Back-end Web Development

February 11th, 2020

Dorset College Dublin

JavaScript

What we covered last week....

- **Refresher**
  - Primitive data types
  - JS specifics
  - Declaring a variable
  - Data type conversion
  - Operators
- **var, let, const**
- **Arrays**
  - Creating an array
  - Accessing values
  - Changing values

- **Control Flow**
  - Logical operators
  - Falsy values
  - Equality
  - Relational Expressions
  - Variable Hoisting
  - not defined, undefined, null

JavaScript

What we will cover today....

- **Functions**
  - Parameters and Arguments
  - Return
  - Function declaration
  - Function expression
- **History of JS**
- **ES6 syntax**

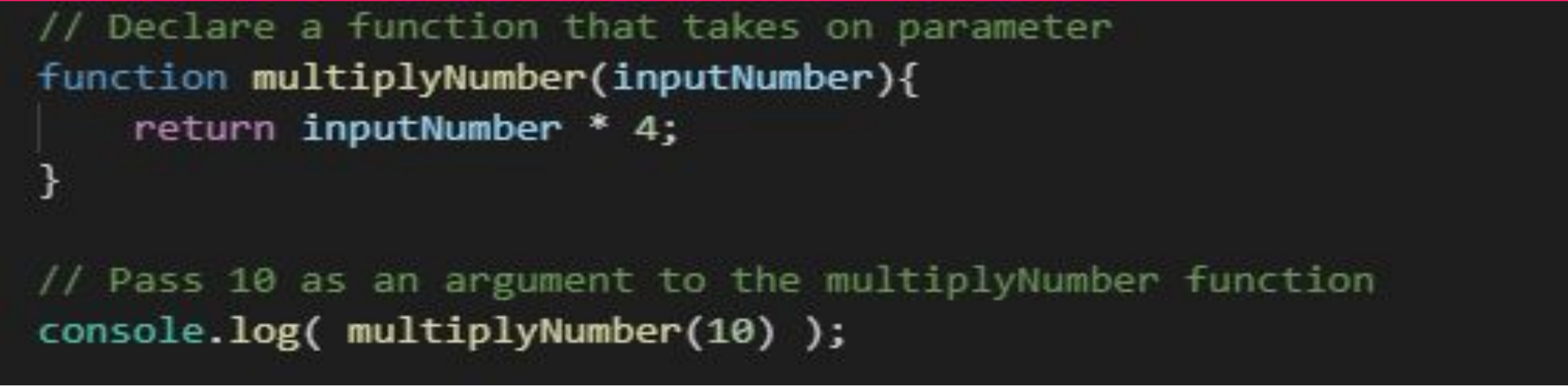
- **Arrow Functions**
  - Refactoring
  - Steps
  - Activities
- **Functions as Values**
- **Function Hoisting**

Functions

A function is a **block** of JS code that is defined once but may be executed or invoked **any number** of times.

- If a function is assigned to the **property** of an object, it is known as a **method** of that object
- In JS functions are **objects** and they can be manipulated by programs
- JS can **assign** functions to variables and **pass them** to other functions

Example Function - Parameters and Arguments



```
// Declare a function that takes on parameter
function multiplyNumber(inputNumber){
    return inputNumber * 4;
}

// Pass 10 as an argument to the multiplyNumber function
console.log( multiplyNumber(10) );
```

- **inputNumber** is a **parameter**
- The number inside the **multiplyNumber** call is an **argument**
- Arguments are provided when **calling** a function and parameters **receive** arguments as their value
- We pass a value to the function when setting a value as the argument

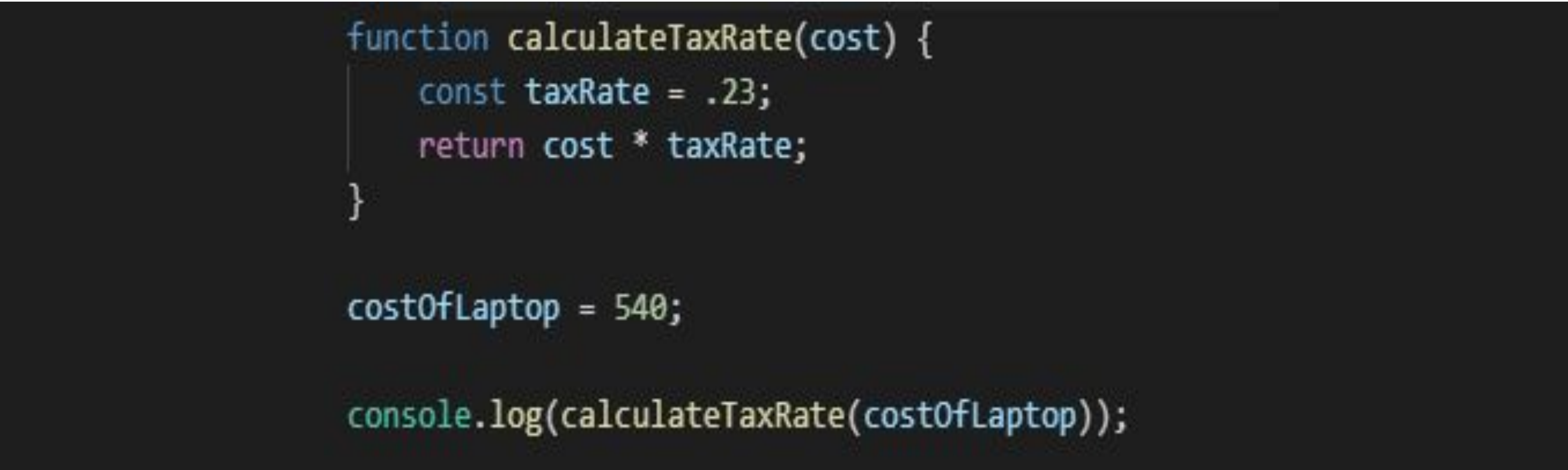
Return

- Sends a **value back** from the function to where it was invoked
- We can just log the result to the console inside the function body however its **best practise** to use return
- There are **2 times** you'll want to use return in a function:
  - When you literally want to **return a value**
  - When you want the function to **stop running**
- If we return a value inside a function, it can be used **anywhere else** in the code

Types of Functions

Function Declaration

A function declaration is a function that is bound to an **identifier** or name.



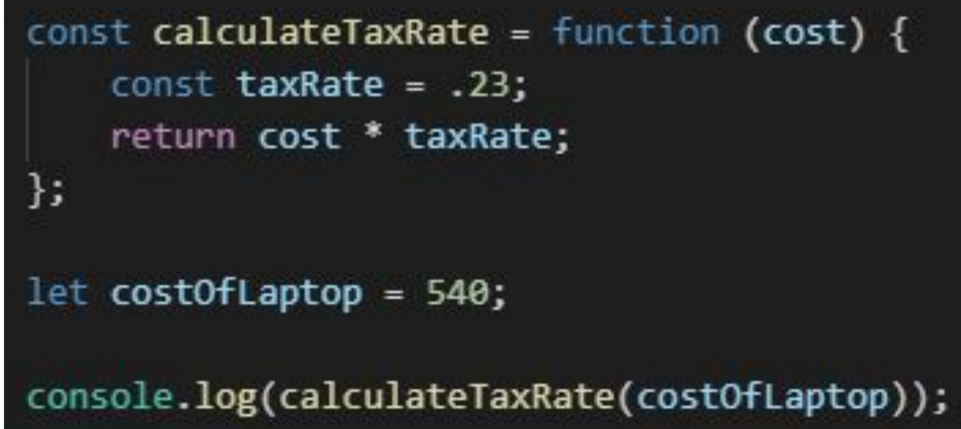
```
function calculateTaxRate(cost) {
    const taxRate = .23;
    return cost * taxRate;
}

costOfLaptop = 540;

console.log(calculateTaxRate(costOfLaptop));
```

Types of Functions

Function Expression

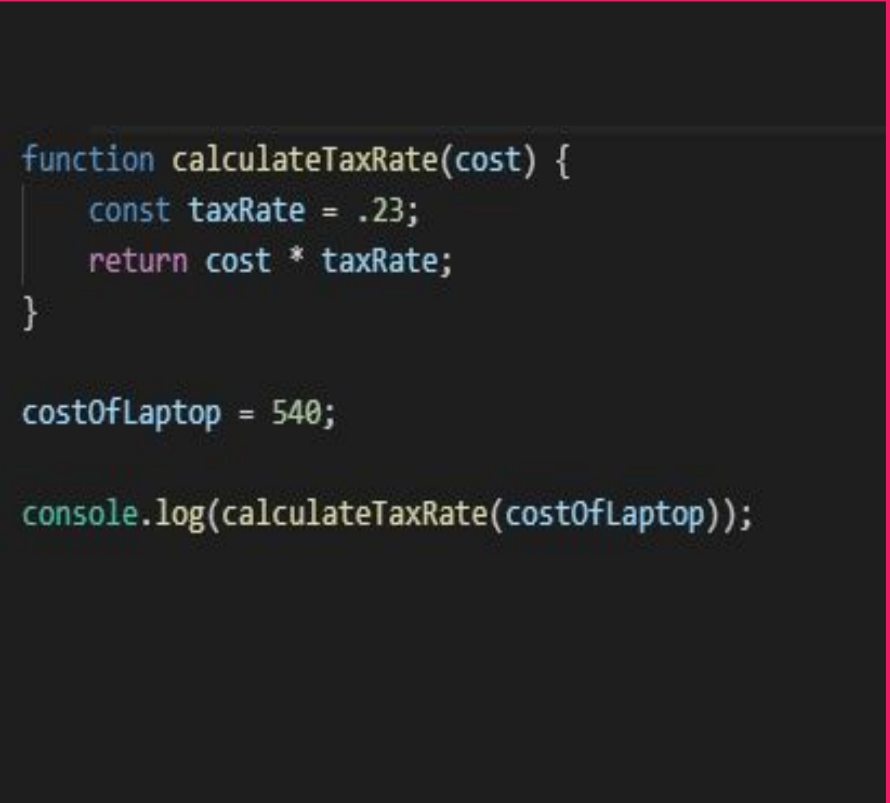
A function expression is similar to a function declaration, with the exception that the **identifier can be omitted** which creates an **anonymous** function.

- Function expressions are often **stored** in a variable
- You can identify a function expression by the **absence of a function name** trailing the function keyword

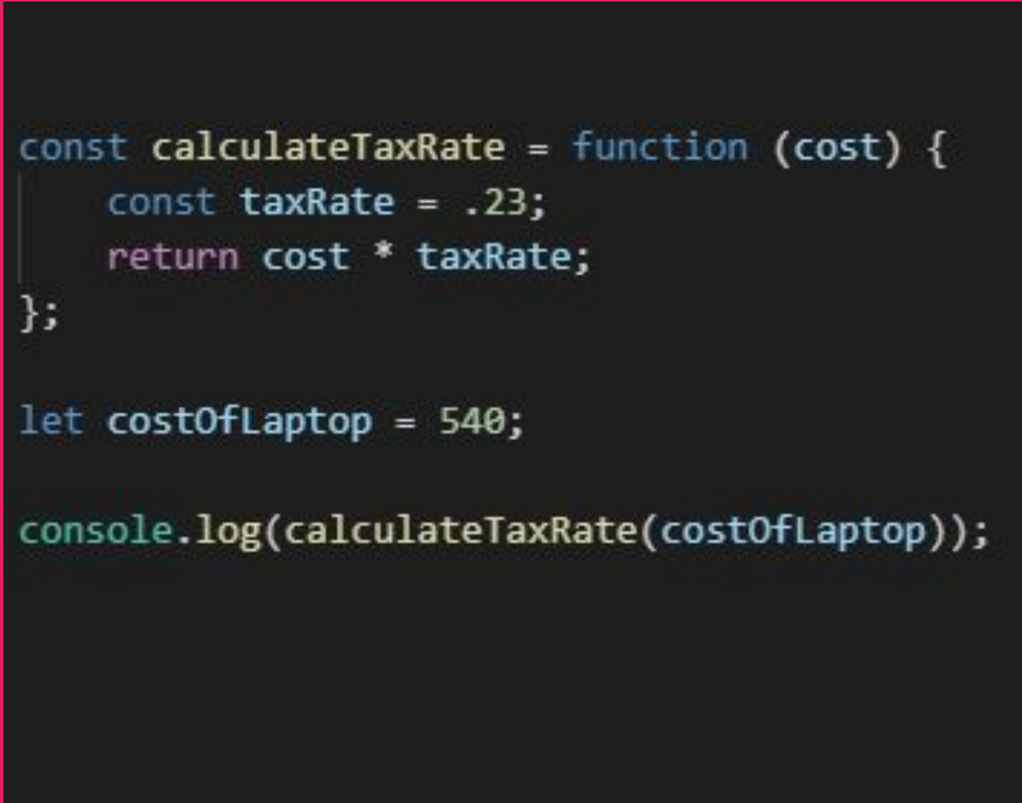


```
const calculateTaxRate = function (cost) {
    const taxRate = .23;
    return cost * taxRate;
};

let costOfLaptop = 540;

console.log(calculateTaxRate(costOfLaptop));
```

Declaration Vs. Expression



```
function calculateTaxRate(cost) {
    const taxRate = .23;
    return cost * taxRate;
}

costOfLaptop = 540;

console.log(calculateTaxRate(costOfLaptop));
```



```
const calculateTaxRate = function (cost) {
    const taxRate = .23;
    return cost * taxRate;
};

let costOfLaptop = 540;

console.log(calculateTaxRate(costOfLaptop));
```

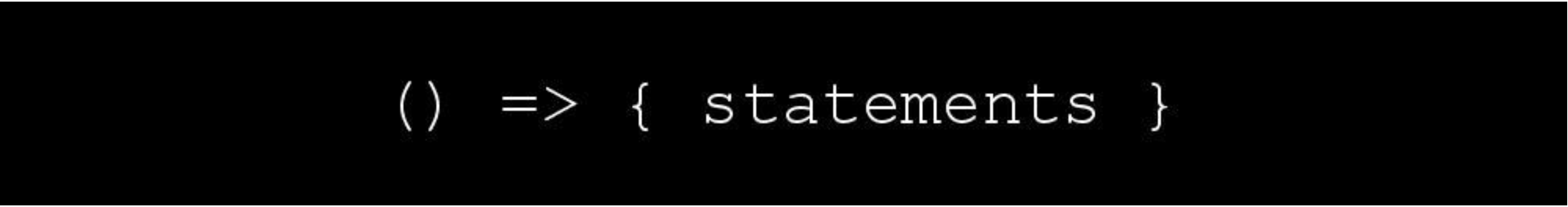
Year	History of JS
1995	JavaScript is born as LiveScript
1997	ECMAScript standard is established (ES1)
1998	ES2
1999	ES3 comes out and IE5 is all the rage….ES4 is abandoned
2000-2005	XMLHttpRequest, a.k.a. AJAX, gains popularity in apps such as Outlook Web Access (2000), Gmail (2004) and Google Maps (2005).
2009	ES5 comes out (this is still widely used now) with forEach,Object.keys,Object.create, and standard JSON
2015	ES6/ECMAScript2015 comes out with significant new features
2016	ES7, 2017: ES8, 2018: ES9 Smaller new features added in each

New in ES6

- Let and const
- Arrow functions
- Classes
- Enhanced object literals
- Template strings
- destructuring
- Default + rest + spread
- Iterators + for...of
- Modules
- Promises
- And many more...

Arrow Functions (ES6)

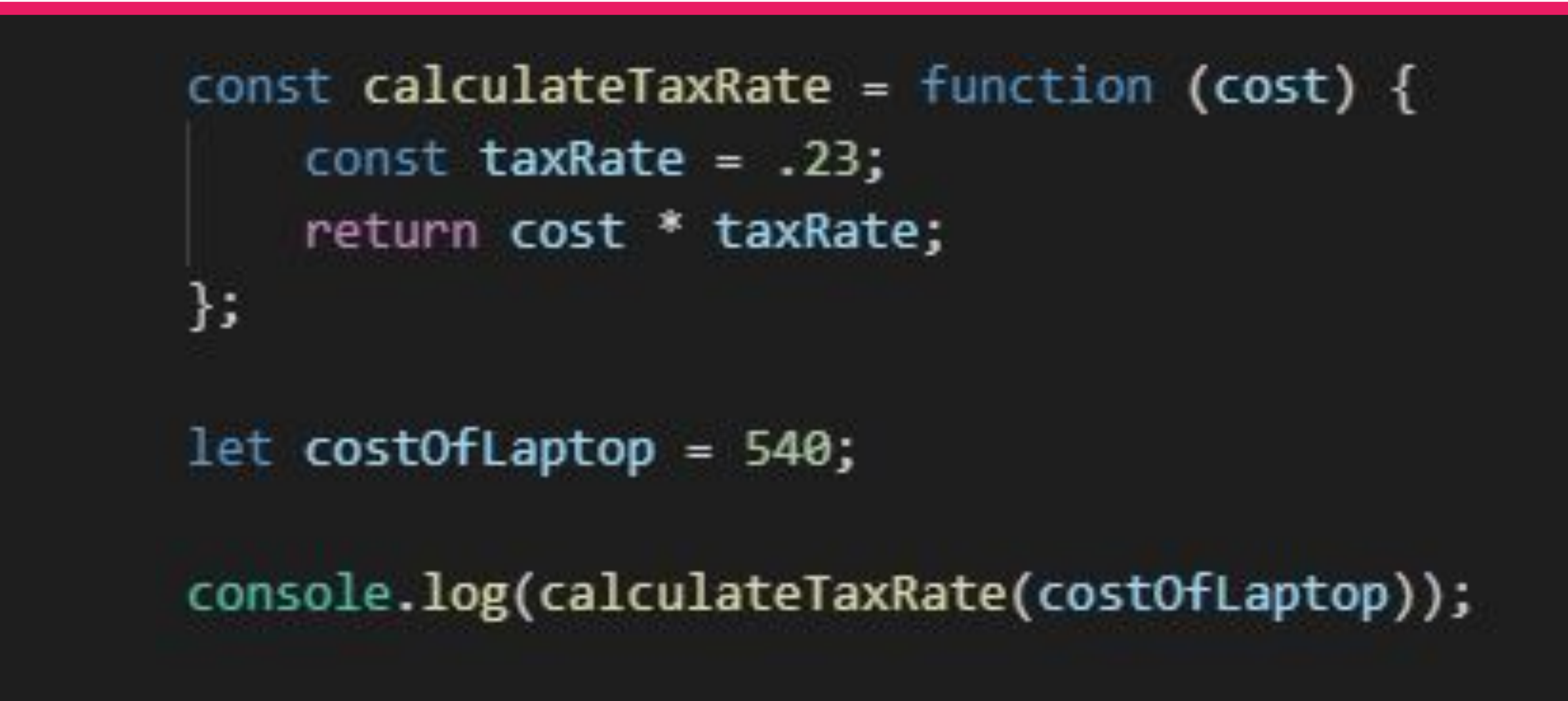
- JS ES6 provides **new syntax** to write functions in a more concise way using **arrow tokens**



```
() => { statements }
```

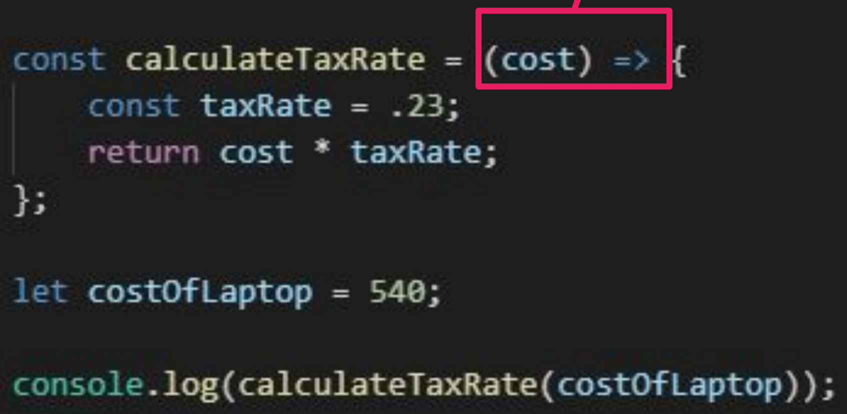
- The process of changing code without changing its **external behavior** is called **refactoring.** We can refactor code in a number of ways to make it more readable, maintainable and extensible

Before Refactoring



```
const calculateTaxRate = function (cost) {
    const taxRate = .23;
    return cost * taxRate;
};

let costOfLaptop = 540;

console.log(calculateTaxRate(costOfLaptop));
```

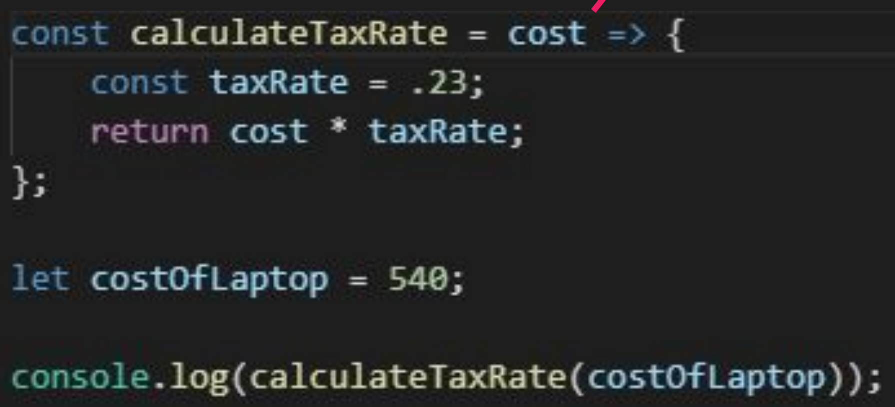
Step 1



Get rid of the function keyword and add the arrow token

```
const calculateTaxRate = (cost) => {
    const taxRate = .23;
    return cost * taxRate;
};

let costOfLaptop = 540;

console.log(calculateTaxRate(costOfLaptop));
```

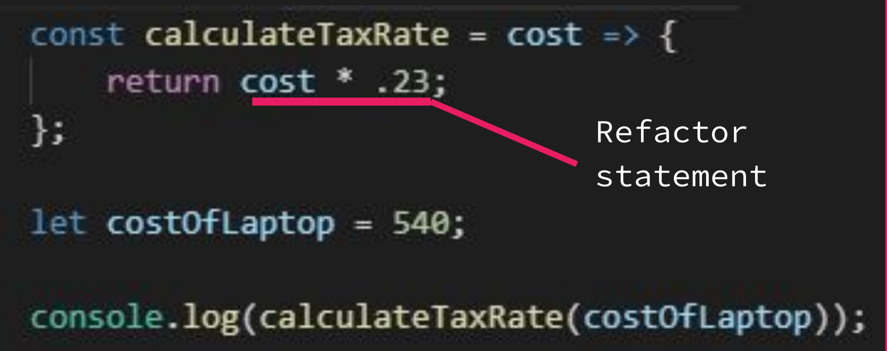
Step 2



Get rid of parenthesis

```
const calculateTaxRate = cost => {
    const taxRate = .23;
    return cost * taxRate;
};

let costOfLaptop = 540;

console.log(calculateTaxRate(costOfLaptop));
```

Step 3

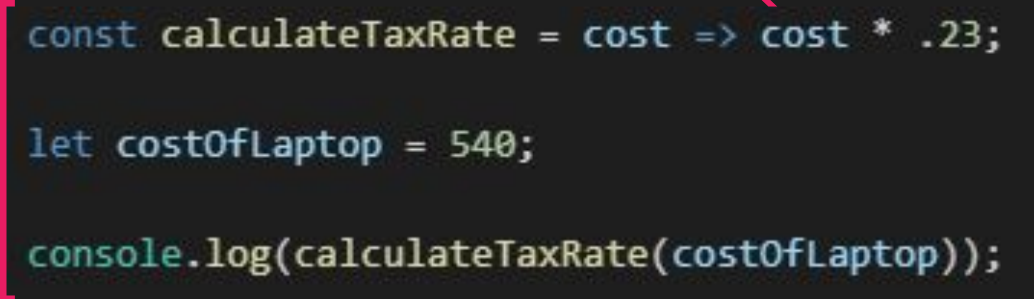


```
const calculateTaxRate = cost => {
    return cost * .23;
};

let costOfLaptop = 540;

console.log(calculateTaxRate(costOfLaptop));
```

Refactor statement

Step 4

Get rid of curly brackets, return and semicolon



```
const calculateTaxRate = cost => cost * .23;

let costOfLaptop = 540;

console.log(calculateTaxRate(costOfLaptop));
```

Refactoring Explained

Step 1: function is replaced with arrow token

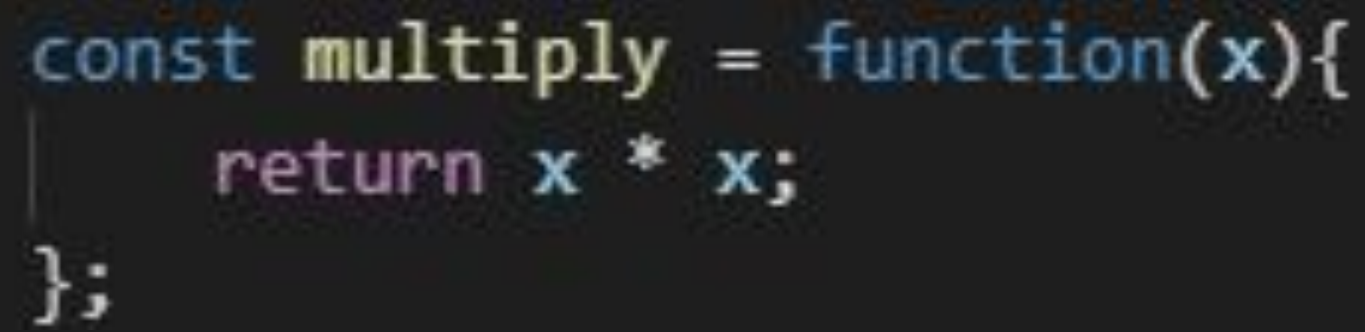
Step 2: as this function takes one parameter the parentheses can be removed. However, if a function takes zero or multiple parameters, parentheses are required

Step 3: We reduce the code needed in the body

Step 4: as the function contains a single-line block it can be placed immediately after the arrow =>. This is called an implicit return.

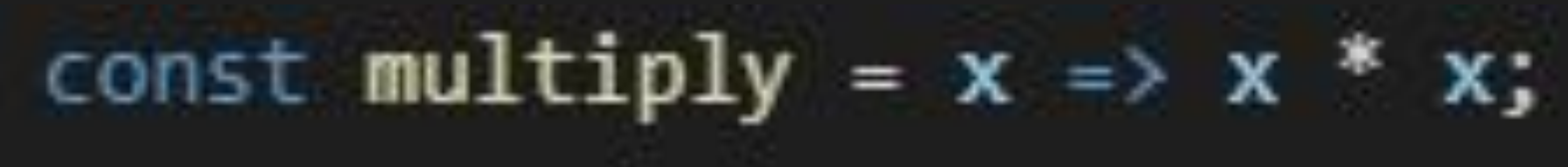
Activity

Refactor this function expression to use arrow tokens.



```
const multiply = function(x){
    return x * x;
};
```

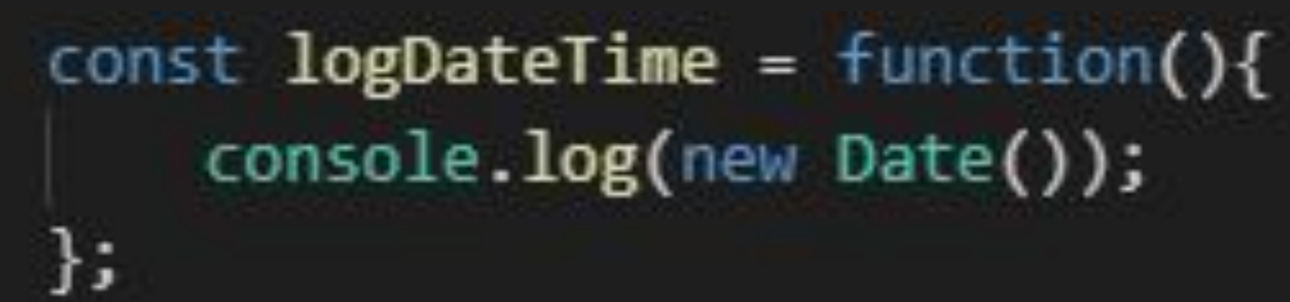
Solution



```
const multiply = x => x * x;
```

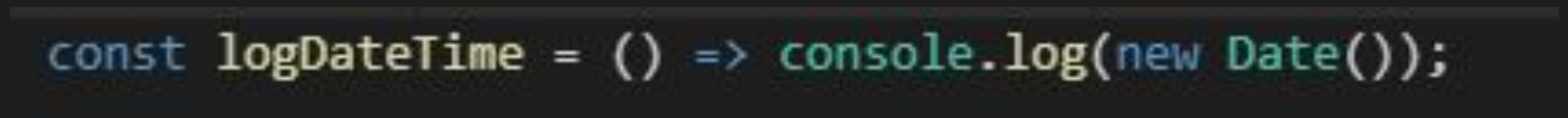
Activity

Refactor this function expression to use arrow tokens.



```
const logDateTime = function(){
    console.log(new Date());
};
```

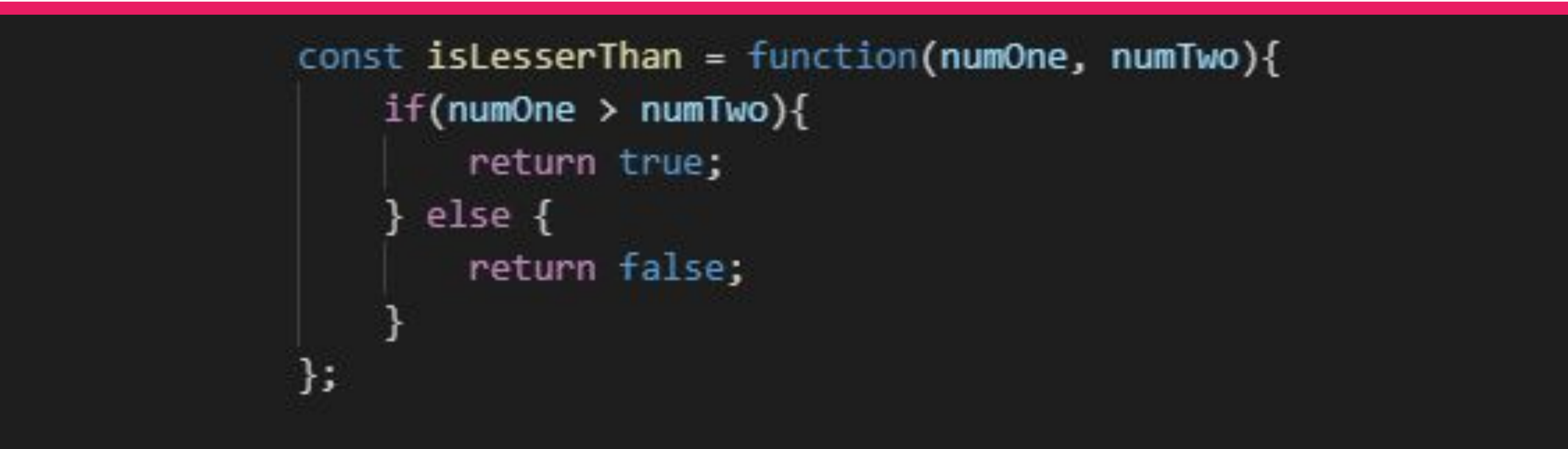
Solution



```
const logDateTime = () => console.log(new Date());
```

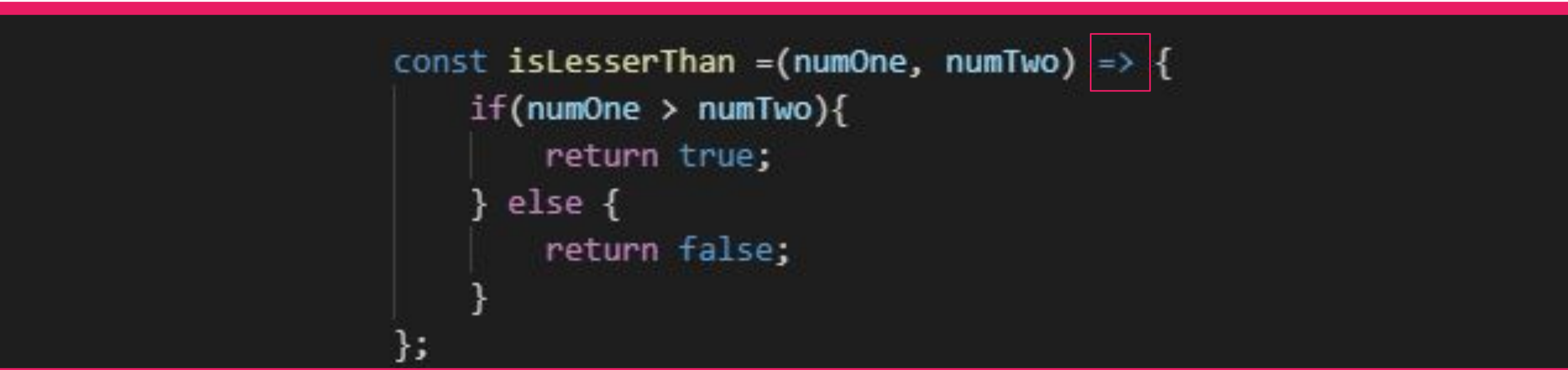
Activity

Refactor this function expression to use arrow tokens.



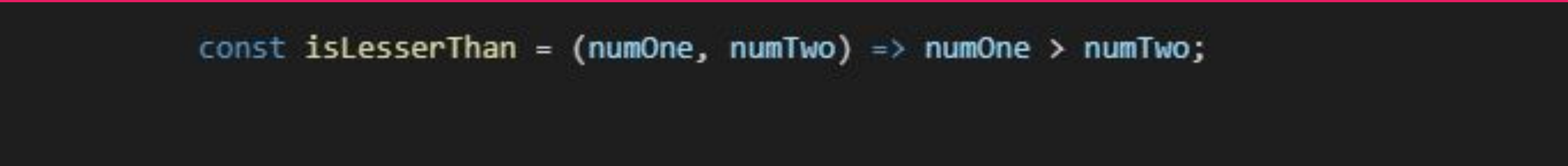
```
const isLesserThan = function(numOne, numTwo){
    if(numOne > numTwo){
        return true;
    } else {
        return false;
    }
};
```

Refactored to use arrow token



```
const isLesserThan =(numOne, numTwo) => {
    if(numOne > numTwo){
        return true;
    } else {
        return false;
    }
};
```

Refactored to remove if else

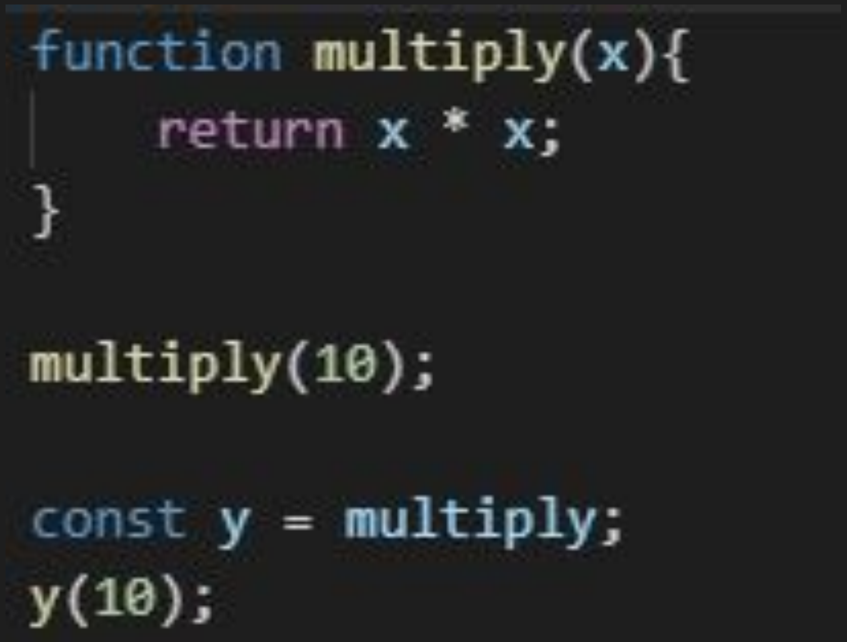


```
const isLesserThan = (numOne, numTwo) => numOne > numTwo;
```

Functions as Values

JS has **first-class functions** which means that they can be treated like other variables and have methods and properties **like objects.**

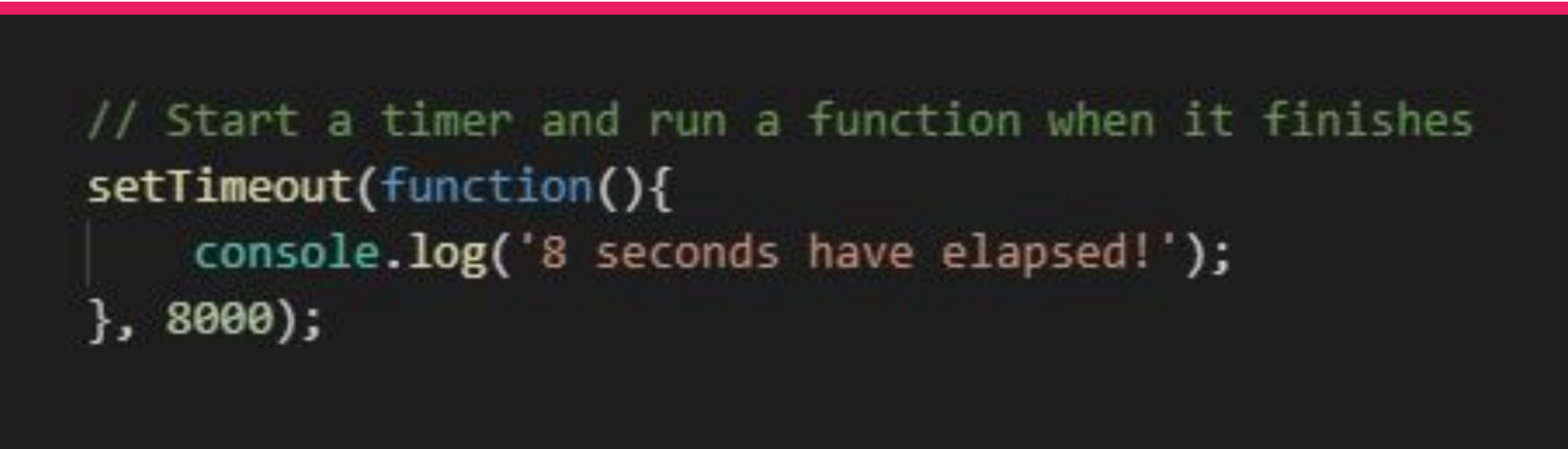
Assigning a function to a variable



```
function multiply(x){
    return x * x;
}

multiply(10);

const y = multiply;
y(10);
```

Functions as Values

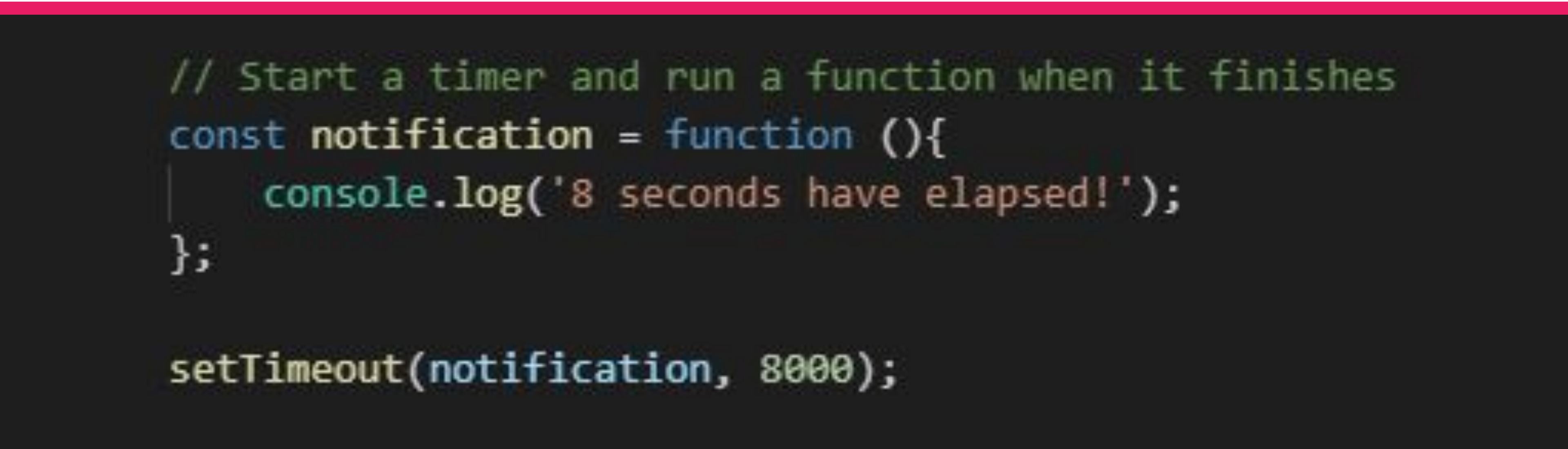
Here we are passing an **anonymous function** in to another function (**setTimeout**) as an argument:



```
// Start a timer and run a function when it finishes
setTimeout(function(){
    console.log('8 seconds have elapsed!');
}, 8000);
```

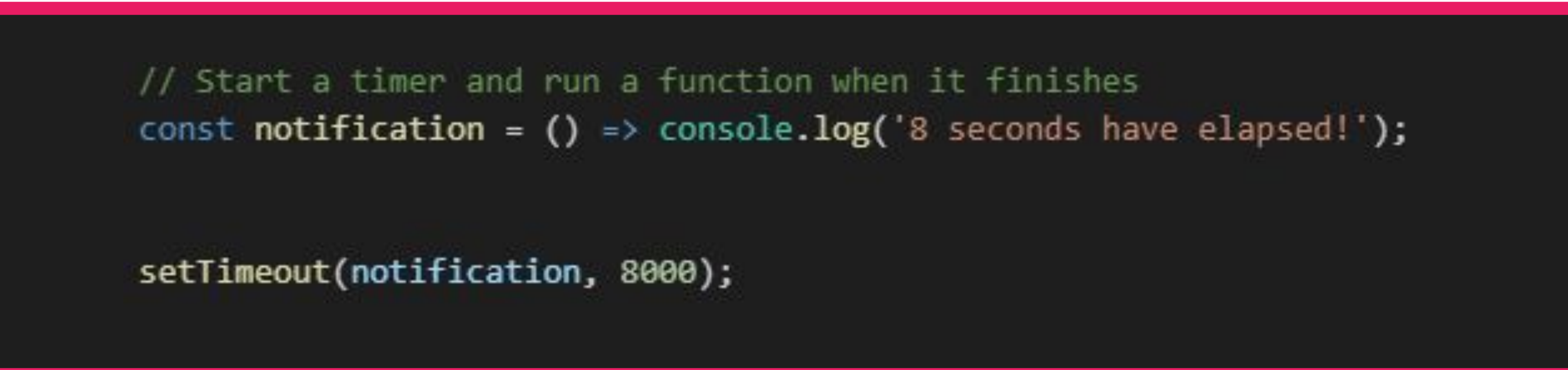
Functions as Values

However to **reuse** the function, we need to store it in a variable and then pass the **setTimeout** function:



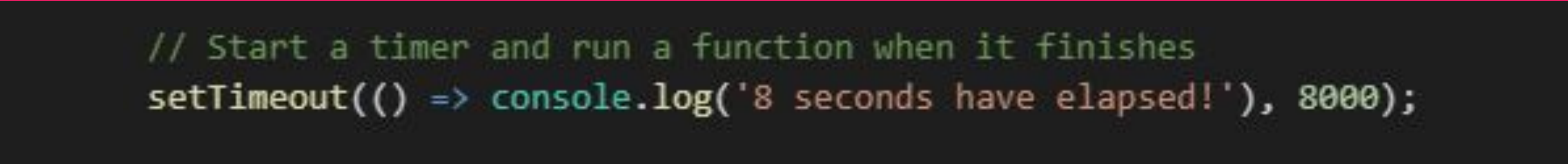
```
// Start a timer and run a function when it finishes
const notification = function (){
    console.log('8 seconds have elapsed!');
};

setTimeout(notification, 8000);
```

Refactored to use arrow token



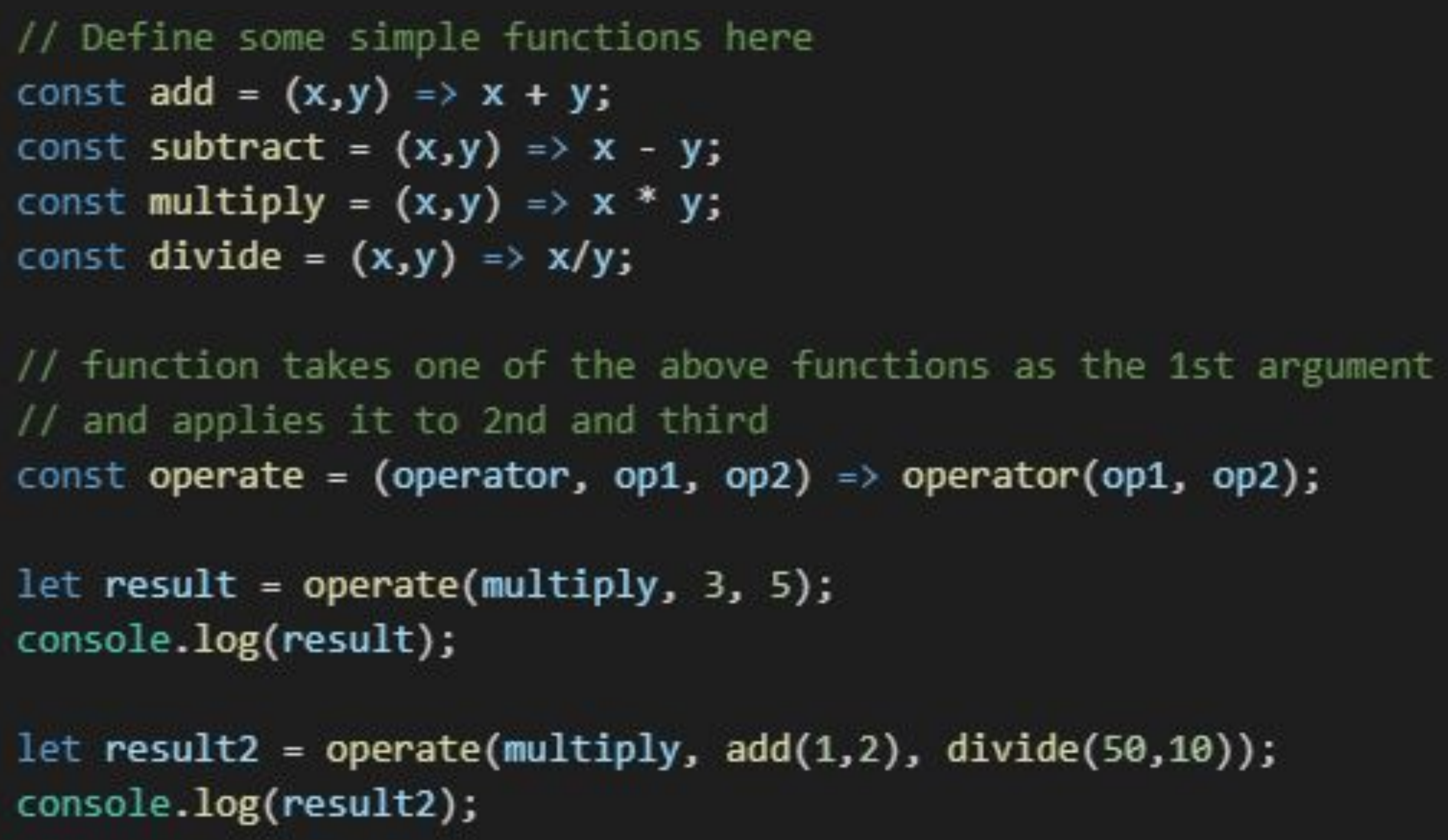
```
// Start a timer and run a function when it finishes
const notification = () => console.log('8 seconds have elapsed!');


setTimeout(notification, 8000);
```

Refactored to a single statement



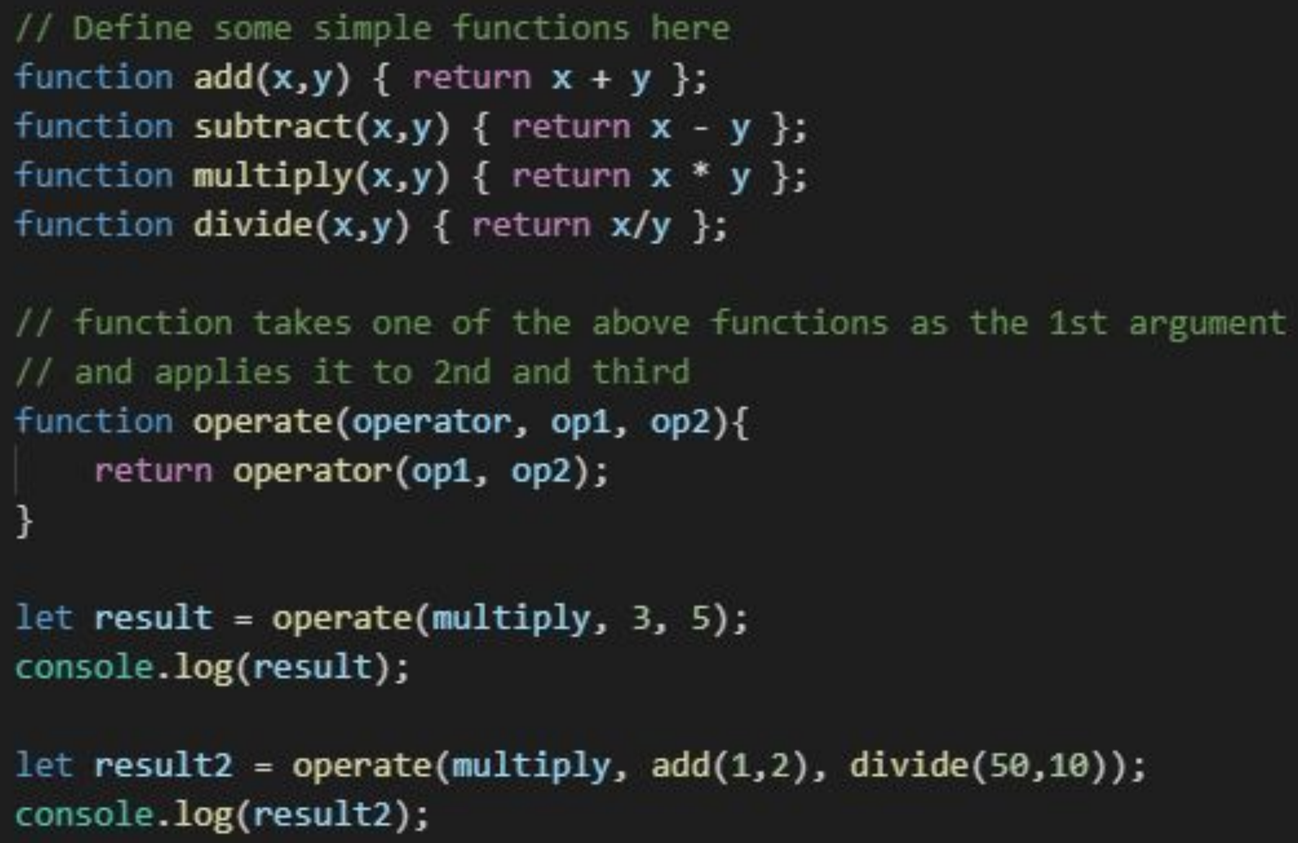
```
// Start a timer and run a function when it finishes
setTimeout(() => console.log('8 seconds have elapsed!'), 8000);
```

Refactored function declaration



```
// Define some simple functions here
const add = (x,y) => x + y;
const subtract = (x,y) => x - y;
const multiply = (x,y) => x * y;
const divide = (x,y) => x/y;

// function takes one of the above functions as the 1st argument
// and applies it to 2nd and third
const operate = (operator, op1, op2) => operator(op1, op2);

let result = operate(multiply, 3, 5);
console.log(result);

let result2 = operate(multiply, add(1,2), divide(50,10));
console.log(result2);
```

Old-style function declaration



```
// Define some simple functions here
function add(x,y) { return x + y };
function subtract(x,y) { return x - y };
function multiply(x,y) { return x * y };
function divide(x,y) { return x/y };

// function takes one of the above functions as the 1st argument
// and applies it to 2nd and third
function operate(operator, op1, op2){
    return operator(op1, op2);
}

let result = operate(multiply, 3, 5);
console.log(result);

let result2 = operate(multiply, add(1,2), divide(50,10));
console.log(result2);
```

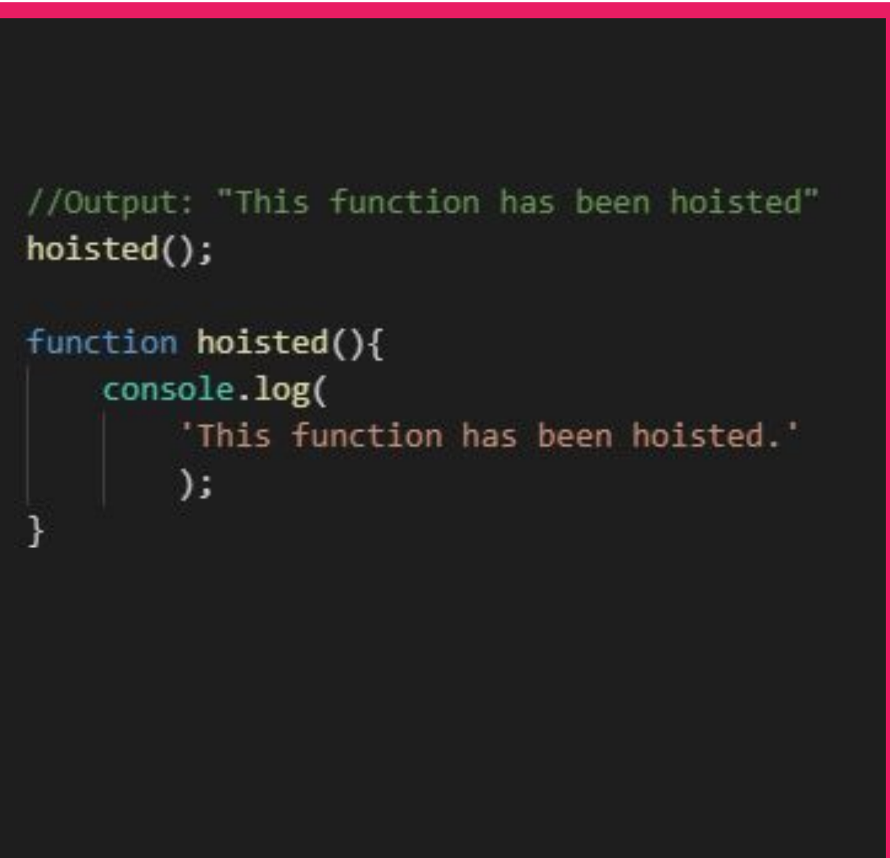
Function Hoisting

Hoisting is a JavaScript **mechanism** where variables and function declarations are **moved to the top** of their current scope before code execution.

- Function declarations are “hoisted” to the top of the enclosing script or enclosing function.

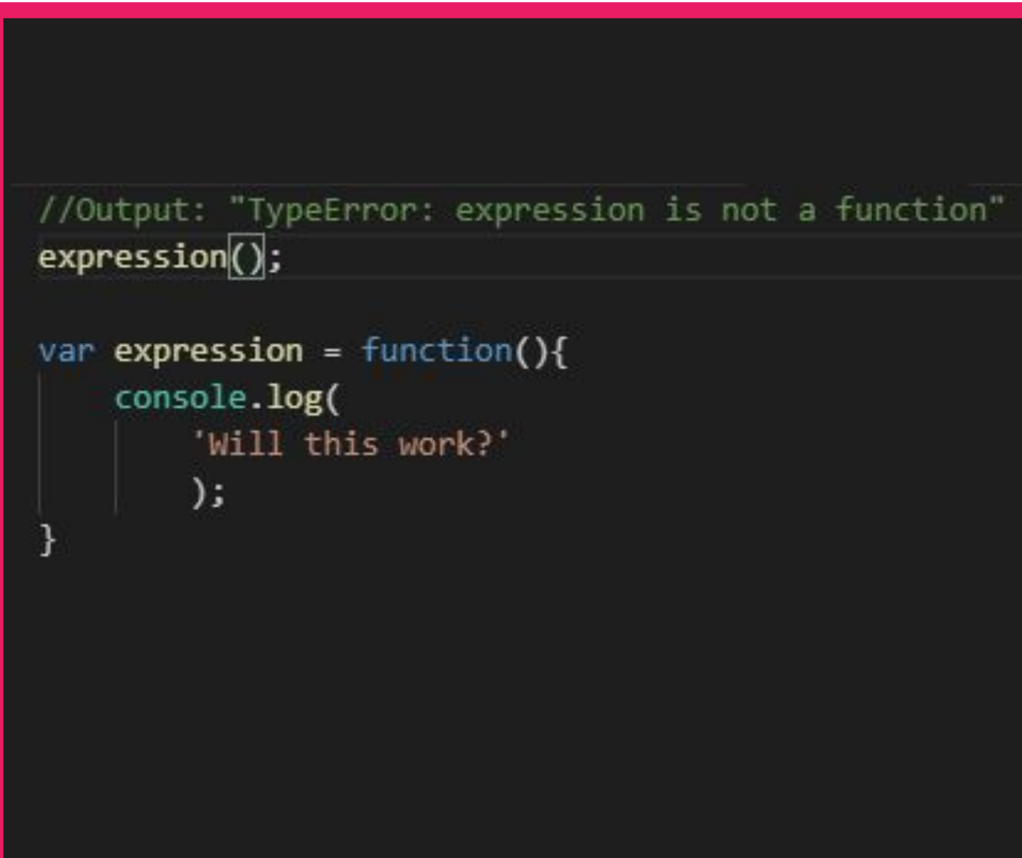
- Function Expressions however **cannot be hoisted**.

Function Declaration



```
//Output: "This function has been hoisted"
hoisted();

function hoisted(){
    console.log(
        'This function has been hoisted.'
        );
}
```

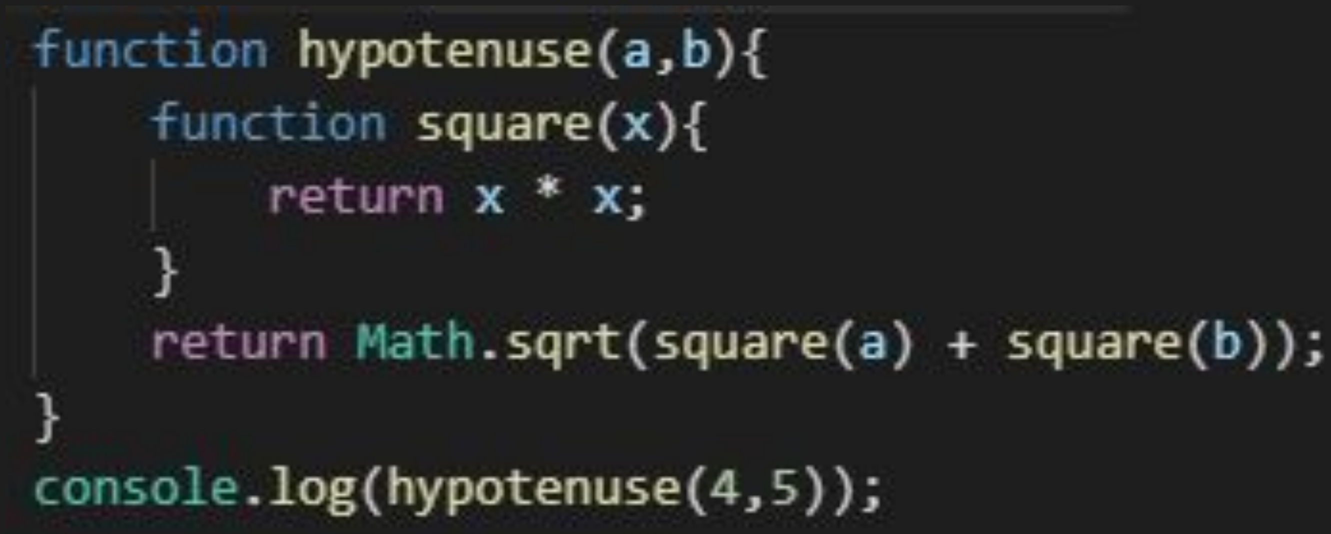
Function Expression



```
//Output: "TypeError: expression is not a function"
expression();

var expression = function(){
    console.log(
        'Will this work?'
        );
}
```

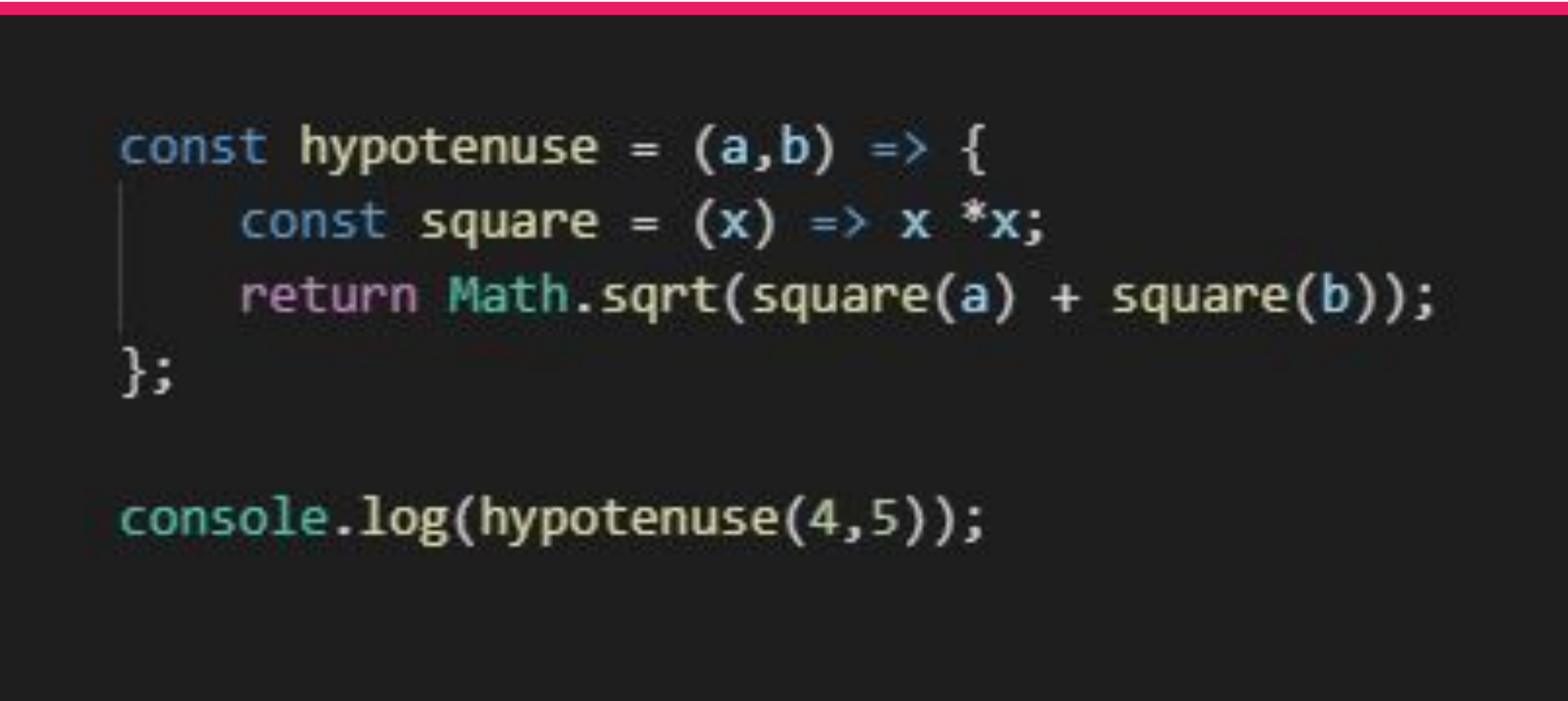
Activity

Refactor this function to use arrow tokens.



```
function hypotenuse(a,b){
    function square(x){
        return x * x;
    }
    return Math.sqrt(square(a) + square(b));
}
console.log(hypotenuse(4,5));
```

Solution - Nested Functions



```javascript
const hypotenuse = (a,b) => {
    const square = (x) => x *x;
    return Math.sqrt(square(a) + square(b));
};

console.log(hypotenuse(4,5));
```

Lets Code!

Continue on with your lab on CodeAcademy.

https://www.codecademy.com/learn/introduction-to-javascript