Welcome!

Agenda

- Review and Homework Recap
- Pseudocode
- Advanced Functions
    - Callbacks
    - Scope and Hoisting
    - Closures
    - Higher Order Functions
    - Rest Parameters
    - Spread Operator

Review

Pseudocode

Algorithms

An algorithm is a step-by-step set of operations to be performed.

Think of it like a recipe. Every program we write is a recipe that tells the computer how to do something.

JavaScript is the syntax in which we write those recipes for the web.

Pseudocode

Pseudocode is the language we use when we want to prepare to write a program (without using the syntax of a programming language).

It's a universal programming language for humans - it's essentially a shorthand we use to plan.

Pseudocode - Area

```
STORE the rectangle width as rectangleWidth

STORE the rectangle height as rectangleHeight

CALCULATE and STORE the area by:
  MULTIPLYING rectangleWidth and rectangleHeight
```

Pseudocode - Click Count

```
STORE the number of clicks as numClicks
  SET the value to be 0

EVERY TIME the button with ID "click" is clicked:
  INCREMENT numClicks
  UPDATE TEXT of paragraph with ID of "main"
```

Pseudocode - Events

```
EVERY TIME the user scrolls down the page

CHECK to see if the user is over 100px down:
  IF they are:
    SHOW the button with ID "backToTop"
  ELSE:
    HIDE the button with ID "backToTop"
```

SOLO!

Exercise

Write Pseudocode for Rock, Paper, Scissors

Part One

For a single game

Part Two

For a best of three game

SOLO!

Exercise

Create a Virtual Roll of the Dice Function

Start by writing pseudocode!

Hint: Look into `Math.random()` and `Math.floor()`

Bonus: Receive a parameter to decide how many sides the dice actually has (e.g. the function receives a 12, you are rolling a 12-sided dice)

Advanced Functions

Methods

Methods

A method is just a function that is called upon a piece of data - think of things like `"".toUpperCase()`

```
const person = {
  firstName: "Jacques",
  lastName: "Cousteau",
  sayHi: function() {
    console.log("Hi, I am Jacques");
  }
};

person.sayHi();
```

Callbacks

What are callbacks?

A callback function is really just a regular function passed into another function as an argument.

They are very useful because they allow us to schedule asynchronous actions - they are functions that serve as a response (could be an event, or an interaction with an API - or anything, really)

Callbacks

```
function runCallback(cb) {
  // Wait a second...
  cb();
}

function delayedFunction() {
  console.log("I was delayed");
}

runCallback(delayedFunction);
```

Callbacks

```javascript
function sayHi(name) {
  alert("Hello " + name);
}

function processInput(cb) {
  const name = prompt("Please enter your name.");
  cb(name);
}

processInput(sayHi);
```

Let's see some examples!

Scheduling

Scheduling

Occasionally, we don't need to run a function straight away - we want to run it after some time has elapsed, or at some regular interval.

`setTimeout`

Delays a function's execution by some amount of milliseconds

`setInterval`

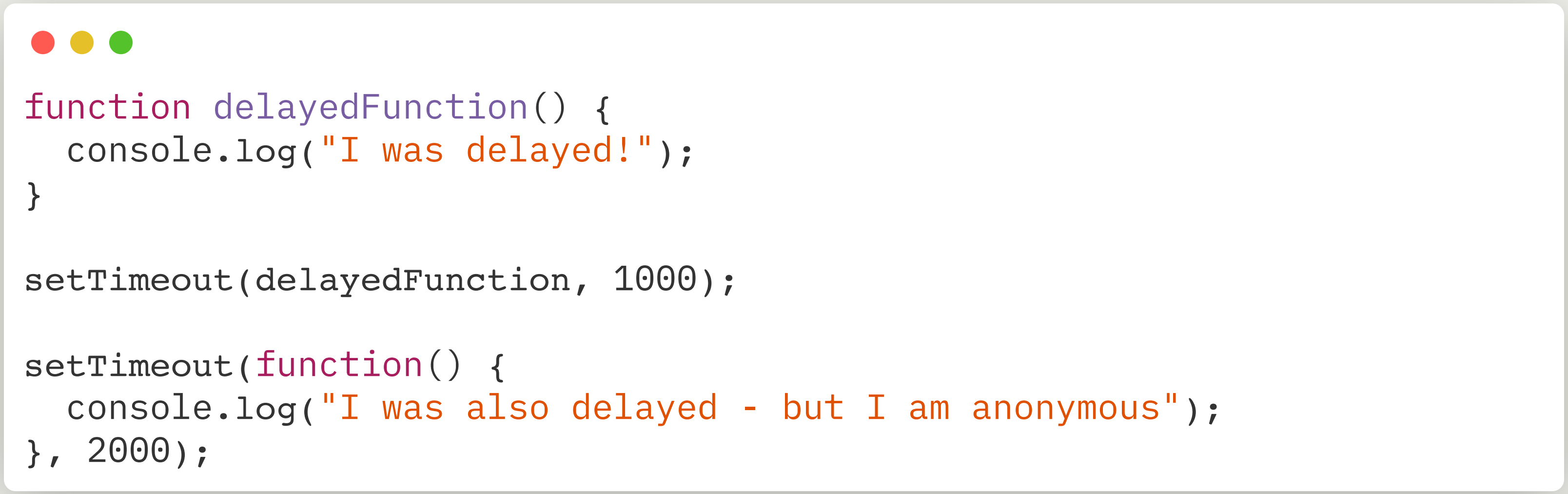
Repeats the execution of a function continuously with an interval in between each call

setTimeout

setTimeout

Occasionally, we don't need to run a function straight away - we want to run it after some time has elapsed.

setTimeout



```
function delayedFunction() {
  console.log("I was delayed!");
}

setTimeout(delayedFunction, 1000);

setTimeout(function() {
  console.log("I was also delayed - but I am anonymous");
}, 2000);
```

`setInterval`

setInterval

```javascript
function regularlyCalledFunction() {
  console.log("I am called regularly");
}

const timer = setInterval(regularlyCalledFunction, 1200);

clearInterval(timer); // At some point, you can cancel the inteval too!

setInterval(function() {
  console.log("I am also called regularly - but I am anonymous");
}, 2000);
```

Scope and Hoisting

What is scope?

- Scope defines everything that you have access to at some point in your code (values, variables, functions etc.)
- Scope is like a pyramid. Lower scopes can access those above them - but not below! Another metaphor is they are like one-way mirrors
- The top level is called the global scope (the complete JavaScript environment)
- Essentially, scoping is name resolution. Where can you access JavaScript identifiers in your code?

Lexical Scoping

- JavaScript uses Lexical Scoping
- Lexical scoping means that scope is defined by the **position in source code**
- In a JavaScript context, there are two main types of scoping:
  - Block Scoping (`let` and `const` use this)
  - Function Scoping (`var` uses this)
- The lexical environment consists of two parts - all local variables and its properties, and a reference to the outer lexical environment

Block Scoping

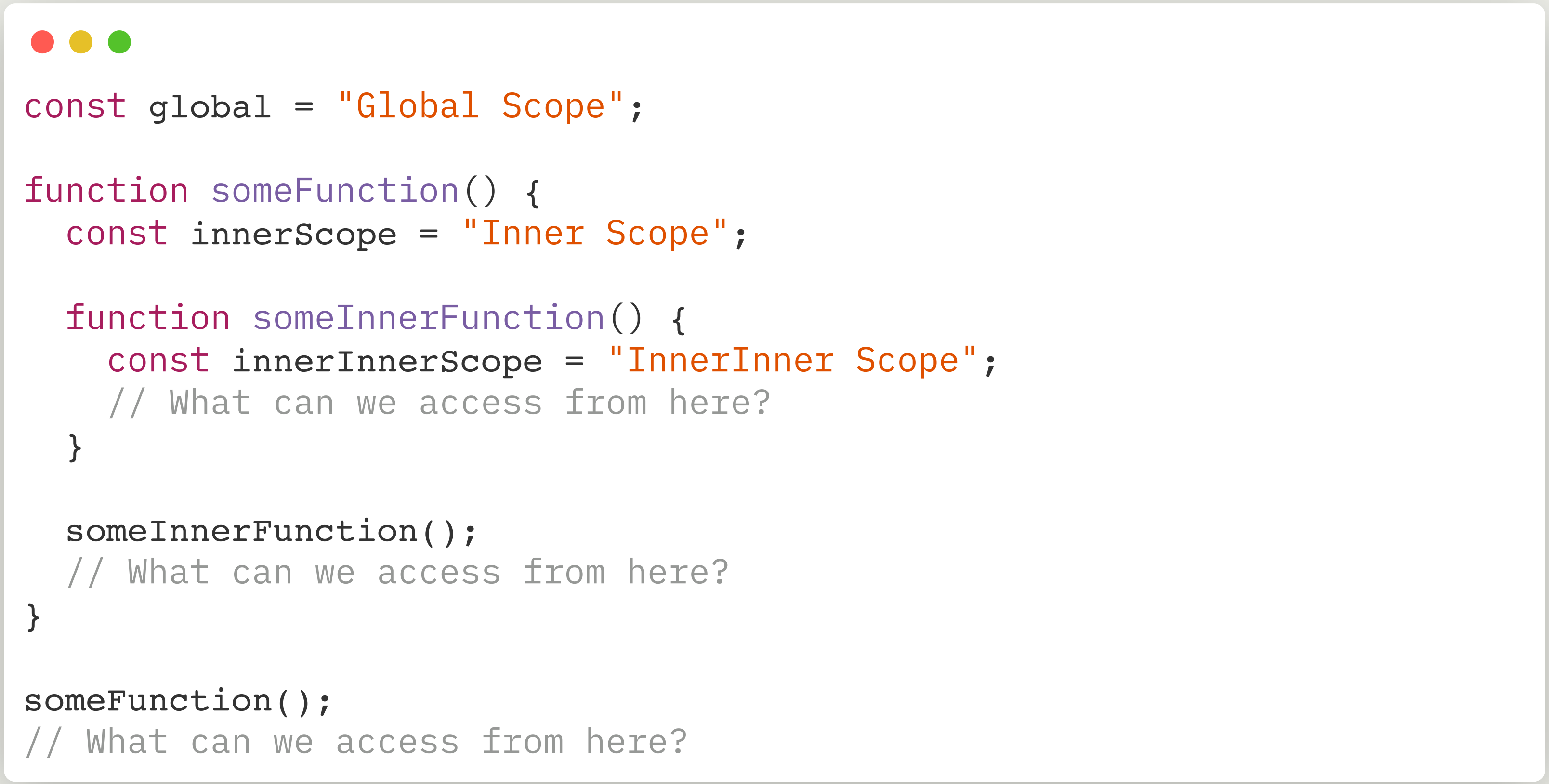
In the most simplest terms, when using block scoping - you can think of it as any curly brackets will create a new lexical environment (any function, any conditional, any loop etc.)

```javascript
const global = "Global Scope";

function myFunction() {
  const local = "Local Scope";
  console.log(global, local);  // Both work!
}

console.log(global); // Works!
console.log(local); // ReferenceError (local is hidden!)
```

Block Scoping

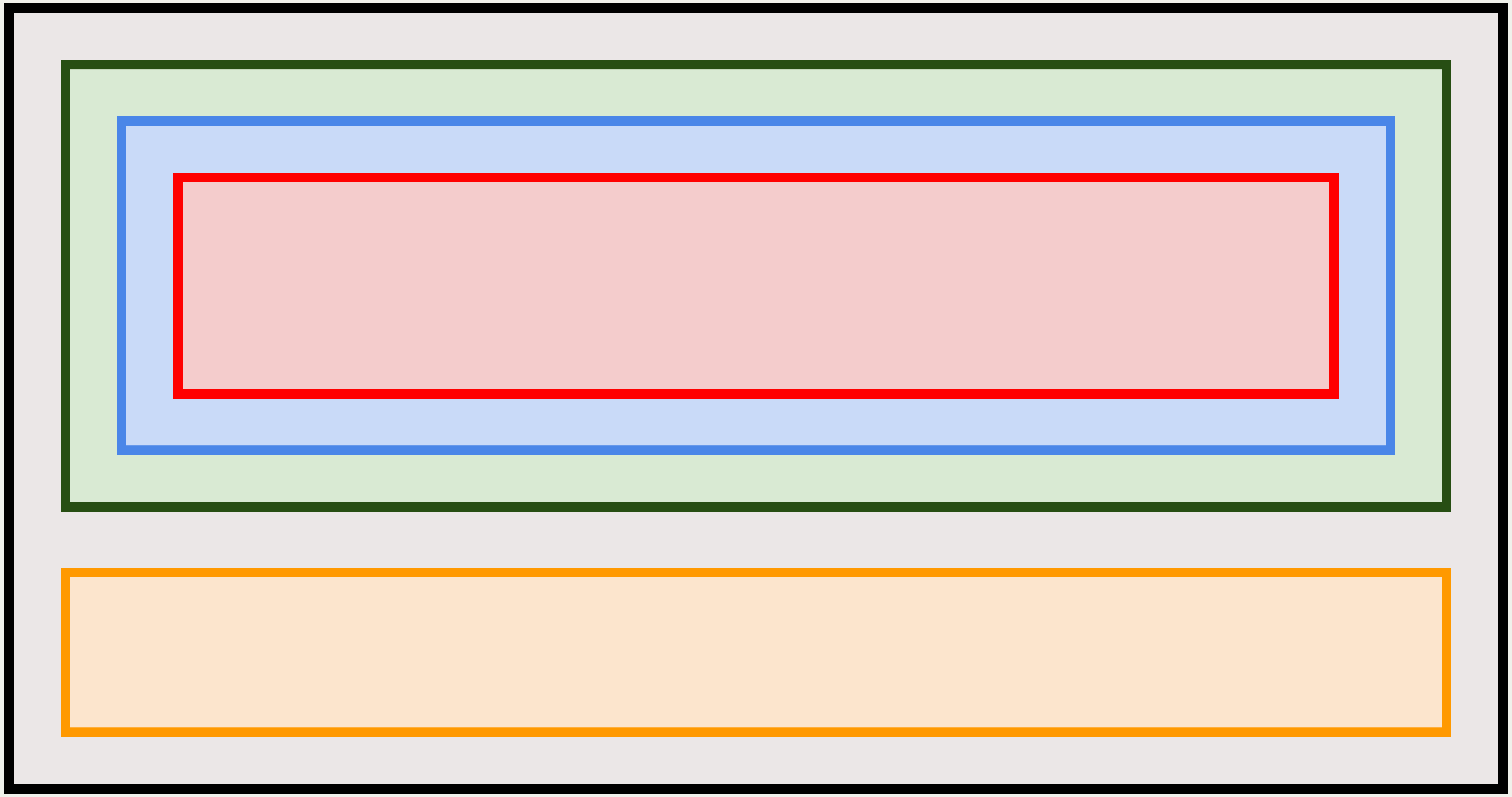
```javascript
const global = "Global Scope";

function myFunction() {
  const local = "Local Scope";

  if (true) {
    const evenMoreLocal = "Even More Local";
  }

  console.log(global, local);  // Both work!
  console.log(evenMoreLocal); // ReferenceError!
}

console.log(global); // Works!
console.log(local); // ReferenceError (local is hidden!)
```

Block Scoping

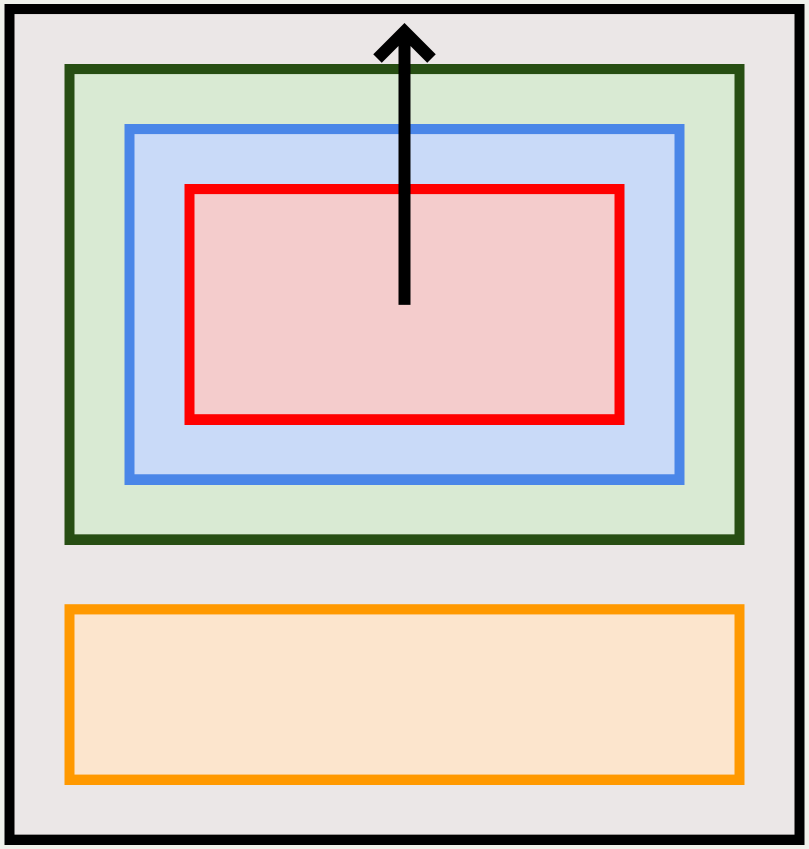


```
const global = "Global Scope";

function someFunction() {
  const innerScope = "Inner Scope";

  function someInnerFunction() {
    const innerInnerScope = "InnerInner Scope";
    // What can we access from here?
  }

  someInnerFunction();
  // What can we access from here?
}

someFunction();
// What can we access from here?
```

Block Scoping



Block Scoping



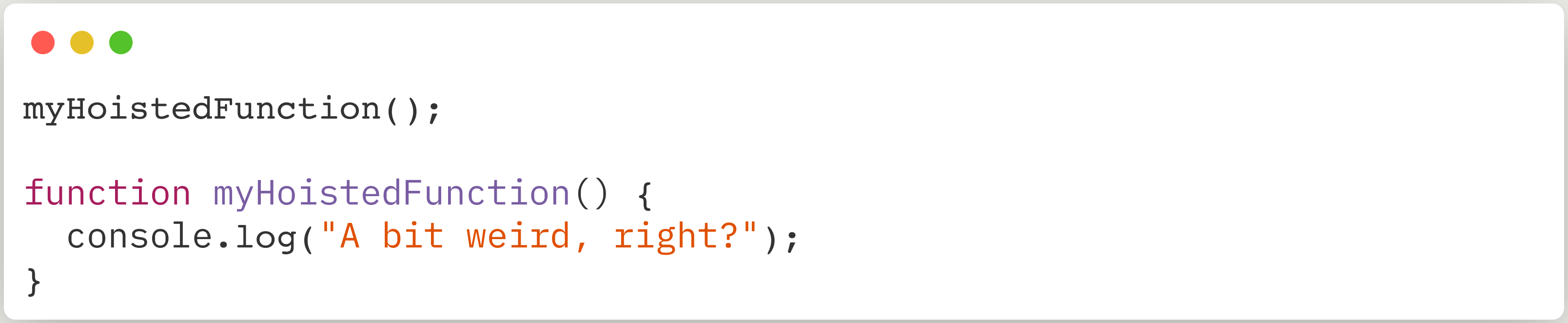
Each of these is a scope.

You can always look out!

What is hoisting?

- One way to think of it is that variable declarations and function declarations get moved to the top of the scope
  - But really, they get put in memory during the *compile phase*
- `var` (but not `let` and `const`) declarations get hoisted too

Hoisting



```javascript
myHoistedFunction();

function myHoistedFunction() {
  console.log("A bit weird, right?");
}
```

Even though this sort of code will work, I don't suggest it! Try to work in order.

Closures

What are closures?

All functions retain the scope of wherever they were defined. A closure is a fancy name for a function that has access to an outer scope's variables.

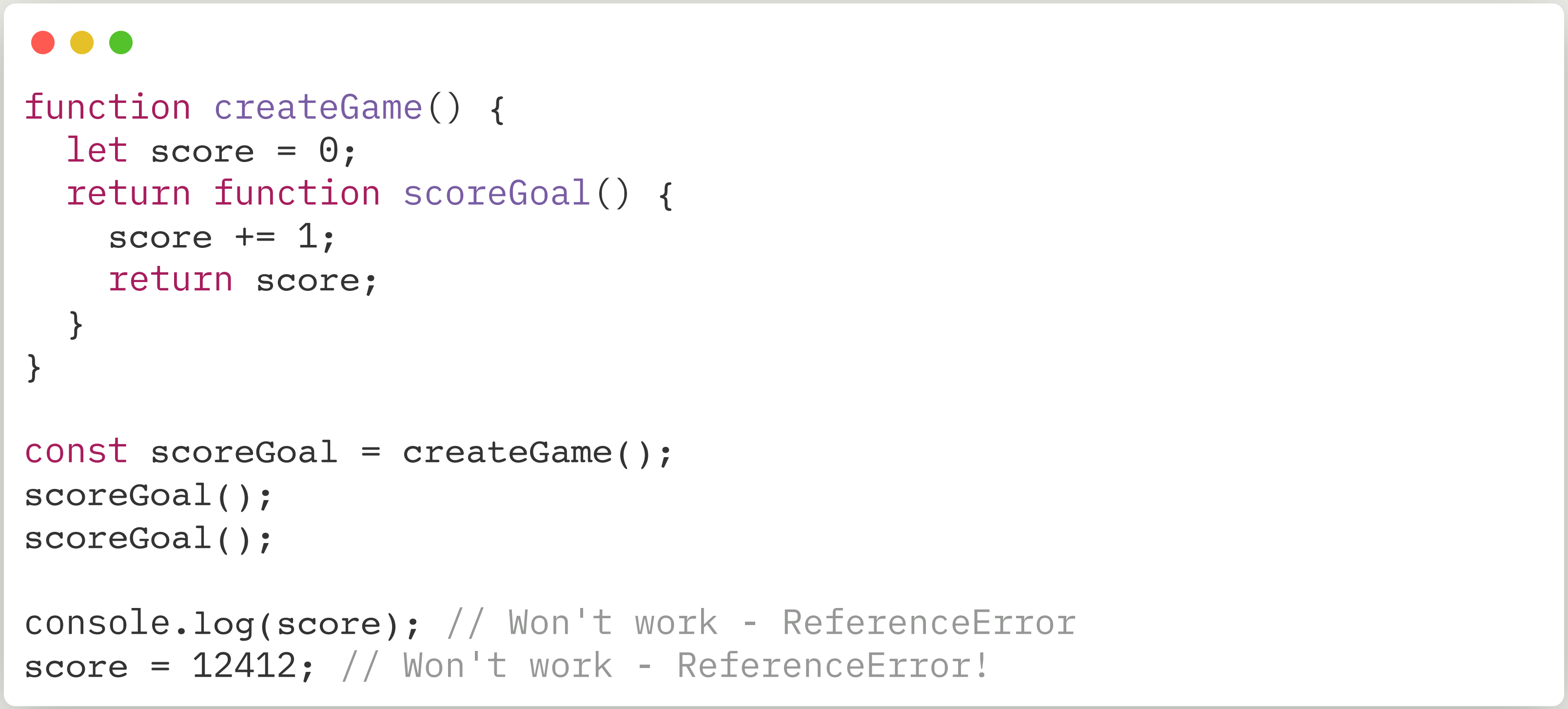
Why would use them?

- Useful for securing your web applications
- You can create private data and functions
- You can create utility functions easily

The Problem

```
let gameScore = 0;

function scoreGoal() {
  gameScore += 1;
}

scoreGoal();
console.log(score);

// You probably wouldn't want this to be possible
score = 10201240;
console.log(score);
```

Closures



```javascript
function createGame() {
  let score = 0;
  return function scoreGoal() {
    score += 1;
    return score;
  }
}

const scoreGoal = createGame();
scoreGoal();
scoreGoal();

console.log(score); // Won't work - ReferenceError
score = 12412; // Won't work - ReferenceError!
```

Closures

```
function createGame() {
  let score = 0;
  return {
    gainPoint: function() {
      return score += 1;
    },
    losePoint: function() {
      return score -= 1;
    },
    getScore: function() {
      console.log(score);
    }
  };
}

const player = createGame();
player.gainPoint();
```

IIFE

IIFE

Immediately Invoked Function Expressions

It is very useful for creating a new scope! Essentially, it is a function that runs straight away.

```
(function () {
    console.log("This runs");
})();


(function (x) {
    console.log("Parameters work too", x);
})(20);
```

Rest Parameters

Rest Parameters - ...

Regardless of the function signature, a function can be called with any amount of arguments (but they will likely be ignored).

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

add(1, 2, 3, 4, 5); // 3
```

Rest Parameters - `...`

Rest Parameters are a way to combine the rest of the parameters (hence the name) into an array.

Rest parameters must be at the end!

```
function add(...nums) {
  let sum = 0;
  for (let i = 0; i < nums.length; i += 1) {
    sum += nums[i];
  }
  return sum;
}

add(1, 2, 3, 4, 5); // 15
```

Spread Operators

Spread Syntax

Occasionally, we need the exactly the opposite of Rest Parameters - sometimes we need to expand an array (spread it out to individual values)

```
Math.max(10, 6, 2);

const myNums = [17, 2, 15, 3];
Math.max(myNums); // NaN - It expects individual items!

Math.max(...myNums); // 17
```

Copying an Array/Object

The spread syntax is one of the things we can copy an array or an object.

```
let nums = ["One", "Two", "Three"];
let myNums = nums; // Points to the same place in memory as `nums`

myOtherNums[0] = "Satu";
console.log(nums, myNums); // What do you think will be printed?

let alphabet = ["A", "B", "C"];
let myAlphabet = [...alphabet]; // Copied!

myAlphabet[0] = "Did this work?";
console.log(alphabet, myAlphabet); // Yes! It worked
```

Higher Order Functions

What are Higher Order Functions?

A higher order function is a function that operates on other functions (either by receiving it as a parameter, or by returning a function).

```
function delay() {
  console.log("Delayed");
}

setTimeout(delay, 1000); // setTimeout is an HOF!
```

Why would you use HOFs?

- Creating utility functions
- Leads to **D.R.Y** code (**D**on't **R**epeat **Y**ourself)
- Creates more declarative programming
  - Declarative is where you describe patterns. Imperative programming is where you describe every single step
- Leads to more maintainable, readable and composable code

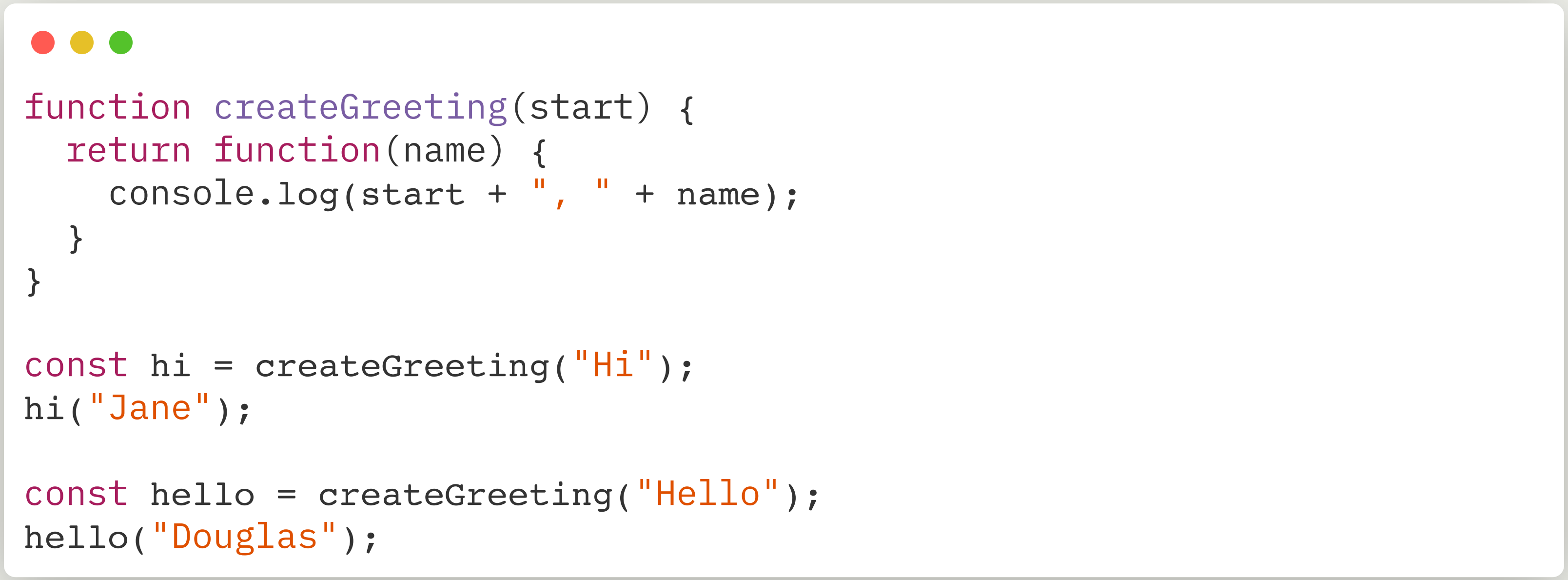
Functions as Input

```javascript
function forEach(arr, callback) {
  for (let i = 0; i < arr.length; i += 1) {
    callback(arr[i], i);
  }
}

function handler(item, index) {
  console.log(item, index);
}

forEach(["one", "two", "three"], handler);

forEach(["one", "two", "three"], function (item, index) {
  console.log(item, index);
});
```

Functions as Output

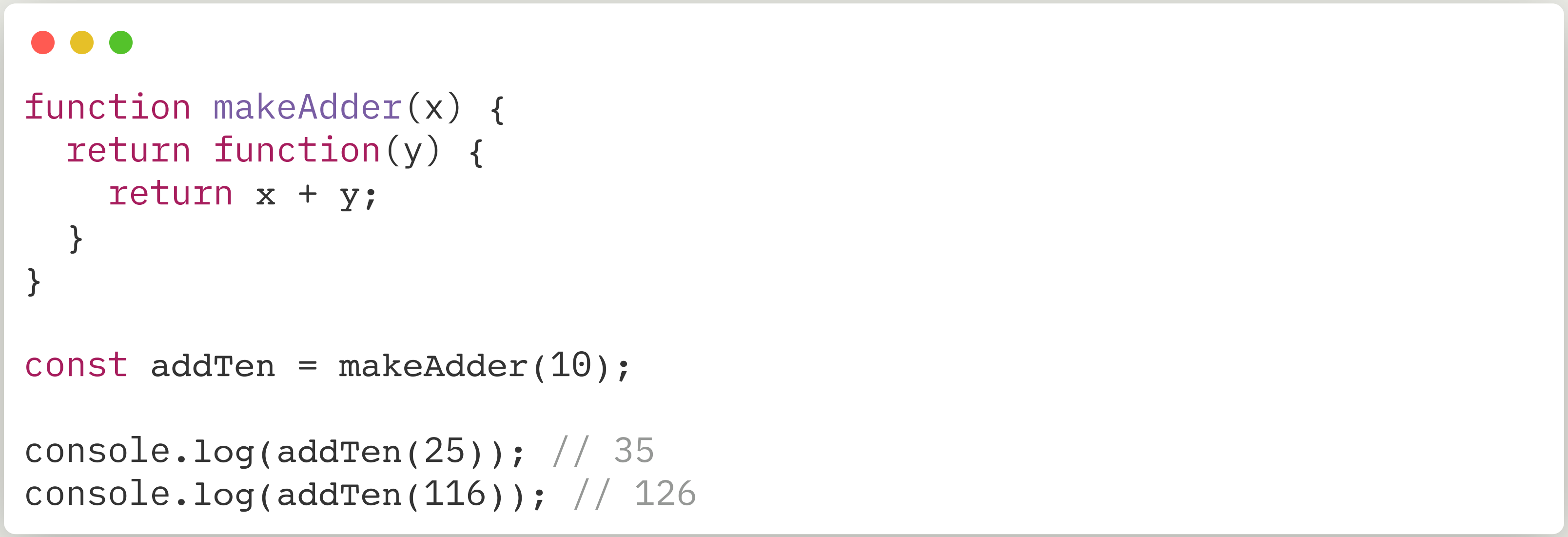
```javascript
function creator() {
  return function () {
    console.log("Returned function");
  }
}

const created = creator();
created();
```

Functions as Output



```
function createGreeting(start) {
  return function(name) {
    console.log(start + ", " + name);
  }
}

const hi = createGreeting("Hi");
hi("Jane");

const hello = createGreeting("Hello");
hello("Douglas");
```

Functions as Output



```javascript
function makeAdder(x) {
  return function(y) {
    return x + y;
  }
}

const addTen = makeAdder(10);

console.log(addTen(25)); // 35
console.log(addTen(116)); // 126
```

That's all!

Homework

- Finish off in-class exercises
    - Pseudocode
- Any previous homework
- (Continue working on in-class content)
- Extra: Begin reviewing the next lesson's content

What's next?

- Methods
- Functional JavaScript
- `this`
- Classes
- Inheritance
    - Classical vs Prototypal

Thank you!