JS Functions

Course: Java

S1

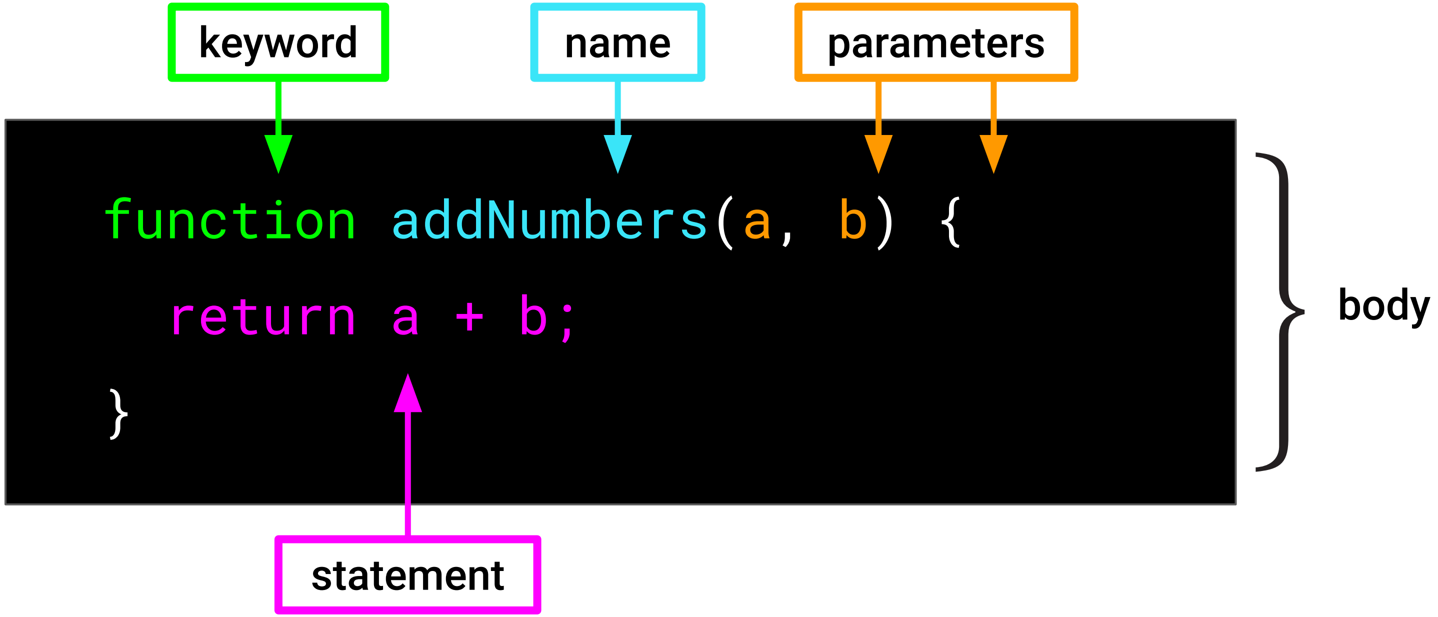
JS Functions

Modern applications can quickly grow in size and complexity and usually involve entire teams of developers.

One thing that helps immensely—both with complex software and other complex tasks—is to break a large problem or application down into smaller pieces, or pieces of functionality. Enter functions!

JS Functions

Functions allow us to wrap up blocks of code that take in some input (or not) and then return (or not) some output. We can give these functions names, much like we do with other data.

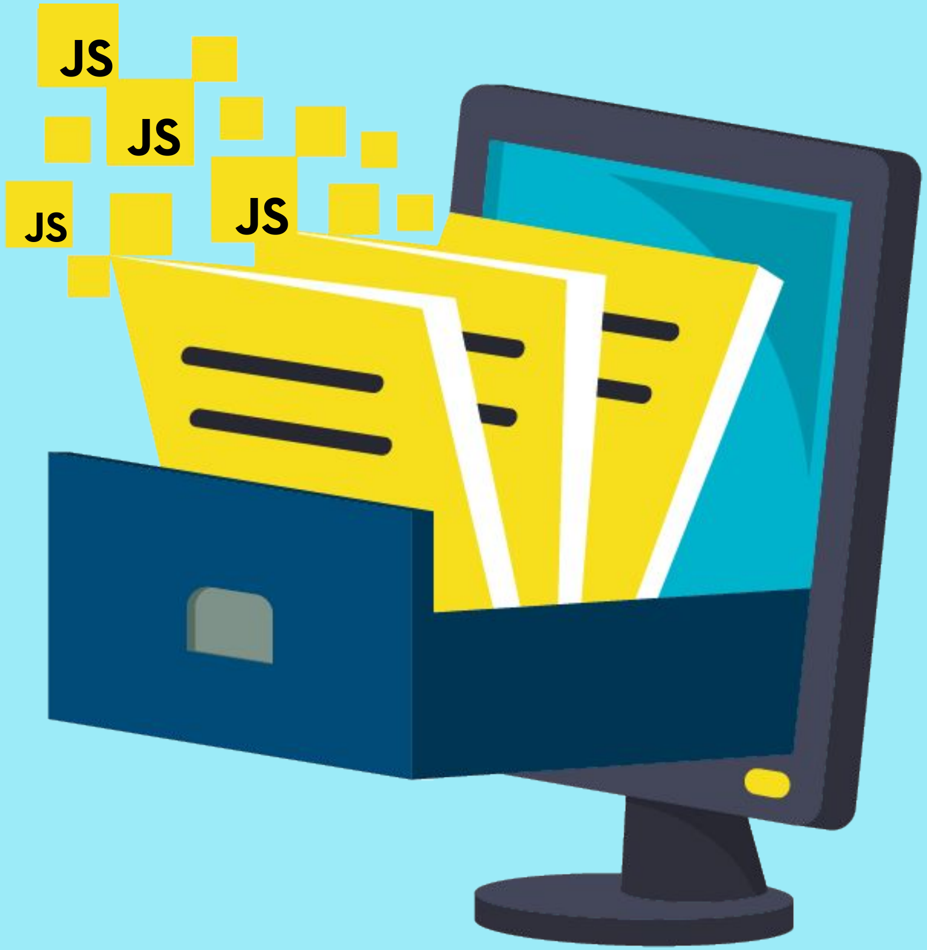


```
function addNumbers(a, b) {
  return a + b;
}
```

JS Functions

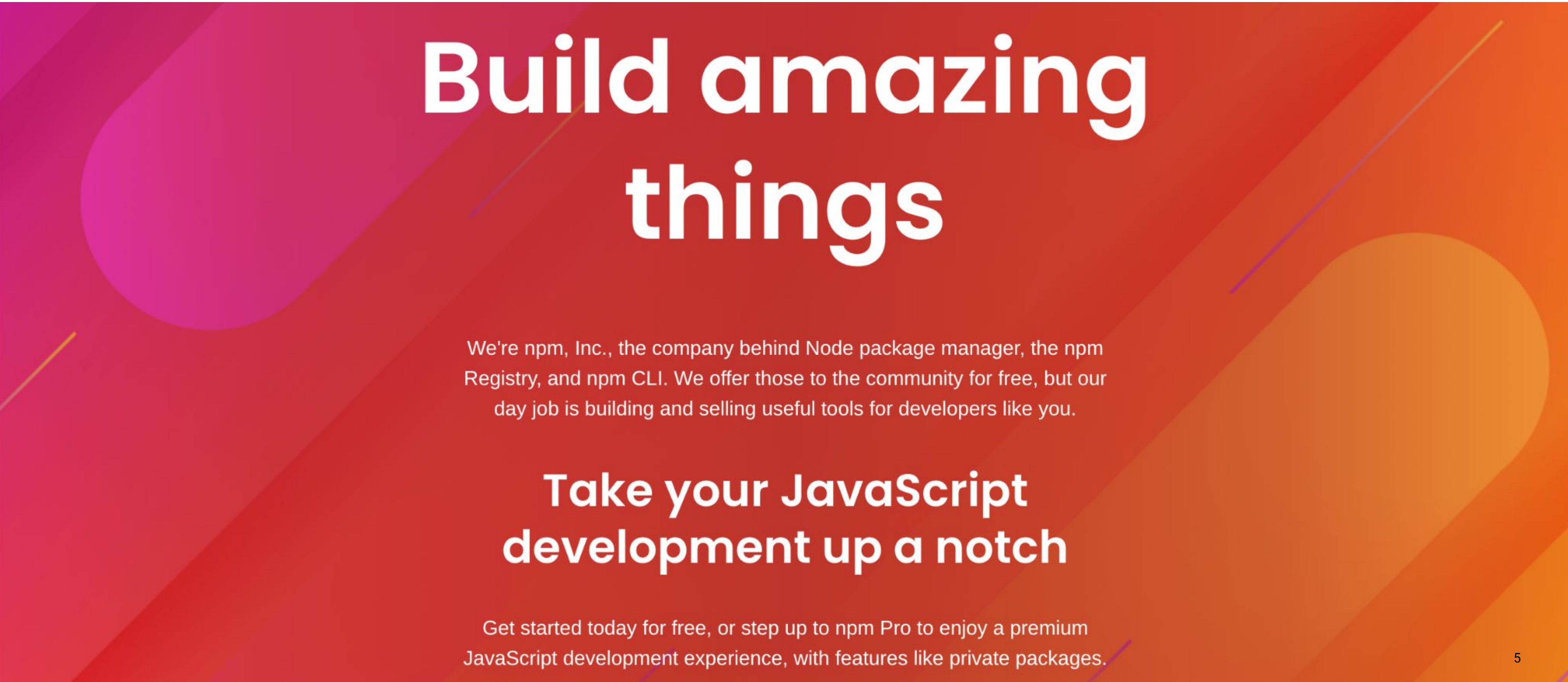
We can also build our own function libraries to decrease code clutter and promote better project architecture.

Whenever we need to use those functions, we can `import`.



JS Functions

We could even publish these for public consumption on npmjs.com.



Build amazing things

We're npm, Inc., the company behind Node package manager, the npm Registry, and npm CLI. We offer those to the community for free, but our day job is building and selling useful tools for developers like you.

Take your JavaScript development up a notch

Get started today for free, or step up to npm Pro to enjoy a premium JavaScript development experience, with features like private packages.

Functions are first-class citizens in the world of JS. They are treated like any other data type.

Learning Outcomes

By the end of this lesson, you will be able to:

01 Use `function`, parameters (default parameters), arguments, and `return`.

02 Explain function naming and pure functions.

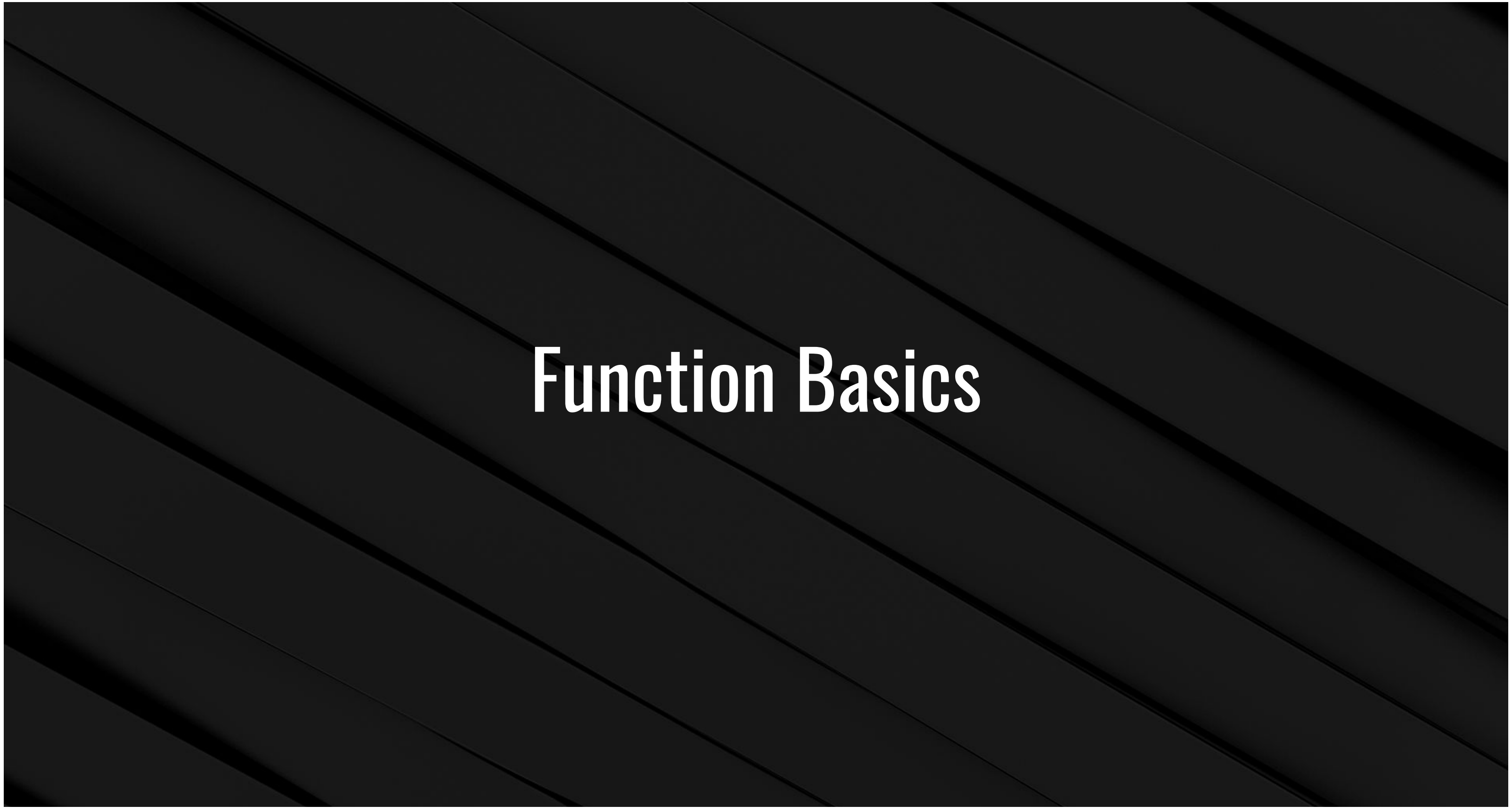
03 Write simple functions.

04 Use methods.

05 Use callback functions for asynchronous event-driven programming.

06 Use the rest parameter and variadic functions.

07 Use object destructuring and parameters.



Function Basics

Function Basics

Functions are relatively simple to write.

Since JS is a loosely typed language, we don't need to specify all of the input and output types up front.

Functions can be created in the global scope...

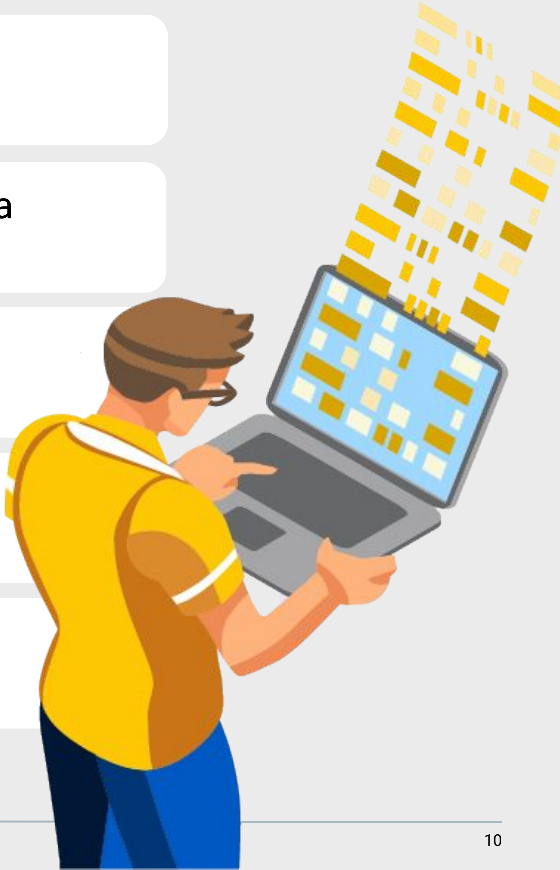
or they can be scoped inside of object literals. In the latter case, we call them **methods**. But the terms are often used interchangeably.

Functions can even be created inside of other functions.

However, outside of something like functional components in React, for example, this is less common.

Best Practices for Writing Functions

- Start the name with a verb (except in certain cases).
- Avoid side effects such as `console.log()`, or any cases where a function has to reach outside of its scope to perform a task.
- Avoid variable shadowing. Generally, this means not reusing the same variable names in the same file.
- Generally, a function should do one specific task and do it well.
- Include a `return` whenever possible.



Best Practices: Notable Exceptions

To check work and understand the code, we will keep `console.log()s` in. Remove it when finished testing.

When interacting with elements in the browser, we will want functions to reach outside of their scope and change elements that are on our webpage.

In React, and/or when implementing OOP, we will name the function based on a specific thing (e.g., a component) that it might return.

Time to Code

Function Basics

Suggested Time:

10 Minutes

Activity: Write Basic Functions

Suggested Time:

15 Minutes

Time’s Up! Let’s Review.

Questions?

Break

Early Returns

As soon as a function hits a `return`, the function invocation ends. We can use this approach to avoid some `else`s.

```
const age = 13;

// Notice that we deliberately avoid VARIABLE SHADOWING in the PARAMETER
function checkIsMinor(someAge) {
  if (age < 18) {
    return true
  }

  // No need for `else`!
  return false
}
```

Implicit Return is undefined

If we don't specify our own `return` in a function, it will return `undefined` implicitly. This is **usually** not what we want. Most of the time (but not always), functions should receive some input and **explicitly return** some output that we specify.

```
// DON'T do like this one!

function greet(greeting) {
  // Using a side effect - reaching outside of scope to `log` - NO!
  console.log(greeting)

  // No explicit return - will return `undefined` - NO! (usually)
}
```

You can use `console.log()` in your functions to test things out and view the arguments, etc. Just remove them when you're done.

Time to Code

Destructured Parameters

Suggested Time:

10 Minutes

Activity: Fix the Bug in Destructured Parameters

Suggested Time:

5 Minutes

Time's Up! Let's Review.

Questions?

Default Parameters

Whenever possible, use default parameters to avoid `undefined`s:

```
// If we don't pass in an argument to bind to num1, it will be `333`, etc.
function add2Nums(num1 = 333, num2 = 444) {
  return num1 + num2;
}

// Just use both default parameters above
add2Nums();

// Use only the default for `num2` above
add2Nums(5)
```

Default Parameters

We can even use a default parameter based on another parameter:

```
// Order matters - NOT: `(num1 = num2, num2 = 3)`
function add2Nums(num1 = 3, num2 = num1) {
  return num1 + num2;
}

/**
 * `num1` will be `3`
 * `num2` will follow suite.
 */
add2Nums() // 6
```

Activity: Check age for isMinor

Suggested Time:

20 Minutes

Time's Up! Let's Review.

Questions?

Break

Structure, Arrays, and Destructuring

Code Structure

Let’s review some **good coding habits** that help avoid pitfalls.

Code Structure

There are some nuances regarding when `const`, `let`, `function`, and even `var` are read into memory (hoisting and scoping).

A nice convention for ordering things in JS:

- Put `const` and `let` on top. Perhaps even alphabetize them by variable name. These are our **global variables**.
- Then, `function` declarations go below that, perhaps also alphabetized.
- Business logic goes below that.

Later on, when we modularize our code with EcmaScript modules, we will put `import` at the top and `export` at the bottom.

Avoid variable shadowing. It's easy to do when naming function parameters.

Activity: Sum an Array of Numbers

Suggested Time:

10 Minutes

Time’s Up! Let’s Review.

Questions?

Variadic Parameters

Variadic Parameters

`console.log()` can take **any number of arguments**. We call this **variadic parameters**; the number of parameters can be 0 or infinite (within the bounds of memory).

To accomplish this, we use `(...)`. Even though this looks like the spread operator that we saw in the JS Fundamentals lesson, when we use it in the context of a parameter, it is a **rest parameter**.

The result of `(...)` (rest parameter) is to wrap up any/all arguments into an array. So, we don't pass in our own array. `(...)` does that for us.

A rest parameter takes in the rest of all of the arguments and wraps them in an array.

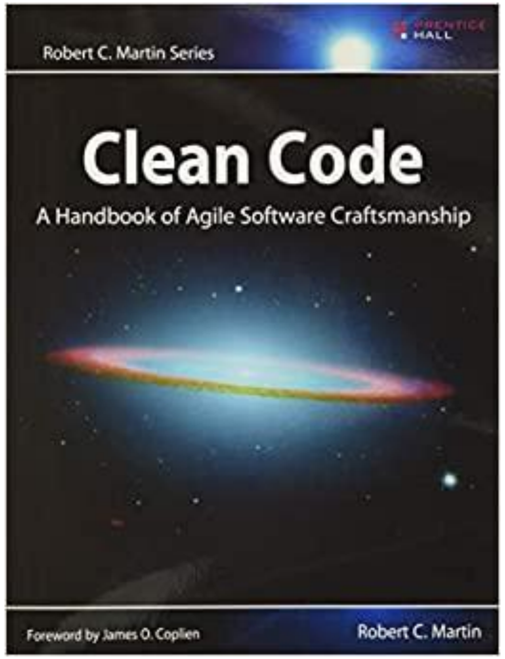
Variadic Parameters

```
// We can pass in as many or as few numbers as needed - NOT AS AN ARRAY this
time.
function sumNums(...nums) {
  // An array
  console.log(nums)

  // TODO: Implement same logic as before to sum up the numbers in an Array
}

// Pass in as many arguments as desired - not an array
sumNums(1, 3, 4, 5, 6, 7, 8, 9, 1111);
```

When to Destructure

When to Destructure

As we learned earlier, we can always destructure. In some cases, we should—for example, when our function specifies more than two parameters.


Robert C. Martin Series
PRENTICE HALL
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

This best practice is originally from a famous book on programming, *Clean Code*.

Clean Code: A Handbook of Agile Software Craftsmanship
by Robert C. Martin
(Pearson, 2008)

When to Destructure

```
function processOrder(username, cartTotal, numOfItems, deliveryZipCode,
billingZipCode, paymentMethod) {

  // TODO: Do something with this info.
}
```

When we invoke this function, it is easy to mess up the order of the arguments.

Here, we should **require** destructuring. That way, the order of the arguments doesn't matter as long as the property names match up.

When to Destructure

If we have more than two parameters, we should use a destructured object.

```
function processOrder({username, cartTotal, numOfItems, deliveryZipCode,
billingZipCode, paymentMethod}) {

  // TODO: Do something with this info.
}

// Pass the arguments inside of an object - order is irrelevant.
processOrder({billingZipCode: 62235, cartTotal: 100.23, deliveryZipCode: 62235,
numOfItems: 21, paymentMethod: "COD", username: "Narcos" })
```

Destructuring with Arrays

Destructuring with Arrays

Although it’s less common, we can also apply destructuring when we receive an array. However, it's rare that we want to just destructure the first few elements and then do nothing with the rest of them.

```
function getFirst2Things([thing1, thing2]) {
  console.log(thing1, thing2)
  // TODO: Do something
}

// Only "Theodore" and "Alvin" will be used by the function
getFirst2Things(["Theodore", "Alvin", "Simon", "James", "Punky" ])
```

Activity: Use Array Destructuring to Get the First Three Characters of a String

Suggested Time:

10 Minutes

Strings are “array-like” and can be destructured.

Array Destructuring

We will fully explore the dual nature of the string primitive (as an object and as a primitive) in the next lesson.

```
// TODO: Use Array destructuring to get the first 3 characters from the argument
function stripFirst3Chars() {
  // TODO: Return those 3 characters inside of template literal
  return;

}

stripFirst3Chars("Hello");
```

Time’s Up! Let’s Review.

Questions?

Destructuring with Default Parameters

Break

Function Activities

Time to Code

Build Basic Form Validation Functions

Suggested Time:

30 Minutes

Activity: Generate a Bio

Suggested Time:

30 Minutes

Time’s Up! Let’s Review.

Questions?

RECAP

Recap

You have learned several techniques and best practices for working with functions:

01 Function declarations and definitions

02 Parameters vs. arguments

03 Variadic parameters with rest

04 Default parameters and destructuring

Break

Methods and Asynchronicity

Functions Inside Object (Methods)

Methods: Functions Inside Objects

Functions can be written in the global scope, as we have been doing.

Functions can also be scoped inside of objects.

These are called **methods**.

For example, `log` is a method scoped inside of the object `console`.

Time to Code

Refactor Bio Generator to Use Methods

Suggested Time:

15 Minutes

Functions Are First Class

Functions Are First Class

In JS, unlike in Java, functions are considered **first class**.

- This means that we can do anything with functions that we can do with other data types.

Since we can use `const` and `let` with other data, it means that we can create **function expressions** with `const` and `let`, too.

- `const sayHello = function() { return "Hello!" }`

In the previous example, we could have included a label after the `function` keyword: `const sayHello = function sayHello()`. But, that's optional.

- Sometimes it's helpful to keep this label on for debugging logs.

Since we can pass other data types into functions, we can pass functions into other functions. We call these **callback** functions.

Asynchronicity and JS

We will discuss asynchronicity in more depth later.

For now, the simplest mechanism for displaying asynchronicity in JS is using `setTimeout`.

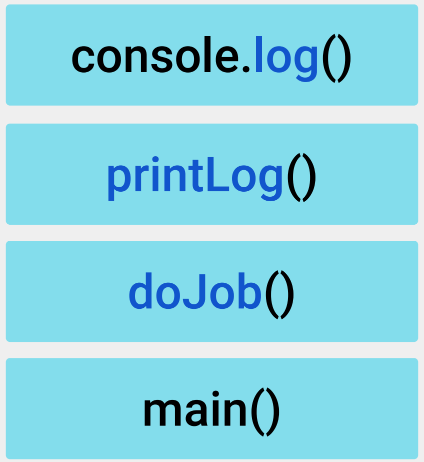
Asynchronicity and JS

Whenever we invoke a function, it is put on **JS's call stack**. Functions that call other functions are stacked on top of each other and popped off once they are complete.

Code

```
function printLog() {
  console.log('blah-blah');
}

function doJob() {
  printLog();
}

doJob();
```

Call Stack



- JS uses Last-In-First-Out (LIFO) to do this, similar to Java.
- Unlike Java, JS is **synchronous** and **single-threaded**.
- This means that JS can only do one thing at a time and that it executes code in order from top to bottom.
- JS cannot do anything else until its call stack is empty.

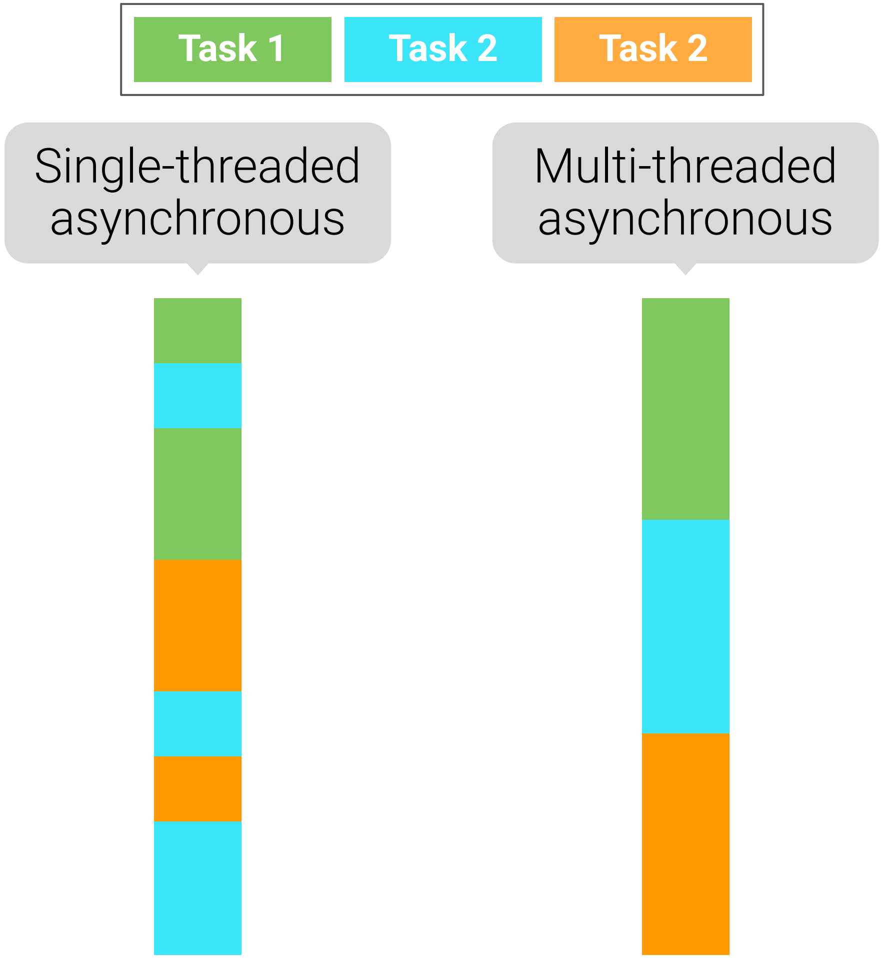
If a function took too long to run, users would essentially be locked out from doing anything else. This is known as **blocking**.

Asynchronicity and JS

The browser excels at keeping track of all types of asynchronous events: for example, a `“click”`, a `“submit”`, or the completion of a timer.

If JS needed to listen for a click, then its call stack would be occupied by this process and we couldn’t do anything else with it. On the other hand, in Java, we can spawn multiple threads.

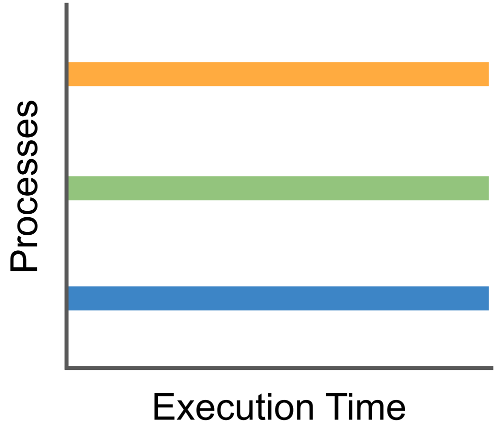
JS is single-threaded.



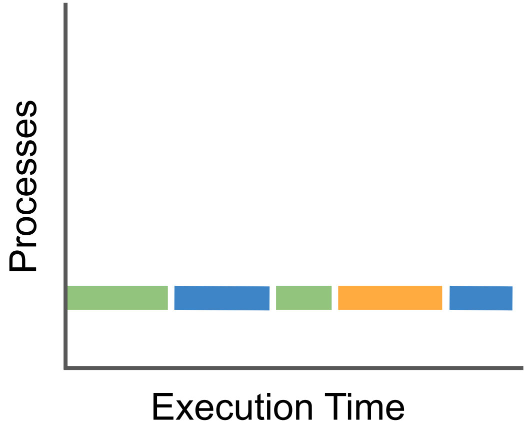
Asynchronicity and JS

Concurrency is merely a broader term used for defining multiple tasks that have the ability to run in parallel. Asynchrony is a more specific type of concurrency in which tasks are able to run in parallel by allowing a task to “pause” and allow other tasks to run while it awaits for its result.

Parallelism



Asynchrony



KEY 🟩 Task 1 🟦 Task 2 🟧 Task 3

Synchronous behavior is like...

...putting something in a microwave and then staying perfectly stationary, unable to perform any other tasks, while waiting for the microwave process to finish.

Asynchronicity and JS

The events that we just mentioned (and many more that can occur in a browser) are all asynchronous.

Unlike the code that we have written until now, we have no way of controlling when, if, or how they will occur.

Fortunately, the browser provides an application programming interface (API) for JS—the **Web API**.

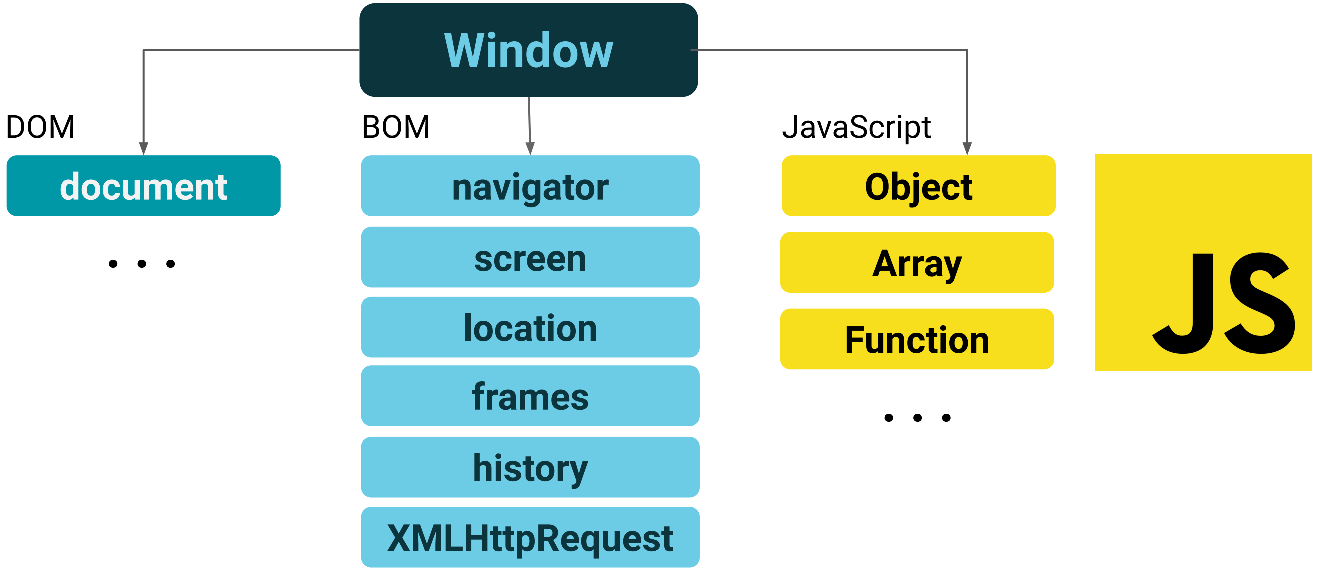
In this way, JS can behave asynchronously and respond to events.

Asynchronicity and JS

This Web API is accessible via a special global object known as the `window`.

The `console` that we have been using is really: `window.console`.

Fortunately, we don't have to specify `window`. JS knows to check here.



One of the primary mechanisms by which we leverage asynchronous behavior via the Web API is by using callback functions.

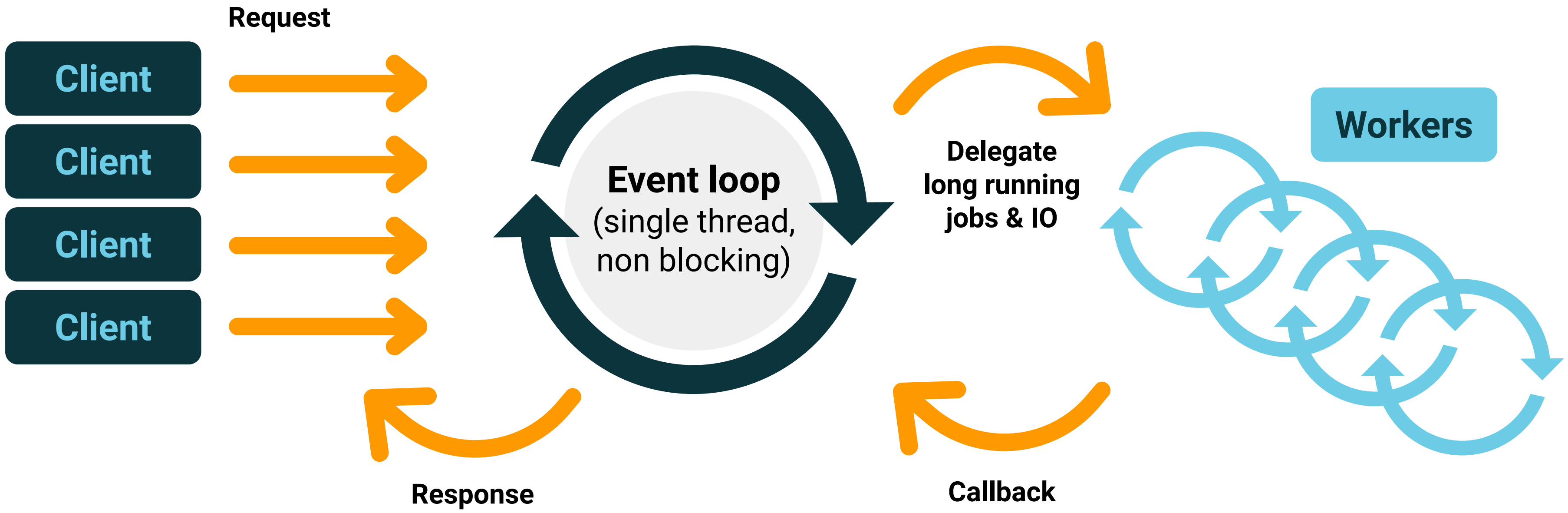
Callback Functions

Callback functions are event-driven functions. They are **called back** in response to an event.

- A function specifies a callback function as one of its parameters.
- When it is invoked, it goes on the call stack but is deferred to the browser.
- Once the event is complete, the browser notifies JS. JS then invokes the callback function, provided its call stack is empty.

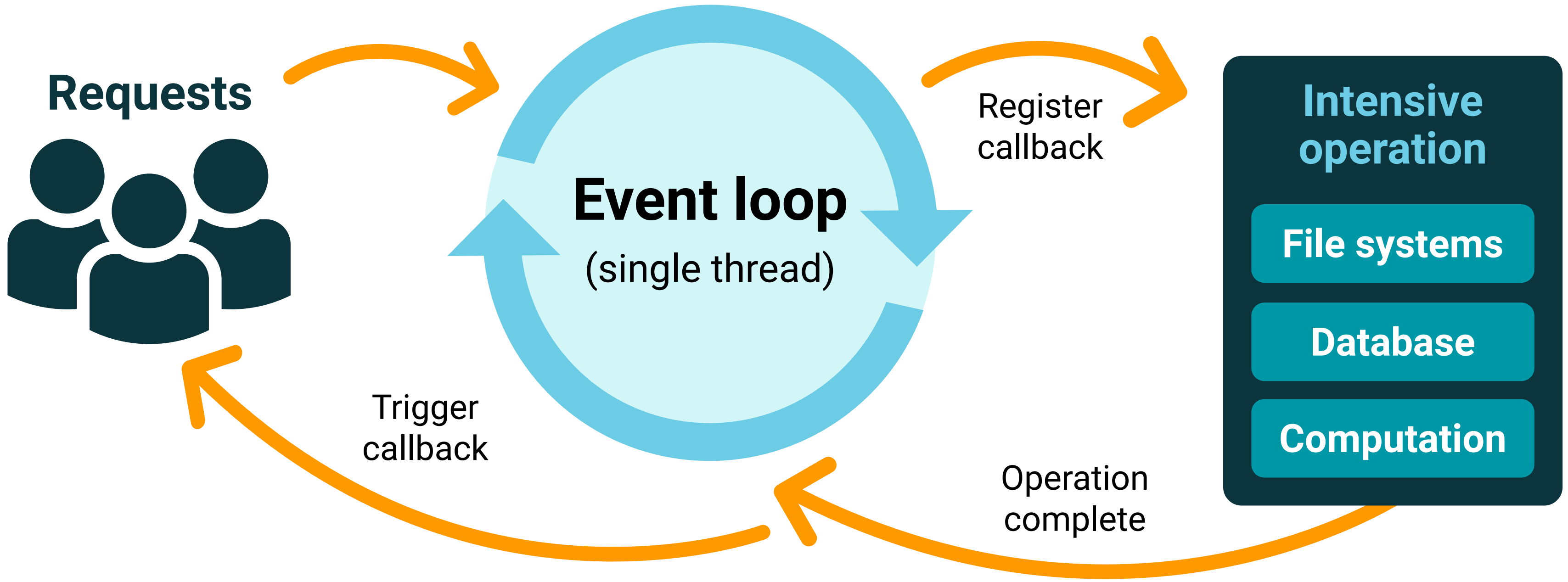
Callback Functions

This callback function goes onto the call stack, and the process continues.
This is known as the JS **event loop**.



Callback Functions

This callback function goes onto the call stack, and the process continues.
This is known as the JS **event loop**.



Callback Functions

Asynchronous, event-driven behavior is like putting something in a microwave, proceeding with our lives, and then performing another function when the microwave signals that its process is complete.

Of course, we can only respond to this event if we are not in the middle of another.

But, as soon as our call stack is empty, we can run a callback function to deal with the microwave event.

Callback Functions: setTimeout

Callback Functions: setTimeout

Since JS is synchronous, if we tasked it with setting a 2-second timer, it would block everything else while this timer sat on the call stack for 2 seconds (or 200)!

For this reason, JS knows nothing about timers, but it does know about functions—while the browser knows how to set a timer.

```
/**
 * Use `window.setTimeout` to leverage the browser's ability to run a timer asynchronously
(`window.` is optional syntax).
 * setTimeout needs to know just two things:
 * 1. What's the callback function? What is it that we want to do in the future?
 * 2. How long should we wait, in milliseconds?
 */
setTimeout(function() {
  console.log("Time's up!")
}, 2000)
```

Break

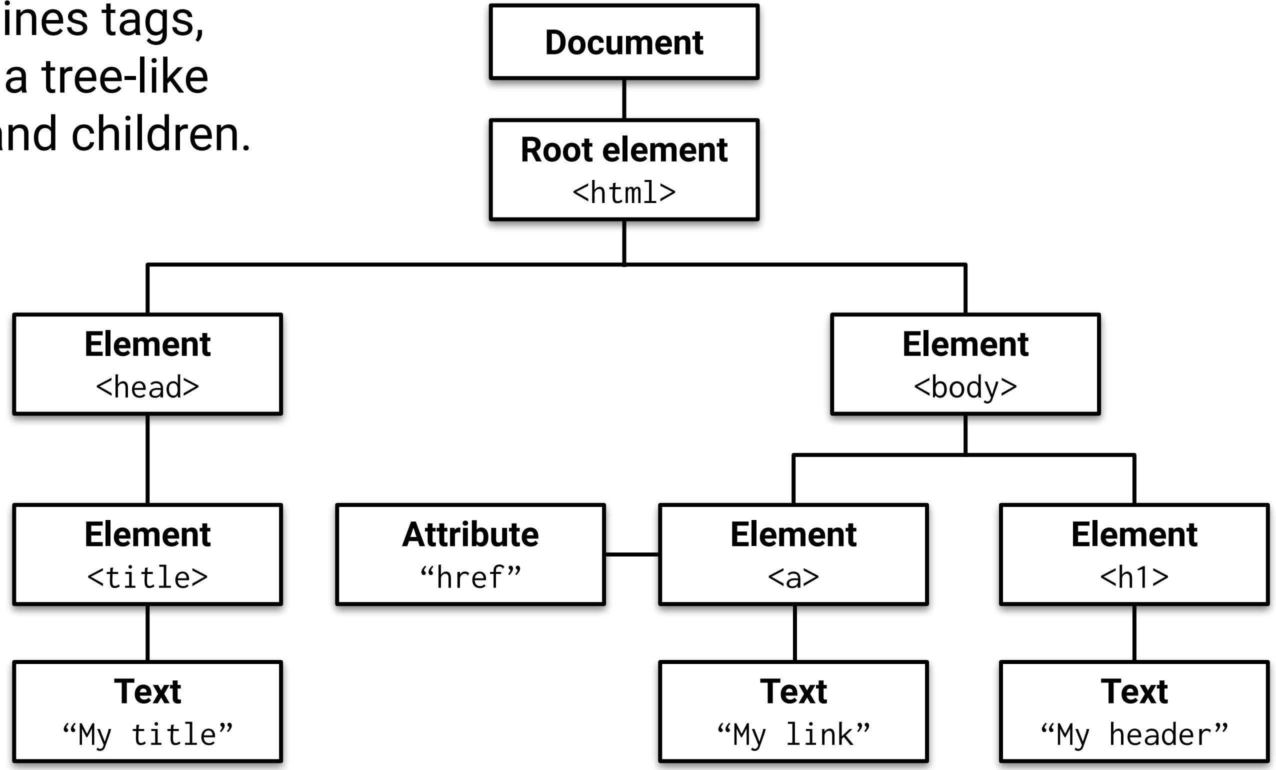
Document Object Model (DOM)

One of the most commonly used parts of the Web API is the DOM.

Document Object Model (DOM)

Recall that HTML combines tags, attributes, and text into a tree-like structure with parents and children.

Each of the “leaves” on this tree, whether a tag, attribute, or text, is known as a **node**.



This has nothing to do with NodeJS, which is JS that runs in a wholly separate environment in conjunction with an operating system (OS) (as opposed to the browser).

Document Object Model (DOM)

JS **does not know** anything about HTML.

JS **does know** a lot about objects.

- The browser provides a DOM API (a subset of the Web API).
- This takes an entire HTML **document** and **models** it as an **object**.
- The DOM API provides many properties and methods. All of them are accessible via `window.document` (`window` is not necessary).

Time to Code

Select a Button and Register a “click” Event

Suggested Time:

30 Minutes

Activity: Events Playground

Suggested Time:

20 Minutes

Time’s Up! Let’s Review.

Questions?

Break

Time to Code

Getting Information From Events

Suggested Time:

10 Minutes

Time to Code

value.length

Suggested Time:

10 Minutes

Time to Code

Prevent the Browser’s Default Behavior:
.preventDefault()

Suggested Time:

10 Minutes

Activity: Disable <input> if length Is Too Long

Suggested Time:

20 Minutes

Time’s Up! Let’s Review.

Questions?

RECAP

Learning Outcomes

By the end of this lesson, you will be able to:

01 Use `function`, parameters (default parameters), arguments, and `return`.

02 Explain function naming and pure functions.

03 Write simple functions.

04 Use methods.

05 Use callback functions for asynchronous event-driven programming.

06 Use the rest parameter and variadic functions.

07 Use object destructuring and parameters.

The End