



# Girls Who Code At Home

**Debug the Missing Code: Part 2**

Logic Bugs

## Activity Overview

In Part 1, we learned how to identify and debug syntax errors in the Buggy Personality Quiz using the console window. In Part 2, we are going to examine a new type of bug, a logic bug, and learn a set of debugging strategies to repair our Buggy Personality Code. Programmers use logic to tell their program what actions to take. Sometimes, however, our own logic doesn't line up with the instructions that the computer needs to run a program successfully. We'll be working with JavaScript just like we did in Part 1, but the debugging approaches apply to all languages. You do not have to know JavaScript to complete this activity.

What **you think** you programmed:

IF it's cold outside,  
THEN wear a jacket.

What the **computer read** as your program:

IF it's cold outside,  
THEN dance to pizza.

You should have already completed [Part 1](#) of Debug the Missing Code before embarking on this activity.

## Learning Goals

By the end of this activity you will be able to...

- ❑ describe the difference between logic errors and syntax errors.
- ❑ describe a variety of strategies for debugging code.
- ❑ apply debugging strategies to fix the logic error in a broken website.

## Materials

- [Repl.it](#) Editor
- [Buggy Personality Quiz Sample Project](#)
- [Buggy Personality Quiz - Broken Project](#) (with logic error only)
- [Missing Code Reference Guide](#)

## Prior Knowledge

Before embarking on this project, we recommend that you:

- have some familiarity with core computational concepts including [variables](#), [functions](#), and [conditional statements](#) in any programming language.
- have beginner experience using a text-based language like JavaScript, Python, Swift, etc.

If you want to learn more about programming in JavaScript, check out the Girls Who Code at Home [Virtual Hike](#) activity.

If you want to practice debugging in the Scratch, check out the [Brave Not Perfect Debugging with Scratch](#) Girls Who Code at Home activity.

## Women in Tech Spotlight: Simone Giertz



Source: [Mashable](#)

Simone Giertz's inventions include robots that wash hair, chop vegetables, and apply lipstick in the morning. However, her robots rarely (if ever) succeed and are mostly useless, and Simone embraces this failure! She says that "the true beauty of making useless things [is] this acknowledgment that you don't always know what the best answer is."

Known to her fans as the Mistress of Malfunction, inventor and robot enthusiast Simone Giertz spends most of her time tinkering with Arduinos on her YouTube channel, hosting *Tested* with Adam Savage, and speaking about her wacky projects to live audiences.

Watch [this video](#) to learn more about the Applause Machine, one of Simone's wacky inventions, and her inspiration behind the product and design.

After being diagnosed with a non-cancerous brain tumor in 2018, Simone took time away from her inventions for treatment and reflection. Now that she's in recovery, Simone is still inventing, but she's shifted her focus to bigger and more functional products. Her most recent big hack? Converting a Tesla into a pickup truck. Meet the [Truckla](#).

## Reflect

Remember, being a computer scientist is about more than just being great at coding. Discuss how Simone and her work relates to the strengths that great computer scientists focus on building - just like you are in your Girls Who Code program.



### PURPOSE

Simone keeps a positive attitude even when her robots or projects don't work as expected. How can you help yourself stay optimistic in the face of setbacks?

Share your responses with a family member or friend. Encourage others to read more about Simone to join in the discussion!

## Step 1: Meet the logic bugs (6-8 mins)

### What is logic? (1 min)

Contrary to what you might think, computers are not smart. They are machines that (at least for now) do not know how to make decisions and perform an action on their own. Computers need humans to tell them what to do - to program them with a set of instructions. This instruction set is called an **algorithm**. But how does the program know which instruction to follow or when to follow it? This is where **logic** comes in. We use logic to tell our program the order or **sequence** to use to complete its tasks.

### How do I use logic? (3-4 mins)

Here are three common structures programmers use to add logic and determine sequencing in their programs. We've written these examples in JavaScript, but the usage of these structures translates to most other programming languages.

#### CONDITIONAL STATEMENTS

These statements tell our program how to make decisions by evaluating if a statement is true or false using [if...else if...else](#) and comparison or logical [operators](#).

```
if(numPlayers >= 5){
  playAmongUs();
} else {
  playScattergories();
}
```

This conditional makes a decision about which game to play based on the number of players. If the number of players is greater than or equal to five, play *Among Us*, otherwise play *Scattergories*.

#### FUNCTIONS

Create a new function whenever you have chunks of code you want to reuse or to keep your code readable. Some functions return specified values while others do not.

```
if(score == 10){
  updateWinner();
  restartGame();
}

function updateWinner(){
  console.log("Winner!");
}

function restartGame{
  score = 0;
}
```

This if statement tells the program to update the winner and restart the game if the score is 10. The functions make the code more readable because we group the substeps into functions that have descriptive names.

#### LOOPS

Loops to repeat a series of instructions until a condition is met. The main types of loops are [for](#) loops and [while](#) loops.

```
while(winner == false){
  keepPlaying();
}
```

This while loop tells the program to run the keepPlaying function as long as there is no winner.

**Note:** We will not be working with loops in this activity, but they are important to know about!

### Where do bugs live? (1 min)

Bugs can happen at any point in your program. You might find them hiding in your [variables](#), [functions](#), [conditionals](#), and more.

#### CODE

```
if (temp <= 65){
  wearJacket();
} else if (temp >= 66){
  removeJacket();
}

function wearJacket(){
  Take jacket out of closet
  Unzip
  Put one arm in
  Put the other arm in
  Zip up the jacket
}

function removeJacket(){
  Unzip jacket
  Take one arm out
  Take the other arm out
  Hang in closet
}
```

#### SEQUENCE

The sequence of this code if the temperature is **47 degrees** Fahrenheit:

##### Test Condition 1: temp <= 65?

- Condition 1 evaluates as TRUE
- Go to wearJacket function
- Run all lines of code in wearJacket function
- Exit

The sequence of this code if the temperature is **84 degrees** Fahrenheit:

##### Test Condition 1: temp <= 65?

- Condition 1 evaluates as FALSE

##### Test Condition 2: temp >= 66?

- Condition 2 evaluates as TRUE
- Go to removeJacket function
- Run all lines of code in removeJacket function
- Exit

Even though the code is the same both times, the order of instructions is different. Here, the sequence of tasks depends on the value of temp.

### Why do logic errors cause problems? (2-3 mins)

These errors are a bit more difficult to solve than syntax errors. When we debugged our syntax errors, we used the error console to identify our bugs and make changes. This is a super useful strategy for syntax errors, but you can't rely on the console to tell you when you have a logic error. In this case, your code might run, but it won't work the way you want it to. If your code doesn't throw an error while it compiles (i.e. converts your code to a form that the computer can execute), then the computer doesn't think there is an error. Logic bugs create a problem with the **program flow** - the sequence or order that your lines of code execute.

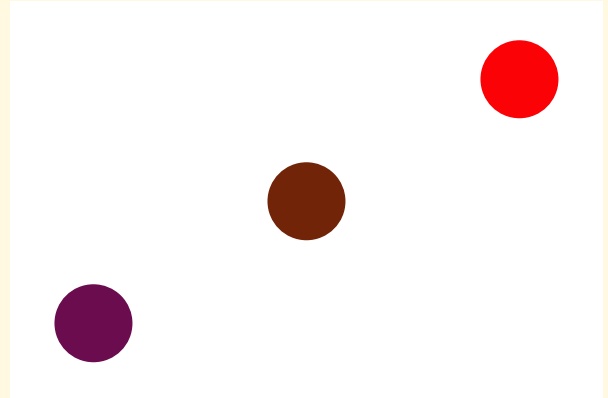
## Step 1: Meet the logic bugs (cont.)

Let's say you want to paint bright red, green, and purple colored circles. Check out this example instruction set for creating your painting:

### CODE

```
Dip brush in red paint
Paint circle in top right
Dip your brush in green paint
Paint circle in center
Dip your brush in purple paint
Paint circle in bottom left
Rinse brush
Let canvas dry
```

### RESULTS



In our current program, the last circle will likely be a muddy purplish-brown color. Why do you think that is? How would you rewrite this “program” so the last circle is purple?

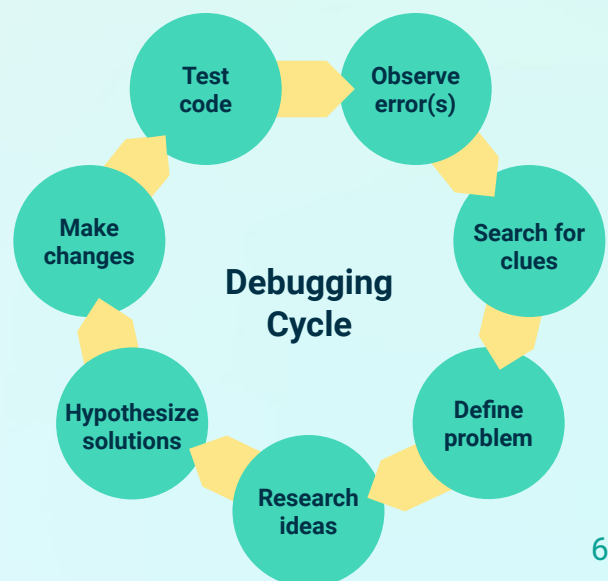


Check your ideas in the Reference Guide on pg.2

## Step 2: Build your debugging strategies (5-10 mins)

In Part 1, we learned a few debugging techniques such as using the console to read error messages, making observations, and testing a change you made. As you progress into more complex code, it is helpful to have a toolbox full of debugging strategies and a system for implementing them.

Debugging involves a cycle of testing code, defining the problem, hypothesizing solutions, making changes, and testing code again until you figure it out. All bugs are different, but there are common strategies you can use to help you find a solution. On the next page is a list of strategies you can use when a bug flies into your code.



## Step 2: Build your debugging strategies (cont.)

### 1. Put yourself in the right mindset for debugging

Chances are you will only find a bug after you've spent a lot of time and invested a lot of energy into your code. You might need to go pet your dog, get a snack, take a walk, or even sleep on it before diving in. Debugging should happen slowly and methodically. If you are moving fast, you will likely miss something or create new bugs.

### 2. Go back to basics

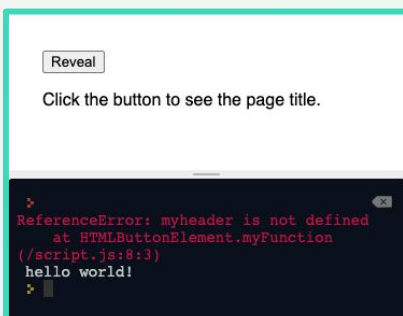
Are you connected to the internet? Is your code editor or IDE (i.e integrated developer environment) functioning properly? Does your browser need to restart for an update? It might sound silly...until it fixes everything.

### 3. Make a copy

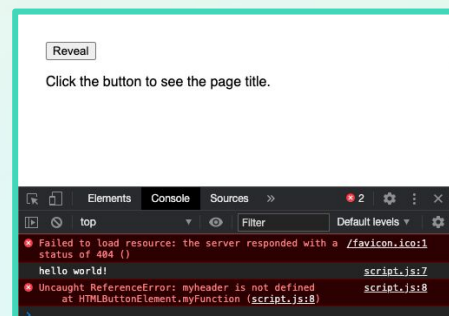
Trying to fix problems can lead to more problems. Always make a copy if you need to make any big changes. Better safe than sorry!

### 4. Start simple

Check the simple things first. This is a process of elimination, so get the easy ones out of the way. Search for ground-level errors and check for mistakes in syntax. If your IDE has a debugging console, check it for error messages. If you are programming for the web, you can also use [developer tools](#) in the browser. Below is an example of how the same error appears in the Repl.it console on the left and using developer tools in Chrome on the right.



The Repl.it console is the black box under the display window. It shows an error message in red text with the error location, then prints **hello world!** to the screen.



Developer tools displays on the right or bottom screen. The **Console** tab prints **hello world!** to the screen, then shows an error message in red text on the left and the location in white on the right.

### 5. Search for patterns

Programming, like other human languages, relies on patterns as a way to organize and make sense of your code. We can use patterns that we've seen in past code and apply them to a current challenge. For example, let's say you are programming a button to decrease the score each time it's clicked. You've already coded two buttons to increase the score, but you're not sure how to program the second. You can observe similarities or patterns in their behavior to help you find a solution.

Click me to increase by 1!

```
if(clicked == true){  
  score = score + 1;  
}
```

Click me to increase by 2!

```
if(clicked == true){  
  score = score + 2;  
}
```

Click me to decrease by 1!





### 6. Describe the problem

Sometimes when we look at a problem on our own for too long, we start skipping over the simple things. But if you have to describe the problem to someone else, you have to be more detailed. Pretend you are telling or writing a friend about your problem and describe:

- ❑ What you have already done.
- ❑ What is going wrong.
- ❑ What you expected or want to happen.
- ❑ What additions or changes you made in your code since the last time you tested it.

Revisit your planning notes, pseudocode, and/or diagrams too. Chances are you'll spot the mistake on your own.

### 7. Make it observable

It's super helpful to observe how the values of variables change over time or when different events occur, like a button click or text input. You will often use these variables to make decisions about your program flow. For example:

```
if(zoomCalls >= 5){  
  takeWalk();  
}
```

But how can you know what the value of a variable is? How do I know the number of **zoomCalls**? You can use the [console.log\(variableName\)](#) function to print the value of a variable to the IDE's console window or view it using [developer tools](#).

#### CODE

```
console.log("zoomCalls = " + zoomCalls);  
if(zoomCalls >= 5){  
  takeWalk();  
}
```

#### CONSOLE

```
zoomCalls = 3  
zoomCalls = 4
```

**Note:** If you want to know what value you are printing, you can add context to your values with text. Put the text you want to print in double `" "` or single `' '` quotation marks followed by a plus sign `+` and the variable name. This is especially helpful if you want to print more than one at a time!



## Step 2: Build your debugging strategies (cont.)

### 8. Test one thing at a time

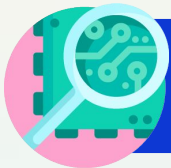
Here's one scenario we definitely do not want: by changing a few "small things" in your code, you create a new problem and now you can't figure out if it's the old problem or the new changes causing it. Your goal is to isolate the problem. Change one thing at a time, then test it.

### 9. Search the interwebs

There are a ton of websites devoted entirely to answering questions about code errors. Chances are pretty good that someone else has encountered your problem. Once you've described your problem in detail, use your favorite search engine to find some answers. If nothing comes up, try reframing or rewording your question - being able to search for a solution efficiently is a skill in itself! You can also copy any error messages and start your search there. If you are programming for the web, [W3Schools](#) and [MDN](#) are great resources. [StackOverflow](#) is a good site for developers working in all languages to ask questions and learn from other people's solutions. Whatever language you are programming in, there are likely forums and many websites or posts devoted to learning it.

### 10. Ask a friend

If you still can't figure it out after trying everything above, phone in a friend, mentor, or teacher.



For more debugging resources, check out [this video series](#) from Clay Shirky at NYU and this fantastic [Field Guide to Debugging](#) from the p5.js community.

## Step 3: Test the quiz (3-4 mins)

Now that we have a handy toolkit of debugging strategies, it's time to solve our last bug - the logic bug. First, we'll need to figure out what and where the error is before we can even think about solving it.

Remember: Much of the time, the program isn't wrong because it is following your instructions. This means that logic bugs are really a result of our own misconceptions about how our program *should* work vs. how it *actually* works. Logic errors are about mistakes in how your program executes. Before we dive in, let's test the quiz for any clues.

### Take the quiz (2-3 mins)

When we tested the buggy version at the beginning of this activity, the quiz never returned our result. Now that we've fixed our syntax bugs, let's test it again just to make sure nothing else has changed.

- ☐ Take the quiz 3-4 times.
- ☐ Answer the questions in different configurations.
- ☐ Use the restart button after you finish to reset the quiz.

### Step 3: Test the quiz (cont.)

There we have it - still no results! And no more error messages. At this point, the best thing to do is move back, literally and figuratively. If you need a quick break and a snack, now's the time. In the next section, we'll start our search for clues by reviewing how the code works.

**You are a...**

Restart

Screenshot of the result text at the bottom that says "You are a..." and a Restart button underneath it.

### Step 4: Review the code (3-7 mins)

You probably read the code and comments as you fixed the first bugs, but if you didn't, go back and do it now. It's totally ok if you don't understand everything that is going on. Our goal is to get a handle on the program flow, so we just need a high level understanding at this point.



Check your ideas in the Reference Guide on pg 2.

### Step 5: Describe the problem (5-7 mins)

Let's start by describing what we want to happen. Grab a pen and paper or open a text editor and write out what the program should do when a person takes the quiz. We've included the first step below.

- When a person clicks an answer button, the program calls the specified bug function (i.e. **bee**, **butterfly**, **grasshopper** or **ladybug**) attached to it in the event listener.



Check your ideas in the Reference Guide on pg 4.

After writing this out, we can see how important it is to know the value of our scoring variables over the course of our program. For example, if **grasshopperScore** is greater than 4 and no result is returned, that is a huge clue for us!

## Step 6: Make it observable (10-15 mins)

How do we access the value of our variables? With `console.log()` of course! We want to know the values of each bug score variable and `questionCount` each time they are updated. This last part is important because it helps us know where to put the `console.log()` function. We want to know these values after a person has clicked the button, so we will put `console.log()` inside each bug score function.

- ❑ Use `console.log()` to print the score variable and `questionCount` value inside the functions below. Add context to your messages by using single or double quotations and a plus sign to add explanatory text so you know which value goes with which score (see Step 3 for an example if you need it).
  - ❑ `bee` function
  - ❑ `butterfly` function
  - ❑ `grasshopper` function
  - ❑ `ladybug` function

**Tip:** You can use `"\t"` to create a space between the two values!



Check your ideas in the Reference Guide on pg 4.

Once you have added `console.log()` to your functions, it's time to test your code!

- ❑ Click the **Run** button.
- ❑ Click on **Mochi** for question one, then check the console window.

Finally! We have some output from our program! The following text will display in the console:

```
questionCount = 1      butterflyScore = 1
```

- ❑ Now click **City** for question two and check the console. The following text should display:

```
questionCount = 1      butterflyScore = 1
questionCount = 2      beeScore = 1
```

Looking good so far! Our values are increasing with each click, so we know our bug functions are working.

## Step 6: Make it observable (cont.)

- ❑ For the third question, click on **2018** and check the console. The following text should display:

```
questionCount = 1      butterflyScore = 1
questionCount = 2      beeScore = 1
questionCount = 3      butterflyScore = 2
```

At this point, we should have a result returned to us. The **questionCount** is equal to 3 and we have a score that is greater than or equal to 2 (the **butterflyScore**), but the result text has not changed.

Let's think about these clues and ask ourselves a few questions:

- ❑ What controls the result text?
- ❑ When do we call (or use) the **updateResult** function?
- ❑ Is the conditional statement running when we need it to?



Check your ideas in the Reference Guide on pg 5.

## Step 7: Search for patterns (1-2 mins)

We are starting to narrow down the problem to the conditional statement at line 82:

```
81 // Track for end of quiz
82 if (questionCount == 3){
83     updateResult();
84 }
```

Right now, this conditional tests the **questionCount** value after all of the bug score tracking functions. You may have noticed a pattern that updates to the score and **questionCount** variables happen inside each bug function. Maybe we need to test the value of **questionCount** inside each function!

## Step 8: Test one thing at a time (5-10 mins)

Let's test this hypothesis. Instead of changing all the functions, let's try changing the bee function first. If that works, then we can change the remaining functions.

## Step 8: Test one thing at a time (cont.)

- ❑ **Copy the full `questionCount` conditional statement.** Be sure to get all the curly brackets!

```
if (questionCount == 3){  
  updateResult();  
}
```

- ❑ **Paste it inside of the bee function.** It should be underneath the `console.log()` function, and above the closing curly bracket.
- ❑ **Run your code.**
- ❑ **Test your hypothesis by selecting all bee responses.** Click the button of the bee responses for each question:
  - ❑ Q1: Fruit
  - ❑ Q2: City
  - ❑ Q3: 2019
- ❑ **Check the result text.** Do the results show up or do we still have a bug? You should notice that we did get the bee result which makes us think that maybe we have identified the problem! Let's test to see if our hypothesis works for the other results.

**You are a bee!**

Restart

Screenshot of the result text at the bottom that says "You are a bee!" and a Restart button underneath it.

We have a result that matches our choices when testing our hypothesis on the bee responses! It seems we've identified the problem causing the error.

- ❑ **Add the conditional statement in the same location for the remaining functions.** Remember to *test one hypothesis at a time*. Follow similar steps to test the following functions:
  - ❑ **butterfly** function
  - ❑ **grasshopper** function
  - ❑ **ladybug** function

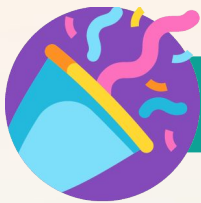
Test the program again. Answer questions in various configurations and see what result gets returned. There is a scoring key on the next page to help you test.



Check your ideas in the Reference Guide on pg 6.

## Step 8: Test one thing at a time (cont.)

Bug Result	bee	butterfly	grasshopper	ladybug
Q1: Dessert	Fruit q1a4	Mochi q1a1	Cookies q1a2	Cake q1a3
Q2: Vacation	City q2a2	Beach q2a3	Woods q2a1	Mountains q2a4
Q3: Color	2019 q3a4	2018 q3a2	2020 q3a3	2017 q3a2



## Congratulations!

You debugged the Buggy Personality Quiz successfully! Now that we have a fully functioning quiz, you can move onto the Extensions to learn more.

**Tip:** Want to try to avoid logic errors altogether in the future? Put in the time to plan, diagram, and writing out your code in human readable language (i.e. pseudocode).

## Step 9: Extensions (5-75 mins)

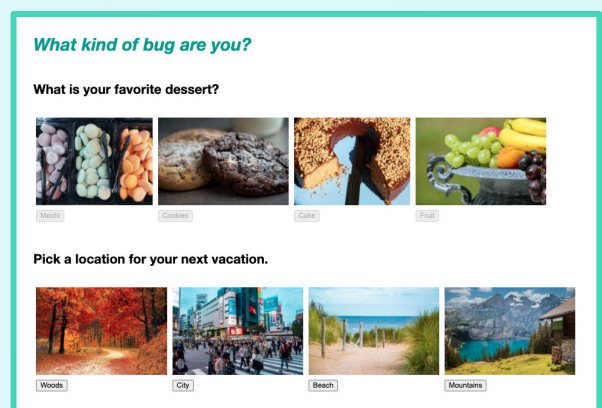
### Extension 1: Write your personal debugging checklist (5-10 mins)

Customize the strategies above to find an approach that works for you! When you first start programming and encountering different errors, it's not easy to remember where to start. Grab a pen and paper or open your fav text editor and create a list of the strategies you want to remember. Put it somewhere you can easily access so you know where to go when you face your next bug!

### Extension 2: Disable the buttons (15-20 mins)

Right now, the user could click more than one answer choice for a given question and mess up the results of the quiz. Consider how you might disable the answer choice buttons once one of them is clicked.

You can check out an [example of this Extension](#) here.



To get started you might:

- ❑ Create functions that disable all the answer buttons for a given question.
- ❑ Then you can add another event listener for each answer choice button that calls your disable function whenever one of the answer choice buttons are clicked.
- ❑ Finally, if you added an extension to restart the game, you'll also want to make sure you enable all of your buttons again as part of your restart function.

### Extension Resources

- [HTML DOM Button Disable Property](#)
- [Event listeners](#)
- [JavaScript functions](#)
- [JavaScript HTML DOM](#)

### Extension 3: Customize the Personality Quiz (25-45 mins)

Create your own personality quiz! It's pretty rare that programmers create something completely from scratch. In fact, it's better to write code that you can reuse. Try reusing this code to customize your quiz. You will have to update the **index.html** file in this Extension, so it is helpful if you have some knowledge of HTML. If you don't, no problem - see the Extension Resources for support. Below are some of the steps you might take to get started.

### Planning

Fill in this planning guide as you answer the questions in this section:

QUESTION 1				
Answers	1.	2.	3.	4.
Result Scoring	+1	+1	+1	+1
QUESTION 2				
Answers	1.	2.	3.	4.
Result Scoring	+1	+1	+1	+1
QUESTION 3				
Answers	1.	2.	3.	4.
Result Scoring	+1	+1	+1	+1



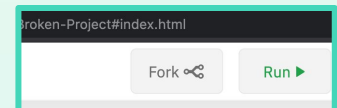
## Step 9: Extensions (cont.)

To get started you might:

- ❑ Choose four possible final results. For this quiz choose only four possible outcomes that your user will have at the very end. Later, you will need to map each answer to each question to one of the possible outcomes.
  - ❑ For example: Bee, Butterfly, Grasshopper, and Ladybug
- ❑ Choose three questions to ask. Map out what questions you will ask in your quiz. Each question should be related to each other and have four possible answers.
  - ❑ For example: What is your favorite dessert?
- ❑ Choose four possible answers for each question. Each answer should map directly to one of the final outcomes of the quiz.
  - ❑ For example: Answer 1 to Question 1 is Mochi. Mochi maps to butterfly, so by choosing Mochi, the **butterflyScore** will increase by 1.
- ❑ Select the photo you want to represent each answer. Save it locally on your computer. If it has a long name, you may want to rename it to something shorter and more descriptive like **q1a3-cake.jpg**.
  - ❑ Remember that free stock images or [Creative Commons](#) images are great to use as you're working. Take a look through the options available on sites like [Unsplash](#), [Pixabay](#), and [Burst](#).

### Update the text and images in the index.html file

- ❑ Fork your working Repl.it project. Rename it and give it a description.
- ❑ Upload your images to the **assets** folder in the Files pane on the left.
- ❑ Update the title of your quiz by changing the text inside the **<h1>** tags.
- ❑ Update the text and images for Question 1 and its answers using the planning guide above.
  - ❑ Replace the current question inside the first set of **<h2>** tags with your first question.
  - ❑ Update the image for the first answer. Right under **<div class="answer-choice">** there is an **<img>** tag that displays the image you want to use for the first answer to Question 1. Replace the existing link with a link to the image you want to use. The link will be **assets/imageName.jpg**.
  - ❑ Update the button text. Beneath the **<img>** tag you should see a **<button>** tag with the answer text. For example, **<button id="q1a1">Mochi</button>**. Delete the existing answer and add your own.
  - ❑ Repeat this process for the remaining answers to Question 1.
- ❑ Update the text and images for the remaining questions and answers in the **index.html** file using the planning guide above.



## Step 9: Extensions (cont.)

### Update the Logic in the script.js file

There is logic built into the quiz that determines a person's result based on how they answer each question. As you noticed when reviewing the code, each answer maps to a specific result using an event listener. You will need to update this logic based on your own questions, answers, and results. For a full review of the logic and how it is built, see Step 4 in the Reference Guide on pg. 4.

- ❑ At the top under the `/* Global Variables */` comment, update the score variable names to the four possible results of your own quiz.
- ❑ Update the names of each variable **everywhere** in your program. *If your program doesn't work, this is the first place you should check.*
- ❑ Update the function names in each event listener. The name should be the result that the answer maps to. For example, if the answer to **q1a1** maps to the result ice cream, then you could name the function **iceCream**.
- ❑ Replace the current function names with ones you just created in the event listeners. *Be sure they match the correct score variables.*

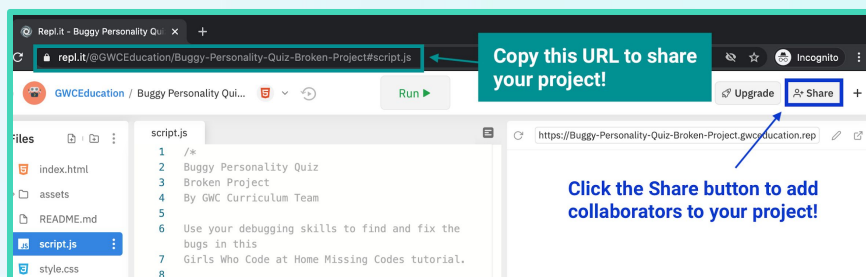
### Extension Resources

- [Introduction to HTML](#)
- [Share Your Skillz Girls Who Code at Home](#) for a hands-on introduction to HTML
- [Heading tags \(e.g. <h1>\)](#)
- [<img> tag](#)
- [<button> tag](#)
- [Event listeners](#)
- [JavaScript functions](#)
- [JavaScript HTML DOM](#)

## Step 10: Share Your Girls Who Code at Home Project! (5 mins)

### View only access (1-2 mins)

Sharing your work on Repl.it is easy! Just copy and paste the URL address of your Repl project in the web address bar at the top. This will allow others to run your project, view your code, and fork your project and remix on their own. We would love to see your debugging work and we know others would as well. Share your fixed code with us!



Be sure to share this link on your social media accounts and don't forget to tag **@girlswhocode** **#codefromhome** and we might even feature you on our account!

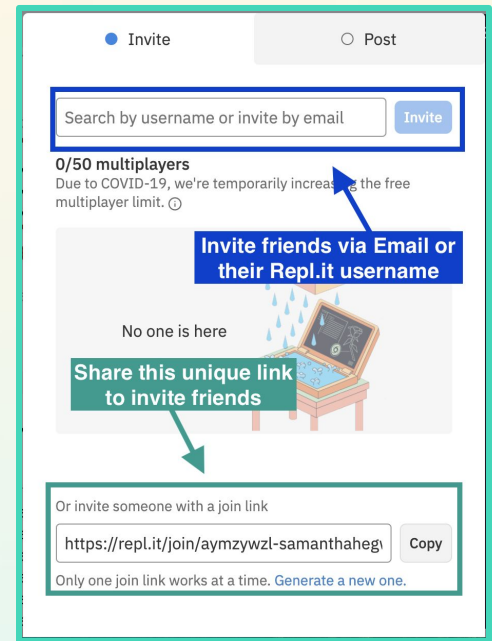
## Step 10: Share Your Girls Who Code Project! (cont.)

### Adding collaborators (2 mins)

If you want to work with a group of friends on a project, you can easily invite them to collaborate using the **Share** button at the top-right of the window. This should pop out a new window with two options for inviting others to collaborate on your project.

- ❑ **Invite by email or Repl.it username.** This option allows you to share your project with specific people by typing in their email address or Repl.it username if they already have an account with Repl.it. We recommend this option to ensure that you are sharing your project with the correct people!
- ❑ **Share invite link.** At the bottom of the window there is a unique invite link. You can copy and paste this link to friends which will allow them to access your project.

**Note about collaborators:** Remember that adding collaborators gives others edit access to your project. This will allow them to change your code, name, and description. **DO NOT share your invite link on social media!** Be selective on who you share these edit rights with.



Stay tuned for more Girls Who Code at Home activities!

