3.3.3 Add the Start Game Function

The first thing we need to do is encapsulate our game logic in a function that can be called whenever we want to restart the game. Without this function, we would have to duplicate our code for every time we want the player to play again. That would be a terrible approach because we don't know how many times a player will want to retry.

REWIND

We've mentioned the DRY, or Don't Repeat Yourself, principle a few times. This is a great concept to keep in the back of your mind as you work. Less code is easier to maintain!

Let's start making edits to our `game.js` file. Wrap the current `for` loop in a `startGame()` function like so:

```
// function to start a new game
var startGame = function() {
  for (var i = 0; i < enemyNames.length; i++) {
    if (playerHealth > 0) {
      window.alert("Welcome to Robot Gladiators! Round " + (i + 1));

      var pickedEnemyName = enemyNames[i];

      enemyHealth = 50;

      fight(pickedEnemyName);
    }
    else {
      window.alert("You have lost your robot in battle! Game Over!");
      break;
    }
  }
};
```

If you save and refresh the browser, nothing will happen after you enter your robot's name, because every actionable line of code is waiting to be called in the `fight()` function or the `startGame()` function. You need to call `startGame()` first to get the ball rolling.

Add this line to the bottom of `game.js`:

```
// start the game when the page loads
startGame();
```

PAUSE

Why did the call to `startGame()` need to go at the bottom?

Function expressions (e.g., `var startGame = function() { };`) need to be defined before they can be used.

Hide Answer

This will return the game to a playable state, but the game logic still only runs once. Fortunately, with the `startGame()` function in place, we can easily call it again to play another round.

Inside the `startGame()` function and after the `for` loop, call the function:

```
var startGame = function() {
  for (var i = 0; i < enemyNames.length; i++) {
    ...
  }

  // play again
  startGame();
};
```

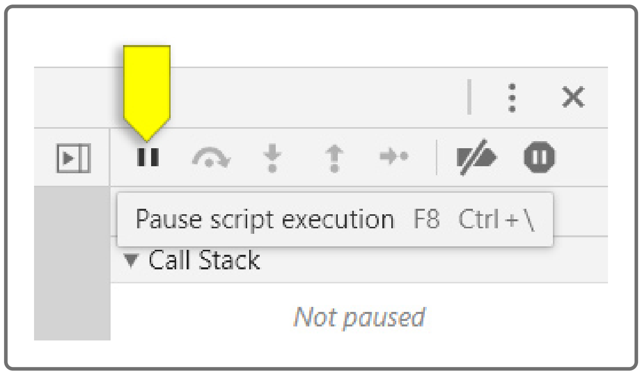
Fair warning—this will create an infinite loop because we haven't defined any condition where `startGame()` doesn't get called after completing the game. If you already tested the game in the browser and are stuck in the loop, don't panic! To stop it, close the browser tab completely and reopen the `index.html` file or use the Chrome DevTools to pause the JavaScript code.

IMPORTANT

> The DevTools approach will only work if the DevTools were already open before the loop happened. If the loop is preventing you from opening and using the DevTools, you can temporarily comment out the first call to `startGame()`:
>
> ```
> // startGame();
> ```

> Close the tab and reopen `index.html` in Chrome to stop the looping alerts.

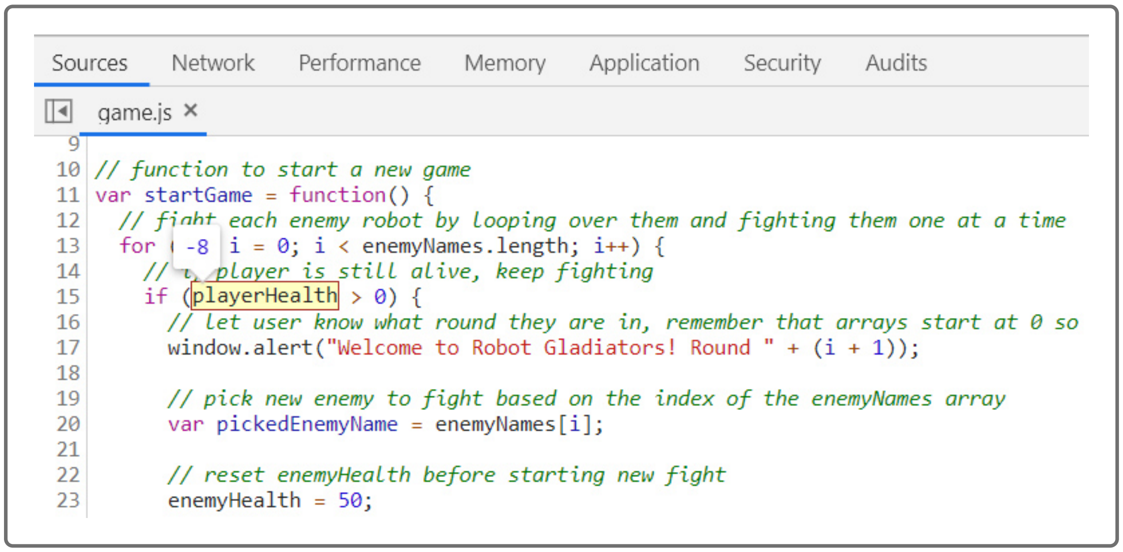
To stop the game using DevTools, navigate to the Sources tab. On the right side of the window, there is a pause button that will say "Pause script execution" when you point your cursor at it:



Click the button to tell Chrome to pause the currently running JavaScript code. This will put the page in a pause state similar to the `debugger` statement. From here, you can stop and reload the page as normal.

Another problem you might have noticed as we continually restart the game is that the player keeps the same health and money values from the previous session. This means if the player runs out of health during their first play, they'll start with zero health the second time they play, and they will immediately lose! To verify if that's really what's happening, use `console.log()` or the DevTools debugger to track `playerHealth`.

If you choose to use the DevTools debugger method to check `playerHealth`, add the `debugger` keyword on the first line of `startGame()`. When you reload `index.html`, the debugger console will open as soon as `startGame()` is called. Point the cursor at the variable name to see its current value:



```
// function to start a new game
var startGame = function() {
  // fight each enemy robot by looping over them and fighting them one at a time
  for (  i = 0; i < enemyNames.length; i++) {
    // if player is still alive, keep fighting
    if (playerHealth > 0) {
      // let user know what round they are in, remember that arrays start at 0 so
      window.alert("Welcome to Robot Gladiators! Round " + (i + 1));

      // pick new enemy to fight based on the index of the enemyNames array
      var pickedEnemyName = enemyNames[i];

      // reset enemyHealth before starting new fight
      enemyHealth = 50;
```

We'll need to reset these player variables every time `startGame()` is called. It will be similar to resetting the `enemyHealth` variable in the `for` loop.

To do this, add these lines at the beginning of the `startGame()` function:

```
var startGame = function() {
  // reset player stats
  playerHealth = 100;
  playerAttack = 10;
  playerMoney = 10;

  // other logic remains the same...
};
```

This will ensure that the player starts with the correct values every time they play the game.

Note that the `startGame()` function is allowed to read and update these three variables because of the **scope** they were declared in. Variables like `playerHealth` that are declared outside of any functions are considered **global**, meaning that any function can access them. Compare this to the following line that's inside the `startGame()` function:

```
var pickedEnemyName = enemyNames[i];
```

The `pickedEnemyName` variable only exists within the scope of the `startGame()` function, so other functions like `fight()` can't access it. This is called **local scope**.

Variable Scope

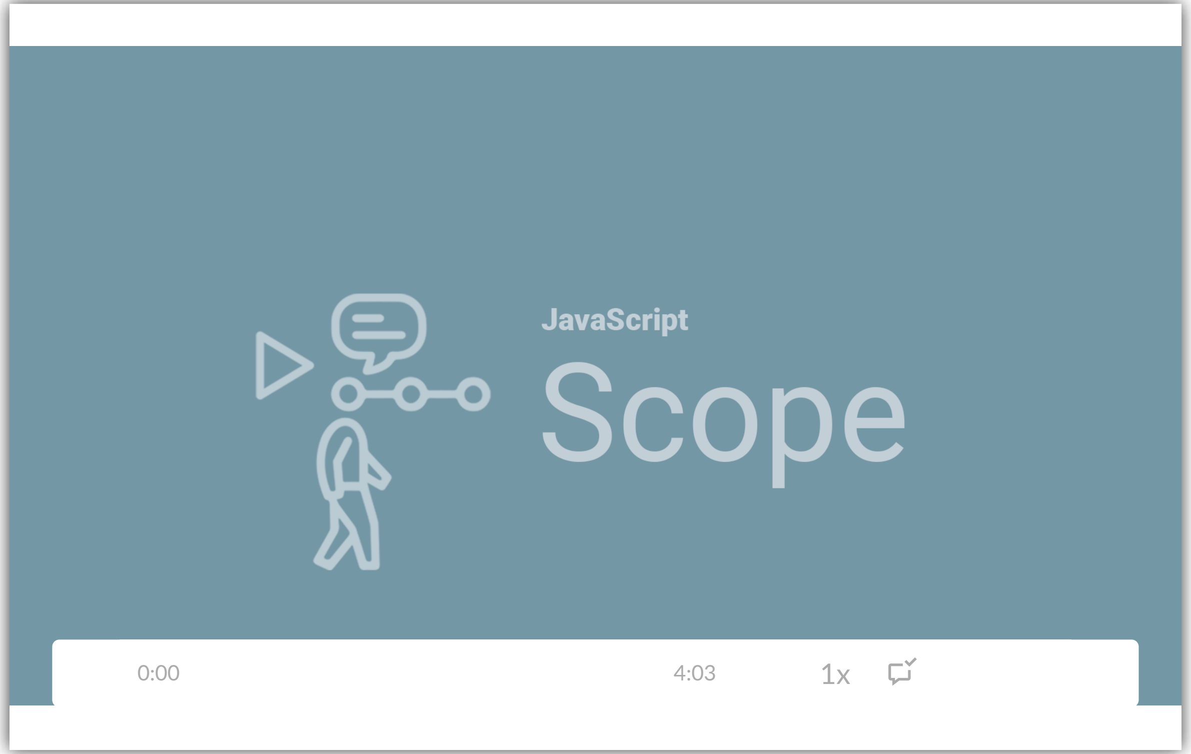
Let's look at one more example of variable scope before moving on to the next step of the project:

```
var a = "a";

var logStuff = function() {
  var b = "b";
  console.log(a);
  console.log(b);
};

console.log(a);
console.log(b); // error
```

The variable `a` was declared outside any function, making it global in scope. Therefore, the `logStuff()` function can access it. The variable `b` was declared inside a function, so only that function has access to it. If we were to try to access `b` outside of `logStuff()`, we'd get an error.

You might be thinking, "Why don't we just use global variables for everything?" That might be fine for a small app, but in larger apps this would lead to having many global variables, which would be hard to keep track of. How would we know which functions are supposed to use which variables? We could run into issues where a function accidentally overwrites a variable that was intended for a different function.

Scope can be tricky, so it's something we'll continue to practice in this and future JavaScript projects. In fact, the next function we write will touch on this, too! Before we get into it, let's check out this video to help understand scope a bit more:



HIDE PRO TIP

Scope presents a good case for writing unique, meaningful variable names. If a global variable and a local function variable have the same name, the local variable will take precedence. To avoid such confusion, it's best not to reuse the same variable names.