LEARN INTRODUCTORY JAVASCRIPT BY BUILDING A PYRAMID GENERATOR

Introduction:

JavaScript is a powerful scripting language that you can use to make web pages interactive. It's one of the core technologies of the web, along with HTML and CSS. All modern browsers support JavaScript.

In this practice project, you'll learn fundamental programming concepts in JavaScript by coding your own Pyramid Generator. You'll learn how to work with arrays, strings, functions, loops, `if/else` statements, and more.

Step 1:

JavaScript is the programming language that powers the web. Unlike the HTML and CSS you have learned previously, JavaScript is most commonly used to write logic instead of markup.

In this project, you will learn the basics of Javascript and apply those concepts to building a pyramid generator.

A pyramid generator is a program where you can set the type of character, the count for the pyramid, and the direction of the pyramid. The program will then generate a pyramid based on those inputs.

Click on the "Check your code" button to proceed to the next step and start the project.

Step 2:

One of the most important concepts in programming is variables. A variable points to a specific memory address that stores a value.

Variables are given a name which can be used throughout your code to access that value.

Declaring a variable means giving it a name. In JavaScript, this is often done with the `let` keyword. For example, here is how you would declare a `hello` variable:

Example Code:

```
let hello;
```

Variable naming follows specific rules: names can include letters, numbers, dollar signs, and underscores, but cannot contain spaces and must not begin with a number.

Use the `let` keyword to declare a variable called `character`.

Note: It is common practice to end statements in JavaScript with a semicolon. `;`

Step 3:

Your `character` variable currently does not have a value. You can assign a value using the assignment operator `=`. For example:

Example Code:

```
let hello = "Hello";
```

Assigning a value to a variable at the moment of its declaration is known as initialization.

Initialize your `character` variable by assigning it the value `"Hello"` during its declaration.

Step 4:

JavaScript has seven primitive data types, with String being one of them. In JavaScript, a string represents a sequence of characters and can be enclosed in either single (') or double (") quotes.

Note that strings are immutable, which means once they are created, they cannot be changed. The variable can still be reassigned another value.

Change your "Hello" string to use single quotes.

Step 5:

The console allows you to print and view JavaScript output. You can send information to the console using console.log(). For example, this code will print "Naomi" to the console:

Example Code:

```
let developer = "Naomi";

console.log(developer);
```

The code above accesses the developer variable with its name in the console.log(). Note that the value between the parentheses is the value that will be printed.

Print the value of the character variable to the console. Then, click the "Console" button to view the JavaScript console.

Step 6:

When a variable is declared with the let keyword, you can reassign (or change the value of) that variable later on. In this example, the value of programmer is changed from "Naomi" to "CamperChan".

Example Code:

```
let programmer = "Naomi";

programmer = "CamperChan";
```

Note that when reassigning a variable, you do **not** use the `let` keyword again.

After your `console.log`, assign the value `"World"` to your `character` variable.

Step 7:

Now log your `character` variable to the console again. You should see the string `"Hello"`, then the string `"World"`, in the console.

Step 8:

When variable names are more than one word, there are specific naming conventions for how you capitalize the words. In JavaScript, the convention to use is camel case.

Camel case means that the first word in the name is entirely lowercase, but the following words are all title-cased. Here are some examples of camel case:

Example Code:

```
let variableOne;

let secondVariable;

let yetAnotherVariable;

let thisIsAnAbsurdlyLongName;
```

Use camel case to declare a new `secondCharacter` variable.

Step 9:

When you declare a variable without initializing it, it is considered uninitialized. Currently, your `secondCharacter` variable is uninitialized.

Add a `console.log()` to see what the value of your `secondCharacter` variable is.

Step 10:

The default value of an uninitialized variable is `undefined`. This is a special data type that represents a value that does not have a definition yet.

You can still assign a value to an uninitialized variable. Here is an example:

Example Code:

```
let uninitialized;

uninitialized = "assigned";
```

Assign the string `"Test"` to your `secondCharacter` variable below your declaration. Open the console to see how your log has changed.

Step 11:

You can also assign the value of a variable to another variable. For example:

Example Code:

```
let first = "One";

let second = "Two";

second = first;
```

The `second` variable would now have the value `"One"`.

To see this in action, change your `secondCharacter` assignment from `"Test"` to your `character` variable.

Then open the console to see what gets logged.

Step 12:

You are now ready to declare your next variable. Remove both `console.log` statements, and the `character` reassignment.

Also remove your `secondCharacter` variable, but leave the `character` initialization unchanged.

Step 13:

Before moving forward, you should take a moment to review the concepts you have learned.

Use the `let` keyword to declare a `profession` variable and an `age` variable. Initialize `profession` with the string `"teacher"`, but do not initialize `age` with any value.

Log both of your variables to the console to see the results.

Step 14:

Now that you have reviewed declaration and initialization, remove the code you wrote for your review.

Do not remove your `character` variable.

Step 15:

Use `let` to declare a `count` variable. Assign it the number `8`. When using a number value, you do not use quotes. For example:

Example Code:

```
let money = 100;
```

Step 16:

With the `number` data type, you can perform mathematical operations, like addition. Try printing `count + 1` to the console.

Step 17:

You can also perform subtraction (`-`), multiplication (`*`), and division (`/`). Feel free to experiment with the operators and numbers in your `console.log`. When you are ready to move on, remove the `console.log`.

Step 18:

In programming, you will often need to work with lots of data. There are many data structures that can help you organize and manage your data. One of the most basic data structures is an array.

An array is a non-primitive data type that can hold a series of values. Non-primitive data types differ from primitive data types in

that they can hold more complex data. Primitive data types like strings and numbers can only hold one value at a time.

Arrays are denoted using square brackets (`[]`). Here is an example of a variable with the value of an empty array:

Example Code:

```
let array = [];
```

Declare a `rows` variable and assign it an empty array.

Step 19:

When an array holds values, or elements, those values are separated by commas. Here is an array that holds two strings:

Example Code:

```
let array = ["first", "second"];
```

Change your `rows` declaration to be an array with the strings `"Naomi"`, `"Quincy"`, and `"CamperChan"`. The order of values in an array is important, so follow that order. Remember that strings are case-sensitive.

Step 20:

You can access the values inside an array using the index of the value. An index is a number representing the position of the value in the array, starting from `0` for the first value.

You can access the value using bracket notation, such as `array[0]`.

Use `console.log` and bracket notation to print the first value in your `rows` array.

Step 21:

Arrays are special in that they are considered mutable. This means you can change the value at an index directly.

For example, this code would assign the number `25` to the second element in the array:

Example Code:

```
let array = [1, 2, 3];

array[1] = 25;

console.log(array); // prints [1, 25, 3]
```

Update the **third** element of your `rows` array to be the number `10`. Then print the `rows` array to your console.

Step 22:

Notice how the value inside your `rows` array has been changed directly? This is called mutation. As you learn more about arrays, you will learn when to mutate an array, and when you should not.

Before moving on, this is a great opportunity to learn a common array use. Currently, your code accesses the last element in the array with `rows[2]`. But you may not know how many elements are in an array when you want the last one.

You can make use of the `.length` property of an array - this returns the number of elements in the array. To get the last element of any array, you can use the following syntax:

Example Code:

```
array[array.length - 1]
```

`array.length` returns the number of elements in the array. By subtracting `1`, you get the index of the last element in the array. You can apply this same concept to your `rows` array.

Update your `rows[2]` to dynamically access the last element in the `rows` array. Refer to the example above to help you.

You should not see anything change in your console.

Step 23:

For now, remove your first console log and your `rows[rows.length - 1]` assignment. Leave the second `rows` log statement for later.

Step 24:

In the last few steps, you learned all about working with arrays. Take a moment to review what you have learned.

Start by declaring a `cities` variable and initializing it as an array of the strings `"London"`, `"New York"`, and `"Mumbai"`. Then log that variable to the console.

After logging, change the last element of `cities` to the string `"Mexico City"`, then log the `cities` variable again.

When done correctly, you should see this output in the console.

Example Code:

```
[ "London", "New York", "Mumbai" ]

[ "London", "New York", "Mexico City" ]
```

Step 25:

Now you are ready to move onto the next set of array lessons.

Remove all of your code from the previous step.

Step 26:

A method in JavaScript is a function that's associated with certain values or objects. An example you've already encountered is the `.log()` method, which is part of the `console` object.

Arrays have their own methods, and the first you will explore is the `.push()` method. This allows you to "push" a value to the end of an array. Here is an example to add the number `12` to the end of an array:

Example Code:

```
array.push(12);
```

Use `.push()` to add the string `"freeCodeCamp"` to the end of your `rows` array. Add this code before your `console.log` so you can see the change you made to your array.

Step 27:

Another method essential for this project is the `.pop()` method. It removes the last element from an array and returns that element.

When a method returns a value, you can think of it as giving the value back to you, making it available for use in other parts of your code.

Create a new variable called `popped` and assign it the result of `rows.pop()`. Then, log `popped` to the console.

Step 28:

You should have seen `"freeCodeCamp"` printed to the console. This is because `.pop()` returns the value that was removed from the array - and you pushed `"freeCodeCamp"` to the end of the array earlier.

But what does `.push()` return? Assign your existing `rows.push()` to a new `pushed` variable, and log it.

Step 29:

Were you expecting to see `4` in the console? `.push()` returns the new length of the array, after adding the value you give it.

It is important to be aware of what values a method returns. Take some time to experiment with `.push()` and `.pop()`. When you are ready, remove all of your `.push()` and `.pop()` calls, and your `console.log` statements.

Step 30:

Change your `rows` declaration to be assigned an empty array again.

Also, change your `'Hello'` string to use double quotes again. Generally, it does not matter which of the two you prefer, but you will want to be consistent in that choice throughout your project.

Step 31:

Declaring a variable with the `let` keyword allows it to be reassigned. This means you could change `character` later to be a completely different value.

For this project, you will not want to change these variable values. So instead, you should use `const` to declare them. `const` variables are special.

First, a `const` variable cannot be reassigned like a `let` variable. This code would throw an error:

Example Code:

```
const firstName = "Naomi";

firstName = "Jessica";
```

A `const` variable also cannot be uninitialized. This code would throw an error:

Example Code:

```
const firstName;
```

Replace your `let` keywords with `const`.

Step 32:

You are now ready to start building your pyramid generator. Your `character` variable will serve as the building block for the pyramid.

`"Hello"` might not work very well for that. Change the value of `character` to be the hash character `"#"`.

Step 33:

To generate a pyramid, you will need to create multiple rows. When you have to perform a task repeatedly until a condition is met, you will use a loop. There are many ways to write a loop.

You are going to start with a basic `for` loop. `for` loops use the following syntax:

Example Code:

```
for (iterator; condition; iteration) {

  logic;

}
```

In the upcoming steps, you'll explore each component of a loop in detail. For now, construct a `for` loop that includes the terms `"iterator"`, `"condition"`, and `"iteration"` for the three components. Keep the loop body, the section within the curly braces `{}`, empty.

Step 34:

Your loop now needs a proper iterator. The iterator is a variable you can declare specifically in your `for` loop to control how the loop iterates or goes through your logic.

It is a common convention to use `i` as your iterator variable in a loop. A `for` loop allows you to declare this in the parentheses `()`. For example, here is a `for` loop that declares an `index` variable and assigns it the value `100`.

Example Code:

```
for (let index = 100; "second"; "third") {

}
```

Replace the string `"iterator"` with a `let` declaration for the variable `i`. Assign it the value `0` to start. This will give the `i` variable the value `0` the **first time** your loop runs.

Step 35:

The condition of a `for` loop tells the loop how many times it should iterate. When the `condition` becomes false, the loop will stop.

In JavaScript, a Boolean value can be either `true` or `false`. These are not strings - you will learn more about the difference later on.

For now, you will use the less than operator (`<`). This allows you to check if the value on the left is less than the value on the right. For example, `count < 3` would evaluate to `true` if `count` is `2`, and `false` if `count` is `4`.

Replace your `"condition"` string with a condition to check if `i` is less than `count`.

Step 36:

Your iteration statement will tell your loop what to do with the iterator after each run.

When you reassign a variable, you can use the variable to reference the previous value before the reassignment. This allows you to do things like add three to an existing number. For example, `bees = bees + 3` would increase the value of `bees` by three.

Use that syntax to replace your `"iteration"` string with a reassignment statement that increases `i` by one.

Step 37:

Your loop should now run eight times. Inside the body of the loop, print the value of the `i` iterator and see what happens.

Step 38:

You should see the numbers zero through seven printed in your console, one per line. This will serve as the foundation for generating your pyramid.

Replace your log statement with a statement to push `i` to your `rows` array.

Step 39:

Unfortunately, now you cannot see what your loop is doing.

Use `let` to declare a `result` variable, and assign it an empty string. An empty string is represented by quotation marks with nothing between them, such as `""`.

Step 40:

Add a log statement to print the value of `result`. Depending on which console you use, you may not see anything printed.

Step 41:

To manipulate the `result` string, you will use a different type of loop. Specifically, a `for...of` loop, which iterates over each item in an iterable object and temporarily assigns it to a variable.

The syntax for a `for...of` loop looks like:

Example Code:

```
for (const value of iterable) {

}
```

Note that you can use `const` because the variable only exists for a single iteration, not during the entire loop.

Create a `for...of` loop to iterate through your `rows` array, assigning each value to a `row` variable.

Step 42:

Remember in your previous loop that you used the addition operator `+` to increase the value of `i` by `1`.

You can do a similar thing with a string value, by appending a new string to an existing string. For example, `hello = hello + " World";` would add the string `" World"` to the existing string stored in the `hello` variable. This is called concatenation.

In your `for...of` loop, use the addition operator to concatenate the `row` value to the `result` value.

Step 43:

Now all of your numbers are appearing on the same line. This will not work for creating a pyramid.

You will need to add a new line to each row. However, pressing the return key to insert a line break between quotes in JavaScript will result in a parsing error. Instead, you need to use the special escape sequence `\n`, which is interpreted as a new line when the string is logged. For example:

Example Code:

```
lineOne = lineOne + "\n" + lineTwo;
```

Use a second addition operator to append a new line after the existing `result` value and the added `row` value.

Step 44:

Printing numbers won't result in a visually appealing pyramid. Now that you're outputting the formatted content of your `rows` array, it's time to update your original loop.

Instead of pushing `i` to the array, push the value of your `character` variable.

Step 45:

Now you have a series of `#` characters, but the pyramid shape is still missing. Fortunately, the `i` variable represents the current "row" number in your loop, enabling you to use it for crafting a pyramid-like structure.

To achieve this, you will use the `.repeat()` method available to strings. This method accepts a number as an argument, specifying the number of times to repeat the target string. For example, using `.repeat()` to generate the string `"Code! Code! Code!"`:

Example Code:

```
const activity = "Code! ";

activity.repeat(3);
```

Use the `.repeat()` method on your `character`, and give it `i` for the number.

Step 46:

You're getting closer! At this point, you're encountering what's known as an off-by-one error, a frequent problem in zero-based indexing languages like JavaScript.

The first index of your rows array is 0, which is why you start your for loop with i = 0. But repeating a string zero times results in nothing to print.

To fix this, add 1 to the value of i in your .repeat() call. Do not assign it back to i like you did in your loop conditions.

Step 47:

The logic for formatting this pyramid is likely going to get complicated, which means it's a great time to extract that code into a function.

A function is a block of code that can be reused throughout your application. Functions are declared with the following syntax:

Example Code:

```
function name(parameter) {

}
```

The function keyword tells JavaScript that the name variable is going to be a function. parameter is a variable that represents a value that is passed into the function when it is used. A function may have as many, or as few, parameters as you'd like. Like a for loop, the space between the curly braces is the function body.

Declare a padRow function. Do not create any parameter variables yet. The function body should be empty. Remember that you need to use camel case for your naming convention.

Step 48:

In order to use a function, you need to call it. A function call tells your application to run the code from the function wherever you choose to call it. The syntax for a function call is the function name

followed by parentheses. For example, this code defines and calls a `test` function.

Example Code:

```
function test() {

}

test();
```

Call your `padRow` function.

Step 49:

You are calling your `padRow` function, but not doing anything with that function call. All functions in JavaScript return a value, meaning they provide the defined result of calling them for you to use elsewhere.

To see the result of calling your `padRow` function, declare a `call` variable and assign your existing `padRow` call to that variable.

Step 50:

Now add a log statement to print the value of your `call` variable.

Step 51:

Your `call` variable has an `undefined` value, even though you defined it! This is because your `padRow` function does not currently return a value. By default, functions return `undefined` as their value.

In order to return something else, you need to use the `return` keyword. Here is an example of a function that returns the string `"Functions are cool!"`:

Example Code:

```
function demo() {

  return "Functions are cool!";

}
```

Use the `return` keyword to have your function return the string `"Hello!"`.

Step 52:

When you have a value that is explicitly written in your code, like the `"Hello!"` string in your function, it is considered to be hard-coded. Hard-coding a value inside a function might not make it as reusable as you'd like.

Instead, you can define parameters for the function. Parameters are special variables that are given a value when you call the function, and can be used in your function to dynamically change the result of the function's code.

To add a parameter to your function, you need to add a variable name inside the parentheses. For example, this `demo` function has a `name` parameter:

Example Code:

```
function demo(name) {

}
```

`name` sounds like a useful parameter, so go ahead and add it to your `padRow` function.

Step 53:

A function does not have to return a hard-coded value. It can return the value stored in a variable. Parameters are special variables for a function, so they can also be returned.

Change your `padRow` function to `return` the `name` parameter directly.

Step 54:

If you open your console again, you'll see that your `padRow` function is returning `undefined`, even though you defined a return value! This is because parameters need to be given a value when you **call** the function.

When you pass a value to a function call, that value is referred to as an argument. Here is an example of calling a `demo` function and passing `"Naomi"` as the argument for the `name` parameter.

Example Code:

```
function demo(name) {

  return name;

}

demo("Naomi");
```

Pass your own name as the argument for the `name` parameter in your `padRow` call. Remember that your name is a string, so you'll need to use quotes.

Step 55:

Before moving on, take a moment to review how functions work.

Declare a function named addTwoNumbers. This function should take two arguments and return the sum of those two arguments.

Your function should not use hard-coded values. An example of a hard-coded function might be:

Example Code:

```
function sayName(firstName, lastName) {

  return "John Doe";

}

sayName("Camper", "Cat");
```

This function would return "John Doe" regardless of the arguments passed to the parameters firstName, and lastName, so "John Doe" is considered a hard-coded value.

Declare a sum variable and assign it the value of calling your addTwoNumbers function with 5 and 10 as the arguments. Log the sum variable to the console.

Step 56:

With that quick review complete, you should remove your addTwoNumbers function, sum variable, and log statement.

Step 57:

Variables in JavaScript are available in a specific scope. In other words, where a variable is declared determines where in your code it can be used.

The first scope is the global scope. Variables that are declared outside of any "block" like a function or for loop are in the global

scope. Your `character`, `count`, and `rows` variables are all in the global scope.

When a variable is in the global scope, a function can access it in its definition. Here is an example of a function using a global `title` variable:

Example Code:

```
const title = "Professor ";

function demo(name) {

  return title + name;

}

demo("Naomi")
```

This example would return `"Professor Naomi"`. Update your `padRow` function to return the value of concatenating your `character` variable to the beginning of the `name` parameter.

Step 58:

Variables can also be declared inside a function. These variables are considered to be in the local scope, or block scope. A variable declared inside a function can only be used inside that function. If you try to access it outside of the function, you get a reference error.

To see this in action, use `const` to declare a `test` variable in your `padRow` function. Initialise it with the value `"Testing"`.

Then, below your function, try to log `test` to the console. You will see an error because it is not defined outside of the function's local scope. Remove that `console.log` to pass the tests and continue.

Step 59:

Values returned out of a function are used by calling the function. You can use the function call directly as the value it returns, or capture the returned value in a variable. This way, you can use the value assigned to a locally scoped variable, outside the function it was created in.

Example Code:

```
function getName() {

  const name = "Camper cat";

  return name;

}

console.log(getName()); // "Camper cat"

const capturedReturnValue = getName();

console.log(capturedReturnValue); // "Camper cat"

console.log(name); // reference error
```

To use your `"Testing"` value, return it out of the `padRow` function by updating your `return` statement to return only the `test` variable.

Step 60:

Below the `return` statement, log the string `"This works!"` to the console.

After doing that, you will see that the string `"This works!"` does not display in the console, and the `console.log("This works!")` line is greyed out.

Copy the console log and paste it above the `return` statement. Now, the string `"This works!"` should appear in the console.

An important thing to know about the `return` keyword is that it does not just define a value to be returned from your function, it also stops the execution of your code inside a function or a block statement. This means any code after a `return` statement will not run.

Step 61:

Now your `call` variable has the value `"Testing"`. But your function is no longer using the `name` parameter.

Remove the `name` parameter from your function declaration, then remove your `"CamperChan"` string from the `padRow` call.

Also, remove both `console.log` from the `padRow` function.

Step 62:

Because your function was no longer using the parameter, changing the argument did not affect it.

Go ahead and remove the test declaration and return statement from your padRow function, so the function is empty again.

Step 63:

As expected, your function now returns `undefined` again. Your `call` variable is not necessary any more, so remove the `call` declaration and the `console.log` for the `call` variable.

Step 64:

In order to know how to format a row, your padRow function will need to know which row number you are on, and how many rows in total are being generated.

The best way to do this is by creating function parameters for them. Give your padRow function a rowNumber and rowCount parameter. Multiple parameters are separated by a comma:

Example Code:

```
function name(first, second) {

}
```

Step 65:

Remember in an earlier step, you learned about return values. A function can return a value for your application to consume separately.

In a function, the return keyword is used to specify a return value. For example, this function would return the value given to the first parameter:

Example Code:

```
function name(parameter) {

  return parameter;

}
```

Use the return keyword to return the value of the character variable, repeated rowNumber times.

Step 66:

A function call allows you to actually use a function. You may not have been aware of it, but the methods like `.push()` that you have been using have been function calls.

A function is called by referencing the function's name, and adding `()`. Here's how to call a `test` function:

Example Code:

```
test();
```

Replace the `character.repeat(i + 1)` in your `.push()` call with a function call for your `padRow` function.

Step 67:

Your `padRow` function has two parameters which you defined. Values are provided to those parameters when a function is called.

The values you provide to a function call are referred to as arguments, and you pass arguments to a function call. Here's a function call with `"Hello"` passed as an argument:

Example Code:

```
test("Hello");
```

Pass `i + 1` and `count` as the arguments to your `padRow` call. Like parameters, arguments are separated by a comma.

Step 68:

You should now see the same bunch of characters in your console. Your `padRow` function is doing the exact same thing you were doing earlier, but now it's in a reusable section of its own.

Use the addition operator to concatenate a single space `" "` to the beginning and end of your repeated `character` string.

Remember that you can use the + operator to concatenate strings like this:

Example Code

```
" " + "string"
```

Step 69:

Now it is time for a bit of math. Consider a three-row pyramid. If we want it centered, it would look something like:

Example Code:

```
··#··

·###·

#####
```

Empty spaces have been replaced with interpuncts, or middle dots, for readability. If you extrapolate the pattern, you can see that the spaces at the beginning and end of a row follow a pattern.

Update your blank space strings to be repeated `rowCount - rowNumber` times.

Open up the console to see the result.

Step 70:

You can pass full expressions as an argument. The function will receive the result of evaluating that expression. For example, these two function calls would yield the same result:

Example Code:

```
test(2 * 3 + 1);
```

```
test(7);
```

Looking at the pattern again:

Example Code:

```
··#··
·###·
#####
```

Update the `character` value to be repeated `2 * rowNumber - 1` times.

Open up the console again to see the updated result.

Step 71:

Your pyramid generator now functions as expected. But this is an excellent opportunity to further explore the code you have written.

The addition operator is not the only way to add values to a variable. The addition assignment operator can be used as shorthand to mean "take the original value of the variable, add this value, and assign the result back to the variable." For example, these two statements would yield the same result:

Example Code:

```
test = test + 1;
```

```
test += 1;
```

Update your iteration statement in the `for` loop to use addition assignment.

Step 72:

Because you are only increasing `i` by `1`, you can use the increment operator `++`. This operator increases the value of a variable by 1, updating the assignment for that variable. For example, `test` would become `8` here:

Example Code:

```
let test = 7;

test++;
```

Replace your addition assignment with the increment operator for your loop iteration.

Step 73:

Rather than having to pass `i + 1` to your `padRow` call, you could instead start your loop at `1`. This would allow you to create a one-indexed loop.

Update your iterator to start at `1` instead of `0`.

Step 74:

The pyramid looks a little funny now. Because you are starting the loop at `1` instead of `0`, you do not need to add one to `i` when you pass it to `padRow`.

Update the first argument of your `padRow` call to be `i`.

Step 75:

Unfortunately, now the bottom of the pyramid has disappeared. This is because you have created another off-by-one error.

Your original loop went for `i` values from `0` to `7`, because `count` is `8` and your condition requires `i` to be less than `count`. Your loop is now running for `i` values from `1` to `7`.

Your loop needs to be updated to run when `i` is `8`, too. Looking at your logic, this means your loop should run when `i` is less than or equal to `count`. You can use the less than or equal to operator `<=` for this.

Update your loop condition to run while `i` is less than or equal to `count`.

Step 76:

Comments can be helpful for explaining why your code takes a certain approach, or leaving to-do notes for your future self.

In JavaScript, you can use `//` to leave a single-line comment in your code.

Add a single-line comment above your `for` loop to remind yourself to change the code to a different kind of loop.

Step 77:

JavaScript also has support for multi-line comments. A multi-line comment starts with `/*` and ends with `*/`.

Unlike a single-line comment, a multi-line comment will encapsulate multiple lines.

Use `/*` and `*/` to turn your current `for` loop, including the body, into a multi-line comment.

Step 78:

Your pyramid has disappeared again. That's okay - that is to be expected.

Before you create your new loop, you need to learn about `if` statements. An `if` statement allows you to run a block of code only when a condition is met. They use the following syntax:

Example Code:

```
if (condition) {

  logic

}
```

Create an `if` statement with the boolean `true` as the condition. In the body, print the string `"Condition is true"`.

Step 79:

You'll see the string printed in the console, because `true` is in fact true.

Change the condition of your `if` statement to the boolean `false`.

Step 80:

Now the string is no longer printing, because `false` is not `true`. But what about other values?

Try changing the condition to the string `"false"`.

Step 81:

The text has appeared again! This is because `"false"` is a string, which when evaluated to a boolean becomes `true`. This means `"false"` is a truthy value.

A truthy value is a value that is considered true when evaluated as a boolean. Most of the values you encounter in JavaScript will be truthy.

A falsy value is the opposite - a value considered false when evaluated as a boolean. JavaScript has a defined list of falsy values. Some of them include `false`, `0`, `""`, `null`, `undefined`, and `NaN`.

Try changing your `if` condition to an empty string `""`, which is a falsy value.

Step 82:

The text is gone again! Empty strings evaluate to `false`, making them a falsy value. You will learn more about truthy and falsy values in future projects.

In addition to `if` statements, JavaScript also has else if statements. `else if` statements allow you to check multiple conditions in a single block of code.

Here is the syntax for an `else if` statement:

Example Code:

```
if (condition1) {

  // code to run if condition1 is true

} else if (condition2) {

  // code to run if condition2 is true

} else if (condition3) {

  // code to run if condition3 is true

}
```

If the first condition is `false`, JavaScript will check the next condition in the chain. If the second condition is `false`, JavaScript will check the third condition, and so on.

Below your `if` statement, add an `else if` statement that checks if `5` is less than `10`. Then inside the body of the `else if` statement, log the string `"5 is less than 10"` to the console.

Check the console to see the results.

Step 83:

Sometimes you will want to run different code when all of the `if...else if` conditions are `false`. You can do this by adding an `else` block.

An `else` block will only evaluate if the conditions in the `if` and `else if` blocks are not met.

Here the `else` block is added to the `else if` block.

Example Code:

```
if (condition) {

  // this code will run if condition is true

} else if (condition2) {

  // this code will run if the first condition is false

} else {

  // this code will run

  // if the first and second conditions are false

}
```

Add an `else` block to the `else if` block. Inside the `else` block, log the string `"This is the else block"` to the console.

To see the results in the console, you can manually change the `<` in the `else if` statement to `>`. That will make the condition `false` and the `else` block will run.

Step 84:

Now that you have practiced working with `if...else if...else` statements, you can remove them from your code.

Once you complete that, use `let` to declare a `continueLoop` variable and assign it the boolean `false`. Then use `let` to declare a `done` variable and assign it the value `0`.

Step 85:

A `while` loop will run over and over again until the `condition` specified is no longer true. It has the following syntax:

Example Code:

```
while (condition) {

  logic;

}
```

Use that syntax to declare a `while` loop with `continueLoop` as the condition. The body should be empty.

Step 86:

Right now, if you change `continueLoop` to true, your `while` loop will run forever. This is called an infinite loop, and you should be careful to avoid these. An infinite loop can lock up your system, requiring a full restart to escape.

To avoid this, start by using the increment operator to increase the value of the `done` variable inside your loop.

Step 87:

The equality operator `==` is used to check if two values are equal. To compare two values, you'd use a statement like `value == 8`.

Below `done++` inside your loop, add an `if` statement. The statement should check if `done` is equal to `count` using the equality operator.

Step 88:

The equality operator can lead to some strange behavior in JavaScript. For example, `"0" == 0` is true, even though one is a string and one is a number.

The strict equality operator `===` is used to check if two values are equal and share the same type. As a general rule, this is the equality operator you should always use. With the strict equality operator, `"0" === 0` becomes false, because while they might have the same value of zero, they are not of the same type.

Update your `done == count` condition to use the strict equality operator.

Step 89:

When `done` has reached the value of `count`, we want the loop to stop executing.

Inside your `if` body, assign the boolean `false` to your `continueLoop` variable.

Step 90:

To make your pyramid generate again, push the result of calling `padRow` with `done` and `count` as the arguments to your `rows` array, similar to what you did in your first loop.

Step 91:

The strict inequality operator `!==` allows you to check if two values are not equal, or do not have the same type. The syntax is similar to the equality operator: `value !== 4`.

Update your `while` loop condition to check if `done` is not equal to `count`.

Step 92:

Since you have moved the comparison into the `while` condition, you can remove your entire `if` statement.

Step 93:

Your loop is no longer relying on the `continueLoop` variable. This makes the variable an unused declaration. Generally, you want to avoid unused declarations to prevent future confusion.

Remove your `continueLoop` variable.

Step 94:

Your pyramid generator is still working. However, it could be possible to end up with an infinite loop again.

Because you are only checking if `done` is not equal to `count`, if `done` were to be **larger** than `count` your loop would go on forever.

Update your loop's condition to check if `done` is less than or equal to `count`.

Step 95:

Using `done` to track the number of rows that have been generated is functional, but you can actually clean up the logic a bit further.

Arrays have a special `length` property that allows you to see how many values, or elements, are in the array. You would access this property using syntax like `myArray.length`.

Note that `rows.length` in the `padRow` call would give you an off-by-one error, because `done` is incremented *before* the call.

Update your condition to check if `rows.length` is less than `count`.

Step 96:

Replace the `done` reference in your `padRow` call with `rows.length + 1`.

Step 97:

Now you no longer need your `done` variable. Remove the increment operation from your loop, and the variable declaration for `done`.

Step 98:

That's a very clean and functional loop. Nice work! But there's still more to explore.

Use a multi-line comment to comment out your `while` loop.

Step 99:

What if you made your pyramid upside-down, or inverted? Time to try it out!

Start by creating a new `for` loop. Declare your iterator `i` and assign it the value of `count`, then use the boolean `false` for your condition and iteration statements.

Step 100:

Because you are going to loop in the opposite direction, your loop needs to run while `i` is greater than `0`. You can use the greater than operator `>` for this.

Set your loop's condition to run when `i` is greater than `0`.

Step 101:

Your iteration statement is also going to be different. Instead of adding `1` to `i` with each loop, you need to subtract `1`.

Like you did earlier with `i = i + 1`, update your iteration statement to give `i` the value of subtracting `1` from itself.

Step 102:

Again, push the result of calling `padRow` with your `i` and `count` variables to your `rows` array.

Open up the console to see the upside-down pyramid.

Step 103:

Just like addition, there are different operators you can use for subtraction. The subtraction assignment operator `-=` subtracts the given value from the current variable value, then assigns the result back to the variable.

Replace your iteration statement with the correct statement using the subtraction assignment operator.

Step 104:

Because you are only subtracting one from `i`, you can use the decrement operator `--`.

Replace your subtraction assignment with the decrement operator.

Step 105:

Use a multi-line comment to comment out this loop as well, to prepare for the next approach.

Step 106:

You can actually build the inverted pyramid without needing to loop "backwards" like you did.

To do this, you'll need to learn a couple of new array methods. Start by using `const` to declare a `numbers` variable. Assign it an array with the elements `1`, `2`, and `3`. Then log the `numbers` array.

Step 107:

The `.unshift()` method of an array allows you to add a value to the **beginning** of the array, unlike `.push()` which adds the value at the end of the array. `.unshift()` returns the new length of the array it was called on.

Example Code:

```
const countDown = [2, 1, 0];

const newLength = countDown.unshift(3);

console.log(countDown); // [3, 2, 1, 0]

console.log(newLength); // 4
```

Use `const` to declare an `unshifted` variable, and assign it the result of calling `.unshift()` on your `numbers` array. Pass `5` as the argument. Then print your `unshifted` variable.

Step 108:

Arrays also have a `.shift()` method. This will remove the first element of the array, unlike `.pop()` which removes the last element. Here is an example of the `.shift()` method:

Example Code:

```
const numbers = [1, 2, 3];

numbers.shift();
```

The `numbers` array would be `[2, 3]`.

Directly below your `numbers` array, declare a `shifted` variable and assign it the result of calling `.shift()` on the `numbers` array. On the next line, log the `shifted` variable to the console.

Step 109:

Now that you've tried these methods, you can do another inverted pyramid approach. But first you need to clean up your experimentation.

Remove your `numbers` array, and the method calls and log calls.

Step 110:

Sometimes you may wish to bring back previous code that you commented out. You can do so by removing the `/*` and `*/` around that code. This is called uncommenting.

Uncomment only your first `for` loop. Leave the single line comment and the other two multi line comments in place.

Step 111:

Your pyramid is no longer inverted. This is because you are adding new rows to the end of the array.

Update your loop body to add new rows to the beginning of the array.

Step 112:

What if you had a way to toggle between an inverted pyramid and a standard pyramid?

Start by declaring an `inverted` variable, and assigning it the value `true`. You are not changing this variable in your code, but you will need to use `let` so our tests can modify it later.

Step 113:

Use an `if` statement to check if `inverted` is true. Remember that you do not need to use an equality operator here.

Step 114:

Now move your `.unshift()` call into your `if` block.

Step 115:

If your pyramid is not inverted, then you will want to have an `else` block that builds the pyramid in the normal order.

In earlier steps, you learned how to work with `else` statement like this:

Example Code:

```
if (condition) {

  // if condition is true, run this code

} else {

  // if condition is false, run this code

}
```

Add an `else` block to your `if` block.

Step 116:

When `inverted` is false, you want to build a standard pyramid. Use `.push()` like you have in previous steps to achieve this.

Step 117:

Your pyramid generator is now in a finished state, with more functionality than you originally planned! The next step is to clean up your code.

Remove all comments, both single- and multi-line, from your code.

Step 118:

Nice work! Experiment with different values for your `character`, `count`, and `inverted` variables.

When you are ready to move on to your next project, set `character` to `"!"`, `count` to `10`, and `inverted` to `false` to continue.

Congratulations on completing your first JavaScript project!