3.1.5 Use JavaScript Functions to Communicate with the Player

When we added the `window.alert()` and opened the HTML file in the browser, we ran into a web development staple: an alert dialog window. But where do alerts come from, and what's the `window` in `window.alert()`? Let's dissect that code.

For now, all we need to know about `window` is that it refers to the browser itself. Whenever we write JavaScript and run our code in a browser, this `window` will always be present, as we'll see later.

DEEP DIVE

DEEP DIVE

To learn more, see the **MDN web docs on the window.alert() function (https://developer.mozilla.org/en-US/docs/Web/API/Window/alert)**.

Introducing Functions

Next let's look at the `alert()`. An alert is a function built into the browser that allows the developer to send messages to the user. A **function** is a predefined action that we can call or invoke later in our code. The `alert()` function is provided by the browser, but we can also define our own functions.

After the `alert` function, you'll notice a set of parentheses that enclose a sentence in double quotes: `("This is an alert! JavaScript is running!")`

To run a JavaScript function, we need parentheses—even if there's nothing between them. When the browser reads the function name and sees the parentheses, it knows to execute the function.

Placing content between the parentheses is called **passing an argument into the function**. This content will be displayed as the dialog's message every time it's used.

Lastly, a semicolon `;` closes out the alert function's code. This semicolon tells the browser that this particular line is completed and any code after it is a new piece of code, called an **expression**.

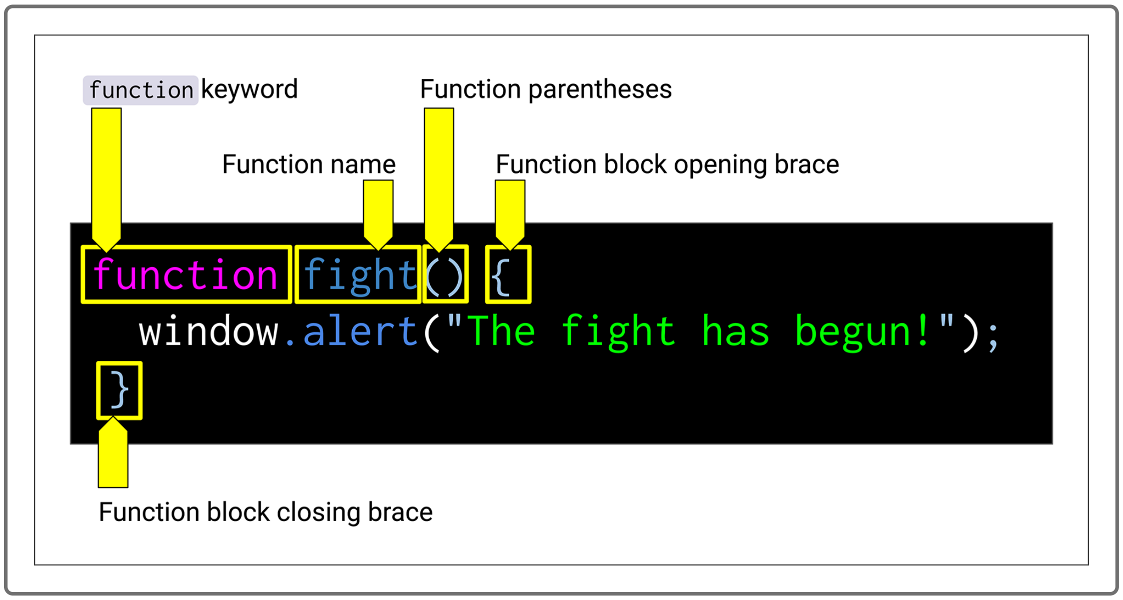
Before we dive further into passing arguments into a function, let's create our own function by replacing the code `game.js` with the following:

```
// this creates a function named "fight"

function fight() {
  window.alert("The fight has begun!");
}
```

Congratulations! You just wrote your first of many custom JavaScript functions.

Let's dissect this syntax. The following image breaks it down:



```
function fight() {
  window.alert("The fight has begun!");
}
```

When we declare a function, we use the keyword `function` followed by the name we want to give the function. We can name our functions anything we want, but best practice is to name them for their functionality. For example, we named this function `fight`, but we could instead name it `x`. We might remember what `x` does, but will other developers immediately know?

After naming the function, we need to include parentheses. Our function doesn't need data passed into it the same way that the alert does, but the parentheses are required regardless.

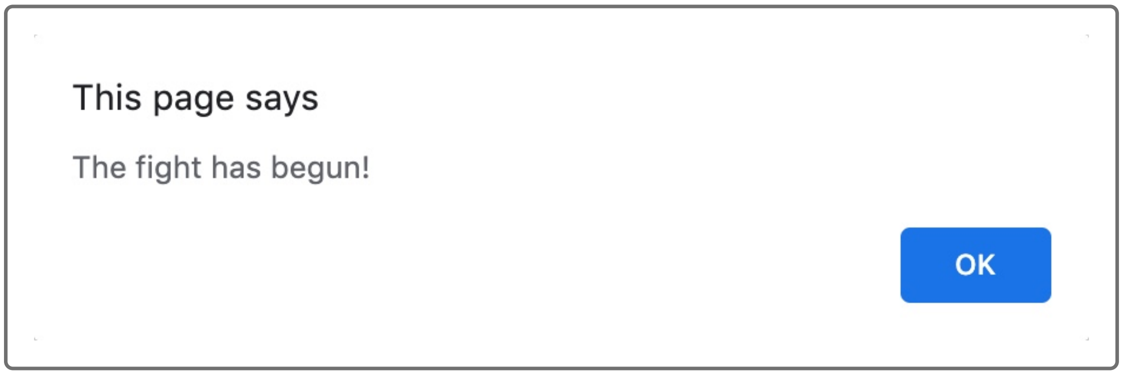
The **curly braces** `{ }` wrap the code belonging to this function. The function will run any code within the braces, and it won't run any code outside them. The code between these curly braces is called a **code block.**

Save `game.js` and refresh `index.html` in the browser. Did you get an alert stating that "The fight has begun!"? No? Why not?!

We defined our function but didn't call it. To call the function, add the following to the bottom of `game.js`:

```
fight();
```

Then save the file and refresh `index.html` in your browser. You should see something like the following:


This page says

The fight has begun!

This step might seem unnecessary—why would we create a function that calls an alert when we could just call the alert? This is because we will add more and more code to this main function throughout the build of this game. So it won't seem unnecessary for long!

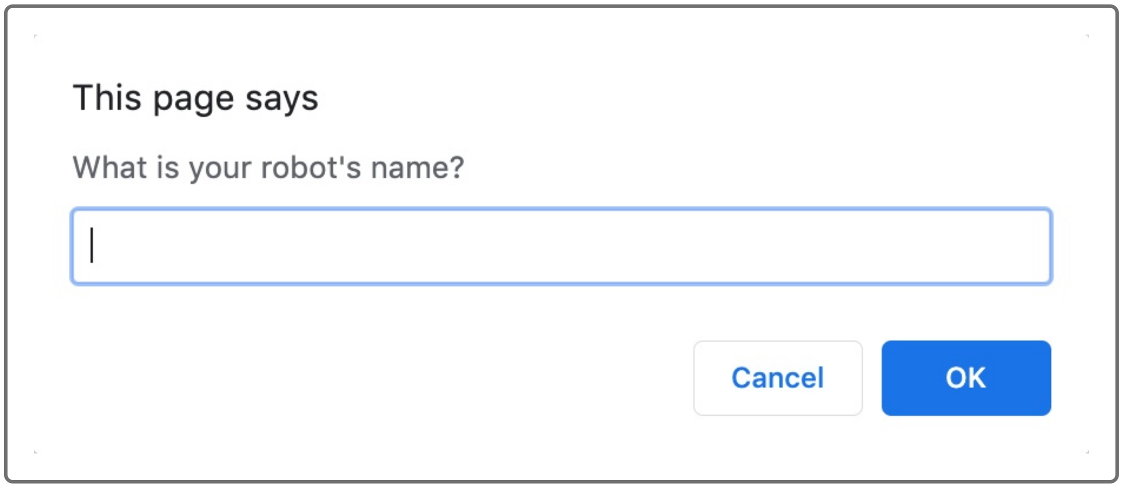
Storing User Input

So far, we've communicated with the user in one direction: program to user. But how do we use JavaScript to ask the user for information and capture their input? This is a game, after all—the user should have some say!

Like the alert function we used previously, we can use another `window` function called a **prompt()**. Let's add this line of code to the top of our `game.js` file:

```
window.prompt("What is your robot's name?");
```

Now save `game.js` and refresh `index.html` in the browser, and you should see something like this:


This page says

What is your robot's name?

Now the user can add their information to our program. Whereas the `window.alert()` function sent a message from the program to the user through the browser's window, the `window.prompt()` function asks the user to send some information back to the program.

DEEP DIVE

DEEP DIVE

To learn more, see the **MDN web docs on the window.prompt() function (https://developer.mozilla.org/en-US/docs/Web/API/Window/prompt)**.

Enter your robot's name in the input field in this prompt dialog window, and click the OK button (the name of this button might vary depending on the browser you're using). You will then see the `alert()` dialog created by our `fight()` function. The user has now provided the JavaScript program with that information, and the program can use it throughout the code.

DEEP DIVE

DEEP DIVE

The `window` is a collection of functions and many other properties built natively into the browser. Whenever we open our program in the browser, our JavaScript file's code executes in the context of the `window`. This way, our code can always access the tools and information (called "properties") the `window` provides.

A collection of functionality and data that's accessed through one "name" (e.g., `window`) is an **object**—which we'll learn more about later when we create one. To access an object's data, we name the object, then use a period followed by the property name or function name we're looking for. Using a period between the object name and property is called **dot notation**.

So the `window.prompt()` instructs the browser to find the `window` object, then find and execute the `prompt()` function that belongs to the `window` object. When a function like `prompt()` belongs to an object, it's called a **method**.

Even though our JavaScript code can access these properties, the `window` object isn't actually part of the JavaScript language. We can use it because browsers have built this functionality into their software and put it behind the name `window`. This is called a **web application**

programming interface (API), which we'll learn more about throughout this project and in the coming weeks.

For more information, see the **MDN web docs on the Window object (https://developer.mozilla.org/en-US/docs/Web/API/Window)**.

So how do we use this data? When a user provides an answer to the prompt, where does it go?

If we were to run this program now and fill out the prompt, our response would be sent back to the browser—but there is nothing in place to capture and store that response.

We need to capture this data somehow. To do that, we'll use an expression. Let's edit the `window.prompt()` line in `game.js` to look like this:

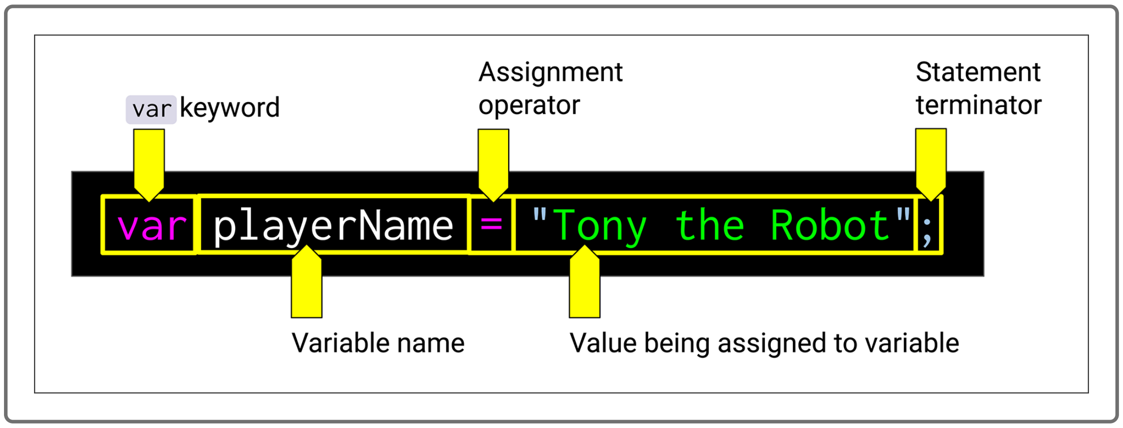
```
var playerName = window.prompt("What is your robot's name?");
```

Introducing Variables

We just used a critical part of JavaScript: **variables**. A variable is a named location for a value that gets stored in the browser's memory when a program is run. Because the data coming from the prompt is user driven, the value is unpredictable. Giving the data a variable name (in our case, `playerName`) allows us to refer to it consistently by just calling on that name.

Now when we answer the prompt with our robot's name, the code we just placed into `game.js` will store that response under the variable name `playerName`. So whenever we want to use our robot's name in our program, we can just refer to it as `playerName`.

Let's dissect basic syntax for creating a variable. In the following example we will declare a variable `playerName` and **assign** it the value "Tony the Robot":


var keyword
Assignment operator
Statement terminator
var playerName = "Tony the Robot";
Variable name
Value being assigned to variable

- **The `var` keyword:** Just like the `function` keyword we used previously, `var` is a keyword built into the JavaScript language. Whenever it is used, it tells the program that we are creating a new variable and the next word is going to be the name of the variable.

- **Variable name:** This is the actual name that will store the information assigned to the variable. When the browser reads this line, it will store this name to recall it later in the program. As a rule, give each variable a meaningful (but concise) name, and use camel casing for more than one word. In **camel casing**, the first letter of a name is lowercase, but the first letter of every word that follows is uppercase—which is useful because the JavaScript language can't interpret hyphens. We used camel case to create our `playerName` variable above, and there are more examples in the code below.

- **Assignment operator `=`:** The assignment operator is a single equal sign that is used to set the value to a variable name. Everything on the left of the operator sets up the variable name for the program. Everything on the right of the operator is the value being stored into the variable name. This value can come in varying forms, called **data types**. Here are some examples of variables with different data types with an explanation of each one in the comments:

```
// This is a String data type; it must be wrapped in double quotes ("
var stringDataType = "This is a string, which is a fancy way to say te

// This is a Number data type; it can be an integer (whole number) or
var numberIntegerDataType = 10;
var numberFloatDataType = 10.4;

// This is a Boolean data type, which can only be given a value of tru
var booleanDataType = true;
```

The text above each variable declaration is how JavaScript handles comments. There are two ways to create a comment in JavaScript:

- Use two slashes `//`. Then anything written afterwards on the same line will be a comment. This is the more common way to create comments.
- For longer comments, we can wrap all of our comment's text to look like this:

```
/* This is a comment */

/*
It
can
work
like
this.
*/
```

Ideally, try to keep a comment succinct and use the `//` syntax to keep your code neat and tidy.

There are eight data types in JavaScript—some you will use more often than others! You'll learn more about the other types as you need to use them, but for more information, see the **MDN web docs on data types (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures)**.

Let's turn our attention back to the variable that's holding our robot's name: `playerName`. We set up this variable differently than we did in the other examples. Rather than providing a set value to the variable, we're assigning it whatever value the `window.prompt()` function returns to us on completion. The data type of the value returned from the prompt is still a string, however. The variable isn't assigned the prompt function but rather is assigned the value that returns from the function.

Use Variables

We've learned how to create data, but now let's use it in our application. The first thing we might want to check is that the data is there correctly.

For now, let's comment the `fight()` function's call at the bottom of the `game.js` file by putting two slashes in front of it so it looks like this:

```
// fight();
```

By turning this function into a comment, we just told the browser to ignore it. This way we can keep code that we don't want the browser to run yet. Before we revisit the `fight()` function, let's add a line of code underneath the variable declaration for `playerName`, so that it looks like this:

```
var playerName = window.prompt("What is your robot's name?");
// Note the lack of quotation marks around playerName
window.alert(playerName);
```

Save the `game.js` file and refresh the HTML page in the browser. After you enter your robot's name and press Enter, you should see an alert appear with your robot's name in it. Nice job!

For now, we can use an alert to ensure that we've entered data properly, but that doesn't work so well with multiple variables. No one wants all those dialog windows popping up! Think of the alert as a way to get information to the user but not the developer.

The Console: A Dev's Best Friend

So if an alert isn't ideal to test and debug code, what is? Let's go into our `game.js` file and replace `window.alert(playerName);` with something (new to us!) called `console.log()`, so that it looks like this:

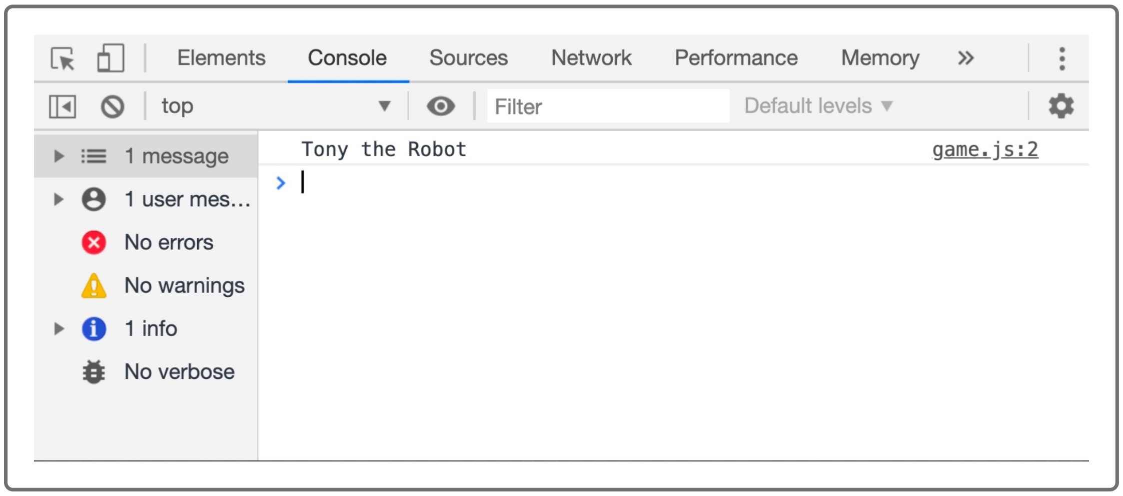
```
var playerName = window.prompt("What is your robot's name?");
// What is this?
console.log(playerName);
```

Save the file, refresh the browser, and answer the prompt with a name for the robot.

What gives? It seems like adding `console.log` didn't do anything at all. Actually, `console.log` does do something very important—only, users won't see it. It's for developers' eyes only!

Let's open Chrome DevTools by right-clicking in the blank HTML document and selecting the word `Inspect`. When it opens, we'll see the Elements tab selected with the HTML and CSS debugging tools. But we need to look at JavaScript now, so click the Console tab at the top of the inspector window.

You should see something like this:



```
Tony the Robot                                        game.js:2
>
```

There's our robot's name! The tool we just used (`console.log`) is called the **developer console**. This is a crucial tool that allows us to test our code without interrupting the page like an alert does.

To the right of our robot's name, the console also tells us which JavaScript file ran the code and which line it occurred on. This can help us pinpoint errors and debug our code.

The `console` isn't part of the JavaScript language. It's another object like the `window` provided to us by the browser, which makes it a **web API**. The `console` object has its own set of functions and properties that interface with the browser's console window (in Chrome, it's the DevTools Console tab we just used).

DEEP DIVE

DEEP DIVE

Think about how we use tools in real life. We can't use our hands to push a nail into a piece of wood or tighten a screw, so we need something specifically made to solve that problem. Hammers and screwdrivers don't work on their own; they work because we use our hands to power them. Similarly, **web APIs** extend our programming capabilities when the tools we have can't solve the current problem.

We have many web APIs at our disposal. The most popular ones enable us to manipulate HTML with JavaScript, play audio or video on a webpage, and even securely process credit card payments.

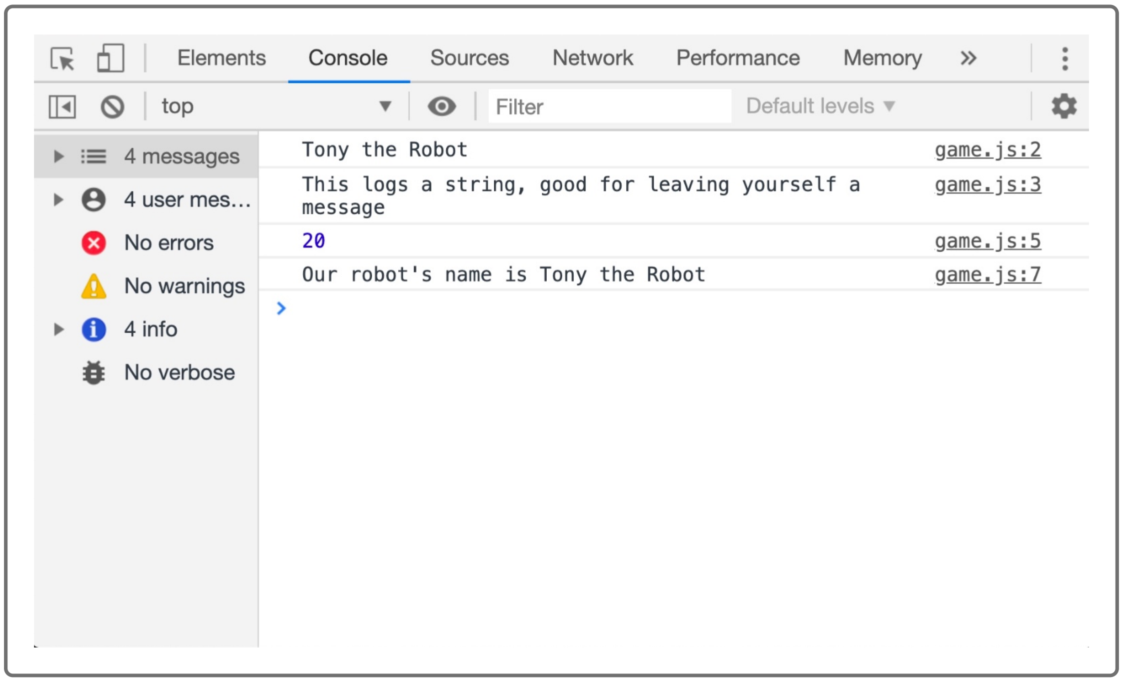
To learn more, see the **MDN web docs on web APIs (https://developer.mozilla.org/en-US/docs/Web/API)** . As you grow as a developer, you'll gain a better understanding of how to use them.

Developers of all experience levels rely on the `console` when writing JavaScript. We'll use it more throughout our Robot Gladiators game, but first let's practice with a few more examples.

Place the following code into `game.js` under the first `console.log()` you added earlier:

```
console.log("This logs a string, good for leaving yourself a message")
// this will do math and log 20
console.log(10 + 10);
// what is this?
console.log("Our robot's name is " + playerName);
```

When you save this file, refresh the browser window, and look at the DevTools Console tab, you'll see something like this:



```
Tony the Robot                                              game.js:2
This logs a string, good for leaving yourself a             game.js:3
message
20                                                          game.js:5
Our robot's name is Tony the Robot                          game.js:7
```

We've used the `console.log()` function in a couple of ways here. We logged a plain string sentence, which can help us trace when certain functionality executes if we run a lot of code at once. We also logged the sum of two numbers (more on arithmetic operators soon). But what's going on in this last log we ran?

Before we learn the official name of the action we performed, let's put it into our own words. We wanted to take the name of our robot and place it into a sentence that would read, "Our robot's name is ." To do this, we needed to combine a string with a variable. This is called **string concatenation**.

In string concatenation, we can write out our string normally, but in order to include variable data, we need to close the string. To do that, put a plus sign `+` after the closing quotation, then write the variable name.

Here are more examples:

```
var playerName = "Tony the Robot";

// Tony the Robot is ready for battle!
console.log("Tony the Robot" + " is ready for battle!");

```

```
// "Tony the Robot is ready for battle!"
console.log(playerName + " is ready for battle!");

// "Your robot, Tony the Robot, has won!
console.log("Your robot, " + playerName + ", has won!");
```

The first example concatenates two strings, which is usually unnecessary but a helpful comparison to the second example. The second example is like the one we used before, except that we flipped where the variable was concatenated. The last one, however, needed the variable in the middle of the sentence. So we closed the first string, used the plus sign to include the `playerName` variable, used another plus sign, and continued with our string.

Also, when we concatenate strings we need to include a leading or trailing space in our string so that the concatenated variable doesn't run up against the word before or after it.

Let's touch on one more thing before continuing with our Robot Gladiators game. Open the Console tab in Chrome DevTools. In the main window, there's a blinking line that indicates that we can type in it.

We can use the console to write and test simple JavaScript with different features or functionality rather than writing it into our files, then saving and refreshing. We can just write a line of JavaScript and press Enter, and it'll run the code.

Let's add the following to the console:

```
var name = "your name";
console.log(name);
```

IMPORTANT

> When we create a variable or function in the DevTools Console and press Enter, it will often return a value of `undefined`. Don't worry—this

> isn't an error. The console evaluates every piece of code and attempts to return a result. But there is nothing to return for a variable, so it returns the value `undefined` by default.
>
> To learn more, see the **MDN web docs on undefined data type (https://developer.mozilla.org/en-US/docs/Glossary/Undefined)** .

The above code should create a variable called `name`, with your name stored as the value. Then the next line will `console.log()` the value of the `name` variable.

DEEP DIVE

DEEP DIVE

Chrome DevTools can help developers discover bugs and issues in front-end development, but sometimes the options are overwhelming and hard to understand at first. Take a look at **Google's documentation about DevTools (https://developers.google.com/web/tools/chrome-devtools)** for a detailed explanation.

We've now learned how to create functions, execute functions, interact with the user, and store data. Let's revisit our `fight()` function and start giving our robot something to do!

HIDE HINT

Make sure you commit the code you've completed so far using Git!