JavaScript `this` Keyword Masterclass

The Fundamentals

In JavaScript, `this` is a special keyword that refers to the context in which a function is executed. Unlike some other languages where `this` might always refer to the instance of a class, JavaScript's `this` is dynamically scoped - its value depends on how a function is called, not where it's defined.

The Five Rules of `this`

1. Global Context

When used outside any function or object, `this` refers to the global object:

- In browsers: `window`
- In Node.js: `global`

```javascript
console.log(this === window); // true (in a browser)

var globalVar = "I'm global";
console.log(this.globalVar); // "I'm global"
```

2. Function/Method Context

When a function is called as a method of an object, `this` refers to the object the method belongs to:

```javascript
const person = {
  name: "Alex",
  greet: function() {
    return `Hello, my name is ${this.name}`;
  }
};

console.log(person.greet()); // "Hello, my name is Alex"
```

3. Constructor Context

When a function is used as a constructor with the `new` keyword, `this` refers to the newly created instance:

```javascript
function Person(name) {
  this.name = name;
  this.greet = function() {
    return `Hello, my name is ${this.name}`;
  };
}

const alex = new Person("Alex");
console.log(alex.greet()); // "Hello, my name is Alex"
```

4. Explicit Binding

You can explicitly set the value of `this` using methods like:

- `call(thisArg, arg1, arg2, ...)`
- `apply(thisArg, [argsArray])`
- `bind(thisArg, arg1, arg2, ...)`

```javascript
function greet() {
  return `Hello, my name is ${this.name}`;
}

const person = { name: "Alex" };

console.log(greet.call(person)); // "Hello, my name is Alex"
console.log(greet.apply(person)); // "Hello, my name is Alex"

const boundGreet = greet.bind(person);
console.log(boundGreet()); // "Hello, my name is Alex"
```

5. Arrow Functions

Arrow functions don't have their own `this` context. Instead, they inherit `this` from the enclosing lexical context:

```javascript
const person = {
  name: "Alex",
  // Traditional function
  greet: function() {
    // Helper function inside the method
    const innerFunc = () => {
      // This arrow function captures `this` from greet
      return `Hello, my name is ${this.name}`;
    };
    return innerFunc();
  }
};

console.log(person.greet()); // "Hello, my name is Alex"
```

Common Pitfalls

Callback Functions

When passing methods as callbacks, they lose their original `this` context:

```javascript
const person = {
  name: "Alex",
  greet: function() {
    return `Hello, my name is ${this.name}`;
  }
};

// Problem: `this` is no longer the person object
setTimeout(person.greet, 100); // "Hello, my name is undefined"

// Solutions:
// 1. Bind the method
setTimeout(person.greet.bind(person), 100);

// 2. Use an arrow function wrapper
setTimeout(() => person.greet(), 100);
```

Event Handlers

In DOM event handlers, `this` refers to the element that triggered the event:

javascript

```javascript
document.getElementById("myButton").addEventListener("click", function() {
  // `this` refers to the button element
  this.style.backgroundColor = "red";
});
```

Advanced Concepts

`this` in Classes

ES6 classes provide a more structured way to work with `this`:

javascript

```javascript
class Person {
  constructor(name) {
    this.name = name;
  }

  greet() {
    return `Hello, my name is ${this.name}`;
  }

  // Arrow method maintains lexical `this`
  delayedGreet = () => {
    setTimeout(() => {
      console.log(`Delayed greeting: Hello, my name is ${this.name}`);
    }, 1000);
  };
}
```

Using `this` with `.bind()`, `.call()`, and `.apply()`

These methods allow you to control the `this` value explicitly:

```javascript
function introduce(greeting, punctuation) {
  return `${greeting}, my name is ${this.name}${punctuation}`;
}

const person = { name: "Alex" };

// Different ways to call with specific context
console.log(introduce.call(person, "Hi", "!")); // "Hi, my name is Alex!"
console.log(introduce.apply(person, ["Hello", "."]))  // "Hello, my name is Alex."
const boundIntroduce = introduce.bind(person, "Hey");
console.log(boundIntroduce("?")); // "Hey, my name is Alex?"
```

Best Practices

1. Use arrow functions when you want to preserve the outer `this` context.
2. Use method binding or wrappers when passing callbacks.
3. Consider using modern class syntax for cleaner code.
4. Avoid relying on `this` in complex nested functions.

The Purpose of the `this` Keyword in JavaScript

The `this` keyword exists in JavaScript for one fundamental purpose: **to provide a way for functions to dynamically access and interact with their execution context**.

Core Purposes

1. Object-Oriented Programming

The primary purpose of `this` is to enable object-oriented programming patterns in JavaScript. It allows methods to:

- Access and modify properties of the object they belong to
- Interact with other methods of the same object
- Maintain state within an object

javascript

```javascript
const counter = {
  count: 0,
  increment() {
    this.count++;
    return this.count;
  },
  reset() {
    this.count = 0;
    return this;  // For method chaining
  }
};
```

Without `this`, methods would need explicit references to their parent objects, making code rigid and hard to reuse.

2. Code Reusability

`this` enables the creation of reusable function templates that can operate on different objects:

javascript

```javascript
function greet(greeting) {
  return `${greeting}, my name is ${this.name}`;
}

// The same function works with any object that has a name property
const person1 = { name: "Alex" };
const person2 = { name: "Taylor" };

console.log(greet.call(person1, "Hello"));  // "Hello, my name is Alex"
console.log(greet.call(person2, "Hi"));     // "Hi, my name is Taylor"
```

3. Constructor Functions and Class Patterns

`this` is essential for JavaScript's constructor pattern, allowing the creation of multiple object instances with shared behavior but individual state:

```javascript
function Person(name) {
  this.name = name;  // Each instance gets its own name property
}

Person.prototype.greet = function() {
  return `Hello, I'm ${this.name}`;  // Shared method accessing instance-specific data
};

const alex = new Person("Alex");
const taylor = new Person("Taylor");
```

4. Event Handling

In browsers, `this` provides a convenient reference to DOM elements that triggered events:

```javascript
document.querySelector("button").addEventListener("click", function() {
  // `this` refers to the button element that was clicked
  this.classList.toggle("active");
});
```

Philosophical Purpose

At a deeper level, `this` embodies one of JavaScript's core design principles: **functions as first-class citizens**. By making `this` dynamic and contextual rather than fixed, JavaScript creates a flexible system where:

1. The same function can serve different purposes in different contexts
2. Functions can be borrowed, shared, and reused across objects
3. Behavior can be composed and combined in powerful ways

In essence, the purpose of `this` is to make JavaScript more expressive by separating what a function does (its code) from what it operates on (its context), allowing for more dynamic and compositional patterns than would be possible with static scoping alone.

This fundamental design choice is what enables many of JavaScript's most powerful patterns and frameworks, from method chaining to jQuery's design to React's component architecture.

Core Design Principles of JavaScript: The Original Vision

JavaScript was created by Brendan Eich in just 10 days in 1995, yet its core design principles have shaped web development for decades. Here are the foundational principles that defined early JavaScript:

1. Functions as First-Class Citizens

Perhaps the most influential design decision was treating functions as first-class objects that could be:

- Passed as arguments to other functions
- Returned from functions
- Assigned to variables
- Stored in data structures

```javascript
// Functions as values
const sayHello = function() { return "Hello"; };

// Functions as arguments
function repeat(fn, times) {
  for (let i = 0; i < times; i++) fn();
}
```

This principle enabled higher-order functions, callbacks, and eventually the functional programming patterns that are now central to modern JavaScript.

2. Prototype-Based Inheritance

Unlike class-based languages popular at the time (Java, C++), JavaScript adopted prototype-based inheritance:

```javascript
function Animal(name) { this.name = name; }
Animal.prototype.speak = function() { return `${this.name} makes a noise.`; };

function Dog(name) {
  Animal.call(this, name);
}
Dog.prototype = Object.create(Animal.prototype);
```

This simplified inheritance model was inspired by Self and Scheme, offering flexibility without the complexity of formal class hierarchies.

3. Dynamic Typing

JavaScript embraced dynamic typing where variables could change types at runtime:

```javascript
let x = 5;        // Number
x = "hello";      // String
x = true;         // Boolean
x = [1, 2, 3];    // Array
```

This principle prioritized ease of use and rapid development over strict type safety.

4. Automatic Type Coercion

JavaScript would automatically convert values between types when needed:

```javascript
"5" + 3      // "53" (number converted to string)
"5" - 3      // 2 (string converted to number)
if ("hello") // Truthy non-empty string
```

While controversial, this made the language more forgiving for beginners.

5. Lexical Scoping with Function Scope

Variables were scoped to functions rather than blocks:

```javascript
function example() {
  var x = 10;
  if (true) {
    var y = 20;  // Not block-scoped, available throughout function
  }
  console.log(x, y);  // 10, 20
}
```

This simplification (before `let` and `const`) reflected JavaScript's design goal of being accessible to non-programmers.

6. Event-Driven Programming

JavaScript was designed from the beginning to handle asynchronous user interactions:

```javascript
document.getElementById("button").onclick = function() {
  alert("Button clicked!");
};
```

This event-driven model was fundamental to creating dynamic web pages.

7. Loose Object Literal Syntax

Objects could be created on-the-fly without formal class definitions:

```javascript
const person = {
  name: "Alex",
  age: 30,
  greet: function() { return "Hello"; }
};
```

This lightweight syntax enabled rapid prototyping and data representation.

8. Garbage Collection

Memory management was handled automatically:

```javascript
function createObjects() {
  const obj1 = { data: "large data" };
  const obj2 = { data: "more data" };
  // Objects automatically cleaned up when function exits
}
```

This eliminated a major source of bugs in languages like C/C++.

9. Forgiving Design

JavaScript was intentionally designed to be forgiving of errors:

```javascript
// Undefined variables become global (without strict mode)
x = 5;

// Missing semicolons often okay
let y = 10
console.log(y)

// Extra commas in arrays tolerated
let arr = [1, 2, 3, ,];
```

This reflected the language's goal of being accessible to non-programmers creating simple scripts.

10. Browser Integration

JavaScript was designed specifically to interact with the browser environment:

```javascript
// Access to document
document.write("Hello World");

// Built-in dialogs
alert("Warning!");
const name = prompt("Enter your name:");

// Window control
setTimeout(function() { console.log("Later"); }, 1000);
```

These principles reflect JavaScript's original purpose: to be an accessible scripting language for the web that could be picked up by designers and non-programmers. While some of these decisions have been criticized over time, they helped JavaScript become the world's most widely used programming language and the foundation of modern web development.

Functions as First-Class Citizens in JavaScript: A Deep Dive

The concept of functions as first-class citizens is arguably JavaScript's most influential design principle, setting it apart from many languages of its era. Let's explore this concept in depth:

The Core Principle

When we say functions are "first-class citizens" in JavaScript, we mean they have the same rights and privileges as any other value. Functions can go anywhere and do anything that numbers, strings, or objects can do.

Key Characteristics

1. Functions Can Be Assigned to Variables

javascript

```
// Function declaration
function greet(name) {
  return `Hello, ${name}!`;
}

// Function assigned to variable
const sayHello = greet;
console.log(sayHello("Alice")); // "Hello, Alice!"

// Anonymous function assigned directly
const add = function(a, b) {
  return a + b;
};
```

2. Functions Can Be Passed as Arguments

javascript

```
function executeOperation(operation, a, b) {
  return operation(a, b);
}

function multiply(x, y) {
  return x * y;
}

console.log(executeOperation(multiply, 5, 3)); // 15

// Common with array methods
[1, 2, 3, 4].filter(function(num) {
  return num % 2 === 0;
}); // [2, 4]
```

3. Functions Can Be Returned From Other Functions

```javascript
function createMultiplier(factor) {
  // Returns a new function that "remembers" factor
  return function(number) {
    return number * factor;
  };
}

const double = createMultiplier(2);
const triple = createMultiplier(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

4. Functions Can Be Stored in Data Structures

```javascript
const operations = {
  add: (a, b) => a + b,
  subtract: (a, b) => a - b,
  multiply: (a, b) => a * b,
  divide: (a, b) => a / b
};

console.log(operations.multiply(4, 5)); // 20

const functionArray = [
  () => console.log("First"),
  () => console.log("Second"),
  () => console.log("Third")
];

functionArray.forEach(fn => fn());
```

5. Functions Can Have Properties and Methods

```javascript
function counter() {
  counter.count = (counter.count || 0) + 1;
  return counter.count;
}

console.log(counter()); // 1
console.log(counter()); // 2

counter.reset = function() {
  counter.count = 0;
};

counter.reset();
console.log(counter()); // 1
```

Transformative Patterns Enabled by First-Class Functions

Closures

Closures—one of JavaScript's most powerful features—are a direct result of functions being first-class citizens:

```javascript
function createCounter() {
  let count = 0;  // Private state

  return {
    increment: function() {
      count++;
      return count;
    },
    decrement: function() {
      count--;
      return count;
    },
    getValue: function() {
      return count;
    }
  };
}

const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
console.log(counter.getValue());  // 2
```

The returned functions "close over" the `count` variable, maintaining access to it even after `createCounter` finishes execution.

Higher-Order Functions

Functions that operate on other functions opened the door to powerful abstractions:

```javascript
// Function composition
function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}

const addOne = x => x + 1;
const double = x => x * 2;
const doubleAddOne = compose(addOne, double);

console.log(doubleAddOne(5)); // 11 (5*2 + 1)
```

Callback Pattern

Before promises and async/await, callbacks were the primary way to handle asynchronous operations:

```javascript
function fetchData(url, onSuccess, onError) {
  // Simulated async operation
  setTimeout(() => {
    if (url) {
      onSuccess({ data: "Success!" });
    } else {
      onError(new Error("Failed to fetch data"));
    }
  }, 1000);
}

fetchData(
  "example.com/api",
  result => console.log(result.data),
  error => console.error(error.message)
);
```

Function Decorators

Functions that enhance other functions:

```javascript
function logExecutionTime(fn) {
  return function(...args) {
    const start = Date.now();
    const result = fn(...args);
    console.log(`Execution took ${Date.now() - start}ms`);
    return result;
  };
}

const slowAdd = logExecutionTime(function(a, b) {
  let result = 0;
  for (let i = 0; i < 1000000; i++) result += 1;
  return a + b + result;
});

slowAdd(3, 4); // Logs execution time and returns result
```

Partial Application and Currying

Breaking down multi-parameter functions into sequences of single-parameter functions:

```javascript
// Partial application
function partial(fn, ...fixedArgs) {
  return function(...remainingArgs) {
    return fn(...fixedArgs, ...remainingArgs);
  };
}

function greet(greeting, name) {
  return `${greeting}, ${name}!`;
}

const sayHello = partial(greet, "Hello");
console.log(sayHello("World")); // "Hello, World!"

// Currying
function curry(fn) {
  return function curried(...args) {
    if (args.length >= fn.length) {
      return fn(...args);
    }
    return function(...moreArgs) {
      return curried(...args, ...moreArgs);
    };
  };
}

const curriedAdd = curry((a, b, c) => a + b + c);
console.log(curriedAdd(1)(2)(3)); // 6
```

Historical Context and Influence

JavaScript's treatment of functions was heavily influenced by Scheme, a Lisp dialect known for its powerful functional programming capabilities. This was a radical departure from mainstream languages of the 1990s like Java and C++, which treated functions as second-class constructs.

This design decision has proven remarkably prescient. Modern JavaScript frameworks like React (with its functional components), Redux (with its reducers and action creators), and the entire functional programming renaissance in JavaScript all build upon this foundation.

Even JavaScript's async/await, introduced decades later, is ultimately a syntactic abstraction over functions returning promises—only possible because functions can be passed around and returned like any other value.

The concept of first-class functions has been so successful that many newer languages—from Swift to Kotlin to TypeScript—have adopted similar capabilities, validating Brendan Eich's early design choices despite JavaScript's rushed creation.

Using `this` in Node.js

When you use `console.log(this)` in Node.js, what you'll see depends significantly on where the code is executed. Let me walk through the different contexts:

In the Global Scope

When you run `console.log(this)` at the top level of a Node.js script:

```javascript
// In a file called test.js
console.log(this);
```

And execute it with `node test.js`, you'll see:

```
{}
```

This is an empty object, which is the global "module" context in Node.js. It's different from browsers where the global `this` would reference the `window` object.

In a Function Declaration (Not in Strict Mode)

```javascript
// Non-strict mode
function checkThis() {
  console.log(this);
}

checkThis();
```

You'll see the global object:

```
<ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout],
  queueMicrotask: [Function: queueMicrotask],
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate],
  // ... many more global properties
}
```

This is Node.js's global object, similar to `window` in browsers.

In a Function Declaration (Strict Mode)

```javascript
'use strict';
function checkThis() {
  console.log(this);
}

checkThis();
```

You'll see:

```
undefined
```

In strict mode, `this` is `undefined` when the function is called without a context.

In a Method

```javascript
const obj = {
  name: 'Test Object',
  showThis: function() {
    console.log(this);
  }
};

obj.showThis();
```

You'll see the object itself:

```
{ name: 'Test Object', showThis: [Function: showThis] }
```

In an Arrow Function

```javascript
const arrowThis = () => {
  console.log(this);
};

arrowThis();
```

You'll see the same value as the enclosing scope:

```
{}
```

(At global scope, which is an empty object in modules)

In a Class Constructor

```javascript
class TestClass {
  constructor() {
    console.log(this);
  }
}

new TestClass();
```

You'll see the instance being created:

```
TestClass {}
```

In a Class Method

```javascript
class TestClass {
  constructor() {
    this.name = "Test Instance";
  }

  showThis() {
    console.log(this);
  }
}

const instance = new TestClass();
instance.showThis();
```

You'll see:

```
TestClass { name: 'Test Instance' }
```

In a Callback

```javascript
function callWithCallback(callback) {
  callback();
}

callWithCallback(function() {
  console.log(this);
});
```

You'll see the global object (non-strict) or `undefined` (strict mode).

Node.js Module System

Node.js's CommonJS module system is a bit different:

```javascript
// In a module file
console.log(this === module.exports); // true
```

In a module, `this` refers to `module.exports` (which starts as an empty object).

A Complete Example

Here's a more comprehensive example showcasing different contexts:

```javascript
// file: this-test.js
console.log("Global scope:", this);

function regularFunction() {
  console.log("Regular function:", this);
}

const arrowFunction = () => {
  console.log("Arrow function:", this);
};

const obj = {
  method() {
    console.log("Object method:", this);

    function innerFunction() {
      console.log("Inner function:", this);
    }

    const innerArrow = () => {
      console.log("Inner arrow:", this);
    };

    innerFunction();
    innerArrow();
  }
};

regularFunction();
arrowFunction();
obj.method();
```

Running this in Node.js would output:

```
Global scope: {}
Regular function: Object [global] { ... }
Arrow function: {}
Object method: { method: [Function: method] }
Inner function: Object [global] { ... }
Inner arrow: { method: [Function: method] }
```

The key takeaway is that Node.js's behavior with `this` follows JavaScript's rules, but the global context is different from browsers, and modules have their own specific behavior where `this` equals `module.exports`.

How `this` Is Determined by Call Context, Not Definition

Here's a clear example that demonstrates how the value of `this` is determined by how a function is called rather than where it's defined:

```
javascript
```

```javascript
// Let's create a simple function that reports its 'this' context
function reportThis() {
  console.log("The value of 'this' is:", this);
  if (this.name) {
    console.log(`This has a name property: ${this.name}`);
  } else {
    console.log("This does not have a name property");
  }
  console.log("-------------------");
}

// Create different objects with name properties
const object1 = { name: "Object 1", method: reportThis };
const object2 = { name: "Object 2", method: reportThis };

// SAME FUNCTION, different calls
// ===============================

// 1. Call as a standalone function
console.log("Calling as standalone function:");
reportThis();
// 'this' will be the global object (or undefined in strict mode)

// 2. Call as a method of object1
console.log("Calling as a method of object1:");
object1.method();
// 'this' will be object1

// 3. Call as a method of object2
console.log("Calling as a method of object2:");
object2.method();
// 'this' will be object2

// 4. Store the method in a variable and call it
console.log("Calling via a variable:");
const standalone = object1.method;
standalone();
// 'this' loses its context and becomes global object again

// 5. Using call to explicitly set 'this'
console.log("Calling with explicit this using call:");
reportThis.call(object2);
// 'this' will be object2

// 6. Using bind to create a function with fixed 'this'
console.log("Calling with bound this:");
```

```
const boundFunction = reportThis.bind(object1);
boundFunction();
// 'this' will always be object1, regardless of how it's called

// 7. Even when attached to a different object
console.log("Calling bound function as method of object2:");
object2.boundMethod = boundFunction;
object2.boundMethod();
// 'this' will STILL be object1, because of bind
```

When you run this code in Node.js, you'll see:

```
Calling as standalone function:
The value of 'this' is: Object [global] { ... }
This does not have a name property
-------------------

Calling as a method of object1:
The value of 'this' is: { name: 'Object 1', method: [Function: reportThis] }
This has a name property: Object 1
-------------------

Calling as a method of object2:
The value of 'this' is: { name: 'Object 2', method: [Function: reportThis] }
This has a name property: Object 2
-------------------

Calling via a variable:
The value of 'this' is: Object [global] { ... }
This does not have a name property
-------------------

Calling with explicit this using call:
The value of 'this' is: { name: 'Object 2', method: [Function: reportThis] }
This has a name property: Object 2
-------------------

Calling with bound this:
The value of 'this' is: { name: 'Object 1', method: [Function: reportThis] }
This has a name property: Object 1
-------------------

Calling bound function as method of object2:
The value of 'this' is: { name: 'Object 1', method: [Function: reportThis] }
This has a name property: Object 1
-------------------
```

The Key Insight

Notice how the **exact same function** (`reportThis`) returns different values for `this` based solely on how it is called:

1. When called directly: `this` is the global object
2. When called as `object1.method()`: `this` is `object1`
3. When called as `object2.method()`: `this` is `object2`
4. When called via a variable: `this` is the global object again

5. When called with `.call(object2)`: `this` is `object2`
6. When bound using `.bind(object1)`: `this` is permanently set to `object1`

This demonstrates that it's not the definition or location of the function that determines what `this` will reference – it's entirely about the calling context at runtime.

Practical Implications

This behavior is why JavaScript developers often encounter issues when:

- Passing object methods as callbacks (losing the intended `this`)
- Using `this` in nested functions (inner functions don't inherit the outer `this`)
- Working with event handlers (where `this` is the element that fired the event)

The solution to these problems often involves:

1. Using arrow functions (which don't have their own `this` and inherit from the surrounding scope)
2. Using `.bind()` to fix `this` to a specific value
3. Storing `this` in a variable like `const self = this;` before entering a new function context

This dynamic behavior of `this` is both a powerful feature and a common source of bugs in JavaScript, which is why understanding it is crucial for effective JavaScript programming.

Troubleshooting Node.js Console Output

If you're running this specific code in Node.js and not seeing any output:

```javascript
function reportThis() {
  console.log("The value of this is", this);
}

reportThis();
```

Here are troubleshooting steps:

Step 1: Create a clean test file

Create a new file called `this-test.js` with exactly this content:

```javascript
function reportThis() {
  console.log("The value of 'this' is:", this);
}

console.log("About to call reportThis()");
reportThis();
console.log("After calling reportThis()");
```

Step 2: Run it directly from the terminal

Open your terminal/command prompt, navigate to the directory containing the file, and run:

```
node this-test.js
```

You should see output like:

```
About to call reportThis()
The value of 'this' is: {}
After calling reportThis()
```

Common Issues & Solutions

If nothing appears at all:

1. **Check file path:** Make sure you're in the correct directory where the file exists.
2. **Check Node.js installation:** Run `node -v` to verify Node.js is properly installed.
3. **Try a simpler test file:**

   ```javascript
   // simple-test.js
   console.log("Hello, world!");
   ```

   Then run `node simple-test.js`
4. **Check for syntax errors:** Make sure there are no syntax errors in your file.

If you see "The value of 'this' is: [Object: null prototype] {}" or similar:

This is actually normal and expected. In Node.js modules, at the global level, `this` is equal to `module.exports`, which starts as an empty object.

If you're running this in an IDE or editor:

Some IDEs have their own integrated terminals or run configurations. Try running directly from the system terminal instead to rule out issues with IDE console output.

If there are no visible errors but still no output:

Add more debug statements before and after your function call to see where execution might be stopping.