Girls Who Code At Home

Debug the Missing Code: Part 1

Syntax Bugs

Activity Overview

Debugging is one of the most important skills a programmer can develop. And it has nothing to do with insects. Bugs are errors in your program that stop it from working the way you want it to. They can be as simple as a spelling mistake or as big as writing your code out of order. Bugs happen in every programming language to programmers of all experience levels. Bugs even happen in hardware (i.e. the circuits that run your software). Bugs can be frustrating when you can't figure it out and they can be enlightening when you finally realize the problem.

> If **debugging** is the process of removing software bugs, then **programming** must be the process of putting them in.
>
> ~ Edsger Dijkstra

Part 1 of this activity will introduce you to the most common type of bug: syntax bugs. We'll work together to solve two syntax errors in the broken Buggy Personality Quiz. We'll be working with JavaScript, but the approaches apply to all languages. Don't know JavaScript? Don't worry, you do **not** need to know it to complete this activity.

Learning Goals

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

- ❏ describe why debugging is an important part of learning how to program.
- ❏ identify different types of syntax bugs you might encounter in your code.
- ❏ debug the syntax errors in a broken website.

Materials

- ➔ Repl.it Editor
- ➔ Buggy Personality Quiz Sample Project
- ➔ Buggy Personality Quiz - Broken Project
- ➔ 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

It's Computer Science Ed Week and we've teamed up with IF/THEN®, to celebrate women in technology. The internet as we know it wouldn't be possible without the contributions of women in tech. And yet, despite the fact that 26% of coders are female, there is a stubborn perception out there that the products we use everyday are built by men. Wrong!

Checkout www.missing-code.com, to see what could happen to your favorite platforms in a world where women don't code.

Reflect

Take some time to learn more about the contributions female-identifying people have made to computer science. As you explore the Missing Code website, choose one fact that you want to learn more about and spend a few minutes researching it. When you are finished, reflect on the question below.

PURPOSE

The internet won't necessarily break if women and female-identifying people stop coding, but it will have damaging effects in other ways. How does this lack of representation in computer science impact individuals and communities?

Share your responses with a family member or friend. Encourage others to read more about the contributions of women and female-identifying people to computer science fields!

Step 1: Welcome the bugs (3-5 mins)

Close your eyes and think about a time you encountered a problem that you didn't know how to solve, whether nailing a new viral TikTok dance, getting your favorite cookie recipe right, or building the perfect abode in Animal Crossing. Think about the process you took to figure it out. Think about the emotions and mindset you had along the way. Now - eyes still closed - remember how awesome it felt to find a solution.

Open your eyes. With all those warm fuzzies still bouncing around, let us take this opportunity to tell you that **errors will happen in your code**.

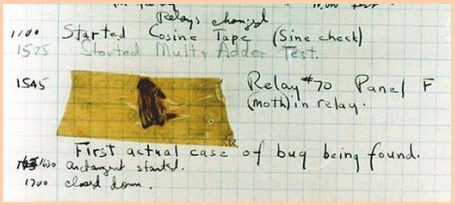
> It doesn't matter if you are just getting started or if you've been programming for years in three languages. Your programs will break. By break we mean that they won't work as intended, not that your computer will explode. This might seem like a bummer, but it's not. Solving these errors is the best way to learn!

Embrace your bugs (1 min)

We call these errors **bugs**. We call the process of locating and fixing these errors **debugging**.

But bugs are not just an error in your code. They are a technical problem in your software program AND a gap in your understanding between what you wanted to happen and what actually happened. This is precisely why bugs are so frustrating, but also helpful. Finding and fixing a bug means that you have a chance to update your program and your understanding of how to write a program.

We've already updated our understanding that bugs happen no matter what your level of expertise is. So, since we'll be meeting old and new bugs for quite a while, it's important to develop a process for solving them.



Source: National Geographic

They are called bugs because back in the day when computers were made of vacuum tubes, actual insects could get inside the tubes and cause them to malfunction. We will be talking about software bugs in this activity, but bugs can happen in your hardware as well!

Step 2: Recognize your bugs (7-10 mins)

The first step in debugging is knowing when you have a bug. But what actually happens when you have one? What form can they take? Is my project destroyed forever?! (Highly, highly unlikely.) Let's start with these questions.

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.

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.

```
// The variable is not declared: var myNumber = 1;
myNumber = 1;

// There is no closing curly bracket to our conditional
if(myNumber > 10){
  addOne();

// The variable name is misspelled inside the function.
function addOne(){
  myNumbr += 1;
}
```

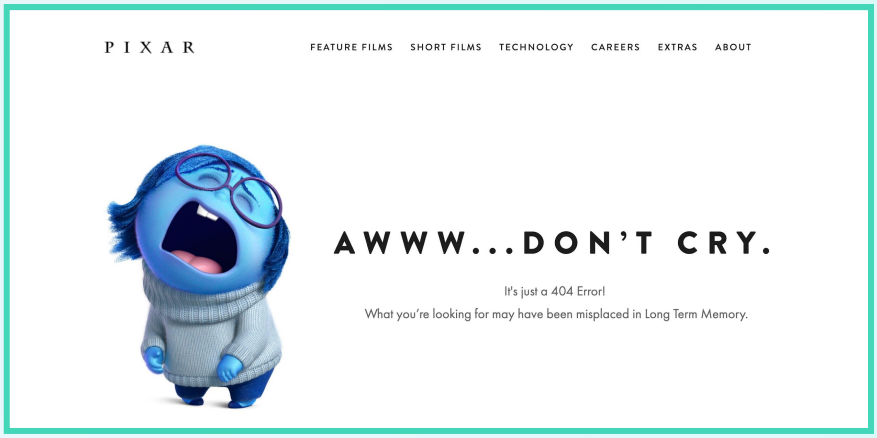
What happens when I have a bug? (1 min)

All bugs affect the functionality of your program. Some bugs are small and some bugs are big. Bugs would be much more fun if they caused images to stretch, buttons to fall off the screen, and text to fly around, but 99.9% of the time this isn't the case.

Instead, a bug might…

- ➔ cause a button to stop working on a website.
- ➔ prevent an image from displaying.
- ➔ send a person to the wrong page of a website.
- ➔ stop keeping track of a player's score when it reaches a certain value.
- ➔ cause a website to display incorrectly in one or more browsers.

And of course, a big bug can cause entire websites, games, and more to just stop working because the program can't compile and/or run the way it should.



If you've ever seen a 404 page, you've seen a bug in action.

Source: Pixar

Are there different types of bugs? (5-7 min)

Yup! Categorizing bugs is helpful since identifying the error is half the battle. There are **syntax bugs** and logic bugs. We'll focus on syntax bugs in this activity and cover logic bugs in the next activity.

Syntax bugs

You may have heard the word **syntax** in reference to grammar. What does that have to do with programming? A lot! The syntax of a programming language is like the syntax of any other language. There are rules that letters (e.g. b and B), numbers (e.g. 11 or 3.14), and symbols (e.g. { } or #) must follow to process and run a program. Different languages have different syntax.

For example, JavaScript is a programming language that is mainly used to add interactivity to websites and it is the language we will use in this activity. Let's look at two of the most recognizable syntax rules in JavaScript:

1. **JavaScript uses curly brackets `{ }` and parentheses `( )` to separate chunks of code, like functions or conditional statements.** Other languages like Python use indentation instead of curly brackets to separate chunks of code.
2. **JavaScript uses semicolons `;` to end a statement in your code.** A statement is an instruction you are giving your program like `console.log(score);`. Just as you must use a period at the end of a sentence, you should end your statements with a semicolon.

Here are a few guidelines to help your avoid the most common syntax bugs:

Lots of syntax errors are spelling errors.

For example, we use variables to store data so we can change them or compare them to other values when we need to. Because you will use them repeatedly, there is a higher chance you might misspell them at some point.

```
var mothScore = 0;

if(motScore > 2)

mothScre = 10;
```

Variables, function names, and keywords are case sensitive.

You cannot interchange lowercase and uppercase letters.

`var bee` is not the same as `var Bee`

`function` is not the same as `Function`.

If you open it, you must close it.

It's easy to lose or add an extra curly bracket or parenthesis when you group chunks of code together. Remember that all curly brackets and parentheses should have a partner.

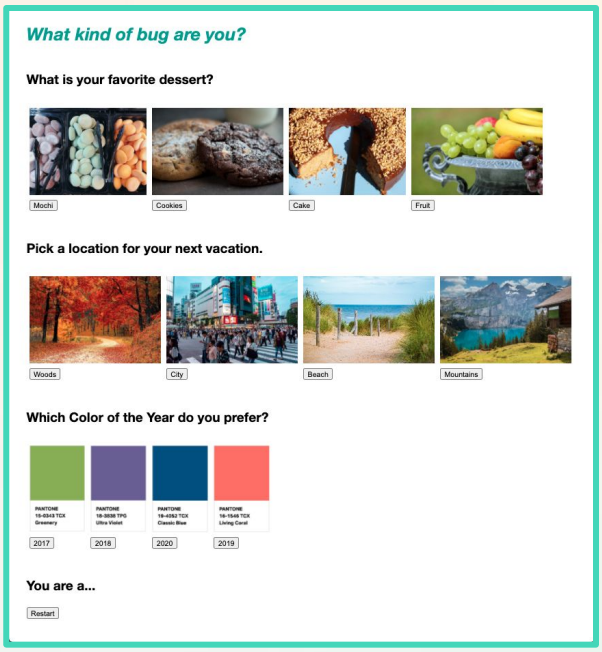
```
function restartQuiz(){
 if(score > 10){
  score = 0;
  }
}
```

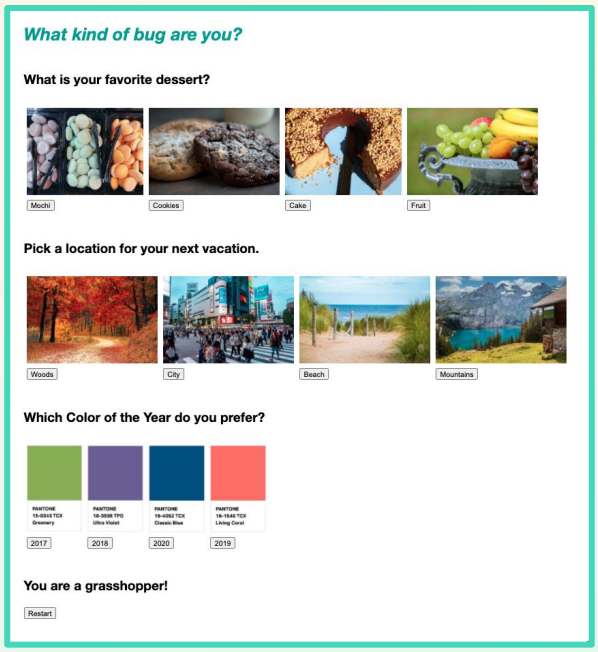
```
function restartQuiz(){
 if(score > 10){
  score = 0;
  }
```

For more on JavaScript syntax, check out **this guide** from Tania Rascia.

Step 3: Examine the buggy site (3-5 mins)

Before we can debug anything, we need to understand the problem. The best way to start this process is to test the program. First you'll take the quiz on the functioning site. Next, you'll check out the buggy version to investigate how its behavior compares to the working version.





Take the fixed Buggy Personality Quiz (3-5 mins)

Follow the link to the fixed version of the quiz. Now take the quiz to see which bug you are! When you're finished, think about the following questions. Write down your ideas if it's helpful.

- ❏ What actions did you take to get your result?
- ❏ How do you think the program figured out which bug you are?

Check your ideas in the Reference Guide on pg 2.

Take the broken quiz (1-2 mins)

Let's see how the broken quiz compares. Follow this link to the buggy version and test it out. Take the quiz a few times to figure out how the program behaves. Try answering the questions in different configurations to see if anything changes. If everything is properly broken, the quiz will never ever return the result.

Step 4: Get started with Repl.it (10 - 15 mins)

For this activity we will be using the Repl.it web editor. Repl.it is a free, collaborative, browser-based editor that supports multiple programming languages. This powerful tool can even allow you to code and talk with a group of friends all at the same time!

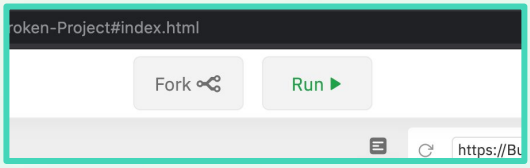
Create an account (5-7 mins)

- ❏ **Sign up or login to Repl.it.** In order to save your work you will need to create an account. Follow the instructions on the sign up form to create an account. If you are under 13 you'll need your parent's email address to sign up.
- ❏ **Follow the instructions to create an account.** You may choose to sign up with your Google, GitHub, or Facebook account for faster login.

Fork the broken Buggy Personality Quiz (2-3 mins)

- ❏ **Open the broken Buggy Personality Quiz.**
- ❏ **Create a copy of the broken Buggy Personality Quiz by clicking the Fork button next to the Run button.** Creating a fork duplicates the entire project and adds it to your Repl account as a new project.

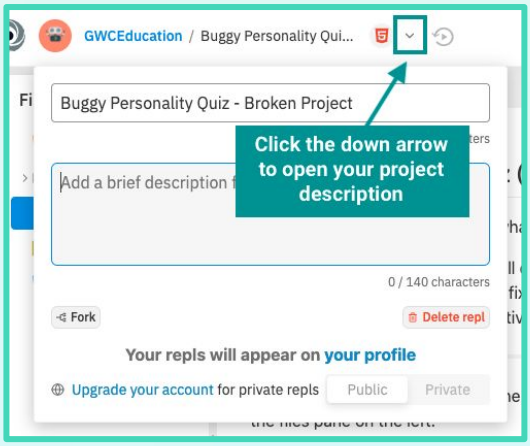
You can fork other people's source code or you can fork your own projects if, say, you are making bug fixes, but want to have an older version of your code that you can go back to in case you create more errors while debugging.



Add a project description (1-2 mins)

Now that you have your own copy of the buggy quiz, let's add a brief project description to our Repl.

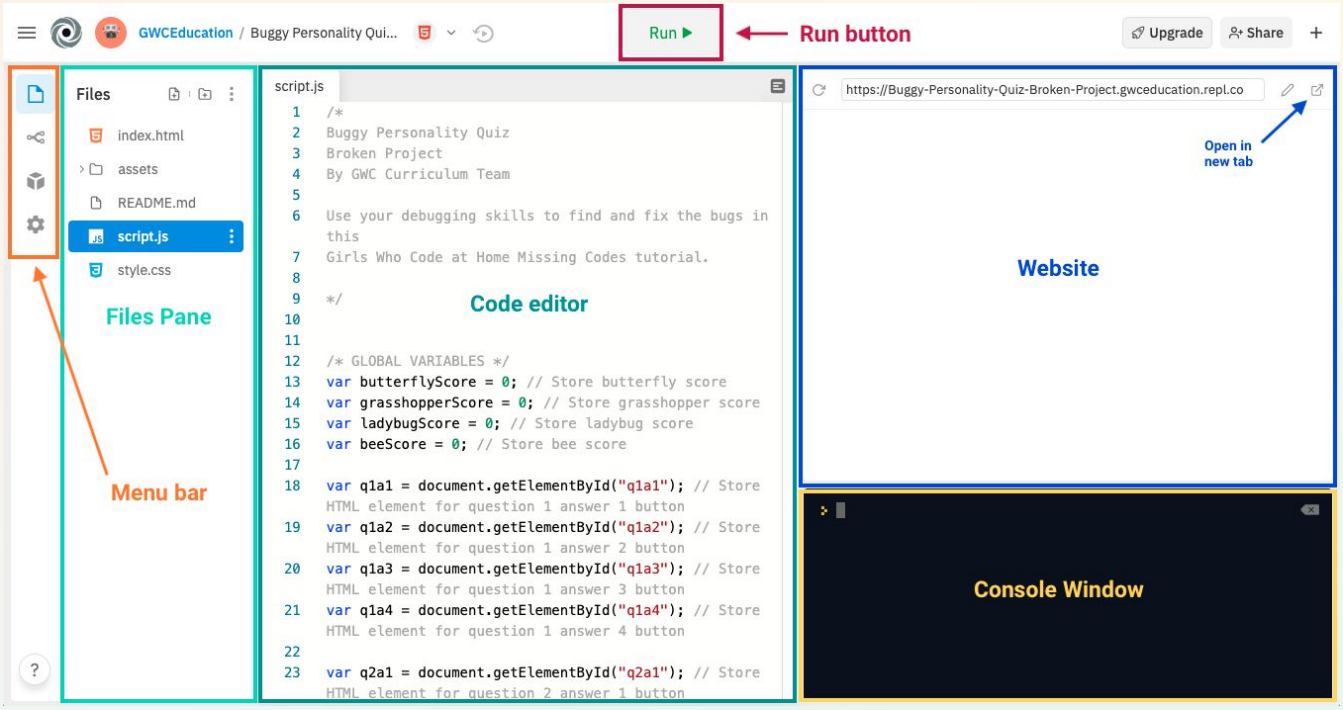
- ❏ **Open the project description.** Locate your project name at the top left of your screen. Click the small down arrow to the right of the name. This should open a new window containing your project name and an area for a brief description.
- ❏ **Add a brief description.** Something you might want to include in your description may include:
  - ❏ **Overview:** How is it supposed to work?
  - ❏ **Instructions:** Are there any specific instructions needed to run your project?
  - ❏ **Attributions:** Did you get help from others or additional resources? Make sure you shout out these people and resources!



Learn more about Repl.it in the Reference Guide on pg 3-4.

Explore the Editor View (3-5 mins)

Let's take a look into the editor view for Repl.it. Now that we created a new project, we need to understand where to code, navigate between files and assets, and how to save and run your code!



- **Files Pane:** This window displays all of your project files. For this activity, we will only be working in one file, script.js, but you will also see index.html, style.css, and README.md files. Click on the README file for more information about what each of these files does.
- **Menu Bar:** This bar will allow you to change the view in the file pane. Some options include changing version control, adding packages, and updating settings. In the settings you can customize the layout, theme, font size, and other text settings.
- **Code Editor:** This is where you will write your code!
- **Run Button:** After making changes to your code, click the run button at the top of the editor. You should see your result in the output/console window on the right.
- **Website:** This window displays your website. If you want to view it as a full webpage, click the new tab button in the top right corner.
- **Console Window:** This displays any output in your code. All outputs will be visible, so if you click the run button multiple times, each result will be displayed here.

You may have noticed that there is no **save** button in Repl.it. As long as you have internet connection, all changes in your code will be saved automatically. Once you click the Run button your Repl is saved, so **make sure you always run your code before closing your editor**!

Step 5: Fix the first bug (5-7 mins)

In Step 3, we observed how our broken quiz behaved and described how it is supposed to track a person's bug score. Now it's time to review the actual code in our **script.js** file to see where the problems are.

Press the *Run* button in the center of the top navigation bar. Notice the console on the bottom right side of the screen.

```
SyntaxError: Unexpected end of input at /script.js:111:1
```

We have our first error message! Error messages are the best. They give us a clue to what the problem is and help us locate it. Stop for a moment and try to decode this message:

- ➔ `SyntaxError` means that a problem with your syntax (i.e. spelling, punctuation, formatting, etc.) is causing the bug.
- ➔ `Unexpected end of input` means the computer didn't know where to end. It is a common error message to get if you forget to close a curly bracket { } somewhere in your code.
- ➔ `at /script.js` means your error is in the **script.js** file.
- ➔ `111:1` indicates that your error is on line 111 in the first space.

Now that's a lot of information that we can work with. Let's start by going to line 109. That's weird. It seems like we already have a curly bracket there. Try clicking on the curly bracket and see what happens.

```
101  // Restart quiz
102  function restartQuiz() {
103    result.innerHTML = "You are a...";
104    questionCount = 0;
105    beeScore = 0;
106    butterflyScore = 0;
107    grasshopperScore = 0;
108    ladybugScore = 0;
109  }
```

Click the closing curly bracket

Here's a hot tip: In most IDEs when you move your cursor next to one curly bracket or parenthesis, it highlights the partner curly bracket or parenthesis in the existing code. Using this technique, we know that:

1. The curly bracket on line 109 already has its partner on line 102
2. There are no other open curly brackets in the `restartQuiz` function.

This means we need our detective hats!

Follow the steps below, then check your code in the Reference Guide:

- ❏ **Look nearby.** If the error doesn't appear directly where the message said it would, try looking nearby. There's no code after the `reStartQuiz` function, so let's examine the `updateResult` function right before it starting on line 87:

```
110  // Update quiz result
111  function updateResult() {
112    if (beeScore >= 2) {
113      result.innerHTML = "You are a bee!";
114    } else if (butterflyScore >= 2) {
115      result.innerHTML = "You are a butterfly!";
116    } else if (grasshopperScore >= 2) {
117      result.innerHTML = "You are a grasshopper!";
118    } else if (ladybugScore >= 2) {
119      result.innerHTML = "You are a ladybug!";
120    } else {
121      result.innerHTML = "Hmm...not sure. Try again later.";
122
123  }
```

- ❏ **Make observations.** What is the first thing you notice? A LOT of curly brackets! This is a prime bug location. The `updateResult` function has a long conditional statement that evaluates a person's score and returns which bug they are.
- ❏ **Think about what you need.** At this point, it's helpful to move back and think about what you know you need. An open curly bracket at the beginning of the function and a closed curly bracket at the end. If there was nothing inside, it would appear like this:

```
function updateResult(){

}
```

- ❏ **Identify your tools.** What tools do you have or know to solve this? Earlier we learned that we can use our cursor to display the partner of a curly bracket. Let's try this.
- ❏ **Test it.** Check each curly bracket in the `updateResult` function to see which one partners to the first open curly bracket.
- ❏ **Make your changes.** You only need to make one change.
- ❏ **Run the program.** Test your program to see if it runs without an error.

Check your code in the Reference Guide on pg 4-5.

Step 6: Solve the second bug (3-5 mins)

We resolved our first bug triumphantly - now let's look into that second error message:

```
TypeError: document.getElementbyId is not a function at /script.js:20:21
```

We have our first error message! Error messages are the best. They give us a clue to what the problem is and help us locate it. Stop for a moment and try to decode this message:

- ➔ `TypeError` means that the program can't run the operation because a value isn't the type of value the computer expected. In this case, the value is a function name.
- ➔ `document.getElementbyId` is the name of the function that is causing the bug.
- ➔ `is not a function` means the computer doesn't recognize this as a function.
- ➔ `at /script.js` means your error is in the **script.js** file.
- ➔ `20:21` indicates that your error is on line 20 in the 21st space.

Based on what we learned in the last step, we know that the error is in the **script.js** file on line 20.

```
var q1a3 = document.getElementbyId("q1a3"); // Store HTML element for question 1 answer 3 button
```

The error message says that **`document.getElementbyId`** is not a function. But we know that this is a function because we use it in many other places. So what's wrong? Follow these steps to figure it out:

- ❑ **Look for patterns.** Do you see any differences in the other locations where it is used?

```
18  var q1a1 = document.getElementById("q1a1"); // Store HTML element for question 1 answer 1 button
19  var q1a2 = document.getElementById("q1a2"); // Store HTML element for question 1 answer 2 button
20  var q1a3 = document.getElementbyId("q1a3"); // Store HTML element for question 1 answer 3 button
21  var q1a4 = document.getElementById("q1a4"); // Store HTML element for question 1 answer 4 button
```

- ❑ **Try making any changes.** If you think you know where the problem is, fix it.
- ❑ **Run the program.** Test your program to see if it runs without an error.

Check your code in the Reference Guide on pg 6.

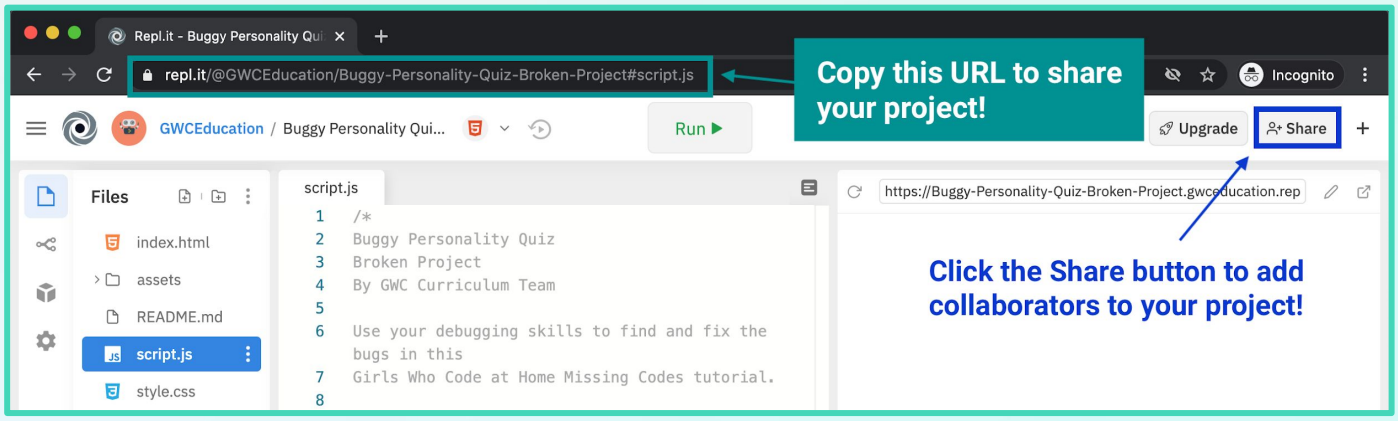
Congratulations!

You successfully debugged all the syntax errors. When you run the program, you should no longer see an error message. However, if you tried to take the quiz, you may have noticed that it still isn't working. That is because there is one bug left: a logic bug. These bugs happen because there is a problem with the program flow or the order that your lines of code execute. In the next activity, we will explore a set of debugging strategies you can use to fix this bug and any other bug you might encounter in the future!

Step 9: Share Your Girls Who Code at Home Project! (5 mins)

View only access (1-2 mins)

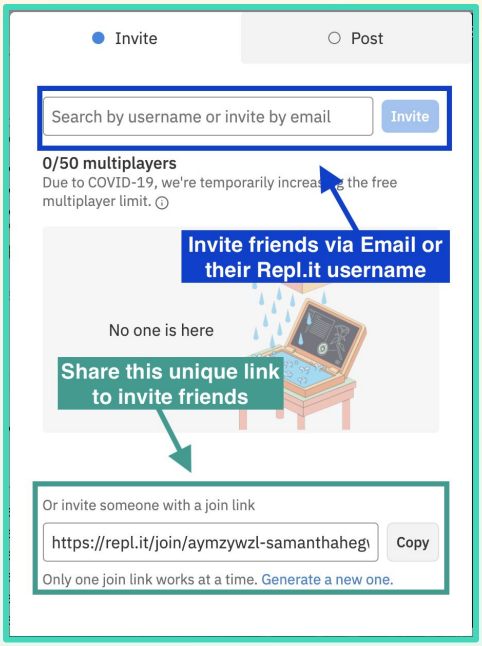
Sharing your work on Repl.it is easy! Just copy and paste the URL address of your Repl project in the web address bar at the top. This will allow others to run your project, view your code, and fork your project and remix on their own. We would love to see your debugging work and we know others would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



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 Debug the Missing Code Part 2!