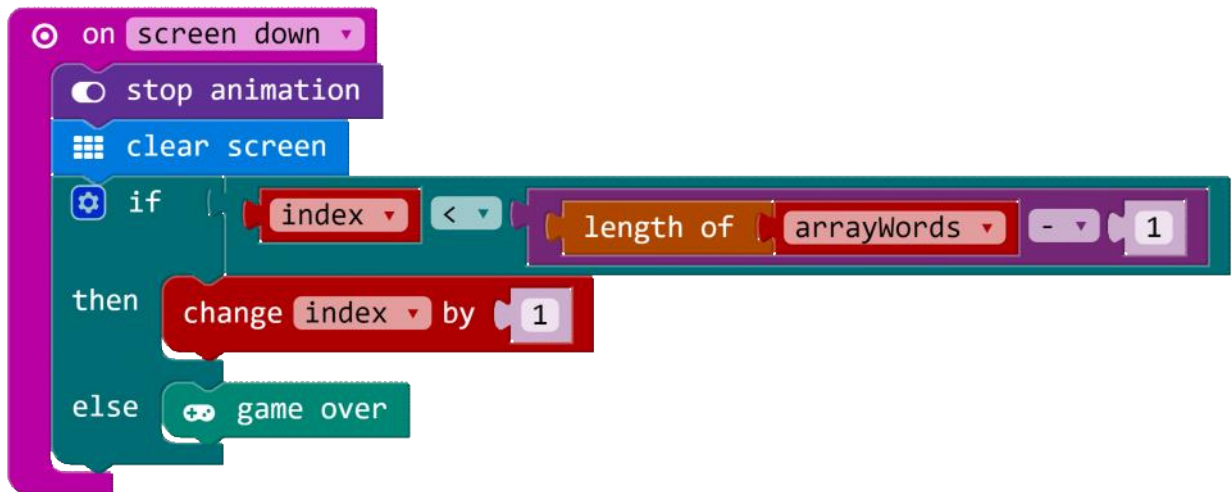
We can put this all together with an 'if...then...else' block and a 'less than' comparison block from the Logic Toolbox drawer, a subtraction block from the Math Toolbox drawer, and a 'game over' block from the Game Toolbox drawer (located under the Advanced menu).



```
input.onGesture(Gesture.ScreenDown, () => {
  if (index < arrayWords.length - 1) {
    index += 1
  } else {
    game.gameOver()
  }
})
```

To make our game more polished, we'll add 2 more blocks for smoother game play.

- In case a word is already scrolling on the screen when a player places the micro:bit screen down, we can stop this animation and clear the screen for the next word by using a 'stop animation' block from the Led More Toolbox drawer, and a 'clear screen' block from the Basic More Toolbox drawer.



```
input.onGesture(Gesture.ScreenDown, () => {
  led.stopAnimation()
  basic.clearScreen()
  if (index < arrayWords.length - 1) {
    index += 1
  } else {
    game.gameOver()
  }
})
```

Game Play

There are different ways you can play charades with our program. Here is one way you can play

with a group of friends.

- With the micro:bit on and held so Player A cannot see the screen, another player starts the program to see the first word.
- The other players act out this word charades-style for Player A to guess.
- When Player A guesses correctly or decides to pass on this word, a player places the micro:bit screen down.
- When ready for the next word, a player turns the micro:bit screen up. Play continues until all the words in the array have been used.

Mod this!

- Add a headband to hold the micro:bit on the Players' foreheads (using cardboard, paper, rubber bands, etc.)
- Add a way to keep score
- Keep track of the number of correct guesses and passes
- Add a time limit

Headband Charades Complete Program (simple version - no time limit or scoring)

```
on start
  set arrayWords to create array with
    "cat"
    "guitar"
    "flashlight"
    "cupcake"
    "tree"
    "frisbee"
  set index to 0
  show leds
  pause (ms) 100
  show number 3
  pause (ms) 100
  show number 2
  pause (ms) 100
  show number 1
  pause (ms) 100
  show string arrayWords get value at index

on screen up
  show string arrayWords get value at index

on screen down
  stop animation
  clear screen
  if index < length of arrayWords - 1
  then
    change index by 1
  else
    game over
```

let index = 0

```

let arrayWords: string[] = []
input.onGesture(Gesture.ScreenUp, () => {
    basic.showString(arrayWords[index])
})
input.onGesture(Gesture.ScreenDown, () => {
    led.stopAnimation()
    basic.clearScreen()
    if (index < arrayWords.length - 1) {
        index += 1
    } else {
        game.gameOver()
    }
})
arrayWords = ["cat", "guitar", "flashlight", "cupcake", "tree", "frisbee"]
index = 0
basic.showLeds(`
. # # # .
. . . # .
. . # # .
. . . . .
. . # . .
`)
basic.pause(100)
basic.showNumber(3)
basic.pause(100)
basic.showNumber(2)
basic.pause(100)
basic.showNumber(1)
basic.showString(arrayWords[index])

```

Charades



Project: Musical Instrument

This is a project in which students are challenged to create a musical instrument that uses arrays to store sequences of notes. The array of notes can be played when an input occurs, such as one of the buttons being pressed, or if one or more of the pins is activated.

Ideally, the micro:bit should be mounted in some kind of housing, perhaps a guitar shape or a music box. Start by looking at different kinds of musical instruments to get a sense of what kind of shape you might want to build around your micro:bit.

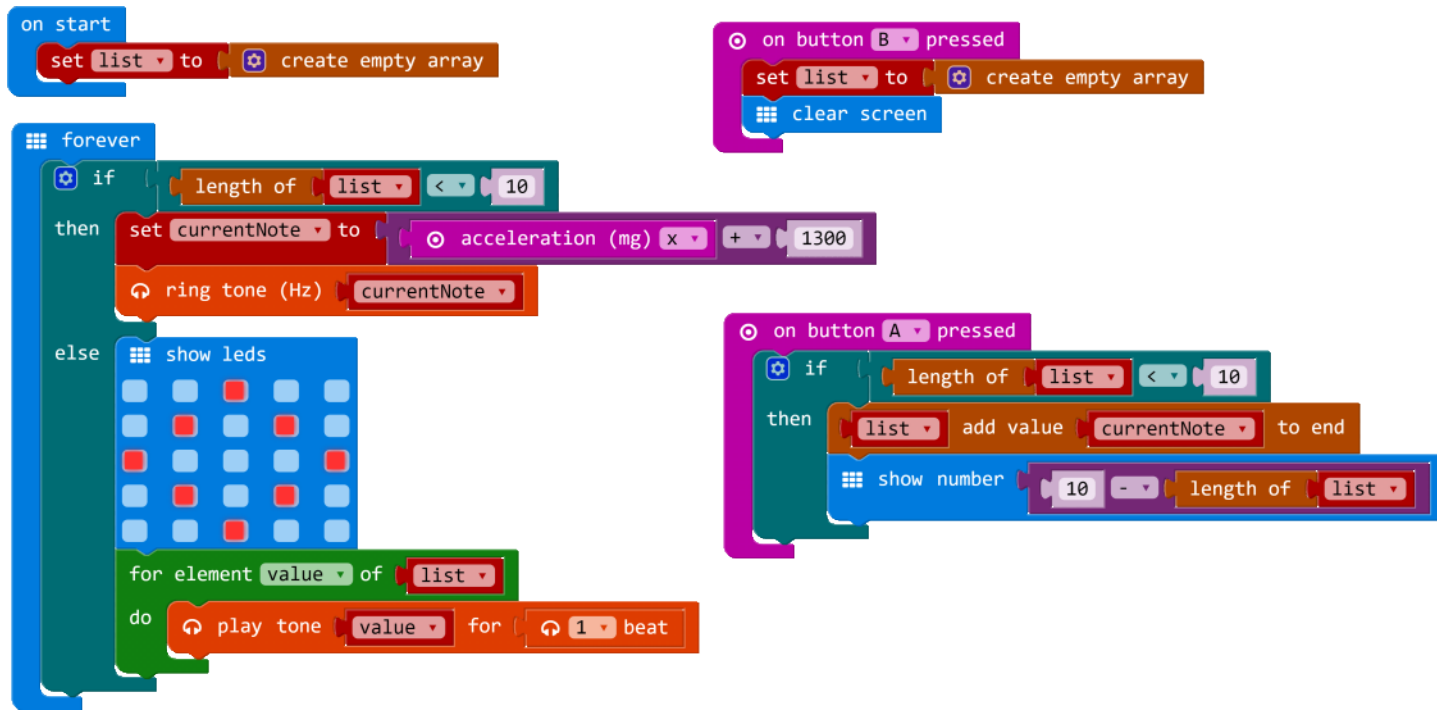


Here are some examples of guitars that were made out of cardboard and colored, patterned duct tape that you can buy in craft stores.

Example Guitar Code

This is an example of a project that uses the micro:bit accelerometer to play different tones when the guitar is held and tilted while playing. Pressing the A button will save the current tone to an array. After ten tones, a repeating melody will be performed. Press the B button to clear the array and start over.

Song-Maker



```

let currentNote = 0
let list: number[] = []
basic.forever(() => {
  if (list.length < 10) {
    currentNote = input.acceleration(Dimension.X) + 1300
    music.ringTone(currentNote)
  } else {
    basic.showLeds(`
      . . # . .
      . # . # .
      # . . . #
      . # . # .
      . . # . .
    `)
    for (let value of list) {
      music.playTone(value, music.beat(BeatFraction.Whole))
    }
  }
})
input.onButtonPressed(Button.A, () => {
  if (list.length < 10) {
    list.push(currentNote)
    basic.showNumber(10 - list.length)
  }
})
input.onButtonPressed(Button.B, () => {
  list = []
  basic.clearScreen()
})

```

[SongMaker](#)



Using Arrays with Musical Notes

You can create an array of notes by attaching Music blocks to an array. Musical notes are described in words (e.g., Middle C, High C) but they are actually numbers. You can do Math operations on those numbers to change the pitch of your song.

Here is an example of how to create an array with musical notes. Button A plays every note in the array. Button B plays the notes at twice the frequency (but doesn't alter the original notes.)

The image shows Scratch code blocks for creating and playing musical notes. It starts with an 'on start' block containing a 'set list to' block followed by a 'create array with' block. The 'create array with' block contains five 'Middle C' notes. Below this are two 'on button pressed' blocks. The first, 'on button A pressed', contains a 'for element value of list' loop with a 'do' block containing a 'play tone value for 1 beat' block and a 'pause (ms) 1000' block. The second, 'on button B pressed', contains a similar 'for element value of list' loop with a 'do' block containing a 'play tone value x 2 for 1 beat' block and a 'pause (ms) 1000' block.

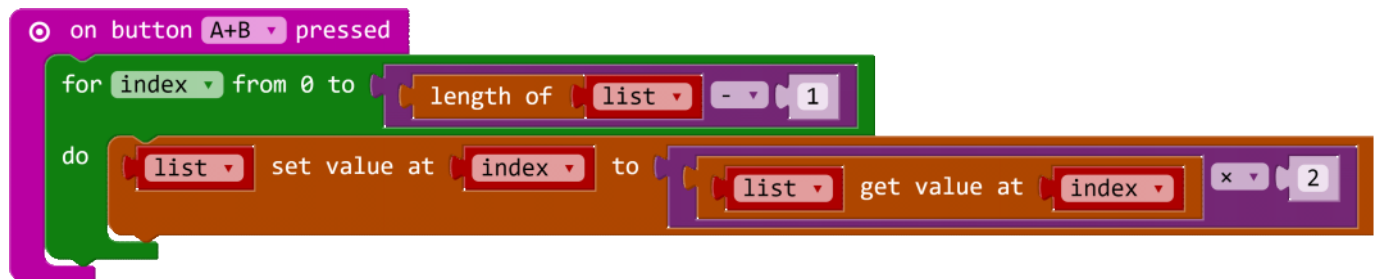
```
let list: number[] = []
let value = 0
input.onButtonPressed(Button.A, () => {
  for (let value of list) {
    music.playTone(value, music.beat(BeatFraction.Whole))
  }
  basic.pause(1000)
})
```

```

})
input.onButtonPressed(Button.B, () => {
  for (let value of list) {
    music.playTone(value * 2, music.beat(BeatFraction.Whole))
  }
  basic.pause(1000)
})
list = [262, 392, 330, 392, 262]

```

Remember that a 'for element value of list' loop makes a temporary copy of the value, so even if you change a value, it will not change the original element in the array. If students want to permanently change the values in their array (transpose music to increasingly higher keys, for example) they can use a for loop like this:

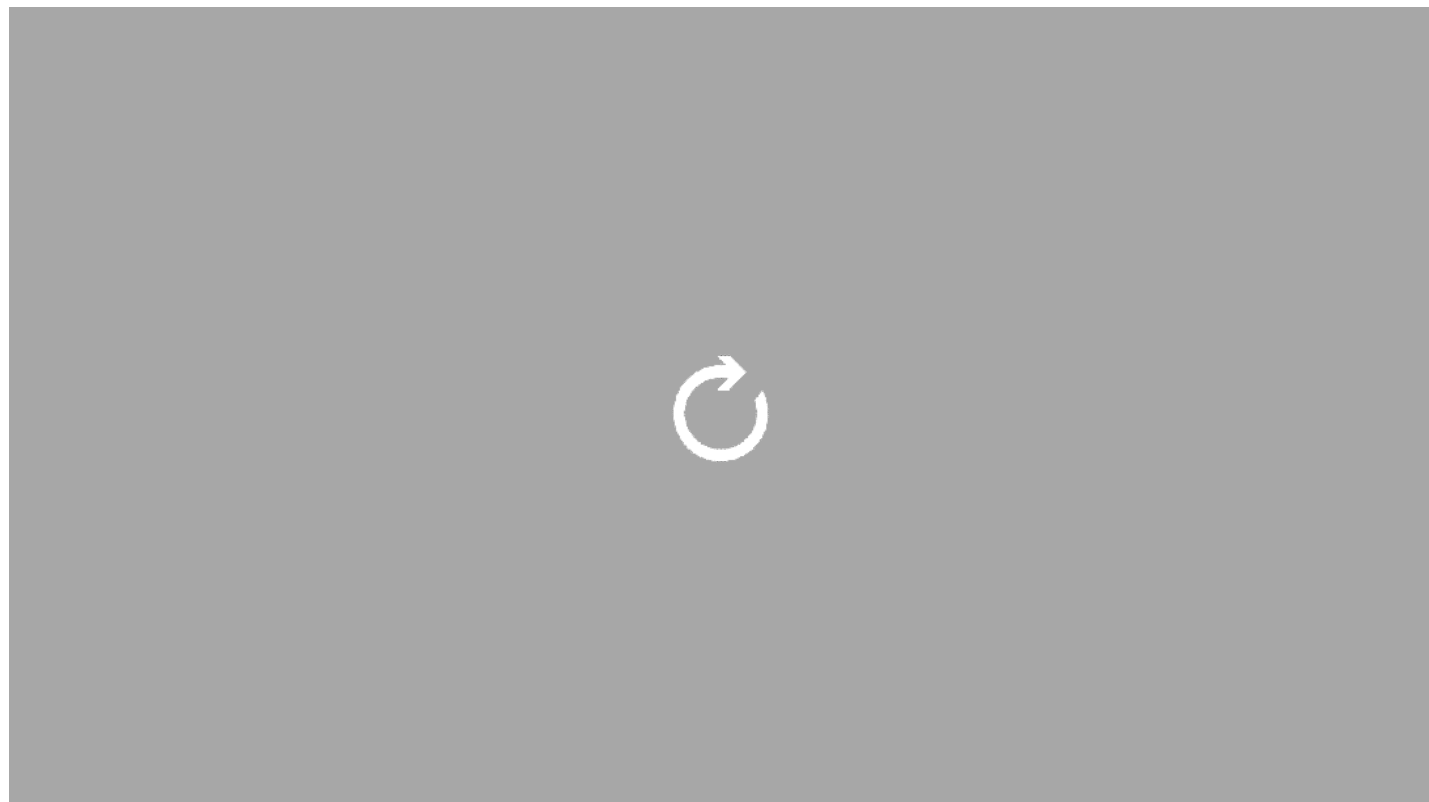


```

let list: number[] = []
input.onButtonPressed(Button.AB, () => {
  for (let index = 0; index <= list.length - 1; index++) {
    list[index] = list[index] * 2
  }
})

```

[MusicArray](#)



Reflection

Have students write a reflection of about 150–300 words, addressing the following points:

- Explain how you decided on your musical instrument. What brainstorming ideas did you come up with?
- What properties does it share with a real musical instrument? What is unique?
- Describe the type of array you used (*Numbers, Strings, or Notes*) and how it functions in your project.
- What was something that was surprising to you about the process of creating this program?
- Describe a difficult point in the process of designing this program, and explain how you resolved it.
- What feedback did your beta testers give you? How did that help you improve your musical instrument?

Assessment:

	4	3	2	1
Array	Stores and iterates through each element of the array successfully	Stores each element of the array successfully	Array skips values or has other problems with storing and/or retrieving elements	Array doesn't work at all or no array present
Micro:bit program	<ul style="list-style-type: none"> • Uses at least one array in a fully integrated and meaningful way • Compiles and runs as intended • Meaningful comments in code 	Uses an array in a tangential way that is peripheral to function of project and/or program lacks 1 of the required elements	Array is poorly implemented and/or peripheral to function of project, and/or lacks 2 of the required elements	Micro:bit program lacks 3 or more of the required elements
Collaboration reflection	Reflection piece includes: <ul style="list-style-type: none"> • Brainstorming ideas • Construction • Programming • Beta testing 	Reflection piece lacks 1 of the required elements.	Reflection piece lacks 2 of the required elements.	Reflection piece lacks 3 of the required elements.

Standards

CSTA K-12 Computer Science Standards

- CT.L1:6-02 Develop a simple understanding of an algorithm using computer-free exercise
- CPP.L1:6-05 Construct a program as a set of step-by-step instructions to be acted out
- 2-A-2-1 Solicit and integrate peer feedback as appropriate to develop or refine a program
- 2-A-6-10 Use an iterative design process (e.g., define the problem, generate ideas, build, test, and improve solutions) to solve problems, both independently and collaboratively.
- CT.L3B-06 Compare and contrast simple data structures and their uses (e.g., arrays and lists).
- CL.L2-05 Implement problem solutions using a programming language, including: looping behavior, conditional statements, logic, expressions, variables, and functions.

Introduction

In this unit, we will be reviewing the concepts we covered in the previous weeks, and providing some ideas for an independent final project that students can focus on in the next several weeks. We will also provide a rubric for keeping students on task and tracking the learning that they are doing as they work on their projects. This is an expanded version of the process students followed in the Mini-Project, in Lesson 6.

Students are asked to create an independent project that demonstrates the use of something they have already learned, something they went out and researched for themselves, something they borrowed from somewhere else (with citations) and something completely original. They are also asked to document their learning process throughout the next couple of weeks using an independent project framework that emphasizes metacognitive development and process-oriented work.



Review



Here is a brief review of the topics we covered in lessons 7–12

Coordinate Grid and LEDs

The micro:bit's 25 LEDs are arranged in a 5x5 grid, with the origin at the top left. Values for both the x and y axes start at zero and increase as you move down and to the right. Individual LEDs can be turned off and on by specifying a pair of coordinates. The current value of an LED can be checked, and its brightness can be changed, as well.

Booleans

A Boolean is a data type that only has two possible values: True or False. You can use boolean variables to keep track of the state of a game (gameOver is either true or false) or check to see whether a certain action has taken place yet (messageSent is either true or false). Boolean operators such as AND OR and NOT allow you to combine boolean expressions to make more complex conditions.

Bits, Bytes, and Binary

Computers work with base-2, which uses binary numbers. Binary numbers only have two possible values: 0 or 1.

Radio Communication

Micro:bits can send a combination of strings and numbers using the Radio blocks. The Infection activity is an example of a thought-provoking group simulation that uses the Radio to send and receive data between micro:bits.

Arrays

Arrays in MakeCode are used to store and retrieve numbers, strings, musical notes, or sprites. Everything in a particular array needs to be the same data type and elements in an array are numbered starting from zero, also called the index. Objects can be accessed, changed, added to, or removed from an array using their index. Three common methods of sorting elements in an array are bubble sort, selection sort, and insertion sort.

Final Project

The final project is a chance for you to use all of the skills you have been learning throughout the semester to create something that is original, and that solves a problem or serves a purpose.

Possible ideas

- Create a game
- Create something that helps somebody by solving a problem
- Create something beautiful
- Create a musical instrument

In addition, your project code must do each of the following things:

Show something you already know

You should demonstrate your knowledge of one or more concepts we have covered in these lessons.

Show something new

You should demonstrate a technique, efficiency, or block that you went out and learned how to do on your own, either from the documentation, or from another classmate.

Incorporate a maker component

You should not create a project that exists solely and independently on the micro:bit. Your project should work together with tangible components such as servos, real buttons, switches, to do something unique.

Timeframe

Three weeks of in-class work and activities

Due each week:

- 2–3 work logs
- 1 Record of Thinking

Due in three weeks:

- Beta testing period
- Final Narrative
- Final Project Code
- Final project showcase and celebration at the end

Assessment:

- 50% Process (initial proposal, work logs, records of thinking, final narrative)
- 50% Product (project code and maker component)

Teacher Note: This form of assessment places just as much weight on documenting the process of designing the project, as it does on the finished product itself. This is because in my classroom I want to prioritize "sustained effort over an extended period of time" over a project that might have resulted from three all-nighters in the final week it is due.

However, you may decide to assign more or less weight to each of these pieces, and you should certainly feel free to scale up or down the documentation piece as appropriate for your classroom, grade level, and teaching priorities.

While Working on the Project

The expectation is that you are working steadily on your independent project for three weeks, testing out ideas, trying things out, getting stuck, and getting yourself unstuck. Because everyone is working on a different project, we can't assign the same homework to everybody so besides the project work itself, you are also responsible for documenting the work you are doing on the project using work logs, and reflecting on the process of your learning in a record of thinking. Here are more details on these.

Work Logs

A work log is a short, bullet point list of what you worked on, and how long it took. Stick to just the facts. It shouldn't take more than thirty seconds or so to write up a work log. Students should do one for every class, several times a week. A shared Microsoft OneNote notebook is a great way to keep a work log that students can update regularly. Alternately, you might use a collaborative shared document, or your classroom management system, or even e-mail.

Sample Work Log:

Flappy Dino Project

3/31: 45 min. Worked on attaching cardboard arm to servo and mounting servo to inside

4/1: 30 min. Looked up documentation on talking to NeoPixel strip, worked through demos

4/3: 45 min. Hot glued NeoPixel strip to outside of dinosaur, finished painting

4/4: 30 min. Coded lighting patterns in pxt.

Teacher Note: We generally don't accept late work logs. If a student simply didn't have time to do any work on the project, he should still file a work log, and report that no work got done. Work logs are worth a few points each, so missing one or two isn't a problem, but if it happens a lot it's usually time to do a check-in with that student and see where she is with the project.

Record of Thinking

A Record of Thinking is like a journal entry (or like the reflection that you did for the mini-project) that tells the story of your learning throughout the past week. Go through your work logs for the week and look at what you did, where you got stuck, and how you figured it out.

Then write a 150- to 300-word Record of Thinking addressing the following:

- Describe something that surprised you this week as you worked on your project.
- Describe a moment where you got stuck. How did you get unstuck?
- Did anyone help you this week? Who and how?
- Choose an adjective that describes how you are feeling about your project this week. Explain why you chose this word.
- What are you working on next week? (for weeks 1 and 2)
- If you had more time to work on this project, what would you add? (for week 3)

Sample Record of Thinking Excerpt:

Week of April 6

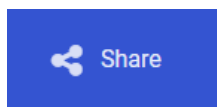
I guess I would choose the word "elated" because that's what I am feeling right now. After Mr. Kiang helped me figure out why my code wasn't working I was able to see it working exactly how I pictured it last week! That was a great moment. I was surprised how hard writing code that works is. I planned out the steps I wanted it to do but I didn't realize that the loops had to be nested one inside the other so I was stuck for a while. It always seems more simple than it is, that's one thought I will take into next week. Now I have the head attached to the body and the jaws work. I'm going to keep trying to get the lights working.

Teacher Note: A Record of Thinking is not an expanded work log! Students will sometimes just write a more detailed list of all of the tasks they completed over the week, and that's not the point of the Record of Thinking. The Work Logs are to show WHAT you did. The Record of Thinking is to show HOW you learned how to do it. Unlike Work Logs, I will accept late Records of Thinking as long as they come no later than the due date for the next week's Record of Thinking. It is an important form of documentation of the learning process.

Turning in the Final Project

When you turn in the final project, you should turn in your code, and a final narrative.

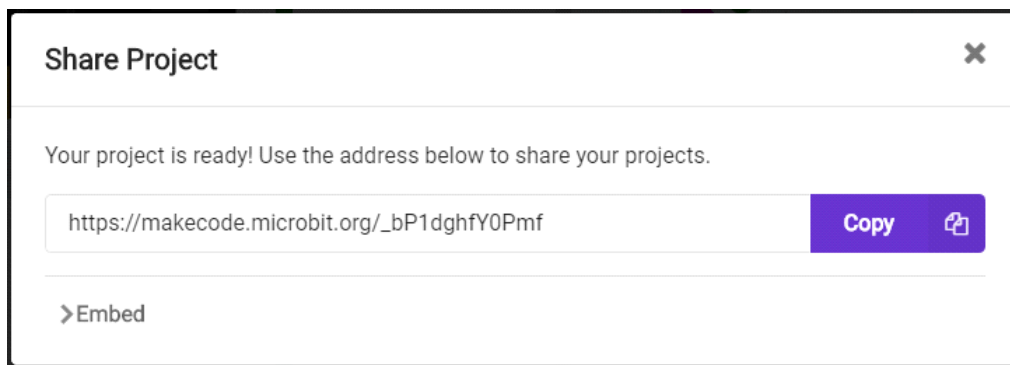
To turn in your code, you can Share the code by clicking the Share button at the top of the MakeCode window (next to Projects).



You acknowledge that you have consented to sharing your code by clicking Publish project.



You can then copy the URL, paste it into a OneNote page, or send it to your teacher.



You also need to create a written final narrative to accompany your code.

You have worked for the past three weeks to propose, design, and test an original micro:bit independent project. I am looking for an honest, accurate assessment of your work over this time.

Please go back and read through all of your Work Logs, Records of Thinking, Beta Testing feedback, and any notes from teacher conferences.

Then, compose a comprehensive narrative that tells the story of the development of this app, and your progress toward your goals along the way. How you tell the story is up to you, but you might consider following most, if not all, of the following questions:

- How did you start the process of designing the product/meeting your goals?
- What did you hope to learn?
- What challenges did you face? How did you overcome them?
- What was the outcome?
- What did you learn in the end?
- Who in the class provided help to you along the way? How?
- What were you proud of?
- What would you do differently next time?

Throughout your narrative, you must cite evidence from your work logs and records of thinking (e.g., Record of Thinking 4/17, Work Log #3, Conference notes, etc.) You may use footnotes for this or add it in parentheses after the material you are citing.

I will read this carefully and grade it along with your final project code and average it with the total of your work logs and records of thinking to come up with your final grade for the project.

Sample Final Narrative:

It's clear to me now that in the second week, I was a little lost and confused with the direction my project was taking. I can see now that in my chats with Mr. Kiang and with classmates (Conference Notes 4/3) I was not being very precise in my questions, and I didn't totally understand what he was explaining.

Looking back, one of my goals was to meet more regularly with my table mates. Hopefully I would be a lot less confused and at least this time when I got stuck, we would be able to solve it together. I wrote about this in my Record of Thinking (Record of Thinking #2) but I am surprised that things cleared up for me so quickly once we did start meeting together. This allowed me to get past something that was really bothering me, specifically adding and removing things from an array, and I was able to complete that in less than a day after having been stuck for more than a week (Work Log #4).

Once I started to get a little more clear on what to do, I was able to get more effective help from my classmates. Specifically, Jordan helped me a lot with figuring out how to get an image to display properly on the screen. He also showed me how to search through the online documentation more effectively. I think if I could do this over again, I would have scheduled more time earlier to meet with Mr. Kiang and/or found a better way to share the different online sites with my table mates because we all found different places to go. I didn't even find out until the end that you could jump into JavaScript to make changes to the code, and it makes it all with the right blocks when you go back! (Beta Testing notes) That would have saved me a lot of time.

Beta Testing

Beta testing is an important part of testing the final projects to uncover bugs or design issues that could make the projects difficult to use. One way to test the projects is to ask all students to come in to class on a specific day with the projects ready to test. This is not the final deadline, but projects should be "feature-complete" i.e., all features need to be incorporated into the micro:bit, and the construction of the real world elements of the project need to be done or almost done.

Students can take turns presenting their projects to the entire class, or they can work in pairs to take turns trying their partner's project out and offering feedback. Students who are being critiqued should take beta testing feedback notes and turn them in as part of their final project narrative.

Final Showcase

Have a celebration of your students' hard work and hold an event at your school for parents, administrators, and other community members to appreciate all of the hard work that went into making each of the final projects.

We have found that a "science fair" format works nicely, with students sitting at tables where they can demonstrate their projects and answer questions. Some schools do a "shark tank" type of event where students take turns "pitching" their project ideas to a panel composed of local software developers, entrepreneurs, and investors. Either way, a little public recognition of all of your students' hard work goes a long way!

Assessment

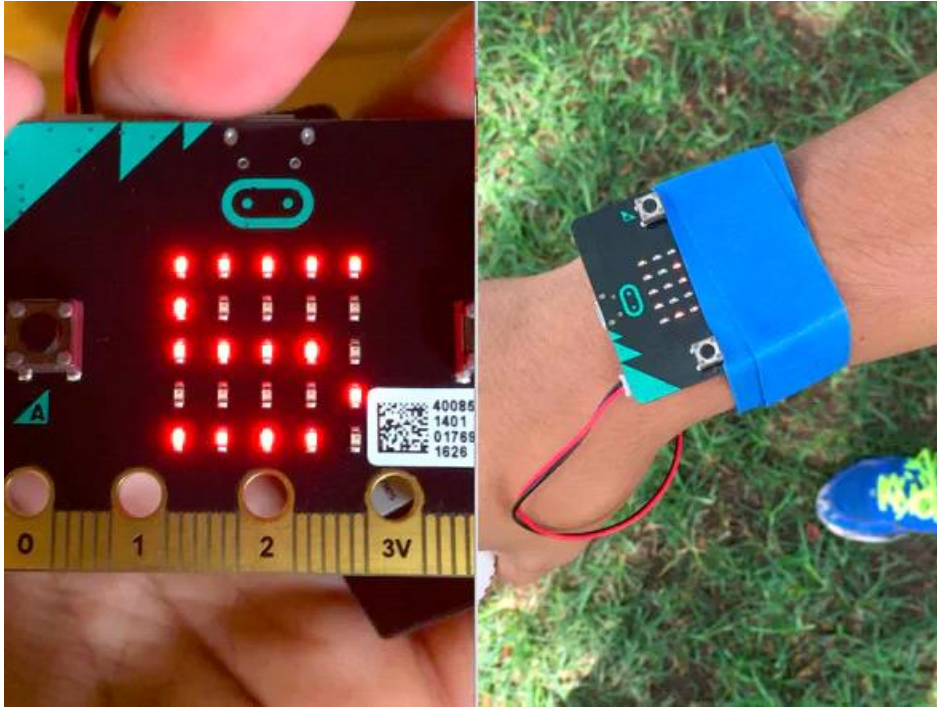
	4	3	2	1
Code - Show what you know	Code very effectively demonstrates the use of previous concept(s). All variable names are unique and clearly describe what information values the variables hold. Code is highly efficient.	Code effectively demonstrates the use of previous concept(s). Most variable names are unique and/or clearly describe what information values the variables hold. Code is mostly efficient.	Code somewhat effectively demonstrates the use of previous concept(s). Only some variable names are unique and/or clearly describe what information values the variables hold. Code is somewhat efficient.	Code demonstrates the use of previous concept(s), yet is not effective. Few or no variable names are unique and/or clearly describe what information values the variables hold. Code is not efficient.
Code - Show something new	Code very effectively demonstrates the use of new concept(s). All variable names are unique and clearly describe what information values the variables hold. Code is highly efficient.	Code effectively demonstrates the use of new concept(s). Most variable names are unique and/or clearly describe what information values the variables hold. Code is mostly efficient.	Code somewhat effectively demonstrates the use of new concept(s). Only some variable names are unique and/or clearly describe what information values the variables hold. Code is somewhat efficient.	Code demonstrates the use of new concept(s), yet is not effective. Few or no variable names are unique and/or clearly describe what information values the variables hold. Code is not efficient.
Maker	Tangible component is tightly integrated with the micro:bit and each relies	Tangible component is somewhat integrated with the micro:bit but is not	Tangible component does not add to the functionality of the program.	No tangible component

component	heavily on the other to make the project complete.	essential.		
Work Logs	All work logs submitted on time, and accurate	Two late or missing work log and/or work logs not accurate nor sufficiently detailed.	Four late or missing work logs and/or work logs not accurate nor sufficiently detailed.	More than four late or missing work logs and/or not accurate nor sufficiently detailed.
Final narrative	Narrative piece is thoughtful and detailed and contains all required elements.	Reflection piece is mostly thoughtful and/or lacks 1 of the required elements.	Reflection piece is superficial and lacks 2 of the required elements.	Reflection piece is trivial and lacks 3 of the required elements.

Examples

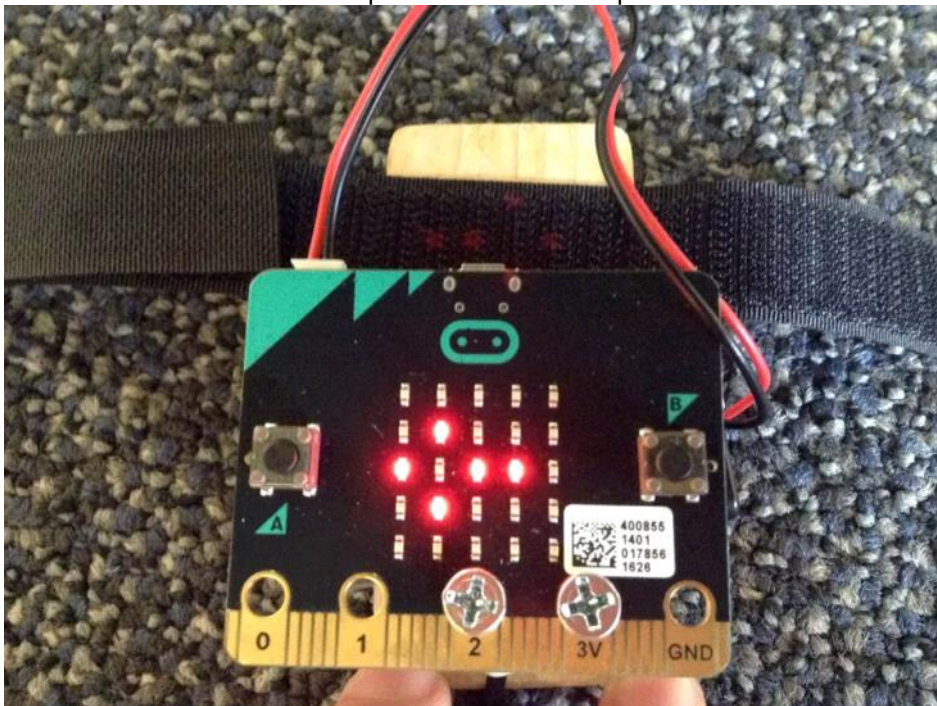
Final Project Examples

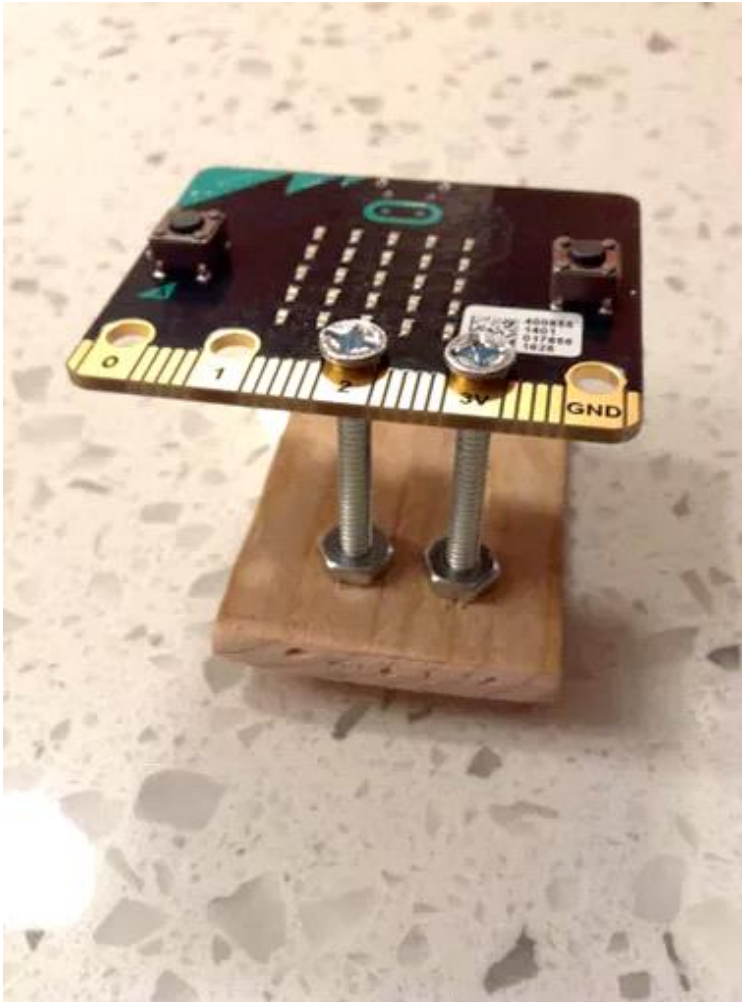
1. Baseball Pitch Counter



This project straps to a pitcher's arm and uses the micro:bit accelerometer to record how many pitches have been thrown in a session.

2. Micro:bit Wrist-Mounted Step Counter and Compass





This project straps to your wrist, displays a compass that updates as you walk around, and keeps track of your steps. The micro:bit is elevated to allow room for the battery pack to fit underneath.

3. Combination Lock Box

[micro:bit Combination lockbox](#)



This project features a secret combination that opens the top of the box using a servo motor

4. Violin Tuner





This project uses a piece of cardboard to mount the micro:bit to the side of a violin. This student wanted to use it to tune his violin by playing a specific series of tones. The micro:bit displays the note being played.

5. Trumpet Angle Detector



This example was used for Marching Band practice, where students must hold the trumpet at a 15-degree angle to avoid hitting the person in front of them or playing directly into their ears. Because the trumpet is heavy, new trumpet players tend to let the trumpet droop. This displays an icon (a check mark or an X) to help new trumpet players learn what the proper angle is supposed to feel like.

Standards

CSTA K-12 Computer Science Standards

- CL.L2-03 Collaborate with peers, experts, and others using collaborative practices such as pair programming, working in project teams, and participating in group active learning activities.
- CL.L2-04 Exhibit dispositions necessary for collaboration: providing useful feedback, integrating feedback, understanding and accepting multiple perspectives, socialization.
- CL.L2-05 Implement problem solutions using a programming language, including: looping behavior, conditional statements, logic, expressions, variables, and functions.