31. Section Intro

32. Activating Strict Mode

Strict mode is a feature in JavaScript that allows you to opt into a stricter set of rules and behaviors for writing JavaScript code. When strict mode is enabled, the JavaScript interpreter enforces stricter rules and generates more errors for common coding mistakes. It helps you write cleaner and more reliable code by preventing certain types of errors and discouraging the use of problematic language features.

Here are some key characteristics of strict mode:

1. Opting into Strict Mode:

- Strict mode can be enabled for an entire script by placing the following statement at the beginning of the script or within a function: `'use strict';`

- Once enabled, strict mode remains in effect for the entire script or function, and any nested scripts or functions also inherit strict mode.

2. Strict Mode Behavior:

- Strict mode eliminates or modifies some of the silent errors and problematic behavior in JavaScript.

- It prevents the use of undeclared variables by throwing an error when an undeclared variable is accessed.

- It prohibits assignments to undeclared variables, which helps catch potential mistakes and promotes better variable declaration practices.

- It prevents the use of octal literals, which can lead to confusion and unintended behavior.

- It restricts the use of deprecated features, such as `with` statements and duplicate parameter names in function declarations.

3. Benefits of Strict Mode:

- Enhanced debugging: Strict mode helps catch common coding mistakes and errors at an early stage, making it easier to identify and fix issues.

- Safer code: Strict mode prevents certain unsafe or error-prone practices, promoting better coding practices and reducing the likelihood of bugs.

- Better performance: Strict mode allows the JavaScript engine to perform certain optimizations that are not possible in non-strict mode, potentially leading to faster code execution.

4. Compatibility Considerations:

- Strict mode is backward-compatible, meaning it can be safely used with existing JavaScript code.

- However, enabling strict mode may change the behavior of existing code, as it disallows certain practices that were allowed in non-strict mode.

- It is recommended to test code thoroughly when enabling strict mode in an existing codebase to ensure compatibility and identify any issues introduced by the stricter rules.

To enable strict mode for an entire script, you can include the `'use strict';` directive at the beginning of the script:

'use strict';

*// JavaScript code in strict mode*

*// ...*

To enable strict mode for a specific function, you can include the `'use strict';` directive at the beginning of the function:

*function* myFunction() {

  'use strict';

*// JavaScript code in strict mode within the function*

*// ...*

}

By using strict mode, you can catch more errors, write more reliable code, and benefit from a safer and more standardized JavaScript development environment. It's generally considered a good practice to enable strict mode in all your JavaScript code to improve code quality and

33. Functions

In JavaScript, functions are a fundamental building block of the language. They are used to define reusable blocks of code that can be invoked and executed at different points in a program. Functions in JavaScript can have parameters, perform actions, and return values.

Here's an example of a basic function in JavaScript:

*function* greet(*name*) {

  console.log('Hello, ' + name + '!');

}

*// Invoking, running, calling*

greet('John'); *// Output: Hello, John!*

In this example, we have defined a function named `greet` that takes a single parameter `name`. Inside the function, we use the `console.log` statement to output a greeting message with the provided name.

To invoke (call) the function, we simply use its name followed by parentheses `()` and provide the necessary arguments. In this case, we pass the string `'John'` as an argument to the `greet` function.

Functions can also return values using the `return` statement. Here's an example:

*function* add(*a*, *b*) {

  return a + b;

}

*// Invoking the function and storing the returned value*

*var* result = add(3, 5);

console.log(result); *// Output: 8*

In this example, the `add` function takes two parameters `a` and `b` and returns their sum using the `return` statement. We invoke the function with arguments `3` and `5`, and store the returned value in a variable `result`. Finally, we log the value of `result` to the console, which outputs `8`.

Functions in JavaScript can be declared using the `function` keyword as shown in the examples above. However, they can also be defined using function expressions or arrow functions, which provide alternative syntax and behavior.

Function expressions:

*var* multiply = *function* (*a*, *b*) {

  return a \* b;

};

console.log(multiply(2, 4)); *// Output: 8*

Arrow functions:

*var* divide = (*a*, *b*) => a / b;

console.log(divide(10, 2)); *// Output: 5*

These are just basic examples to demonstrate the concept of functions in JavaScript. Functions can be more complex and can include conditional statements, loops, and other JavaScript features to perform a variety of tasks. They are a powerful tool for organizing and structuring code, promoting reusability, and encapsulating logic.

Note: function cannot only reuse a piece of code but if can also receive data and return data back.

Note: we should call it at least once, because if we never called a function then code that’s in the function will never be executed.

34. Function Declarations vs. Expressions

In JavaScript, there are two main ways to define functions: function declarations and function expressions. While they both accomplish the same goal of defining a function, there are some differences in how they are written and behave.

*function* greet(*name*) {

  console.log('Hello, ' + name + '!');

}

greet('John'); *// Output: Hello, John!*

1. Function Declarations:

- Function declarations are created using the `function` keyword followed by the function name and a block of code enclosed in curly braces `{}`.

- They are hoisted to the top of their scope, meaning they can be invoked before they are defined in the code.

- Function declarations can be used both in the global scope and within other functions.

- Here's an example of a function declaration:

2. Function Expressions:

- Function expressions are created by assigning a function to a variable or a constant using the assignment operator `=`. The function is defined as part of an expression.

- They are not hoisted like function declarations and must be defined before they are invoked.

- Function expressions can be named (where the name is optional) or anonymous.

- They are commonly used as callbacks or to create immediately invoked function expressions (IIFE).

- Here's an example of a function expression:

*var* greet = *function*(*name*) {

  console.log('Hello, ' + name + '!');

};

greet('John'); *// Output: Hello, John!*

In terms of behavior, both function declarations and function expressions can be invoked, take arguments, and have a body of code. However, the key differences lie in their hoisting behavior and the way they are defined and assigned.

In general, function declarations are often preferred when you want the function to be accessible throughout the entire scope, including before its actual declaration in the code. Function expressions, on the other hand, provide more flexibility in terms of assigning functions to variables and can be useful in scenarios where you need to create functions dynamically or use them as first-class objects.

It's worth noting that with the introduction of arrow functions in ES6, there is now a third way to define functions in JavaScript. Arrow functions have a more concise syntax and different behavior regarding the `this` keyword. They are often used in scenarios where you need to preserve the lexical context of `this` or create shorter, inline functions.

It's important to choose the appropriate method of defining functions based on your specific use case and the desired behavior in your code.

35. Arrow Functions

Arrow functions, introduced in ECMAScript 6 (ES6), provide a more concise syntax for writing functions in JavaScript. They are commonly used as alternatives to traditional function expressions, offering some benefits and differences in behavior. Here's an overview of arrow functions:

1. Syntax:

- Arrow functions are defined using the arrow (`=>`) syntax, hence the name "arrow functions".

- The basic syntax consists of the parameters (if any) followed by the arrow `=>` and the function body.

- If there's only one parameter, the parentheses around the parameter can be omitted. However, parentheses are required if there are no parameters or if there are multiple parameters.

- If the function body is a single expression, it can be written without curly braces `{}`. This implicitly returns the value of the expression.

- Here are a few examples of arrow function syntax:

*// Single parameter*

*const* square = (*num*) => num \* num;

*// Multiple parameters*

*const* add = (*a*, *b*) => a + b;

*// No parameters*

*const* sayHello = () => console.log("Hello!");

*// Function body with multiple statements*

*const* multiply = (*a*, *b*) => {

*const* result = a \* b;

  return result;

};

2. Lexical `this` Binding:

- Arrow functions do not have their own `this` value. Instead, they inherit the `this` value from the surrounding context (lexical scope).

- In traditional functions, the `this` value is dynamically determined based on how the function is called, which can lead to unexpected behavior and complexities. Arrow functions provide a more predictable and convenient way to access the `this` value.

- Arrow functions are especially useful when working with callbacks or when you need to preserve the value of `this` from the enclosing scope.

*// Example using arrow function and traditional function*

*const* person = {

  name: "John",

  sayHello: *function* () {

*// Using arrow function*

    setTimeout(() => {

      console.log("Hello, " + *this*.name); *// "this" refers to the person object*

    }, 1000);

*// Using traditional function*

    setTimeout(*function* () {

      console.log("Hello, " + *this*.name); *// "this" refers to the global object (or undefined in strict mode)*

    }, 1000);

  },

};

person.sayHello(); *// Output: Hello, John (from arrow function)*

*//         Hello, undefined (from traditional function)*

3. No `arguments` Object:

- Arrow functions do not have their own `arguments` object. If you need to access the arguments passed to an arrow function, you can use the rest parameters syntax (`...args`).

*const* sum = (...*args*) => {

*let* total = 0;

  for (*let* num of args) {

    total += num;

  }

  return total;

};

console.log(sum(1, 2, 3, 4)); *// Output: 10*

4. No `new` Binding:

- Arrow functions cannot be used as constructors and cannot be invoked with the `new` keyword. They lack the internal `[[Construct]]` method, which is necessary for creating new object instances.

- Attempting to use an arrow function with `new` will result in a runtime error.

36. Functions Calling Other Functions

In JavaScript, functions can call other functions to perform a specific task or to delegate certain operations. Calling one function from another allows for modular and reusable code, as well as better organization and separation of concerns. Here's how functions can call other functions in JavaScript:

1. Function Invocation:

- The simplest way to call a function from another function is by directly invoking it using its name followed by parentheses `( )`.

- When a function is invoked, the code inside that function is executed, and it may return a value or perform some actions.

- Here's an example of one function calling another function:

*function* greet(*name*) {

  console.log('Hello, ' + name + '!');

}

*function* greetUser() {

  greet('John'); *// Call the greet function from within greetUser*

}

greetUser(); *// Output: Hello, John!*

In this example, the `greetUser` function calls the `greet` function, passing the name `'John'` as an argument. The `greet` function is invoked and prints the greeting message to the console.

2. Returning Values from Functions:

- Functions can also return values, which can be used as inputs for other functions or stored in variables for further processing.

- By using the `return` statement, a function can specify the value it wants to pass back to the calling function.

- Here's an example:

*function* multiply(*a*, *b*) {

  return a \* b;

}

*function* calculate() {

*var* result = multiply(3, 4); *// Call the multiply function and assign the result to a variable*

  console.log(result); *// Output: 12*

}

calculate();

In this example, the `calculate` function calls the `multiply` function and assigns the returned value (`12`) to the `result` variable. The `result` is then printed to the console.

3. Function Chaining:

- Function chaining is a technique where multiple functions are called in sequence, with each function being called on the return value of the previous function.

- This approach is commonly used with methods that modify or transform data, allowing for concise and readable code.

- Here's an example using array methods in JavaScript:

*var* numbers = [1, 2, 3, 4, 5];

*var* sum = numbers.reduce((*acc*, *curr*) => acc + curr) *// Calculate the sum of numbers*

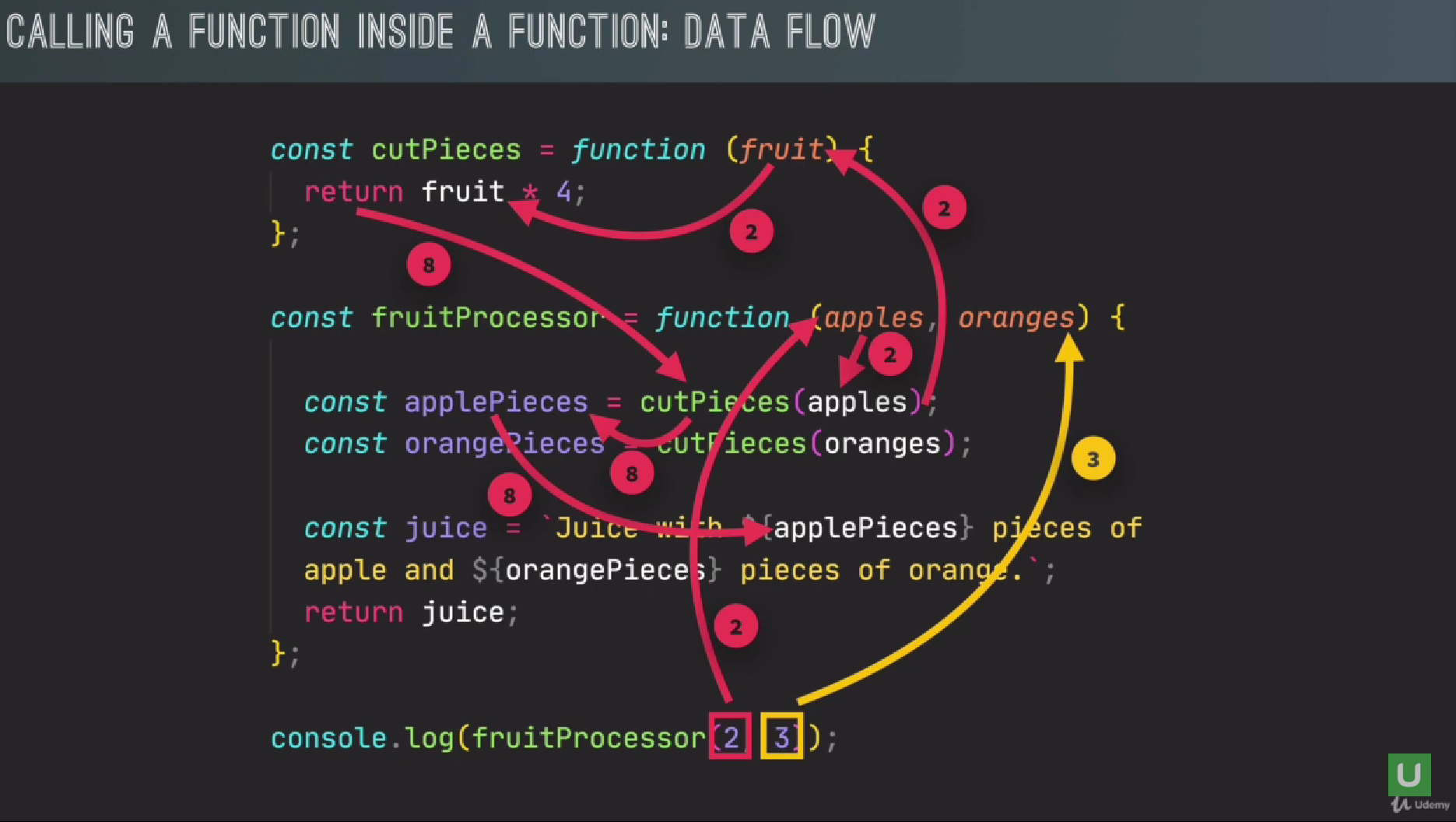
                    .toFixed(2); *// Convert the sum to a fixed decimal precision*

   console.log(sum); *// Output: 15.00*

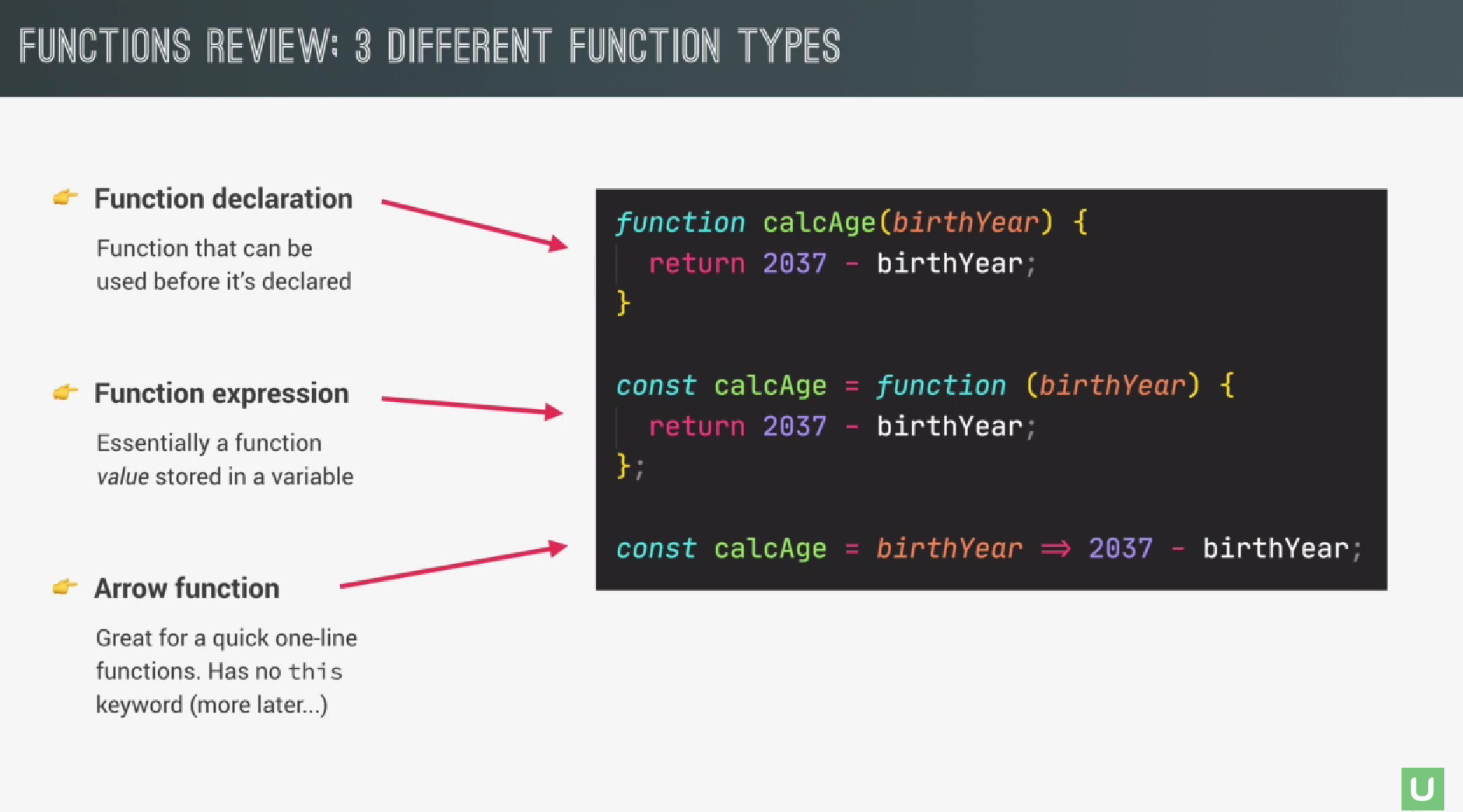
In this example, the `reduce` method is called on the `numbers` array to calculate the sum. The resulting sum is then passed to the `to Fixed` method to round it to two decimal places.

By calling functions from other functions, you can create a hierarchy of operations, reuse code, and make your code more modular and maintainable. This enables you to break down complex tasks into smaller, manageable functions, promoting code reusability and separation of concerns.

Example:

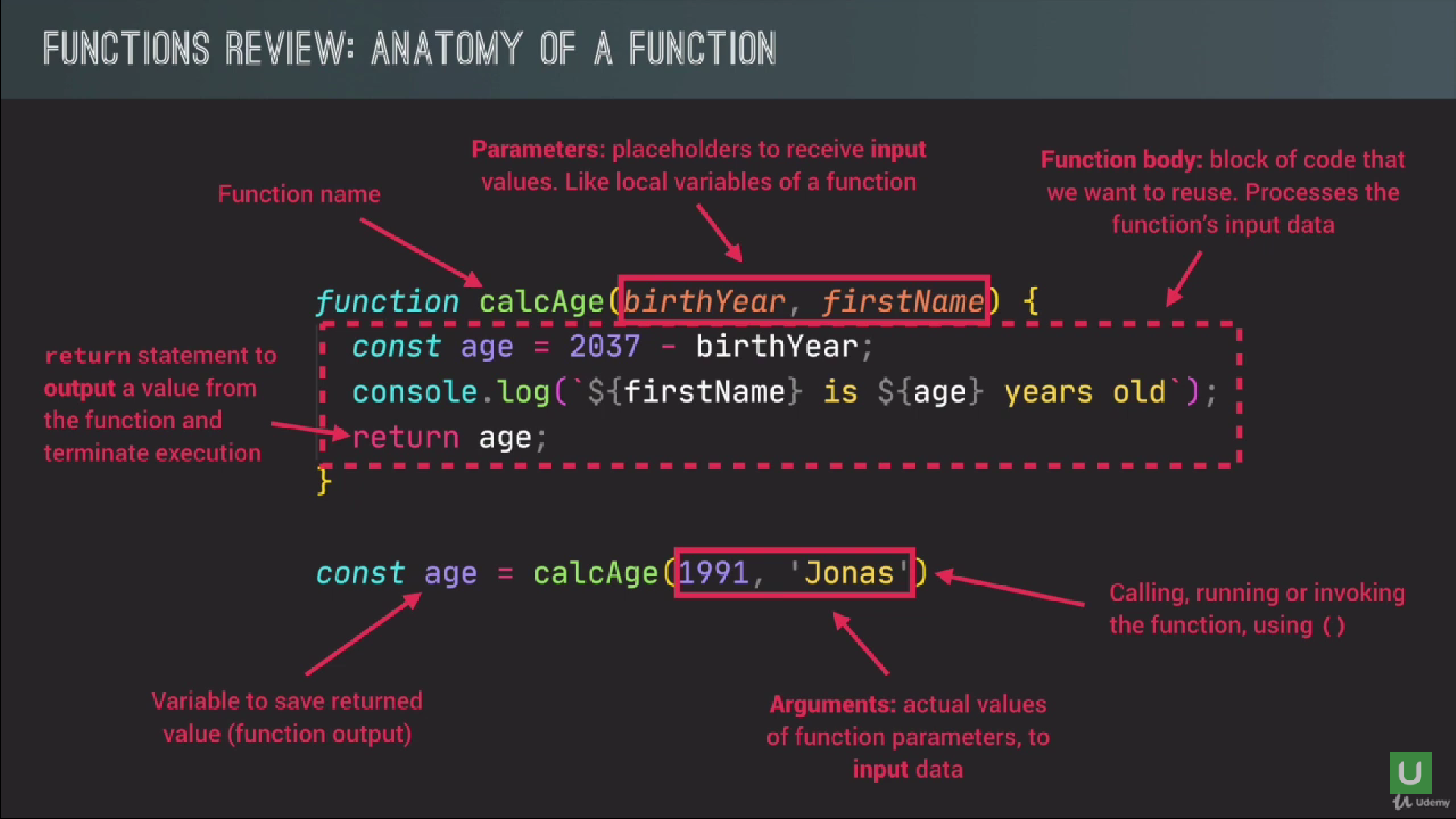


Example:



Note: Thes are three difference way to writing function. But all work in a similar way. So, all of them receive input data, transform data, and they can output data. It’s all optional but usually that’s the things that function do.

37. Reviewing Functions



38. Coding Exercise 5: CHALLENGE

39. Introduction to Arrays

In JavaScript, an array is a data structure that allows you to store and manipulate multiple values in a single variable. Arrays are indexed collections of elements, where each element is identified by its position, or index, within the array. Here are some key points about arrays in JavaScript:

1. Array Syntax:

- Arrays are defined using square brackets `[]`.

- Elements within an array are separated by commas `,`.

- An array can contain values of any data type, including numbers, strings, objects, functions, or even other arrays.

2. Array Indexing:

- The elements in an array are accessed using zero-based indexing.

- You can access individual elements by referring to their index within square brackets `[]`.

- The first element of the array has an index of 0, the second element has an index of 1, and so on.

3. Array Length:

- Arrays have a `length` property that indicates the number of elements in the array.

- The `length` property is automatically updated as elements are added or removed from the array.

4. Common Array Methods:

- JavaScript provides various built-in methods for working with arrays, such as `push`, `pop`, `shift`, `unshift`, `slice`, `splice`, `concat`, `join`, `indexOf`, `forEach`, `map`, `filter`, `reduce`, and many more.

- These methods allow you to add, remove, modify, or transform array elements efficiently.

5. Iterating Over Arrays:

- You can iterate over the elements of an array using loops like `for`, `while`, or `for...of`.

- The `forEach` method and other array iteration methods provide a more convenient way to iterate over arrays.

Here's an example to illustrate the basic usage of arrays:

*var* fruits = ['apple', 'banana', 'orange'];

console.log(fruits[0]); *// Output: 'apple'*

console.log(fruits.length); *// Output: 3*

fruits.push('grape'); *// Add an element to the end of the array*

console.log(fruits); *// Output: ['apple', 'banana', 'orange', 'grape']*

fruits.pop(); *// Remove the last element from the array*

console.log(fruits); *// Output: ['apple', 'banana', 'orange']*

fruits.forEach(*function*(*fruit*) {

  console.log(fruit);

});

*// Output:*

*// 'apple'*

*// 'banana'*

*// 'orange'*

Arrays are versatile and widely used in JavaScript for storing collections of data, iterating over elements, performing operations, and organizing related values. They provide a convenient way to work with multiple values as a cohesive unit.

Note: only primitive values, are immutable. But an array is not primitive value.

40. Basic Array Operations (Methods)

JavaScript provides a variety of built-in methods for working with arrays. These methods allow you to add, remove, modify, and transform array elements efficiently. Here are some commonly used array methods:

1. Adding and Removing Elements:

- `push(element1, element2, ...)`: Adds one or more elements to the end of the array and returns the new length of the array.

- `pop()`: Removes the last element from the array and returns that element.

- `unshift(element1, element2, ...)`: Adds one or more elements to the beginning of the array and returns the new length of the array.

- `shift()`: Removes the first element from the array and returns that element.

2. Modifying and Replacing Elements:

- `splice(startIndex, deleteCount, element1, element2, ...)`: Removes or replaces existing elements and/or adds new elements to the array.

- `fill(value, start, end)`: Fills the array with a static value between the specified start and end indexes.

3. Accessing Elements:

- `concat(array1, array2, ...)`: Returns a new array that combines the current array with one or more other arrays or values.

- `slice(startIndex, endIndex)`: Returns a shallow copy of a portion of the array into a new array.

4. Searching and Filtering:

- `indexOf(searchElement, startIndex)`: Returns the first index at which a given element is found in the array, or -1 if not found.

- `lastIndexOf(searchElement, startIndex)`: Returns the last index at which a given element is found in the array, or -1 if not found.

- `find(callback)`: Returns the first element in the array that satisfies the provided testing function.

- `filter(callback)`: Creates a new array with all elements that pass the provided testing function.

5. Iterating Over Elements:

- `forEach(callback)`: Executes a provided function once for each array element.

- `map(callback)`: Creates a new array by calling a provided function on each element in the array.

- `reduce(callback, initialValue)`: Applies a function to an accumulator and each element in the array, reducing it to a single value.

- `some(callback)`: Checks if at least one element in the array satisfies the provided testing function.

- `every(callback)`: Checks if all elements in the array satisfy the provided testing function.

These are just a few examples of the many array methods available in JavaScript. Each method serves a specific purpose and can be used to manipulate, iterate over, or transform arrays in different ways. By combining and using these methods effectively, you can perform complex operations on arrays with ease.

41. Coding Exercise 6: CHALLENGE #2

Arrow

42. Introduction to Objects

Arrow

43. Dot vs. Bracket Notation

Arrow

44. Object Methods

Arrow

45. Coding Exercise 7: CHALLENGE #3

Arrow

46. Iteration: The for Loop

Arrow

47. Looping Arrays, Breaking and Continuing

Arrow

48. Looping Backwards and Loops in Loops

Arrow

49. The while Loop

Arrow

50. CHALLENGE #4:

Arrow