15 Fun Coding Projects!

Coding for kids

Second Edition

- Learn the basics of coding
- Create apps and games
- No experience required

Camille McCue, PhD
Kids' coding teacher and author

for dummies®
A Wiley Brand

Coding for kids

2nd Edition

by Camille McCue, PhD

for dummies®
A Wiley Brand

Coding For Kids For Dummies®, 2nd Edition

Published by: **John Wiley & Sons, Inc.,** 111 River Street, Hoboken, NJ 07030-5774, www.wiley.com

Library of Congress Control Number: 2019934591

ISBN 978-1-119-55516-2 (pbk); ISBN 978-1-119-55519-3 (ebk); ISBN 978-1-119-55522-3 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Table of Contents

Introduction

So you want to learn to *code* — awesome! *Coding* — writing computer programs — has something for everyone: creativity, logic, art, math, storytelling, design, and problem solving. From games and simulations to helpful tools and electronic gadgets, this book coaches you step by step through coding *real programs* in *real programming languages* that you can share with family and friends.

About This Book

Many kids want to learn to code, but not every kid has computer programming classes at school or a camp he or she can attend during the summer. That's where this book comes in!

Coding for Kids For Dummies will help you learn all of the basic coding ideas and skills used by real computer programmers. Everything you do here will be useful in learning new skills and more advanced programming languages in the future. Best of all, the tools in this second edition are free, available online, and easy-to-use.

This edition of the book covers the following:

- **Scratch**, a learning language developed at MIT that has risen in prevalence to the point where it is arguably the most popular kid programing language available. As such, this book features numerous projects in the most recent version of Scratch — Scratch 3.0. Scratch is a block-based language that offers new coders an easy entry into computer programming. And it's fun!

- **JavaScript,** which is used in everything from apps to websites to electronics. New programming environments have made JavaScript more accessible than ever through interfaces that allow you to switch between block-based and text-based formats. You can begin learning in block-based mode (as in Scratch), and then transition to text-based mode as you build skills and confidence in coding. In this book, JavaScript projects are presented through two different vehicles (officially called *IDEs* — integrated development environments): Code.org's App Lab, for building mobile device apps, and MakeCode, for coding instructions to operate a small electronics board called a micro:bit.

- **Fundamental computer programming concepts,** which apply to both the projects in this book and additional coding (and, more generally, computer science) work you might pursue in the future.

Additionally, graphic design and animation are incredibly important skills that go hand-in-hand with coding to create great-looking and easy-to-understand digital tools. Although this book provides a little bit of guidance in these areas, the main focus of the content in these pages is coding.

Foolish Assumptions

Hello person buying this book and reading this intro! I assume you are a kid who wants to learn to code. Awesome! You are starting on an adventure that will take you from being a user of technology to being a maker of technology. And it's a lot easier than you might think.

Here are a few other assumptions I make about you (or your technology) as you get started:

- You are comfortable typing on a tablet or a computer and using a mouse or touchpad. Your experience can be either on a Windows or Mac system because instructions for coding each project are platform-independent.

- You have an Internet connection and know how to open a web browser to access websites.
- For readers choosing to use the micro:bit electronics board, you have a USB port on your computer (via which you'll connect the micro:bit).
- You've played with a few apps, websites, or games on a computer, so you have some idea regarding how user interfaces (UI) look and how people interact with a computer via the UI.
- You're comfortable with basic math, math operations such as adding whole numbers, and logical operations such as comparing two whole numbers. I introduce algebraic variables in this book, but you don't need to have any prior knowledge of variables.

Lastly, if you struggle with spelling and punctuation — and you're operating in text-based mode — you may need to spend extra time troubleshooting your code for misspellings. The IDE for a programming language can give you clues about which commands it doesn't understand, but you will need to pay special attention to the details.

Icons Used in This Book

As you work through the projects in this book, you'll see four icons. These icons point out different things.


TIP

The Tip icon gives you a tip that you can use to make your work easier. You'll see some tips over and over again.


REMEMBER

The Remember icon helps you remember and connect the coding concepts and skills you're working on with the big ideas of coding!


WARNING

The Warning icon tells you to watch out! It marks important information that may save you headaches.

The Technical Stuff icon lets you know more about the nuts and bolts of technical details and hardware help.

Beyond the Book

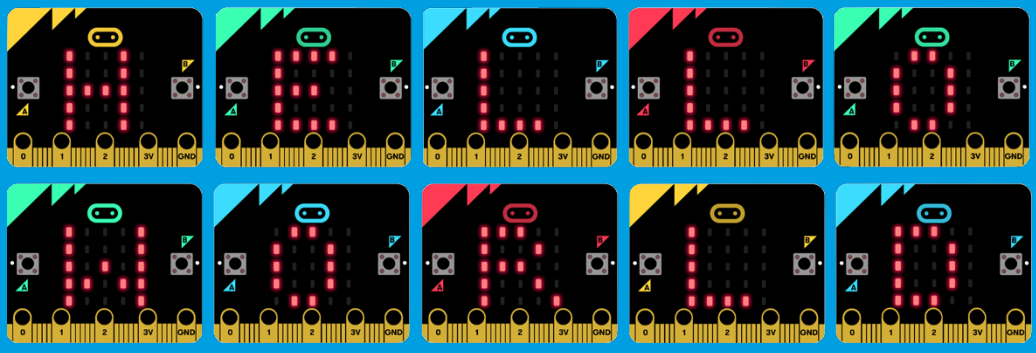
On the Dummies.com website, I give you some extra goodies that you won't find in this book. Go online to `www.dummies.com/cheatsheet/codingforkids` for a cheat sheet of coding commands in Scratch and JavaScript. (You can also type **Coding for Kids cheat sheet** in the search bar at `www.dummies.com`.) Download the information, print it, and keep it with your computer!

Where to Go from Here

You can work on the projects in order, or you can jump around and work on any project you choose. After you gain a little experience coding, you can go in a bazillion new directions. Learn more advanced concepts in Scratch and JavaScript. Make up your own projects. Work on learning more advanced programming languages.

I hope this book inspires you to continue learning more about coding and making things with tech. Kudos on taking the first step! Now go get started!

Part 1
Getting Started

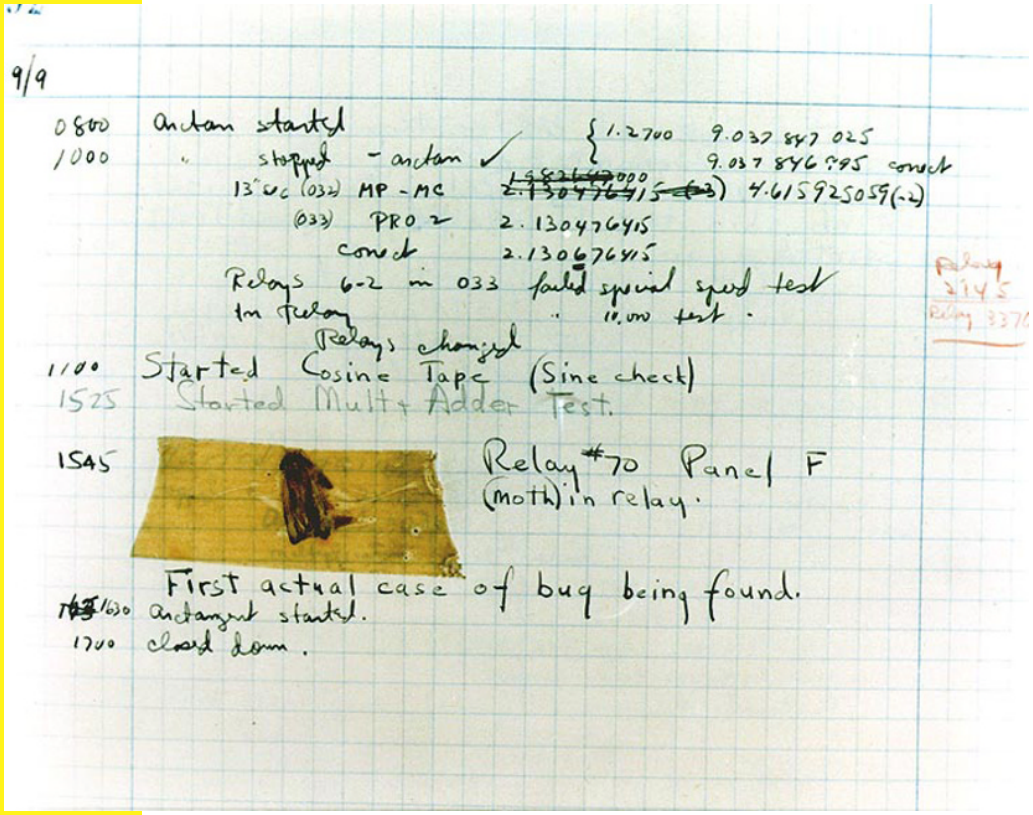


CHAPTER

3

When Things Go Wrong

Programming is a picky enterprise. A program will often contain errors, and you have to fix them before the program will run just right. Whether you're just starting out in coding or you've been programming for a while, you — like everyone else — will make mistakes when writing code. As a coder, you'll spend a lot of time fixing the mistakes, or *bugs*, in your programs. (They're called *bugs* because one of the "mistakes" in the early days of computing was an actual bug in the computer's electronic circuitry!)



https://upload.wikimedia.org/wikipedia/commons/8/8a/H96566k.jpg

Fixing mistakes in your code is called *debugging* — literally, getting the bugs out of your code. Think of your program like an essay you write in English class or a lab report you write in science class. In each written document, you must get the form and content right for the document to make sense and tell its story successfully.

When you write and run your code, you must also get the form and content correct; if you don't, the code will encounter errors. When the bug is due to the code's form, you get a *syntax error.* When the bug is the result of the code's behavior, you get a *logic error.*

Testing your program means running it to see if it operates the way you want it to. In this chapter, you discover some ways that syntax and logic errors pop up when you test your programs. You also learn some strategies for debugging your code in Scratch and JavaScript (using both App Lab and MakeCode). Remember, half of coding is writing the code — and the other half is debugging it!

Syntax Errors

The form, or *syntax,* of what you code consists of the structure, the commands, the grammar, and the punctuation of your program. Here's what each of those pieces means:

- The *structure* of your program is its overall layout. For example, an essay is often structured using five paragraphs: an introductory paragraph, three content paragraphs, and a closing paragraph. In a Scratch program, code is structured to go with the object that executes it. For example, a butterfly sprite has code that makes it fly, and a monkey sprite has code that makes it bounce around. In a JavaScript program, code is structured to feature global variables at the top, then the main program, and then function definitions.

- The *commands* in your program are like the vocabulary in an essay. Commands, like vocabulary, have specific meaning and must be spelled correctly. You can write new commands (and use any spelling you want), but you must define what these new vocabulary words mean in your program.

- The *grammar* of your program is similar to the grammar you use when writing sentences in any language. In English, you say "I'm not hungry," not "I ain't hungry" and not "Hungry not I'm." In the same way, when coding in Scratch, the proper grammar is `repeat 10 play sound boom`, not `repeat 10 boom sounds`. Grammar requires that you use certain words together and that you order those words in a specific way.

- The *punctuation* of your program looks a lot like the punctuation in an essay. You must put the periods, commas, and semicolons in the correct places for your code to work. The punctuation of your code also includes some of the symbols you use when writing math equations, such as greater than and less than signs, plus and minus signs, equals signs, parentheses, brackets, and curly braces.

When coding with blocks, it's almost impossible to make syntax errors. Block-based coding requires almost no typing, and the blocks fit together in specific ways. (In fact, the point of coding with blocks is that it helps beginners avoid syntax errors.) When coding in text-based mode, you type everything into the program. You can easily misspell commands and forget to end a line of code with a semicolon. Any error in syntax will be reported to you, either as you type or when you try to run your code. You have to fix errors to execute your program correctly.

Many programming languages, such as Java, require you to compile your program before you can run it. *Compiling* is like checking your code for obvious errors. If you have syntax errors, they are reported to you when you try to compile. Bugs must be fixed to get a successful compilation, after which you can then run your program.

Logic Errors

The *logic* of your program is the behavior that results when you run your code. You can easily get the syntax of a program correct but make mistakes in the logic of the code. For example, if the game score goes down each time a hero defeats an enemy,

you may have coded a `score` variable to decrease, not increase. The code runs because the syntax is correct, but the program doesn't do what you want it to do.

Whether coding with blocks or coding in a text-based mode, you can easily make logic errors. You probably won't notice logic errors until you run your code. At that point, you need to figure out what is going wrong, and then find the part of your code that controls the behavior that is not working correctly. Sometimes tracking down a logic error is easy, but other times it's challenging!

Debugging Scratch Programs

Scratch helps prevent you from making syntax errors because you don't type the commands. Instead, you drag command blocks into the Code workspace. You don't have to worry about misspelling a command or forgetting to add a semicolon because Scratch prevents you from making these types of errors. Instead of typing a premade element such as a variable name, for example, you select it from a drop-down list, as shown in Figure 3-1.

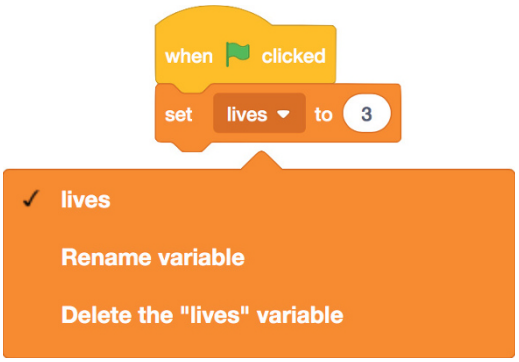


Figure 3-1

When coding in Scratch, you're likely to make mistakes in the logic of your program. These mistakes show up when you run your program, so they are also called *runtime errors.* To fix this type of error, you try to pinpoint the place where something is going wrong in the execution of your program. Then you change the code to correct your error (or errors).

Here are some ways to locate and fix runtime errors in Scratch:

- **Focus on one sprite at a time.** If a sprite is working properly, move on to the next sprite and check its operation. (In many projects, you also write code for the stage, and you can use this method to check the stage as well.)

- **When you identify a sprite with a runtime error, run the program several times to see how the sprite is behaving.** Try to figure out exactly what the sprite is doing wrong and when that behavior occurs.

- **Look carefully at the code for the sprite with the runtime error.** Step through the code one command at a time. Separate some of the code (remove it temporarily by dragging it to the side, away from the `event` command). Then add sections of your code back in, one or two commands at a time, and test the operation of the sprite after each addition. You should be able to identify where the error is occurring. At this point, correcting the error is usually easy.

Figure 3-2 shows an example of a logic, or runtime, error in Scratch. I want my butterfly to fly towards the left of the screen, but it is flying towards the right. I think to myself, "Hmmm, my butterfly is flying, and he's flying at the speed I want. But he's going the wrong way. Maybe I set my direction incorrectly."

So I look at the part of my code that relates to direction. I see that I have set the direction to positive 90, pointing the butterfly to the right. I should have set the direction to negative 90 (by typing `-90`), pointing the butterfly to the left.

Figure 3-2

Debugging App Lab Programs

As you program in JavaScript using App Lab, you will likely produce some errors in your code. Mistakes in the syntax of your code must be fixed before you can run the program.

In App Lab, syntax errors are displayed as yellow diamonds or red squares. The error symbol appears on the line where you have a bug. If you hover your cursor over the symbol, App Lab provides a clue about the error. Here's a general description of what each error symbol means:

- **Yellow diamond:** Your code has a possible syntax error, but the app might still run. The syntax error might be due to misspelling something or any number of other problems.


REMEMBER

The `onEvent()` command always display a yellow diamond when you don't delete the default function call name, `event`. However, this is one error that does not prevent your app from executing, so you don't need to fix it. But if you find that yellow diamond irritating, you can usually just remove `event` from between the parentheses and the warning will vanish.

- **Red square:** A syntax error is preventing the app from running. For example, you might be using a variable or a function that you haven't declared. You must fix syntax errors before attempting to run your app.

If you program in block mode, you won't make too many syntax errors, but it is possible to make a few. For example, you might make a syntax error when spelling the name of a variable or function — or anything else you have to type in a block. Figure 3-3 shows a snippet of block code that App Lab has flagged with three errors.

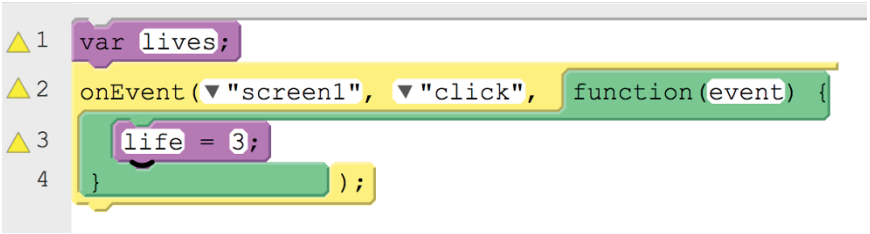


Figure 3-3

If you hover your cursor over the yellow triangle on line 1, App Lab displays the message shown in Figure 3-4. This message tells you that you defined a variable, `lives`, and then didn't use it in your code.

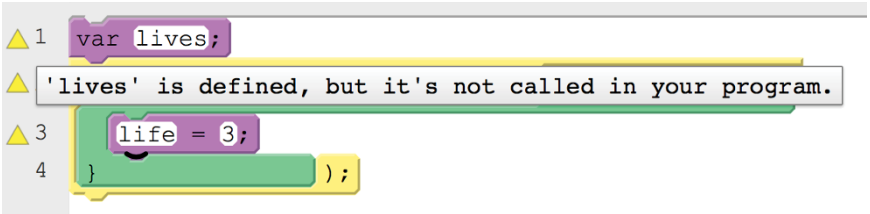


Figure 3-4

If you hover your cursor over the yellow triangle on line 2, the message shown in Figure 3-5 appears. This message tells you that the default function name, `event`, is not called in your code. As mentioned, in App Lab it's okay to have yellow triangles on all `onEvent` commands. If you want to correct the error, just delete `event`.

If you hover your cursor over the yellow triangle in line 3, the message shown in Figure 3-6 appears. This message lets you know that the variable for which you're attempting to set the value, `life`, doesn't exist. (You never declared it because you meant to set the value of `lives` not `life`.)

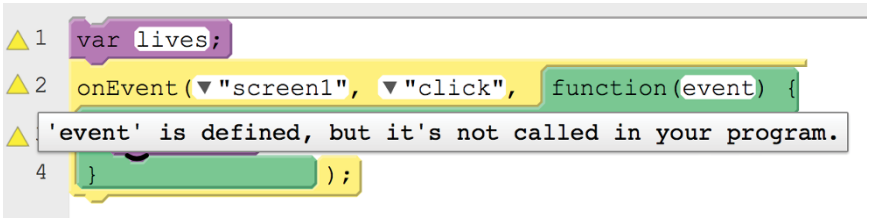


Figure 3-5

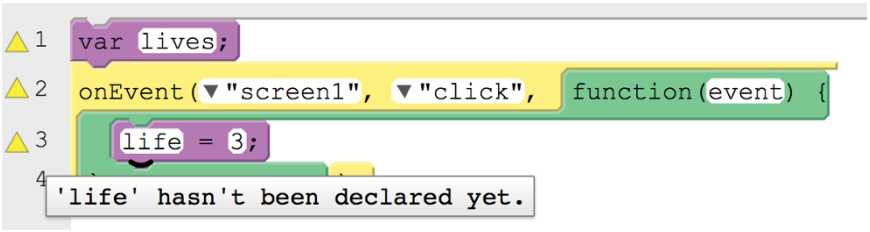


Figure 3-6

To debug this code snippet, delete `event` from line 2 and change `life` to `lives` in line 3, as shown in Figure 3-7. Note that changing `life` to `lives` also gets rid of the error in line 1.

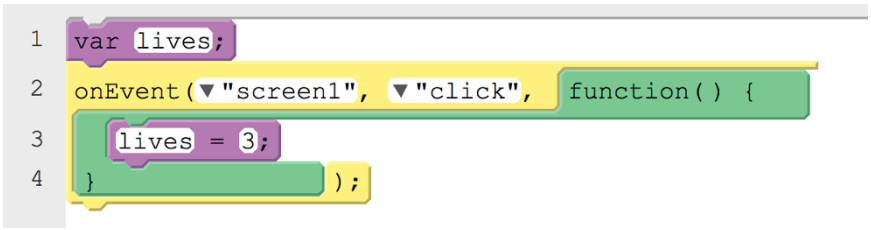


Figure 3-7

Other syntax errors are not flagged as you code but show up at runtime. Figure 3-8 shows a code snippet built in text-based mode. As you can see, App Lab does not display any yellow triangles or red squares.

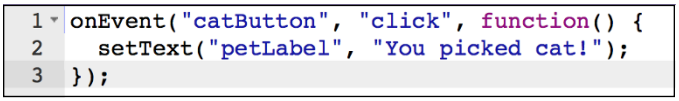


Figure 3-8

When you run the JavaScript code, however, a warning message appears at the bottom of the App Lab IDE, in the Debug Console. As shown in Figure 3-9, the warning tells where the error is located and what the error might be. Here, I forgot to name a button in my app. The name `catButton` doesn't exist because I forgot to change the default name from `button1`.

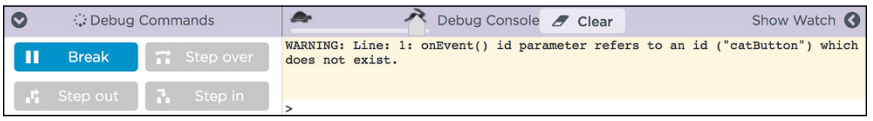


Figure 3-9

The Debug Console offers several additional tools for helping you debug your app. Besides catching syntax errors that you do not catch while coding, the console can help you track down logic errors that show up at runtime. Here are two key ways you can use the Debug Console to find and fix your errors:

- **Use the slider to adjust the speed of execution of your program.** You can change the speed between turtle (slow) and rabbit (fast). Controlling the speed allows you to catch the moment when an error is first encountered.
- **Pay attention to the details of the error messages.** The messages in the Debug Console often tell you which line number the bug appears on, and what the error might be.

The Debug Console has additional buttons and tools; read App Lab's online documentation for details.

TIP

Click the Reset button in the App Lab simulator to stop code execution and reset the app.

Lastly, you can also use the `console.log` command to report tracking information to you in the Debug Console as the program runs. Figure 3-10 shows a code snippet that asks the user to type his or her name. The code should store the name, but because the code doesn't print the name, you can't tell whether the name was stored as you intended. By adding a `console.log`

command, you can look behind the scenes by making the name print in the Debug Console. As you can see in the figure, the program read and stored the name.

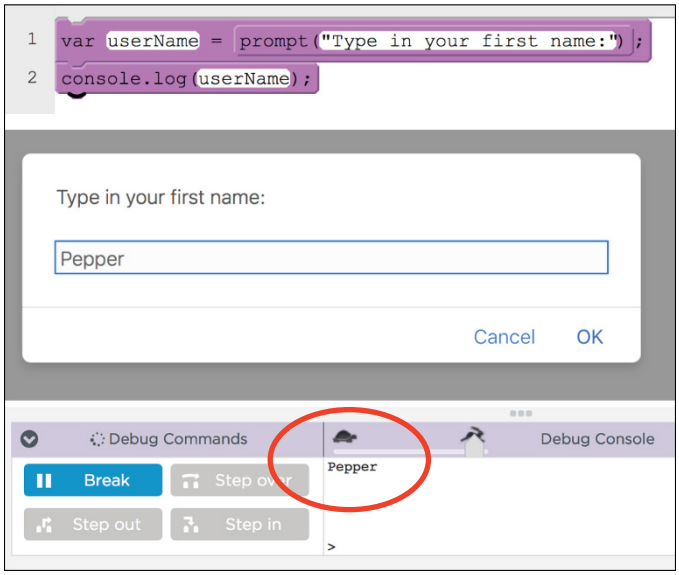


Figure 3-10

Debugging MakeCode Programs

The block mode of MakeCode is like Scratch in that it keeps you from making syntax errors. Because you use premade blocks, as shown in Figure 3-11, you can't type misspellings or forget to include punctuation.

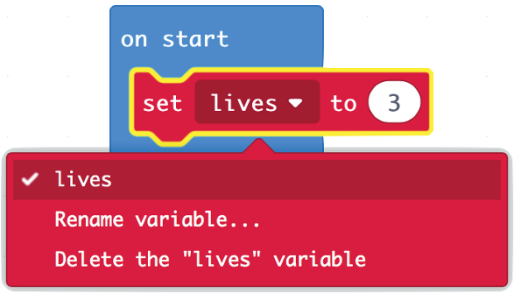


Figure 3-11

However, if you use the JavaScript text-based mode of MakeCode, you type your code, so it is possible to make syntax errors. For example, Figure 3-12 shows you what happens if you try to use a `life` variable when you should be using `lives`. MakeCode underlines the problem code with a red squiggly line and displays the error message, `Cannot find name 'life'.`

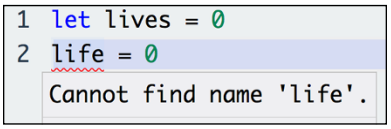


Figure 3-12

When coding in MakeCode, errors can also show up at runtime. If you run a MakeCode program in the micro:bit simulator and get an error, an Explorer window appears just below the micro:bit simulator. The window displays the number of errors in the program, as shown in Figure 3-13.

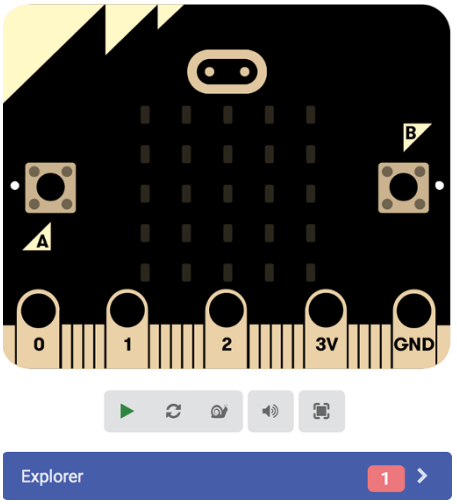


Figure 3-13

You can click the Explorer window to expand it and view additional information about the execution of your program and the errors. But be aware that it's a little challenging for new coders to read and understand the Explorer window!

Your best bet for fixing runtime errors in MakeCode is to run the program several times, tracing the code execution step-by-step. Try to isolate the exact moment when an error takes place, and then go to that part of the code and examine it. Update the code to correct your error (or errors).


TIP

You can press the Slo-Mo button in the simulator to slow the speed at which the program runs, making it easier to trace the action.

Commenting Out Code when Debugging

One final way you can debug code is to comment out sections of code. *Commenting out code* allows you to temporarily remove commands from your program so that you can isolate where things are going wrong. You comment out code by using special symbols in front of one or more lines of code when working in text-based mode. For the projects in this book, commenting out code is a debugging technique you can use with text-based JavaScript in App Lab and MakeCode.

Suppose you wrote the program shown in Figure 3-14, where something is wrong with `carrotVotes`. When a user clicks `carrotImage`, the number of votes for the carrot increases only on the first click.

```
1  var carrotVotes = 0;
2  var cookieVotes = 0;
3  onEvent("carrotImage", "click", function() {
4    carrotVotes = 1;
5  });
6  onEvent("cookieImage", "click", function() {
7    cookieVotes = cookieVotes + 1;
8  });
```

Figure 3-14

So you, as the app developer, decide to comment out any code that has to do with the carrot. For a single line of code, as in line 1, you use two forward slash symbols (`//`) in front of the code. For a larger block of code, as in lines 3 to 5, you use a single forward slash and an asterisk (`/*`) at the beginning of the block, and an asterisk and a single forward slash (`*/`) at the end of the block. The code now looks like Figure 3-15.

```
1  // var carrotVotes = 0;
2  var cookieVotes = 0;
3  /*onEvent("carrotImage", "click", function() {
4    carrotVotes = 1;
5  }); */
6  onEvent("cookieImage", "click", function() {
7    cookieVotes = cookieVotes + 1;
8  });
```

Figure 3-15

After commenting out the code, you run the program and everything else works correctly. You've confirmed that the errors are in the code that you temporarily removed. You change line 4 to read `carrotVotes = carrotVotes + 1`. Then you remove the backslashes and asterisks, and run the code again. This time, the program works as you intended.

Debugging your code is a normal part of programming. You won't write code perfectly the first time — no one does. Patience and practice are vital to the debugging process.

The computer only understands the commands you give it, so it's up to you to work carefully and pay attention to every line of code you write. Although debugging your code can be time-consuming (and frustrating), it's just part of the job of being a coder!

Part 2

Sounds, Color, Random Surprises


```
onEvent("Cat", "click", function() {
  playSound("sound://category_animals/cat.mp3", false);
});
onEvent("Bee", "click", function() {
  playSound("sound://category_animals/bee_buzz.mp3", false);
});
onEvent("Bear", "click", function() {
  playSound("sound://category_animals/bear.mp3", false);
});
onEvent("Cow", "click", function() {
  playSound("sound://category_animals/cow.mp3", false);
});
```

Hailey Nemeth

CHAPTER

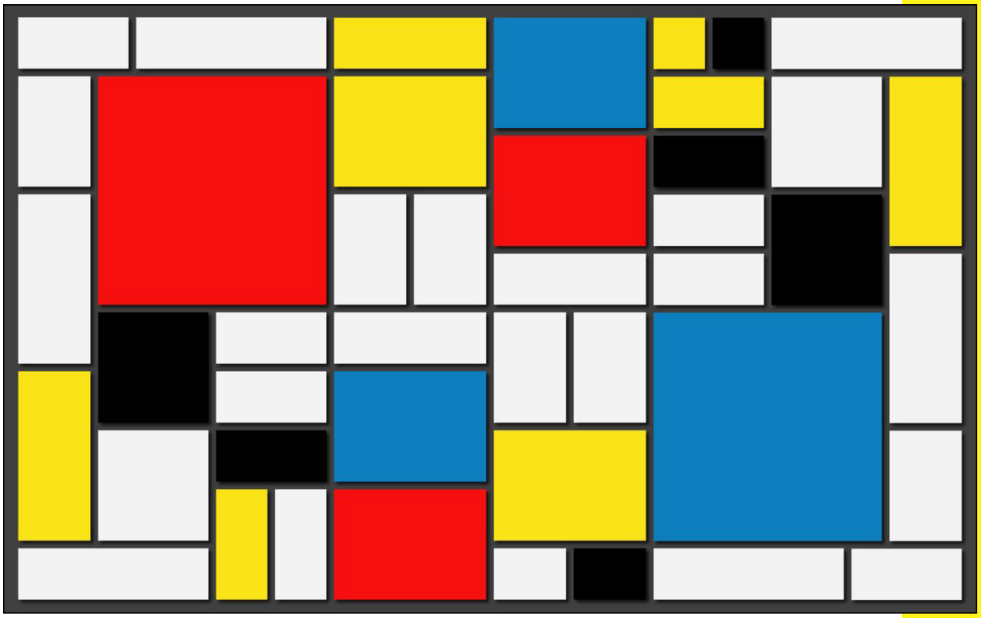
7

Mondrian Art Toy

Coding color makes programs more interesting and realistic. You may be surprised to know that early video games, such as Pong, were black and white! Now you can mix light from red, green, and blue pixels on the computer screen to make millions of colors.

In this chapter, you add a new level of excitement to a game by coding color. You use skills you gained coding randomness (see Chapter 6) and learn how to use touchscreen sensors to locate position. (You discover even more about position in the next chapter.)

In the Mondrian Art toy, you use App Lab to build an app that lets your end user create art in the style of Piet Mondrian, the Dutch artist known for his abstract paintings of black rectangles filled with the primary colors red, yellow, and blue. You code your program to make rectangles of random sizes with user-selected colors, making every end user's painting a unique work of digital art!



https://commons.wikimedia.org/wiki/File:Piet_Mondriaan_-_03.jpg

Brainstorm

You can create your Mondrian Art drawing toy using lines of any width and color you want. You can set the random range of the rectangles' *dimensions* — the width and length — however you want. And you can offer the end user any fill colors you want, including randomly selected colors or colors with some transparency. Figure 7-1 shows an example of a completed toy featuring a canvas of art digitally painted by an end user.

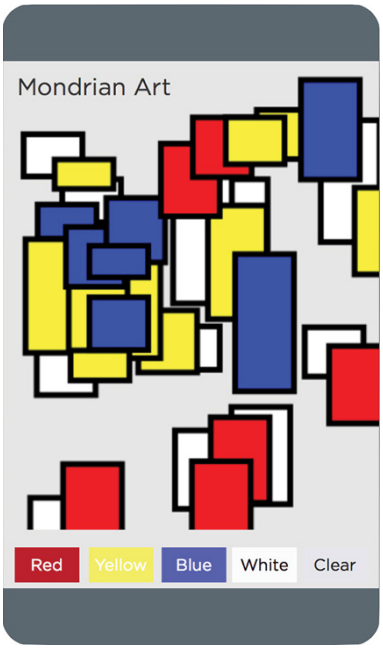


Figure 7-1

Start a New Project

Begin creating your Mondrian Art toy app by starting a new project:

1. Open App Lab at `https://code.org/educate/applab`. Log in to the account you created to use App Lab (see Chapter 2). Under the App Lab heading, click the Try It Out button.

   A new project opens.

2. Name your program by clicking the Rename button and typing a name in the Project Name field at the top of the App Lab interface.

3. Click the Save button.

Add a Background Color

Most Mondrian paintings have a white background, but you can choose to paint your background a different color. To make the colorful rectangles pop, consider changing the background to a light gray.

You can change the background color of your app as follows:

1. Click the Design button to switch to Design mode in App Lab.

   The Design toolbox and workspace are displayed.

2. On the Properties tab of the workspace, rename the ID of screen1 to a more meaningful name such as `artScreen`, as shown in Figure 7-2.

WARNING

Do not include spaces in ID names. In most programming languages, including App Lab, spaces are not allowed.

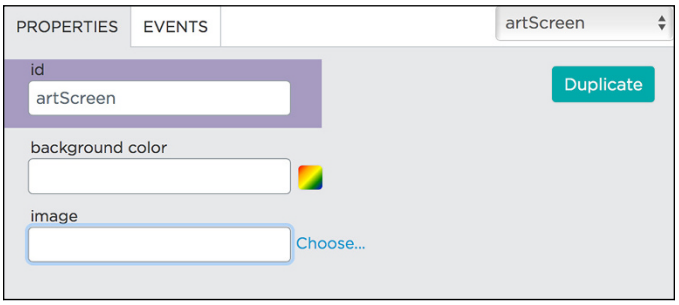


Figure 7-2

3. Click the small square of color to the right of the Background Color field to open the Color editor for the background.

4. Click the toggle arrows (the up and down arrows) to switch to RGBA color mode.

   Other options shown are HSLA and HEX. HSLA stands for Hue-Saturation-Lightness-Alpha and HEX stands for hexadecimal. Each is a different method of setting color.

5. The default white background color in the Background Color field is rgba(0,0,0,1). Create any background color you want by typing a value from 0 to 255 in each of the R, G, and B fields, and a value of 1 in the A (alpha) field. Or slide the color slider (the first slider) and the alpha slider (the second slider) to create the color and transparency you want.

   To follow along with the example and create a light gray background, type `230` (or thereabouts) in each of the first three fields and `1` in the A field. See Figure 7-3. See the "RGBA Color" sidebar for more information on working with color and transparency.

Figure 7-3

RGBA Color

An RGBA color code give values for red, green, blue, and alpha. Red, green, and blue each range from 0 to 255 in value. A value of 0 means no light is displayed in that color, resulting in the darkest color. A value of 255 is the greatest quantity of light in that value, resulting in the brightest color. The alpha parameter is a number from 0 (transparent) and 1 (opaque). If no alpha value is given, it is assumed to be 1. An alpha value of 0.5 would be half transparent and half opaque.

Here are some examples of RGBA values and their associated colors:

(0,0,0,1)	Black
(255,0,0,1)	Red
(0,255,0,1)	Green
(0,0,255,1)	Blue
(255,255,0,1)	Yellow — red light and green light produce yellow light
(0,255,255,1)	Cyan — green light and blue light produce cyan light
(255,0,255,1)	Magenta — red light and blue light produce magenta light
(50,50,50,1)	Dark gray
(150,150,150,1)	Medium gray
(200,200,200,1)	Light gray
(255,255,255,1)	White

Keep in mind that you're mixing colored pixel lights on a computer screen, which produces different results than mixing crayon colors on a piece of paper. For example, mixing red and green light produces yellow light. But mixing red and green crayons produces an icky brown shade.

When coding, you can set a pixel color value by typing a number in the color range, which is 0 to 255. Remember, small numbers means less light and make a darker color on a display. Large number means more light and make a brighter (lighter) color on a display.

Putting together randomness and color, you can write code to make random colors for your apps. Replace any pixel color value with `random(0, 255)` to create 256 variations for that color. Replacing each pixel color value (each of red, green, and blue) creates 256 color variations for each color channel. That means you can produce 256 * 256 * 256 colors, which is more than 16 million combinations!

Also remember that alpha values less than 1.0 cause your color to have some transparency. When layering colored objects (such as filled rectangles) on top of each other, color blending can be seen in the overlapping regions. This produces some interesting artistic effects.

Add a Title Label

Next, you should name your app with a label that helps the user know the purpose of the app. Add a title as follows:

1. Remain working in the Design mode of App Lab. If you're not in Design mode, click the Design button.

   The Design toolbox and workspace are displayed.

2. In the Design toolbox, drag the Label icon and position it near the top left of the app display.

   The label will be placed in front of the drawing canvas, so that it can be seen.

3. On the Properties tab of the workspace, change the attributes of the label as follows:

   - ID: Rename the ID to `titleLabel`.
   - Text: Type the title of your app, such as **Mondrian Art.**
   - Width (px): Increase to width of your label to something like 240 pixels (or more), so that the title will appear on a single line.

- Height (px): No change.
- x Position (px): No change; you change the x position later by dragging the label into position.
- y Position (px): No change; you change the y position later by dragging the label into position.
- Text Color: Click the small square of color to the right of the Text Color field and choose a color for your title that will contrast well with the background.
- Background Color: No change.
- Font Size (px): Type a new font size in the field or use the selection arrows to make the title the appropriate size for your app.
- Text Alignment: Click the selection arrows and choose Left.

4. Click and drag the label on your app to position it where you want.

   Refer to Figure 7-1 to see the position of the title label on the app simulator display.

Add Fill and Clear Buttons

Your app will feature buttons that the user can click or tap to select the fill color for each Mondrian rectangle. Follow these instructions to add each of these buttons to your app:

1. Remain working in the Design mode of App Lab. If you're not in Design mode, click the Design button.

   The Design toolbox and workspace are displayed.

2. In the Design toolbox, drag the Button icon to the app display.

3. On the Properties tab of the workspace, change the attributes of the button as shown here:

   - ID: Rename the ID to `redButton` (or something similar) to indicate that this button will allow the user to draw a red-filled rectangle (or whatever color you choose).
   - Text: Type the text that will appear on the button, such as **Red** (or whatever color you choose).
   - Width (px): Increase to width of your button, to something like 55 pixels.
   - Height (px): Increase to height of your button, to something like 55 pixels.
   - x Position (px): No change; you change the x position later by dragging the button into position.
   - y Position (px): No change; you change the y position later by dragging the button into position.
   - Text Color: Click the small square of color to the right of the Text Color field and choose a text color that will contrast well with the button background.
   - Background Color: Click the small square of color to the right of the Background Color field and choose a button color that will contrast well with the background color of the app. For example a button named Red would logically have a red background color.
   - Font size (px): Type a new font size in the field or click the selection arrows to make the text on your button a good size for your app.
   - Text alignment: Click the selection arrows and choose Left.

4. Drag the button on your app display to position it where you want. Remember that you will create several buttons and need to leave room for them all.

5. Create another button by selecting the first button you created and then clicking the aqua-colored Duplicate button in the workspace.

6. Repeat Step 5 until you've created all your buttons.

7. Change the ID and text on each of your duplicate buttons so that each one represents a different color. For example, I created a button for Red, Yellow, Blue, and White. I also created a button for Clear, which will be used to clear the screen of a painting.

8. Drag each of your new buttons to the app display, below the drawing canvas.

   The goal is to create a clean and readable user interface for the end user of your Mondrian Art toy. Refer to Figure 7-1.


TIP

To make the use of your app understandable to end users, organize your buttons in easy-to-find locations.

Code a Canvas and Paintbrush

The *canvas* is a rectangle-shaped area on the app screen where end users can paint — like a real artist's canvas! The canvas is see-through, so the background color that you set previously appears. You'll create the canvas and also define the *paintbrush* — the line color and thickness of the rectangles that the user will paint.

Create and code a canvas on the background and then code the paintbrush as follows:

1. Switch to Code mode in App Lab.

   The toolbox of commands and the workspace are displayed.

2. In the toolbox, select Canvas. Drag the `createCanvas` command into the workspace. Click in the `ID` field and type the name `"drawingCanvas"` (include the quotation marks). Set the width (the first number field) to `320`. Set the height (the second number field) to `400`.

   The command looks like this:

```
createCanvas("drawingCanvas", 320, 400);
```

3. Drag the `setActiveCanvas` command into the workspace, below the first command. Click in the `ID` field and type the name `"drawingCanvas"` (include the quotation marks).

   The `drawingCanvas` command you previously created is set to the active canvas. The command looks like this:

```
setActiveCanvas("drawingCanvas");
```


WARNING

You can add a canvas also in Design mode. However, it's best to create this element by using code because this is how you would code this JavaScript outside App Lab.

4. Drag a `setStrokeWidth` command into the workspace, below the previous command. Type the number `5` in the empty field.

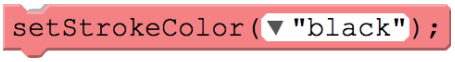
   The code looks like this:

```
setStrokeWidth(5);
```

   When the end user draws a rectangle, its border width will be 5 pixels — very Mondrian in style!

5. Drag a `setStrokeColor` command into the workspace, below the previous command. Type `"black"` in the Color field.

The code looks like this:



Any rectangle that the end user draws will have a black border.

Code to Draw a Rectangle

Next, you write code so that when the user touches anywhere on the app display, a randomly sized rectangle is painted at that location (read by the touch sensor at `event`). Follow these steps:

1. Continue working in Code mode.

2. In the toolbox, select UI controls. Drag an `onEvent` command into the workspace, below the previous command. In the `onEvent` command, click the `ID` field and type `"drawingCanvas"` (include the quotation marks). Leave the other attributes unchanged.

   When the user clicks or taps the drawing canvas, the code block inside this `onEvent` will execute.

3. In the toolbox, select Canvas. Drag a `rect` command into the `onEvent` command. In the `rect` fields, add the following attributes:

   - x: The first empty field of the rect command is the x-coordinate of the top-left corner of the rectangle. Type `event.x` in this field. (Note that a purple tile will appear around your typing.)

   - y: The second empty field of the rect command is the y-coordinate of the top-left corner of the rectangle. Type `event.y` in this field. (A purple tile will appear around your typing.)

- Width (px): The third empty field of the rect command is the width of the rectangle. Type `50` or a similar number in this field.
- Height (px): The fourth empty field of the rect command is the height of the rectangle. In the toolbox, go to the Math commands and drag a `random` command into this height field. Set the range of the random numbers by typing `20` or a similar number (for the minimum) and `120` or a similar number (for the maximum) in the empty fields.

These new lines of code are shown in Figure 7-4.

```
onEvent( ▼ "drawingCanvas", ▼ "click", function(event) {
    rect( event.x, event.y, 50, randomNumber(20, 120) );
} );
```

Figure 7-4

Run your code to test it so far. In the simulator, clicking anywhere on the app display should draw an unfilled rectangle of varying dimensions. Try clicking lots of times to make lots of rectangles!

Code to Fill Rectangles with Color

Now you write some code to allow the user to set the fill color to be used when a rectangle is drawn:

1. Continue working in the Code mode of App Lab.
2. In the toolbox, select UI Controls. Drag the `onEvent` command into the workspace, below the previous command.
3. In the `onEvent` command, click the `ID` tab and choose `redButton` in the list. Remove `event` from `function()`. Leave the other attributes unchanged.

   When the Red button is clicked, the code block inside `onEvent` will execute.

4. Select Canvas. Drag a `setFillColor` command into the `onEvent` command. Type `"red"` in the field, or click the tab and select an `RGB` command and then type in values.

   When the user clicks the Red button, the rectangle fill color is set to red. Then, when the user touches the app display at any location, a randomly sized red-filled rectangle will be painted.

5. Repeat Steps 1–4 to code each of the remaining fill color buttons: Yellow, Blue, and White.

Figure 7-5 shows the code for the Red, Yellow, Blue, and White buttons.

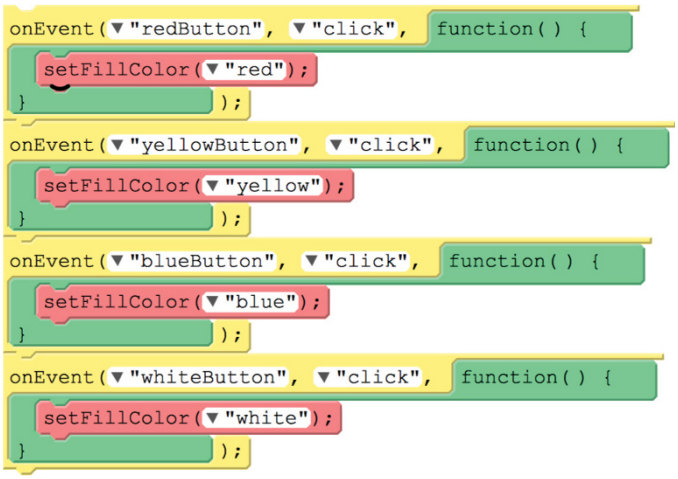


Figure 7-5

Code a Clear Button to Erase a Painting

After painting several rectangles, the end user may want to clear the painting and start fresh! Code a Clear button as follows:

1. Continue working in the Code mode of App Lab.

2. In the toolbox, select UI Controls. Drag the `onEvent` command into the workspace, below the previous command.

3. In the `onEvent` command, click the `ID` tab and choose `clearButton` in the list. Remove `event` from `function()`. Leave the other attributes unchanged.

   When the Clear button is clicked, the code block inside `onEvent` will execute.

4. Select the Canvas commands. Drag a `clearCanvas()` command into the `onEvent` command.

   Now, when the end user clicks the Clear button, the painting will be cleared from the display.

That's it! The design and code for the Mondrian Art toy are complete. Refer to Figure 7-1 to see the display of the completed app. Figure 7-6 shows the program in blocks.

```
1  createCanvas("drawingCanvas", 320, 400);
2  setActiveCanvas("drawingCanvas");
3  setStrokeWidth(5);
4  setStrokeColor("black");
5  onEvent("drawingCanvas", "click", function(event) {
6      rect(event.x, event.y, 50, randomNumber(20, 120));
7  });
8  onEvent("redButton", "click", function() {
9      setFillColor("red");
10 });
11 onEvent("yellowButton", "click", function() {
12     setFillColor("yellow");
13 });
14 onEvent("blueButton", "click", function() {
15     setFillColor("blue");
16 });
17 onEvent("whiteButton", "click", function() {
18     setFillColor("white");
19 });
20 onEvent("clearButton", "click", function() {
21     clearCanvas();
22 });
```

Figure 7-6

Here is the complete code in JavaScript:

```
createCanvas("drawingCanvas", 320, 400);
setActiveCanvas("drawingCanvas");
setStrokeWidth(5);
setStrokeColor("black");
onEvent("drawingCanvas", "click", function(event) {
  rect(event.x, event.y, 50, randomNumber(20, 120));
});
onEvent("redButton", "click", function() {
  setFillColor("red");
});
onEvent("yellowButton", "click", function() {
  setFillColor("yellow");
});
onEvent("blueButton", "click", function() {
  setFillColor("blue");
        });
onEvent("whiteButton", "click", function() {
  setFillColor("white");
});
onEvent("clearButton", "click", function() {
  clearCanvas();
});
```

Save, Test, and Debug Your App

As you work, App Lab automatically saves your program in the cloud. Test your program and fix any bugs to ensure that it works the way you want it to. For help with testing and debugging, see Chapter 3.

Share Your App with the World

After your app operates as you want it to, you can set the status of the program to Share. See Chapter 19 for details on sharing apps you create in App Lab.

Enhance Your App

Add some or all of these cool new features to your app:

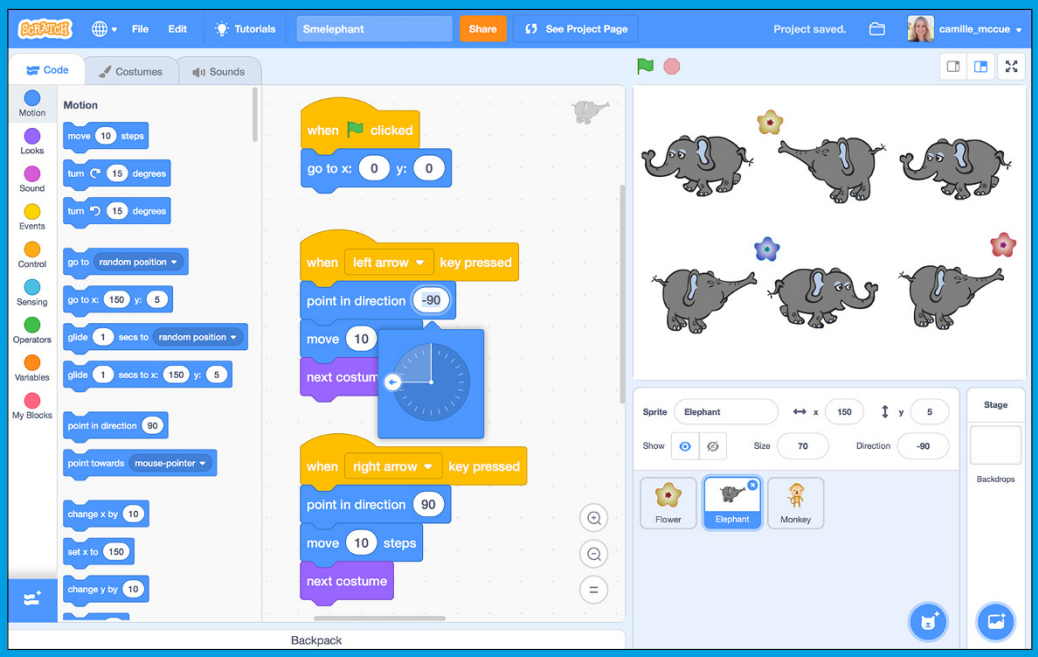
- **More randomness in rectangle sizes:** As built, only the height of the rectangle is random. Make the width of the rectangle random, too!
- **More color buttons:** Create additional buttons to produce other rectangle fill colors, such as green or purple.
- **Buttons that fill rectangles with half-transparent colors:** Instead of naming a fill color such as red, use an `rgb` command in the field of `setFillColor` for each button. The RGB command with four values is actually an RGBA command. Set the fourth value, the alpha (the A value) of each command to 0.5 to make the color half-transparent. For details, see the "RGBA Color" sidebar.
- **A button that fills rectangles with a randomly selected color:** Create a Random button and add this code:

```
onEvent("randomButton", "click", function() {
  setFillColor(rgb(randomNumber(0, 255),randomNumber(0, 255),
  randomNumber(0, 255),1));
});
```

  The quantities of each color channel — red, green, and blue — are randomly assigned. Every button click produces a new random color. See the "RGBA Color" sidebar for more on RGBA color.

Part 3

Moving from Here to There, Again and Again



CHAPTER

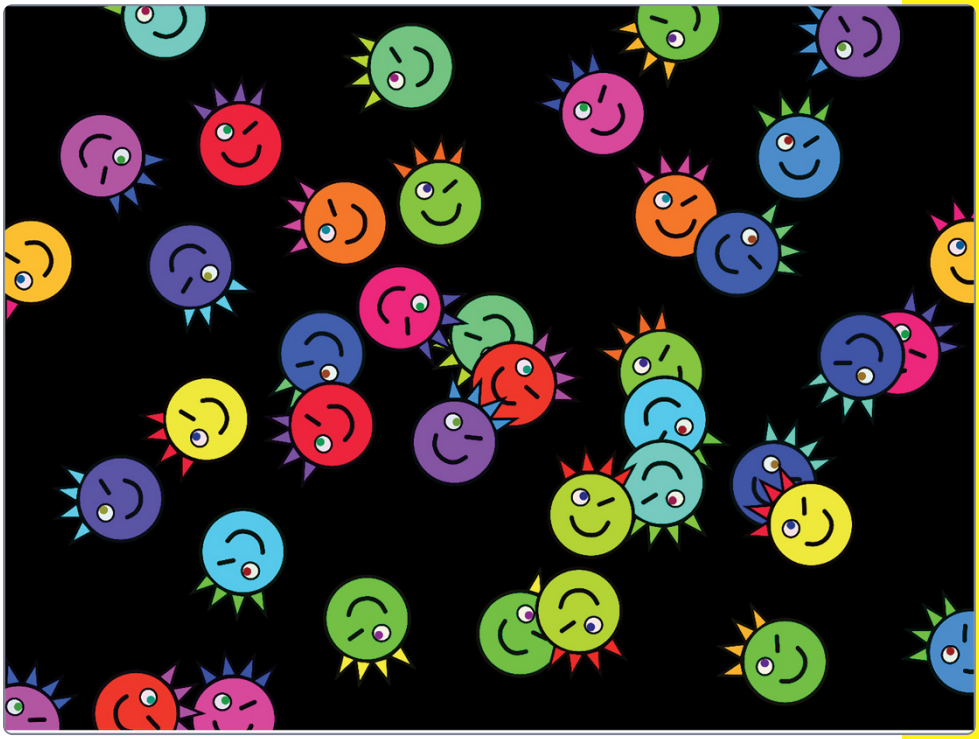
8

Emoji Explosion

In this chapter, you code Emoji Explosion, a fun animated scene where emoji objects of many colors bounce around the screen. You learn to set the coordinates and direction of the objects — using randomness to scatter the emojis everywhere! — and then make them move. Simple loops control the action.

To get started, you use Scratch to draw an emoji and then clone it to make a large group of emojis moving and bouncing around the screen. Now that you are writing longer programs, you also create new code blocks (sometimes called *functions*) to organize your code — like a real coder.

Putting together randomness, motion, loops, cloning, and functions in one project is an explosive leap forward. You're really coding now!



Brainstorm

You can draw any emoji you want — a smiling face, a winking face, a face with rolling eyes — countless options are available! Or you can choose any other object to clone and create a population. Star Wars Stormtroopers? Waldo? Cows? Skittles candy pieces? Any object works, just use your imagination. Time to get coding!


TIP

The code you write for this project will be useful for many games and models you may want to code because so many programs require making, scattering, and moving a population of objects.

Start a New Project

Begin creating your Emoji Explosion program by starting a new project as follows:

1. Open Scratch at `https://scratch.mit.edu/`. If prompted, enable Flash to run Scratch. Log in to the account you created to use Scratch (see Chapter 2).

2. On the Scratch home page, select Create. Or if you're already working in Scratch, choose File➪New from the menu bar.

   A new project opens.

3. Name your program by typing a name in the Project Name field at the top of the Scratch interface.

4. Cut Scratch Cat from the project by clicking the X in the Scratch Cat icon in the sprite area in the lower-right corner.

Add a Backdrop

The *backdrop* is the background color or image that fills the screen of your toy. Add a backdrop as follows:

1. At the Stage, hover over the Choose a Backdrop icon.

2. Click Paint from the pop-up menu.

   The backdrop editor opens at the Backdrops tab. The default costume name for this backdrop is `backdrop1`. You can leave this name, or change it. I changed mine to `night`.

3. Click the Convert to Bitmap button.

4. Select a Fill color.

   I selected black, to make the backdrop look like nighttime.

5. Click the Paint bucket and then click the empty backdrop (the checkerboard region) to fill the backdrop with your selected color. Figure 8-1 shows the finished backdrop.

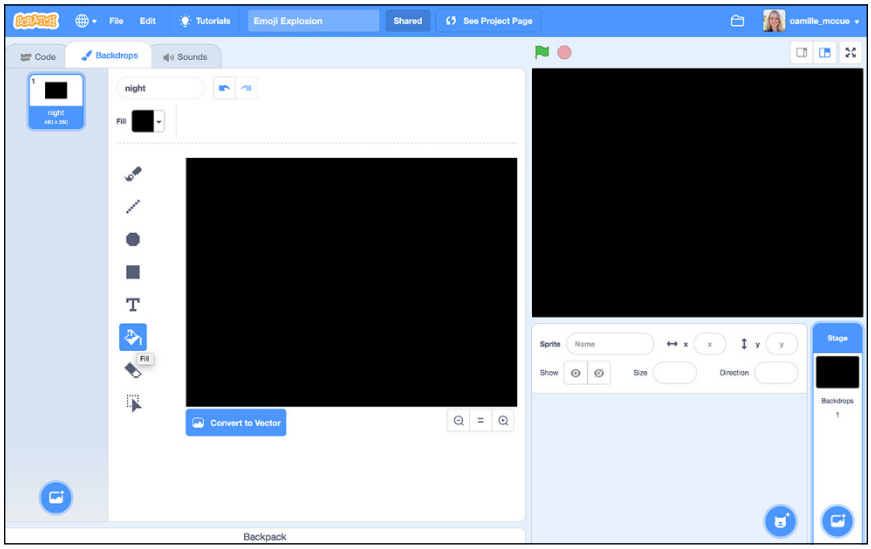


Figure 8-1

Add an Emoji Sprite

Draw an emoji sprite for your animated scene as follows:

1. In the sprite area of the Scratch interface, hover over the Choose a Sprite icon.

2. Click Paint to paint your own sprite costume.

   The sprite costume editor opens at the Costumes tab. The default costume name for this sprite is `costume1`. You can leave this name, or change it. I changed mine to `yellow`.

3. Use the drawing tools to draw your emoji (see Figure 8-2).

   The sprite appears on the stage. Don't worry about its size because you can adjust that later.

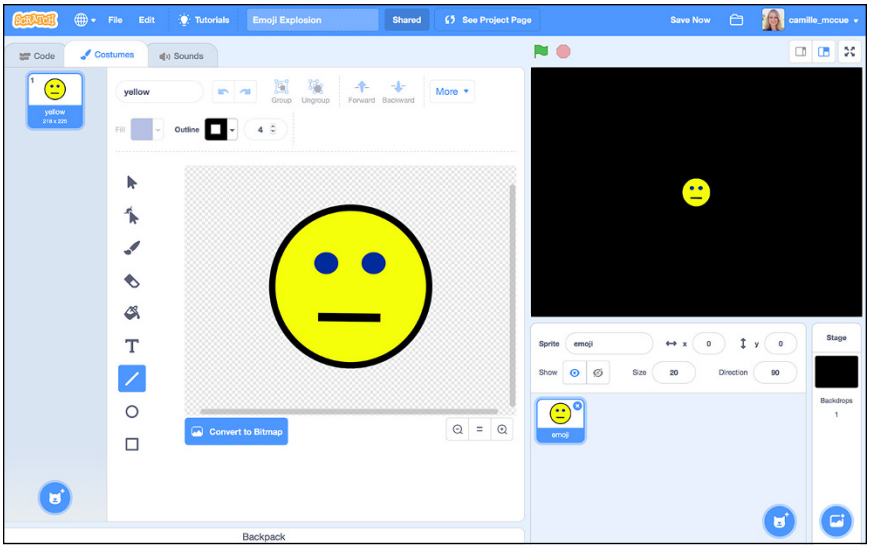


Figure 8-2

4. In the sprite attributes editor in the bottom-right corner of the Scratch interface, edit the name of the sprite.

   The default name is `Sprite1`. I named my sprite `emoji`.

5. Still in the sprite attributes editor, resize your sprite by typing a new number in the Size field above the sprite.

   The default size is `100`. I changed the size of my emoji to `20`.


TIP

Tinker with the drawing tools and the fill colors in the sprite costume editor to see how each can be used to produce different features for your emoji costume.


REMEMBER

If you add a sprite and then decide you don't want it, cut it by clicking the X in its icon.

Code the Stage to Play a Sound

Code the stage to play a sound throughout your animated scene. You can choose which sound you want, and set it to play start to finish, over and over, as the animation runs.

1. Click the icon of your backdrop in the stage area.
2. On the Code tab of the Scratch interface, select the Events icon. Drag a `when green flag is clicked` command to the Code workspace.
3. Select the Control icon. Drag the `forever` loop command to the Code workspace, and attach it to the previous command.

   The `forever` command is one of four loop-type control commands. The other three are `repeat`, `repeat until`, and `wait until`. See Table 8-1 for details. (See the "Repeat Loops" sidebar for more information.)
4. Select the Sound icon. Drag the `play sound until done` command to the Code workspace and attach it inside the `forever` event command.
5. Click the Sounds tab to open the sound editor for the stage.
6. Click the X in the corner of the Pop sound icon to delete it because you won't be using this sound.
7. Click the Choose a Sound icon in the lower-left corner of the Scratch interface.

   The sound library appears on the Choose a Sound screen.

8. Click the icon for the sound you want to add to the stage.

   I selected Boop Bing Bop, as you can see in Figure 8-3. This is a wacky audio track that sounds like emojis bouncing around!

Figure 8-3

9. Return to the Code tab.

10. Click the tab in the `play sound until done` command and select the sound you just added.

Table 8-1 Scratch Control (Loop) Commands

Command	Event
`forever`	The code block runs forever, until the program stops.
`repeat` *`number`*	The code block runs *`number`* times.
`repeat until` *`condition`*	The code block runs until *`condition`* is met.
`wait until` *`condition`*	The code block runs after *`condition`* is met.

Now, when the end user clicks the green flag, the sound you added to the stage will play, looping forever in the background. Figure 8-4 shows the completed code on the stage.

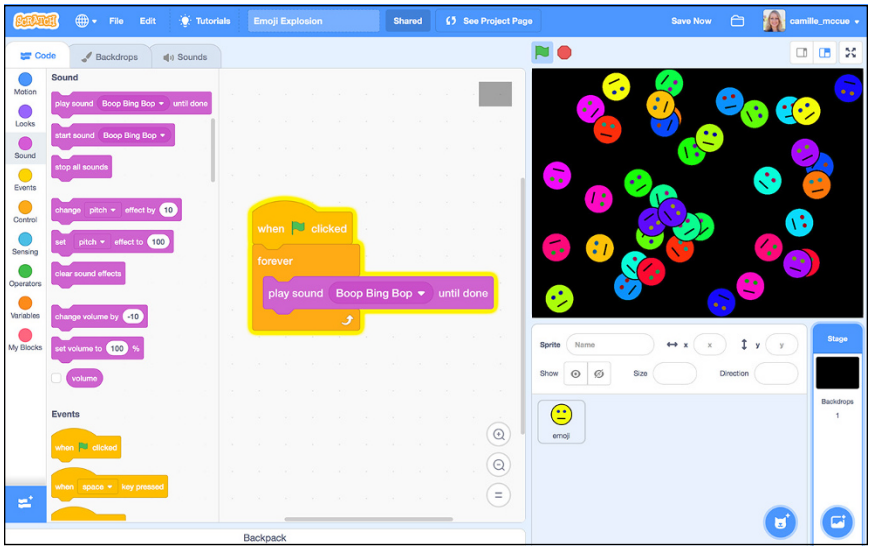


Figure 8-4

REMEMBER

When working on the stage or on a sprite, you see only the code associated with that object. Don't let this panic you — you haven't lost any code!

Code the Green Flag for the Emoji Sprite

The emoji sprite you created should follow a simple program. It should show itself onscreen, clone itself to make more emojis, and then hide.

Write the green flag program for the emoji sprite by following these steps:

1. Select the icon of the emoji sprite in the sprite area.

2. On the Code tab of the Scratch interface, select the Events icon. Drag a `when green flag clicked` command to the Code workspace.

Cloning and Inheritance

Cloning means making an exact copy of an object. Some people have heard of cloning in the animal world. Back in 1996, at the University of Edinburgh, a sheep named Dolly was born, the first mammal successfully cloned from an adult cell. In programming, cloning allows a coder to quickly create as many copies as needed of an object, such as a sprite.

Cloning takes advantage of a programming idea called *inheritance.* Many programming languages use inheritance so that new objects can be easily created from parent objects. A clone *inherits* the attributes (costume, size, position, and direction) and all code of its parent sprite. Also, after you create clones (also known as *child* objects), you can change them to add or remove attributes or bits of code. In this way, a child object can be *mutated* (changed to do something different) from its parent object.

3. Select the Looks icon. Drag a `show` command to the Code workspace, and attach it to the previous command.

   The emoji sprite must be displayed to be cloned. (This is an issue only if at some point you hide the sprite, which you will do at the end of your main program.)

4. Select the My Blocks icon. Click the Make a Block button to create a new code block.

   The Make a Block dialog box opens.

5. Name this new code block `makeEmojis`, as shown in Figure 8-5, and click OK.

   The new block header is added to your workspace, and the new block command, `makeEmojis` is added to your commands in the My Blocks category.

You haven't defined what the `makeEmojis` code block does; you do that in the next section. But isn't it neat that you can write your main program without worrying about that detail just yet?

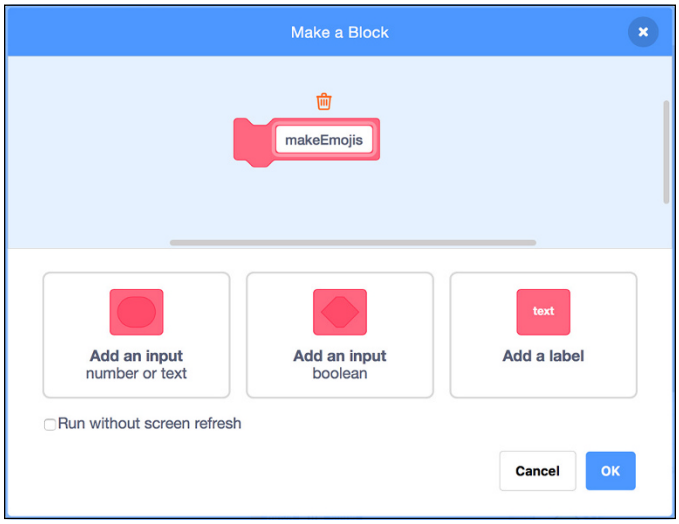


Figure 8-5

6. From the My Blocks category, drag the `makeEmojis` command to the Code workspace, and attach it to the `show` command.

7. Drag a `hide` command to the Code workspace, and attach it to the `makeEmojis` command.

   After the emoji sprite runs `makeEmojis`, it hides. You hide it so that you don't have to write commands to make it move — you'll have plenty of emoji clones moving around the screen already.

See Figure 8-6 for the complete `green flag` code for the emoji sprite.

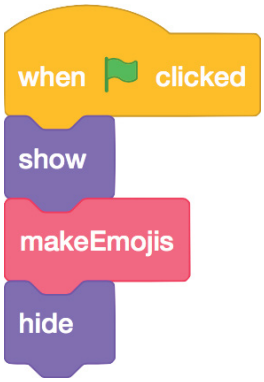


Figure 8-6

Code the `makeEmojis` Block

You created a `makeEmojis` code block header and command tile. Now write the code for that block. The `makeEmojis` code block clones the parent emoji sprite to create many emojis exploding all over the screen.

A *clone* has all of the attributes (costume, size, position, and direction) of its parent sprite and can be made using a `clone` command in the Control category. You can make lots of clones by putting a `clone` command in a simple loop. The `repeat` command (which is also in the Control category) is a simple loop that lets you set the number of times you want it to execute.

Write the code for the `makeEmojis` code block as follows:

1. Work at the `define makeEmojis` code block header in the workspace.

2. Select the Control icon. Drag a `repeat` command to the Code workspace, and attach it to the code block header. Type a number in the `repeat` command.

   This number is how many times the repeat command will execute. I set mine to `40`.

3. Drag a `create clone of` command to the Code workspace, and attach it inside `repeat`. Set the `create clone of` command to `myself`.

4. Select the Looks icon. Drag a `change color effect` to the Code workspace, and attach it to the `create clone` command, inside `repeat`. Type a number in the `change color effect` command.

   This number is how much the color of a clone changes each time the `repeat` command executes. I set the change to `10`. A small number makes slight color changes, and a large number makes bigger color changes. The `define makeEmojis` code block is now complete. See Figure 8-7.

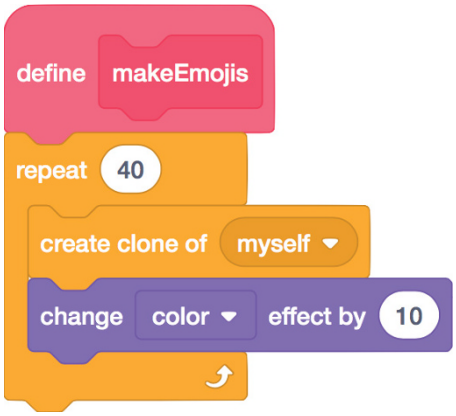


Figure 8-7

Code `when I start as a clone` for the Emoji Sprite

When the green flag program of the emoji sprite runs, it creates clones. The clones need instructions! Each clone needs to follow one instruction: explode!

Write the `when I start as a clone` program by following these steps:

1. Select the emoji sprite.
2. Still working in the Code tab of the Scratch interface, select the Control icon.

   Drag a `when I start as a clone` command to the Code workspace.

   The command you attach to this event will execute each time a new clone is created.
3. Select the My Blocks icon. Click the Make a Block button to create a new code block.

   The Make a Block dialog box opens.
4. Name this new code block `explode` and click OK.

   The new block header is added to your workspace, and the new block command, `explode`, is added to your commands in the My Blocks category. The `makeEmojis` command is there, too! (See Figure 8-8).

   You haven't defined what `explode` does; you do that in the next section. But now you can use the `explode` command.

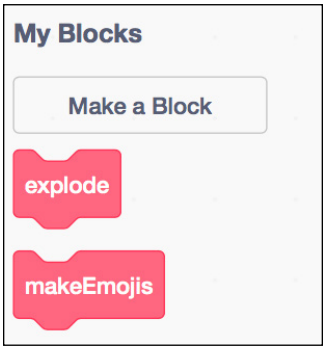


Figure 8-8

5. From the My Blocks category, drag the `explode` command to the Code workspace, and attach it to the `when I start as a clone` command.

See Figure 8-9 for the complete `when I start as a clone` program for the emoji clones.

Figure 8-9

Code the `explode` Block for the Emoji Clones

You created an `explode` code block header and command tile. Now you need to write the code to provide instructions for `explode`. This code should make each clone scatter to a random position, point in a random direction, and move.

Follow these steps to write the code:

1. Work at the `define explode` code block header in the workspace.

2. Select the Motion icon. Drag a `go to random position` command to the Code workspace, and attach it to the code block header.

3. Drag a `point in direction` command to the Code workspace, and attach it to the previous command.

4. Select the Operators icon. Drag a `pick random` command to the `value` field of the `point in direction` command. Set the range of direction angles from `0` to `360`.

5. Select the Control icon. Drag a `forever` command to the Code workspace, and attach it to the previous command.

   The code block you will build inside this command will execute forever until the Stop button is clicked.

6. Select the Motion icon. Drag a `move steps` command to the Code workspace, and attach it inside `forever`. Set the number to steps by typing in number.

   The larger the number, the faster each emoji clone moves. I set mine to a value of `10`.

7. Drag an `if on edge, bounce` command to the Code workspace, and attach it inside the `forever` command.

   This command causes an emoji clone to bounce off the edges of the screen and continue moving.

The `explode` code block is now complete, and should look like Figure 8-10.

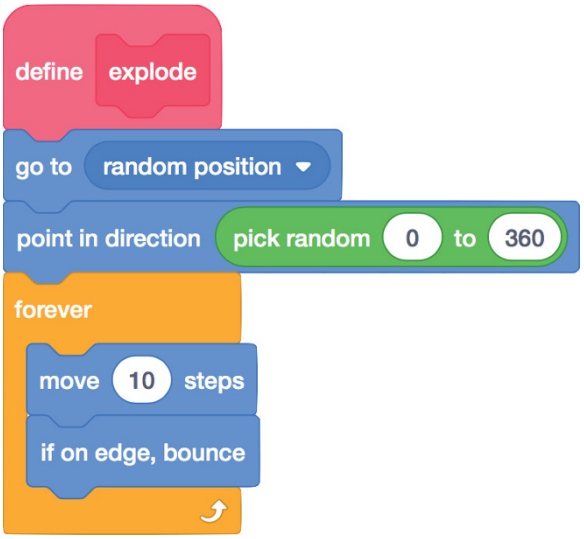


Figure 8-10

Figure 8-11 shows all the code for the emoji sprite. Note that the code for the stage and for the emoji sprite both run when the green flag is clicked. This means they run in parallel (see Chapter 4) — the music on the stage runs forever while the emoji clones move and bounce forever!

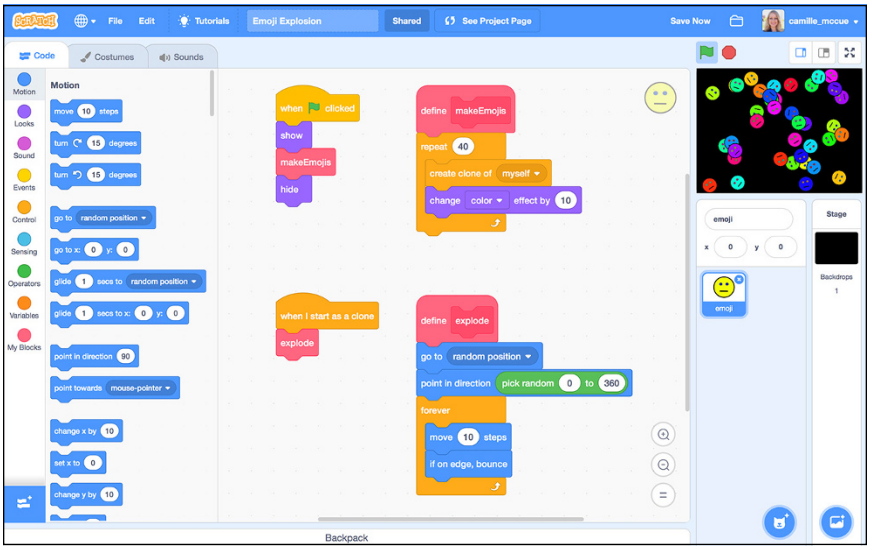


Figure 8-11

Save, Test, and Debug Your Program

Name your project by typing in the Title field at the top of the Scratch interface. As you work, Scratch automatically saves your program in the cloud, so you don't have to take any special actions to save your work.

Test your program and fix any bugs to ensure that it works the way you want it to. (See Chapter 3 for help on debugging Scratch programs.)

Share Your Program with the World

After your program operates perfectly, it's time to share it! Set the status of your program to Share, and then add to your project page a description of your program and directions on how to run it. See Chapter 19 for details on sharing your programs.

Enhance Your Animated Scene

Consider enhancing your Emoji Explosion animated scene with new features:

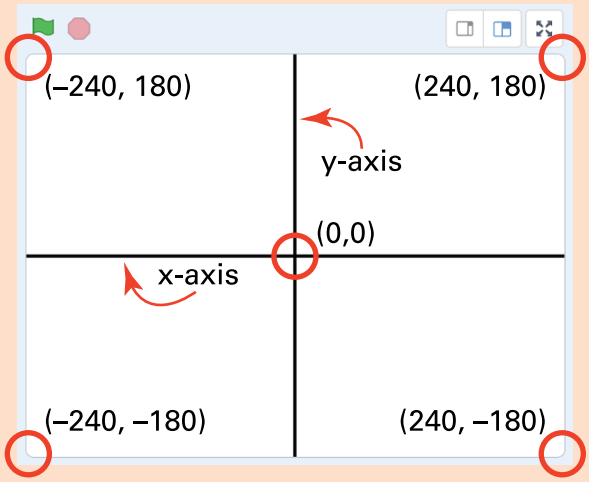
- **New emoji costumes:** Just open the Costumes tab and draw a new costume.
- **New sounds:** Add different sounds to the backdrop.
- **New backdrop:** Instead of painting a simple, solid backdrop for the stage, select the Choose a Backdrop icon to add an exciting location where your emoji can bounce around!

Setting Position

Coordinates are the mathematical way of naming a position on a graph. In two dimensions, like your screens in Scratch, App Lab, and MakeCode for micro:bit, coordinates are described as a pair of values (x, y). The first number is the x-coordinate (position left to right) and the second number is the y-coordinate (position top to bottom).

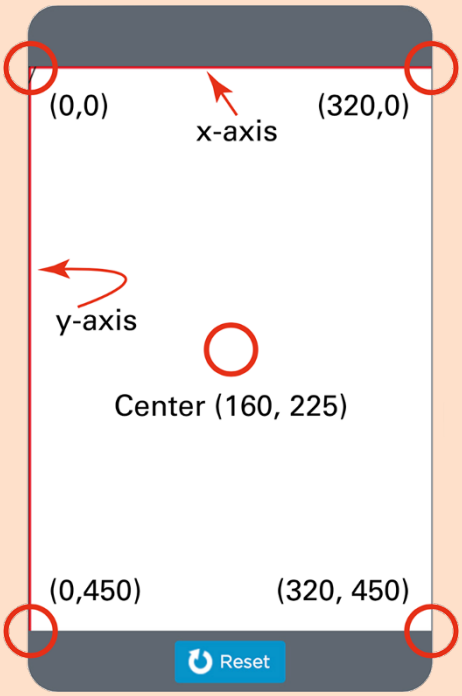
In Scratch, the coordinate (0,0) is called the origin and is located at the center of the screen. The x-axis runs left to right across the screen, with x-coordinate values ranging from –240 at the far left to 240 at the far right.

The y-axis runs top to bottom down the screen, with y-coordinate values ranging from 180 at the very top down to –180 at the very bottom. See the figure to locate the x-axis, the y-axis, and the coordinates of the origin and each corner of the screen in Scratch.



When working in Scratch, you can set the position of a sprite using the `go to x: y:` command. For example, to set the position of a hero in a game at the center of the screen, you could use the command `go to x: 0 y: 0`. To place a sprite at a random position, you can use `go to random position`.

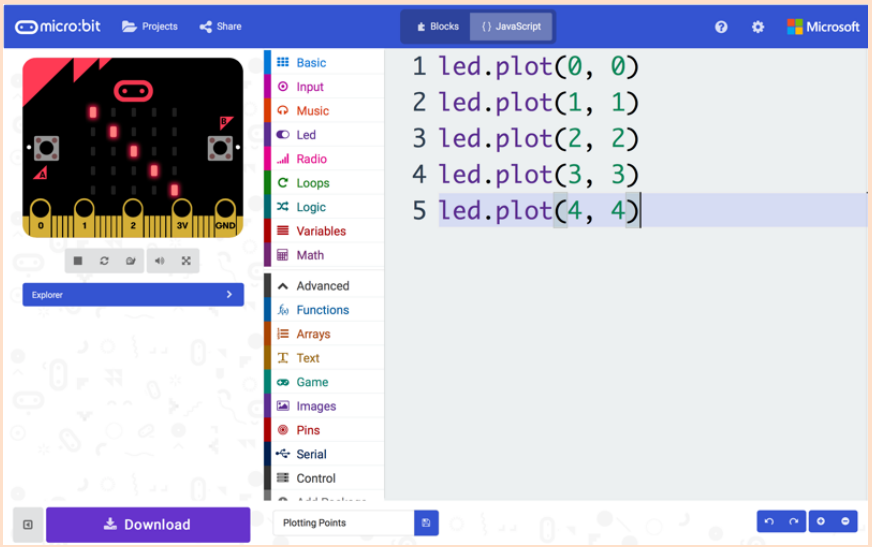
In App Lab, the screen is set up so that the origin — the coordinate (0,0) — is at the top-left corner of the screen. The x-axis runs left to right across the screen, with x-coordinates ranging from 0 at the far left to 320 at the far right. The y-axis runs top to bottom down the screen, with y-coordinates ranging from 0 at the top of the screen to 450 at the bottom. The center of the screen is located at the coordinate (160, 225). The figure shows the x-axis, y-axis, center, and coordinates of each corner of the screen in App Lab.



In MakeCode, you can set positions on the micro:bit screen — a very small LED grid with just 25 points. The micro:bit is set up so that the origin — the coordinate (0,0), — is at the top-left corner. The x-axis runs left to right across the screen, with x-coordinates having values from 0 to 4. The y-axis runs top to bottom down the screen, with y-coordinates having values from 0 to 4. Here are the (x, y) coordinates of all the LED positions on the micro:bit:

```
(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)
```

The figure shows the text-based code (JavaScript) in MakeCode for lighting up a diagonal line on the micro:bit. The “line” runs from the top-left corner at (0, 0) to the bottom-right corner at (4, 4).



Setting Direction

When working in Scratch, you can make a sprite point in a direction by turning it. Directions are indicated by an angle value (the number of degrees) from 0 to 360. Because a full circle consists of 360 degrees, an angle of 0 degrees is the same as an angle of 360 degrees.

The `turn right` and `turn left` commands are *relative turn commands* because they turn the sprite relative to where it currently points. For each of these commands, you must type a number for the angle (the number of degrees) you want the sprite to turn. `Turn right 90` is a "hard right," and `turn left 90` is a "hard left." A turn of 180 in either direction points the sprite in the opposite direction from where it pointed before the turn.

The `point in direction` command is an *absolute turn command* because it causes the sprite to point to the selected heading regardless of where it was pointing before the turn. So, `point in direction 0` is the direction north, `point in direction 90` is east, `point in direction 180` is south, and

`point in direction -90` is west. (Note that `point in direction -90` is the same as `point in direction 270`.) See the figures.

`Point towards` points the sprite towards something else on the screen. Options consist of `mouse pointer` or any other sprite on the screen.

Lastly, when setting direction in Scratch, you can decide how you want the sprite to face. The direction indicator has three facing options. The first option (the circle arrow) makes the sprite rotation match its direction. The second option (the mirror image arrows) makes the sprite flip left and right but not up and down. For example, a person sprite would face left or right, but he would not stand on his head! The third option (the circle arrow with the line through it) freezes the look of the sprite: It can still point and move in any direction, but its appearance won't match its heading.

When working in App Lab, MakeCode, and JavaScript in general, be aware that these programs don't make much use of direction commands. (The only exception is the Turtle command category in App Lab.)

Moving

Motion can be coded by making an object change its position from one coordinate to another coordinate, over and over. In Scratch, you can code motion in three ways. The first way is to place a `move` command (from the Motion category) inside a `repeat` command (from the Control category). Big moves, such as `repeat forever [move 50]`, make an object go faster than small moves, such as `repeat forever [move 10]`. You can also set speed by adding a `wait` command (from the Control category) to each move. For example, `repeat forever [move 10 wait 1]` makes an object move faster than `repeat forever [move 10 wait 2]`.

The second way to make a sprite move in Scratch is by using the `glide` command (from the Motion category). Specify the number of seconds for the glide to create smooth motion (great for floating and swimming sprites).

The third way to make an object move in Scratch is by using the `change x by` and `change y by` commands. `Change x by` moves the sprite left

or right (when the command is followed by a negative or a positive number, respectively). `Change y by` moves the sprite up or down (when the command is followed by a positive or a negative number, respectively). The time in which each move runs affects how the user sees the speed of the moving object.

In App Lab, you can create motion by changing the coordinates of an object and specifying how often the move occurs by using the `timedLoop(1000, function() { })` command, located in the Control category. The default time of the timed loop is 1000 milliseconds, but you can set the time to any value you want. Another way you can control motion in App Lab is to change the coordinates of an object and add a wait time between each change by using the `setTimeout(function() { } 1000)` command, also located in the Control category. Again, you set the milliseconds value.

On the micro:bit, motion is accomplished by changing the coordinates of a light on the LED screen and adding a wait time between moves. In MakeCode, several command categories include a wait command, including `pause(100)` in the Basic category.

Simple Repeat Loops

Sequence, selection, and repetition are the three key processes in any computer program (see Chapter 1). Repetition allows sections of code to execute over and over, or *loop*. Coding loops allows you to make your programs more efficient by identifying the code blocks to be repeated, and then identifying how often they will be repeated — you don't have to keep typing the same code over and over again! In these first chapters, the types of loops you code are fairly simple.

Scratch uses four loop structures: `forever`, `repeat`, `repeat until`, and `wait until`. Each loop offers a different way to control when code will be

repeated and how many times it is repeated. `Forever` loops run the code block over and over until the program is finished. These loops are great for ongoing processes such as playing background music in a game. `Repeat` loops specify an exact number of time the loop will run. A `repeat` loop is useful when you know exactly how many times you want a loop structure to execute, such as cloning 5 players for a basketball team, or counting down a 60-second timer in a game. The `repeat until` loop executes until a condition is met, and `wait until` delays execution until a condition is met.

Every programming language, including App Lab and MakeCode, uses loops. App Lab offers timed loops, a `while` loop (which executes while a condition is met), and a `for` loop. MakeCode features a `forever` loop, a `while` loop, and a `for` loop. You're probably wondering what a `for` loop is. The `for` loop is a more complex loop structure. It's similar to the `repeat` loop in Scratch but uses a variable to count through the loop. You make use of the `for()` loop after you work with variables. Numbers variables are covered in Chapter 12 and `for` loops are covered in Chapter 17.

New Blocks (aka Functions)

Programming languages have built-in commands called *primitives.* Think of primitives as ingredients for making a birthday cake and the computer program as the recipe.

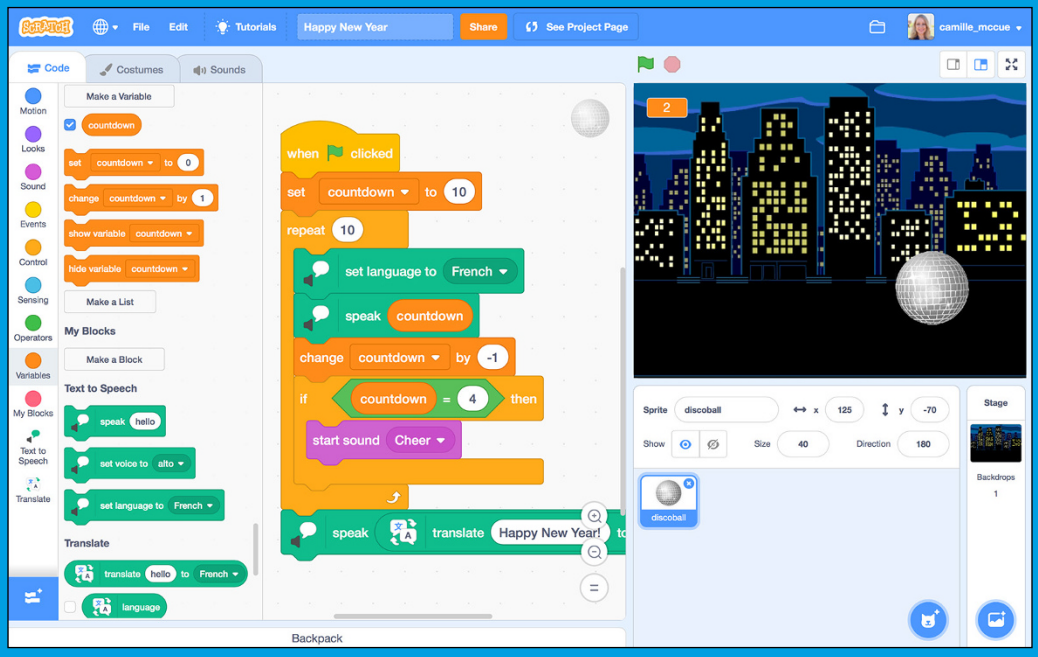
Sometimes, it's easier to understand a computer program or a recipe if we group related instructions. When baking a birthday cake, you make the cake, you make the frosting, and you put them together. The recipe for the birthday cake includes a "frosting" ingredient, but the instructions for making frosting might be on the next page. Instead of explaining all the steps for making frosting in the middle of the cake program, the frosting ingredient represents a smaller program with ingredients and instructions of its own.

By representing this smaller part of the birthday cake with a new name ("frosting") and instructions for making the frosting, the entire recipe-writing process is simplified. In programming languages, the smaller "frosting" program is called a *sub-program*. Scratch calls these sub-programs *blocks,* while App Lab and MakeCode call them *functions.* (Older programming languages called them *procedures.)* You can create as many new blocks (or functions) as you want and then include them in your programs. However, a new block exists only within the program where you made it. Also, the new block executes exactly the same way every time you use it.

You can customize some new blocks (functions). For example, you can include one or more pieces of information, called parameters, with the function. The *parameter* gives extra detail about how to run the function. For example, the `frosting` recipe function might include a parameter that indicates how many servings the recipe will make — `frosting(8)` could represent a function that produces frosting for eight servings of cake. (Yum!)

Part 4

Variables, Simple Conditionals, and I/O



CHAPTER

14

Light Theremin

A *theremin* is an odd electronic instrument, patented in 1928 and known for its spacey sounds. A musician plays a theremin without ever touching it! To make music notes, she moves one hand into different positions near a metal rod sticking out of the top of the theremin. To adjust the loudness of each note, she moves her other hand near a metal loop sticking out of the side of the theremin.

https://en.wikipedia.org/wiki/Theremin#/media/File:Theramin-Alexandra-Stepanoff-1930.jpg

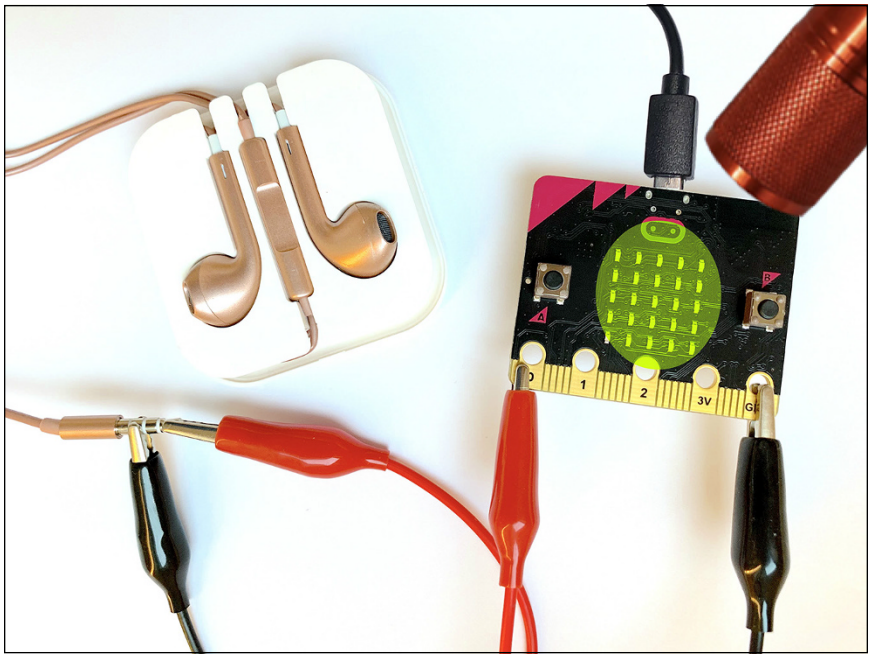
In this project, you code and play your own version of a theremin using MakeCode for micro:bit. You make music with your theremin by moving your hand or shining a flashlight to control the quantity of light that falls on a light sensor. The light sensor on the micro:bit can read values from 0 (dark) to 255 (bright light). The code reads the input light level (a variable) and then uses advanced conditional statements to decide which sound to play. These more advanced conditionals, called *if-then-else if-else* commands, allow your program to branch to many different decisions based on different variable values.

You can then transfer your code to a micro:bit, attach a few wires and some earbuds and have a real working theremin!

The LED light grid on the front of the micro:bit acts as a light sensor and light level meter for the device. LED lights are *directional*; they light up only when they are connected to an electrical circuit in one direction. When they are connected in the opposite direction, they can perform the opposite process: They take in light energy and produce a tiny bit of electricity. So in this project, the LEDs take in the light shining on them and then produce an electrical signal to tell a device to play a sound!

Brainstorm

The MakeCode interface includes commands for producing all the notes from low C to high B. Even if you don't know anything about music, you can select any notes you want your theremin to play on the micro:bit. If you do know a little about music, you might choose to select notes that form a chord, such as middle C, E, G, and high C, which form the C major chord. Regardless of what musical notes you choose to code, your finished theremin will look something like Figure 14-1. Get coding by following these steps.



File:Flashlight CleanBackground.png - Wikimedia Commons

Figure 14-1

Start a New Project

Begin creating your theremin gadget by starting a new project as follows:

1. Open MakeCode for micro:bit at `https://makecode.microbit.org`.

2. On the micro:bit home page, click the big New Project button in the middle of the screen.

   A new project opens and displays the workspace.

3. Name your project by typing a name in the Project Name field at the bottom of the micro:bit interface.

4. Click the Save button next to the Project Name field to save your project.

Code the First Sound Conditional

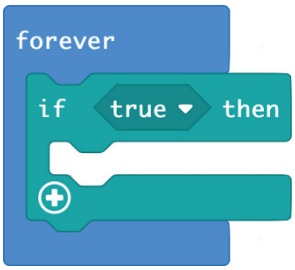
Write your code for the Light Theremin project in the workspace. You can work in Blocks mode or JavaScript text mode. The example shows code written in Blocks mode.

You code the theremin to run its code forever, without a starting event. Follow these steps:

1. Keep the `forever` command in the workspace. Drag the `on start` command back to the command area, or else click it and press the Delete key on your keyboard.

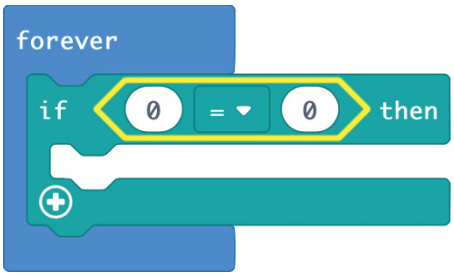
   The code block you'll be creating will run forever as soon as you click the Play icon (the green triangle below the simulator). On a physical micro:bit, the code will run as soon as it is transferred to the micro:bit.

2. Select the Logic category of commands, and drag the `if true then` command to the workspace, placing it inside the `forever` command. The command looks like this:



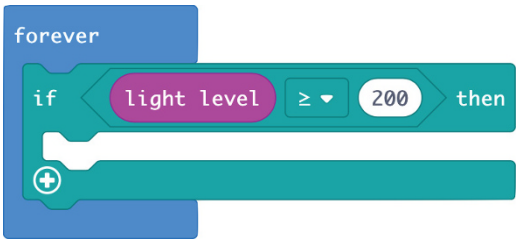
3. From the Logic category of commands, drag the `0 = 0` command to the workspace. Make the `0 = 0` command the condition of the `if true then` command (that is, replace `true`).

   The *condition* is the `if` part of the `if true then` command. Note that the hexagon shape of the `0 = 0` command fits inside the `if` condition.

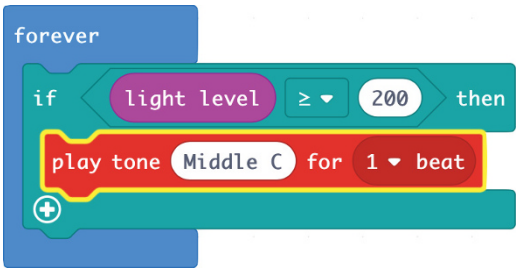


4. Change the `0 = 0` condition to `light level ≥ 200` as follows. From the Input category of commands, drag the `light level` variable to the first `0`, replacing it. Click the tab where the equals sign (`=`) is located and change the operator to `≥`. Type `200` in the field following the `≥` operator.

   The command looks like this:



5. Complete the `if-then` conditional by adding a *consequence* as follows. From the Music category of commands, drag the `play tone Middle C for 1 beat` command inside the `if-then` conditional command. The command looks like this:



In the `play tone Middle C for 1 beat` command, click the note name (`Middle C`). This opens a piano keyboard where you can select a new note, as shown in Figure 14-2. Select `High C`.

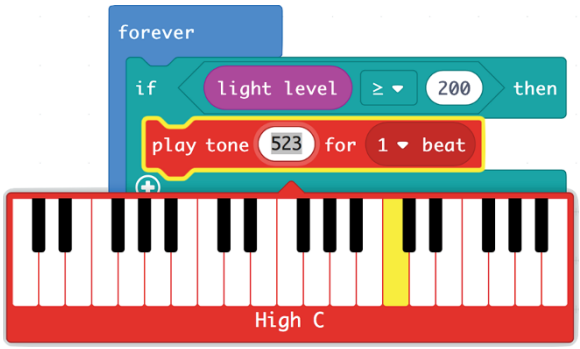


Figure 14-2

Test how your coding is working so far. Click the green Play icon on the simulator. Note the circle-shaped light meter, which measures the quantity of light falling on the micro:bit, in the upper-left corner. The default light level for the simulator is 128.

Using the mouse pointer, click and hold down on the dividing line where the yellow touches the gray inside the circle. Drag the line up and down to manually decrease and increase, respectively, the value of the light sensor variable. (The simulator is not changing the light level; it simply allows you to test how the micro:bit behaves under different lighting conditions.) If you reach a value of 200 or more, the micro:bit simulator plays the high C note.

Note that the simulator shows the micro:bit sending an electrical signal to Pin 0, and that Pin 0 is connected to an external speaker. The simulator shows this because the physical micro:bit board has no built-in speaker, but it can set a voltage at a pin (connection point), which causes a connected speaker to play a sound. See Figure 14-3.

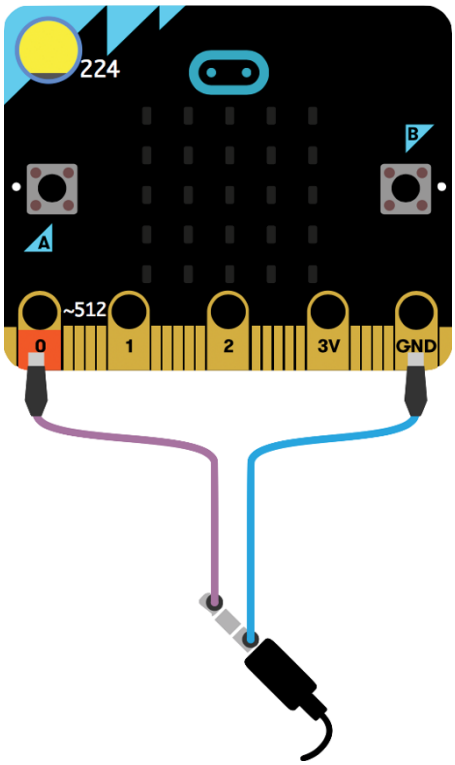


Figure 14-3

Code More Sound Conditionals

Continue writing your Light Theremin code in the workspace. Code more sound conditionals, with each sound conditional responding to a different light level.

Note that you will use a series of `if–then–else` conditionals to decide which sound to play. See the "Advanced Conditionals" sidebar.

1. At the `if then` conditional you have already placed in your code, click the small plus sign (+) located in the lower-left corner of the conditional.

   This adds an `else` command to the conditional. You're not done yet!

Advanced Conditionals

Conditionals can be simple `if–then` commands (see Chapter 13), or they can be more advanced `if–then–else` commands. Depending on the Boolean value (`true` or `false`) of a condition, a path is selected for the next command to be executed in the program.

The `if–then–else` command is a common command for making your code select one of two different paths to run. (See Chapter 1 for details on *selection.*) For example, when playing a game app with a timer, *if* there is time left on the clock, *then* you can keep playing, *else* the game is over. By expanding from an `if–then` to an `if–then–else` conditional, you can ensure that you create a specific outcome for both possible states of the condition.

Here's another example: "*If* it is cold outside, *then* I will wear a coat, *else* I will wear a swimsuit." This structure provides you more control over the consequences of the decision. When the condition is `true`, you make one decision, and when the condition is `false`, you make a different decision. Without the `else`, you would have exited the conditional without taking any action. For example, you don't know what to wear if it isn't cold outside!

When needed, conditionals can be even more complex, taking the form of `if–then–else if–else` commands (as in the Light Theremin project). This structure lets you create many alternative outcomes by adding as many `else if` statements as you need. For example, "*If* it is less than 40 degrees, *then* I will wear a coat; *else if* it is less than 65 degrees, I will wear a light jacket; *else if* it is less than 90 degrees, I will wear shorts and a T-shirt; *else* I will wear a swimsuit." Note that the `else` at the end of your decision is executed only when all of the `if` and `else if` statements are `false`. The `else` is a catchall in your conditional because it catches all the cases that are not caught by the previous statements. It is the "outcome of last resort."

Putting together many `else if` statements in a conditional allows you to create a *conditional sieve:* the different levels of `else if` statements allow you to bypass a condition that is `false` and get "caught" by a condition when it is `true`. Once caught, a statement can't be caught again;

the consequence of the `else if` that caught the statement is executed and the program exits the conditional sieve. (You may have seen a different type of sieve at a construction site: There, sieves are used to sort rocks of different sizes.) Both the Light Theremin program and the weather clothing example are structured as conditional sieves.

2. At the `if then else` conditional, click the small plus sign (+) three more times.

   You now have one `if-then`, three instances of `else if`, and one `else` as shown in Figure 14-4.

Figure 14-4


TIP

When writing a conditional in MakeCode, you can click the plus sign (+) on a conditional to add new conditions to it, and you can click the minus (–) key to remove conditions you previously added.

3. From the Logic category of commands, drag the `0 = 0` command to the workspace. Place it inside the condition of the first `else if-then` command.

4. Change the `0 = 0` condition to read `light level ≥ 150`. Follow these steps:

   a. From the Input category of commands, drag the `light level` variable to the first 0, replacing it.

   b. Click the tab where the equals sign (=) is located and change the operator to ≥.

   c. Type `150` in the field following the ≥ operator.

5. Complete the `else if–then` conditional by adding a *consequence* as follows:

   a. From the Music category of commands, drag the `play tone Middle C for 1 beat` command inside the `if true then` conditional command.

   b. In the `play tone Middle C for 1 beat` command, click the note name (`Middle C`). This opens a piano keyboard where you can select a new note. Select `Middle G`.

6. Repeat Steps 3 through 5 for the remaining two `else if–then` conditionals by using these conditions and consequences:

```
else if light level ≥ 100 then play tone Middle E
for 1 beat

else if light level ≥ 50 then play tone Middle C
for 1 beat
```

7. The `else` conditional has no condition; its consequence executes if none of the other conditionals are `true`. From the Music category of commands, drag the `rest (ms) 1 beat` command inside the consequence of `else`.

   When the light level on the micro:bit is very low, less than a value of 50, none of the other conditionals are `true`. That's when this `else` consequence executes. It "plays" a rest, which is no music sound.

Your completed code should look like Figure 14-5.

Figure 14-5

Here is the complete code in a text-based (JavaScript) format:

```
basic.forever(function () {
    if (input.lightLevel() >= 200) {
        music.playTone(523, music.beat(BeatFraction.Whole))
    } else if (input.lightLevel() >= 150) {
        music.playTone(392, music.beat(BeatFraction.Whole))
    } else if (input.lightLevel() >= 100) {
        music.playTone(330, music.beat(BeatFraction.Whole))
    } else if (input.lightLevel() >= 50) {
        music.playTone(262, music.beat(BeatFraction.Whole))
    } else {
        music.rest(music.beat(BeatFraction.Whole))
    }
})
```

Save, Test, and Debug Your Program

Click the Save button at the bottom of the screen to save your program. Test your code by clicking the green Play icon on the simulator and then dragging the light level line up and down on the light meter. As you artificially adjust the amount of light falling on the micro:bit, you should hear the different notes (and the rest) you coded.

Fix any bugs to ensure that your Light Theremin toy works the way you want it to. (For details on debugging micro:bit programs, see Chapter 3.)

IoT and Sensors in Circuits

With the world becoming more connected than ever, you've probably been hearing a lot about the Internet of Things, or IoT. Not only are your computer, tablet, and phone able to communicate over the Internet, other devices in your home can too, including your television, security cameras, appliances, and thermostat. Computer programs use sensors on these devices to record a TV program at a certain time, start streaming live video of the person currently ringing your doorbell, turn on the coffee pot before you wake up, and adjust the household temperature when no one is home.

Similarly, wearable electronics have code-controlled sensors that can track your health and wellness by monitoring your sleep patterns, record how many steps you take each day, determine your heart rate, and even measure your blood sugar levels.

Even cities make use of IoT by connecting sensors on infrastructure — such as transportation, power production, and water systems — to the Internet. These sensors respond to computer programs that control how they perform their jobs, including changing the timing of traffic lights, balancing power distribution, and adjusting water levels during droughts and floods.

Sensors in electronic circuits perform the same roles as senses in living creatures: They measure information about the world around them. But sensors don't do much by themselves. Sensors require a computer program to understand and "do something" with the information they measure in the same way that your senses (your eyes, ears, and nose) need your brain to make meaning of the information they take in.

The micro:bit board is a simple electronics board with built-in sensors: pushbuttons, a light sensor, a thermometer, a magnetometer (compass), and an accelerometer. See Chapter 2 for additional information on each of these sensors.

The pins can also measure whether or not a circuit (of which the micro:bit board is a component) is complete — in other words, whether electricity is running through the circuit. In MakeCode for micro:bit, the Input category of commands is used to measure information about the micro:bit sensors. You write the code so that the events and conditionals can respond to sensor readings — just like in the Light Theremin project.

Transfer Your Program to the micro:bit

When your code works the way you want it to, you can transfer it to a physical micro:bit. For details on transferring programs to the micro:bit, see Chapter 2.

Continue powering the micro:bit from your computer or attach the portable battery pack to the micro:bit. Use an alligator clip to connect Pin 0 of the micro:bit to the tip of the headphone jack. Use another alligator clip to connect the Ground pin of the micro:bit to the base of the headphone jack. Refer to Figure 14-1.

Put on your headphones, and then change the quantity of light falling on the LED grid. You should hear different notes playing through the headphones as you change the light level.

Share Your Program with the World

If you want, you can share your micro:bit program with others. Set the status of your program to Share, and then copy and paste the link to your project anywhere you want to share it. See Chapter 19 for details on sharing your programs.

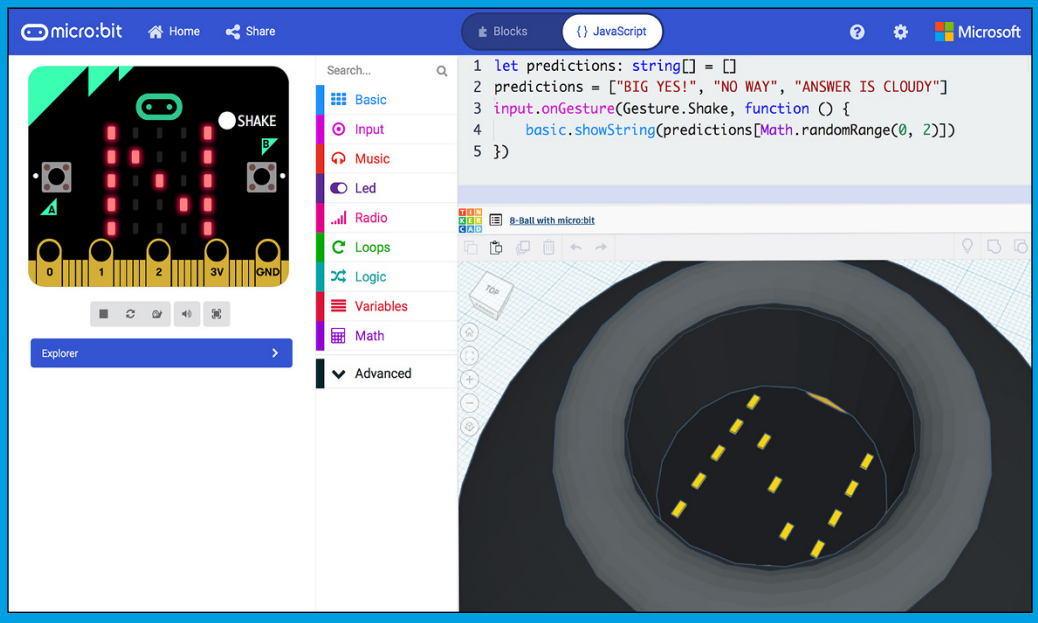
Enhance Your Toy

Consider enhancing your Light Theremin gadget with new features:

- **Additional sound levels:** Add more conditionals to create new sound levels and associated sounds. Can you create an entire scale?
- **Button key shift:** Change the key from C major to a different music key by writing new code. Add an `on button a pressed` event and create a new conditional sieve that uses a new chord progression such as F major.
- **Speaker hardware:** Instead of connecting your micro:bit to headphones, connect it to a speaker. Inexpensive speakers for electronics projects can be purchased online or at hobby shops. Just connect the alligator clips to the lead wires from the speaker and you'll have a theremin everyone can hear, so you can play for an audience!

Part 5

Lists, Loops, and Logic



CHAPTER

15

Magic 8-Ball

The classic Magic 8-Ball is a fun retro toy that gives a randomly chosen answer to a question the user asks. Magic 8-Ball can give 20 answers, with some answers positive ("Big Yes!"), some negative (No Way), and some neutral ("Answer is Cloudy"). The user asks a question, jiggles the ball, and an answer appears!

In this project, you create your own Magic 8-Ball using MakeCode for micro:bit. You first code a list of answers using a new type of variable (called a *list variable*, or *an array variable*). Then you code the micro:bit to respond to a shake by scrolling a randomly chosen answer across the screen. If you want, you can transfer your code to a real micro:bit, and if you're really ambitious, you can design and 3D-print a Magic 8-Ball shell for your toy!

https://commons.wikimedia.org/wiki/File:Magic_8_Ball_-_Instrument_Of_Evil%3F_(2426454804).jpg

Brainstorm

Your MakeCode program can scroll predictions as part of any toy you create. You can just play with your program on the computer screen, or you can transfer your code to a micro:bit and build a toy around the electronics board. That toy can be made of any craft materials you have access to, from cardboard and Styrofoam to wood and 3D filament.

Also, your toy can look any way you want — it doesn't need to look like a Magic 8-Ball. You can create any device to display text predictions for your user. What will you design? A crystal ball? A Zoltar genie machine? A fortune cookie? A closeup of Yoda's head? Tea leaves in a cup of hot tea? You can also write the predictions any way you want, but consider using a combination of "yes," "no," and "maybe" predictions. Figure 15-1 shows the 3D design I created in Tinkercad and then 3D-printed!

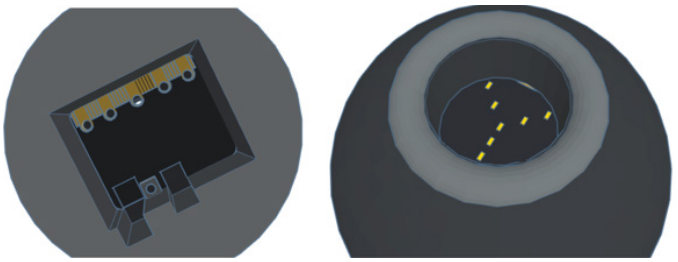


Figure 15-1

Start a New Project

Begin creating your Magic 8-Ball toy by starting a new project as follows:

1. Open MakeCode for micro:bit at `https://makecode.microbit.org`.

2. On the micro:bit home page, click the big New Project button in the middle of the screen.

A new project opens and displays the workspace.

3. Name your project by typing a name in the Project Name field at the bottom of the micro:bit interface.

4. Click the Save button next to the Project Name field to save your project.

Code `on start`

You write your Magic 8-Ball code in the workspace. You can work in Blocks mode or JavaScript text mode.

The example uses code written in Blocks mode to create a list variable containing three predictions. The `on start` event is the starting point for the MakeCode program you will write for the Magic 8-Ball. Follow these steps:

1. Keep the `on start` command in the workspace. Drag the `forever` command back to the command area, or else click it and press the Delete key on your keyboard.

2. In the Variables category of commands, click the Make a Variable button. In the dialog box that opens, type `predictions` (see Figure 15-2) and click OK.

   The `predictions` variable is now added to your available variables.

Figure 15-2

3. Open the Advanced commands, select the Array category of commands, and drag a `set text list to` command to the workspace. Place this command inside the `on start` command. Press the small down arrow at `text list` and select `predictions` from the drop-down menu.

4. Build your list (array) of predictions by typing one prediction into each field of the array.

   You don't need to type the quotes in each prediction — MakeCode adds them automatically to any string you type in a field. (For details on working with strings, see Chapter 10.)

   a. In the first field, replace the `a` by typing a prediction such as `BIG YES!`.

   b. In the second field, replace the `b` by typing a prediction such as `NO WAY`.

   c. In the third field, replace the `c` by typing a prediction such as `ANSWER IS CLOUDY`.

   If you want to add more predictions, add new fields for them by pressing the plus (+) button. Remove unwanted fields from the end of the list (array) by pressing the minus (–) button.

Your complete `on start` code should look like Figure 15-3. When the micro:bit program starts executing, it sets the list variable, `predictions`, to all the string values you typed. In this example, the value of `predictions` at position 0 is `"BIG YES!"` and the value of `predictions` at position 2 is `"ANSWER IS CLOUDY."`

Figure 15-3

In MakeCode — and most programming languages — you start counting list (array) positions at 0. You stop counting list positions at the last item in the list. The last item in the list is at a position that is one less than the number of items in the list. In the example, three items are in the list, but the last item is at position 2.

Simple Lists (Arrays)

A *list* is a variable that represents a group of items stored together in a certain order. *Array* is another word for list. (There are some technical differences between the two, but the explanation of those differences is beyond the scope of this book.) Items in a list (array) are usually called *elements.* Many different types of elements are available, such as numbers, strings, and even other lists.

Programming languages use lists to store related information in some way. For example, a grocery list has items such as broccoli, cheese, and juice boxes. A teacher might have a list of scores earned by students on a test. A school might have a list of classrooms, and for each classroom, and a list of students in that classroom (this is a "list of lists"). For all practical purposes, your list can be as long as you want it to be.

In a block programming language, you name a list in the same way you do any variable, such as groceries, predictions, scores. In a text-based programming language, you might use additional symbols, such as square brackets to show that the variable is a list. Lists usually order their elements starting at the number 0 and counting up to the last element. (But it's important to note that Scratch starts numbering its list elements at 1.) This way, your program can identity the value of an element by its position in the array.

Parallel lists (arrays) are two or more lists that have related information. For example, a teacher might have these arrays:

```
String [] students = ["Alice", "Benito", "Carol", "Dan"];
int [] scores = [87, 93, 94, 80];
```

The value of `students[0]` is `"Alice"` and the value of `scores[0]` is 87. For this classroom of students, the teacher has set student number `0` to `Alice`, so we know that Alice's score on the test is 87.

Note that although the `students[]` array has four students, the last student in the list is at position 3. This is because you start counting the positions in the array at 0, not 1. So the value of `students[3]` is `"Dan"` and the value of `scores[3]` is 80. See Chapter 16 to for an additional project on lists.

Code on shake

The user shakes the micro:bit to reveal a prediction. When the user shakes the micro:bit, the accelerometer sensor detects the movement and runs the code associated with the `on shake` event. (See Chapter 3 and the sidebar on IoT and sensors in Chapter 14 for additional information on micro:bit sensors.) Write this code by following these steps:

1. From the Input category, drag the `on shake` command to the workspace.
2. In the Basic category of commands, drag the `show string` command to the workspace and attach it inside the `on shake` command.
3. Open the Advanced commands, select the Array category of commands, and drag the `list get value at` command to the workspace.
4. Place the `list get value at` command in the field of the `show string` command. Press the small down arrow at `list` and select `predictions` from the drop-down menu.
5. From the Math commands, drag the `pick random` command to the workspace and place it inside the `predictions get value at` field.

6. Set the range of value for `pick random` by typing numbers in the empty fields. Type `0` in the first field, which is the position value of the first item in your array.

   The value of the second field depends on the position of the last item in your array. In my example, I have three predictions, so the range of my `pick random` is 0 to 2 (the three predictions are at position 0, position 1, and position 2 in my array). The second field of your `pick random` has a value of *one less* than the number of items in your array.

   Your complete `on shake` code should looks like Figure 15-4.

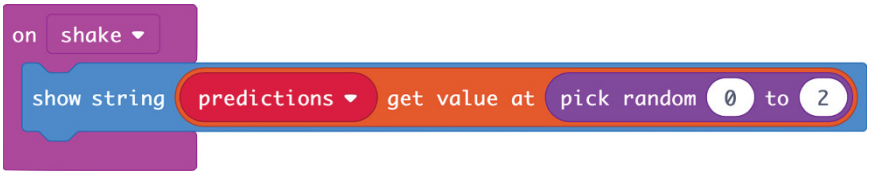


Figure 15-4

When the user shakes the micro:bit, its accelerometer measures motion and triggers the execution of the code block at the `on shake` event. The `on shake` event causes a randomly selected prediction string to be scrolled across the LEDs of the mico:bit. Figure 15-5 shows the micro:bit simulator scrolling a prediction, letter by letter.

The complete code in JavaScript follows:

```
let predictions: string[] = []
predictions = ["BIG YES!", "NO WAY", "ANSWER IS CLOUDY"]
input.onGesture(Gesture.Shake, function () {
    basic.showString(predictions[Math.randomRange(0, 2)])
})
```

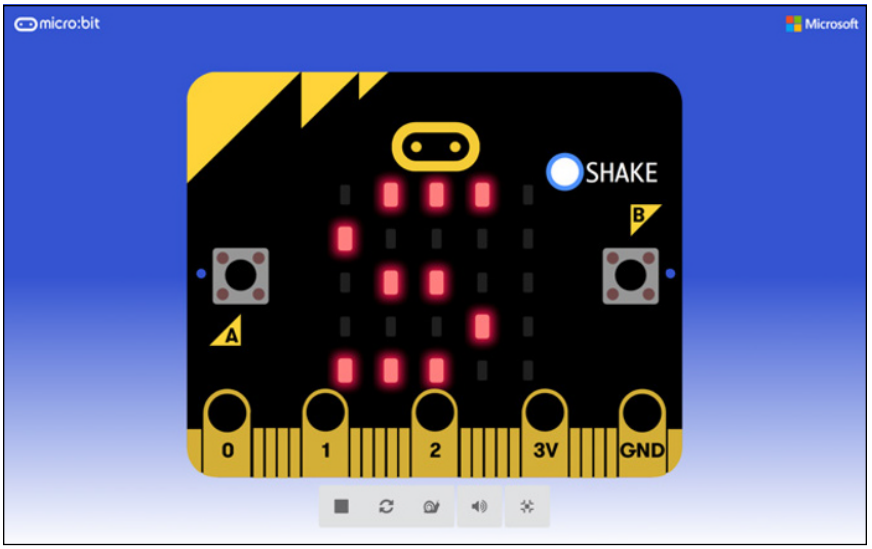


Figure 15-5

MakeCode might place the line showing the string values in the `predictions` array below the event handler (`input.onGesture()`). However, when writing the text-based code, you should write the code as shown here. The standard way of organizing JavaScript code is to place the variable assignments before the event handlers.

Save, Test, and Debug Your Program

Click the Save button at the bottom of the screen to save your program. The program is saved in the cloud and also as a micro:bit .hex file in your Downloads folder.

You test your program in the on-screen simulator by clicking the Shake button (which simulates shaking the micro:bit). Each click should show a prediction from your `predictions` array. The letters in the prediction scroll by, one letter at a time.

Fix any bugs to ensure that your Magic 8-Ball toy works the way you want it to. (See the section in Chapter 3 on debugging micro:bit programs.)

Transfer Your Program to the micro:bit

After you test and debug your code, you can transfer it to a physical micro:bit. For details on transferring programs to the micro:bit, see Chapter 2. After the program is on the micro:bit, you can detach the board from your computer's USB port. Attach the optional battery pack to use the micro:bit and your Magic 8-Ball toy away from the computer.

Share Your Program with the World

You can share your MakeCode for micro:bit program with others. Set the status of your program to Share, and then copy and paste the link to your project anywhere you want to share it. See Chapter 19 for details on sharing your programs.

Enhance Your Toy

Consider enhancing your Magic 8-Ball toy with new features:

- **Add predictions associated with acceleration changes:** Right now, a random prediction of any type is selected from the array in response to `on shake`. You can measure the direction of motion when the micro:bit is shaken and associate a prediction type with the direction of motion.

  For example, you can make positive responses appear when the 8-Ball is shaken along the x-axis, neutral responses for y-axis motion, and negative responses for z-axis motion. That way, you can secretly control the types of responses you get, making it appear that you're truly magical! You'll need to create three arrays to replace your original array, with each array featuring a response type: positive, neutral, or negative. Then you'll need to create some `if-then` conditionals to display one of those response types according to the direction and measure of the micro:bit's accelerometer values. To learn more about measuring accelerometer values, refer to

the micro:bit online documentation at `https://makecode.microbit.org/reference/input/acceleration`.

- **Create a Magic 8-Ball shell to house the micro:bit:** Check out the sidebar on making a cool container for your toy! You can use free design software such as Tinkercad to design the 8-Ball shell. Then you can use a 3D printer (or send your design to a company that does 3D printing) to fabricate your design. I bought some black filament and printed mine on a FlashForge Finder 3D printer. Then I placed my micro:bit and battery pack inside and — voila! — I have my own, custom-made 8-Ball.


TIP

If you choose to print your own 3D designs, a FlashForge Finder single-filament 3D printer runs about $300, and a daVinci Color mini 3D printer costs a whopping $1600. Filament costs approximately $30 per roll (it lasts a long time). If you choose to send out your 3D design for printing, plan on paying around $10 to $30 per pound, depending on the filament materials you request.

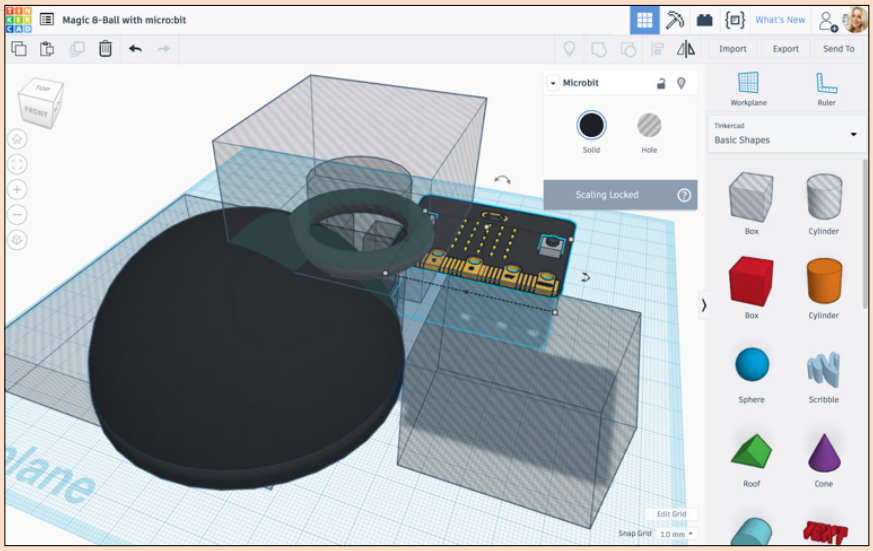
eToys

You've probably noticed many devices named i*Something* or e*Something* these days. The *i* stands for *Internet* and the *e* stands for *electronic.* Devices named i*Something,* such as smart watches and mobile phones, use the Internet in some way. Devices named e*Something* include all sorts of toys that use electronic circuitry in them. Everything from remote-controlled cars to the old-school game of Operation can be considered an eToy.

The micro:bit board can serve as electronics component of a toy, turning it into an eToy. Sensors on the micro:bit can make measurements about its environment. Onboard memory allows it to store code that provides instructions to the device. And external batteries provide the power needed to operate it! But the micro:bit itself is just an electronics board, so as an eToys designer, you need to think about the fancy and fun physical parts you'll need to add to make your eToy say, "Play with me!"

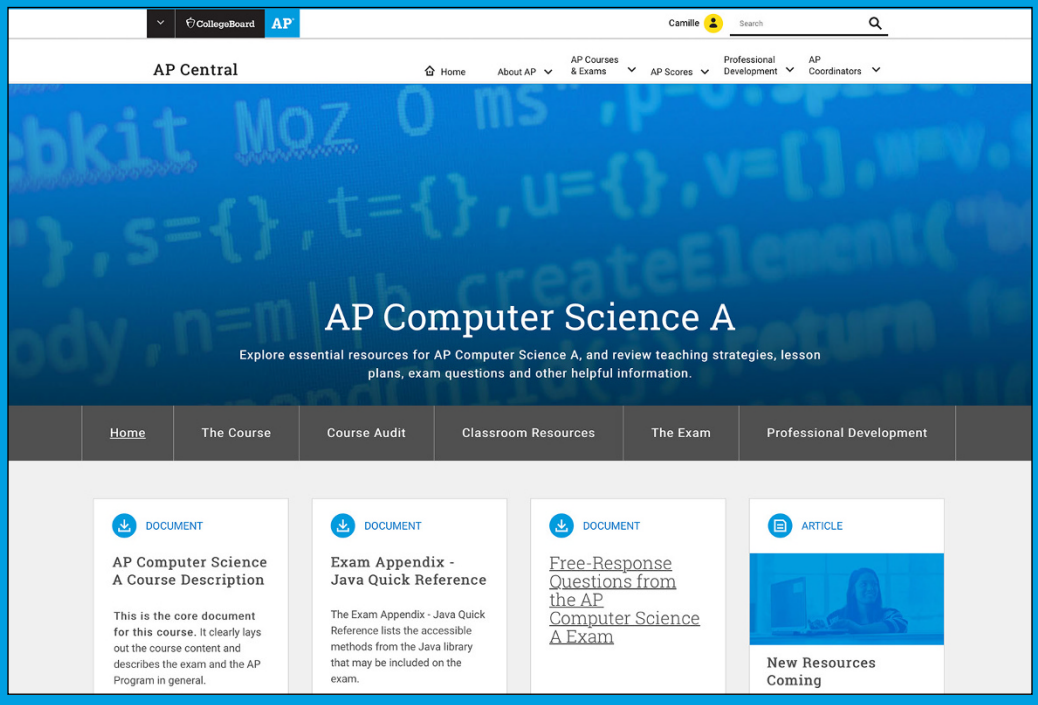
For the Magic 8-Ball toy, you can create a real eToy by making plastic shell and adding the coded micro:bit board and battery pack to it. You can use all sorts of materials to make the shell, from Styrofoam to a Nerf ball to Play-Doh. Another option is to design a shell in a 3D design program such as Tinkercad (`www.tinkercad.com/`) and then print it using a 3D printer.

I created a design at `www.tinkercad.com/things/8QdTqoXzVnl-8-ball-with-microbit/edit`. I first found a micro:bit created and shared in Tinkercad, and then I designed my 8-ball shell around the micro:bit design. You can use this design, "tinkering" with it to make it your own or creating something new. Keep in mind, though, that many 3D printers print in a single filament color, so even if you make your design with lots of colors, it will print in the one color loaded into the 3D printer. I used a FlashForge Finder, but you can find many other 3D printers. And if you don't have a 3D printer, you can send your saved 3D design (as an .stl or .obj file) to a company that will print it for you for a small fee (search online for *3D printing companies*). After you have your shell in hand, just pop in the micro:bit and battery pack (add a bit of tape to hold these in place) and shake to see a prediction in your eToy!



Part 6

Onwards and Upwards



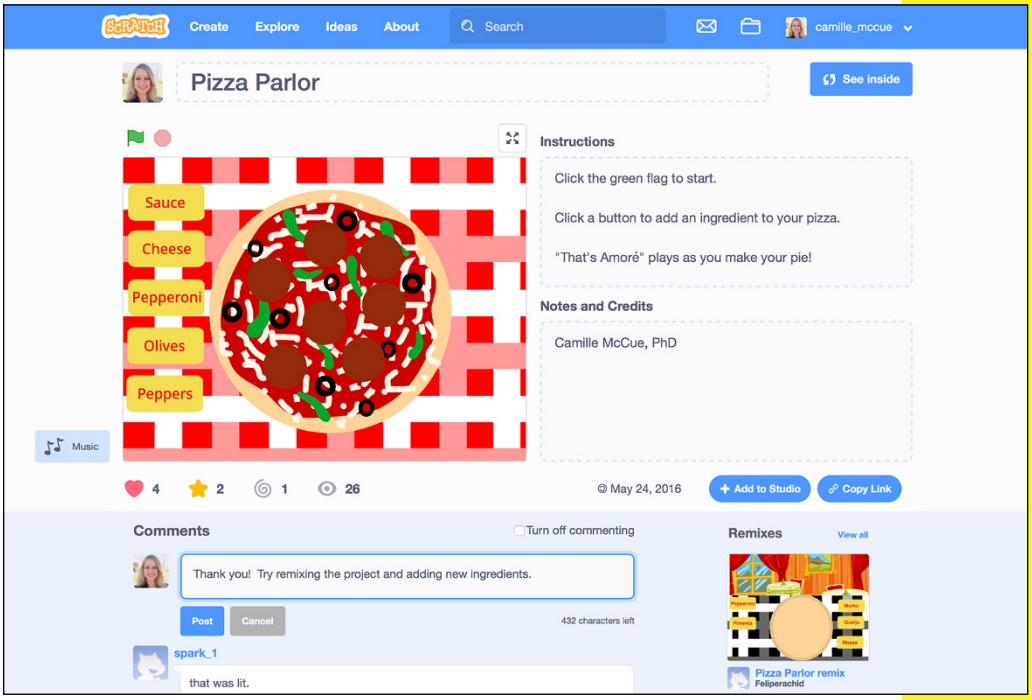
CHAPTER

19

Creating and Sharing

You've tackled a lot of fun projects in this book, but how do you bring your own ideas to life? How do you spark those ideas in the first place?

Unlike what many people think, coding is not a solo activity conducted in a dark room surrounded by caffeinated drinks. Coding is a highly creative — and highly social — process, one that solves real problems. The process of developing a new app, website, or control program for a gadget often brings together programmers, graphic designers, animators, musicians, data scientists, engineers, and advertising professionals. It's not just you and your computer; it's you and a lot of your peers working together towards a common goal!



For now, you don't need an army of people to code cool projects, but you do need a little inspiration to get started. This chapter will help you get your creative juices flowing, as well as get your completed projects out in front of an audience. Sharing your work and engaging in a larger community of coders is one of the best ways to build your skills — and your positive reputation.

Programming Your Own Ideas

You may find it tough to think up something you want to create with code. Or you might have a great idea but then discover that someone else beat you to it. Or maybe you have a cool, revolutionary idea but don't know how to turn something so complicated into code. Every coder on Earth has felt these same feelings and thought these same thoughts — you're not alone!

The key to pushing past these roadblocks is to take concrete action: Design and code something small. Getting a project underway will provide you with something that you can expand and evolve, or use as a springboard for bouncing in a new direction. In this section, you discover some techniques you can use to get creative and get going.

Remixing apps you like

What apps do you enjoy? Arcade games? Tamagotchi pets? Reaction time games? Why do you like them? Can you find something similar in the Scratch or App Lab or MakeCode libraries? In other words, can you find a kid-friendly example of the type of coding you want to do?

If you can find a project that appeals to you, you can *remix* it — start with the existing project and then change it to create your own version, adding new graphics and code however you want. Having a basic project as a starting point can help relieve some of the pressure of trying to create something from the nothing of a blank screen.

Figure 19-1 shows a Scratch project called Scrolling Slopeformer, by im_feeling_itchy. Scratchers are invited to remix the game and add new levels. To remix a project, click the Remix button at the top right of the project page. Your My Stuff folder now contains a copy of the project, for you to modify however you want. An *attribution* — a thank-you noting the original maker of the project — is added to your copy.

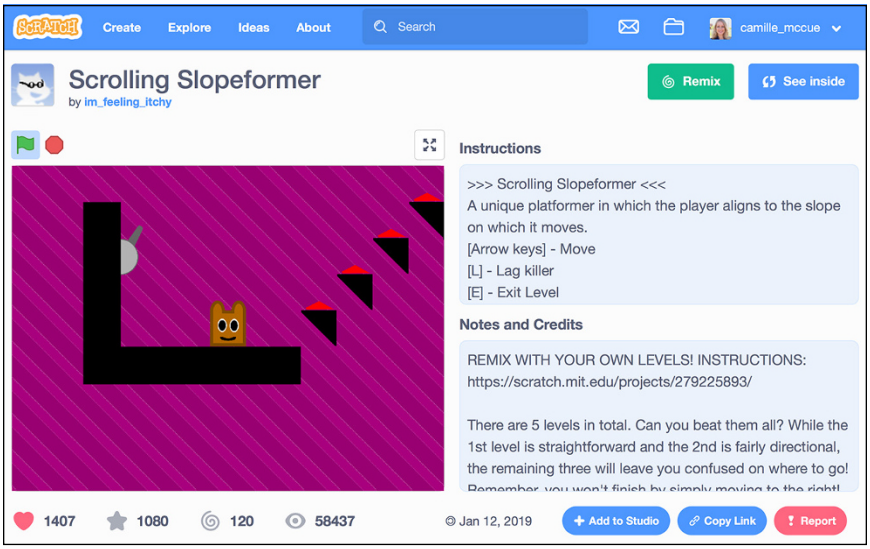


Figure 19-1

Observing daily human challenges

As you go through your day, pay attention to some of the little things that could be improved, and then think about how you could put your coding skills to work to fix them.

Do you want to build a simple website to help your brother with his dog-walking business? Or a Scratch game to quiz you on your French vocabulary? Maybe a simple geolocation app that helps your mom find where she parked her car? (This project is easy to do using a Google Maps API and another simple programming language called App Inventor.)

Or do you want to build a micro:bit temperature gadget that messages you in your room when the dessert you're refrigerating has reached the desired temperature? (I built the one shown in Figure 19-2. You can see the degrees Celsius reported from the micro:bit to the tablet, with the temperature reading dropping the moment the gadget is put into the fridge.)

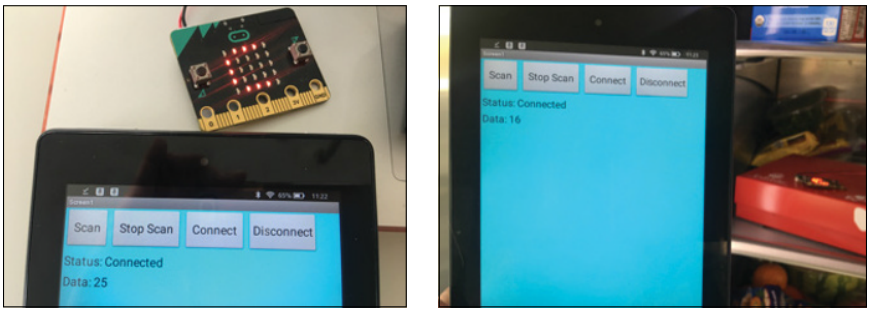


Figure 19-2

Start paying attention to the ways in which people of all ages go about their days, and jot down what you observe in a journal. Are people doing things that cost them time, money, effort, or happiness — things that maybe they could do another way? Reflect on your observations and add little drawings and notes about how to fix some of the challenges you observe. You might be able to transform some of your ideas into apps that could make big differences in people's lives!

Entering some contests

One way to spark ideas is to answer a design challenge in a competition. Contests such as the annual Congressional App Challenge (`www.congressionalappchallenge.us/`) invite students to work individually or in teams to prototype any app concept that will be useful to people in their community. For example, one student team came up with an app to help local veterans find support resources close to their homes. By thinking about and researching a specific audience and their needs, you begin empathizing with that group, and you can better *ideate* (come up with ideas for) new technology products to help them.

Other contests are hosted by a variety of organizations, sometimes leading up to a special event or limited to specific regions or student audiences. For example, the Games for Change Student Challenge (`www.gamesforchange.org/studentchallenge/`) hosts competitions in major metropolitan areas, inviting students to create and submit games that make a difference. Entrants can create games about one of the challenge themes, such as endangered species, disrupting aging, and automated communities 2050. New contest sponsors and events appear online with varying frequencies — just Google *student app development* competitions every so often to see what's out there!

Sharing and Showcasing Your Work

A great way to engage fully in the world of coding is to share your work, making it publicly viewable. Doing so allows others to use your app and provide feedback. Here's how to share your work in each of the three IDEs used in this book.

Sharing a Scratch project

Scratch provides an exciting set of tools for sharing your work with the world. After you've completed a project, follow these steps to share it:

1. At the top of the workspace, click the See Project Page button.

2. At the project page, click the Share button.

   The project page appears, as shown in Figure 19-3.

3. In the empty Instructions box, type information about how to use your app. In the Notes and Credits box, type any additional information you want to provide.

4. If you want to add your project to a Scratch studio, click the Add to Studio button.

   See the next section for help on creating a Scratch studio.