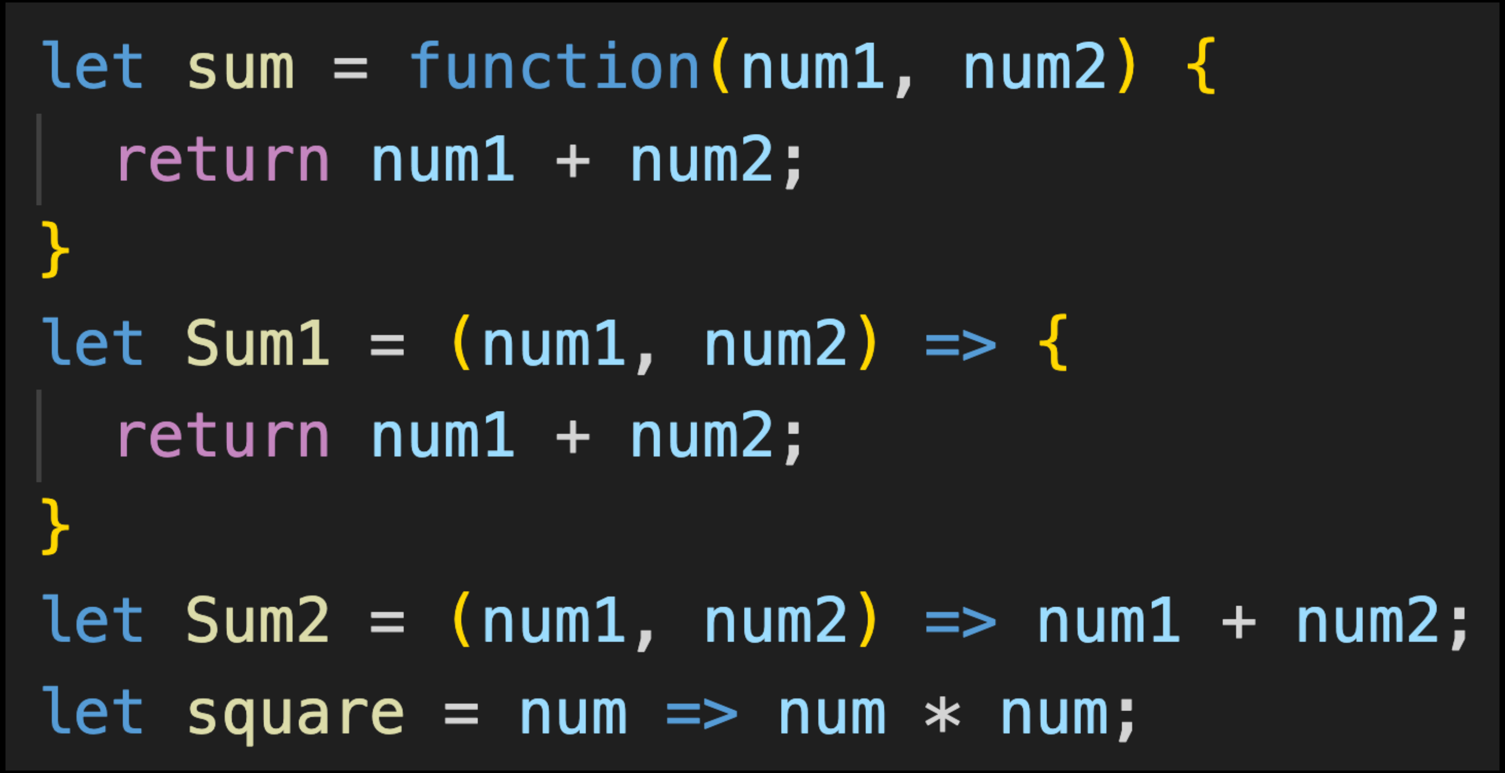
Arrow Functions



```
let sum = function(num1, num2) {
  return num1 + num2;
}
let Sum1 = (num1, num2) => {
  return num1 + num2;
}
let Sum2 = (num1, num2) => num1 + num2;
let square = num => num * num;
```

1. A concise way to write anonymous functions.
2. For Single Argument: Round brackets optional.
3. For Single Line: Curly brackets and return optional.
4. Often used when passing functions as arguments.

Arrow Functions

(Anonymous & Arrow Callbacks)

```
const numbers = [1, 2, 3, 4];

// Callback function without arrow function
function double(num) {
  return num * 2;
}

// Using map with a regular function
const doubled = numbers.map(double);

console.log('Doubled with regular:', doubled);
// Output: [2, 4, 6, 8]

// Using map with an arrow function
const doubledArrow = numbers.map((num) => num * 2);

console.log('Doubled with arrow:', doubledArrow);
// Output: [2, 4, 6, 8]
```

1. Instead of naming the callback function, you can define it directly within the argument list.
2. ES6 arrow functions can also be used as callbacks for a more concise syntax.

Higher-Order Functions

(Function as Argument)

```javascript
const numbers = [1, 2, 3, 4, 5];

// Using map, a higher-order function
const doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6, 8, 10]
```

1. Functions that can take other functions as arguments or return functions as their result.
2. Higher-order functions can accept functions as parameters, allowing you to pass behavior as data.
3. Example: Array.prototype.map(), Array.prototype.filter(), and Array.prototype.reduce() are higher-order functions.

Higher-Order Functions

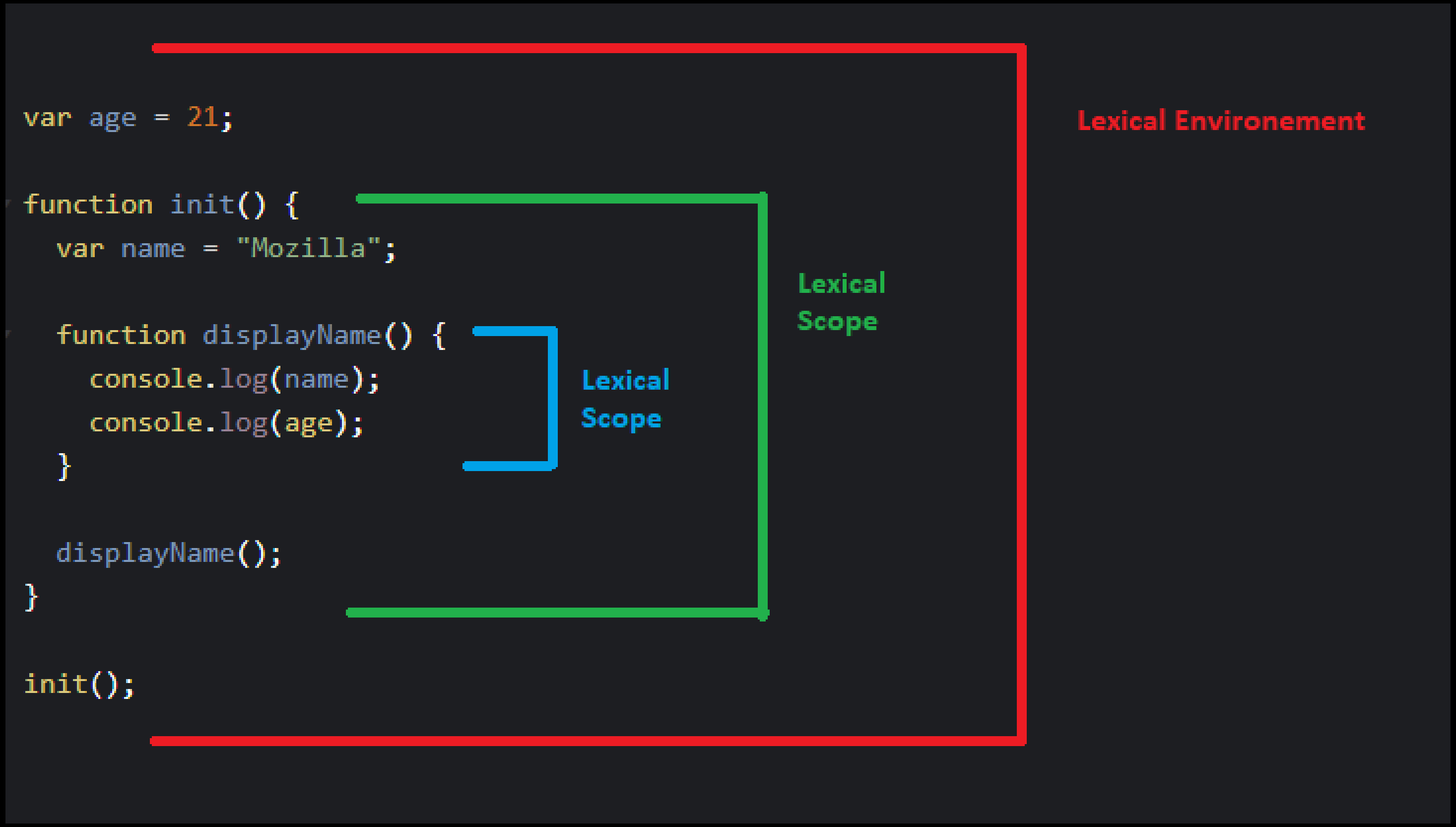
(Return Functions)

```
function createAdder(x) {
  return function(y) {
    return x + y;
  };
}

const addFive = createAdder(5);
console.log(addFive(10)); // 15
```

1. Higher-order functions can return new functions, enabling the creation of function factories or function composition.
2. Higher-order functions allow you to encapsulate behavior and create abstractions for common patterns, making your code more reusable and modular.

Closures (Lexical Scoping)



```
var age = 21;

function init() {
  var name = "Mozilla";

  function displayName() {
    console.log(name);
    console.log(age);
  }

  displayName();
}

init();
```

1. JavaScript uses lexical scoping, which means that the scope of a variable is determined by its position within the source code.
2. Functions can access variables from their own scope, the scope of the parent function, and the global scope.

Closures (Closure Creation)

```javascript
function outerFunction() {
  const outerVariable = 'I am outside!';
  function innerFunction() {
    console.log(outerVariable); // Accesses outerVariable from outerFunction's scope
  }
  return innerFunction;
}

const closureFunction = outerFunction();
closureFunction(); // Output: "I am outside!"
```

1. A closure is created when a function is defined inside another function, and the inner function captures variables from the outer function.
2. The inner function retains access to these variables even after the outer function has finished executing.

Closures (Maintaining State)

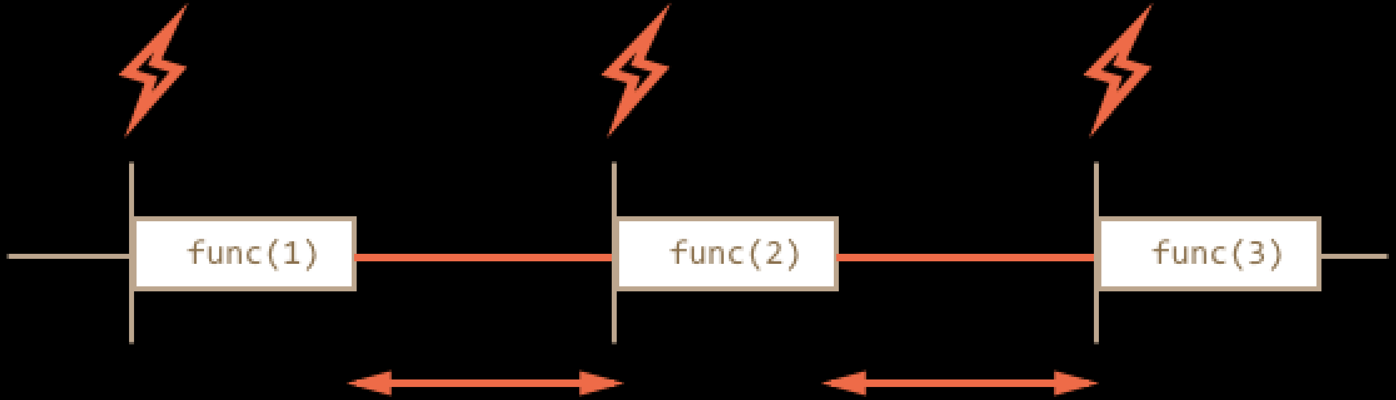
```javascript
function makeCounter() {
  let count = 0; // Private variable

  return function () {
    count += 1;
    return count;
  };
}

const counter = makeCounter();
console.log(counter()); // Output: 1
console.log(counter()); // Output: 2
console.log(counter()); // Output: 3
```

1. makeCounter returns a function that increments the count variable.
2. The count variable is preserved between calls to counter because the inner function forms a closure with it.

setTimeout & setInterval



1. Functions for executing code asynchronously after a delay.
2. setTimeout runs once; setInterval runs repeatedly
3. setTimeout:
   - Syntax: setTimeout(function, time)
   - Cancel: clearTimeout(timerID)
4. setInterval:
   - Syntax: setInterval(function, time)
   - Cancel: clearInterval(intervalID)

setTimeout & setInterval

```javascript
// Example of setTimeout
function greet() {
  console.log("Hello, World!");
}

// Set a timeout to execute the greet function after
// 2 seconds (2000 milliseconds)
setTimeout(greet, 2000);
console.log("This message will display first.");
```

```javascript
// Example of setInterval
function printTime() {
  const now = new Date();
  console.log(`Current time: ${now.toLocaleTimeString()}`);
}

// Set an interval to execute the printTime function
// every second (1000 milliseconds)
const intervalId = setInterval(printTime, 1000);

// Stop the interval after 5 seconds
setTimeout(() => {
  clearInterval(intervalId);
  console.log("Stopped printing the time.");
}, 5000);
```