WEEK 4 - PART 1

DEEP DIVE INTO CODE -

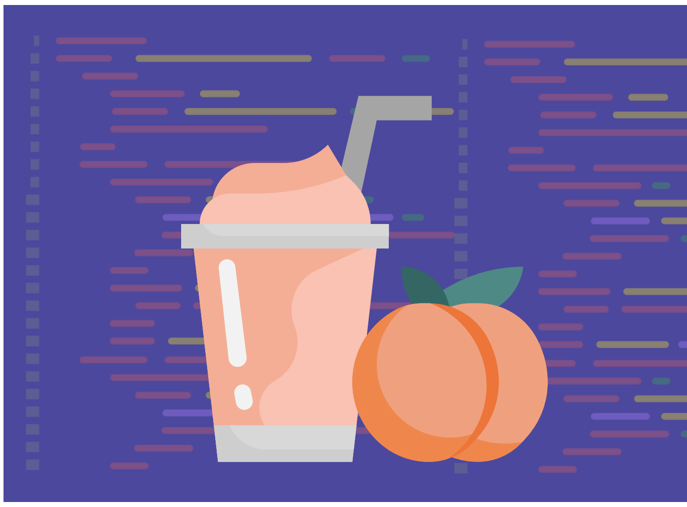
FUNCTIONS

LINNAEUS UNIVERSITY
SUMMER 2024

TABLE OF CONTENTS

WHAT IS A FUNCTION?

In programming, **a function is a set of operations that takes one or more inputs (parameters) and produces a result - performing operations on inputs to produce outputs.**
Functions group instructions together, and **parameters - input** - enable dynamic ways of performing those instructions with **varying data** that is used. In programming, functions allow you to group a series of instructions and reuse them with different inputs.



Imagine you make a smoothie every morning. The recipe is flexible, so you can change the ingredients based on what you have or what you feel like that day (for example, instead specifying that you have a banana everyday in your smoothie, just just say "fruit", encompassing any fruit you chose).

In this scenario, the function is like the process of making the smoothie. The input is the ingredients you decide to use, and the output is the smoothie itself.

Let's define our "smoothie-making" function as:

makeSmoothie(fruit, liquid, sweetener)

Parameter 1: fruit - The type of fruit you add (e.g., "banana", "strawberry").
Parameter 2: liquid - The liquid base you use (e.g., "milk", "orange juice").
Parameter 3: sweetener - The sweetener you choose (e.g., "honey", "sugar").

The important part here, is that you can **switch out** "fruit", "liquid" and "sweetener", but **still perform the same instructions and yield a result.** Unless you input the same details, each smoothie will be different, while still keeping to the basic instruction given - the "formula" and still performing the same instructions.

HOW FUNCTIONS WORK

The function makeSmoothie(fruit, liquid, sweetener) will blend together whatever ingredients you choose:

Scenario 1

- Input: You choose a banana, milk, and honey.
- Process: The function blends these ingredients.
- Output: You get a banana smoothie with milk and honey.

Scenario 2

- Input: You choose strawberries, orange juice, and no sweetener.
- Process: The function blends these ingredients.
- Output: You get a strawberry smoothie with orange juice.

Scenario 3

- Input: You choose mango, coconut water, and sugar.
- Process: The function blends these ingredients.
- Output: You get a mango smoothie with coconut water and sugar.

If you put in different fruits, liquids, or sweeteners, you get a different smoothie. BUT, the smoothie function repeats the same process (blending), just with different ingredients.

The result (the smoothie) changes because the inputs are different, but the instructions remain the same. This allows use to reuse the instructions while being able to switch out the inputs.

FUNCTIONS IN MATH & CODING

In math a function is a set of operations (instructions) that takes one or more inputs (parameters) and produces a result.
In a math function like
f(x) = 2x + 3
the 'x' serves as a parameter, allowing the formula to be applied to various input values.
The formula takes an input value 'x', multiplies it by 2, and then adds 3 to the result. For instance, if we substitute x = 5 into the function, we get
f(5) = 2 * 5 + 3 = 13
If we substitute x = 10 into the function, we get
f(10) = 2 * 10 + 3 = 23
Here, 'x' serves as the parameter of the mathematical function, allowing us to generalize the formula for various values of 'x'.

In programming, functions follow the same blueprint -
functions group instructions together, and parameters enable dynamic input. For example, a JavaScript function
calculateF(x)
takes an input parameter 'x', and produces a result.
Just like in the smoothie example, you can have many parameters in a function.

makeSmoothie(fruit, liquid, sweetener)

KEYWORDS IN CODE

In programming, **keywords are reserved words that have a predefined meanings** in the language's syntax, and are used to **perform specific functions or to define the structure of the code.**

Keywords are reserved by the programming language - you cannot use these words as identifiers (names for variables, functions, etc.). They help in defining the rules for writing code, such as how to create functions or declare variables. Different programming languages have different sets of keywords. For example, JavaScript, Python, and Java all have their own sets of keywords that serve similar or different purposes.

COMPONENTS OF A FUNCTION

STRUCTURE & DECLARING A FUNCTION

Functions need to be declared - this involves defining its structure, logic, and parameters.

When declaring a function, no code is run - you as the programmer are providing the instructions as a “reference” for the computer. The declaration can be compared to the **smoothie recipe** - no smoothie is made, but the instructions for how to make the smoothie are provided.

```
function functionName(parameters) {
        // Code to be executed
}
```

The **function keyword** is used to **create a function**, which is a **block of reusable code that performs a specific task**. The keyword tells JavaScript you’re **declaring a function** - you start with the function keyword, followed by the **name of the function**, a set of **parentheses ()**, and a set of **curly braces {}** that marks the end of the function’s code.

The function must have a **name** - you as the programmer name this yourself, something that describes what the code does. To continue the smoothie analogy - this would be the name of the smoothie.

The **computer has to know when the start and end of the instructions is.** Different programming languages might do this differently, but the most common is to use **{ and } to define the start and end of a function.**

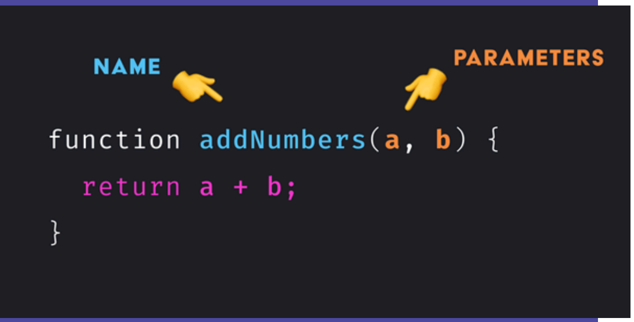
Python uses indentations - a set amount of spaces - to define when a function begins and ends.

In JavaScript, **the curly braces {} define the scope** of the function - meaning, **when the function ends and begins.**
In our smoothie metaphor, this could be the page it is printed on - the end and beginning of the instructions.

Within these braces {} is the set of instructions that make up the function's logic. In essence, here is where the step by step recipe to make our smoothie is written - the instructions, what order to perform then in, and how to use each ingredient. Within these braces, you can manipulate the **parameters (the input - our recipe ingredients)** and perform operations (instructions - blend, smash, juice, pour) to produce the desired result (delicious smoothie).

A function can have parameters - the **swappable place holder "fruit", "liquid" and "sweetener" are parameters, housed between the parenthesis ().**

These are like empty placeholders or labels that represent the data that will be sent in to the function - or, as programmers commonly say, ***passed into the function.***



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

Function don't HAVE to have parameters - it is is strictly based on the use-case and what the function is meant to do.
Here we use parameters to demonstrate how they work.

If there are **no parameters** in the function declaration, you simply **don't need to provide any input** for the function to run.

Example
f(x) = 8 + 3

The result will be the same, no matter what you set as X.

Parameters act as placeholders for the actual ingredients that you will provide later when you use the function. These placeholders help the function know what kind of data it will work with, but they aren't actual data themselves. When we define the function, we don't know exactly what the values will be yet, so we use these placeholder labels.

When you actually **use the function, you replace these placeholders with the actual, real values** that are **relevant for the specific situation**—like the specific ingredients for that particular smoothie.
To explain further, if a function had the parameters (clothes, shoes, jewelry), you might pass in (pants, boots, ring).
These specific, "real" values that you provide when using the function are called arguments.
Inside the function, you can do things like declare new variables, perform if statements, and carry out various operations to process the arguments and produce the desired output.

Variables created within the function does not exist outside the function. If you wrote

let proteinpowder = "chocolate protein powder"

then the variable proteinpowder would not be accessible to any code outside of the **{}**.

When a variable is declared **inside a function**, it is considered a **local variable.** This means that it can **only be accessed within that function.** Once the function finishes executing, the local variable is no longer available, often being described as the variable being "alive" only during the execution of the function.

This helps prevent accidental interference with variables outside of the function and keeps the function self-contained.

Remember, the declaration of a variable does not actually run any code– it simply provides a blueprint, or recipe for how to execute instructions
Asking the computer to execute the function is called calling a function.

CALLING A FUNCTION

When you define a function, it doesn't do anything on its own until it's *called* by some other part of the program.

"The function's caller" refers to the part of the code that instructs the function to run. **Calling a function is like asking the computer to perform the specific task that you've defined beforehand in the function declaration.** When you call a function, you're telling the program to execute the set of instructions inside that function.

To call a function, you use its name followed by parentheses (). The parentheses can also include arguments if the function requires them.

```
makeSmoothie(mango, milk, sugar)
```

For example, if you have a function called **greetUser**, which is designed to print a greeting, you would call it by simply writing its name followed by parentheses like this: **greetUser()**.
This action tells the program to jump to that function declaration, no matter where in the code it is called and **run the code inside the declaration.**

Once the computer is done running the code inside the function, the computer **returns to where it left off and continues to execute the code sequentially.**

When writing the function call, **any parameters specified in the function declaration need to be substituted by arguments** - the actual values that you pass into a function when you call it.

These **arguments correspond to the function's parameters -** the placeholders defined when the function is created. When you call the function and provide arguments, those values are assigned to the corresponding parameters, and the function uses them to perform its operations. Every time the function is called, the a**rguments get swapped in for any parameters written INSIDE the function.** Whenever the function is called, the specific arguments are used in the function.

FUNCTION DECLERATION

```
function makeSmoothie(fruit, liquid, sweetener) {
Blend fruit, liquid, sweetener together.
}
```

FUNCTION CALL

```
makeSmoothie(mango, coconutmilk, sugar)
//  Swaps out fruit for mango, coconutmilk for liquid and
sugar for sweetener.
makeSmoothie(strawberry, milk, sugar)
// Swaps out fruit for strawberry, milk for liquid and sugar
for sweetener.
```

When we say you can **manipulate parameters, we mean that you can use them in calculations, modify them, or apply logic to them** to achieve the desired outcome.
Parameters can be used directly in mathematical operations.

For instance, if a function takes width and height as parameters, you can use them to calculate the area of a rectangle.

```
function exampleFunction(param1, param2) {
console.log(param1, param2)
}

exampleFunction(10, 20) // Outputs: 10 20
```

You can use parameters to make decisions or apply conditions. For example, you might want to check if the quantity of items exceeds a certain number and apply a discount accordingly. Inside the function, you can create new variables based on the parameters and use these new variables in further operations.

```
function calculateRectangleArea(length, width) {
// Create new variables based on the parameters
// Calculate the area using length and width
let area = length * width;

// Calculate the perimeter
let perimeter = 2 * (length + width);


// Use these variables
console.log("Area:", area);
console.log("Perimeter:", perimeter)
}

// Output:// Area: 15// Perimeter: 16
```

RETURNS

Imagine you want to create a function that calculates the total cost of a smoothie based on the cost of its ingredients.

Here's a function named **calculateSmoothieCost** that takes **three parameters: fruitCost, liquidCost, and sweetenerCost.** This function calculates the total cost by summing up these costs.

```
function calculateSmoothieCost(fruitCost, liquidCost, sweetenerCost) {
 // Calculate the total cost by summing up the costs
let totalCost = fruitCost + liquidCost + sweetenerCost
return  totaltCost
}
```

Inside the braces **{}**, we use the **parameters fruitCost, liquidCost, and sweetenerCost** to calculate the total cost of the smoothie. The **totalCost variable** is created to **store the result of this addition.**

When you write a function in programming, you often want it to **perform a task and then give back a result. The return statement is a crucial part of this process.** It's like the function's way of **sending a value back to where it was called from**, marking the end of its execution.
The **function returns the calculated totalCost**, which is the final result of the operations performed inside the function.

```
let smoothieCost = calculateSmoothieCost(5, 30, 10)
console.log("This smoothie costs  " + smoothieCost + " kr");
// Output: This smoothie costs 45 kr.
```

The **function** ***calculateSmoothieCost*** is called with the **arguments 5, 30 and 10.**
The function calculates the sum, which is 45, and **returns** this value.

The **smoothieCost** variable saves the **return** value of 45.
Now, the result of the function is stored in the **variable**, and you can use it in the rest of your code - here we used it for a **console.log()**
When you use the return statement, you specify the value that should be sent back to the caller. The functions only returns what is written after the word return, in this case, **totalCost.**

The return statement immediately stops the function's execution - no matter where in the function it is written. If the return statement is placed as the very first line of code in a function, the function will immediately **return the specified value** and stop executing any remaining code in that function.

This means that the computer will not process or reach any of the code written after the return statement - the code that appears after this statement is effectively ignore.

You can also use a return statement by itself to stop the function from running any further. When you use return without specifying a value, it immediately exits the function and stops any code that follows from being executed. **Control is then handed back to the part of the program where the function was called, and the program continues** executing the subsequent lines of code from that point onward.

```
function exampleFunction(x, y) {
let sum
return

sum = x+y //this gets ignored, the code has jumped out of the function
}
```

When you use **return without a value, the function will still return *undefined***. In JavaScript, undefined is a **special value that indicates the absence of a value.** This is essentially the default return value when no explicit value is provided.

This happens only if you use **return** without specifying a return value, and store the result of the function in a **variable**. If you don’t store the result of the function in a **variable** when using **return** without a return value, the function simply stops executing and the program picks up where it left of.

If a function uses the **return statement** and the result of the function call is **not stored in a variable**, the **function still performs its task and returns a value.** However, that **returned value is not saved or used in any way.**

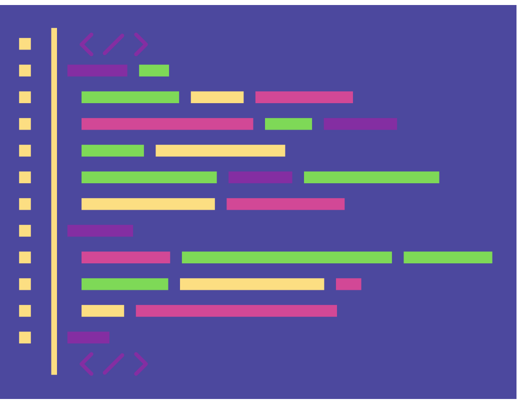
If you do not store the return value in a variable or use it in any way, it gets effectively discarded. **The function performs its task, but you don’t save or make use of the result.**

```
function calculateSmoothieCost(fruitCost, liquidCost, sweetenerCost) {
 // Calculate the total cost by summing up the costs
let totalCost = fruitCost + liquidCost + sweetenerCost

return  totaltCost
}
//Calling the function without storing the result of the return
calculateSmoothieCost(25, 40,5)
```

Since we don’t store this returned value or use it in any way, the result 70 is effectively discarded. **The function performs its calculation, but there’s no record of the result.**

The returned value is lost if it’s not stored, and you won’t be able to use it later in your program.

This can also be helpful. **Say you want to exit a function early based on certain conditions.** For example, if you’re **checking if a user’s input is valid and it’s not, you might want to stop further processing,** allowing you to exit a function early when certain conditions are not met and preventing unnecessary processing. Imagine you have a function that makes a smoothie, but you need to check if all the ingredients are there before starting. If any ingredient is missing, you want to stop the function and avoid making the smoothie.



If a function does not include a return statement, it performs its task but does not send any value back to the caller.

The function prints a greeting to the console but does not return any value. Calling ***printGreeting***("Alice") will display "Hello, Alice!" but there is no value to save use elsewhere in the code.

The instructions in the functions are still performed - but since the return statement is not used, there is no result to store anywhere and reuse.

```
function printGreeting(name) {

 console.log("Hello, " + name + "!")

}
```

MODULARITY & REUSABILITY

Functions are essential for organizing code into **reusable and modular components.** When we say that a function is modular, we mean that it **handles one specific task**, making it a **small**, **self-contained** and **independent part of a larger system.** This is like breaking down a complex recipe into simple, clear steps that are easy to follow.
These parts can be developed and maintained independently of one another, not affecting the rest of the program.

Imagine you want to make a smoothie, but instead of treating the entire process as one big task, you break it down into smaller, manageable steps.

You have a function called **chooseFruit().** This function's sole job is to pick the type of fruit for the smoothie. You can choose bananas, strawberries, or any other fruit you like.

Another function, **selectLiquid()**, is responsible for choosing the liquid. This could be milk, orange juice, or even water. Again, this function only handles this one specific choice.

Then, you have a function called **addSweetener()**. This function decides what type of sweetener, if any, you want to add—like honey, sugar, or agave syrup.

For modular code, one could use functions like chooseFruit(), selectLiquid(), and addSweetener() to break down the process into manageable, reusable components. This modular approach makes the code more flexible and easier to maintain. Finally, blendSmoothie() can combine these elements to complete the smoothie-making process.

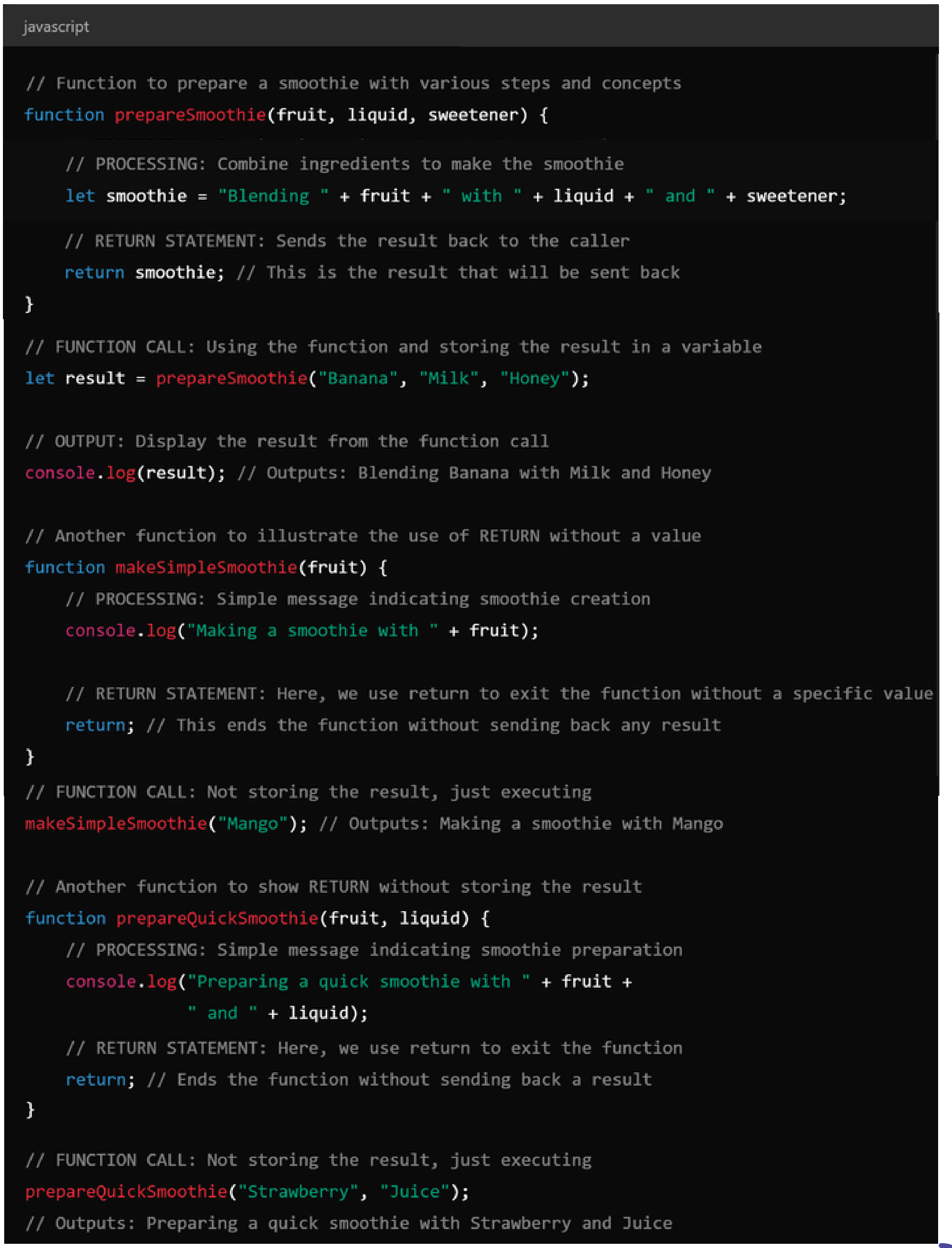
Now, let's talk about reusability - this is where the true power of functions shines.
Once you've created these small, modular functions, you **don't need to rewrite them every time you want to make a smoothie.** Instead, you can reuse them, simply by providing different inputs (arguments) to create different smoothies.

```
makeSmoothie(mango, coconutmilk, sugar)
makeSmoothie(strawberry, milk, sugar)
makeSmoothie(banana, condensedmilk, caramelSauce)
makeSmoothie(banana, milk, maplesyrup)
makeSmoothie(strawberry, vanillasoymilk, agavesuryp)
```

This saves you time and effort, as you **don't need to rewrite or duplicate code. You just use the functions you've already built, adjusting the inputs to get the exact smoothie you want**

CODING - EXAMPLES & HOW TO

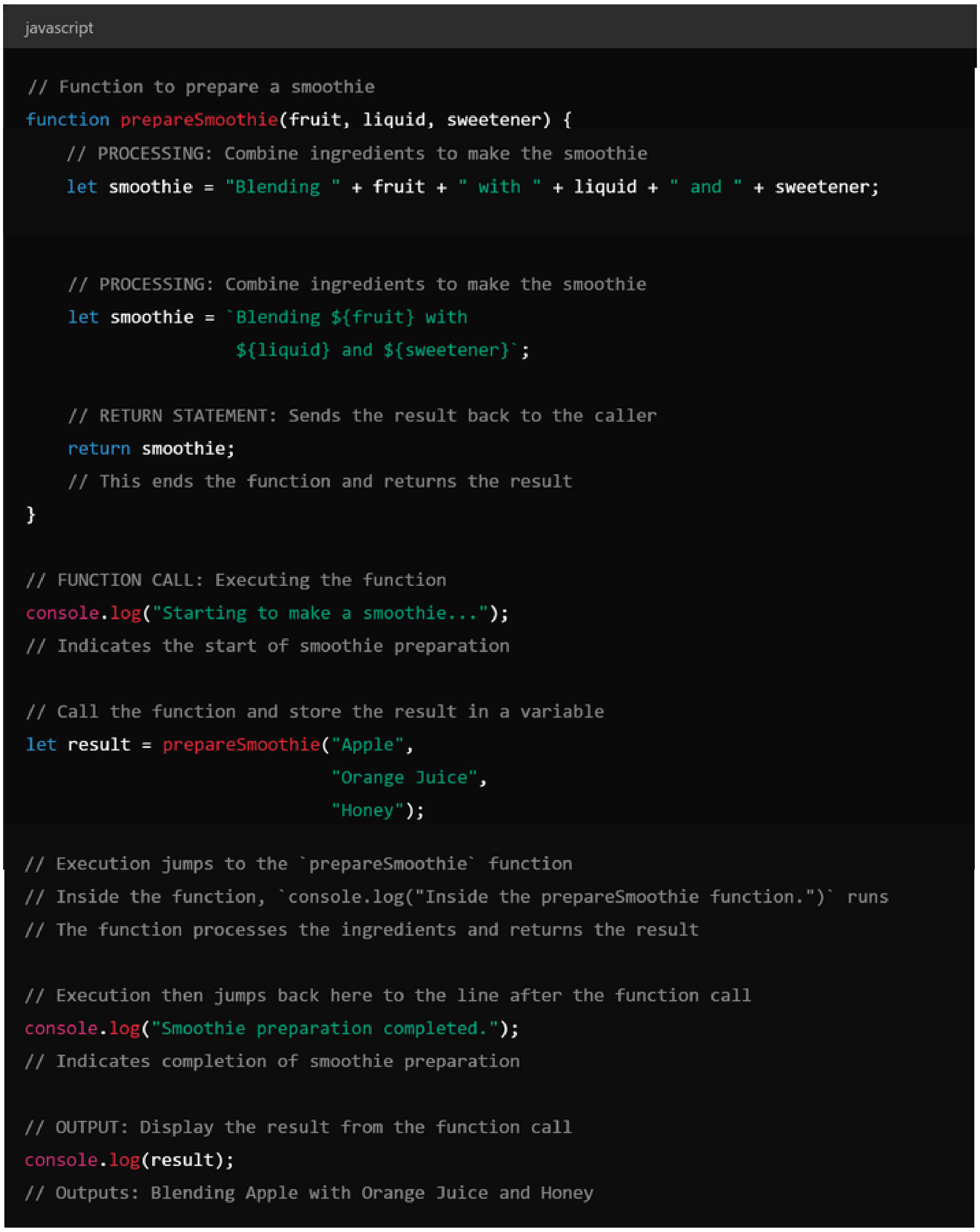
EXAMPLES



```javascript
// Function to prepare a smoothie with various steps and concepts
function prepareSmoothie(fruit, liquid, sweetener) {
    // PROCESSING: Combine ingredients to make the smoothie
    let smoothie = "Blending " + fruit + " with " + liquid + " and " + sweetener;

    // RETURN STATEMENT: Sends the result back to the caller
    return smoothie; // This is the result that will be sent back
}
// FUNCTION CALL: Using the function and storing the result in a variable
let result = prepareSmoothie("Banana", "Milk", "Honey");

// OUTPUT: Display the result from the function call
console.log(result); // Outputs: Blending Banana with Milk and Honey

// Another function to illustrate the use of RETURN without a value
function makeSimpleSmoothie(fruit) {
    // PROCESSING: Simple message indicating smoothie creation
    console.log("Making a smoothie with " + fruit);

    // RETURN STATEMENT: Here, we use return to exit the function without a specific value
    return; // This ends the function without sending back any result
}
// FUNCTION CALL: Not storing the result, just executing
makeSimpleSmoothie("Mango"); // Outputs: Making a smoothie with Mango

// Another function to show RETURN without storing the result
function prepareQuickSmoothie(fruit, liquid) {
    // PROCESSING: Simple message indicating smoothie preparation
    console.log("Preparing a quick smoothie with " + fruit +
                " and " + liquid);
    // RETURN STATEMENT: Here, we use return to exit the function
    return; // Ends the function without sending back a result
}

// FUNCTION CALL: Not storing the result, just executing
prepareQuickSmoothie("Strawberry", "Juice");
// Outputs: Preparing a quick smoothie with Strawberry and Juice
```

CODING - EXAMPLES & HOW TO

JUMPING IN CODE WHEN DOING A FUNCTION CALL

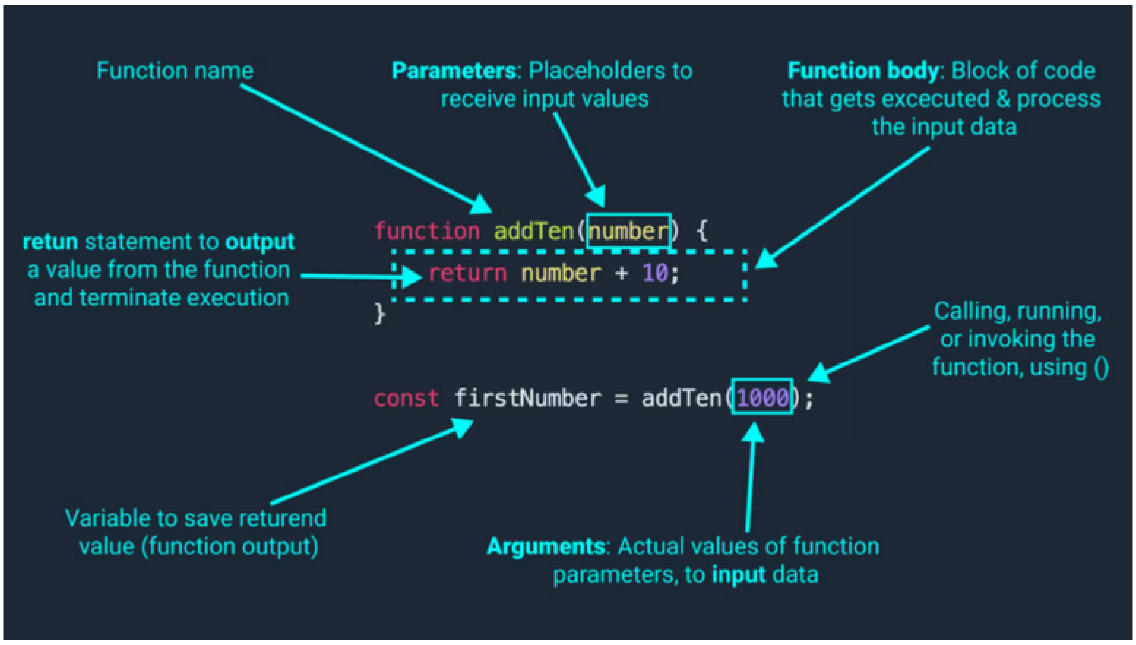


javascript

```javascript
// Function to prepare a smoothie
function prepareSmoothie(fruit, liquid, sweetener) {
    // PROCESSING: Combine ingredients to make the smoothie
    let smoothie = "Blending " + fruit + " with " + liquid + " and " + sweetener;


    // PROCESSING: Combine ingredients to make the smoothie
    let smoothie = `Blending ${fruit} with
                    ${liquid} and ${sweetener}`;

    // RETURN STATEMENT: Sends the result back to the caller
    return smoothie;
    // This ends the function and returns the result
}

// FUNCTION CALL: Executing the function
console.log("Starting to make a smoothie...");
// Indicates the start of smoothie preparation

// Call the function and store the result in a variable
let result = prepareSmoothie("Apple",
                             "Orange Juice",
                             "Honey");

// Execution jumps to the `prepareSmoothie` function
// Inside the function, `console.log("Inside the prepareSmoothie function.")` runs
// The function processes the ingredients and returns the result

// Execution then jumps back here to the line after the function call
console.log("Smoothie preparation completed.");
// Indicates completion of smoothie preparation

// OUTPUT: Display the result from the function call
console.log(result);
// Outputs: Blending Apple with Orange Juice and Honey
```

CODING - EXAMPLES & HOW TO

THE ANATOMY OF A FUNCTION



```
function addTen(number) {
  return number + 10;
}

const firstNumber = addTen(1000);
```

Function name
Parameters: Placeholders to receive input values
Function body: Block of code that gets excecuted & process the input data
retun statement to **output** a value from the function and terminate execution
Calling, running, or invoking the function, using ()
Variable to save returend value (function output)
Arguments: Actual values of function parameters, to **input** data

KEYWORDS IN JAVASCRIPT

function	Used to declare a function.
return	Used to return a value from a function.
var, let, const	Used to declare variables.