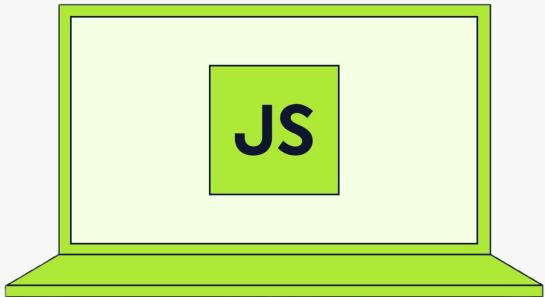




The Complete Javascript Course



@newtonschool

Lecture 1: Basics of Javascript

-Bhavesh Bansal

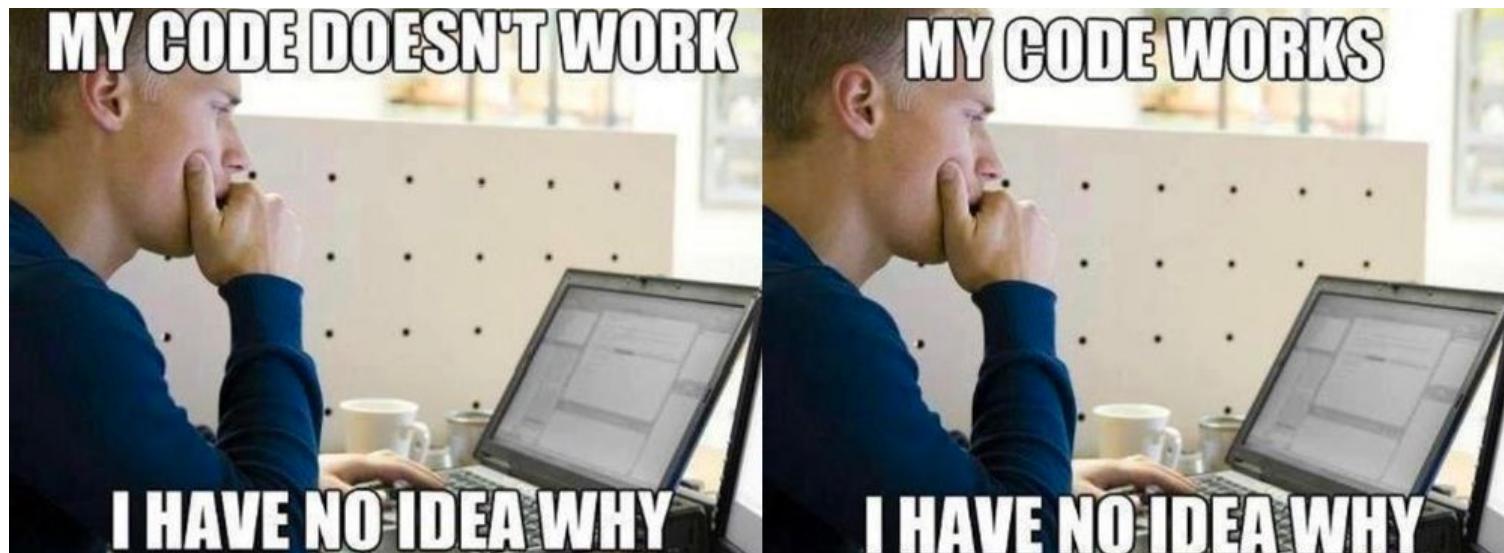


Table of Contents

- What is Javascript?
- Variable Declaration: var, let and const
- Differences between let, var and const
- Example of variable usages in real world scenarios

Considerations before we start...

In first few sections **don't stress about why code works** and how to write efficient code, or clean code. **Just start learning and eventually you will understand everything.**



Let's Start

What is Javascript?

JAVASCRIPT

The language that makes web
pages come alive.



Create web,
mobile, or desktop
apps.

Animate elements
and make pages dynamic.

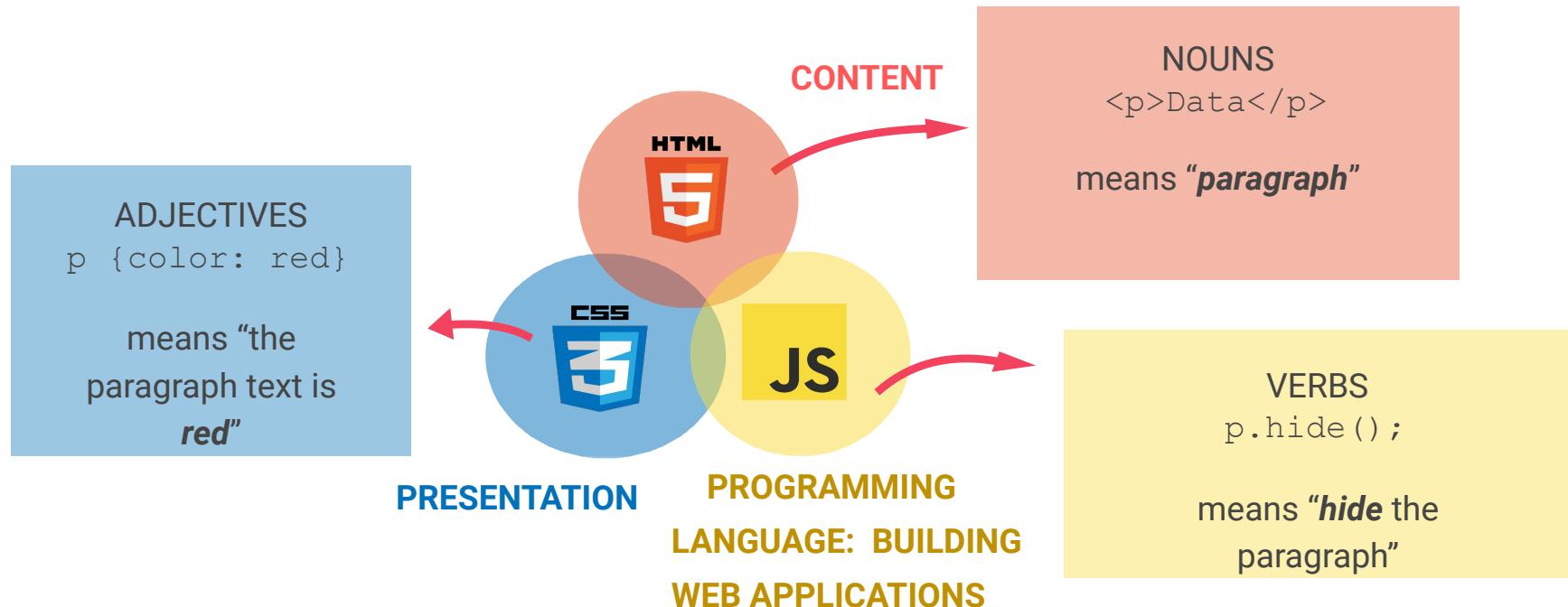
Run
back-end
code with
Node.js.



Fetch and process data dynamically.

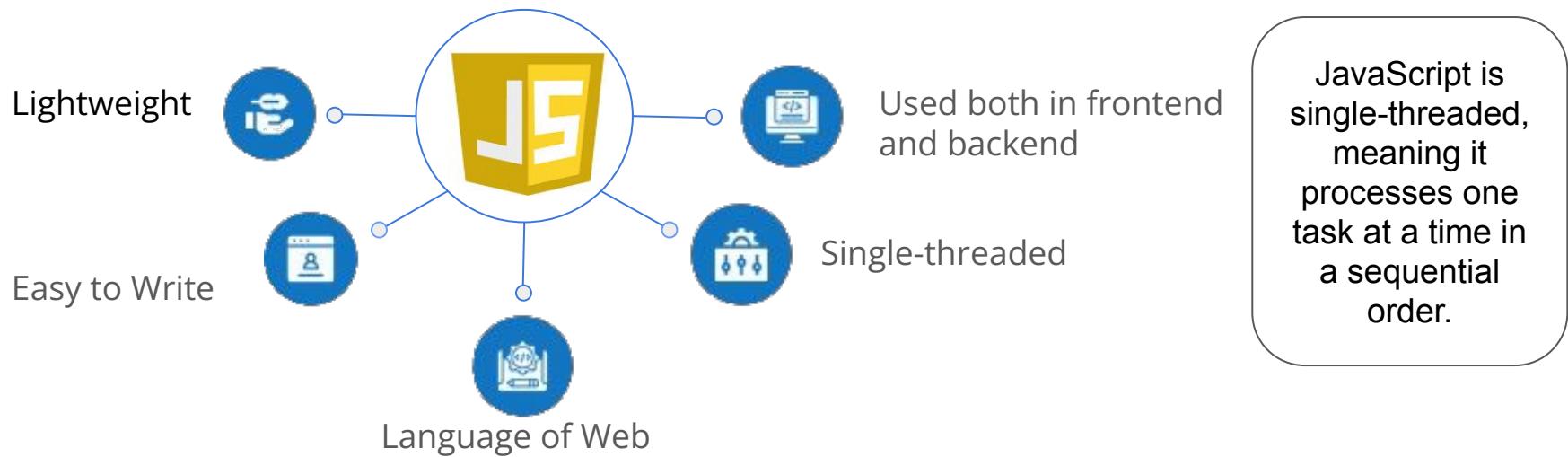
Javascript Role in Web Development

Javascript adds logic to our web pages:-



Features of Javascript

JavaScript is a programming language that makes websites interactive, like adding buttons, animations, and live updates.



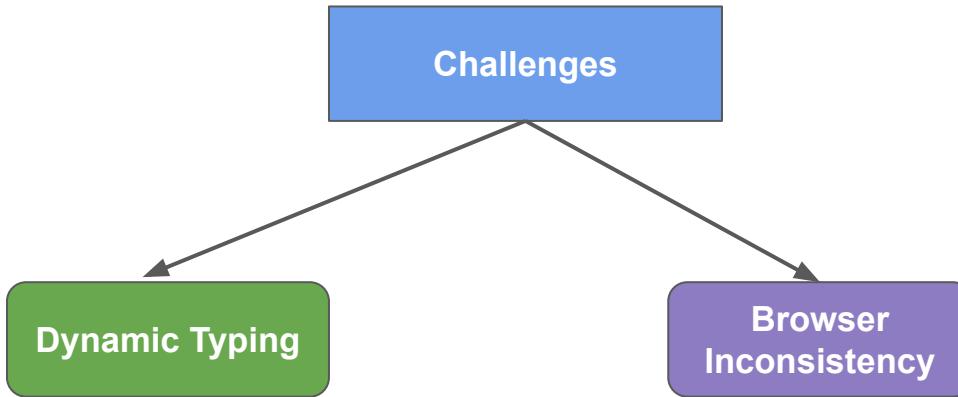
Comparing with other languages

Let's compare javascript with other languages:-

	JavaScript	C#	Java	Ruby	Python
Front-End	✓	✗	✗	✗	✗
Back-End	✓	✓	✓	✓	✓
Mobile Apps	✓	✓	✓	✓	✓
Desktop Apps	✓	✓	✓	✓	✓
Easy to learn	✓	✗	✗	✓	✓

Challenges in Javascript

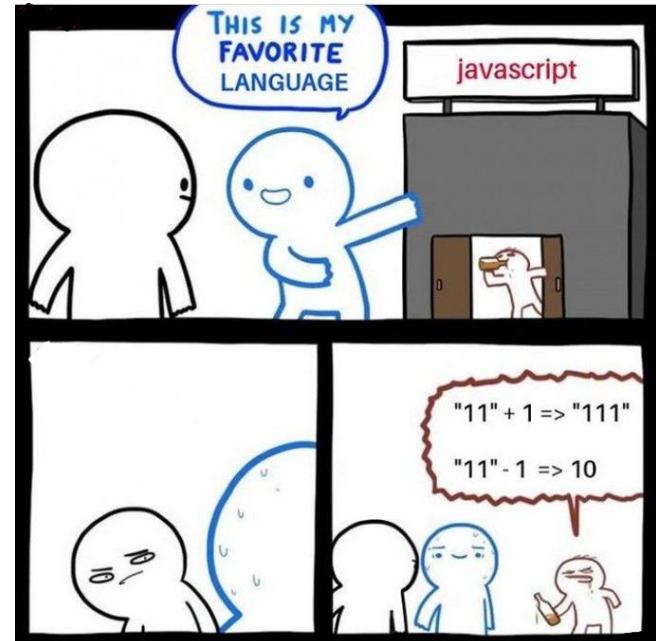
JavaScript's dynamic typing can cause bugs, and browser inconsistencies require extra effort for compatibility.



`"1" + 2 => 12`
`1 + 2 => 3`

Such issues due to dynamic typing leads unpredictable code behaviour

Behaviour of javascript code varies depending on which browser is loading your web page.



The Origin of JavaScript

A Browser Ahead of Its Time

In the 1990s, Netscape Navigator changed the web. It was the first widely-used browser, capable of beautifully displaying HTML and CSS



NetScape Browser



Early HTML/CSS website

A World of Static Websites

But there was a problem – web pages were static. They couldn't respond dynamically to user actions

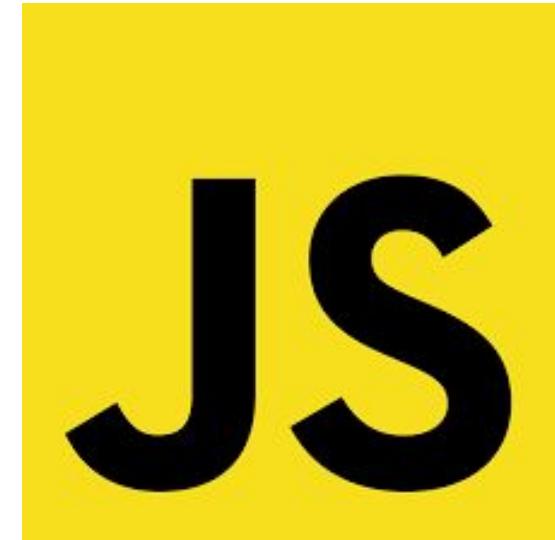


Apple Computer...Features

Apple Computer Names Gilbert F. Amelio Chairman and Chief Executive Officer
CUPERTINO, California—February 2, 1996—Apple Computer, Inc. today announced
agreed that it was in the best interest of Apple Computer to have a transition in lead
Directors has appointed Dr. Gilbert F. Amelio, formerly Chairman, President and Ch

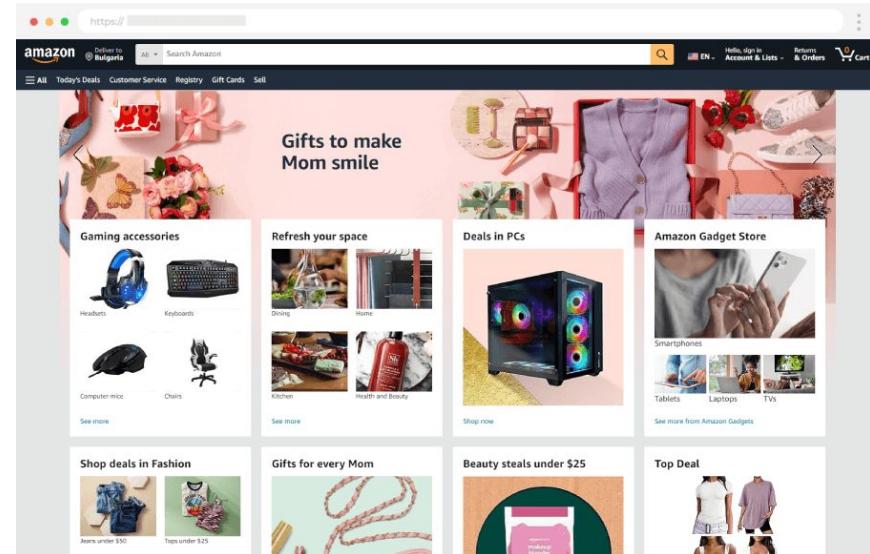
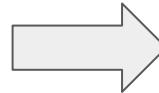
Brainchild: Brendan Eich

To solve the issue, Brendan Eich was tasked with creating a Java-like scripting language for the web, and amazingly, he accomplished it in just 10 days!



JavaScript Revolutionizes the Web

With Brendan Eich's creation of JavaScript, websites transformed from static pages into dynamic, interactive experiences.



Apple Computer...Features

[Apple Computer Names Gilbert F. Amelio Chairman and Chief Executive Officer](#)

CUPERTINO, California--February 2, 1996--Apple Computer, Inc. today announced agreed that it was in the best interest of Apple Computer to have a transition in lead Directors has appointed Dr. Gilbert F. Amelio, formerly Chairman, President and Ch

Advancements in Javascript

1995



- 👉 Brendan Eich creates the **very first version of JavaScript in just 10 days.**



1996



- 👉 Mocha changes to LiveScript and then to JavaScript, in order to attract Java developers. `👉



1997



- 👉 ECMA releases ECMAScript 1 (ES1), the first **official standard for JavaScript**



2009



- 👉 ES5 (ECMAScript 5) is released with lots of great new features;

2015



- 👉 ES6/ES2015 (ECMAScript 2015) was released: **the biggest update to the language ever!**

2016 - ∞



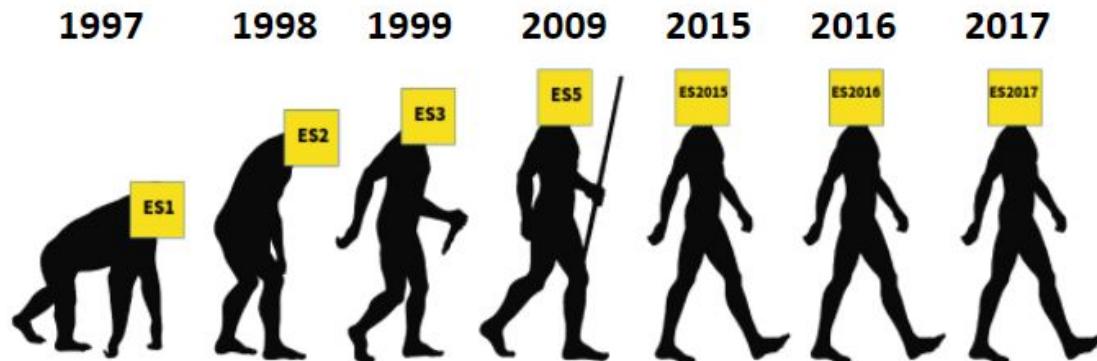
- 👉 Release of ES2016 / ES2017 / ES2018 / ES2019 / ES2020 / ES2021 / ... / ES2089



Javascript
was named
'Mocha'
earlier.

ECMA: Architect Behind Javascript

ECMA(European computer manufacturers association) plays a vital role in standardizing and advancing JavaScript for modern development.



ECMA is a Committee which regularly releases standards for JavaScript updates making javascript more powerful and efficient.

ES6: The Game Changer for JavaScript

ES6 introduced modern features like classes, modules, arrow functions, and promises, making JavaScript more structured, concise, and easier to work with. And not to forget let and const.



Most of the
javascript which we
use today is ES6.

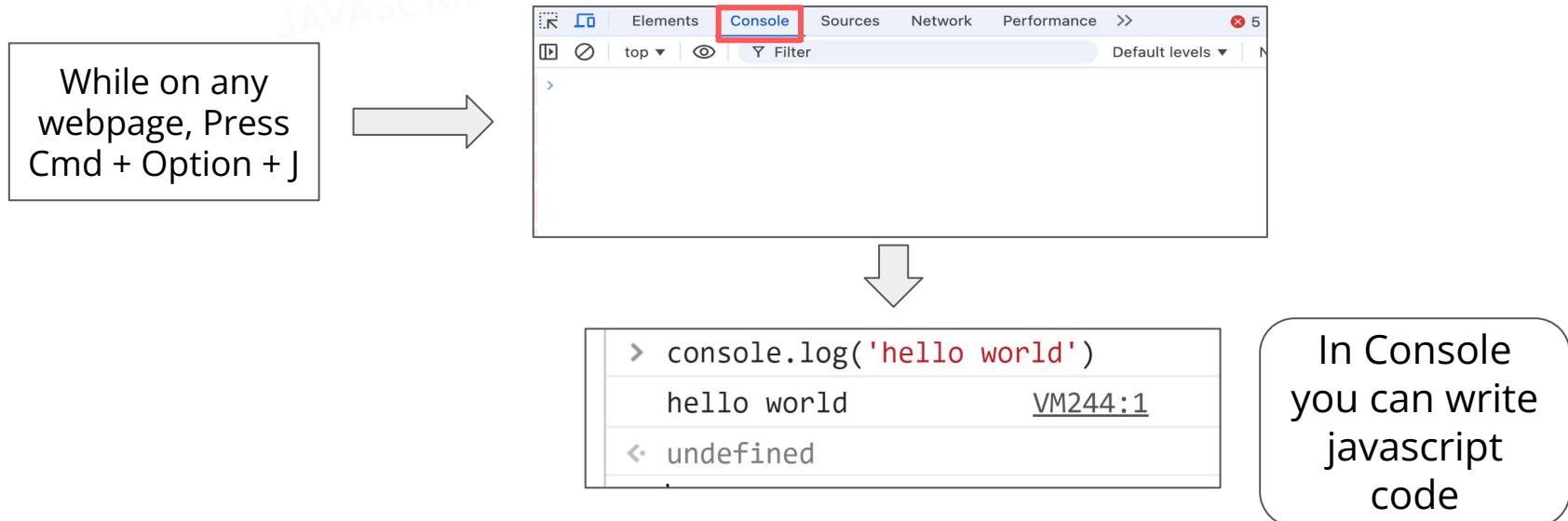
How to run Javascript code?

You can run JavaScript directly in a browser using the Developer Console or by embedding it in an HTML file with <script> tags.

1. Developer Console
2. Embedding in HTML(<script>...</script> tags)

Running javascript in Developer Console

Open the Developer Console in Chrome on Mac using Cmd + Option + J. Type your JavaScript code in the Console tab and press Enter to execute it.



Run javascript: script tag

We use script tag to either embed code inside it or link code:-



```
<script>
    /* Your javascript code here */
    // Code line 1
    // Code line 2
    // Code line 3
    // .....
</script>
```

Embedding Code

```
<script src="script.js"></script>
```



index.html

script.js

```
1 // write your javascript here
```

Linking External Code

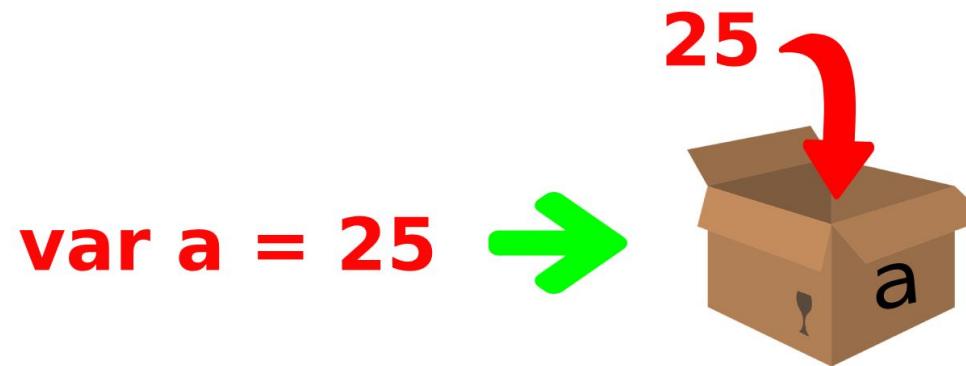


Let's Start Learning

Javascript Syntax

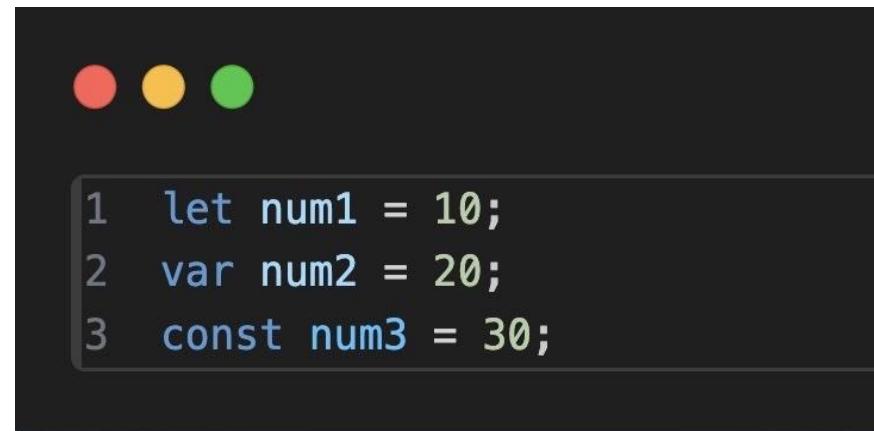
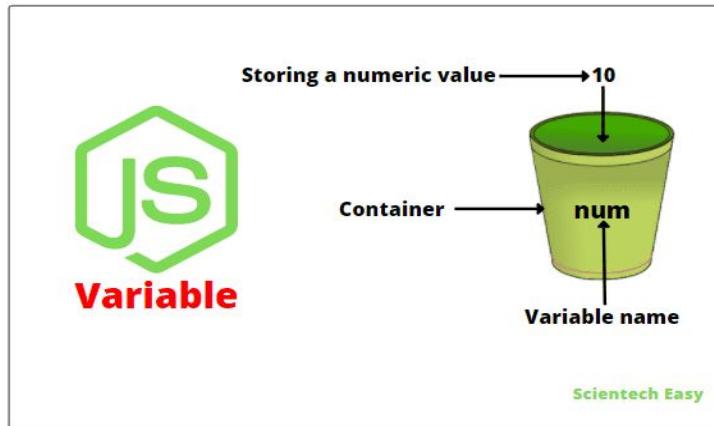
Variables in Javascript

Variables in JavaScript are containers for storing data values. They allow developers to label and reference data in their programs,



Storing Values in Javascript

JavaScript provides three keywords to declare variables: **let**, **var**, and **const**. These can be used to declare and store values.



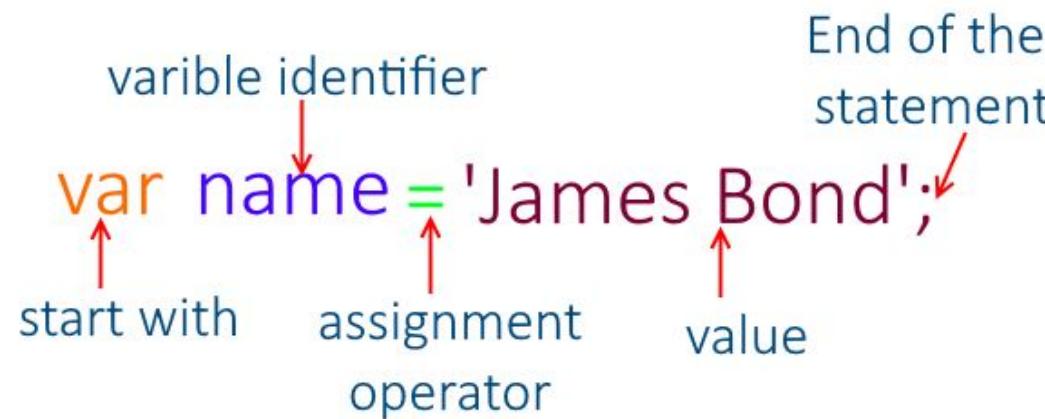
A screenshot of a terminal window on a dark background. At the top, there are three colored circles: red, yellow, and green. Below them, three lines of code are displayed in blue text:

```
1 let num1 = 10;
2 var num2 = 20;
3 const num3 = 30;
```

Syntax for variable declaration

Variables in JavaScript are declared using `let`, `const`, or `var`, followed by a variable name.

Keyword used to declare a variable:-



The diagram illustrates the syntax of a variable declaration in JavaScript. The code is:

```
var name = 'James Bond';
```

Annotations explain the components:

- "variable identifier" points to the word "name".
- "start with" points to the "v" in "var".
- "assignment operator" points to the "=" symbol.
- "value" points to the string "'James Bond'".
- "End of the statement" points to the final ";" character.

Similarly we can declare for let and const

Variables in JavaScript are declared using `let`, `const`, or `var`, followed by a variable name, `let` is a better alternative of `var`

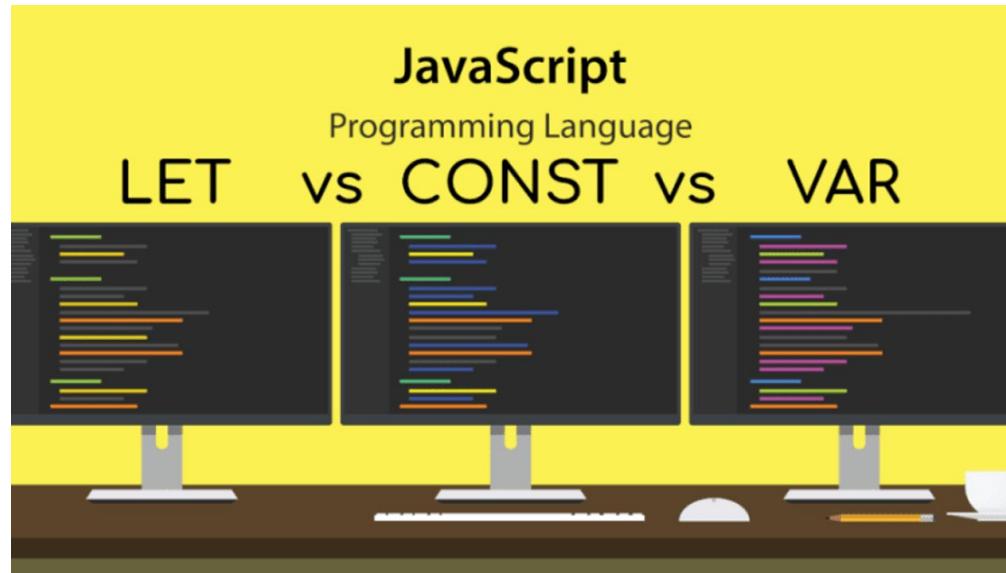


```
1 let firstName = "Alice"; //can be reassigned
2 firstName = "Bob"; //reassignment is allowed
3
4 const age = 25; //can't be reassigned
```

Once assigned,
the value of a
`const` variable
cannot be
changed or
reassigned.

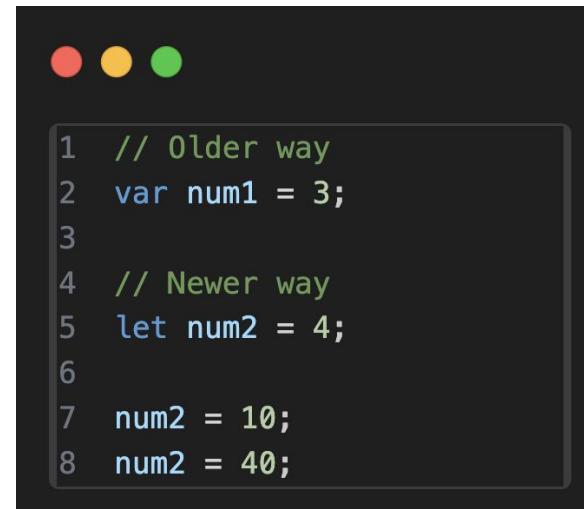
Why three keywords to store values?

Why does JavaScript need three different ways—`var`, `let`, and `const`—just to store values? Is there really a big difference between them?



Difference between var and let

`let` is the newer and preferred way to declare variables in JavaScript, offering better control and safety compared to the older `var`.



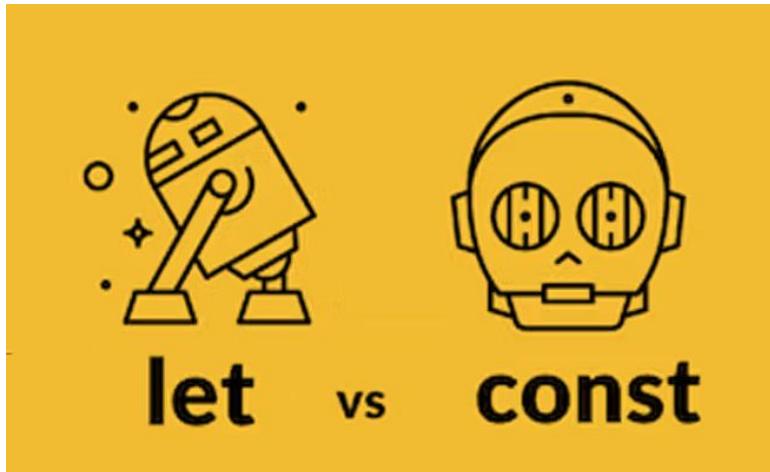
```
1 // Older way
2 var num1 = 3;
3
4 // Newer way
5 let num2 = 4;
6
7 num2 = 10;
8 num2 = 40;
```

The image shows a terminal window with a dark background and light-colored text. It displays two snippets of JavaScript code. The first snippet, starting with '1 // Older way', uses the 'var' keyword to declare a variable 'num1' and assign it the value 3. The second snippet, starting with '4 // Newer way', uses the 'let' keyword to declare a variable 'num2' and assign it the value 4. Both snippets then show the variable being reassigned values of 10 and 40 respectively, demonstrating that variables declared with 'let' can be changed.

Values declared with `var` and `let` can be changed

Understood!! But what about `let` and `const`

`let` lets you change the value later, while `const` keeps the value fixed once set.



```
1 // Using `let`
2 let firstName = "Alice";
3 // Declare and assign a value
4
5 firstName = "Bob"; // Reassign a new value
6
7 // Using `const`
8 const lastName = "Smith";
9 // Declare and assign a value
10 console.log(lastName); // Output: Smith
11
12 // Attempt to reassign a new value to `const` 
13 // TypeError: Assignment to constant variable
```

But, how can we check the stored values?

We can check stored values in variables using `console.log()`.

JS

console.log()

Understanding console.log in javascript

console.log() is a JavaScript method that outputs messages or data, helping developers debug and monitor code execution.

```
> let firstName = "Alice"
< undefined
> console.log(firstName)
Alice
```

Comments: The Footnotes of Code

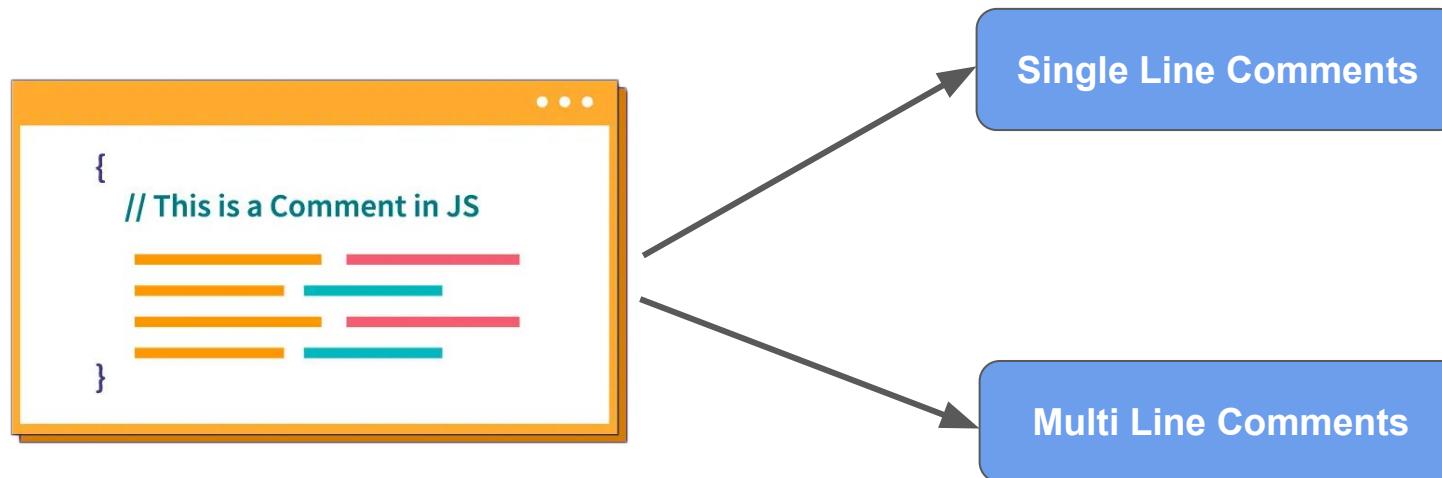
Comments are like notes in a book, offering explanations without affecting the code. Always comment your code, not just for others, but for your future self too.



Comments helps you understand the code better later one.

Types of Comments

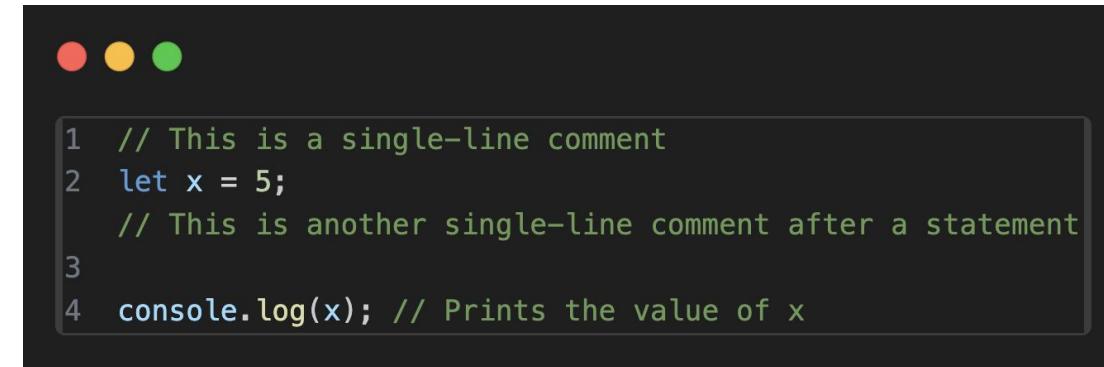
You can use either single line comments or multi-line comments in javascript.



Comments: Single Line

Single-line comments are used to add brief explanations or notes within a single line of code.

Single line comments start with `//` and end within the line.



```
1 // This is a single-line comment
2 let x = 5;
    // This is another single-line comment after a statement
3
4 console.log(x); // Prints the value of x
```

Comments: Multi-Line

Multi-line comments are used to add longer explanations or comment out multiple lines of code, enclosed between `/*` and `*/`.

```
1  /*
2   This function calculates the total of cart items.
3   If the total exceeds $100, a 10% discount is applied.
4 */
5  function calculateTotal(cart) {
6      let total = 0;
7      for (let item of cart) total += item.price;
8      return total > 100 ? total * 0.9 : total;
9 }
```

Use multi-line comments for longer explanations, commenting out blocks of code, or providing detailed documentation.

Some Real World Examples

Temperature Changes Over a Day

The temperature of a room or a city changes throughout the day.

In Programming:

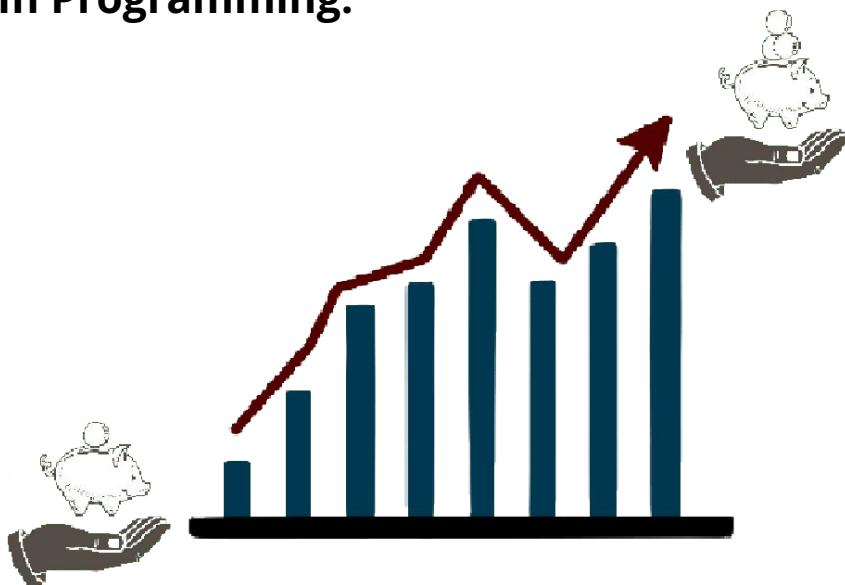


```
1 let temperature;
2 // Declare a variable to store the temperature
3
4 // Morning temperature
5 temperature = 20; // Assign morning temperature
6
7 // Afternoon temperature
8 temperature = 28;
// Update with afternoon temperature
9
10 // Evening temperature
11 temperature = 22;
// Update with evening temperature
```

Bank Account Balance

Your bank account balance fluctuates with deposits and withdrawals.

In Programming:



```
let bankBalance = 1000;  
// initial balance in rupees  
console.log(bankBalance); // output : 1000  
  
// you make a deposit of $500  
bankBalance += 500;  
// Adding money to the account  
console.log(bankBalance); // output : 1500  
  
// you withdraw $300  
bankBalance -= 300;  
// subtracting money from the account  
console.log(bankBalance); // output : 1200
```

Speed of a Car

A car's speed changes depending on driving conditions and actions.

In Programming:



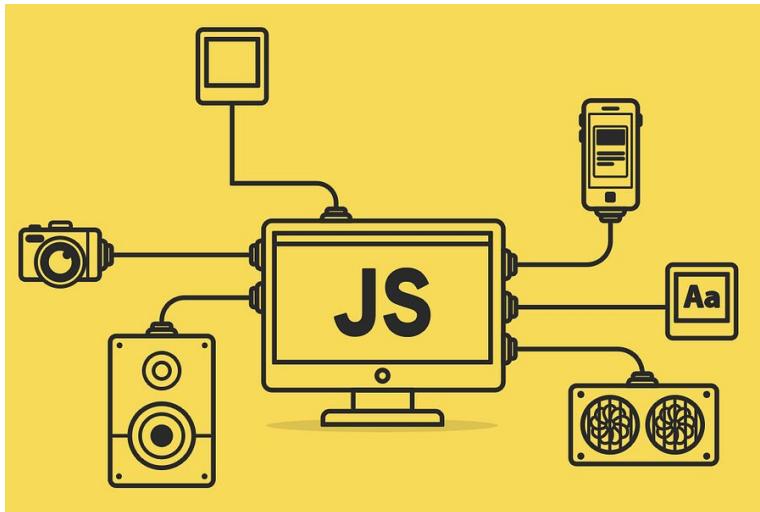
```
● ● ●  
1 // Initial speed  
2 carSpeed = 0;  
3 // The car is stationary  
4  
5 // Speed after starting  
6 carSpeed = 60;  
7 // The car is now moving  
8  
9 // Speed on the highway  
10 carSpeed = 100;  
11 // The car speeds up on the highway  
12
```

In Class Questions

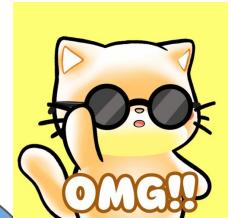
References

1. **MDN Web Docs - JavaScript:** Comprehensive and beginner-friendly documentation for JavaScript.
<https://developer.mozilla.org/en-US/docs/Web/JavaScript>
2. **Eloquent JavaScript:** A free online book covering JavaScript fundamentals and advanced topics.
<https://eloquentjavascript.net/>
3. **JavaScript.info:** A modern guide with interactive tutorials and examples for JavaScript learners.
<https://javascript.info/>
4. **freeCodeCamp JavaScript Tutorials:** Free interactive lessons and coding challenges to learn JavaScript.
<https://www.freecodecamp.org/learn/>

Did you know?



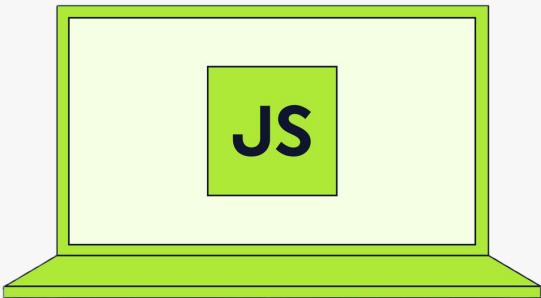
*JavaScript powers over
98% of websites,
making it most popular
programming
language! 🌐 ✨*



**Thanks
for
watching!**



The Complete Javascript Course



@newtonschool

Lecture 2: Data Types and Operators

-Vishal Sharma

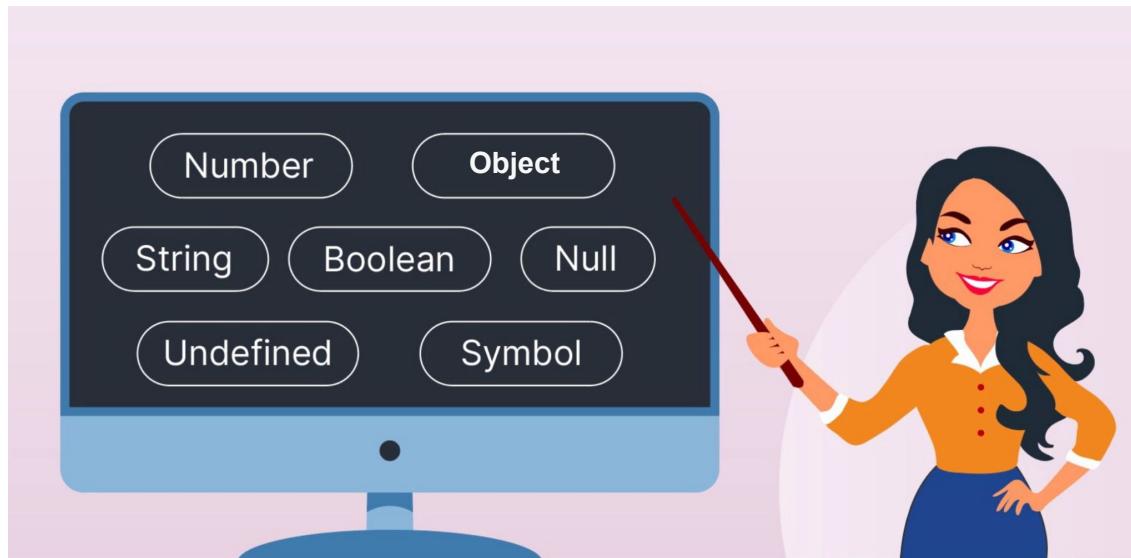


Table of Contents

- Primitive and Non-Primitive Data Types
- Types of primitive data types
- Type checking using typeof
- Arithmetic Operators
- Comparison Operators
- Logical Operators
- Assignment Operators
- Practical Usage and Examples

What are Data Types in javascript?

In JavaScript, **data types** are the classification of data values that tell the type of data being dealt with.



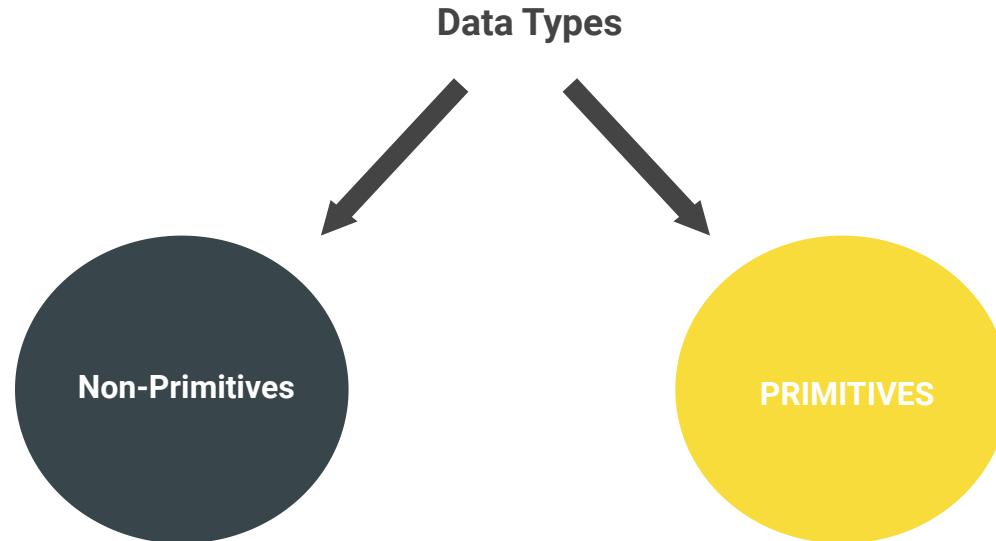
Data types indicate whether data is a number, string, boolean, etc.

Don't worry if you're unfamiliar with them yet.

Primitive and Non-Primitive Data Types

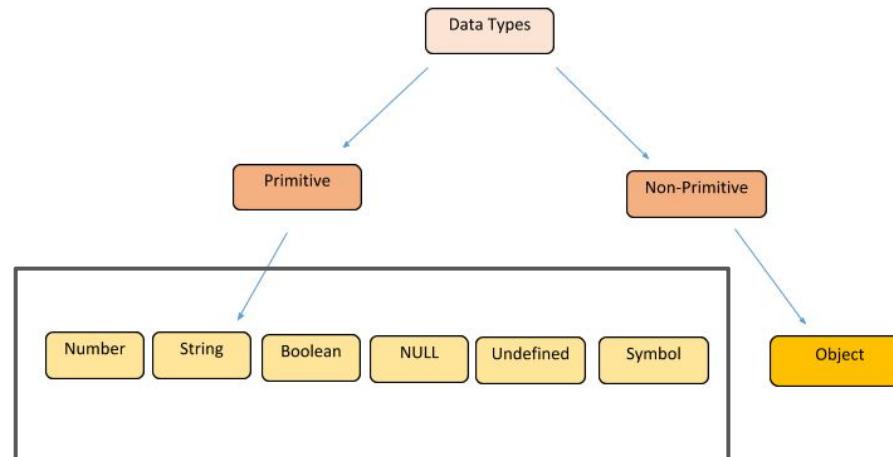
Categories of Data Types

In JavaScript, data types are primitives (simple, immutable) and non-primitives (complex, mutable, and multi-valued).



Let's start with primitives

Primitive data types in JavaScript are simple, immutable values like numbers, strings, booleans, symbols.



Declaring Primitive Data types

Primitive data types in JavaScript are simple, immutable values like numbers, strings, booleans, undefined, null etc. Let's have a look:-

```
1 // Primitive Data Types in JavaScript
2
3 // Number
4 let age = 25; // Integer
5 console.log("Age:", age); // Output: Age: 25
6
7 // String
8 let firstName = "John"; // Sequence of characters
9 console.log("First Name:", firstName);
// Output: First Name: John
10
11 // Boolean
12 let isActive = true; // Logical value
13 console.log("Is Active:", isActive);
// Output: Is Active: true
```

In JavaScript, the data type of a variable is implicitly determined by the value assigned to it, such as a **Number** for numbers, a **String** for text, and a **Boolean** for true/false values

Declaring Primitive Data types

Primitive data types in JavaScript are simple, immutable values like numbers, strings, booleans, symbols.

```
1 // Primitive Data Types in JavaScript
2
3 // Undefined
4 let userLocation;
  // Variable declared but not assigned
5 console.log("User Location:", userLocation);
  // Output: User Location: undefined
6
7 // Null
8 let car = null; // Intentional absence of a value
9 console.log("Car:", car); // Output: Car: null
```

If a variable has no value, it's automatically **undefined**. To intentionally keep it empty, assign it **null**.

Sounds overwhelming! Don't worry

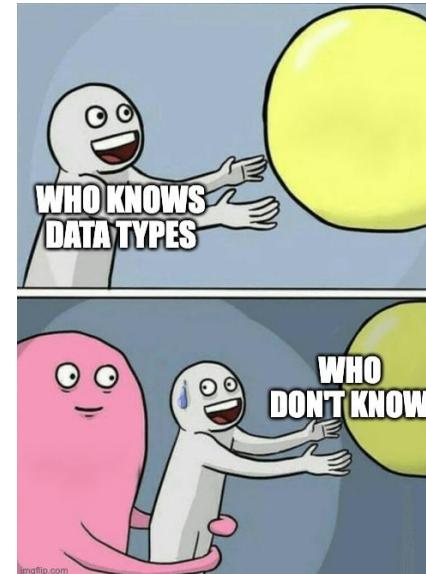
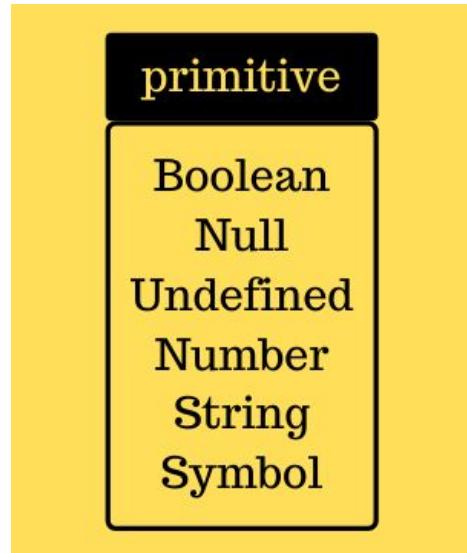
We all find it difficult when we start but it gets easier as we go along



Ask questions when you're unsure, code when things feel confusing, and remember—the more you code, the better you'll understand!

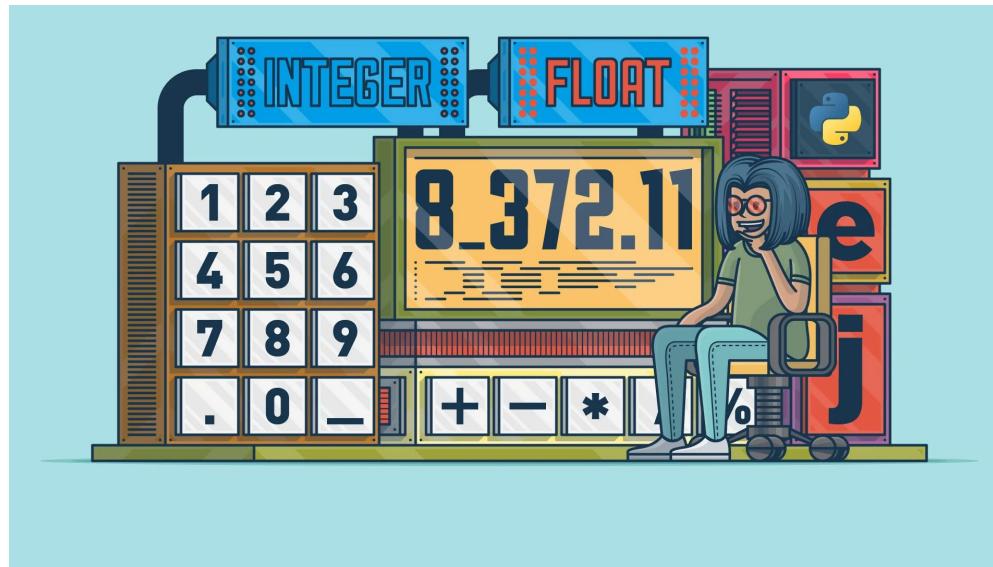
Understanding Data types

Today, we begin our journey through data types, starting with the most common and progressing to the more complex and powerful.



Meet the Numbers

They represent quantities, from counting objects to calculating complex formulas. It could be either simple integers or decimals.

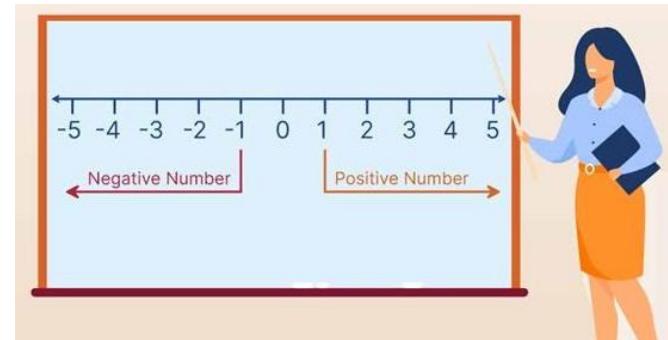


They could either
be integers or
fractions.

Declaration and Assignment

Let's declare and assign some numbers:-

```
1 let intNum = 10;      // Positive integer
2 let negInt = -5;      // Negative integer
3 let floatNum = 3.14;  // Positive float
4 let negFloat = -2.5;  // Negative float
```



Number methods

Number methods in JavaScript help manipulate numeric values efficiently. For instance, `ceil()` rounds fractional value to nearest smaller integer, while `floor()` rounds the number to nearest integer.

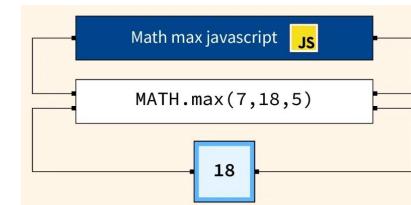
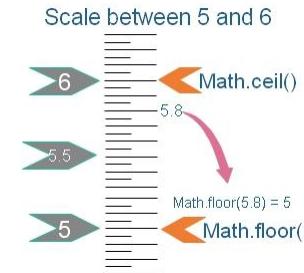


Number Methods

JavaScript provides a variety of built-in methods and operations for working with numbers. Let's start with rounding and converting:-

```

1 // Math.ceil() - Rounds a number UP to the nearest integer
2 console.log("Math.ceil(4.3) =", Math.ceil(4.3)); // 5
3
4 // Math.floor() - Rounds a number DOWN to the nearest integer
5 console.log("Math.floor(4.7) =", Math.floor(4.7)); // 4
6
7 // Math.max() - Returns the largest of the given numbers
8 console.log("Math.max(1,5,3,9,7) =", Math.max(1,5,3,9,7));
9 // 9
10
11 // Math.min() - Returns the smallest of the given numbers
12 console.log("Math.min(1,5,3,9,7) =", Math.min(1,5,3,9,7));
13 // 1
  
```



Number Methods

Let's look at few more:-



```
1 // Generate a random number between 1 and 10
2 let randomValue = Math.random();
3 // Generates a number between 0 (inclusive) and 1 (exclusive)
4
5 // Map the random value to a number between 1 and 10
6 let randomInRange = Math.floor(randomValue * 10) + 1;
7
8 console.log("Random number between 1 and 10:", randomInRange);
9 // Output: Random number between 1 and 10
```

JavaScript Math.random() Method

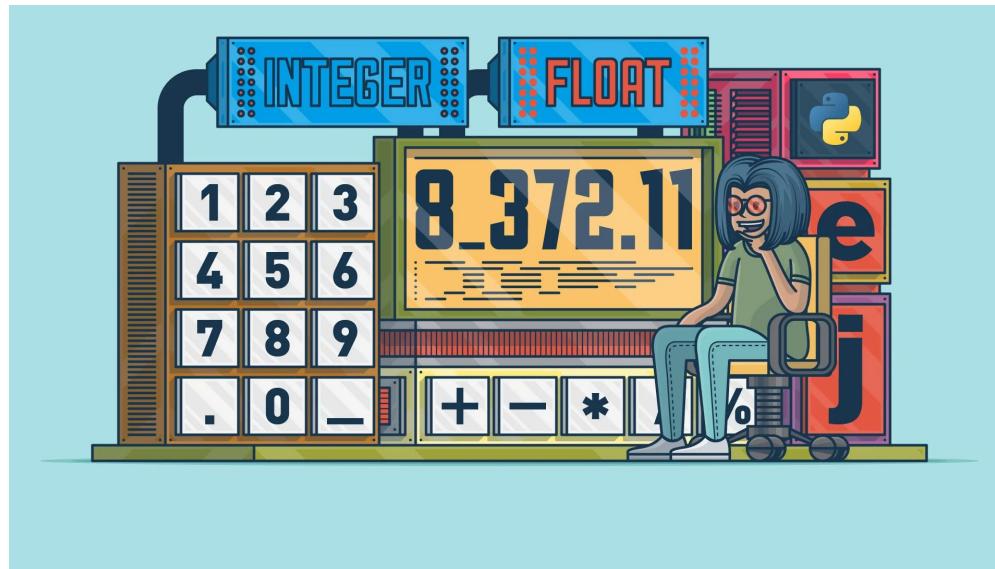


Interactive Activity

Using `Math.random()` find a number
between 2 and 8

Enter the Strings

They are the storytellers of JavaScript. They carry text, allowing us to write messages, descriptions, and even dialogue for our characters.



They could either
be integers or
fractions.

Declaration and Assignment

Let's declare and print some strings:-



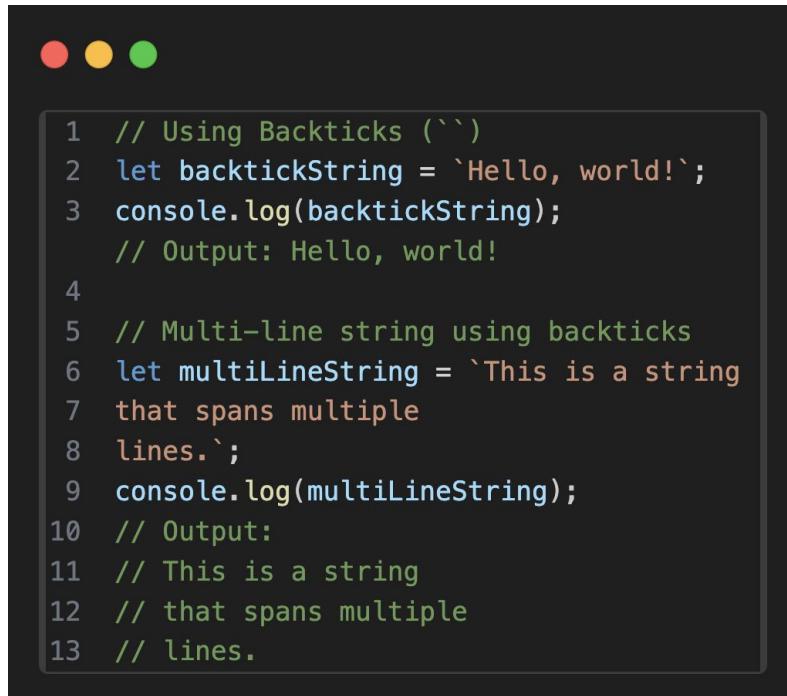
```
1 // Using Single Quotes ('')
2 let singleQuoteString = 'Hello, world!';
3 console.log(singleQuoteString);
4 // Output: Hello, world!
5
6 // Using Double Quotes ("")
7 let doubleQuoteString = "Hello, world!";
8 console.log(doubleQuoteString);
9 // Output: Hello, world!
```

Single Quotes

Double Quotes

Declaration and Assignment

Let's learn how to use backticks and how it is different from '...' and “...”



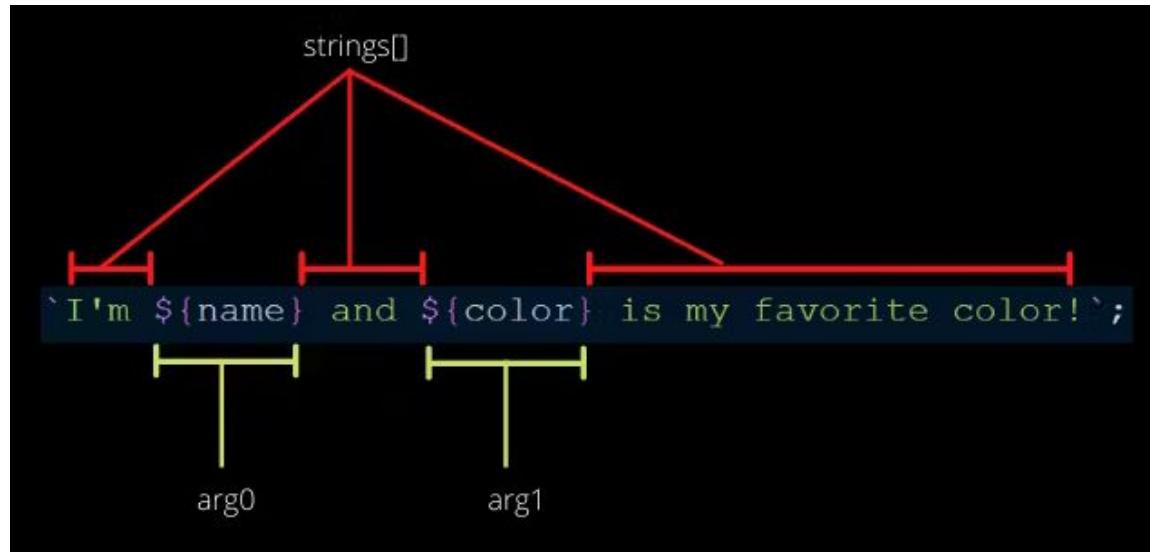
```
1 // Using Backticks (``)
2 let backtickString = `Hello, world!`;
3 console.log(backtickString);
// Output: Hello, world!
4
5 // Multi-line string using backticks
6 let multiLineString = `This is a string
7 that spans multiple
8 lines.`;
9 console.log(multiLineString);
10 // Output:
11 // This is a string
12 // that spans multiple
13 // lines.
```



Backticks
Quotes

Backticks: for template literals

Template literals use **backticks** (```) instead of single or double quotes for defining strings. With them we can embed variables inside the string. Let's see how:-



Here name and
color are
variables.

Backticks: for template literals

Template literals use **backticks** (```) instead of single or double quotes for defining strings. With them we can embed variables inside the string. Let's see how:-

```
1 let nam = 'Narendra';
2 let color = 'blue';
3
4 let str = `I'm ${nam} and ${color}
    is my favourite color`;
5
6 console.log(str);
7 // Output: I'm Narendra and blue is my favourite color!
```

With the help of template literals we embedded variables in the string.

Accessing character values in string

Each character in the string is indexed starting from 0. Consider string as row of mailboxes where each mailbox can hold a character.

For example, let's take string “**Codechef**”:

Index numbers are:

C	o	d	e	c	h	e	f
0	1	2	3	4	5	6	7



Strings are indexed

Let's try to access these character values using index.

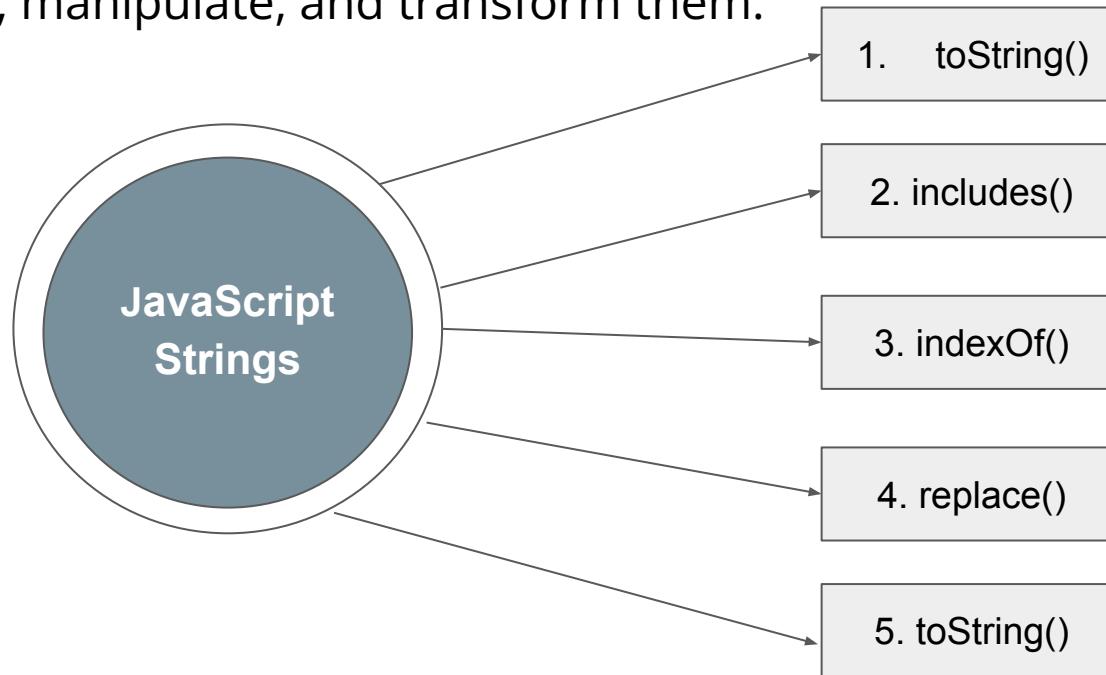
```
1 let str = "codechef";
2
3 // Accessing characters by its index (mailbox number)
4 console.log("At index 0:", str[0]); // Output: c
5 console.log("At index 1:", str[1]); // Output: o
6 console.log("At index 2:", str[2]); // Output: d
7 console.log("At index 3:", str[3]); // Output: e
8 console.log("At index 4:", str[4]); // Output: c
9 console.log("At index 5:", str[5]); // Output: h
10 console.log("At index 6:", str[6]); // Output: e
11 console.log("At index 7:", str[7]); // Output: f
```

We stored codechef in str variable and extracted each character using their respective indexes.

charAt() function achieves the same using index values.

Is that it! There is so much more!

Strings in JavaScript come with a toolbox full of powerful methods to explore, manipulate, and transform them.



String method: `toString()`

The `toString()` method in JavaScript converts a value (such as a number, array, or object) into a string representation.



```
1 let num = 123;
2 console.log(num.toString());
3 // Output: "123"
4
5 let arr = [1, 2, 3];
6 console.log(arr.toString());
7 // Output: "1,2,3"
```

Here we are converting numbers and arrays to strings. Observe how they are being converted.

Converting different data types to strings

String method: trim()

The `trim()` method in JavaScript removes whitespace from both ends of a string without affecting the spaces in between words.

```
1 let str = " Hello, world! ";
2 console.log(str.trim());
3 // Output: "Hello, world!"
```

Trimming down the white spaces

Can you observe the little white spaces and how they have gotten trimmed?

String method: replace()

The `replace()` method in JavaScript is used to replace a specified substring or pattern within a string with another substring.



```
1 let str = "Hello, world!";
2 let newStr = str.replace("world", "JavaScript");
3 console.log(newStr);
4 // Output: "Hello, JavaScript!"
```

We have successfully replaced the word. Repeat it in your code editor.

String method: `toString()`

The `toString()` method in JavaScript converts a value (such as a number, array, or object) into a string representation.

```
1 let num = 123;
2 console.log(num.toString());
3 // Output: "123"
4
5 let arr = [1, 2, 3];
6 console.log(arr.toString());
7 // Output: "1,2,3"
```

Can you observe the little white spaces and how they have gotten trimmed?

String method: includes()

The `includes()` method in JavaScript checks if a string contains a specified substring. It returns `true` if the substring is found, otherwise `false`.

```
1 let sentence = 'JavaScript is awesome!';
2 console.log(sentence.includes('is'));
// true
3 console.log(sentence.includes('awesome'));
// true
4 console.log(sentence.includes('Python'));
// false
5 console.log(sentence.includes('Java', 5));
6 // false (search starts from index 5)
```

It is checking if string includes the word and giving out boolean. Observer the last one, why is it false??

String method: indexOf()

The `indexOf()` method in JavaScript returns the index of the first occurrence of a substring in a string or `-1` if not found. It is case-sensitive.

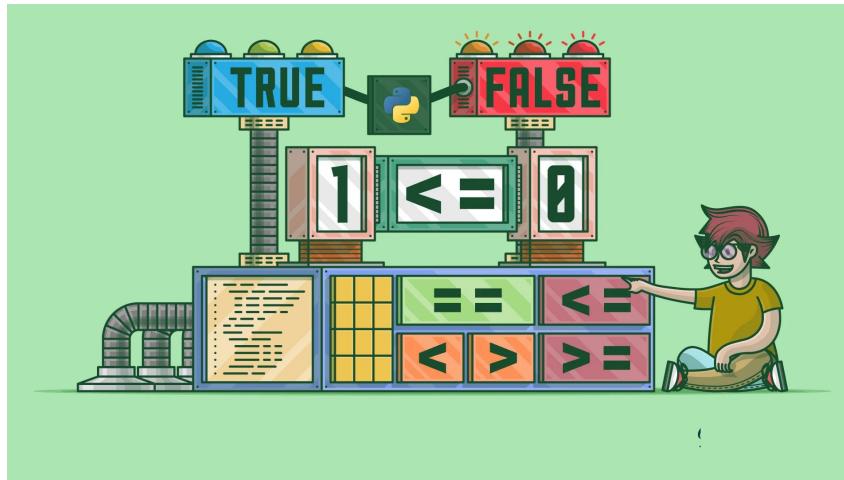
```
1 let sentence = 'JavaScript is awesome!';
2
3 console.log(sentence.indexOf('is'));
// 11 (index where 'is' starts)
4 console.log(sentence.indexOf('awesome'));
// 15 (index where 'awesome' starts)
5 console.log(sentence.indexOf('Python'));
// -1 (not found)
6 console.log(sentence.indexOf('Java', 5));
// -1 (search starts from index 5)
```

Last one returns `-1` since Java word is not available after 5th index.

Unraveling the Basics: boolean, null, and undefined

boolean

Think of it as a switch—only two states: true or false.



```
1 let isSunny = true;
2 let isRaining = false;
```

Assigning boolean values

null

This represents "nothing" or "empty". You can assign it intentionally when a variable is meant to have no value.



```
1 let car = null;  
2 // This variable is explicitly set to have no value  
3  
4 console.log(car);  
5 // Output: null
```

undefined

In JavaScript, **undefined** means a variable has been declared but has not been assigned a value yet. It also represents the default value of uninitialized variables.



undefined



defined

```
1 // Undefined example
2 let userAge;
3 console.log("userAge:", userAge);
4 // Output: undefined
5 // (variable is declared but not assigned a value)
6
7 // Defined example
8 let userName = "John Doe";
9 console.log("userName:", userName);
10 // Output: "John Doe"
11 // (variable is assigned a string value)
```

undefined vs defined

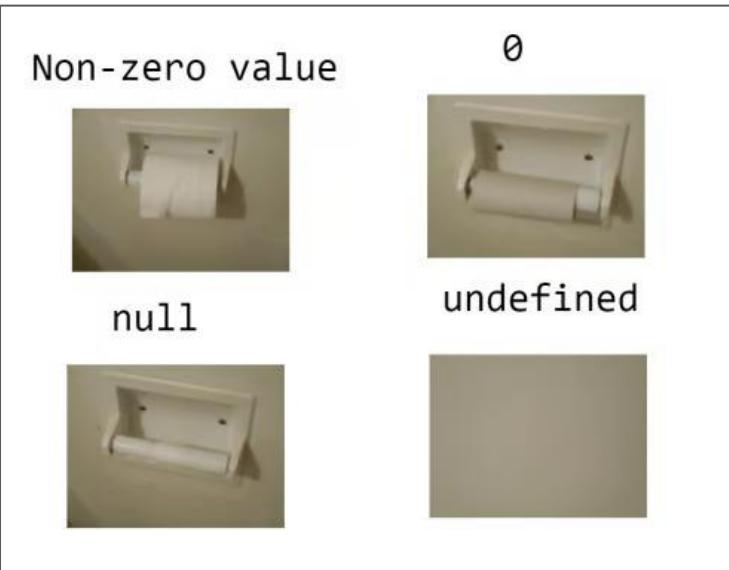
Any difference in null & undefined??

Even though they look similar but they are not same at all. You have fallen in same trap like everyone else:-



Understanding differences

Let's understand difference between zero, null and undefined



undefined	null
has not been assigned	could be assigned
typeof undefined	typeof object

Truthy and Falsy values

Any value which is not falsy is considered true.

Falsy Expressions
False
NaN
Undefined
Null
Empty String (“”/ ”)
0

Apart from these values all the values are considered true.

How to check truthy/falsy values??

We can check truthy/falsy values using Boolean function.

```
1 let value = "Hello";
2 // Example truthy value
3
4 console.log(Boolean(value));
5 // Output: true
6
7 let anotherValue = "";
8 // Example falsy value
9 console.log(Boolean(anotherValue));
10 // Output: false
```

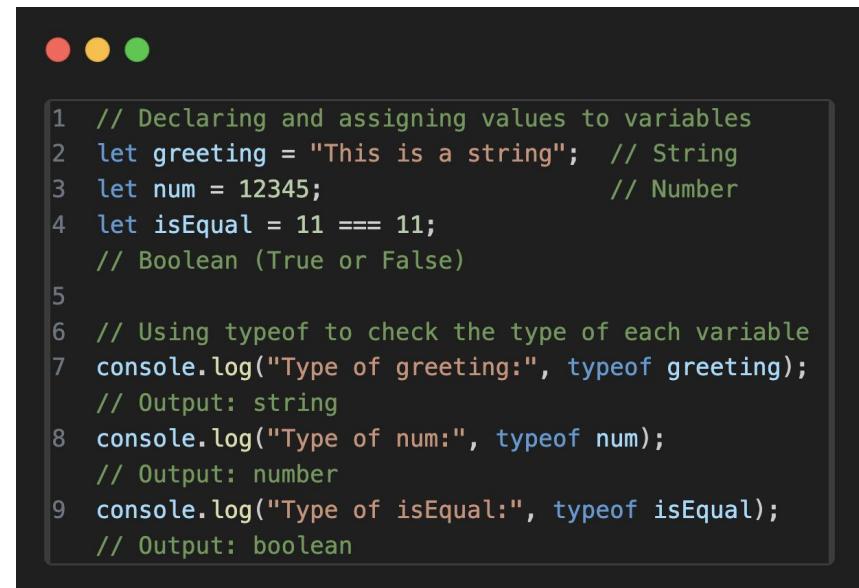
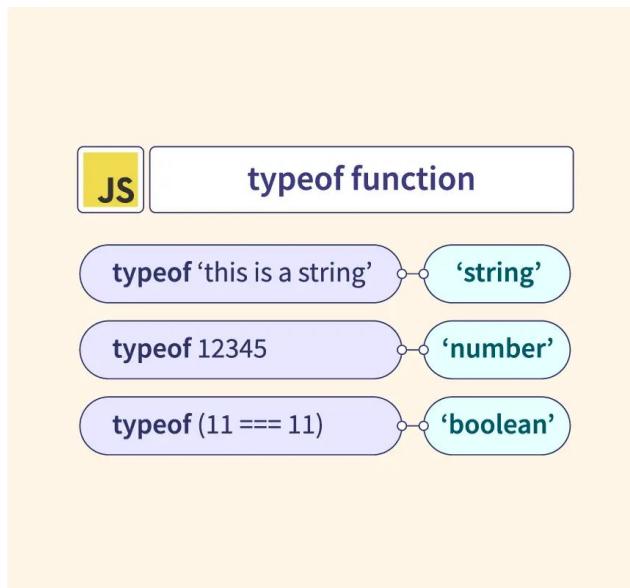
Here we used:-

1. "Hello": Evaluated as true
2. "": Evaluated as false

Try few other examples by your own.

How to check data type: `typeof`

The `typeof` operator is used to determine the type of a given variable or value. It returns a string indicating the type



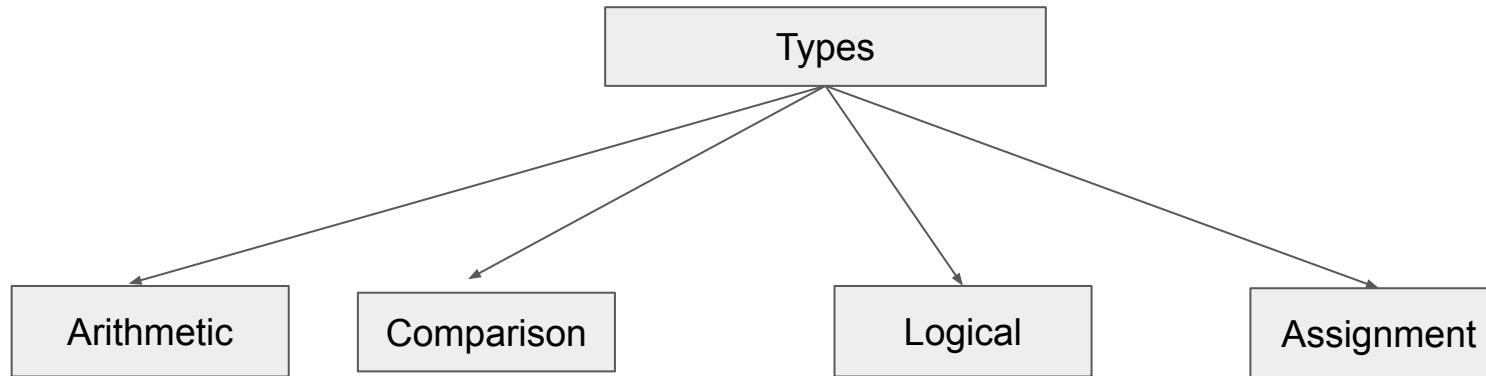
```
● ● ●

1 // Declaring and assigning values to variables
2 let greeting = "This is a string"; // String
3 let num = 12345; // Number
4 let isEqual = 11 === 11; // Boolean (True or False)
5
6 // Using typeof to check the type of each variable
7 console.log("Type of greeting:", typeof greeting);
// Output: string
8 console.log("Type of num:", typeof num);
// Output: number
9 console.log("Type of isEqual:", typeof isEqual);
// Output: boolean
```

Operators

What are operators?

In JavaScript, **operators** are special symbols used to perform operations on values



Arithmetic Operators

Arithmetic operators in JavaScript are used to perform mathematical operations on numbers.

Arithmetic Operator	Name	Example
+	Addition	$a + b$
-	Subtraction	$a - b$
*	Multiplication	$a * b$
/	Division	a / b
%	Modulus	$a \% b$
++	Increment Operator	$a++$
--	Decrement Operator	$a--$

Arithmetic Operators: Example

Here's are few examples of arithmetic operations in JavaScript:

```
● ● ●

1 // Declaring two variables
2 let num1 = 10; // First number
3 let num2 = 5; // Second number
4
5 // Addition
6 let addition = num1 + num2;
7 console.log("Addition: num1 + num2 =", addition);
8 // Output: 15
9
10 // Subtraction
11 let subtraction = num1 - num2;
12 console.log("Subtraction: num1 - num2 =", subtraction);
13 // Output: 5
14
15 // Multiplication
16 let multiplication = num1 * num2;
17 console.log("Multiplication: num1 * num2 =", multiplication);
18 // Output: 50
```

When you see your younger sibling
struggling with basic algebra



Unexpected results we get sometimes

There are few anomalies which we face due to impermissible operations. Let's understand with code:-

```
1 // 1. Division by zero = Infinity
2 console.log("1 / 0 =", 1 / 0);
3 // Output: Infinity
4
5 // 2. Number + String = string concatenation
6 console.log("1 + '2' =", 1 + '2');
7 // Output: '12'
8
9 // 3. String - String = numeric subtraction
10 console.log("'2' - '2' =", '2' - '2');
11 // Output: 0
```

Here you can observe that not all mathematical operations go as expected.

Data type of infinity

In previous slide you can observe that divide by zero provides infinity, but if infinity is a number. Want to check? Use isNaN().

```
1 // Checking if value is a number using isNaN
2 let value1 = Infinity;
3 let value2 = -Infinity;
4 let value3 = 42;
5 let value4 = "Hello";
6
7 // Using isNaN() to check if it's a valid number
8 console.log(isNaN(value1));
// false (Infinity is a number)
9 console.log(isNaN(value2));
// false (-Infinity is a number)
10 console.log(isNaN(value3));
// false (42 is a number)
11 console.log(isNaN(value4));
// true (string is not a number)
```

You can observe infinity
is a number.

Comparison Operators

Comparison operators in JavaScript are used to compare two values and return a Boolean value (**true** or **false**), helping to evaluate relationships between

>	Greater Than
>=	Greater Than or Equal To
<	Less Than
<=	Less Than or Equal To
==	Equal To
===	Strict Equal To
!=	Not Equal To
!==	Strict Not Equal To



Comparison Operators: example

Here's a simple code example to demonstrate the use of comparison operators in JavaScript:

```
1 // Declaring two variables
2 let x = 10; // First variable
3 let y = 5; // Second variable
4
5 // Equality comparison (==)
6 console.log("x == y:", x == y);
// Output: false (x is not equal to y)
7
8 // Strict equality comparison ( === )
9 console.log("x === y:", x === y);
// Output: false (x is not strictly equal to y)
10
11 // Inequality comparison (!=)
12 console.log("x != y:", x != y);
// Output: true (x is not equal to y)
```

Pay caution while choosing '`==`' over '`===`' as they have different outcomes sometimes.

Comparison Operators: example

Continued...

```
1 // Strict inequality comparison (!==)
2 console.log("x !== y:", x !== y);
  // Output: true (x is not strictly equal to y)
3
4 // Greater than comparison (>)
5 console.log("x > y:", x > y);
  // Output: true (x is greater than y)
6
7 // Less than comparison (<)
8 console.log("x < y:", x < y);
  // Output: false (x is not less than y)
9
10 // Greater than or equal to (>=)
11 console.log("x >= y:", x >= y);
  // Output: true (x is greater than or equal to y)
12
13 // Less than or equal to (<=)
14 console.log("x <= y:", x <= y);
  // Output: false (x is not less than or equal to y)
15
```

The correct placement of `<` or `>` before `=` is important for clarity.

Incorrect:- `=<`
Correct: `<=`

Logical Operators

Logical operators in JavaScript are used to perform logical operations on values, typically boolean values (`true` or `false`).

AND, OR, and NOT

Operator	Name	Example	Result
<code>&&</code>	AND	<code>x < 10 && x !== 5</code>	<code>false</code>
<code> </code>	OR	<code>y > 9 x === 5</code>	<code>true</code>
<code>!</code>	NOT	<code>!(x === y)</code>	<code>true</code>

Now how are you feeling??

You must be having love-hate relationship now. Don't worry, you will feel at home more you practice:-



Some Real World Examples

Calculating Discounts and Special Offers

When shopping, it's crucial to calculate the best price after discounts. For example, if an item costs \$100 and has a 20% discount.

```
1 // Original price of the item
2 let originalPrice = 100;
3
4 // Discount percentage
5 let discountPercentage = 20;
6
7 // Calculate discount amount
8 let discountAmount = (originalPrice * discountPercentage) /
100;
9
10 // Calculate final price after discount
11 let finalPrice = originalPrice - discountAmount;
12
13 console.log("Original Price: $", originalPrice);
// Output: 100
14 console.log("Discount Amount: $", discountAmount);
// Output: 20
15 console.log("Final Price After Discount: $", finalPrice);
// Output: 80
```

Using arithmetic operators to determine the discount amount and the final price.

Job Offers: Making Right Choice

To choose the better job offer, you can use comparison operators to evaluate salaries.

```
1 // Salary offers from two job companies
2 let offerA = 60000; // Job offer A salary
3 let offerB = 75000; // Job offer B salary
4
5 // Compare salaries using comparison operators
6 let isOfferAGreater = offerA > offerB;
7 // Checks if offer A is greater than offer B
8 let isOfferBGreater = offerB > offerA;
9 // Checks if offer B is greater than offer A
10
11 // Display the best offer
12 if (offerA > offerB) {
13   console.log("Choose Offer A for a higher salary.");
14 } else if (offerB > offerA) {
15   console.log("Choose Offer B for a higher salary.");
16 } else {
17   console.log("Both offers are the same.");
18 }
```

Using a greater-than and smaller than to help you quickly see which offer provides a higher salary.

Practice! Practice! And become boss

More you practice more you find easy way outs and more you will become confident in writing javascript code. Practice is the key...

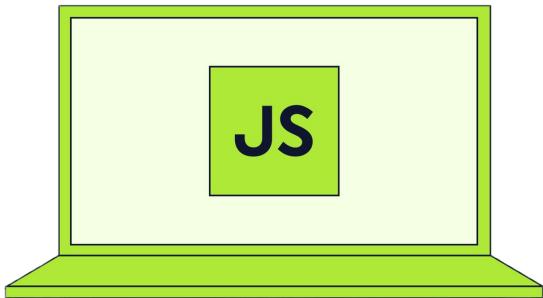


In Class Questions

**Thanks
for
watching!**



The Complete Javascript Course



@newtonschool

Lecture 3: conditionals and loops

-Vishal Sharma



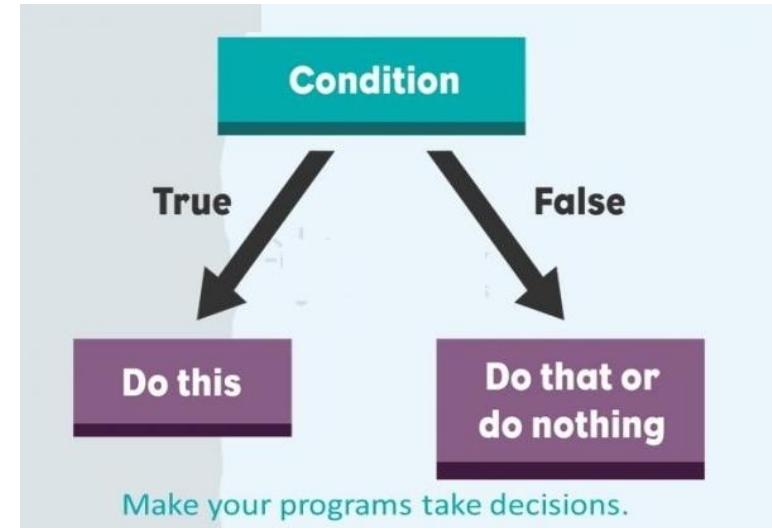
Table of Contents

- Conditional Statements
- Ternary Operator
- Assignment Operator
- Loops
- Loops control statements
- Arrays and Objects
- Practical examples: basic examples with conditionals and loops

Conditional Statements in Javascript

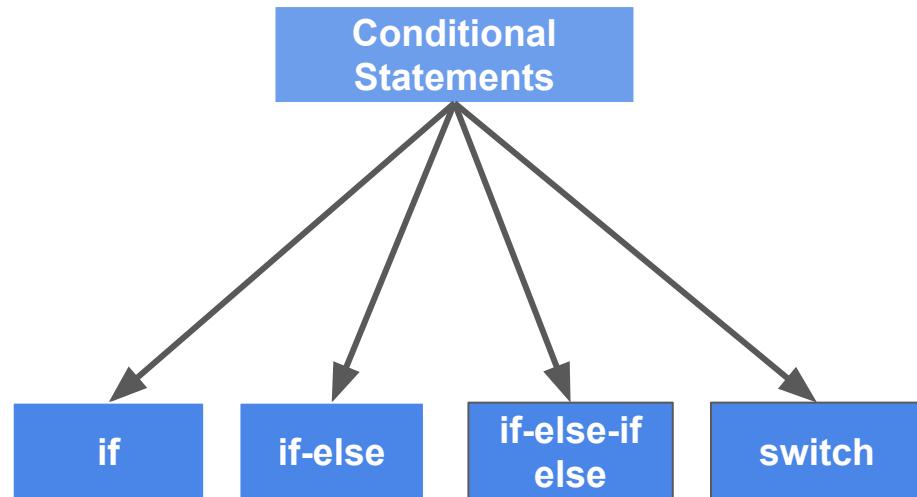
What are conditional statements?

Conditional statements help determine the flow of execution in a program based on specific conditions.



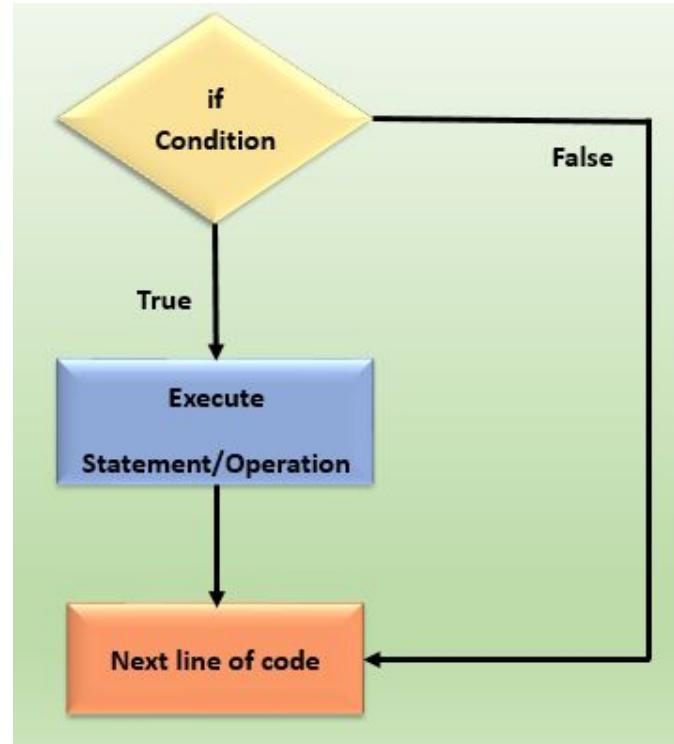
Types of conditional statements

Here are the main types of conditional statements:-



if statement

if statement is the most basic form of conditional statement. Body of *if* statement runs if test condition is true.



if statement: Syntax

Let's have a look at if statement syntax.



```
1 if(condition){  
2     // Code to execute if condition is true  
3 }
```

if statement: Example

Let's understand with an example:-

```
● ● ●

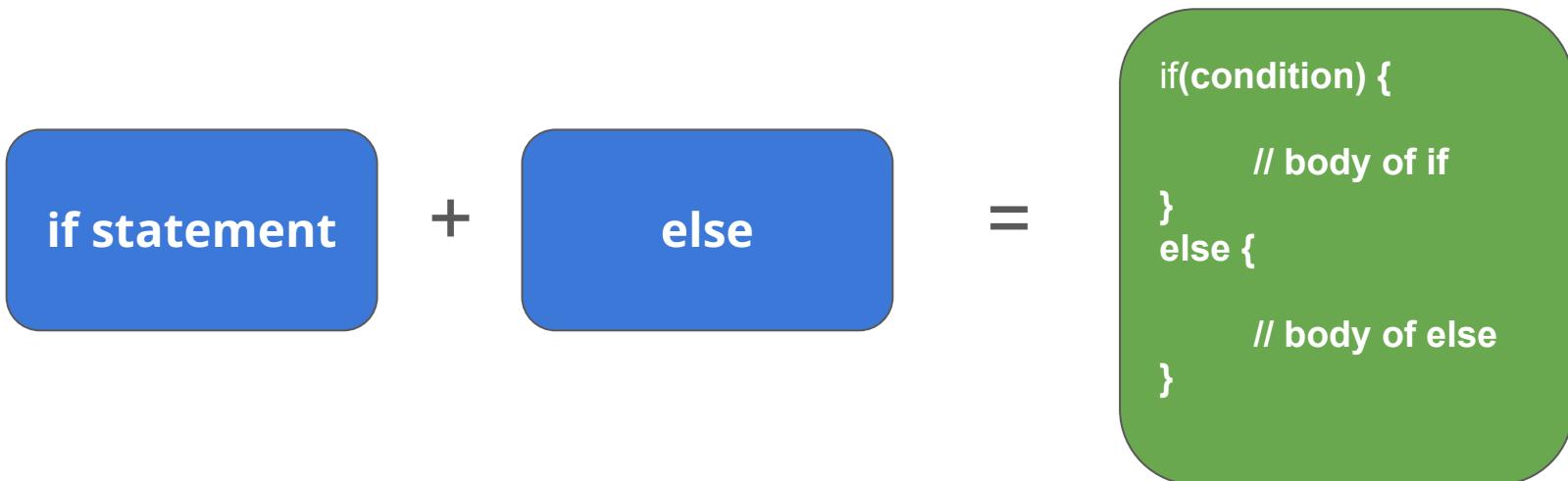
1 // Declaring and assigning the age
2 let age = 19;
3
4 // Checking the condition
5 if (age > 18) {
6     // Executing the code
7     console.log("I am an adult.");
8 }
```



I am an adult.

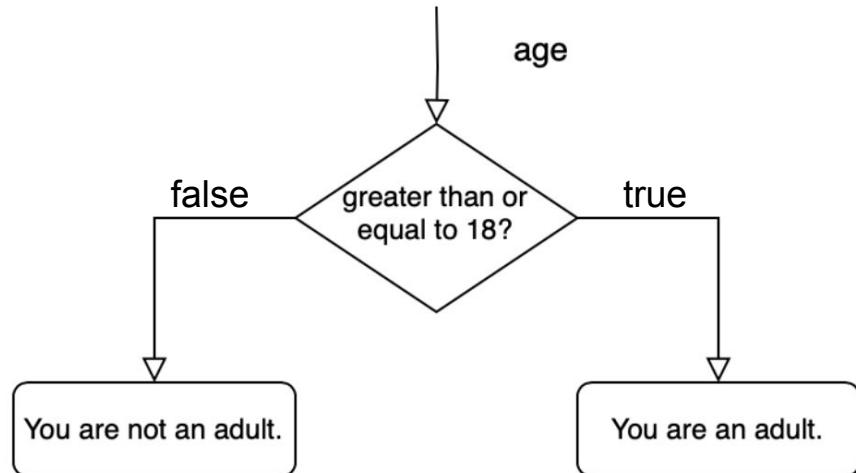
if-else: Building on if statement

If you understand the `if` statement, adding the `else` part is straightforward. It simply provides an alternative action when the condition is false.



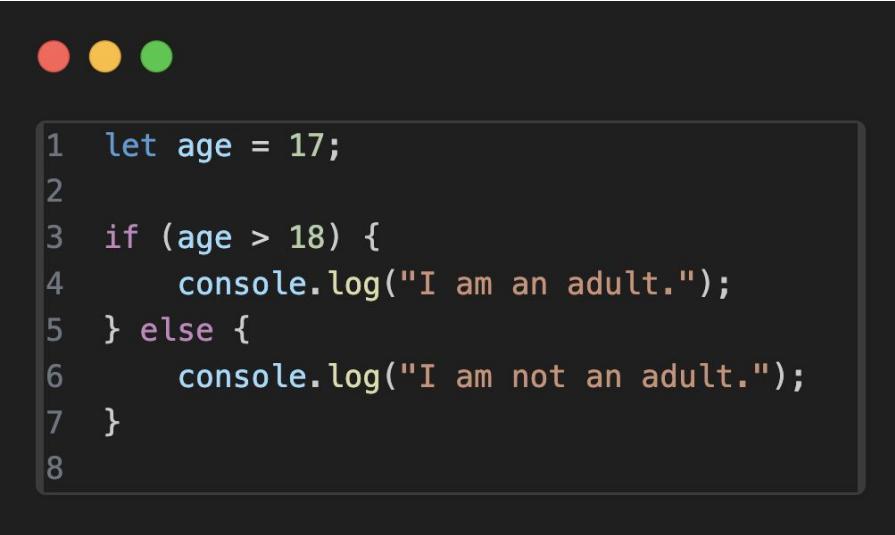
if-else: Building on if statement

In the previous example, we checked if a person is under 18 using only an **if** statement." Modify the code to include an **else** condition that prints "You are not eligible." when the person age is less than 18 years.



if-else: Example

Here age is less than 18 so else statement is executed:-

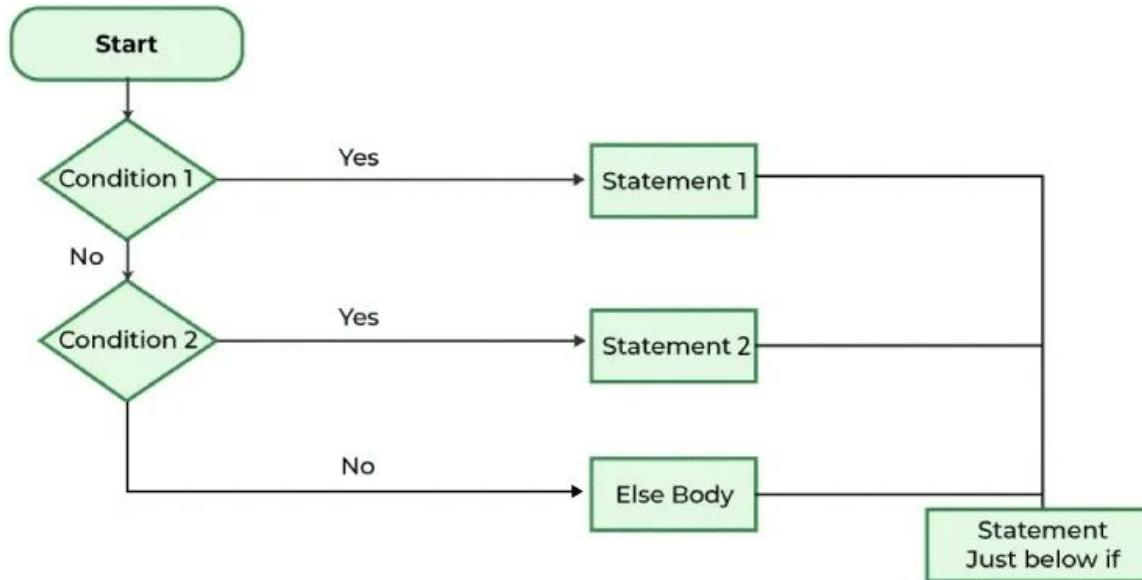


```
1 let age = 17;
2
3 if (age > 18) {
4     console.log("I am an adult.");
5 } else {
6     console.log("I am not an adult.");
7 }
8
```



Going beyond: if-else-if-else ladder

The if-else-if-else ladder helps the program check multiple conditions one by one and take an action based on the first condition that's true.



Going beyond: if-else-if-else ladder

Think of it like making plans: Plan A, then Plan B if A fails, then Plan C, and only facing the worst if all fail. The if-else-if-else ladder works the same way.

Stacking **if/else** statements be like

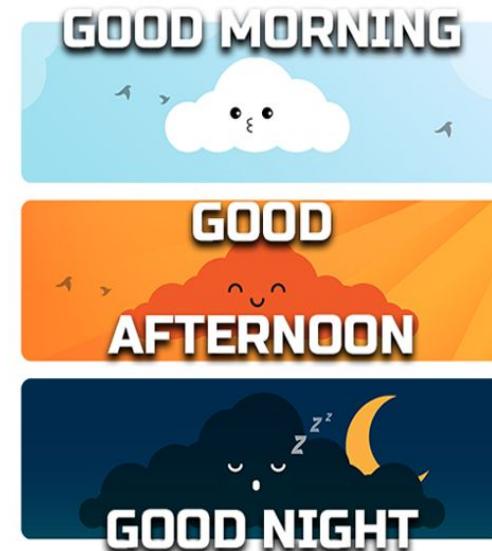


If-else-if-else: example

Let's write code for delivering salutations at different parts of day:-

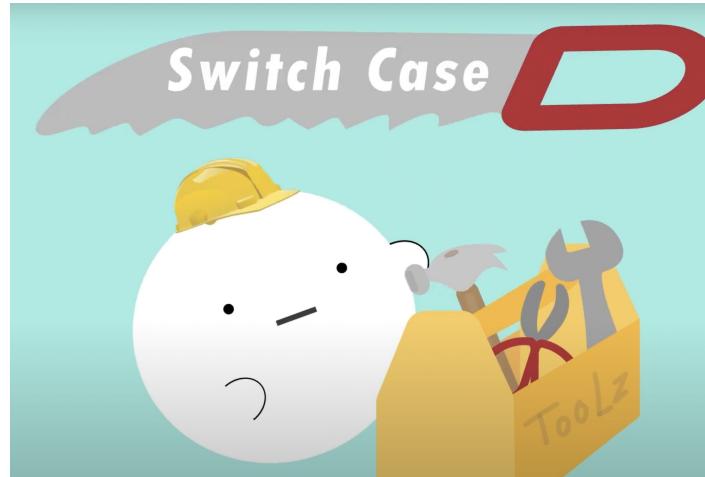


```
1 let hour = 15;
  // Time in 24-hour format
2
3 if (hour >= 5 && hour < 12) {
4     console.log("Good Morning!");
5 } else if (hour >= 12 && hour < 18) {
6     console.log("Good Afternoon!");
7 } else {
8     console.log("Good Evening!");
9 }
```



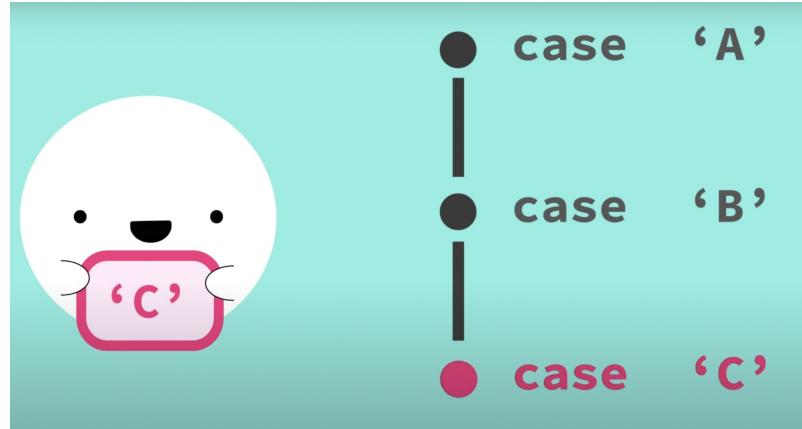
switch statement

switch statement is one among many tools for condition based execution of statements.



switch statement

It selects one among many cases whichever matches with value passed inside switch.



switch statement: syntax

Let's have a look at its syntax:-



```
switch(variable) {  
    case 1:  
        // execute your code  
        break;  
    case n:  
        // execute your code  
        break;  
    default:  
        // execute your code  
        break;  
}
```

We pass a variable inside switch and whichever case matches that variable value gets executed.

switch statement: example

Here `switch` checks `fruit` against each `case`. If no match is found, the `default` block runs as a fallback.

```
1 let fruit = "apple";
2
3 switch (fruit) {
4     case "apple":
5         console.log("It's an apple!");
6         break;
7     case "banana":
8         console.log("It's a banana!");
9         break;
10    default:
11        console.log("Unknown fruit!");
12 }
```

Since case 'apple' matches the fruit variable, "This is an apple." gets printed.

But what is the break statement just after every test case end??

switch statement: break

In this example we have added break after every test case. Do you know why?



```
1 let fruit = "apple";
2
3 switch (fruit) {
4     case "apple":
5         console.log("It's an apple!");
6         break;
7     case "banana":
8         console.log("It's a banana!");
9         break;
10    default:
11        console.log("Unknown fruit!");
12 }
```

Whenever a case matches statements under it run and to avoid other cases to run we break its execution by just adding break at the end of it.

Ternary Operator: A Shortcut for If-Else

The ternary operator is a concise way to write an if-else statement in a single line. It evaluates a condition and returns one value if true, and another if false.

condition

? code to execute if true

: code to execute if false

Ternary Operator: example

The ternary operator simplifies if-else code when only one condition is involved.

```
1 let age = 18;  
2 let category = (age >= 18) ? "Adult" : "Minor";  
3 console.log(category); // Output: Adult
```

Ternary Operator: Comparison with if..else

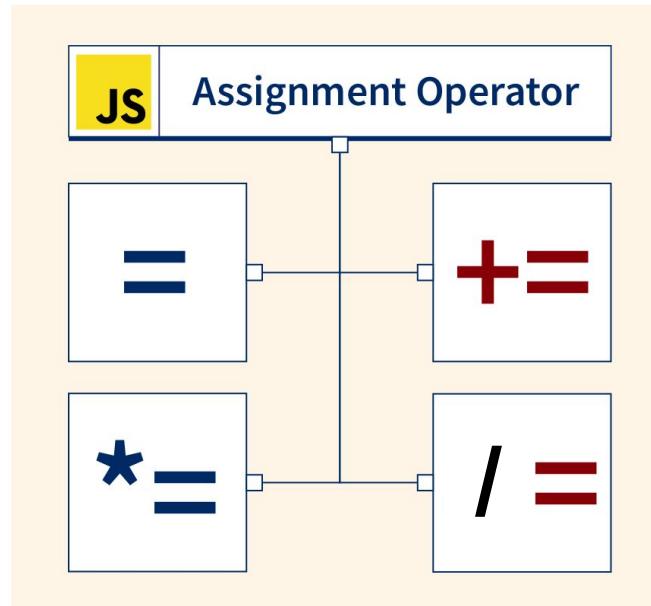
Compared to if-else, the ternary operator offers smoother flow, improving readability and understanding.

```
1 let age = 18;
2 let category;
3
4 if (age >= 18) {
5     category = "Adult";
6 } else {
7     category = "Minor";
8 }
9
10 console.log(category);
11 // Output: Adult
```

Now, after comparison you will appreciate ternary operator.

Assignment Operators

Assignment operators are used to assign values to variables. The basic assignment operator is `=`, but there are also shorthand operators like `+=`, `-=`, `*=` etc.



Usually we use shorthand notations in writing javascript, it saves time and energy

Assignment Operators: Example

Let's have a look at an example:-

```
1 // Basic Assignment
2 let x = 10;
3 console.log("Basic Assignment: x =", x); // Output: 10
4
5 // Addition Assignment (x += y)
6 x += 5; // Equivalent to x = x + 5
7 console.log("Addition Assignment: x += 5 =", x);
// Output: 15
8
9 // Subtraction Assignment (x -= y)
10 x -= 3; // Equivalent to x = x - 3
11 console.log("Subtraction Assignment: x -= 3 =", x);
// Output: 12
12
13 // Multiplication Assignment (x *= y)
14 x *= 2; // Equivalent to x = x * 2
15 console.log("Multiplication Assignment: x *= 2 =", x);
// Output: 24
```

First one is the basic assignment and rest of it are shorthands.

We need not to write shorthands necessarily, but it saves time

Loops in real life

Every day we follow certain routine which we repeat all over the week, like brushing your teeth, taking shower etc.



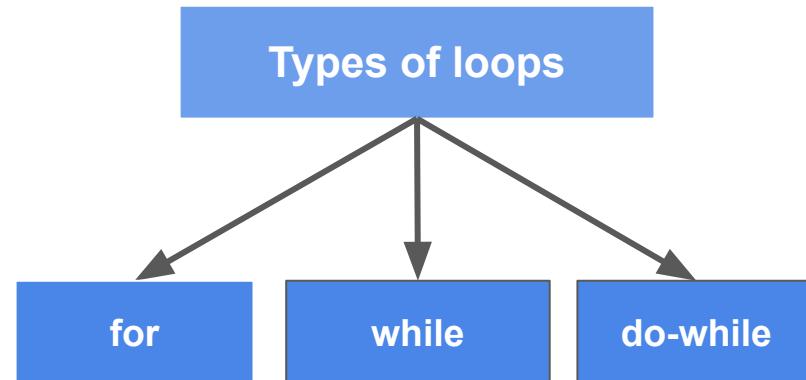
Loops in real life

We wake up, go for a walk, take a shower, then take lunch and so on. And next day we repeat the same process.



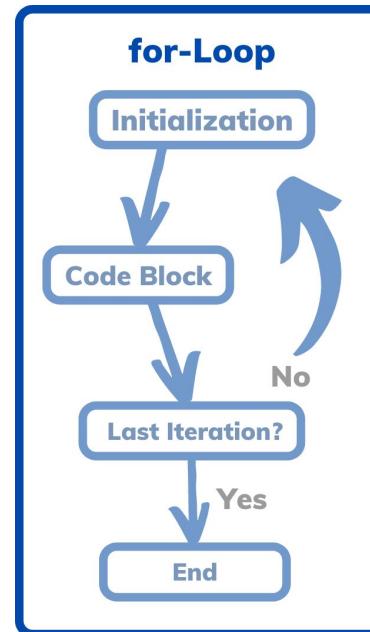
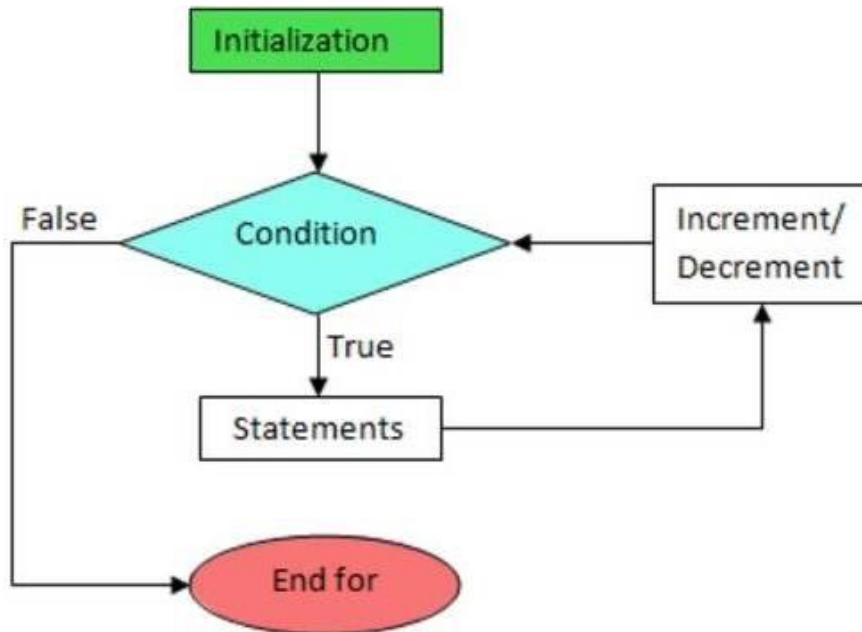
Loops in Programming

Just like we repeat several tasks in daily life, we need some tasks to be repeated in programming and we repeat those tasks with the help of loops.



for loop

A for loop in JavaScript repeats a block of code a set number of times, typically when the number of iterations is known in advance.



for loop: syntax

for loop has three parts, initial state, increment/decrement and end condition.

```
for ( let i = 5 ; i <= 10 ; i + + )
```

Initialization

Test
Condition

Updation

for loop: example

Let's see an example:-

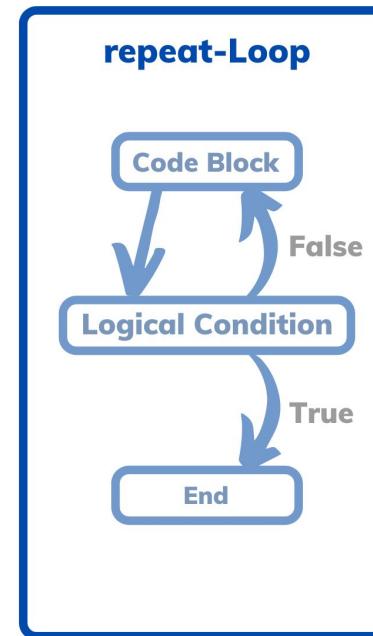
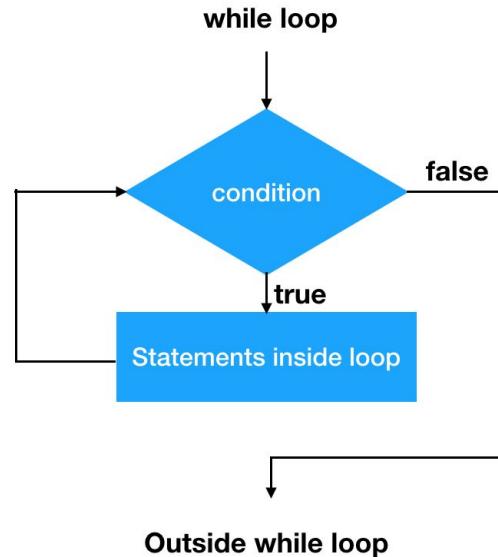


```
1  for (let i = 1; i <= 5; i++) {  
2      console.log("Iteration Number: ", i);  
3  }  
4  // Output  
5  // Iteration Number: 1  
6  // Iteration Number: 2  
7  // ...
```

Here console.log statement is printed 5 times as loop runs for exact 5 times.

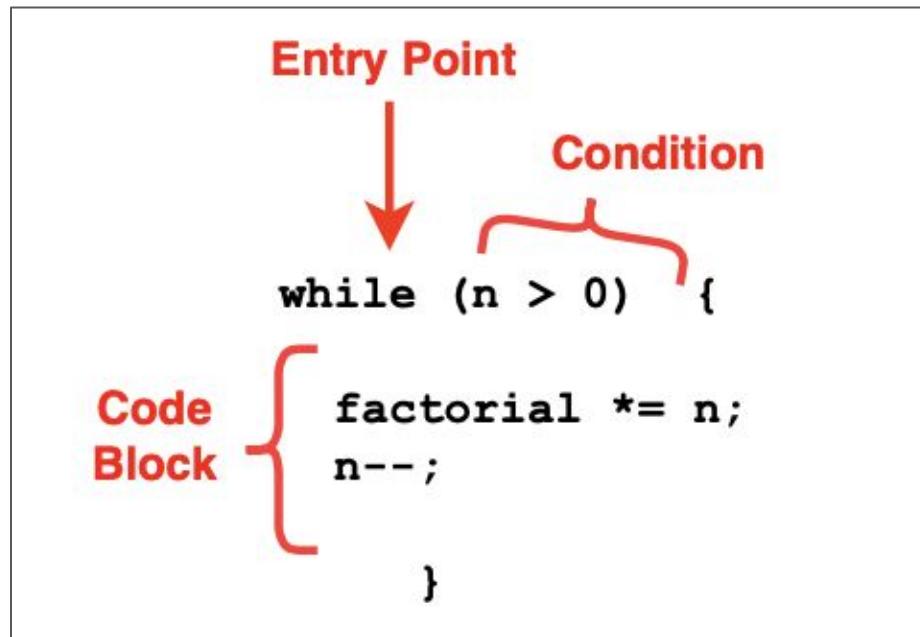
while loop

A while loop in JavaScript repeatedly executes a block of code as long as the condition remains **true**, making it ideal when the number of iterations depends on dynamic conditions.



while loop: syntax

for loop has just end condition, increment/decrement is done in body and initialization is done before the loop starts.



while loop: example

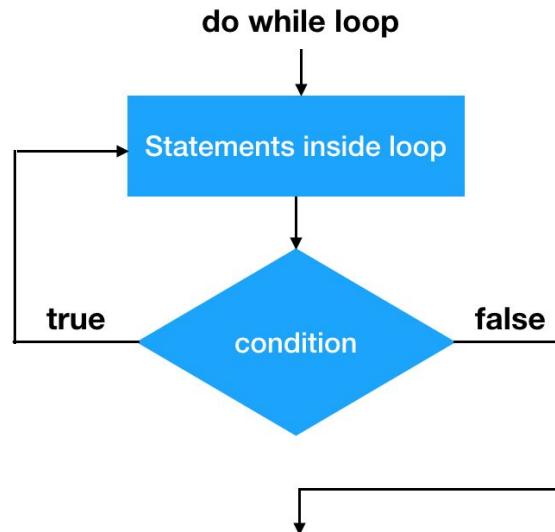
Let's see an example:-

```
1 let i = 1;
2 while (i <= 5) {
3     console.log("Iteration Number: ", i);
4     i++;
5 }
6 // Output
7 // Iteration Number: 1
8 // Iteration Number: 2
9 // ...
```

Here `console.log` statement is printed 5 times as loop runs for exact 5 times. But we did increment/decrement inside body and initialization before the loop

do-while loop

A do-while loop is a control flow statement that ensures a block of code runs at least once, regardless of the condition.



It runs at least once because at start body is run before actually checking the test condition but only for the first time.

do-while loop: syntax

Let's have a look at it's syntax:-

**Construction for
do...while loop**

```
do
{
    Console.WriteLine("I am a do...while loop.");
}
while (...)
```

**Body of
the loop**

Condition

It is same as while loop
with only difference that
condition is checked at
the end.

do-while loop: example

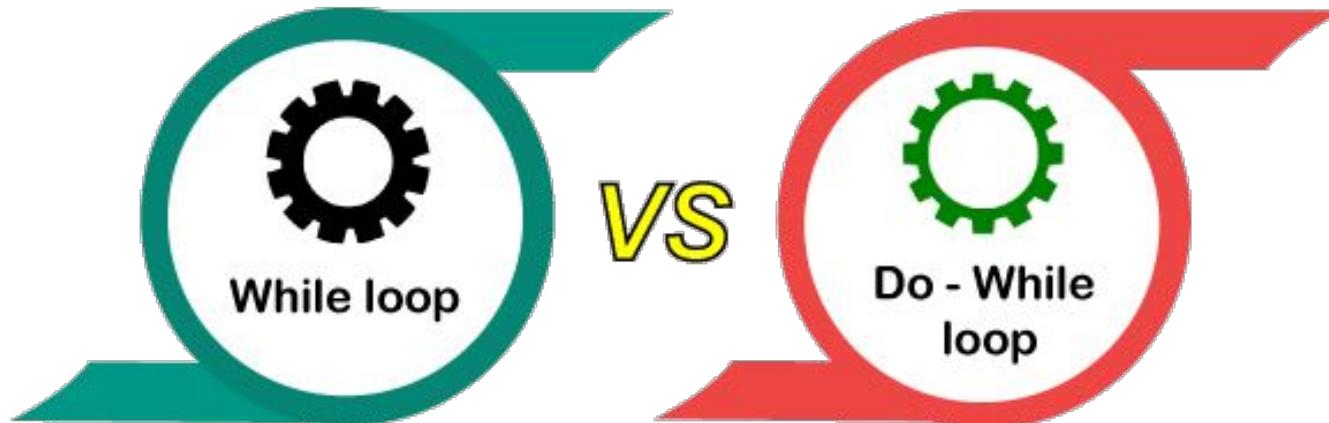
Let's see an example:-

```
1  let i = 1;
2
3  do {
4      console.log("this is iteration number: ", i);
5      i++; // Increment to move to next iteration
6  } while (i <=5 );
7
8 // Output:
9 // This is iteration number: 1
10 // Tjhis iteration number: 2
11 // .....
```

Here condition got checked at the end of the loop.

Difference: while and do-while loop

The while loop checks the condition before execution, potentially skipping the loop entirely if false. The do-while loop runs at least once before checking the condition.



Difference: while loop

Here while loop will not run at all since test condition fails before first iteration.



```
1 let i = 6;
2
3 while (i <= 5) {
4     console.log("This will not run.");
5     i++;
6 }
7
8 // Output
9 // Nothing would appear as while loop didn't run
```

Difference: do-while loop

Here do-while loop will run since body gets executed before it can run test condition for the first time.

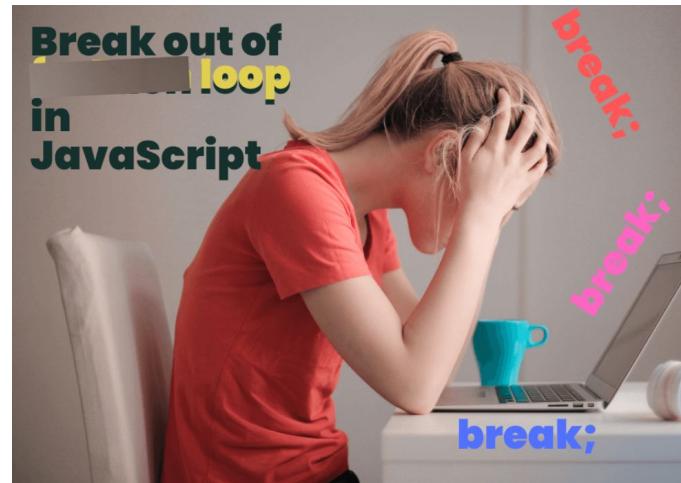


```
1  do {  
2      console.log("This will run once.");  
3      i++;  
4  } while (i <= 5);  
5  
6 // Output  
7 // This will run once
```

Control Statements: break and continue

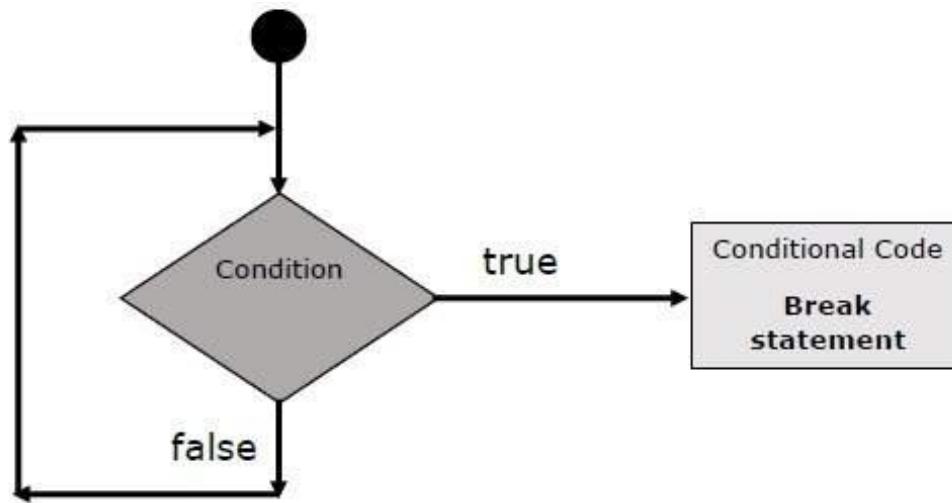
Control statement: break

The `break` statement stops a loop or switch case from running further, allowing you to exit early when a condition is met.



Control statement: break

When the specified condition is met, the **break** statement immediately terminates the loop and transfers control to the next statement after the loop.



Control statement: break (with)

Let's understand with example:-

```
1  for (let i = 1; i <= 5; i++) {  
2      if (i === 3) {  
3          break;  
4      }  
5      console.log(i);  
6  }
```

Here we broke out of loop when condition `i === 3` was true

Output:

```
1  
2
```

Control statement: break (without)

Outcome without break would be different



```
1  for (let i = 1; i <= 5; i++) {  
2      if (i === 3) {  
3          // No break here  
4      }  
5      console.log(i);  
6  }  
7
```

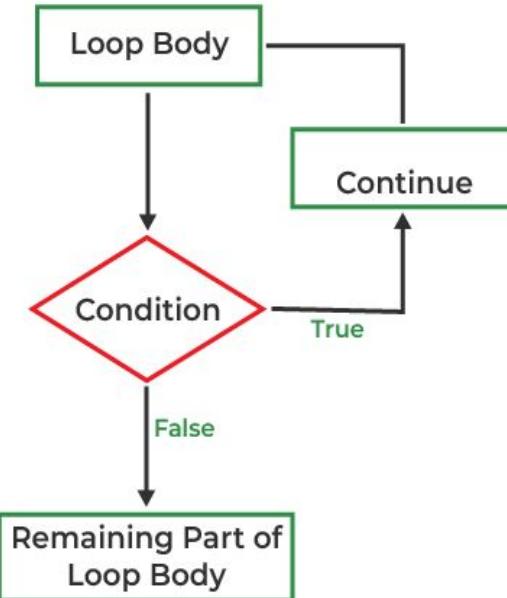
Since there is no break statement all the values got printed

Output:

```
1  
2  
3  
4  
5
```

Control statement: continue

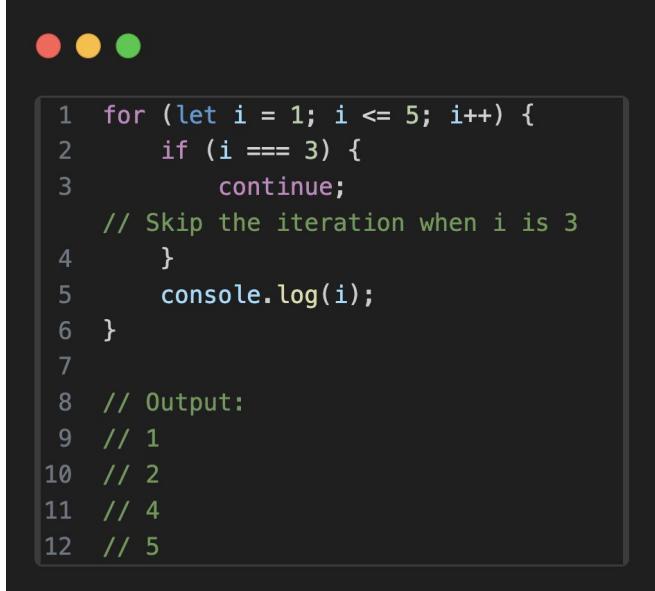
The `continue` statement skips the current iteration of a loop and moves to the next one.



Control statement: continue example

In this example, the `continue` statement skips the iteration when `i` is 3, so 3 is not printed, but the loop continues with the next values.

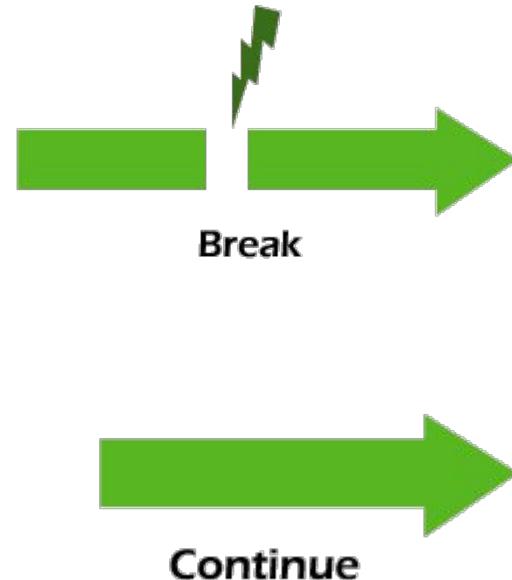
Go back in the previous slides and compare it with break statement.



```
1 for (let i = 1; i <= 5; i++) {
2     if (i === 3) {
3         continue;
4     }
5     console.log(i);
6 }
7
8 // Output:
9 // 1
10 // 2
11 // 4
12 // 5
```

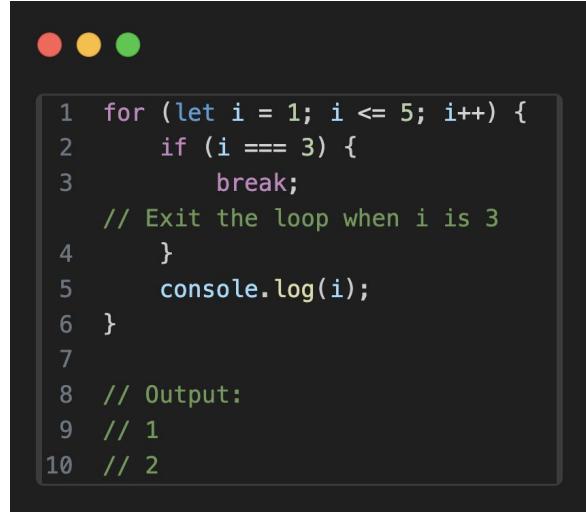
Difference: break vs continue

The `break` statement exits the loop entirely, while the `continue` statement skips the current iteration and moves to the next one.



Difference: break vs continue

Let's compare the break and continue examples: In the break example, the loop stops entirely before reaching 3, while in the continue example, only 3 is skipped, and the loop continues until 5.



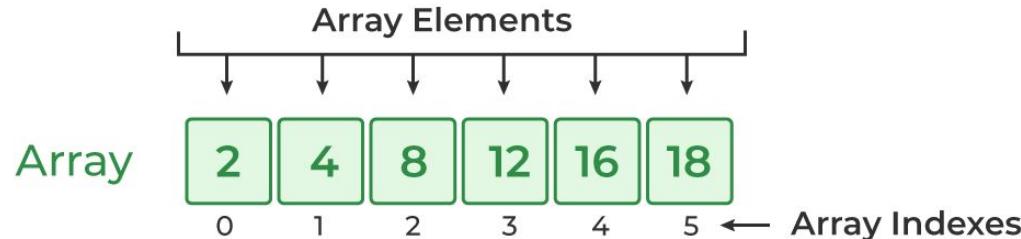
```
1 for (let i = 1; i <= 5; i++) {  
2     if (i === 3) {  
3         break;  
4     }  
5     console.log(i);  
6 }  
7  
8 // Output:  
9 // 1  
10 // 2
```

break

Arrays and Objects

What are arrays?

Arrays are one of the best data structures for storing multiple values in a single variable. Loops, such as `for`, `while`, or `do-while` are commonly used with arrays to process or manipulate their elements efficiently.



Using array with for loop

Let's use arrays along with loop. Here we will iterate array elements using for loop:-

```
1 // Create an array with various elements
2 const myArray = [1, 2, 3, 4, "Hello", true, null];
3
4 // Array declaration
5 const arr = [2, 4, 8, 12, 16];
6
7 // Using a for loop to iterate over the array
8 for (let i = 0; i < arr.length; i++) {
9     console.log(`Element at index ${i}: ${arr[i]}`);
10 }
```

Objects: Real world dictionary

Think of an object as a real-world dictionary, where the key is the word you're looking up (e.g., "name" or "age"), and the value is the definition or meaning of that word (e.g., "Alice" or 20).

Key	Value
name	Alice
age	20



```
1 const person = {  
2   "name": "Alice",  
3   "age": 20  
4 }
```

In Javascript

Objects: Practical Example

When you have data that involves key-value pairs, objects are the most appropriate choice. Here is an example:-

```
1 const book = {  
2     title: "The Great Gatsby",  
3     author: "F. Scott Fitzgerald",  
4     year: 1925,  
5     genre: "Fiction"  
6 };
```

Here, `title`, `author`, `year`, and `genre` are the keys (or properties) of the book object.

Storing Objects in array

You can store several objects inside an array too:-



```
1 const books = [
2   { title: "The Great Gatsby", year: 1925 },
3   { title: "To Kill a Mockingbird", year: 1960 },
4   { title: "1984", year: 1949 },
5   { title: "Moby-Dick", year: 1851 },
6   { title: "Pride and Prejudice", year: 1813 }
7 ];
```

We often store multiple objects in an array is to keep things organized.

Object Array: Iterating using for loop

And then iterate over them using for loop:-



```
1 const books = [
2   { title: "The Great Gatsby", year: 1925 },
3   { title: "To Kill a Mockingbird", year: 1960 },
4   { title: "1984", year: 1949 },
5   { title: "Moby-Dick", year: 1851 },
6   { title: "Pride and Prejudice", year: 1813 }
7 ];
8
9 for (let i = 0; i < books.length; i++) {
10   console.log(`Title: ${books[i].title}, Year: ${books[i].year}`);
11 }
```

Object Array: Iterating using for...in loop

It would be lot simpler if we use for...in loop instead:-



```
1 const books = [
2   { title: "The Great Gatsby", year: 1925 },
3   { title: "To Kill a Mockingbird", year: 1960 },
4   { title: "1984", year: 1949 },
5   { title: "Moby-Dick", year: 1851 },
6   { title: "Pride and Prejudice", year: 1813 }
7 ];
8
9 for (let i in books) {
10   console.log(`Title: ${books[i].title}`);
11   console.log(`Year: ${books[i].year}`);
12 }
```

In for...in loop we get keys and in this case '*i*' is the key. And then we use that key to access the object in the array.

Object Array: Iterating using for...of loop

If we want to access objects directly then we can use for...of loop

```
● ● ●  
1 const books = [  
2   { title: "The Great Gatsby", year: 1925 },  
3   { title: "To Kill a Mockingbird", year: 1960 },  
4   { title: "1984", year: 1949 },  
5   { title: "Moby-Dick", year: 1851 },  
6   { title: "Pride and Prejudice", year: 1813 }  
7 ];  
8  
9 for (let book of books) {  
10   console.log(`Title: ${book.title}`);  
11   console.log(`Year: ${book.year}`);  
12 }
```

for...of loop directly provides values stored in the array.

Which iteration method would you prefer?

Object: for...in vs for...of

The main difference between them with respect to objects is that for...in is iterable over the keys of an object, whereas for...of can only iterate over an array.

```
1 // Using for...in
2 console.log('Using for...in:');
3 for (const key in obj) {
4     console.log(` ${key} : ${obj[key]}`);
5 }
6
7 // Output
8 // a : 1
9 // b : 2
10 // c : 3
```



```
1 // will throw an error if used on an object
2 console.log('\nUsing for...of:');
3 try {
4     for (const value of obj) {
5         console.log(`Value: ${value}`);
6     }
7 } catch (error) {
8     console.log('Error: for...of cannot be used');
9 }
```



Some Real World Examples

Calculating Discounts

When shopping, it's crucial to calculate the best price after discounts. For example, if an item costs \$100 and has a 20% discount, arithmetic operators can help us determine the discount amount and the final price.



```
1 let originalPrice = 100;
2 let discountPercentage = 20;
3 let discountAmount = (originalPrice * discountPercentage) / 100;
4 let finalPrice = originalPrice - discountAmount;
5
6 console.log("Original Price: $", originalPrice);
7 console.log("Discount: $", discountAmount);
8 console.log("Final Price: $", finalPrice);
```

Job Offers: Making Right Choice

To choose the better job offer, you can use comparison operators to evaluate salaries. For example, using a greater-than operator helps you quickly see which offer provides a higher salary.



```
1 let offer1Salary = 50000;
2 let noffer2Salary = 55000;
3
4 if (offer2Salary > offer1Salary) {
5     console.log("Offer 2 has a higher salary.");
6 }j else {
7     console.log("Offer 1 has a higher salary.");
8 }
```

Shopping Cart: Calculating Bill

You are creating a shopping cart where each item has a price. You want to calculate the total cost of all items in the cart.

```
1 const cart = [
2   { item: "Laptop", price: 1200 },
3   { item: "Headphones", price: 150 },
4   { item: "Mouse", price: 50 },
5   { item: "Keyboard", price: 80 }
6 ];
7
8 let totalPrice = 0;
9
10 for (let i = 0; i < cart.length; i++) {
11   totalPrice += cart[i].price;
12   // Add each item's price to totalPrice
13 }
14
15 console.log(`Total Price: ${totalPrice}`);
```

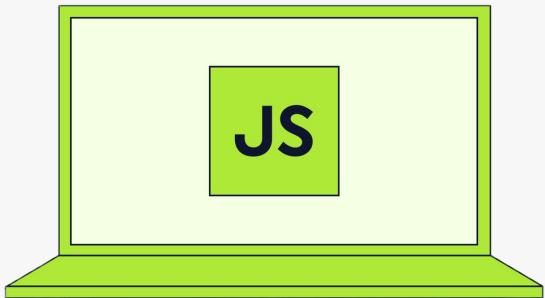
Arrays, objects, and `for` loops
(and their variations) are
commonly used together to
solve many programming
problems.

In Class Questions

**Thanks
for
watching!**



The Complete Javascript Course



@newtonschool

Lecture 4: Functions

-Vishal Sharma



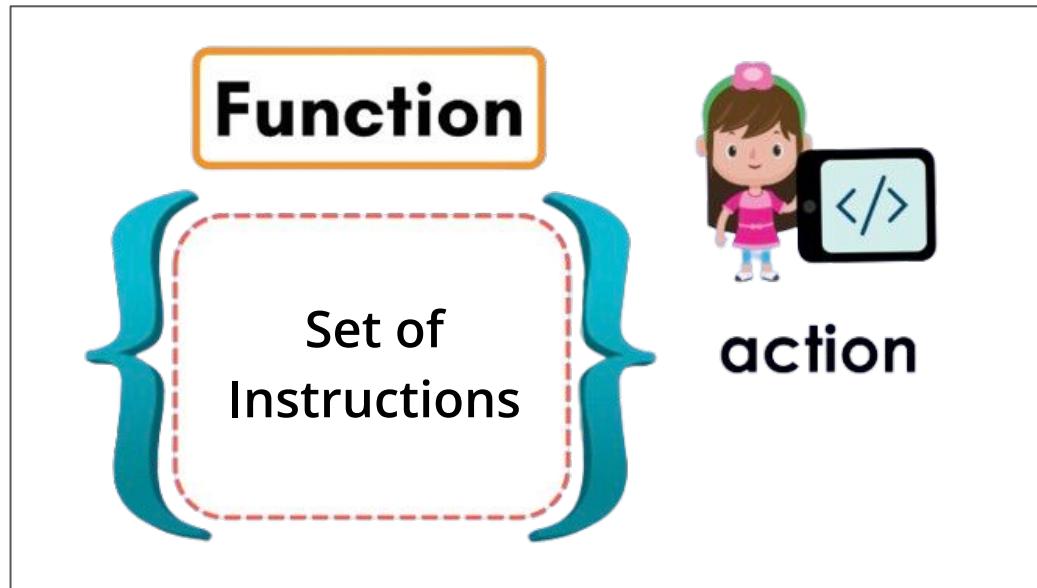
Table of Contents

- Function Declaration and Syntax
- Parameters vs Arguments
- Spread and Rest Operator
- Function Return Values
- Higher Order Functions and Callback Functions

Function Declaration and Syntax

What is a Function?

A function is a block of code made out of a set of steps that result in a single specific action.



What does it mean??
Let's understand with an
analogy

Functions: Like Tying your shoes

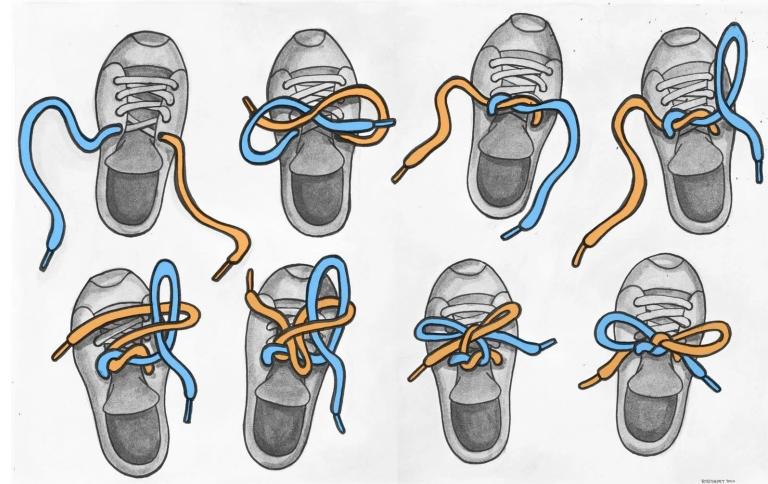
A function is like tying your shoes—once you learn how, you can repeat the steps whenever needed without rethinking the process

Functions

=

Tying your shoes

- 1** Gather laces
- 2** Knot
- 3** Loop
- 4** Swoop
- 5** Pull tight



Similarly we use functions in our programs

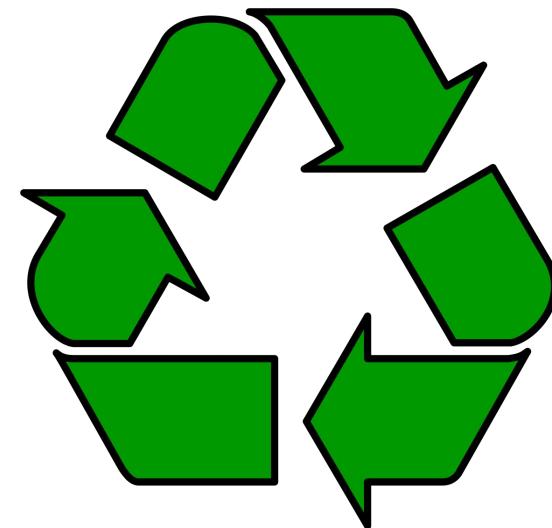
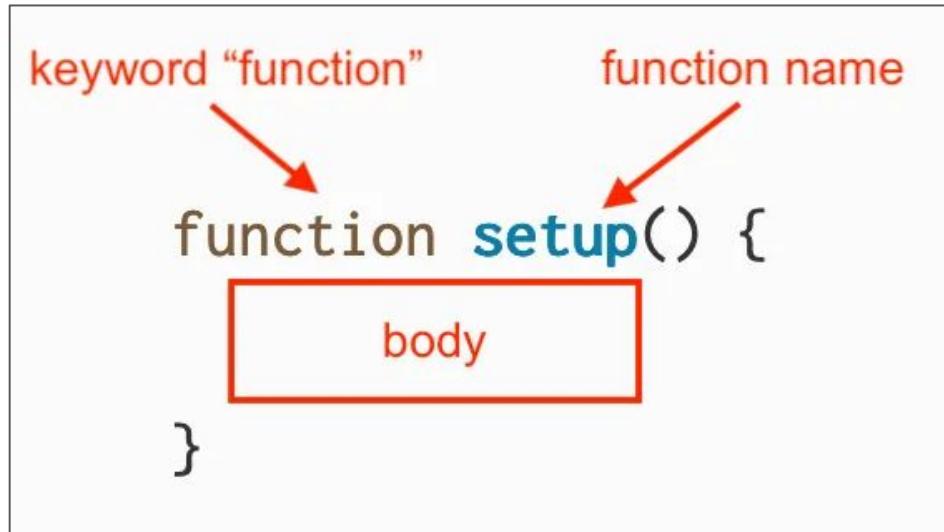
Let us define a tie-shoe function and reuse it multiple times:

```
1 // Function to tie your shoes
2 function tieShoes() {
3     console.log("Cross the laces.");
4     console.log("Make a loop with one lace.");
5     console.log("Wrap the other lace around the loop.");
6     console.log("Pull through and tighten.");
7 }
8
9 // Reusing the function multiple times
10 tieShoes(); // First time
11 tieShoes(); // Second time
```



Naming a function

A programmer can give a function a name and be used again and again.



Challenge:

Write a Function to Display Hello World

Were you able to do it??

Let's start writing:



```
1 function sayHello() {  
2     console.log("Hello, World!");  
3 }  
4  
5 sayHello();  
6 // Output: Hello, World!
```

How simple it is!!

Think of a more complex task/function

Your dad wants you to calculate your daily expenses, but you find it difficult to do manually. You are crying for help!!



Your Rescue: Write an add function

Don't worry, functions are here for rescue.

```
1 function add(){  
2     // your code here  
3 }
```

But how to tell the function
what to add??

Function: parameters

We can define parameters in a function to receive values.



Function Parameters



```
1 function add(num1, num2){  
2     // your code here  
3 }
```

Here “*num1*” and
“*num2*” are
function
parameters.

Function: using parameters to add

Let's use parameters to calculate the sum. And print it:

```
1 function add(num1, num2){  
2     let result;  
3     result = num1 + num2; ←  
4  
5     // printing the result  
6     console.log(result);  
7 }
```

Using parameters to
perform addition

Function: calling using arguments

Let's just call the function by passing values inside it:

```
1 // calling function using ar  
guments  
2 add(4, 5); // Output: 9  
3 add(6, 4); // Output: 10
```

Arguments are the values passed to a function and assigned to its parameters.

Actual values passed to function are called **arguments**

Parameters vs Arguments

Difference: parameters and arguments

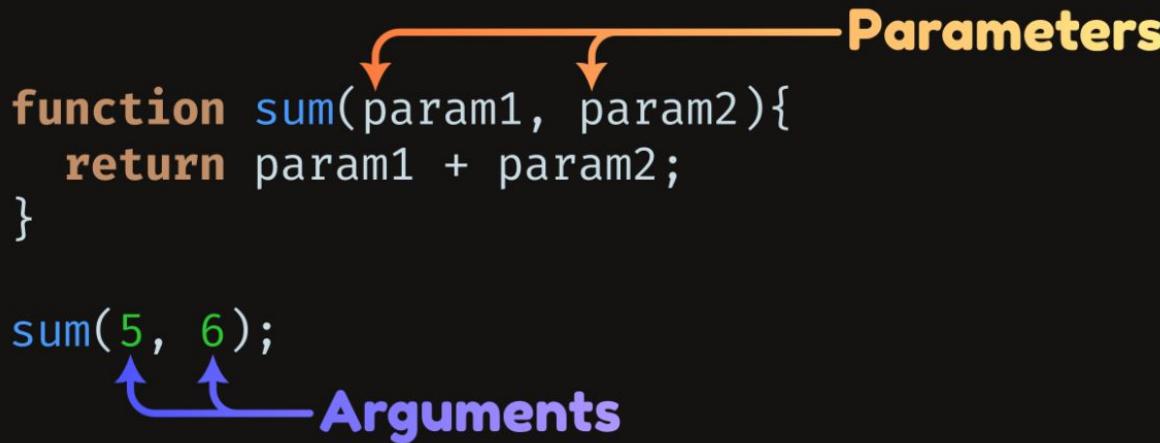
Now, can you tell me the difference between these two??

Parameters vs **Arguments**



Difference: parameters and arguments

Parameters are placeholders in a function declaration, while arguments are the actual values passed during a function call.



The diagram illustrates the difference between parameters and arguments. It shows a function declaration and a call to it, with arrows pointing from the labels to the corresponding parts of the code.

Parameters (in yellow) are labeled above the function declaration, with arrows pointing to the two placeholder variables `param1` and `param2` in the `sum` function definition.

Arguments (in blue) are labeled below the function call, with arrows pointing to the actual values `5` and `6` being passed to the function.

```
function sum(param1, param2){  
    return param1 + param2;  
}  
  
sum(5, 6);
```

Function Return Values

Function: return values

You expect someone to return something whenever you give them a task; this could be the task's status or final result, among other things.



Return Values

Function: return values

Similarly, functions do return values on completion.

return in
Javascript.....



Function: return value syntax

Similarly, functions do return values on completion.



```
1 function findSquare(num) {  
2     let result = num * num;  
3     return result;  
4 }  
5  
6 let square = findSquare(3);
```



We want to return the result value to where the function was called from.

Function: return values with Example

Let's reuse the add function again, but this time we will return values instead of printing.

```
1  function add(num1, num2){  
2      let result;  
3      result = num1 + num2;  
4  
5      // printing the result  
6      console.log(result);  
7  }
```



```
1  function add(num1, num2){  
2      let result;  
3      result = num1 + num2;  
4  
5      // returning the result  
6      return result;  
7  }
```



Function: return values with Example

Displaying returned values by calling the function and storing the result in a variable.

```
1  function add(num1, num2){  
2      let result;  
3      result = num1 + num2;  
4  
5      // returning the result  
6      return result;  
7  }  
8  
9  let sum = add(2, 3);  
10 console.log(sum); // Output: 5
```

Returning values is preferred over displaying the result inside the function.

Functions: Arrow Functions

Arrow functions are a shorter syntax for writing functions in JavaScript. They use the `=>` syntax and are often more concise than traditional function expressions.

`() => { }`

```
() => {}  
  
(a) => { return a + a }  
  
(a) => a + a  
  
a => a + a
```

They not only make functions easier to write but also easier to read and debug.

Functions: Arrow Function example

Let's look at an example:

```
1 // Traditional function
2 function multiply(a, b) {
3     return a * b;
4 }
5
6 // Arrow function
7 const multiply = (a, b) => a * b;
```

Can you observe that arrow function syntax is simpler, easier, and more readable compared to traditional functions?

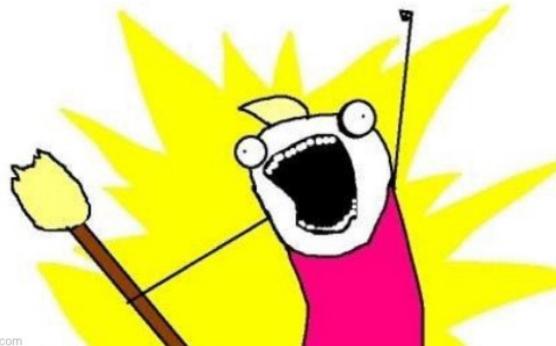
Arrow Functions: Shortcut to Simplicity

Arrow functions make coding faster and cleaner, like taking a shortcut.

Me first learning JS:
`()=> thing`

Me two weeks into learning JS:

ARROW ALL THE FUNCTIONS



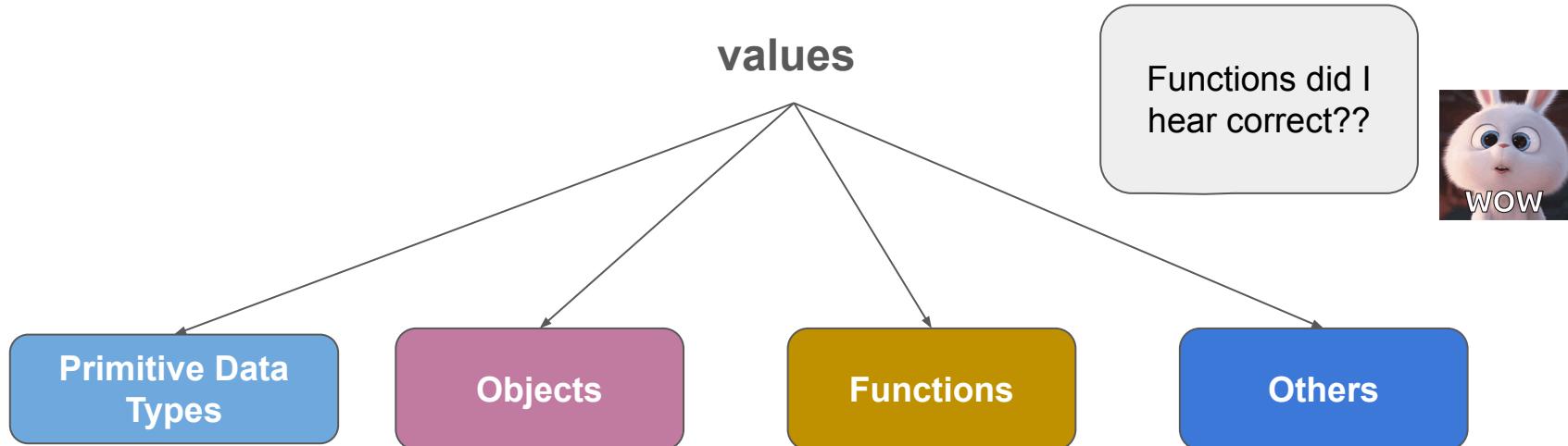
imgflip.com

Feeling awesome,
right??

Higher Order Functions and Callbacks

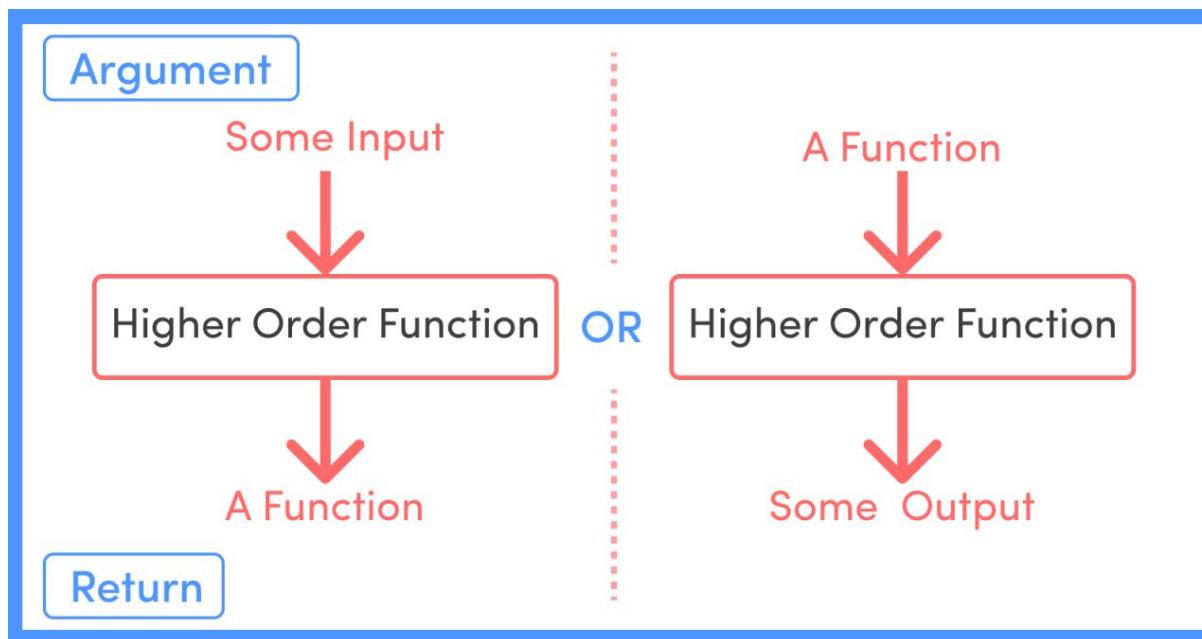
Values we can pass/return in a function

You can pass variety of values in a function. Let's have a look:



Higher-Order Functions

A **higher-order function** is a function either accepts a function as an argument or returns a function.



But why do we need function to receive and accept values??

Advantage of Higher-Order Functions

We receive or return functions as values to make our code flexible and reusable, allowing us to define behavior dynamically and build more modular, maintainable programs.

Understood??



Don't worry? Let's understand it with an example in next slides.

Understanding HOF with example

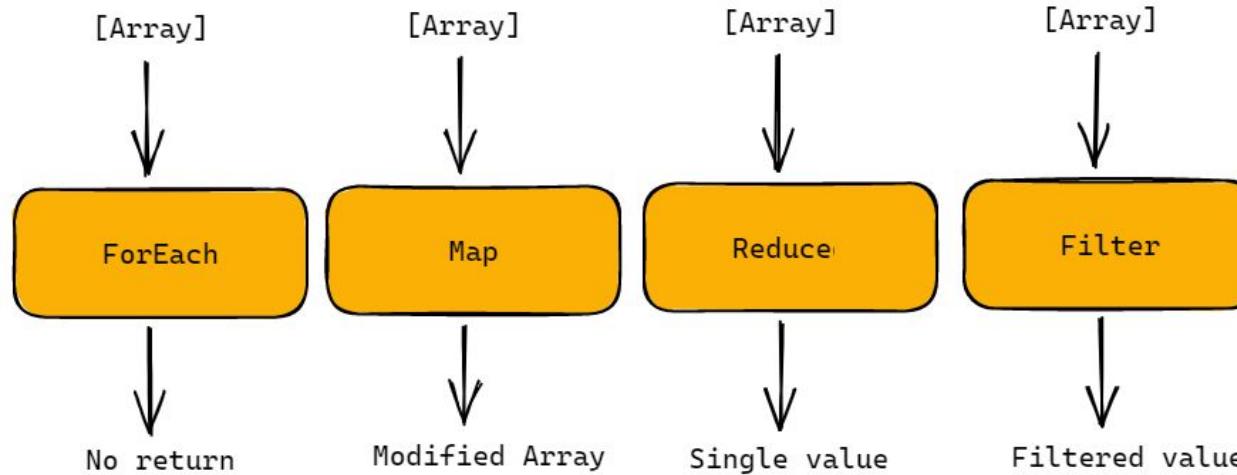
Let's create a greet function which takes name and function as input and returns a greeting.

```
1 // Higher-order function
2 function greet(name, callback) {
3     return callback(name);
4     // The callback function is applied to the name
5 }
6
7 // Simple functions that adds a greeting
8 function sayHello(name) {
9     return `Hello, ${name}!`;
10 }
11
12 function goodMorning(name){
13     return `Good Morning, ${name}`;
14 }
15
16 // Using the higher-order function
17 const greeting1 = greet('Rohan', sayHello);
18 const greeting2 = greet('Sohan', goodMorning);
19
20 console.log(greeting1); // Output: Hello, Rohan!
21 console.log(greeting2); // Output: Good Morning, Sohan
```

This might seem of no use, but when we will read arrays methods like map, filter etc, you will get to know helpful the higher order functions are.

Few popular higher order functions

Here are few most popular higher order functions:-



Don't worry we will learn these higher order functions in later lectures.

Understanding Callbacks

A **callback** is a function passed as an argument to another function, which is then executed inside that function. You can name callback function as you wish.



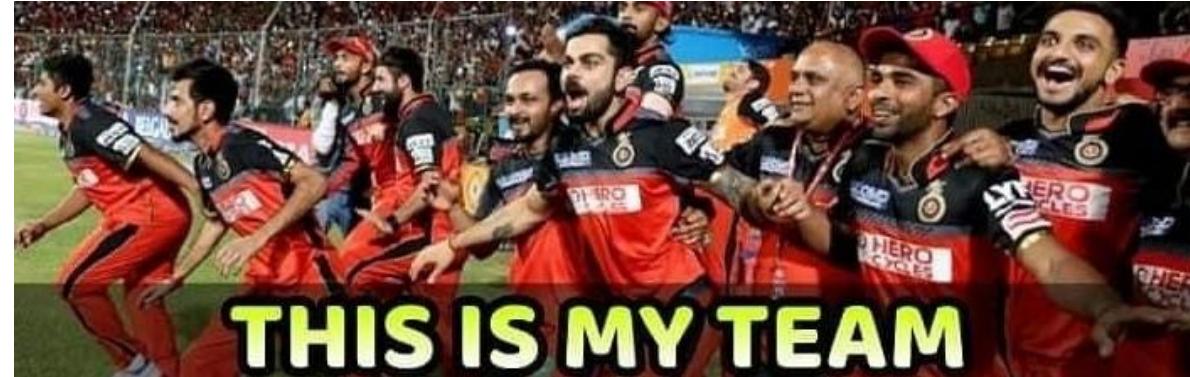
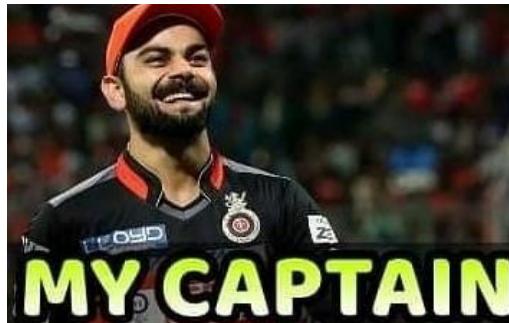
Callback function

```
1 // Higher-order function
2 function greet(name, callback) {
3     return callback(name);
4
5     // The callback function is applied to
6     // the name
7 }
```

Rest and Spread Operator

Rest operator: team player of javascript

You are tasked to form a cricket team. First, pick your favourite duo to open the batting, then select the captain. The rest make up the team.



Rest operator: team player of javascript

Let's write a code for it:

```
1 function formCricketTeam(  
2     captain, ...team  
3 ) {  
4     console.log("Captain:", captain);  
5     console.log("Team:", team);  
6 }
```

rest operator
grouping rest of
the values in
variable team

Output:

Captain: Virat Kohli
Team: ["Jasprit Bumrah", "Hardik Pandya", "Ravindra Jadeja", "Others"]

Rest operator: team player of javascript

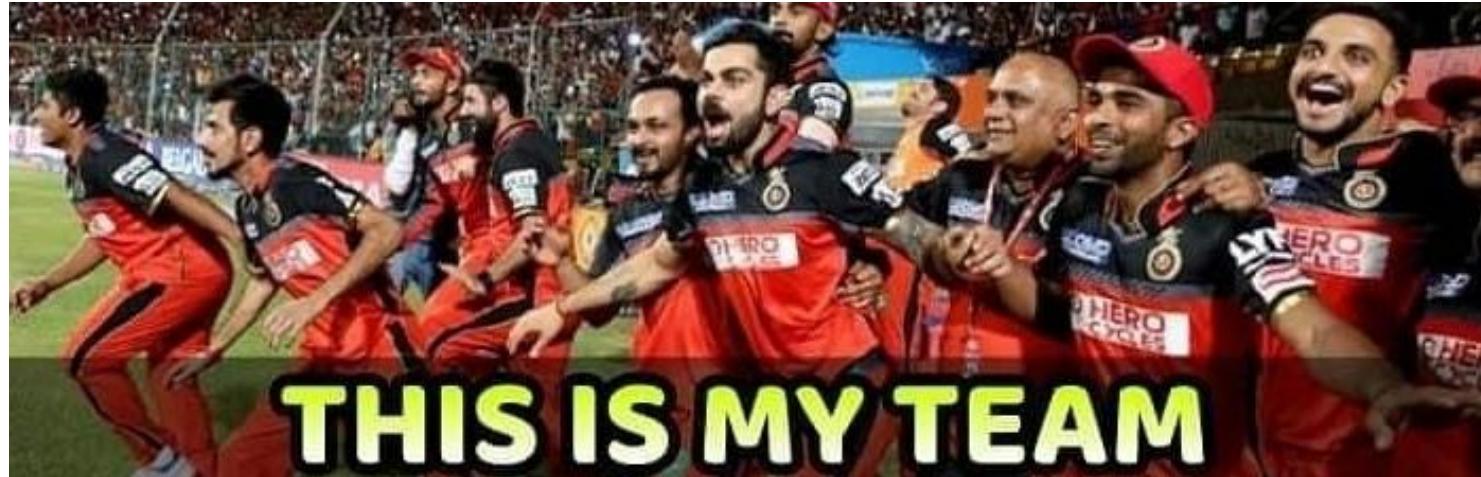
Let's pass values to the function:-

```
1 // Example usage:  
2 formCricketTeam(  
3     // Captain  
4     "Virat Kohli",  
5  
6     // Team  
7     "Jasprit Bumrah",  
8     "Hardik Pandya",  
9     "Ravindra Jadeja",  
10    "others"  
11 );
```

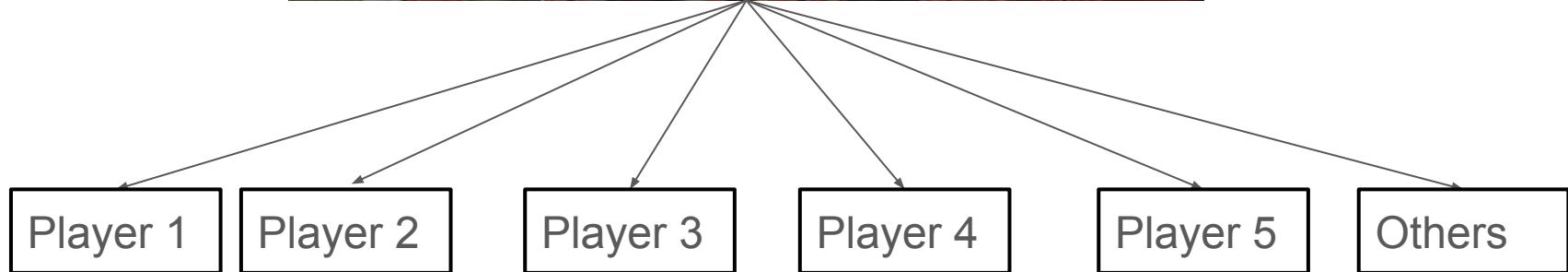
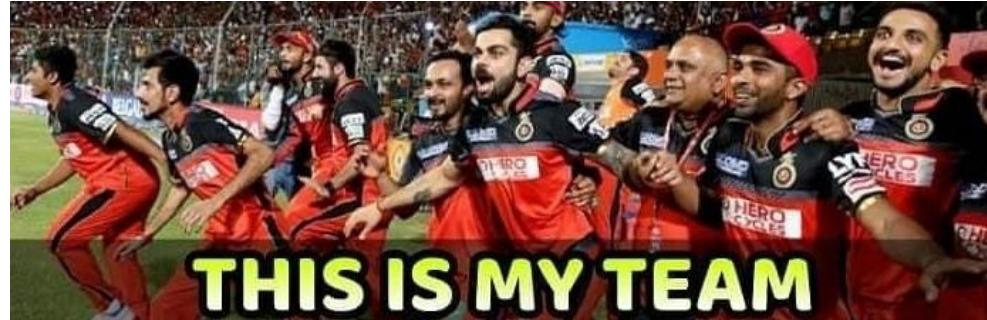
Team members are grouped together in receiving function using rest operator

Spread Operator: The Team Promoter

The spread operator (. . .) takes values from a group (like an array or object) and spreads them out into individual pieces, just like a promoter showing off each team member one by one.



Spread Operator: The Team Promoter



Spread Operator: The Team Promoter

Let's understand it with a code:-

```
1 // Using spread operator
2 const [
3     player1,
4     player2,
5     player3,
6     player4,
7     player5
8 ] = [...rcbTeam]; ←
9
10 console.log('Player 1:', player1);
11 console.log('Player 2:', player2);
12 console.log('Player 3:', player3);
13 console.log('Player 4:', player4);
14 console.log('Player 5:', player5);
```

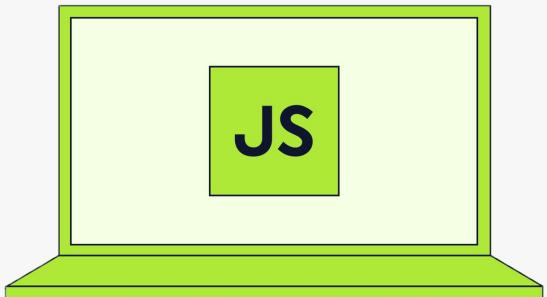
Here We are using spread operator to spread out individual player names in player 1, player 2, ...etc.

In Class Questions

**Thanks
for
watching!**



The Complete Javascript Course



@newtonschool

Lecture 5: Introduction to Arrays

-Vishal Sharma



Table of Contents

- What are Arrays?
- Javascript Arrays vs Python Lists
- Common methods:-
 - push
 - pop
 - shift
 - unshift
 - find
 - includes
 - sort
- Bonus Methods: join and split
- Call by Value vs Call by Reference

Arrays

Planning a party

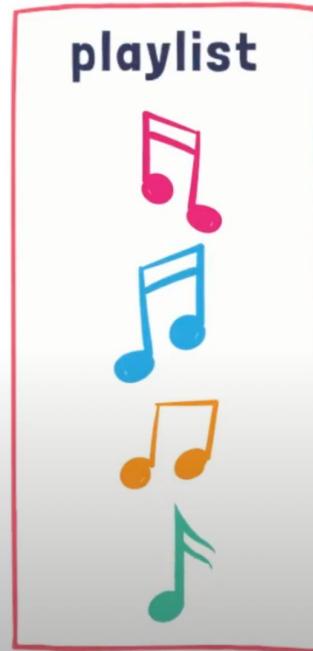
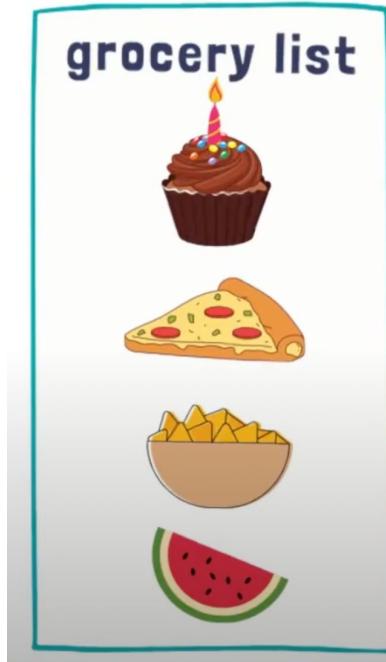
Imagine you are organizing a party, for that you need to purchase lot of items.



You will need, cake, music player, food and decorative items. So you might want to make a list out of it!!

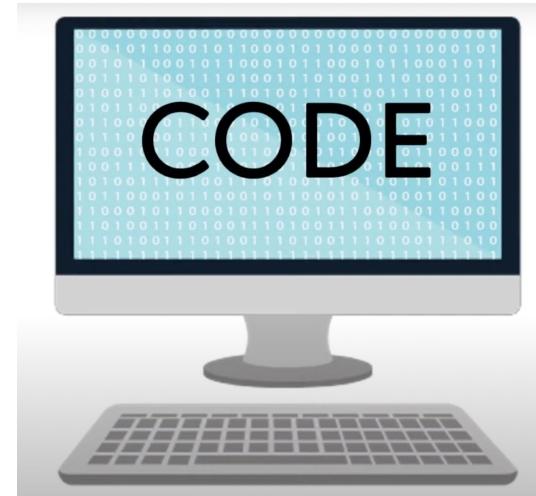
Planning a party: making lists

Let's make lists, separate list for separate group of items:-



Planning a party: making lists

Just like lists help you keep your tasks organized. Lists are very helpful in programming too:-



Array: Lists in javascript

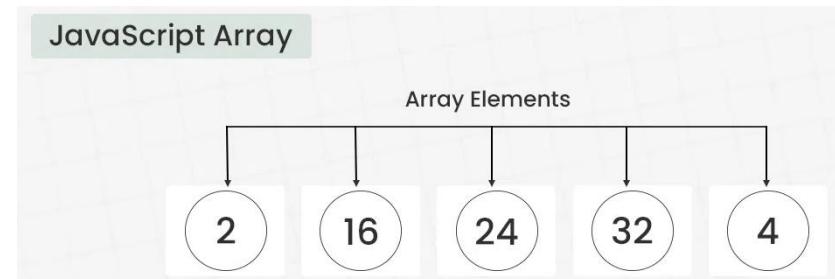
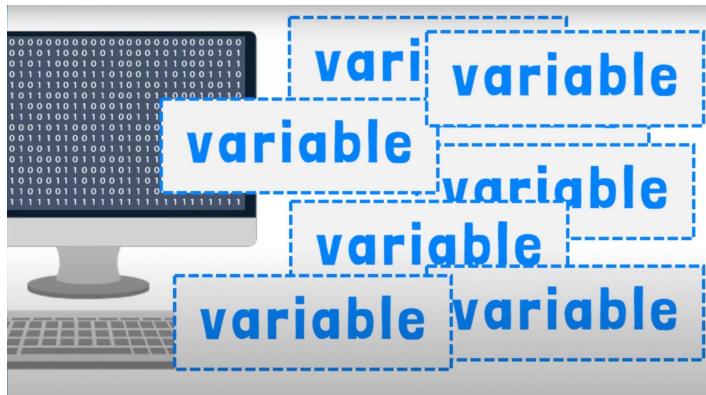
Lists in javascript are called “Array”.

Definition: An array in JavaScript is a special object used to store multiple values in a single variable.



Why do we need array?

Variables store data, but managing many of them is hard. Arrays simplify this by organizing multiple values in one place.

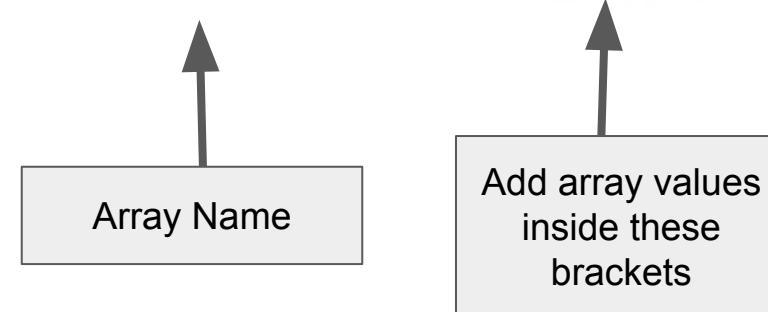


Array: Definition and Syntax

An array in JavaScript is a special object used to store multiple values in a single variable.

Const

MyArray = [] ;



Array: Storing values in arrays

We can declare array as well as store values at the same time.

```
Const MyArray = [ 1, 2, 3, 4, "Hello", true, null ] ;
```



Array Name



You can use mixed data
type in arrays

Array: Accessing the array values

We can access values stored in array by using indexes.

```
1 // Declaring and assigning array
2 const MyArray = [1, 2, 3, 4, "Hello", true , null];
3
4 // Printing array values
5 console.log(MyArray);
6 // Output: [1, 2, 3, 4, "Hello", true , null];
7 console.log(MyArray[0]); // Output: 1
8 console.log(MyArray[1]); // Output: 2
9 console.log(MyArray[4]); // Output: Hello
10 console.log(MyArray[5]); // Output: True
```

If we just type
MyArray in console,
entire array would get
printed

Array Manipulation

Array manipulation

Often, we need to manipulate arrays by adding, removing, updating, or extracting elements. Array methods make this process easy and efficient.



We achieve it using array
methods.

Common Javascript Methods

Here are some commonly used array methods:

push

pop

find

unshift

shift

includes

Add Items

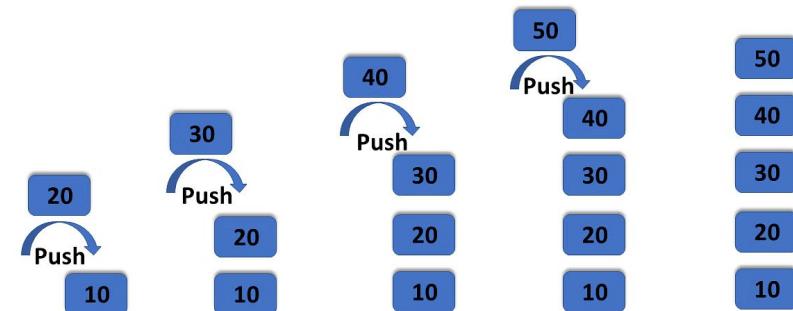
Remove items

Search

Add Items: push

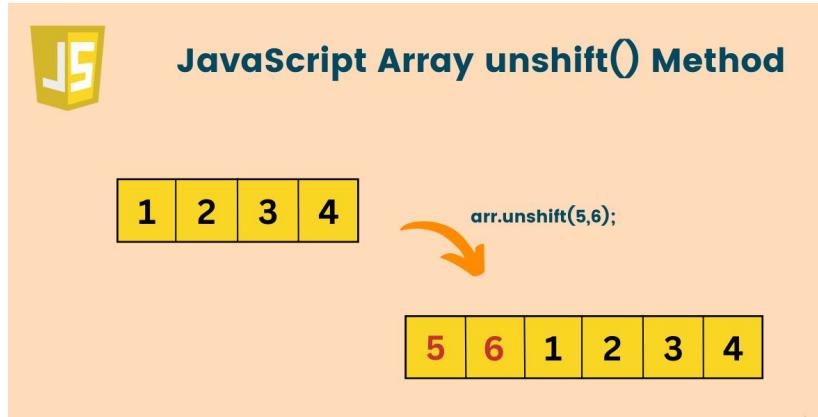
Adds one or more items to the **end** of the array. Initially there was 10, however we could have started with empty array

```
1 // Declaring an empty array
2 let numbers = [10];
3
4 // Adding items one by one
5 numbers.push(20); // [20]
6 numbers.push(30); // [20, 30]
7 numbers.push(40); // [20, 30, 40]
8 numbers.push(50); // [20, 30, 40, 50]
9
10 console.log(numbers); // [20, 30, 40, 50]
11
12 // Adding two items at once
13 numbers.push(60, 70);
14
15 console.log(numbers); // [20, 30, 40, 50, 60, 70]
```



Add Items: unshift

Instead of adding items at the end of the array we can add items to the **beginning** of the array.

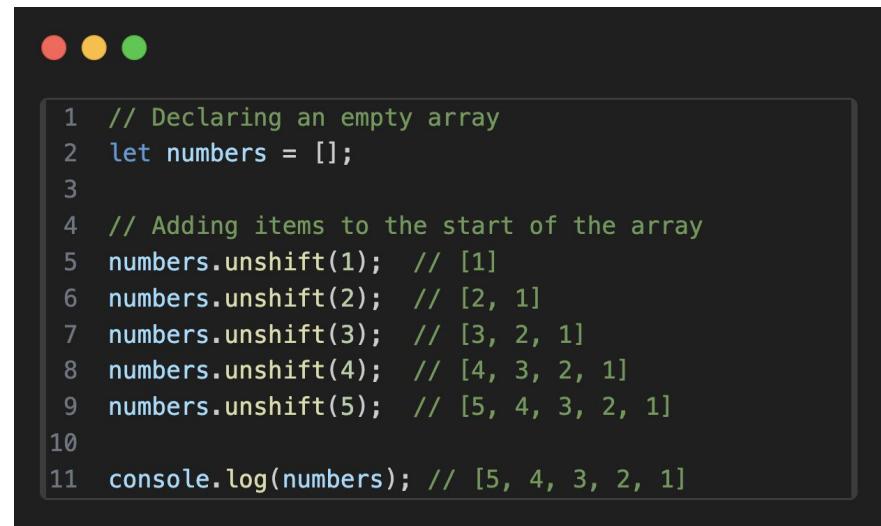


JavaScript Array unshift() Method

1	2	3	4
---	---	---	---

arr.unshift(5,6);

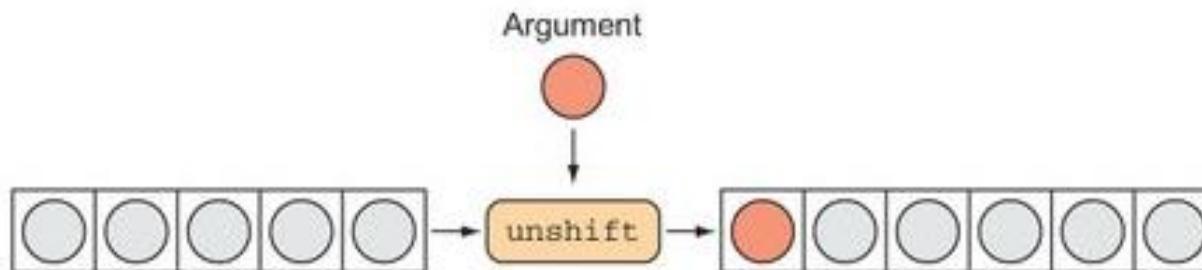
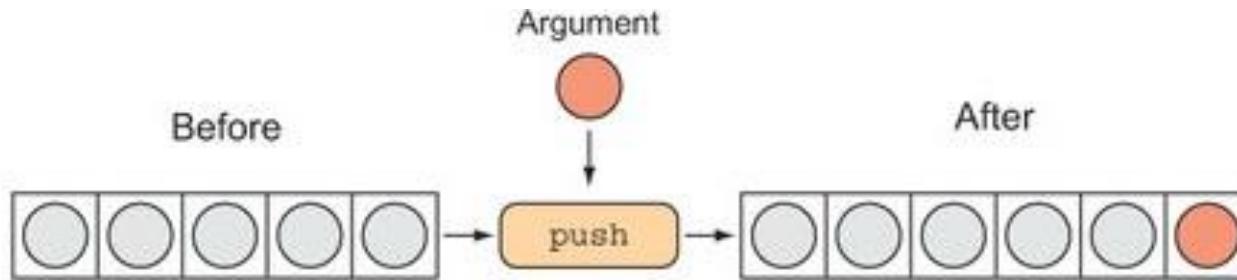
5	6	1	2	3	4
---	---	---	---	---	---



```
1 // Declaring an empty array
2 let numbers = [];
3
4 // Adding items to the start of the array
5 numbers.unshift(1); // [1]
6 numbers.unshift(2); // [2, 1]
7 numbers.unshift(3); // [3, 2, 1]
8 numbers.unshift(4); // [4, 3, 2, 1]
9 numbers.unshift(5); // [5, 4, 3, 2, 1]
10
11 console.log(numbers); // [5, 4, 3, 2, 1]
```

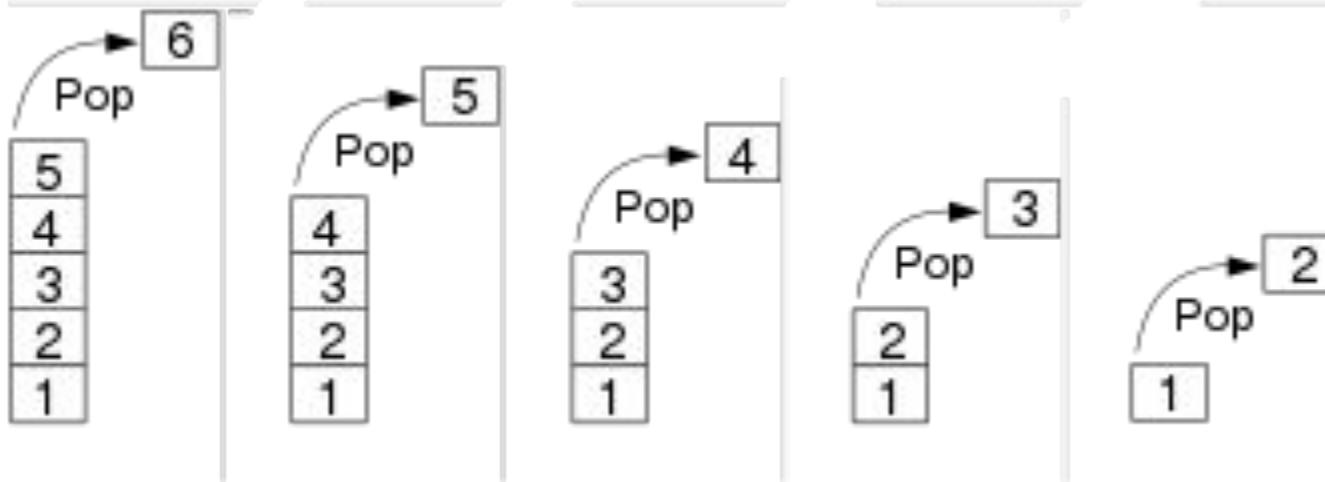
Difference: push and unshift

Let's understand the difference between them two:-



Remove Items: pop

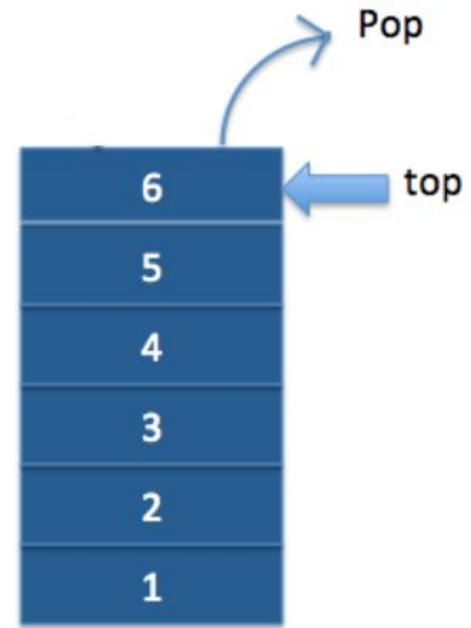
Removes the **last** item from an array.



Remove Items: pop

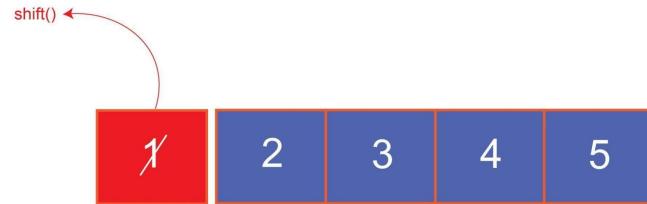
Let's write javascript code for it:-

```
1 // Initial array with 5 items
2 let numbers = [1, 2, 3, 4, 5];
3
4 // Using pop to remove all items
5 numbers.pop(); // [1, 2, 3, 4]
6 numbers.pop(); // [1, 2, 3]
7 numbers.pop(); // [1, 2]
8 numbers.pop(); // [1]
9 numbers.pop(); // []
10
11 console.log(numbers); // []
```



Remove Items: shift

Removes the **first** item from an array.

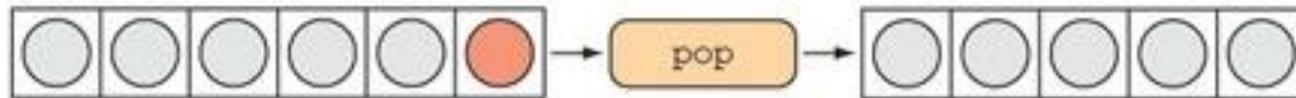


A screenshot of a terminal window with three colored window control buttons (red, yellow, green) at the top. The terminal displays the following code:

```
1 // Initial array with 5 items
2 let numbers = [1, 2, 3, 4, 5];
3
4 // Using shift to remove items
5 numbers.shift(); // [2, 3, 4, 5]
6 numbers.shift(); // [3, 4, 5]
7 numbers.shift(); // [4, 5]
8 numbers.shift(); // [5]
9 numbers.shift(); // []
10
11 console.log(numbers); // []
```

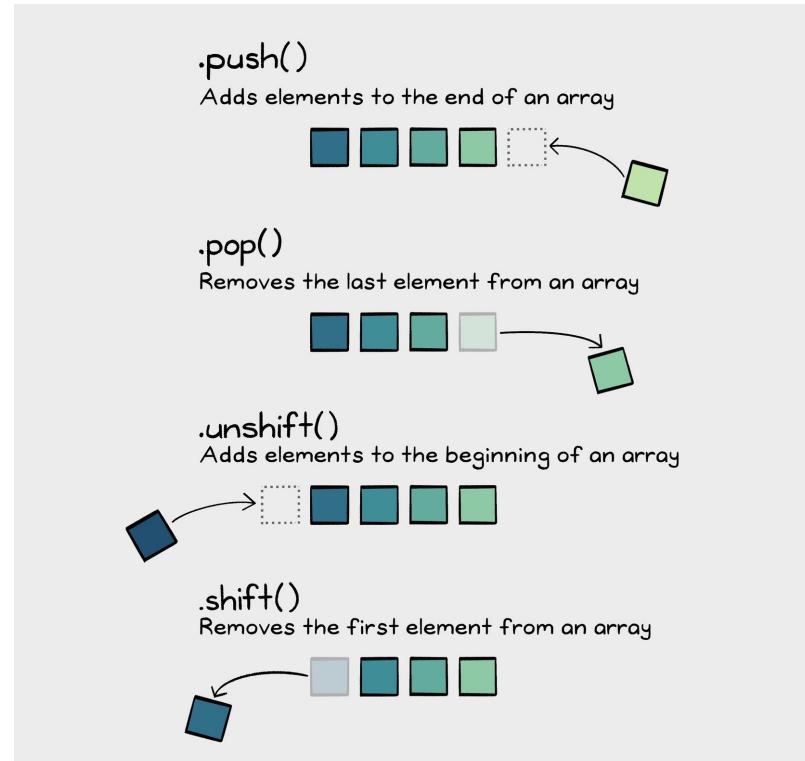
Difference: pop and shift

Let's understand the difference between these two:-



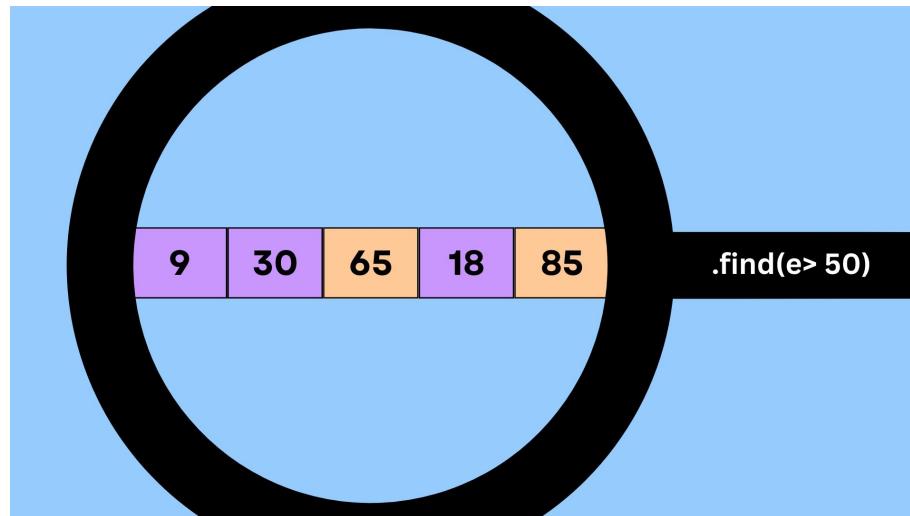
Let's quickly recap what we learned

Here's a quick recap of the array methods we've covered:



Search Methods: `find()`

The `find` method helps you search through an array and find the **first** item that matches a specific condition. Once it finds the item, it stops searching and returns that item.



Since only two values i.e. 65 and 85 is greater than 50, but since 65 is the first occurrence, 65 gets returned.

find(): Syntax

Let's have a look at the syntax of find method.

Syntax:



```
1 numbers.find(number) => {  
2     // code ←  
3     // return true/false ←  
4 }
```

find method logic, you can decide it as per as the problem statement

Based on method logic we decide it match is found or not, if found then we return true

find() example 1

You are given an array of integers and your task is to find the first occurrence of a number which is more than 10. Let's use find() method for this purpose:-

```
1 // Define the array of numbers
2 const numbers = [5, 8, 12, 3, 7];
3
4 // `find` method to get the first number greater than 10
5 const greaterThanTen = numbers.find(num => {
6     // Check if the current number is greater than 10
7     const greater = num > 10;
8     return greater;
9 });
10
11 // Log the result to the console
12 console.log(greaterThanTen); // Output: 12
```

Since 12 is the first number which is more than 10, output is 12.

find() example 2

Given an array having data of different people find out the person with exact age of 30 yrs. Let's use an arrow function to shorten the code this time:-



```
1 // Array of objects with name and age
2 let people = [
3   { name: "John", age: 25 },
4   { name: "Alice", age: 30 },
5   { name: "Bob", age: 22 }
6 ];
7
8 // find the first person who is 30 years old
9 let person = people.find(p => p.age === 30);
10
11 // Printing the result
12 console.log(person);
13 // Output: { name: "Alice", age: 30 }
```

We searched the array and printed the information of the person whose age is exactly 30.

Search Methods: includes()

The `includes` method checks if a certain value exists in an array. It returns `true` if the value is found, and `false` if it's not.

```
months.includes("Apr");
```



The diagram illustrates the `months` array and its `includes` method call. The array is represented as a grid with indices 0 through 4. The elements are `Jan`, `Feb`, `Mar`, `Apr`, and `May`. An orange arrow points from the `includes` method call to the `Apr` element in the array. A dashed blue box labeled `true` is positioned above the array, indicating that the method returns `true` because the value "Apr" is found in the array.

```
const months = [ Jan, Feb, Mar, Apr, May ]
```

```
months.includes("Apr");
```



```
months.includes("apr");
```

includes() example

Given an array of months, check if particular month is present in the array or not.

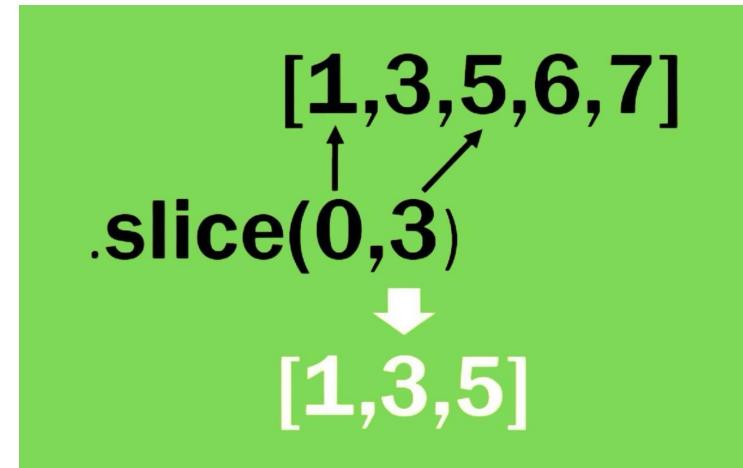
```
1 const months = ["Jan", "Feb", "Mar", "Apr", "May"];
2
3 // Check if the array includes "Apr" and "apr"
4 let includesApr = months.includes("Apr");
5 let includesAprLower = months.includes("apr");
6
7 console.log(includesApr); // Output: true
8 console.log(includesAprLower); // Output: false
```

We haven't stored April in lowercase i.e. "apr" so we got false in second instance.

Copying the array and modifying it directly

Copying the Array: `slice()` method

As the name suggests, the `slice` method cuts out a section of an array and returns a new array containing that section, without modifying the original array.



slice(): Example

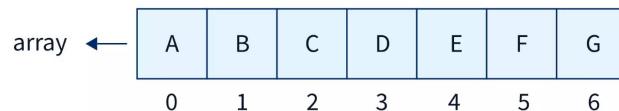
Let's take an initial array [1, 3, 5, 6, 7], and use the slice(0, 3) method to get a portion of it.

```
1 // Initial array
2 let numbers = [1, 3, 5, 6, 7];
3
4 // Using slice from index 0 to 3 (excluding index 3)
5 let slicedArray = numbers.slice(0, 3);
6
7 console.log(slicedArray); // [1, 3, 5]
8 console.log(numbers);
9 // [1, 3, 5, 6, 7] (Original array)
```

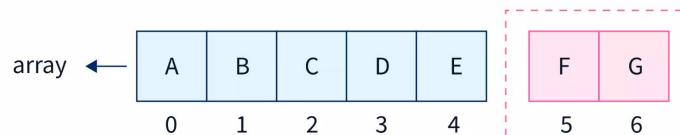
Not the end index
is not included in
the slice!!

Modifying the Array: `splice()` method

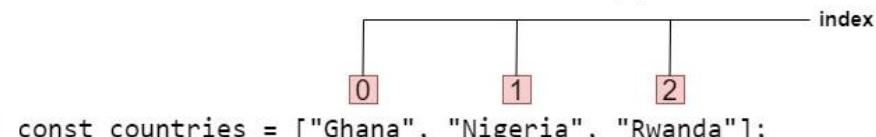
The `splice` method changes the contents of an array by removing, replacing, or adding elements at a specified index, modifying the original array.



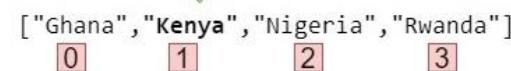
`array.splice(5, 2)` delete_count is 2, so two elements will be removed starting from index 5



`array.splice(2)` starts removing elements from index 2



`countries.splice(1, 0, 'Kenya');`



Removing Items

Adding Items

splice(): Example

`splice(0, 3)` removes 3 elements starting from index 0, modifying the original array to [4, 5].



```
1 // Initial array
2 let numbers = [1, 2, 3, 4, 5];
3
4 // Using splice to remove the first three items
5 numbers.splice(0, 3);
6
7 console.log(numbers); // [4, 5]
```

Here instead of modifying the original array `splice` method modified the original array.

Sort method

Organizing array: `sort()`

The `sort()` method allows us to arrange array elements in a specific order, making data easier to understand and work with.

5	8	2	3	6
---	---	---	---	---

Unsorted Array



2	3	5	6	8
---	---	---	---	---

Sorted Array

Sorting an array organizes data, improves readability and speeds up searches.

sort(): How it works?

By default sort() method sorts in ascending order:-



```
1 // Array declaration and assignment
2 const array = [5, 4, 3, 2, 6];
3
4 // Sorting the array
5 array.sort();
6
7 console.log(array);
8 // Output: [2, 3, 4, 5, 6]
```

We can pass compare function as a callback function to sort() to change its default behaviour.

sort(): How it works?

To sort an array we need to use `sort()` method and pass a compare function to it:-



```
1 // Array declaration and assignment
2 const array = [5, 4, 3, 2, 6];
3
4 // Sorting the array
5 array.sort(compareFunction);
```



```
1 // compare function ascending
2 function compareFunction(a, b){
3     return a - b;
4 }
```



```
1 // compare function descending
2 function compareFunction(a, b){
3     return b - a;
4 }
```

sort(): Example Ascending

Let's sort the array in ascending order:-



```
1 // Sorting the array
2 array.sort(compareFunction);
3
4 // compare function ascending
5 function compareFunction(a, b){
6     return a - b;
7 }
8
9 console.log(array);
10 // Output: [2, 3, 4, 5, 6]
```

If $a - b$ is positive ($a > b$):

- **Action:** b is placed before a because b is smaller.
- **Reason:** The `sort()` function swaps the two elements to ensure the smaller value comes first.

Vice-versa if $a - b$ is negative ($a < b$)

sort(): Example Descending

Let's sort the array in descending order:-



```
1 // Sorting the array
2 array.sort(compareFunction);
3
4 // compare function descending
5 function compareFunction(a, b){
6     return b - a;
7 }
8
9 console.log(array);
10 // Output: [6, 5, 4, 3, 2, 1]
```

If $b - a$ is positive ($b > a$):

- **Action:** b is placed before a because b is larger.
- **Reason:** The sort function keeps b first, ensuring the larger value comes first.

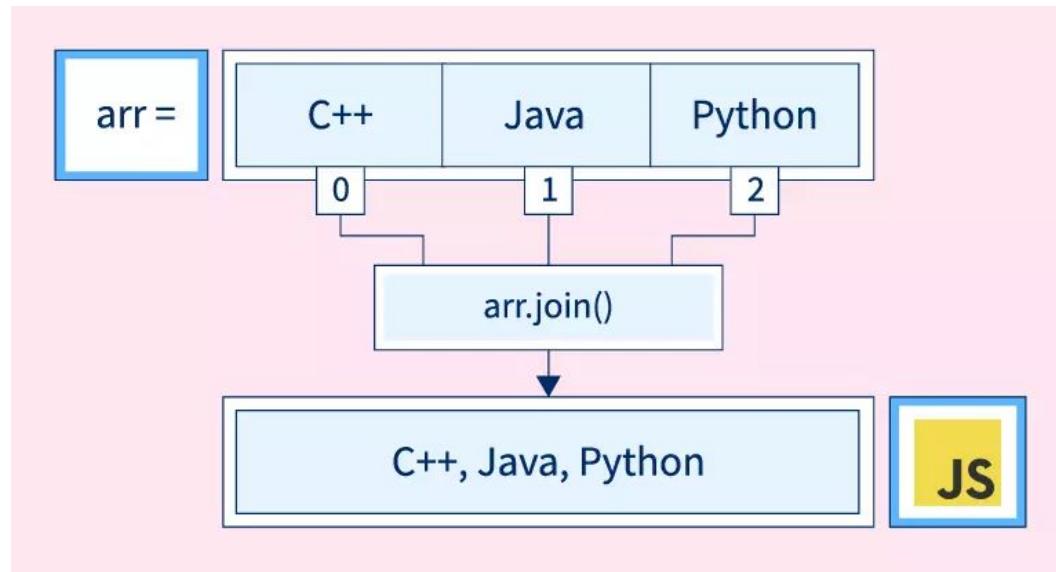
Vice-versa if $b - a$ is negative ($b < a$):

Bonus:

split and join methods

Array to String: join() method

join() method is used to concatenate the array elements into a string.



We can pass separator arguments in join method.

Let's say we did
`arr.join('|')`



C++|Java|Python

join(): Example

Let's understand join method with few examples:-

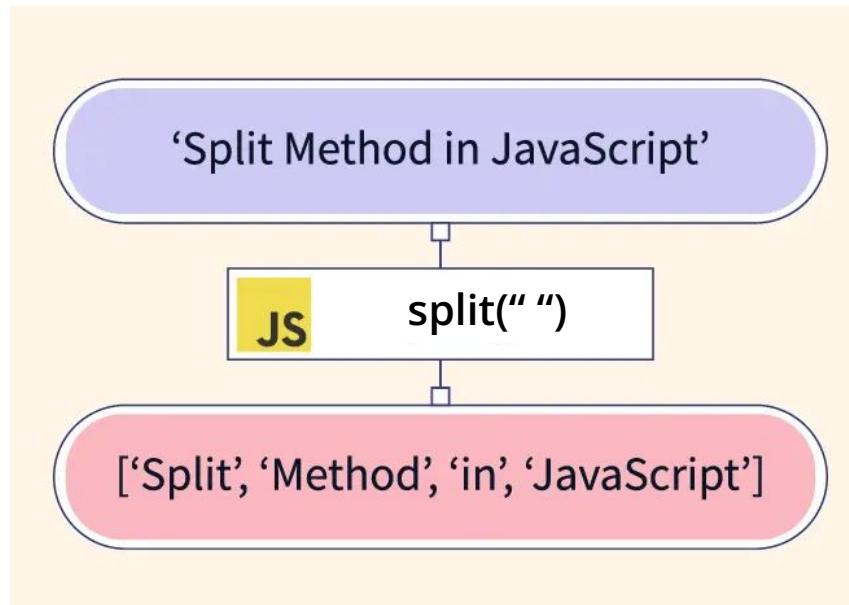


```
1 const languages = ['C++', 'Java', 'Python'];
2
3 // Using the default separator (comma)
4 console.log(languages.join());
5 // Output: "C++,Java,Python"
6
7 // Using a custom separator
8 console.log(languages.join(' | '));
9 // Output: "C++ | Java | Python"
10
11 // Joining without any separator
12 console.log(languages.join(''));
13 // Output: "C++JavaPython"
```

When we don't pass any argument in the join() method we get comma(,) as an separator.

String to Array: split() method

We can reverse the join method i.e. we can create array from string by using split() method:-



We need delimiters inside the split() method. Here in this case we use whitespace:-



split(" ");

split(): Example

Let's understand split method with few examples:-

If we don't pass delimiter it by default returns the whole text



```
1 const text = "C++,Java,Python";
2 const result = text.split();
3 // No separator provided
4
5 console.log(result);
6 // Output: ["C++,Java,Python"]
```

Here we splitted the array using whitespace a delimiter

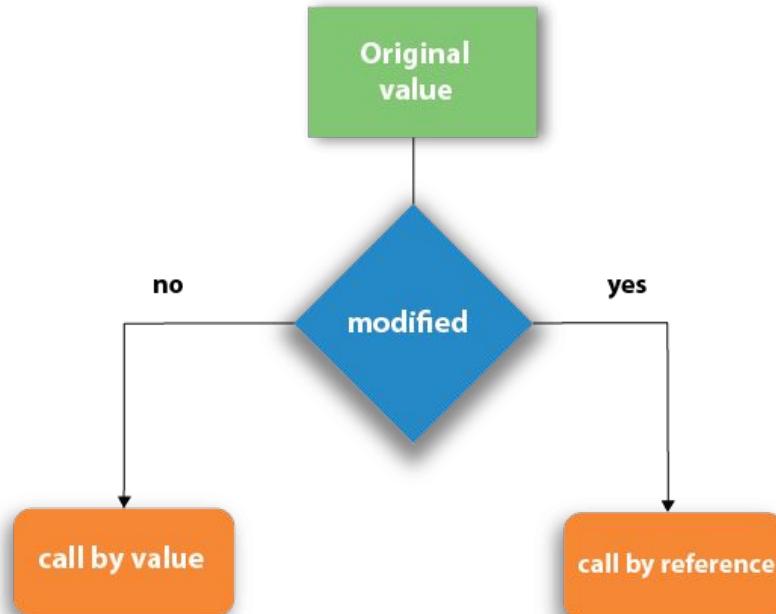


```
1 const sentence = "Split method in javascript";
2 const wordsArray = sentence.split(" ");
3 // Splits the string at each space
4
5 console.log(wordsArray);
6 // Output: ["Split", "method", "in", "javascript"]
```

Call by value vs call by reference

Call by Value vs Call by Reference

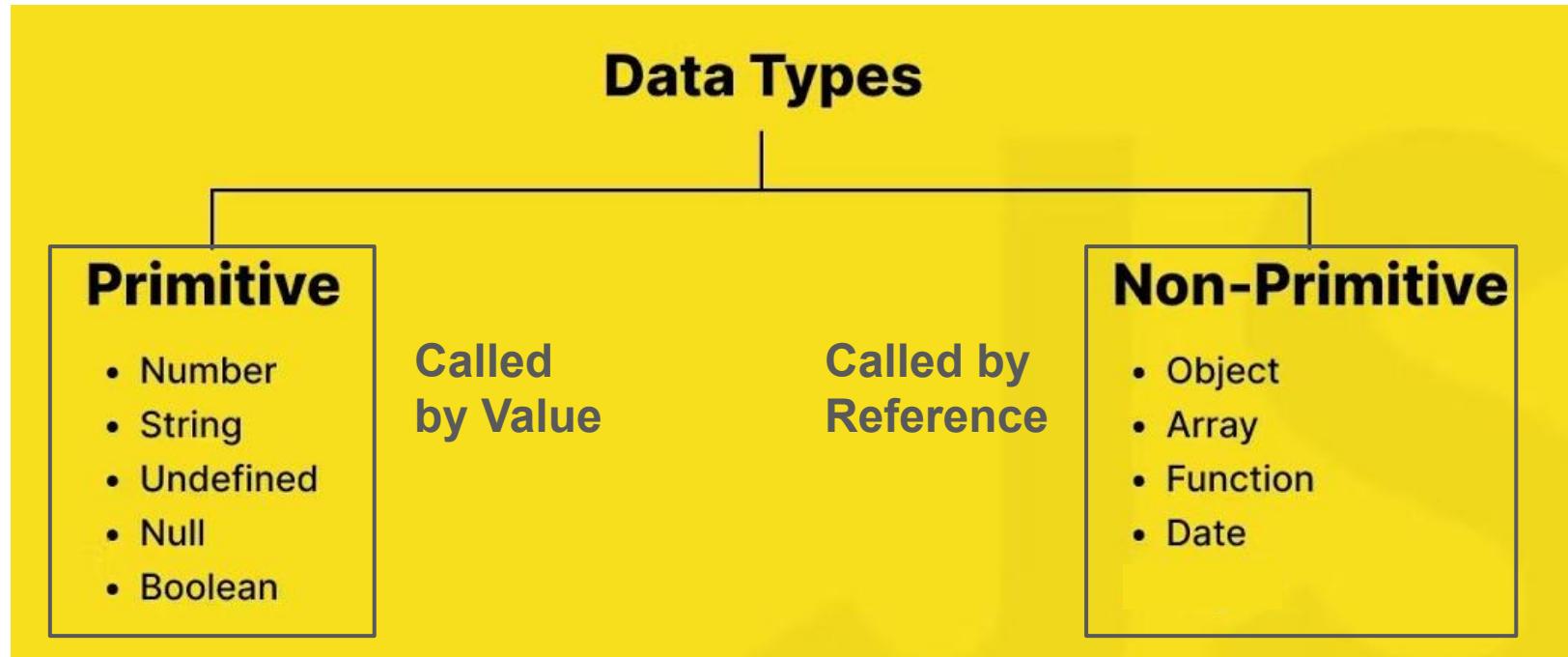
Understanding how JavaScript handles primitives and reference types in functions



Understood?? Don't worry, let's jump into next slides and then come back, to understand.

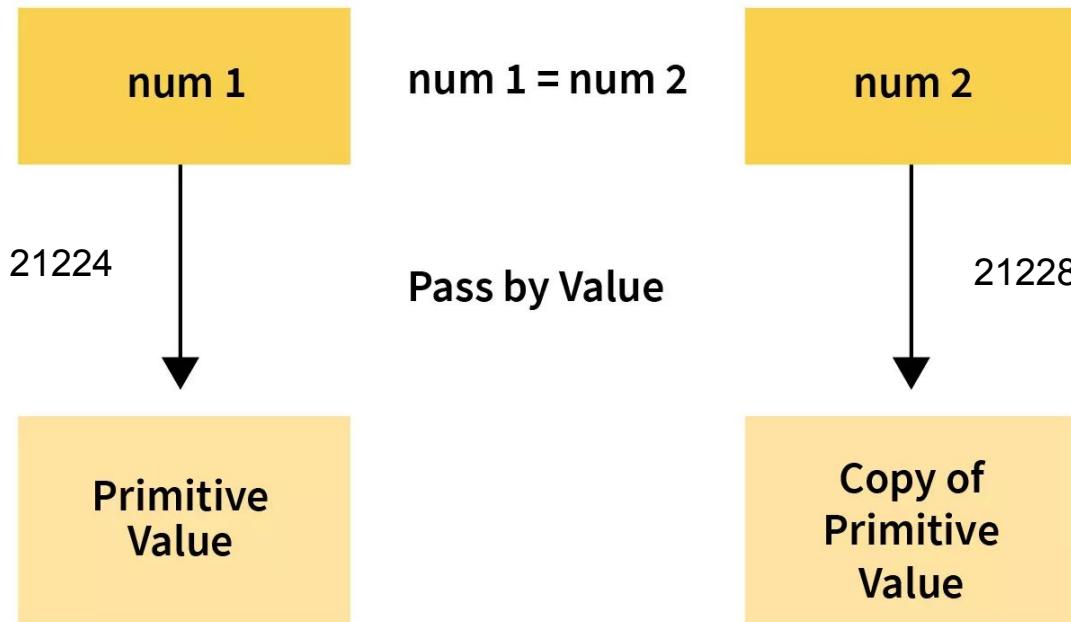
Primitive vs Non-primitive

Primitive data types (like numbers and strings) are passed by value (a copy), while non-primitive data types (like objects) are passed by reference (a link to the original).



Call by Value

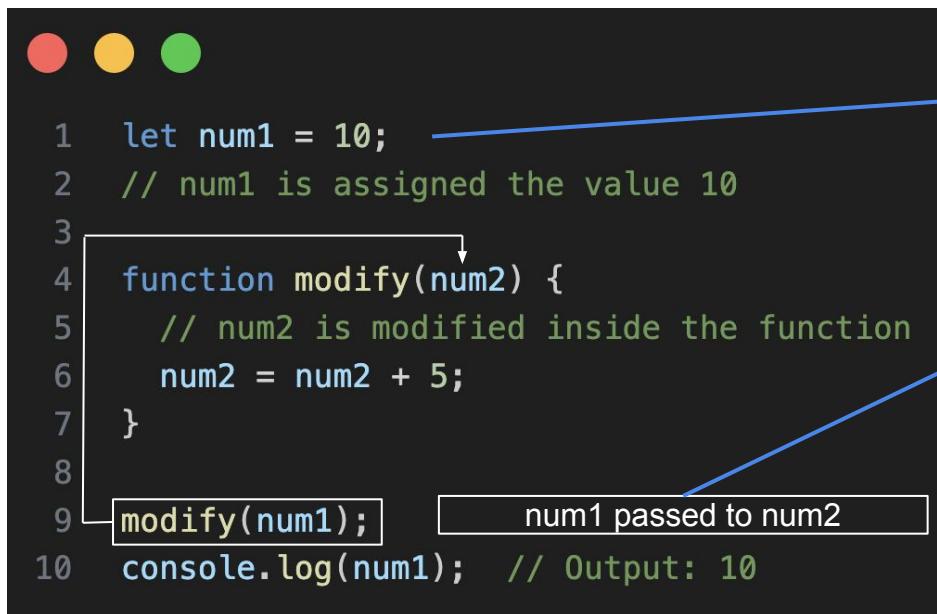
Whenever primitive value is assigned to another variable or passed to a function a new copy of it is generate.



Here 21224 and 21228 are memory locations where num1 and num2 are stored respectively.

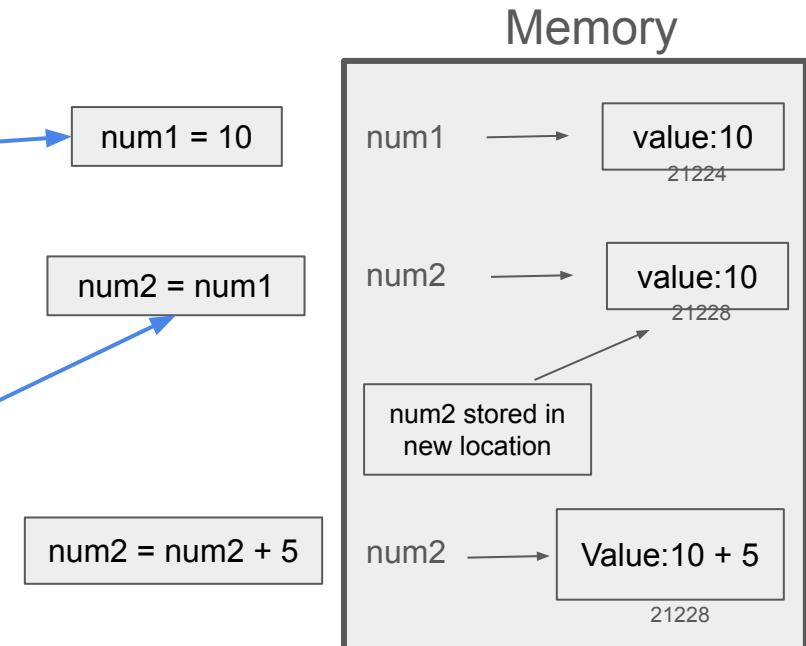
Call by Value: example

Let's understand how it happens with an example.



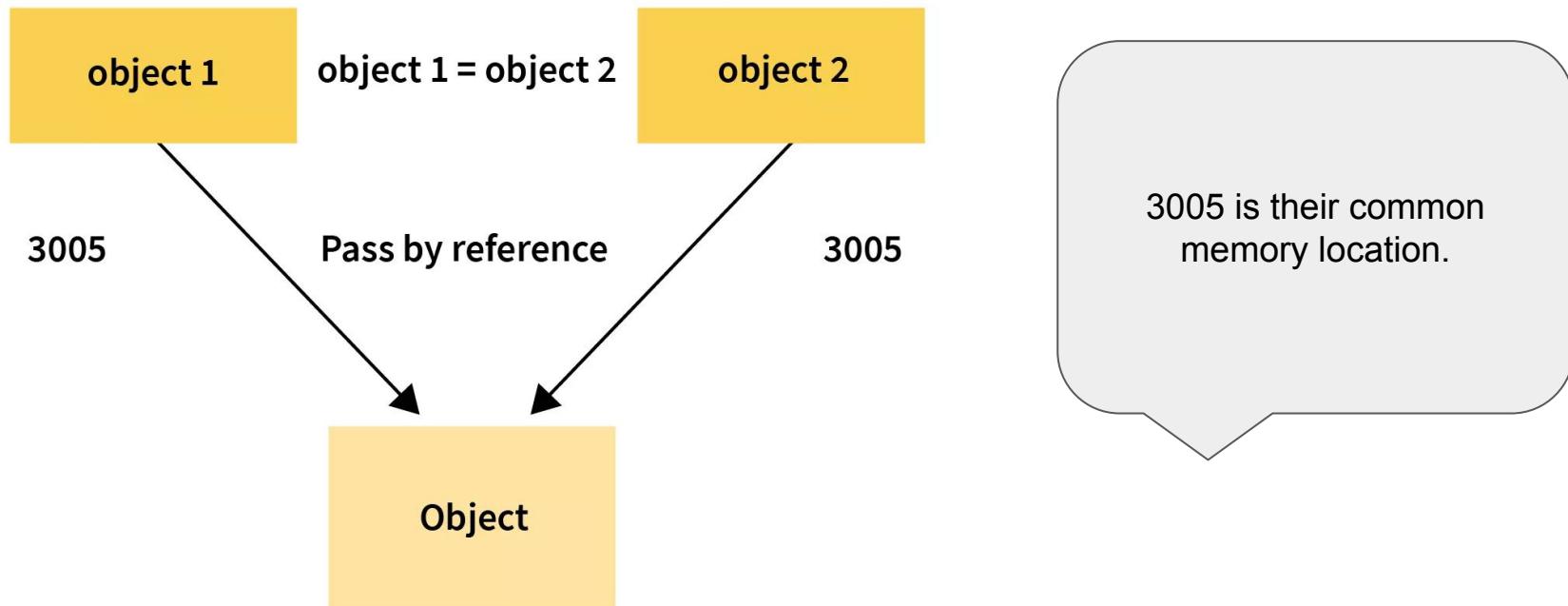
```

1 let num1 = 10;           → num1 = 10
2 // num1 is assigned the value 10
3
4 function modify(num2) {   ↓
5   // num2 is modified inside the function
6   num2 = num2 + 5;
7 }
8
9 modify(num1);           → num1 passed to num2
10 console.log(num1);     // Output: 10
  
```



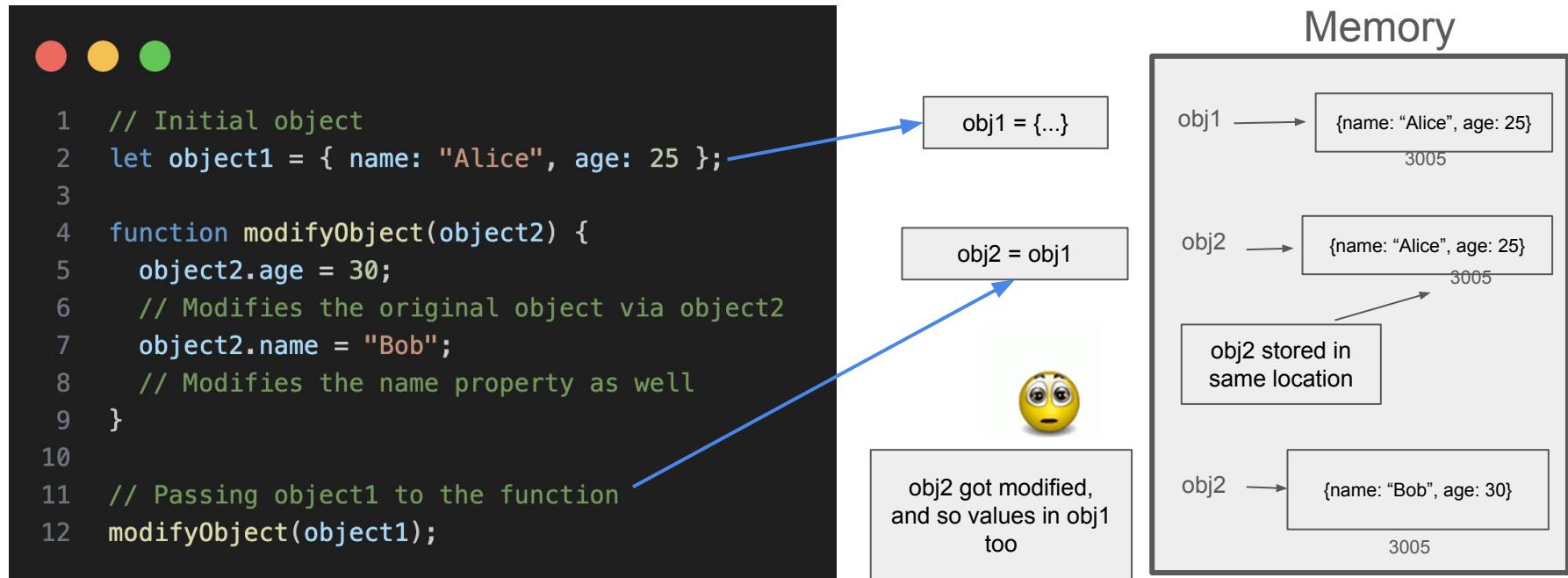
Call by Reference

Whereas in call by reference, the original variable and the function parameter point to the **same memory location**.



Call by Reference: example

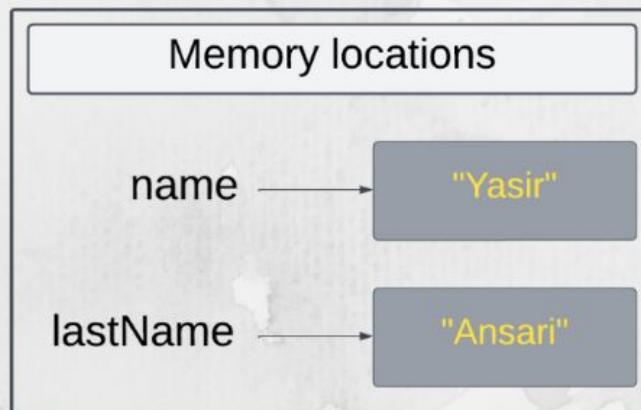
Let's understand how it happens with an example. Here obj1 and obj2 are pointing to same address.



Quick Recap

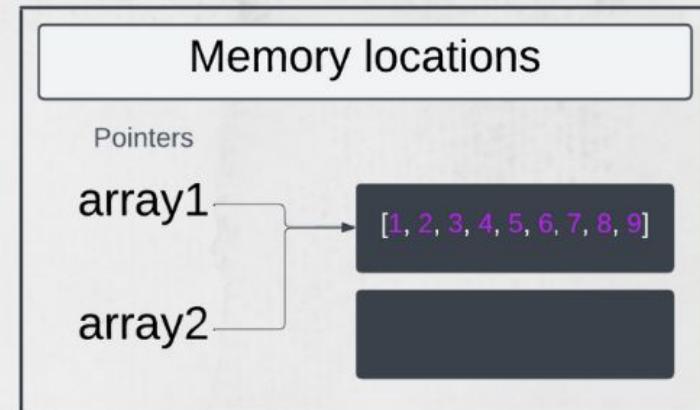
Let's have a quick recap how values are passed based on its type.

Primitive data type



Passed by value.

Reference data type



Passed by reference.

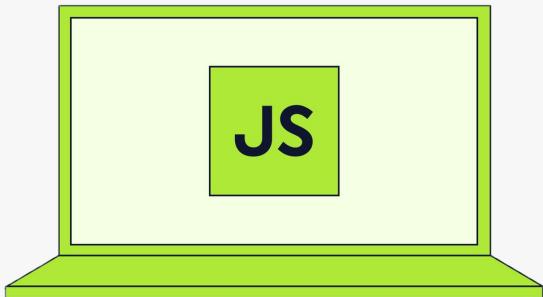


In Class Questions

**Thanks
for
watching!**



The Complete Javascript Course



@newtonschool

Lecture 6: Array and Array Methods

-Vishal Sharma



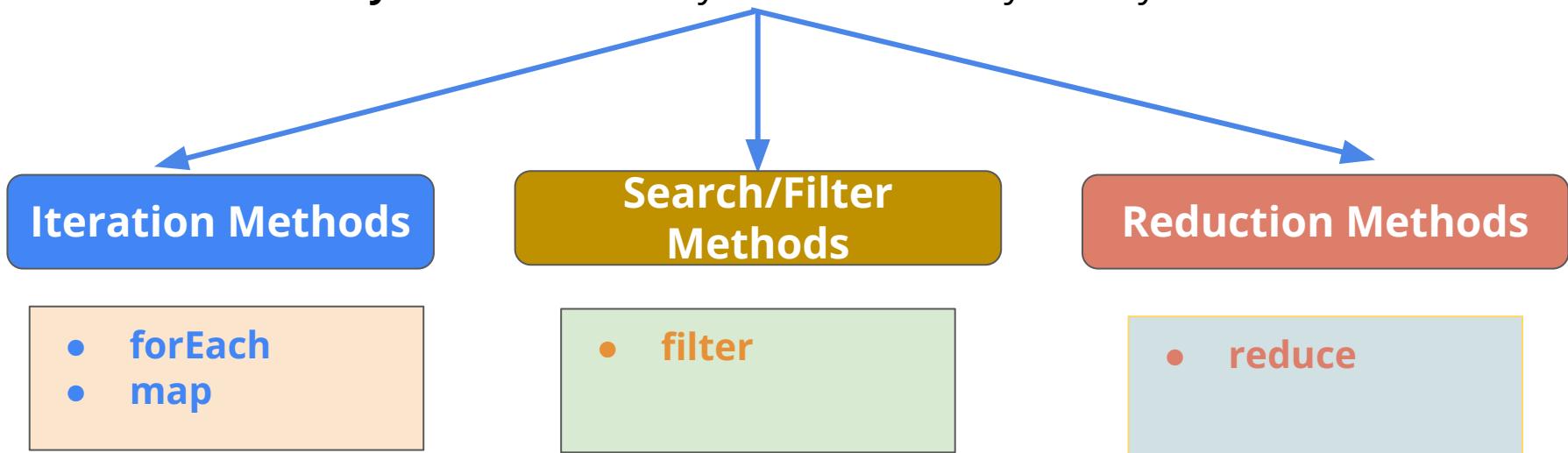
Table of Contents

- Common methods:-
 - forEach
 - map
 - filter
 - reduce
- Iterating through arrays with loops
- Real-world Examples using array methods

Array Traversal and Manipulation Methods

What are these methods??

We need these advanced array methods because they provide a **concise, readable, and efficient way** to work with arrays. We can broadly classify them as:



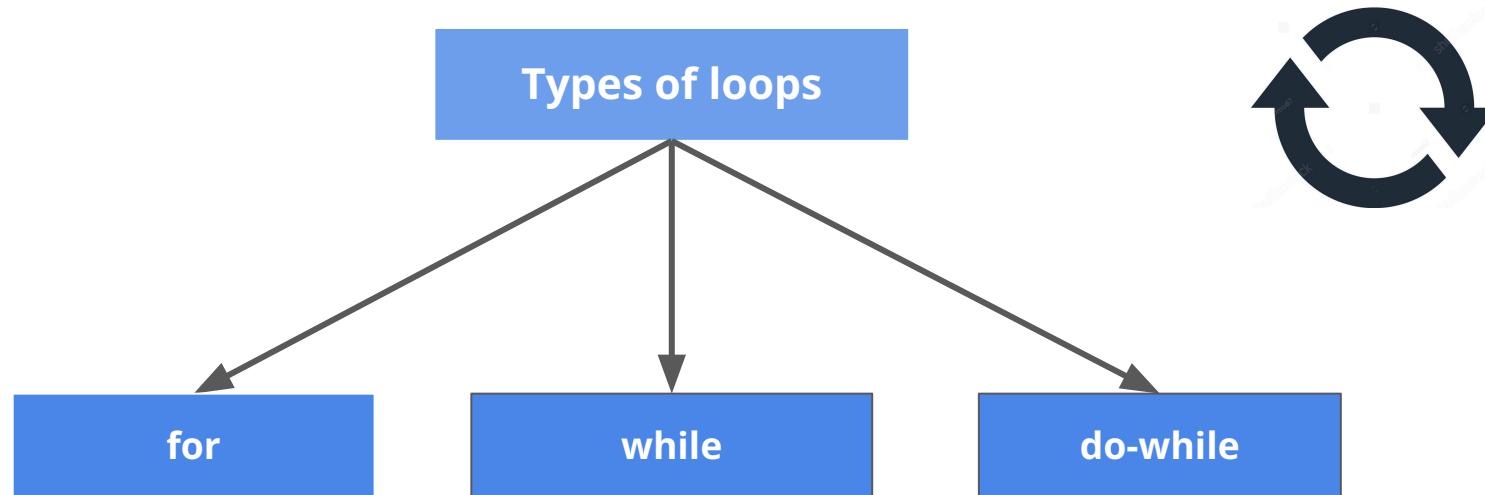
Pre-discussed: Loops in real life

Every day we follow certain routine which we repeat all over the week, like brushing your teeth, taking shower, etc.



Pre-discussed: Loops in Programming

Just like we repeat several tasks in daily life, we need some tasks to be repeated in programming and we repeat those tasks with the help of loops.



Why do we need newer methods?

Need for newer methods

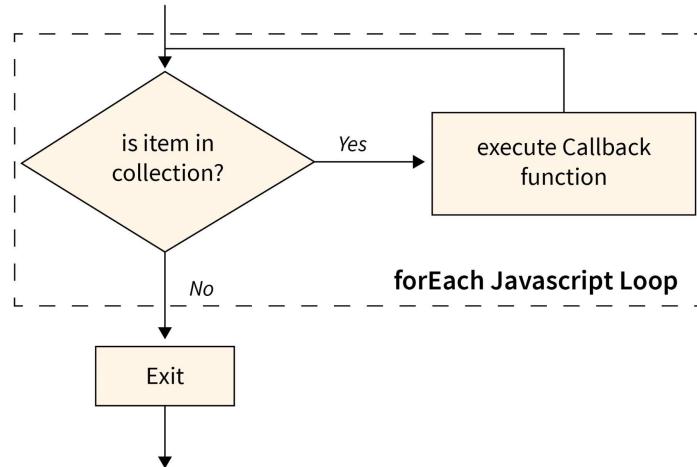
We need newer iteration methods to simplify code, enhance readability, and enable efficient operations like forEach, mapping, filtering and reducing arrays.



Let's talk about forEach and map methods

Iteration Methods: `forEach`

The `forEach()` method executes a provided function once for each array element but does not create a new array.



You need not to write test condition, neither to initialize nor to increment/decrement anything

forEach syntax

Let's have a look at its syntax:-

Syntax:

```
originalArray.forEach((element, index, array) => {  
    // Action logic here  
});
```



Parameters:

- **element:** The current element being processed.
- **index:** The index of the current element.
- **array:** The original array being traversed.

Can you observe that there is a function passed to forEach method? What kind of function it is??

forEach example

Let's learn to write forEach method, here we will declare and assign an array and iterate over it using forEach method. Let's begin:-

```
1 let arr = [1, 2, 3, 4];
2
3 // Iterating using forEach
4 arr.forEach((num) => {
5     console.log("Number: ", num);
6     // Logs each number in the array
7 })
8
9 // Output:
10 // Number: 1
11 // Number: 2
12 // Number: 3
13 // Number: 4
```

Can you appreciate how easy it is compared to traditional javascript loops?

forEach comparison with for loop

Let's compare it with for loop of javascript, which one is easier to write?



```
1 let arr = [1, 2, 3, 4];
2
3 // Iterating using forEach
4 arr.forEach((num) => {
5     console.log("Number: ", num);
6     // Logs each number in the array
7 })
8
9 // Output:
10 // Number: 1
11 // Number: 2
12 // Number: 3
13 // Number: 4
```

forEach



```
1 let arr = [1, 2, 3, 4];
2
3 // Iterating using for loop
4 for (let i = 0; i < arr.length; i++) {
5     console.log("Number: ", arr[i]);
6     // Logs each number in the array
7 }
8
9 // Output:
10 // Number: 1
11 // Number: 2
12 // Number: 3
13 // Number: 4
```

for loop

When to use what: for and forEach

Let's compare it with the for loop in JavaScript; which one is easier to write??

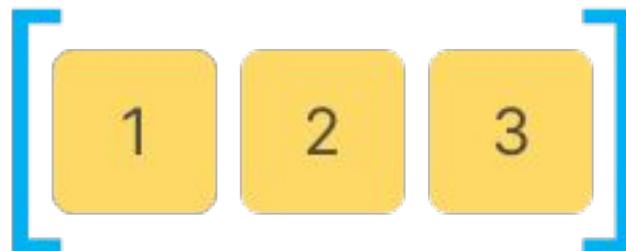
for loop	forEach
When you want greater flexibility, as you can decide start index, test condition, etc.	forEach is utilized for its simplicity and readability, providing a straightforward way to iterate over array
You can conditionally break or skip code based on condition using break and continue. Hence optimize the performance	You cannot use the break and continue statement and hence can't optimize the performance.

Practice Question

Store 10 student names in an array and use a
forEach loop to print their names one by one.

Iteration Methods: What is map()

map is used to iterate over an array and creates a new array with the results, while forEach simply iterates without returning anything.



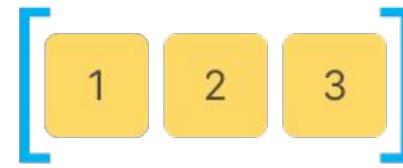
.map($n \Rightarrow n*2$) → [2, 4, 6]

Here we are iterating over an array and returning a new array with values doubled.

map() Example 1

Let's write code for implementing it:-

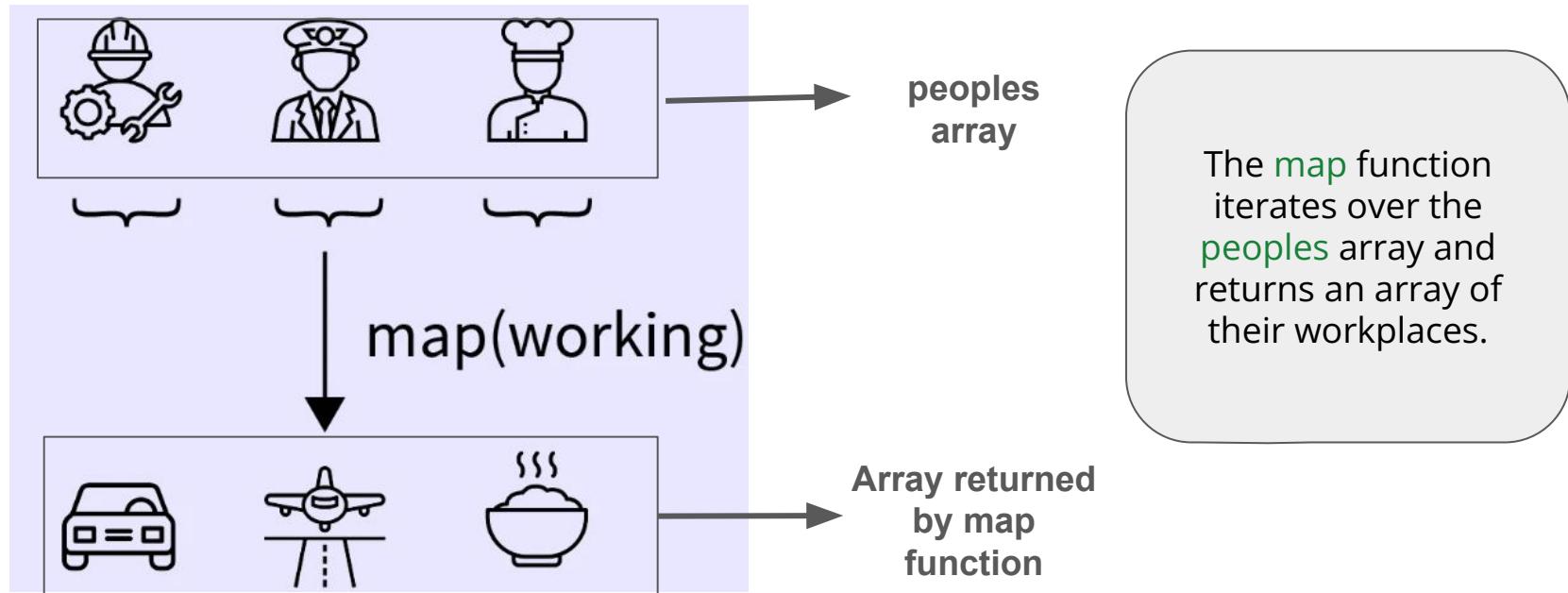
```
1 // Original array
2 let numbers = [1, 2, 3];
3
4 // Using map to create a new array with doubled values
5 let doubledNumbers = numbers.map(num => num * 2);
6
7 // Printing the result
8 console.log(doubledNumbers); // Output: [2, 4, 6]
```



.map(n => n*2) → [2, 4, 6]

map() example 2

You are provided with a workers array and you need to iterate over that array and return an array listing their workplaces.



map() example 2

Let's write code example of illustration in previous slide:-

```
1 // Array of people with their details
2 let people = [
3     { name: "Ram", designation: "Mechanic", workplace:
4         "Garage" },
4     { name: "Shyam", designation: "Pilot", workplace:
5         "Airport" },
5     { name: "Aryan", designation: "Chef", workplace:
6         "Restaurant" }
6 ];
7
8 // Extracting workplaces using map
9 let workplaces = people.map(person => person.workplace);
10
11 // Printing the workplace array
12 console.log(workplaces);
13 // Output: ["Garage", "Airport", "Restaurant"]
```

Here we have iterated over people array and returned an array which lists their workplaces and stored in a variable workplaces.

Practice Question

Use map to convert an array of strings to their uppercase versions.

Difference: `forEach` and `map` method

Let's observe the difference between the two:-

<code>forEach</code>	<code>map</code>
Return value: undefined	Return value: newArray will be created based on your callback function
Original Array: not modified	Original Array: not modified
<code>newArray</code> is not created after the end of a method call	<code>newArray</code> is not created after the end of a method call

Higher Order functions

Higher order functions: map()

In previous illustrations we have used .forEach() and .map() method, these functions are taking a callback function as input. These type of functions are called higher order functions.



```
1 const arr = [1, 2, 3, 4, 5];
2
3 const printItem = function(item) {
4     console.log("Item: ", item);
5 }
6
7 arr.map(printItem);
```

Here we passed a callback function to map which is higher order function

Higher order functions: forEach()

Similarly forEach() is also a higher order function as it receives a callback function as an argument.



```
1 const arr = [1, 2, 3, 4, 5];
2
3 const printItem = function(item) {
4     console.log("Item: ", item);
5 }
6
7 forEach(printItem);
```

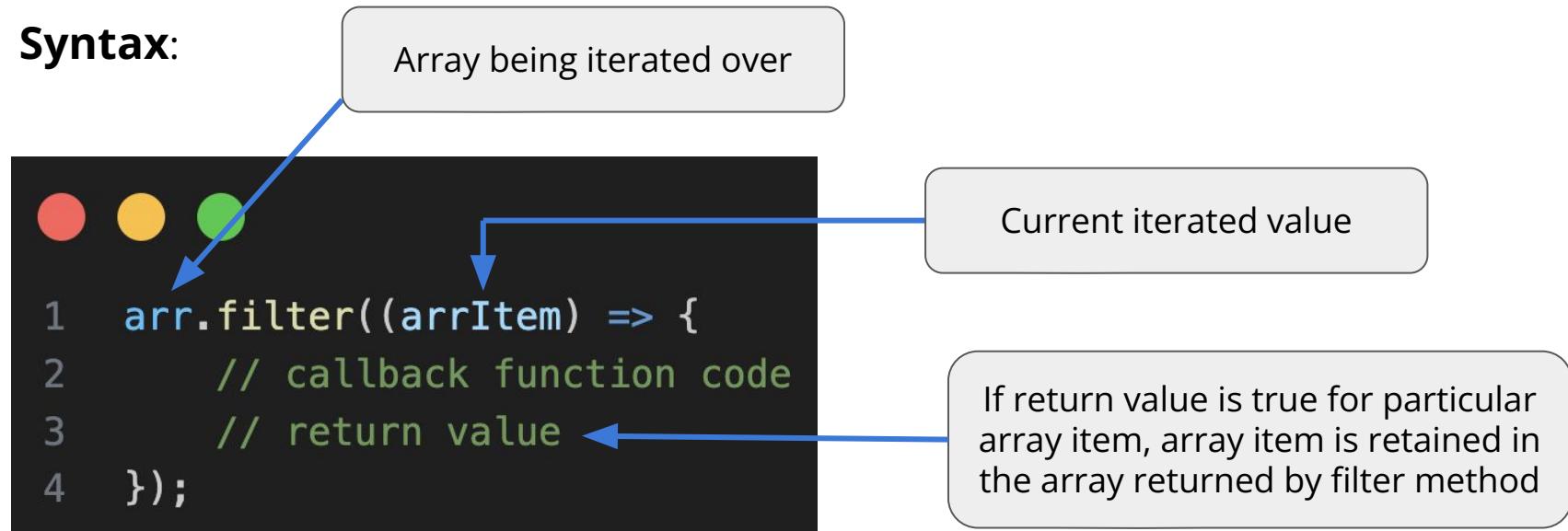
Similarly we would have a look into more higher order functions in upcoming slides.

Callback function passed to higher order function forEach()

Search/Filter Methods: filter()

The `filter` method returns a new array with values from the original array that satisfy the condition in the callback function.

Syntax:



Search/Filter Methods: filter()

Let's say you have an array containing food items and it need to be iterated and return just items having price over 15. Here is diagrammatic representation of the solution:-

```
.filter((food_item)⇒{return food_item.price > 15})
```



Here only those items got filtered out whose price was more than 15.

filter() example

Let's write code for example in the previous slide:-

```
1 let foodItems = [  
2     { name: "milk", price: 12 },  
3     { name: "bread", price: 17 },  
4     { name: "cake", price: 20 },  
5     { name: "eggs", price: 5 }  
6 ];  
7  
8 let filteredItems = foodItems.filter(function(foodItem){  
9     return foodItem.price > 15;  
10});  
11  
12 console.log(filteredItems);  
13 // Output:  
14 // [ { name: 'bread', price: 17 },  
15 //   { name: 'cake', price: 20 } ]
```

Here bread and
cake is filtered
out.



Practice Question

Write a function using a filter to return all elements in an array greater than a given value.

Reduction Methods: What is reduce()

The `reduce` method transforms an array into a single value by applying a function to each element, using an accumulator to store the result. After processing all elements, it returns the final value.



Just like mixing ingredients to make a dish, the `reduce` method combines values to produce a single result.

reduce(): syntax

The `reduce` method iterates through an array and applies a callback function to accumulate all elements into a single value, such as a sum, product, array, or object.

Syntax:

Cumulative value that accumulates the results



```
1 arr.reduce((accumulator, currentValue) => {  
2     // reduce method code  
3 }, initialAccumulatorVal)
```

Current iterated value of the array

Initial value of accumulator

reduce() example 1

Here we were provided with an array, we can use reduce() method to calculate the sum and return the value 14.



reduce() example 1

Let's implement that example:-



```
1 const numbers = [1, 2, 3, 4];
2
3 // Using reduce to calculate the sum
4 const sum = numbers.reduce((acc, num) => acc + num, 0);
5
6 console.log(sum); // Output: 10
```

reducer function accepts these arguments:-

1. Accumulator(acc): cumulative value that accumulates the results
2. Current Value(num): current value being processed

reduce() example 1

The `reduce` method iterates over the array, adding each value to an accumulator, starting from `0`, and returns the final sum after processing all elements. In this case, the sum of `[1, 2, 3, 4]` is `10`.

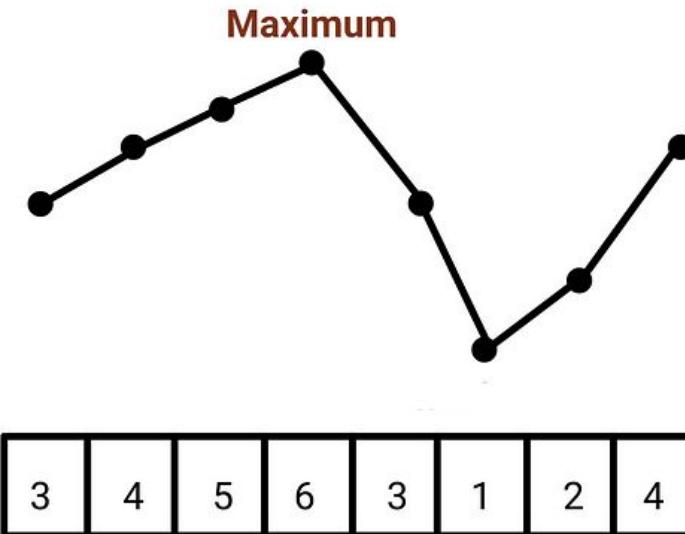
1	2	3	4
---	---	---	---

$$\begin{aligned} & \underbrace{1+2=3}_{\text{Step 1}} \\ & \underbrace{3+3=6}_{\text{Step 2}} \\ & \underbrace{6+4=10}_{\text{Step 3}} \end{aligned}$$

The accumulator starts at zero, adding values at each step, and the final sum is calculated at the end.

reduce() example 2

Here given an array find out maximum value by scanning the array using reduce method.



You might be tempted to use forEach or map method.
Wait...wait... there is even better way i.e. reduce()

reduce() example 2

Let's write the javascript code using reduce().

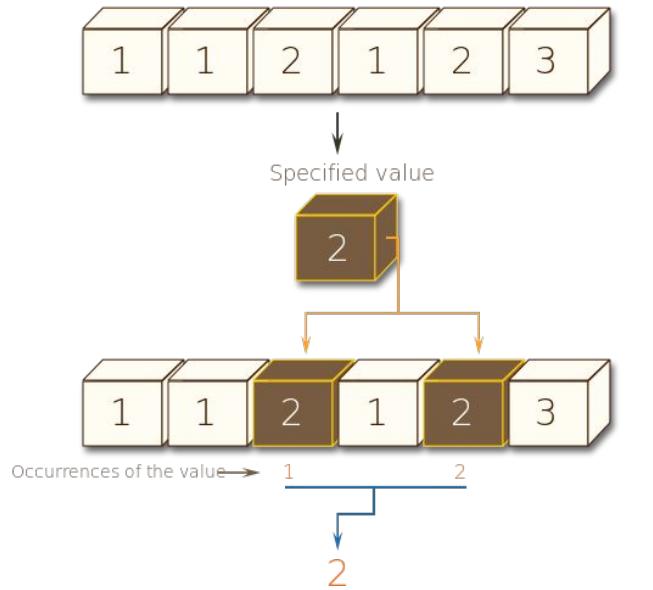


```
1 const numbers = [3, 4, 5, 6, 3, 1, 2, 4];
2
3 const maxValue = numbers.reduce((max, current) => {
4     return current > max ? current : max;
5 }, numbers[0]); // Initialize with the first element of the array
6
7 console.log(`The maximum value is: ${maxValue}`);
8 // Output: The maximum value is: 6
```

Observe how crisp and clear the code is with reduce().

reduce() example 3

Provided an array of integers find out how many times integer value 2 got repeated using reduce method.



Do this one by
yourself..

reduce() example 3

Let's have a look at the solution:-



```
1 const numbers = [1, 1, 2, 1, 2, 3];
2
3 const count = numbers.reduce((total, current) => {
4     return current === 2 ? total + 1 : total;
5 }, 0); // Initialize count to 0
6
7 console.log(`The number 2 appears ${count} times.`);
8 // Output: The number 2 appears 2 times.
```

Were you able to do
it by yourself??

Practice Question

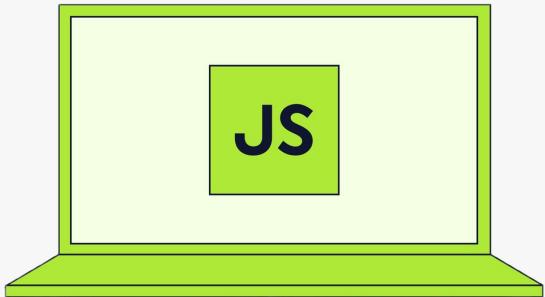
Find the first string in the array having more than 5 characters.

In Class Questions

**Thanks
for
watching!**



The Complete Javascript Course



@newtonschool

Lecture 7: Introduction to Objects

-Vishal Sharma



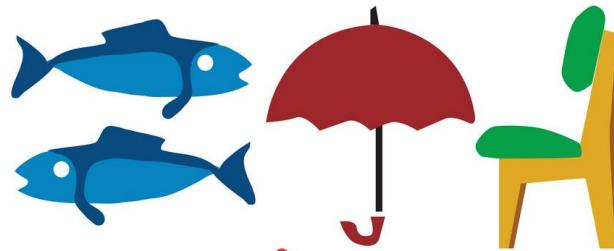
Table of Contents

- Creating Objects
 - What are objects? Real-world analogies
 - Why use Object vs Other Data Types
 - Objects as key value pairs
- Accessing Properties
 - Dot Notation Deep Dive
 - Bracket Notation in Depth
- Modifying Objects
 - Adding Properties
 - Updating Properties
 - Deleting Properties
 - Iterate over an object using for...in loop
 - Iterate over an object using for...of loop

Creating Objects

Objects in real world

In the real world, **objects** are entities or things that have distinct characteristics and behaviors. Objects can be described by their **properties** (attributes) and can perform **actions** (methods).



Each of these objects has some physical properties as well as some functionality. Can you name a few for each of them?

Objects in real world: Example

Let's have a look at this car. It has certain properties and some functions/methods it can perform.



Properties:-

1. color: red
2. make: Toyota
3. year: 2023
4. etc.

Methods/Functions:-

1. start
2. drive
3. etc.

Objects in Programming

They are collections of related data and functionality grouped together, often mirroring real-world entities. Let's write an object from the car from the previous slide.



```
1 const car = {  
2     // Properties  
3     color: "red",  
4     make: "Toyota"  
5     year: 2023  
6  
7     // Methods  
8     start: function() {  
9         console.log("The car has started.");  
10    },  
11    drive: function() {  
12        console.log("The car is driving.");  
13    }  
14};
```

In JavaScript, methods are also considered properties of an object.

Person as an Object

Similarly, we can create an object to represent a person. Imagine Ajay, a busy professional juggling multiple tasks in a day. Ajay has distinct traits like a name, black hair, fair skin, etc along with several actions and functions it can perform.

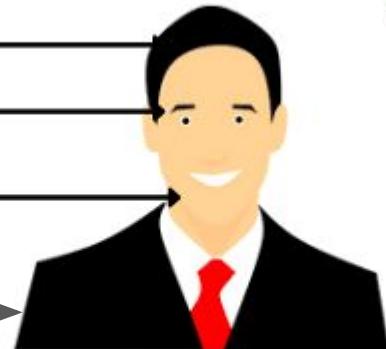
Object's properties

Black hair

Black eyes

Fair skin

Ajay



Object's actions

Eat

Sleep

Walk

Exercise

Attend Meeting

A person is an object

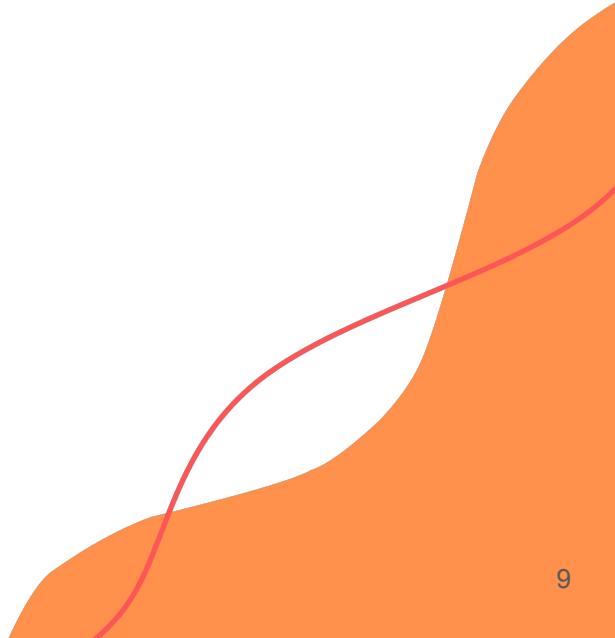
Can you transform this description into an object? Function might only contain console.log() for now?

The Persona of Ajay: A Living Object

Here your implementation might differ; I have only taken a few actions as object methods.

```
1 const ajay = {
2     // Properties
3     name: "Ajay",
4     hairColor: "black",
5     skinTone: "fair",
6
7     // Methods
8     attendMeeting: function() {
9         console.log(` ${ajay.name} is attending a meeting.`);
10    },
11    exercise: function() {
12        console.log(` ${ajay.name} is cycling to stay fit.`);
13    }
14};
15
16 // Example Usage
17 ajay.attendMeeting();
18 console.log(ajay.name);
```

Now you should feel confident writing an object by yourself.



Object Creation

Workshop

Workshop: Question

Create an object to represent a smartphone with the following properties: brand, model, price, and features (as an array). And write some methods like messaging, calling, etc.

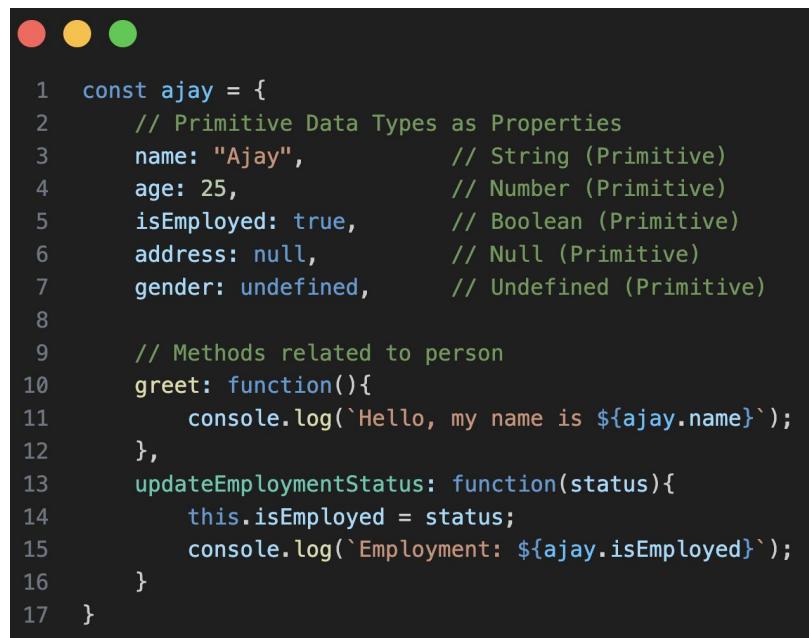
Be confident, and you will be able to write in one go.



Objects vs Other Data Types

Objects vs Primitives: A Unified Concept

Objects are derived from primitive types but differ in storing and managing data, grouping related data and behaviors for more flexibility.



```
1 const ajay = {
2     // Primitive Data Types as Properties
3     name: "Ajay",           // String (Primitive)
4     age: 25,                // Number (Primitive)
5     isEmployed: true,       // Boolean (Primitive)
6     address: null,          // Null (Primitive)
7     gender: undefined,      // Undefined (Primitive)
8
9     // Methods related to person
10    greet: function(){
11        console.log(`Hello, my name is ${ajay.name}`);
12    },
13    updateEmploymentStatus: function(status){
14        this.isEmployed = status;
15        console.log(`Employment: ${ajay.isEmployed}`);
16    }
17 }
```

Notice how various data types are included inside the object named 'ajay'.

How objects are different from primitives

They differ in the following ways:-

Aspect	Primitives	Objects
Structure	Simple, single values like numbers or text.	Complex things that store many related values or actions.
Storage	Small and fast.	Bigger and slower.
Access	Accessed by identifier storing value.	Accessed via properties or methods (e.g., <code>object.property</code>).
Behavior	No inherent behavior; only stores data.	Can include methods for processing data.
Example	<code>var car = "Toyota"</code>	Visit slide 6 to see the example.

We will discuss access methods in later slides

Objects as Key-Value Pairs

Objects as key-value pairs

Objects are essentially collections of related information represented as **key-value pairs**. Values stored are accessible via that key.

key	value
name	Ajay
age	30
occupation	Engineer



```
1 let person = {  
2   name: "Ajay",  
3   age: 30,  
4   occupation: "Engineer"  
5 };  
6  
7 console.log(person.name); // Output: "Ajay"  
8 console.log(person.age); // Output: 30
```

Objects as key-value pairs

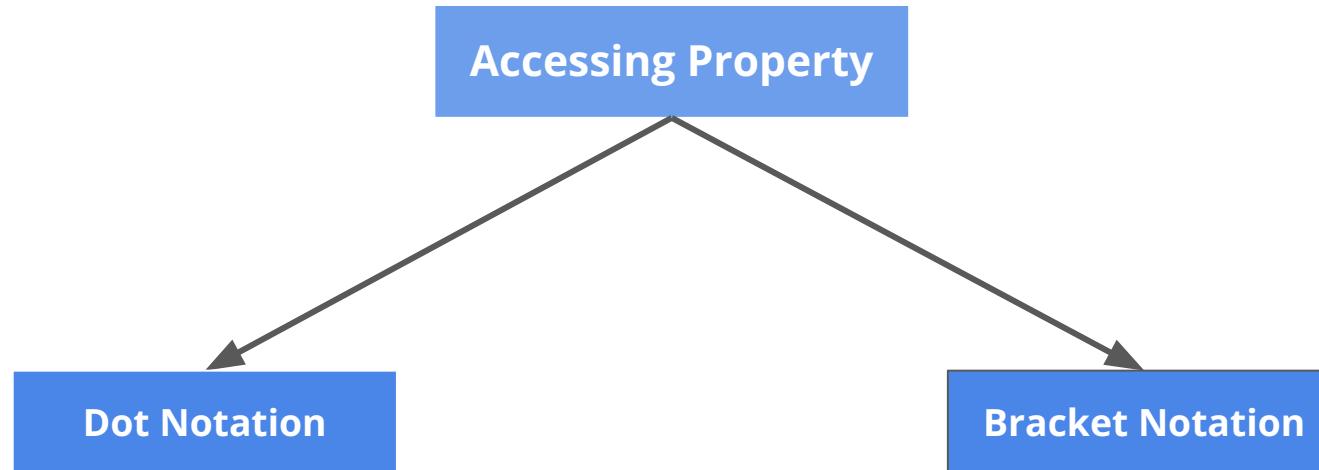
Methods also need to be accessed via its key as it is also considered as property.

```
1 let person = {  
2     name: "Ajay",  
3     greet: function() {  
4         console.log("Hello, ", + person.name);  
5     }  
6 };  
7  
8 person.greet(); // Output: Hello, Ajay
```

Accessing Properties

Different ways to access properties

There are two main ways to access the properties of an object



Accessing Property: Dot Notation

In JavaScript, dot notation is the simplest and most common way to access the properties of an object. You use a dot (.) followed by the property name.

```
object.property
```

- `object` is the object you're working with.
- `property` is the name of the property you want to access.

Dot Notation: example

Let's revisit the car object with the properties `color`, `make`, and `year`, and then access those properties using dot notation.

```
1 let car = {  
2     color: "Red",  
3     make: "Toyota",  
4     year: 2023  
5 };  
6  
7 console.log(car.color); // Output: Red  
8 console.log(car.make); // Output: Toyota  
9 console.log(car.year); // Output: 2023
```

Accessing
properties using dot
notation

Dot Notation: example

Accessing the methods are no different.

```
1  let car = {  
2      color: "Red",  
3      make: "Toyota",  
4      year: 2023,  
5      displayInfo: function() {  
6          console.log(`This is a ${car.year} ${car.make} ${car.color} car. `);  
7      }  
8  };  
9  
10 // Accessing and calling the method using dot notation  
11 car.displayInfo();  
12 // Output: this is a 2023 Toyota Red car.
```

Accessing Property: Bracket Notation

Bracket notation is used to access both properties and methods of an object, especially when the property names are dynamic, contain spaces, or are stored in variables.

```
object[property]
```

Use single or double inverted commas in case property is a string.

Bracket Notation: example

Let's reuse the car example, object creation part remains the same but we will use brackets instead of dot to access the object property.



```
1 // Accessing properties using bracket notation
2 console.log(car["color"]); // Output: Red
3 console.log(car["make"]); // Output: Toyota
4 console.log(car["year"]); // Output: 2023
5
6 // Accessing and calling the method using bracket notation
7 car["displayInfo"]();
8 // Output: This is a 2023 Toyota Red car.
```

This notation is very useful in case property name is dynamic i.e. changes frequently maybe based on some condition.

Bracket Notation: example

Accessing the methods are no different.

```
1 // Let's say the property name is dynamic, based on a condition
2 let propertyName = "make";
3 // This can change based on some condition
4
5 // Accessing the property dynamically using bracket notation
6 console.log(car[propertyName]);
7 // Output: Toyota
8
9 // Changing the condition
10 propertyName = "year";
11
12 // Accessing another property based on the updated condition
13 console.log(car[propertyName]); // Output: 2023
```

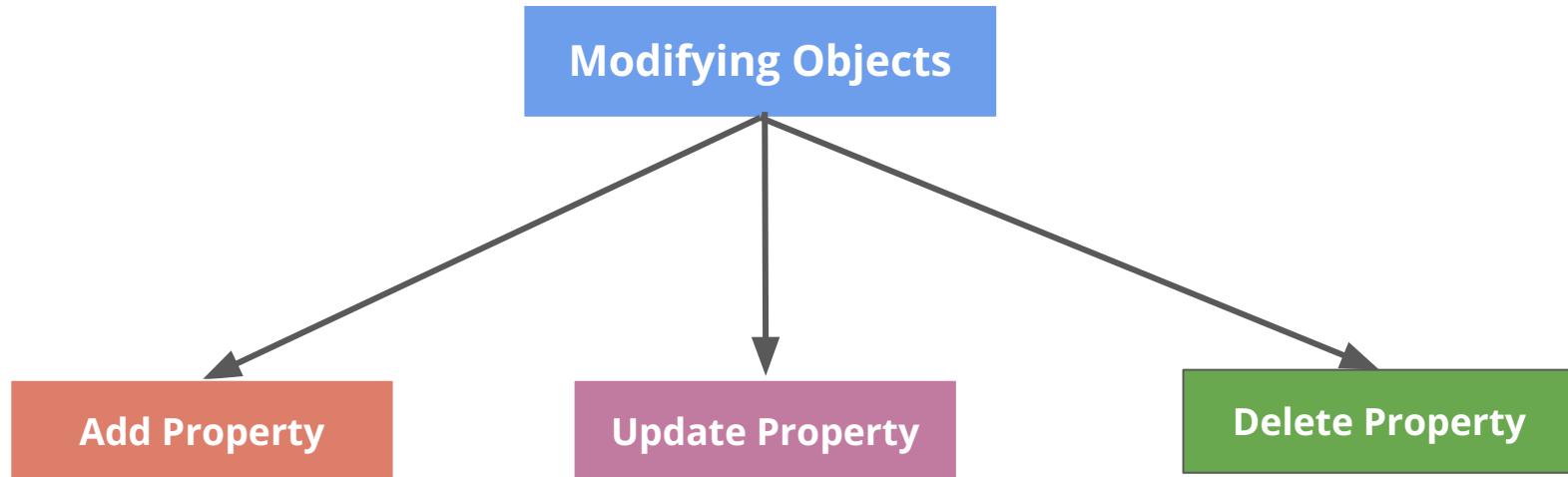
Here we are using `propertyName` to dynamically access different properties of the `car` object.

Modifying Objects

Why Modify Objects?

In JavaScript, objects often need to change to reflect new information.

We'll explore three ways to modify objects:



How to add properties

You can add new properties to an object anytime, expanding it dynamically. For example, if you forget to define the car color, you can add it later like this:



```
1 let car = { make: "Toyota", year: 2023 };
2
3 car.color = "Red"; // Adding a new property
4
5 console.log(car.color); // Output: Red
```

We added a new property using the dot operator and assigned it the value "Red".

Adding a method property

Here we have added a method property using dot operator.

```
1 let car = {  
2     make: "Toyota",  
3     year: 2023  
4 };  
5  
6 // Adding a method using dot notation  
7 car.startEngine = function() {  
8     console.log("The engine has started.");  
9 };  
10  
11 // Calling the method  
12 car.startEngine();  
13 // Output: The engine has started.
```

Try adding one more method property by yourself.

How to Update Properties

You can modify the value of an existing property just the way you create a new property.

```
1 let car = {  
2     make: "Toyota",  
3     year: 2023,  
4     color: "Red"  
5 };  
6  
7 car.color = "Blue";  
8 // Updating an existing property  
9  
10 console.log(car.color);  
11 // Output: Blue
```

Earlier color was 'Red', we have later changed the value to 'Blue' by using assignment operator.

How to Update Properties

In the similar fashion we can update method properties too.

```
1 let car = {  
2   make: "Toyota",  
3   year: 2023,  
4   startEngine: function() {  
5     console.log("The engine has started.");  
6   }  
7 };  
8  
9 // Updating the startEngine method  
10 car.startEngine = function() {  
11   console.log("The engine starts with a roar!");  
12 };  
13  
14 // Calling the updated method  
15 car.startEngine();  
16 // Output: The engine starts with a roar!
```

Sounds simple right??

How to Delete Properties

You can remove properties from objects using the `delete` keyword.

```
1 let car = {  
2   make: "Toyota",  
3   year: 2023,  
4   color: "Red"  
5 };  
6  
7 delete car.color;  
8 // Deleting a property  
9  
10 console.log(car.color);  
11 // Output: undefined
```

Caution

- Deleting a property makes it completely unavailable in the object.
- The deleted property cannot be recovered, but the object itself remains intact.

We got `undefined` because the property was deleted.

How to Delete Properties

Methods can be deleted from an object just like properties using the `delete` operator.



```
1 let car = {  
2     make: "Toyota",  
3     year: 2023,  
4     startEngine: function() {  
5         console.log("The engine has started.");  
6     }  
7 };  
8  
9 // Deleting the startEngine method  
10 delete car.startEngine;  
11  
12 // Trying to call the deleted method  
13 console.log(car.startEngine);  
14 // Output: undefined
```

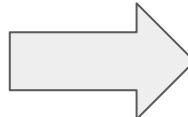
That's it!

Iterating over an Object

Why Iterate Over Objects?

Objects store key-value pairs, but unlike arrays, they lack built-in iteration methods. We need to loop through an object's properties to access each key and value.

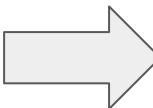
key	value
name	Ajay
age	30
occupation	Engineer



We may need to iterate over this object to display its values, but there are also many practical scenarios where iteration is essential.

How to Iterate: for...in loop

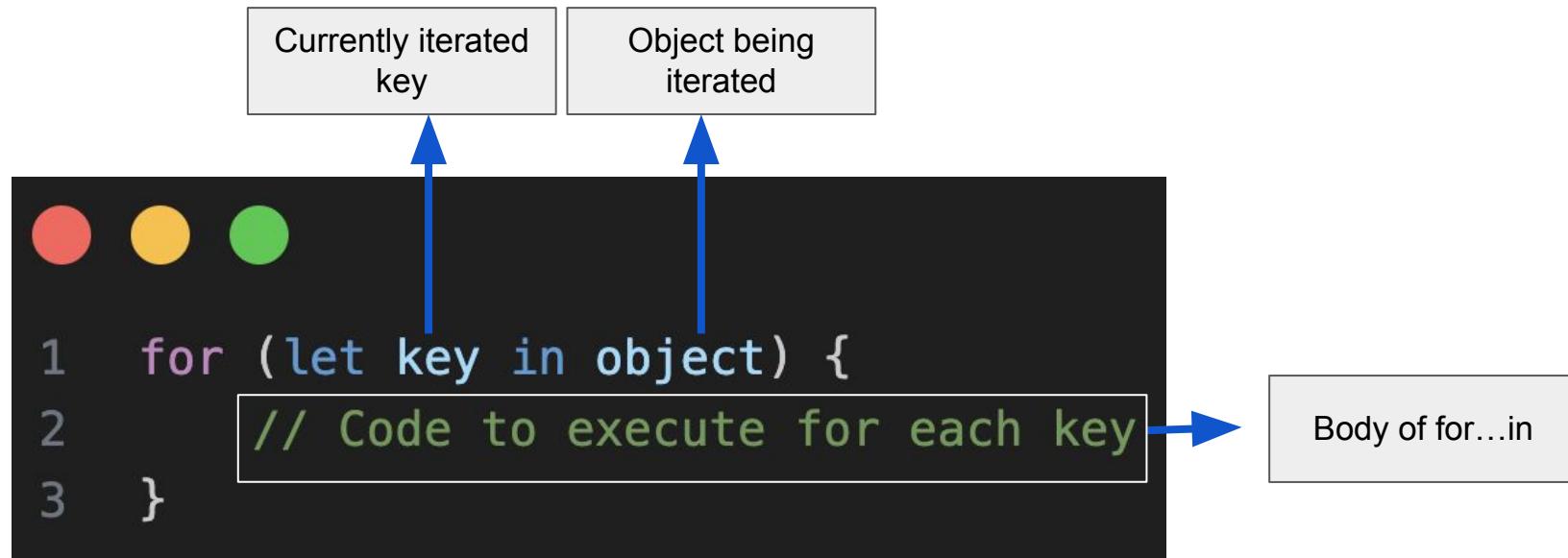
We can iterate over an object using for...in loop



for...in loop is here for rescue

Iterate Object: for...in loop

The `for...in` loop is used to iterate over the keys of an object, allowing you to access both the key and its corresponding value.



for...in: Example

Let's iterate over the `person` object (with name, age, and occupation) and display its keys and values.

```
1 const person = {  
2     name: "Ajay",  
3     age: 30,  
4     occupation: "Engineer"  
5 };  
6  
7 for (let key in person) {  
8     console.log(key, ":", person[key]);  
9 }
```

Output:

```
name : Ajay  
age: 30  
occupation: Engineer
```

Iterate: for...of loop

Plain objects ({}) are not directly iterable with `for...of`. However, we can make them iterable by using methods like `Object.keys()`.



```
1 const obj = { a: 1, b: 2, c: 3 };
2
3 // Iterating over the object using Object.keys
4 for (let key of Object.keys(obj)) {
5     console.log(key); // Outputs: a, b, c
6 }
```

Output:

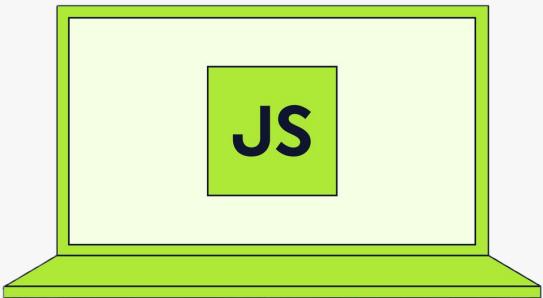
```
a  
b  
c
```

In Class Questions

**Thanks
for
watching!**



The Complete Javascript Course



@newtonschool

Lecture 8: Object Utility Methods

-Vishal Sharma



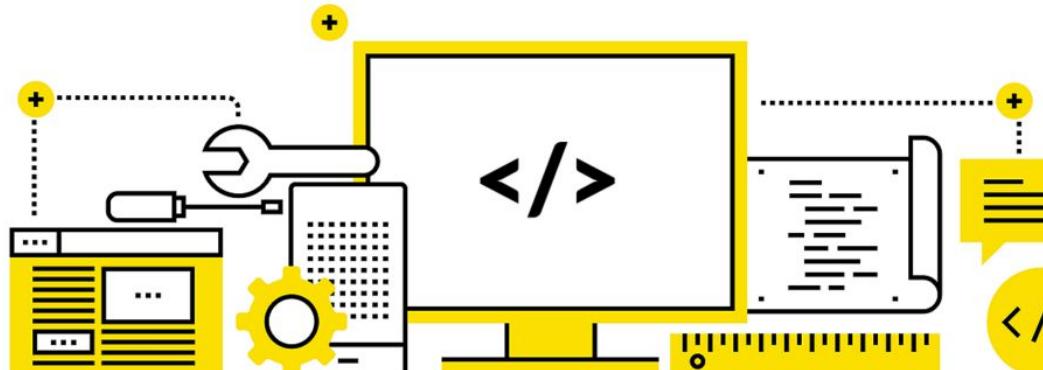
Table of Contents

- Introduction to Object utilities
- Data Retrieval Utilities
 - Object.keys()
 - Object.values()
- Copying Objects: Deep Copy vs Shallow Copy
- Practical Applications
 - Using map method to iterate object values
 - Using filter to filter out object entries

Introduction to Object Utilities

What are Object Utilities?

Object utilities in programming, especially in JavaScript, are built-in or custom methods, functions, or techniques used to manipulate, transform, or analyze objects.



They allow to access and manipulate objects in much simpler and efficient manner.

Uses of Object Utilities

Object utilities help list properties, collect values, pair data, and rebuild structured information.

Object.keys()
List properties



Object.values()
Collect Values

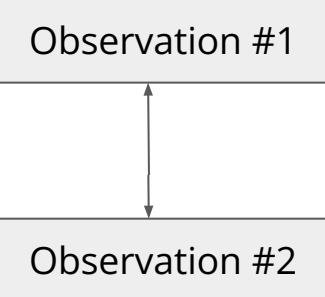
My Collection

Collectibles
2,778

Valued Collectibles
1,893 

Wish List
1,684

Object.entries()
Pair Data

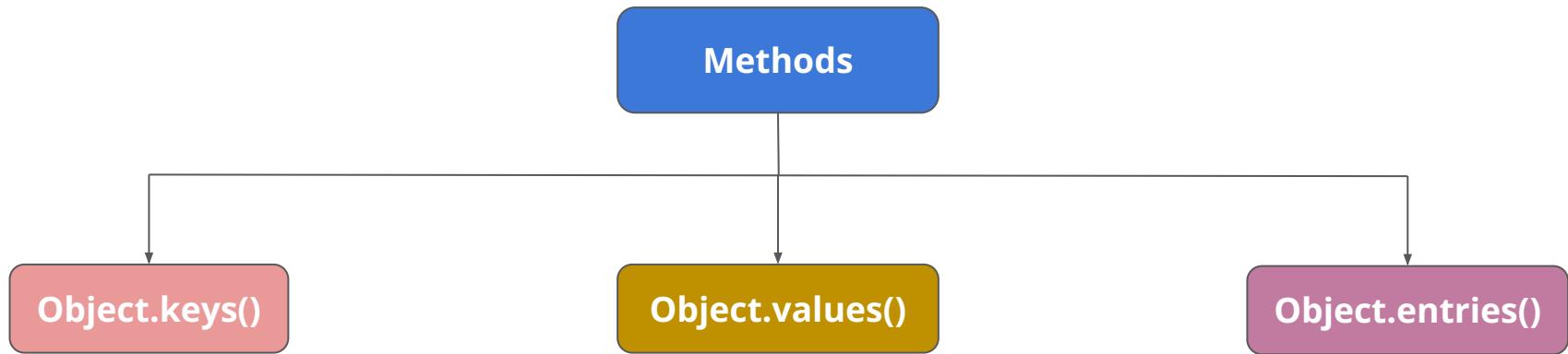


Object.fromEntries()
Rebuilds information

```
[  
  ['Username', 'Alice'],  
  ['Email', 'alice@example.com']  
]  
  
{  
  |---  
  username: 'Alice',  
  email: 'alice@example.com'  
}
```

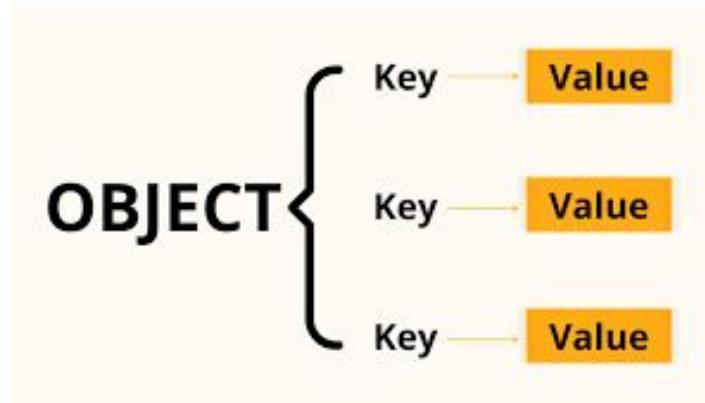
Data Retrieval Utility Methods

We have these three methods for retrieving data:-



Data Retrieval: Object.keys()

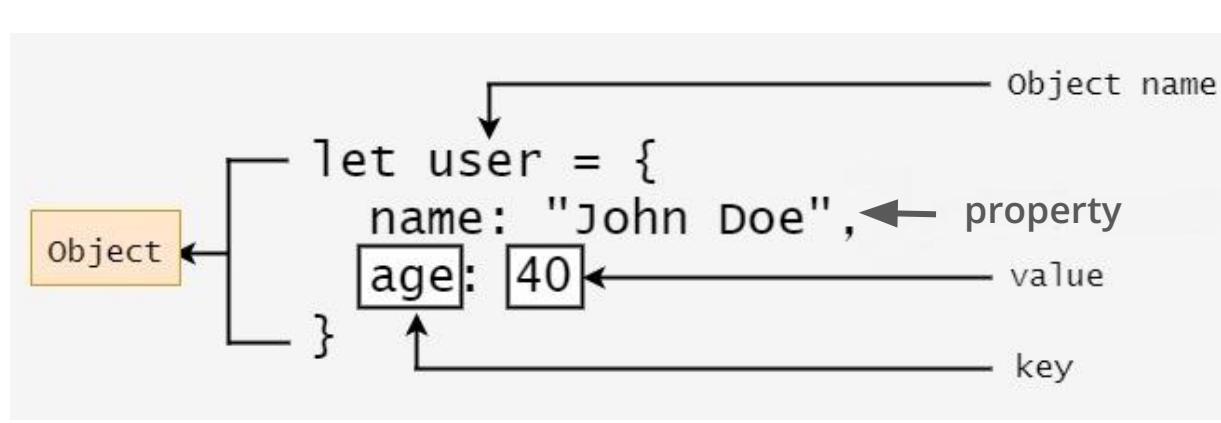
`Object.keys()` is a JavaScript method used to retrieve all the keys (property names) of an object as an array.



Each object has one or more key and these keys correspond to a value.

Data Retrieval: Object.keys()

Before we jump into Object.keys() we need to understand the structure of an object. Here there are two properties each having a key and value.



We can retrieve values of an property using keys in that object.

Data Retrieval: Object.keys()

We can retrieve the values using keys, but how to retrieve keys?? Object.keys() method comes for rescue. Here is an example:-

```
1 const person = {  
2     name: 'Alice',  
3     age: 25,  
4     city: 'New York'  
5 };  
6  
7 const keys = Object.keys(person);  
8  
9 console.log(keys);  
10 // Output: ['name', 'age', 'city']
```

Object.keys() method takes an object as an argument and returns all the keys in form of an array.

Object.keys(): Use Cases

We can use Object.keys() to get an array of keys and then can iterate over that array to access the values of that object. Here is an example:-



```
1  Object.keys(person).forEach(key => {  
2      console.log(` ${key}: ${person[key]}`);  
3  });  
4  // Output:  
5  // name: Alice  
6  // age: 25  
7  // city: New York
```



Accessing the value in the object *person* using its keys.

Object.keys(): Use Cases

Or else we can also utilize to get the number of properties stored in the object.

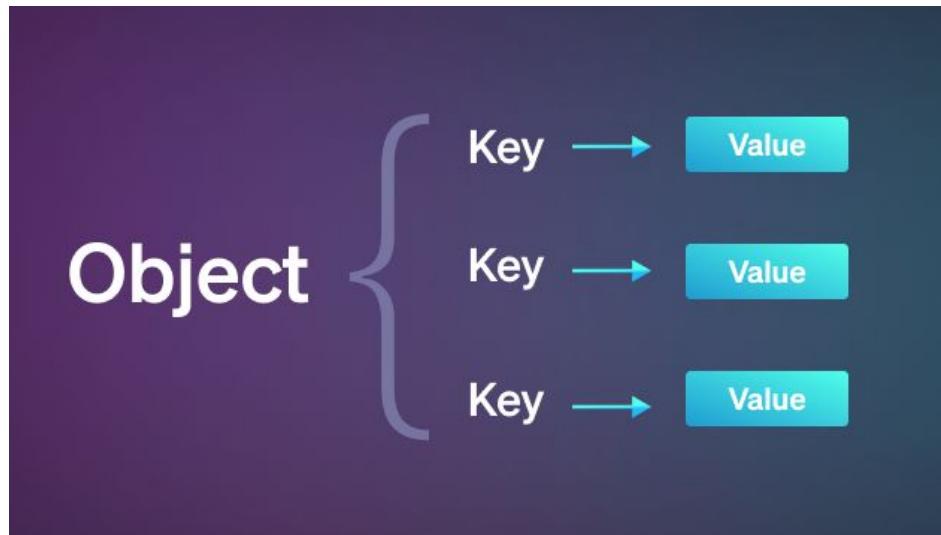


```
1 const count = Object.keys(person).length;  
2 console.log(count); // Output: 3
```

Since `Object.keys()` return an array containing all the keys, we can use array property *length* to get the number of properties stored.

Data Retrieval: Object.values()

Just like `Object.keys()` return all the keys of the object, `Object.values()` retrieves all the values of an object's enumerable properties as an array.



We can retrieve values of properties in an object using its keys. *What if we can access these values directly.*

Here comes `Object.values()` handy.

Data Retrieval: Object.values()

Let's understand it with an example:-



```
1 const values = Object.values(person);
2
3 console.log(values);
4 // Output: ['Alice', 25, 'New York']
```

Here we retrieved all
the values of properties
stored in the Object.

Data Retrieval: Use Cases

We can iterate all values using forEach loop.

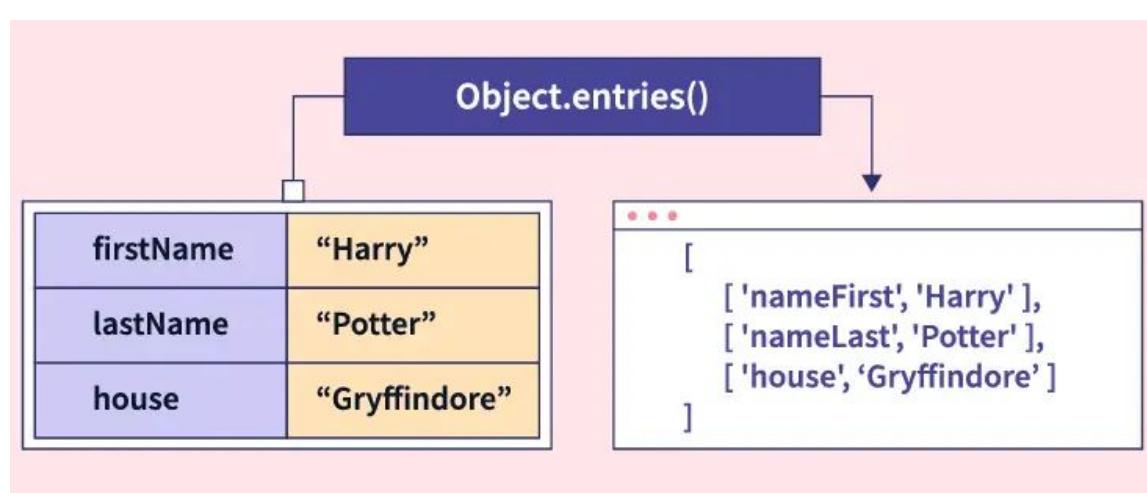


```
1 Object.values(person).forEach(value => {
2   console.log(value);
3 });
4 // Output:
5 // Alice
6 // 25
7 // New York
```

Here we retrieved all the values of properties stored in the Object.

Data Retrieval: Object.entries()

What if we need both key and values?? We can retrieve both key and values using Object.entries.



Object.entries()
converts an object
into an array of
key-value pairs for
easy looping and
modification.

Data Retrieval: Object.entries()

Let's understand it with an example:-

```
● ● ●  
1 const character = {  
2   firstName: "Harry",  
3   lastName: "Potter",  
4   house: "Gryffindor"  
5 };  
6  
7 // Convert object into an array of key-value pairs  
8 const entriesArray = Object.entries(character);  
9  
10 console.log(entriesArray);  
11  
12 // Output:  
13 // [  
14 //   ['firstName', 'Harry'],  
15 //   ['lastName', 'Potter'],  
16 //   ['house', 'Gryffindor']  
17 // ]
```

Object enumerable properties

Converted into array

Object.entries(): Use Cases

We use Object.entries() where we need both key and value pairs at the same time.

```
1 const user = {  
2   firstName: "Harry",  
3   lastName: "Potter",  
4   house: "Gryffindor"  
5 };  
6  
7 for (const [key, value] of Object.entries(user)) {  
8   console.log(`${key}: ${value}`);  
9 }  
10 // Output:  
11 // firstName: Harry  
12 // lastName: Potter  
13 // house: Gryffindor
```

Oh! Great we can get both key, value pairs at the same time.

Data Retrieval: Use Cases

We can iterate all values using forEach loop.

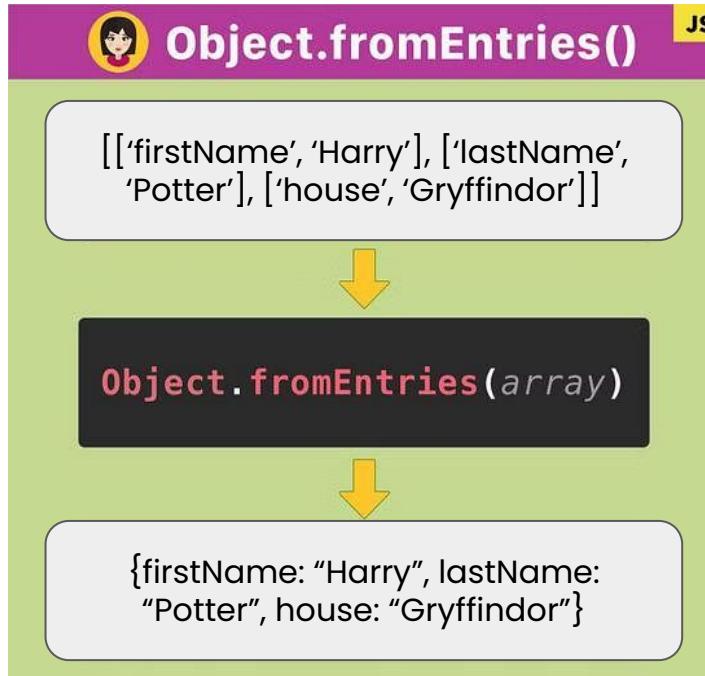


```
1 Object.values(person).forEach(value => {
2   console.log(value);
3 });
4 // Output:
5 // Alice
6 // 25
7 // New York
```

Here we retrieved all the values of properties stored in the Object.

Data Creation: Object.fromEntries()

We learned that we can convert an object to an array using Object.fromEntries(); what if we need the other way around?



We use
Object.fromEntries() to
convert an Array to an
Object.

Object.fromEntries(): Example

Let's Understand it with an example:-

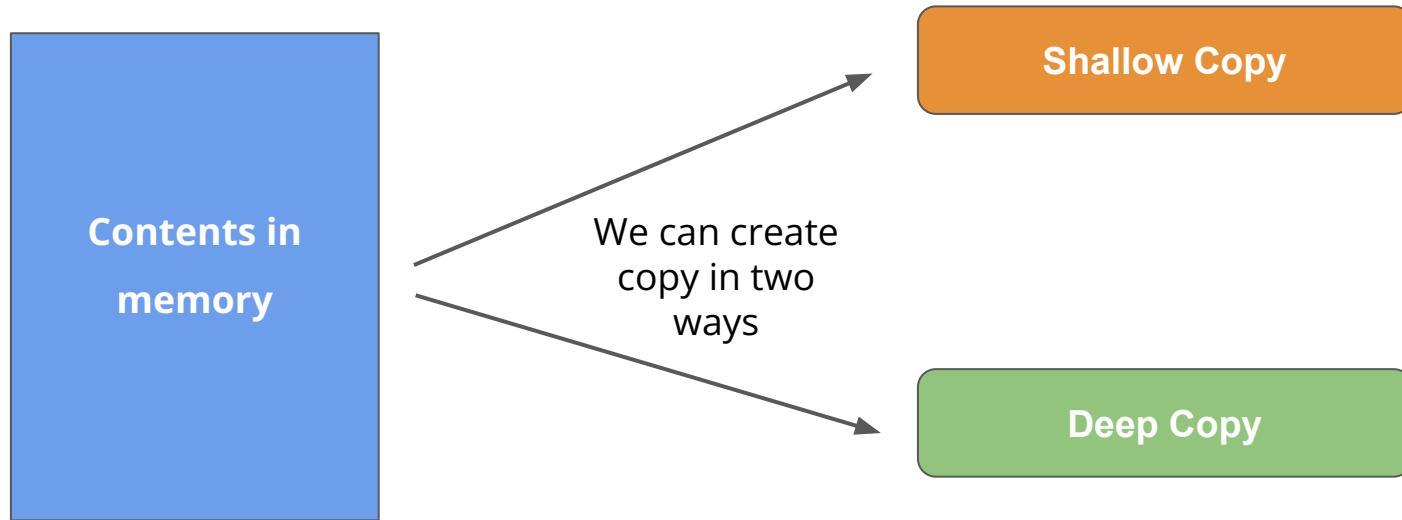
```
1 const entries = [
2   ["firstName", "Harry"],
3   ["lastName", "Potter"],
4   ["house", "Gryffindor"]
5 ];
6
7 const character = Object.fromEntries(entries);
8
9 console.log(character);
10 // Output:
11 // {
12 //   firstName: 'Harry',
13 //   lastName: 'Potter', house: 'Gryffindor'
14 // }
```

Now you should be able to convert array to object and object to array with ease.

Shallow Copy vs Deep Copy

Not all Copies are equal

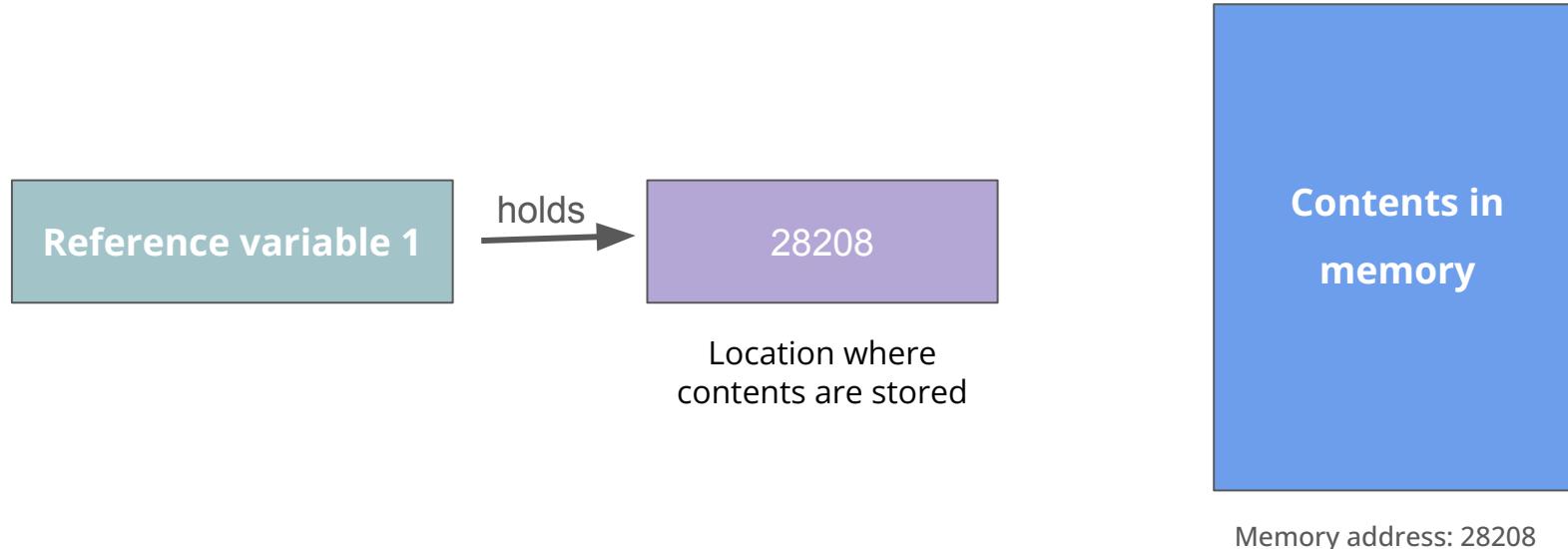
Copying an object in JavaScript isn't always simple—some copies stay connected to the original, while others create a completely separate version.



Memory address: 28208

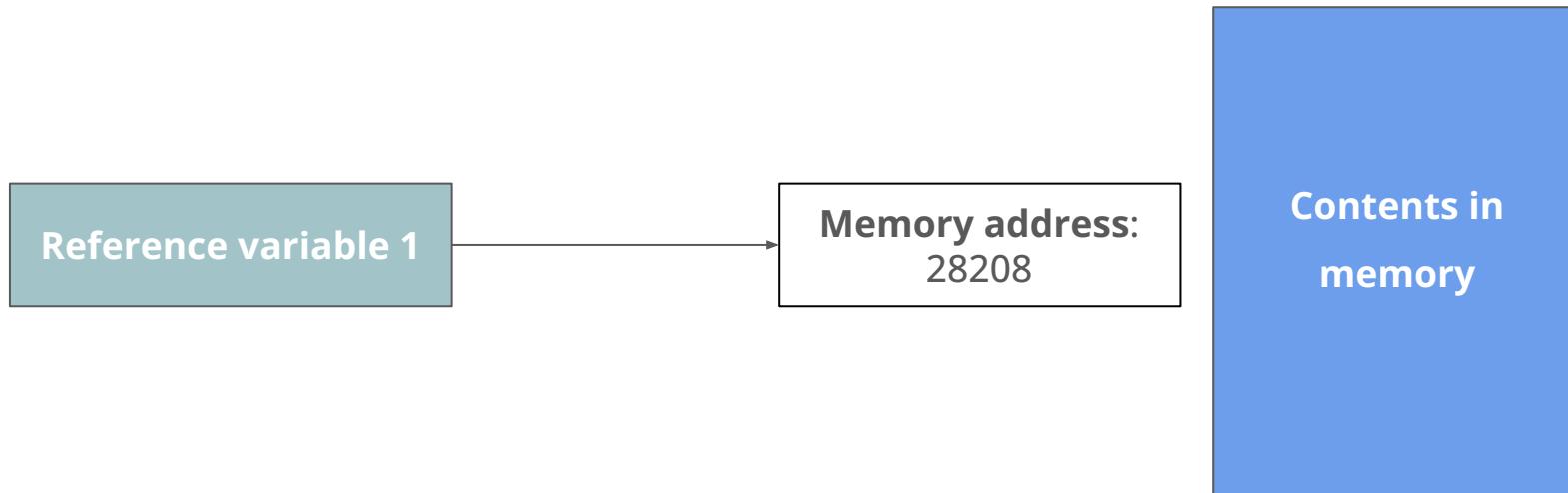
Reference Variables point to memory

In JavaScript, objects are stored in memory, and each object is assigned a memory location. The memory address of that location is stored in a reference variable.



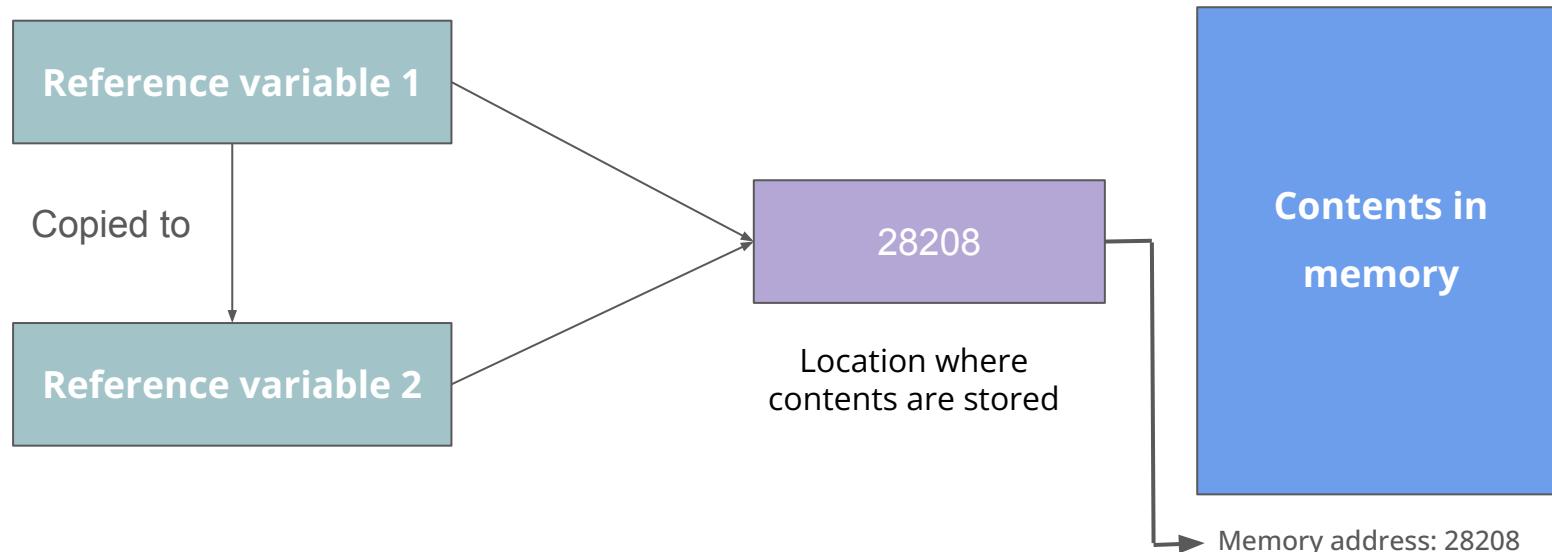
Storing address references

Thus, the reference variable points to the memory location; it doesn't store the content itself.



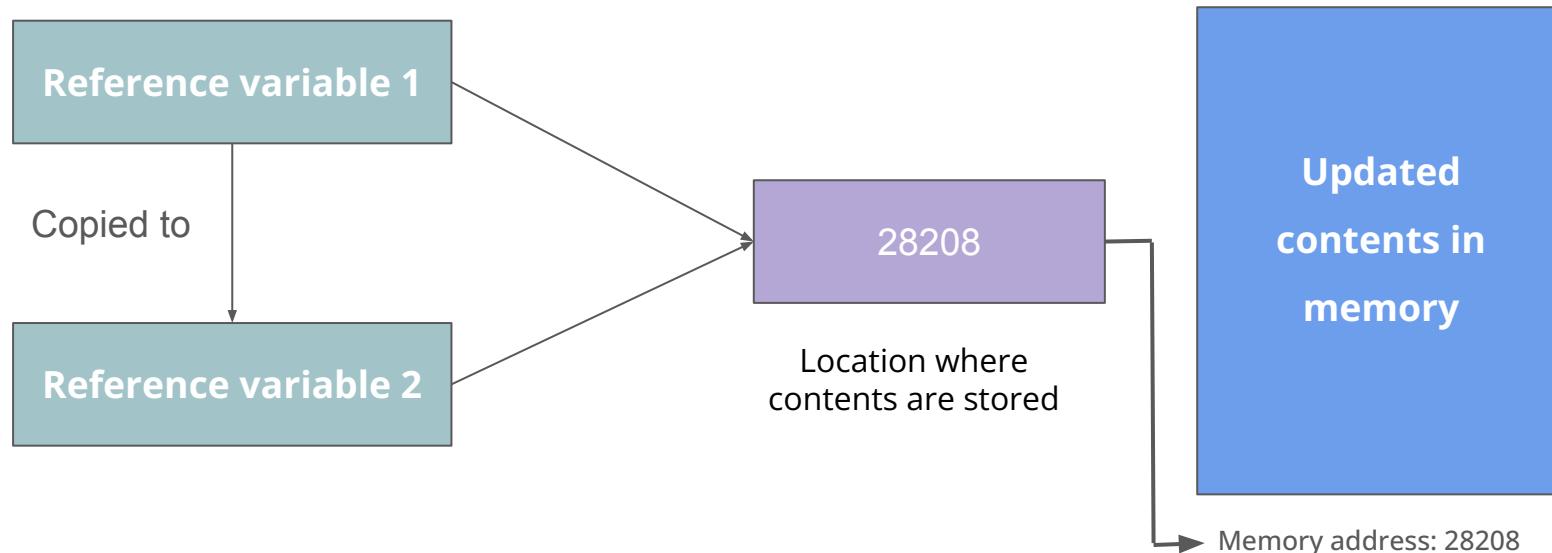
Copying reference values

Copying reference values doesn't mean copying the content itself but memory reference to that content. Such a copy is called **a shallow copy**.



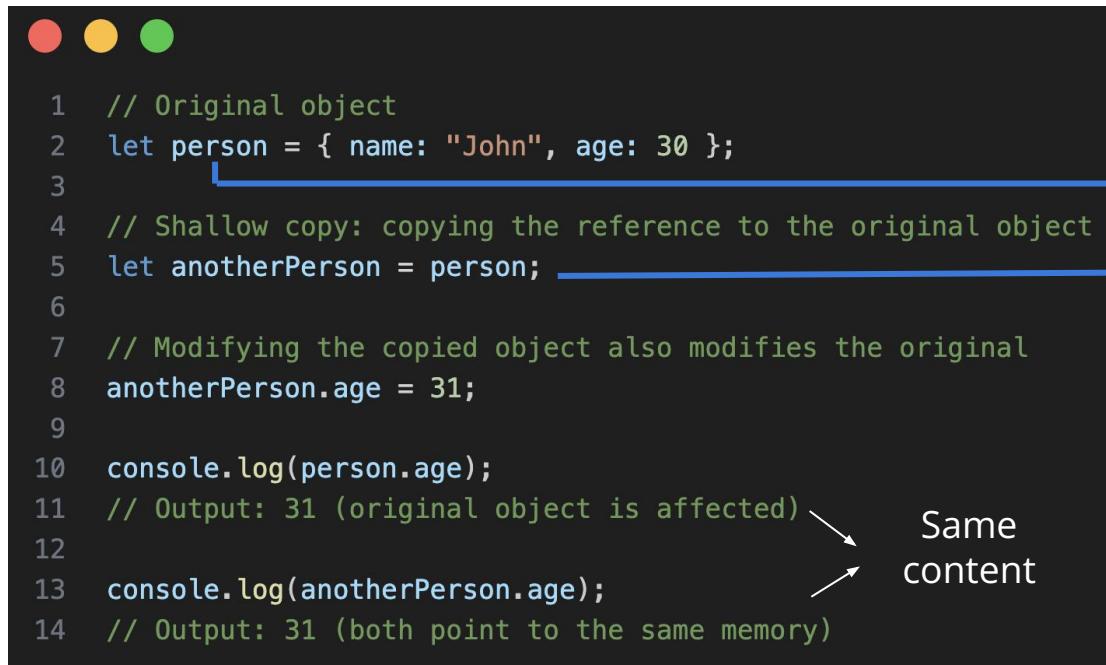
What if we updated the content?

Even if we update our content, still both the reference variables would point to the same memory location and, in turn, the updated content.



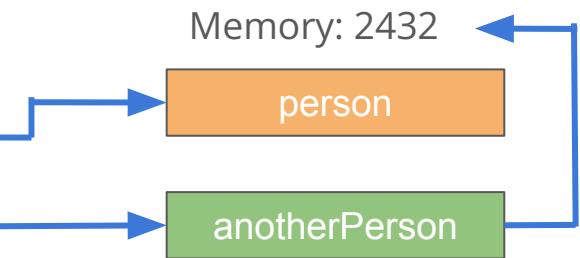
Shallow Copy: Copying references

A shallow copy means copying the memory reference, not the actual content.



The code editor shows the following code:

```
1 // Original object
2 let person = { name: "John", age: 30 };
3
4 // Shallow copy: copying the reference to the original object
5 let anotherPerson = person;
6
7 // Modifying the copied object also modifies the original
8 anotherPerson.age = 31;
9
10 console.log(person.age);
11 // Output: 31 (original object is affected)
12
13 console.log(anotherPerson.age);
14 // Output: 31 (both point to the same memory)
```

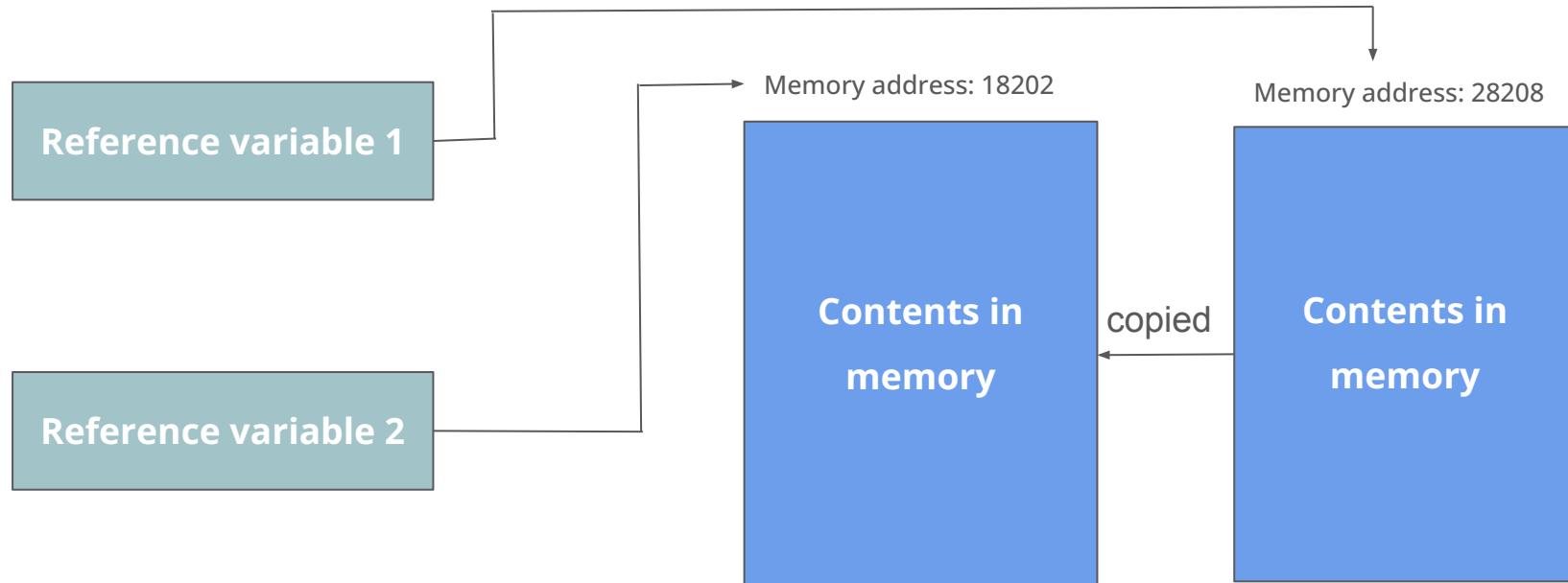


Same content

Both variables point to the same memory. Modifying one affects the other.

Copying the Content itself

What if we made a copy of content in the memory and then stored that new memory location in another reference variable? It is called **Deep Copy**.



Deep Copy: Copying Content

One way of making a deep copy is by using JSON methods, i.e. `JSON.stringify()` and `JSON.parse()`

```
1 let person = {  
2     name: "John",  
3     age: 30,  
4     address: { city: "New York", zip: "10001" }  
5 };  
6  
7 // Deep copy using JSON methods  
8 let deepCopy = JSON.parse(JSON.stringify(person));  
9  
10深Copy.address.city = "Los Angeles";  
11  
12console.log(person.address.city);  
13// Output: "New York" (original is unchanged)  
14  
15console.log(deepCopy.address.city);  
16// Output: "Los Angeles" (deep copy is independent)
```

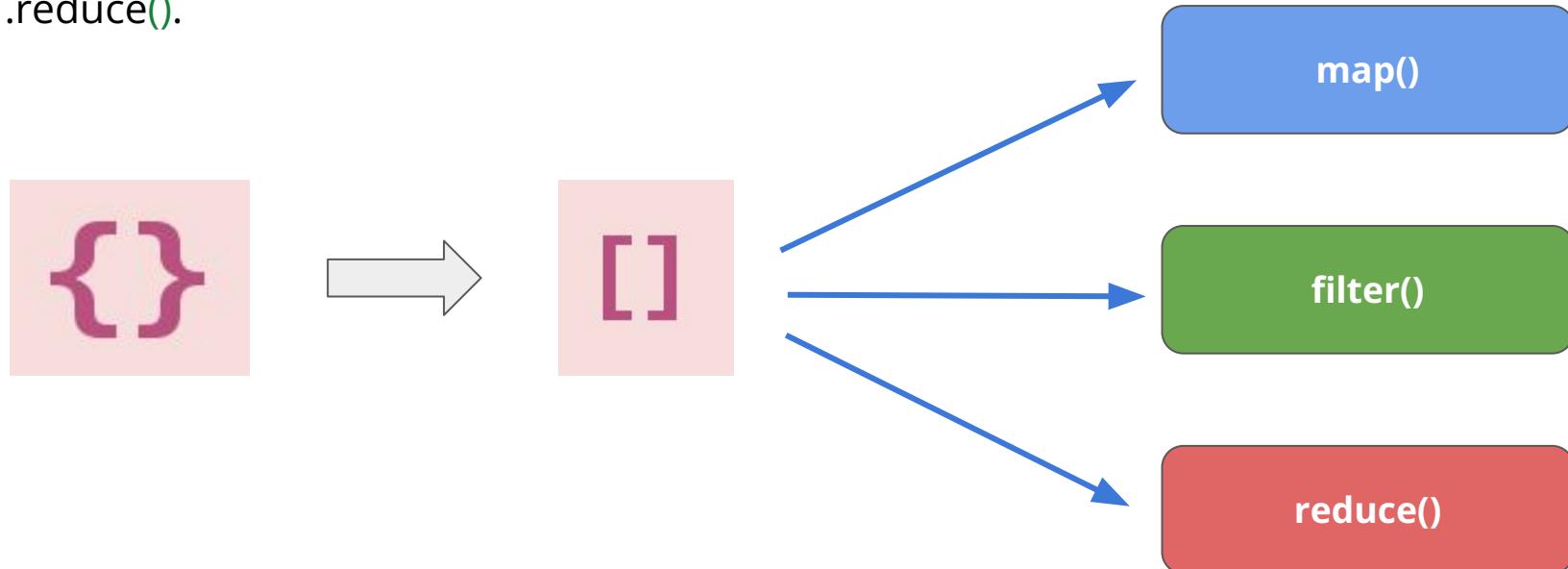
Changing our deep copy wouldn't affect the original copy

Original copy remains Unchanged

Practical Applications

Unlocking the Power of Array Methods

Object utilities like `Object.keys()`, `Object.values()`, etc. transform objects into array-like structures, making it possible to use array methods such as `.map()`, `.filter()`, and `.reduce()`.



Object.keys() with map()

`Object.keys()` gets the keys of an object, and when used with `.map()`, it lets you easily transform those keys into a new array.

```
1 const person = {  
2   firstName: "Harry",  
3   lastName: "Potter",  
4   house: "Gryffindor"  
5 };  
6  
7 // Use Object.keys() with .map() to transform the keys  
8 const keys = Object.keys(person).map((key) => {  
9   return key;  
10});  
11  
12 console.log(keys);  
13 // Output: ['firstName', 'lastName', 'house']
```

Since most of the object utilities return an array, we can iterate those returned values using array methods easily.

Object.values() with map()

Let's iterate values using map:-

```
● ● ●

1 const person = {
2   firstName: "Harry",
3   lastName: "Potter",
4   house: "Gryffindor"
5 };
6
7 // Use Object.values() with .map()
8 const values = Object.values(person).map((value) => {
9   return value;
10 });
11
12 console.log(values);
13 // Output: ['Harry', 'Potter', 'Gryffindor']
```

Object.keys() with map()

`Object.keys()` gets the keys of an object, and when used with `.map()`, it lets you easily transform those keys into a new array.

```
1 const person = {  
2   firstName: "Harry",  
3   lastName: "Potter",  
4   house: "Gryffindor"  
5 };  
6  
7 // Use Object.keys() with .map() to transform the keys  
8 const keys = Object.keys(person).map((key) => {  
9   return key;  
10});  
11  
12 console.log(keys);  
13 // Output: ['firstName', 'lastName', 'house']
```

Since most of the object utilities return an array, we can iterate those returned values using array methods easily.

Filtering Object Entries

We can use `Object.entries()` along with `.filter()` to select specific key-value pairs from an object based on their keys.



```
1 const person = {  
2   firstName: "Harry",  
3   lastName: "Potter",  
4   house: "Gryffindor"  
5 };  
6  
7 // Use Object.entries() and filter to get only firstName and lastName  
8 const filteredEntries = Object.entries(person).filter(([key, value]) =>  
9   key === 'firstName' || key === 'lastName'  
10 );  
11  
12 console.log(filteredEntries);  
13 // Output: [['firstName', 'Harry'], ['lastName', 'Potter']]
```

Iterating over filtered entries using for ... in

We can further iterate over filtered entries using the for...in loop:

```
1 const person = {  
2   firstName: "Harry",  
3   lastName: "Potter",  
4   house: "Gryffindor"  
5 };  
6  
7 // Use Object.entries() and filter to get only firstName and lastName  
8 const filteredEntries = Object.entries(person).filter(([key, value]) =>  
9   key === 'firstName' || key === 'lastName'  
10 );  
11  
12 // Use for...of loop to iterate over the filtered entries  
13 for (const [key, value] of filteredEntries) {  
14   console.log(` ${key}: ${value}`);  
15 }  
16  
17 // Output:  
18 // firstName: Harry  
19 // lastName: Potter
```

We can convert objects into desired array form using Object Utilities, and then we can iterate them just like we iterate arrays.

In Class Questions

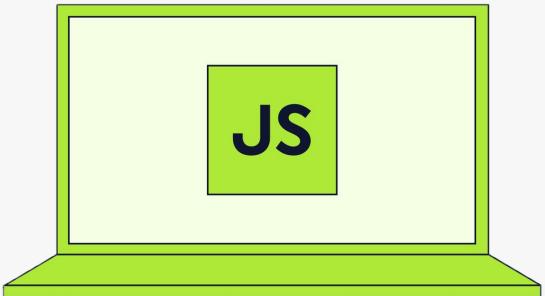
References

1. **MDN Web Docs: JavaScript:** Comprehensive and beginner-friendly documentation for JavaScript.
<https://developer.mozilla.org/en-US/docs/Web/JavaScript>
2. **Eloquent JavaScript:** A free online book covering JavaScript fundamentals and advanced topics.
<https://eloquentjavascript.net/>
3. **JavaScript.info:** A modern guide with interactive tutorials and examples for JavaScript learners.
<https://javascript.info/>
4. **freeCodeCamp JavaScript Tutorials:** Free interactive lessons and coding challenges to learn JavaScript.
<https://www.freecodecamp.org/learn/>

**Thanks
for
watching!**



The Complete Javascript Course



@newtonschool

Lecture 9: Asynchronous Programming

-Bhavesh Bansal



Table of Contents

- Synchronous Vs Asynchronous Programming
- Asynchronous Programming Fundamentals
- Callbacks: a Deep Dive
- setTimeout and setInterval
- setTimeout working
- Real-world applications
- Error Handling

Synchronous vs Asynchronous

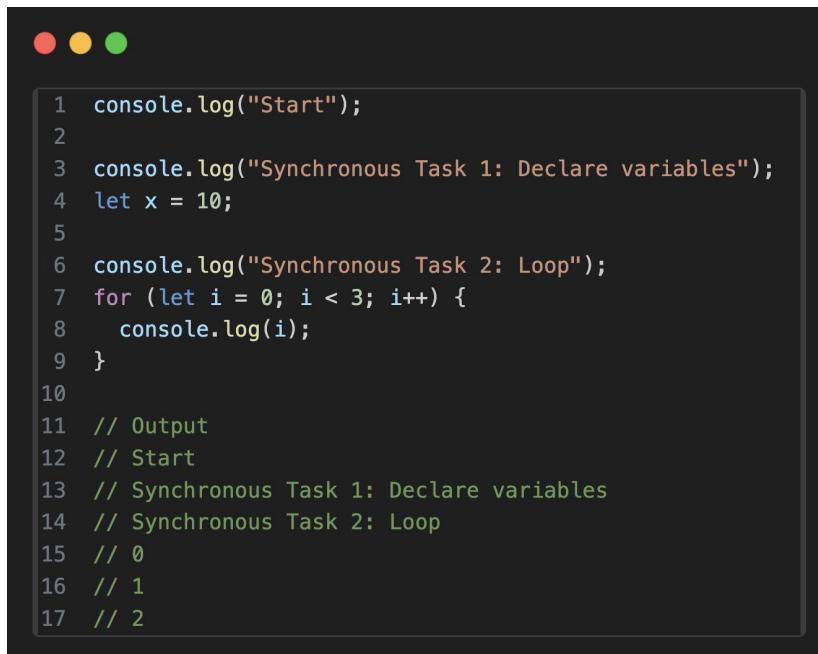
Synchronous Programming

Synchronous programming means doing things one at a time, in the exact order they're written.



Synchronous Programming

So far, we've learned about **functions**, **variable declarations**, **loops**, etc. All these execute one step at a time, in the order written.



```
1 console.log("Start");
2
3 console.log("Synchronous Task 1: Declare variables");
4 let x = 10;
5
6 console.log("Synchronous Task 2: Loop");
7 for (let i = 0; i < 3; i++) {
8   console.log(i);
9 }
10
11 // Output
12 // Start
13 // Synchronous Task 1: Declare variables
14 // Synchronous Task 2: Loop
15 // 0
16 // 1
17 // 2
```

Can you see the output?
Each line executes **only after** the previous line finishes.

But do we always do things in order??

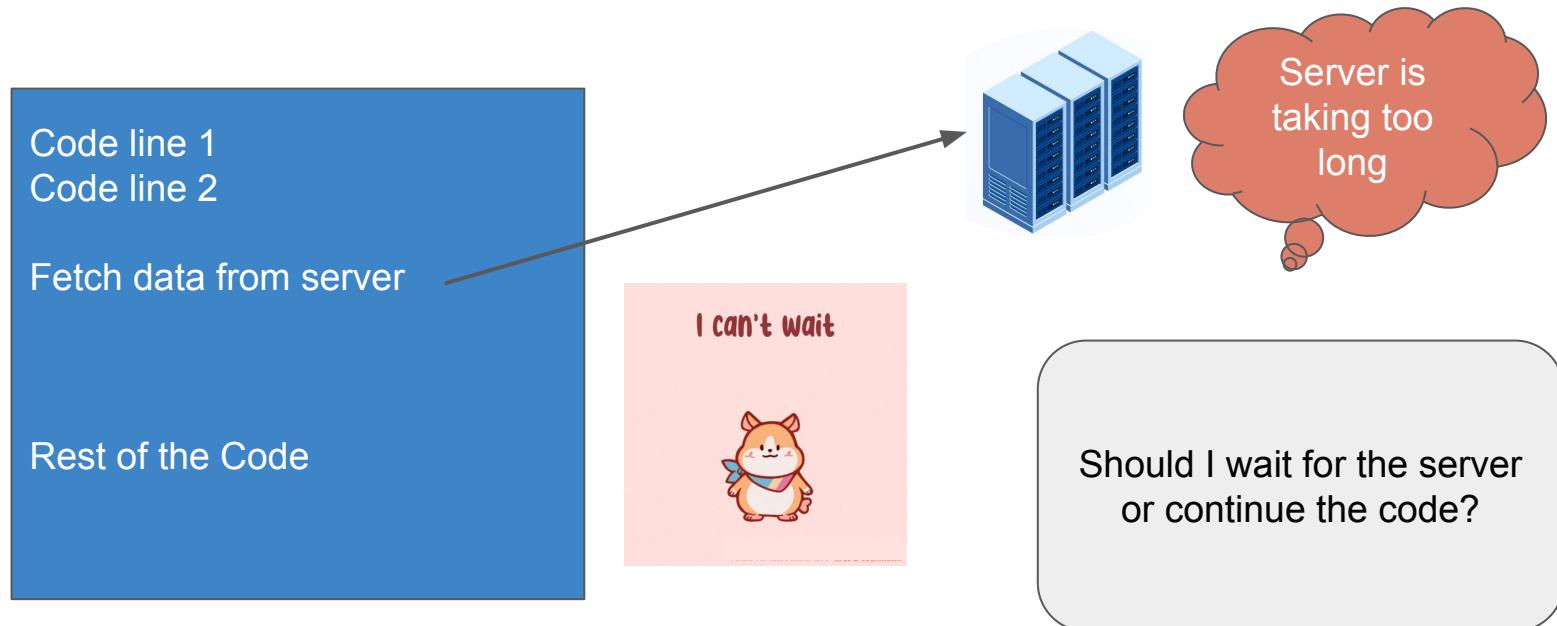
Sometimes we do things one after another, but other times we start something, stop, and switch to something else, i.e., we complete tasks asynchronously.



You might start cooking, but then remember to check a message from work. You stop cooking and check the message first. Did you follow the order?

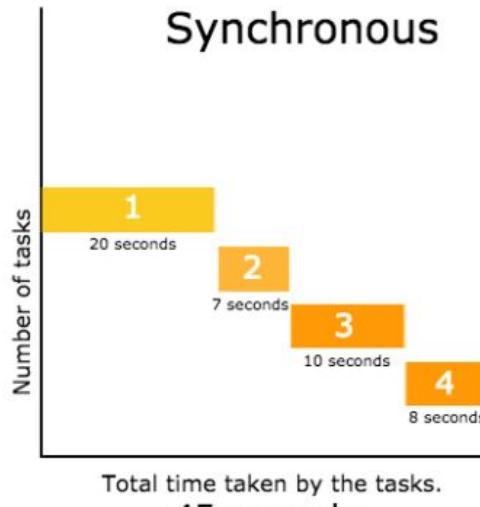
Need for asynchronous programming

Just like in daily life, where tasks aren't always in order, programming uses **asynchronous** tasks to handle unpredictable delays without stopping everything.



Better to Wait and Keep Things Running

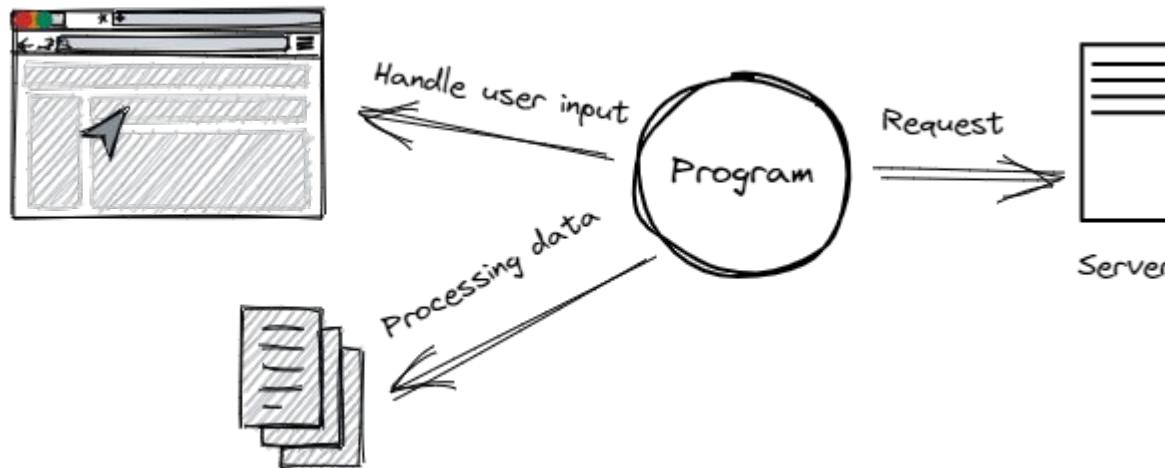
Sometimes it's better to wait for the server response, but instead of pausing everything, we can continue other tasks while waiting.



The idea is to execute shorter tasks first and catch up on longer tasks once they finish, reducing overall execution time.

What is Async Programming?

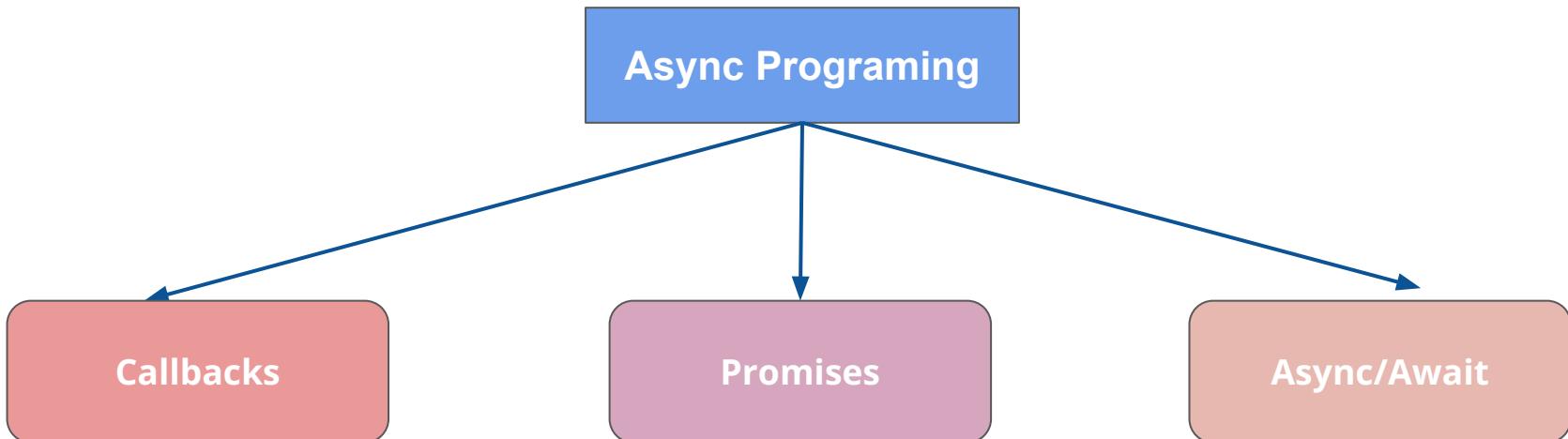
Asynchronous programming lets tasks run independently, allowing your program to continue other tasks while waiting for long operations to finish.



Here, the program sends a request to the server and, while waiting for the response, continues accepting and processing user inputs.

Ways to implement async programming

There are three main ways to implement asynchronous programming in JavaScript:



Callbacks: a Deep Dive

Callbacks: Task to Execute Later

In simple terms, a callback is just a function that you give to another function. The receiving function will then decide when and how to execute.

```
1  function task1(callback) {  
2      console.log("Task 1 completed");  
3      callback();  
4      // Execute the callback after Task 1  
5  }  
6  
7  function task2() {  
8      console.log("Task 2 completed");  
9  }  
10  
11 task1(task2);  
12 // Passing task2 as a callback to task1
```

Here it is up to task1() function when and where it executes callback function. Here it is executing it immediately, but that is not always the case.

Executing callbacks after a while

Let's add a delay of 2000 milliseconds before the callback executes using setTimeout().

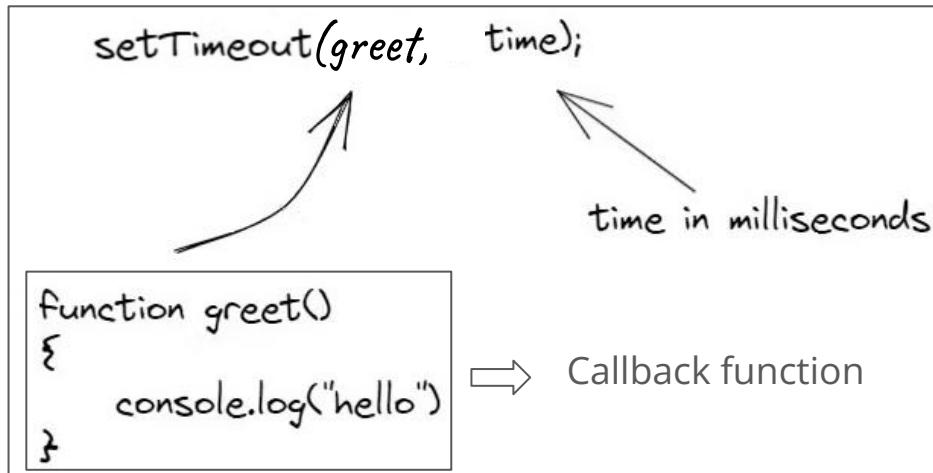
```
1 function task1(callback) {  
2     setTimeout(() => {  
3         console.log("Task 1 completed");  
4         callback();  
5         // Execute the callback once Task 1 is done  
6     }, 2000); // Simulate a delay  
7 }  
8  
9 function task2() {  
10    console.log("Task 2 completed");  
11 }  
12  
13 task1(task2);  
14 // Passing task2 as a callback to task1
```

Don't panic, we will
learn how
setTimeout() works in
the next slides.

Callbacks in action: setTimeout

`setTimeout` is a JavaScript function that executes a callback function after a specified delay (in milliseconds).

Syntax:



setTimeout: example 1

We can print a specific message after a certain period of time using setTimeout.



```
1 function greet() {  
2     console.log("Hello");  
3 }  
4  
5 // greet function runs after 2 seconds  
6 // i.e. (2000 milliseconds)  
7 setTimeout(greet, 2000);
```

Here we get output after 2000 milliseconds, i.e. 2 secs

Output:

Hello

setTimeout: example 1

The same example can be rewritten like this using named functions and arrow functions.



```
1 setTimeout(function greet(){  
2     console.log("Hello");  
3 }, 2000);
```

Using named Function



```
1 setTimeout(()=> {  
2     console.log("Hello");  
3 }, 2000);
```

Using arrow Function

setTimeout: example 2

Here we have implemented a countdown using setTimeout.

```
1  function startCountdown(seconds) {  
2      function countdown() {  
3          if (seconds > 0) {  
4              console.log(seconds);  
5              // Log the current second  
6              seconds--;  
7              // Decrease the time  
8              setTimeout(countdown, 1000);  
9              // Call countdown again after 1 second  
10         } else {  
11             console.log("Time's up!");  
12         }  
13     }  
14     countdown();  
15 }  
16  
17 // Start a countdown from 5 seconds  
18 startCountdown(5);
```

Output:

5
4
3
2
1
Time's up!

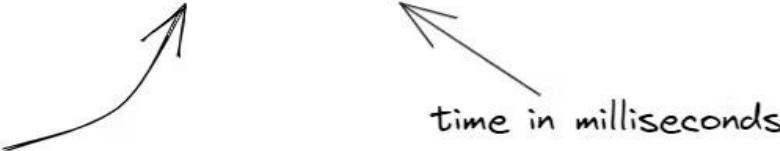
Callbacks in action: setInterval

`setInterval` is a JavaScript function that repeatedly executes a specified callback function at fixed time intervals (in milliseconds).

Syntax:

```
setInterval(greet,    time);  
  
function greet()  
{  
  console.log("hello")  
}
```

time in milliseconds



setInterval: example 1

Here we are printing and updating the message count every 1000 milliseconds. The `setInterval()` function returns a unique identifier, known as an interval ID, which is stored to later clear the interval

```
1 let count = 0;
2
3 const intervalId = setInterval(() => {
4     console.log(`Message ${++count}`);
5 }, 1000);
```

Output:

Message 1

Message 2

Message 3

..

..

setInterval: example 2

Here we are displaying the most updated time using setInterval. The latest time gets updated every 1000 milliseconds, i.e. every second

```
1 function displayTime() {  
2     const now = new Date();  
3     const time = now.toLocaleTimeString();  
4     // Get the current time as a string  
5  
6     console.log(time);  
7     // Display the current time  
8 }  
9  
10 // Update the time every second  
11 setInterval(displayTime, 1000);
```

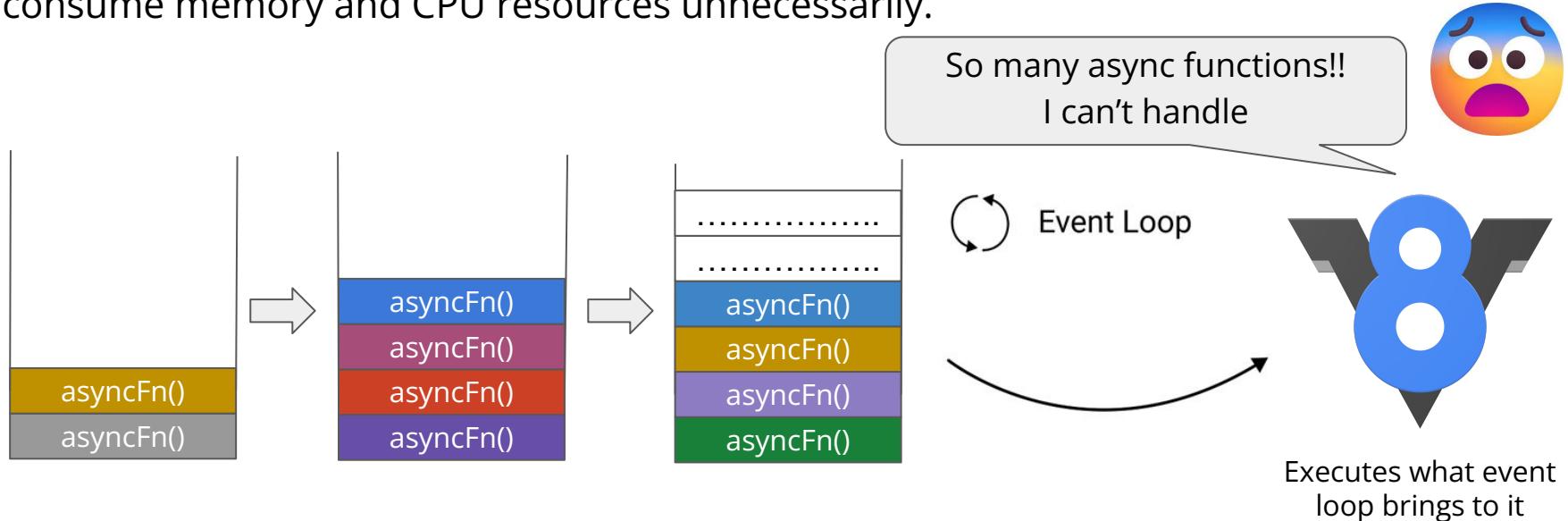
Output:

```
12:34:56 PM  
12:34:57 PM  
12:34:58 PM  
12:34:59 PM  
...
```

In JavaScript, the [Date](#) object represents a specific moment in time. Using the [Date\(\)](#) constructor without arguments creates a new [Date](#) object set to the current date and time.

Zombie Tasks: Task Queue Congestion

When you have too many async functions running (or scheduled but never cleared), they can be described as 'zombie callbacks' or 'zombie tasks.' These lingering tasks consume memory and CPU resources unnecessarily.



Clearing up Zombie Tasks: setTimeout

So it is always a good practice to free up such asynchronous operations that are taking up system resources

```
1 function startCountdown(seconds) {  
2     let countdownTimeout;  
3  
4     function countdown() {  
5         if (seconds > 0) {  
6             console.log(seconds); // Log the current second  
7             seconds--; // Decrease the time  
8             countdownTimeout = setTimeout(countdown, 1000);  
9         } else {  
10             console.log("Time's up!");  
11         }  
12     }  
13  
14     countdown(); // Start the countdown from the given seconds  
15  
16     // Example: Stop the countdown after 3 seconds  
17     setTimeout(() => {  
18         clearTimeout(countdownTimeout); // Stop the countdown  
19         console.log("Countdown stopped!");  
20     }, 3000);  
21 }
```



Clearing up Congestion: setInterval

You should always conditionally clear the interval after use to avoid needless consumption of resources. Otherwise, it would get repeated infinitely. Here is how you can do it:



```
1 // Clearing the interval from memory
2 setTimeout(()=>{
3     clearInterval(intervalId);
4     console.log("Interval cleared");
5 }, 10000);
```



This argument sets clear time to 10000 milliseconds, i.e. 10 secs

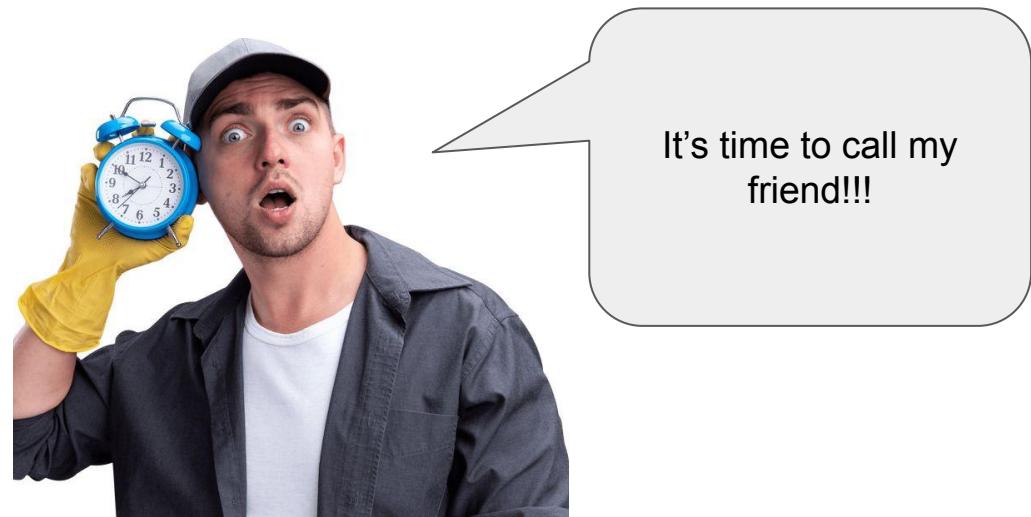
Output:

Interval cleared

Some Real World Examples

Quick Reminder System

Imagine you're working on a task and suddenly remember you need to call a friend in 5 minutes. A quick and simple solution can help you set a one-time reminder to pop up after a specific delay.



Build a timer using setTimeout

A quick solution lets you set a one-time reminder to alert you after a delay.



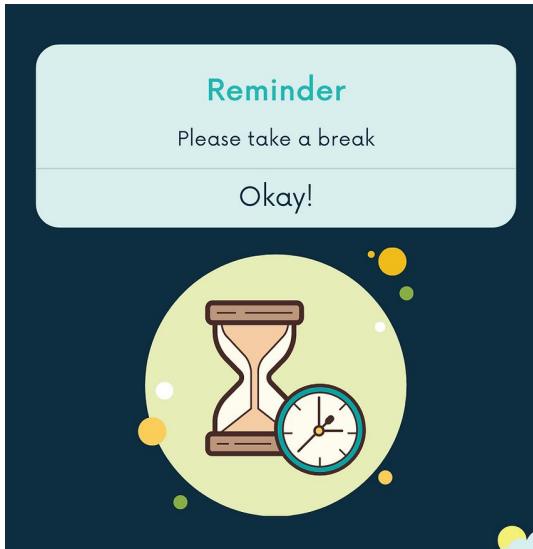
```
1 function setReminder(message, delay) {  
2     console.log(`"${message}" in ${delay / 1000} seconds.`);  
3     setTimeout(() => console.log(`🕒 Reminder: ${message}`),  
4         delay);  
5 }  
6 // Set a reminder for 5 minutes (300000 ms)  
7 setReminder("Call your friend!", 300000);
```

Output:

Call your friend in 300 seconds

Simple Productivity Timer

You're working on a project and want to stay focused by tracking your time. To stay productive, set a timer to remind you every 30 minutes to take a break.



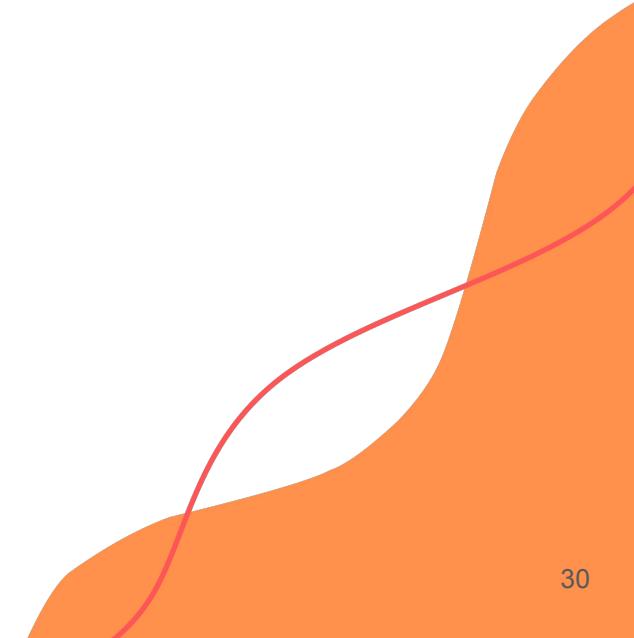
Build a reminder using setInterval

An easy solution is to build a recurring reminder using setInterval

```
1 function startBreakReminder(interval) {  
2     console.log("Timer started! ");  
3     setInterval(() => {  
4         console.log("⏰ Time for a break!")  
5     }, interval);  
6 }  
7  
8 // Reminder every 30 minutes (1800000 ms)  
9 startBreakReminder(1800000);
```

Output:

Time for a break!	←	30 mins
Time for a break!	←	60 mins
Time for a break!	←	90 mins



setTimeout

Workshop

Workshop: Question 1

You are building a web app that reminds users to save their work every 10 minutes. Write a function using `setTimeout` that logs "Please save your work!" after a 10-minute delay.

What should the function look like?

Reminder

Save your work; otherwise, progress would be lost.

Can you write a code to implement this reminder??

Workshop: Question 2

In your app, you want to show a message to the user saying "Action timed out!" if they don't click a button within 5 seconds. Use `setTimeout` to implement this timeout mechanism.

How will you implement this feature using `setTimeout`?

Alert!

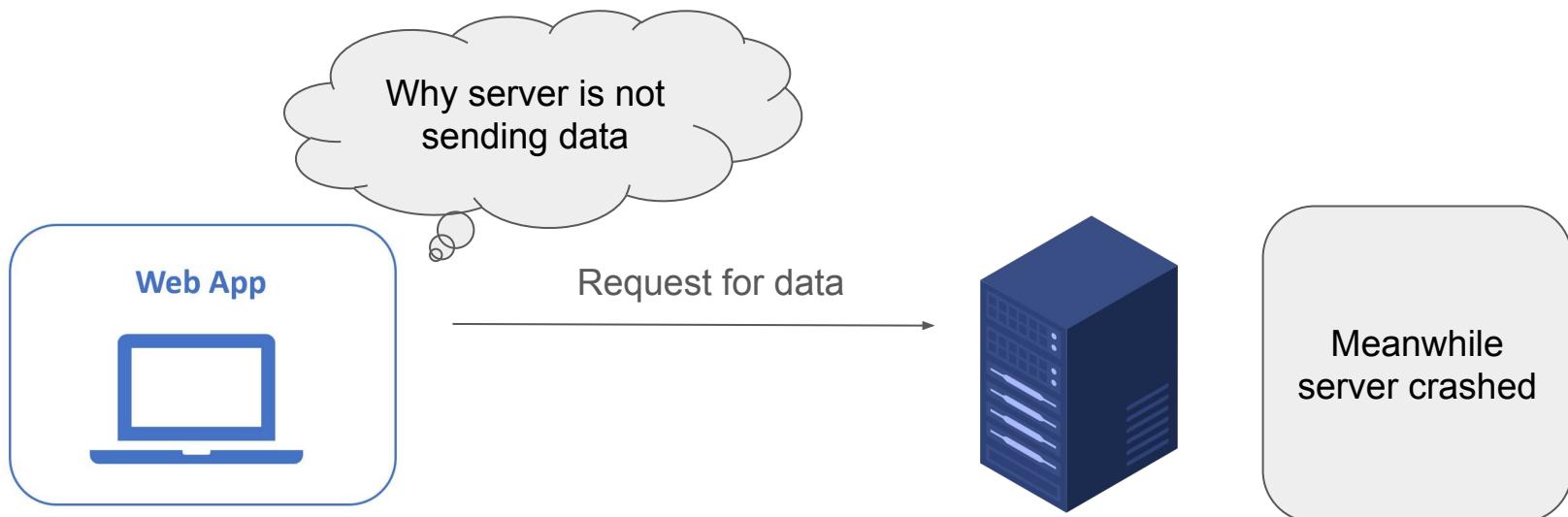
Action timed out!

I'm sure you'll be able to handle
this one!

Error Handling

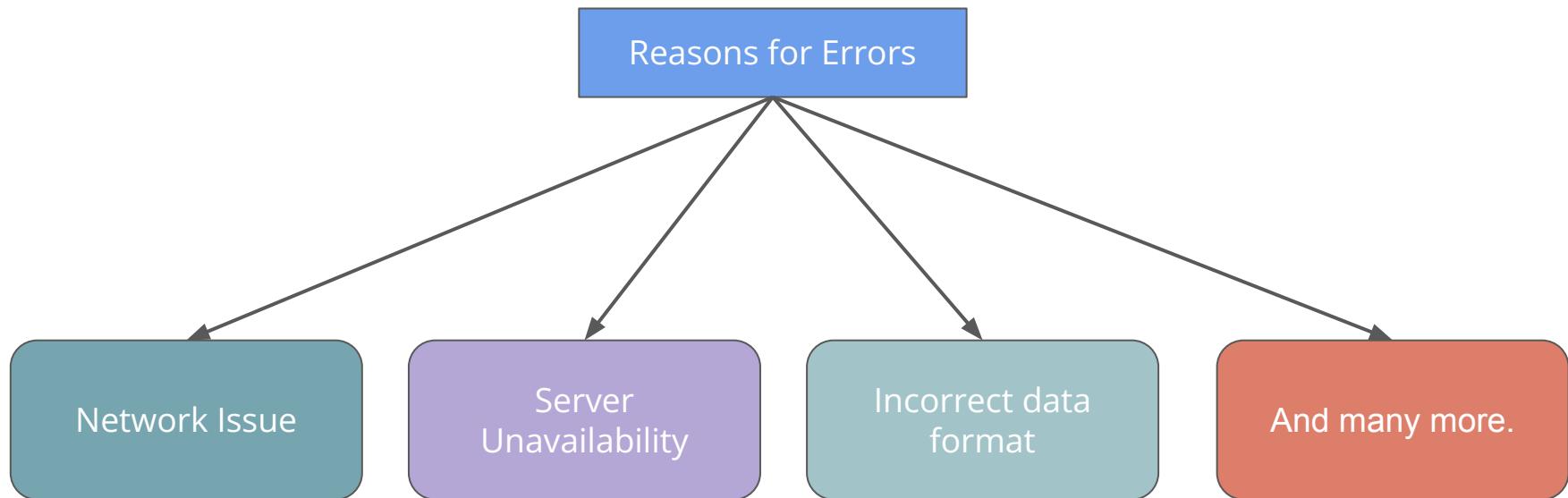
Need for error handling

Imagine building a web app that fetches user data from an API. Network issues, server downtime, or request failures can lead to errors, and if not handled, the app might crash or behave unexpectedly.



Common Causes of Errors

There are several reasons due to which errors can occur:



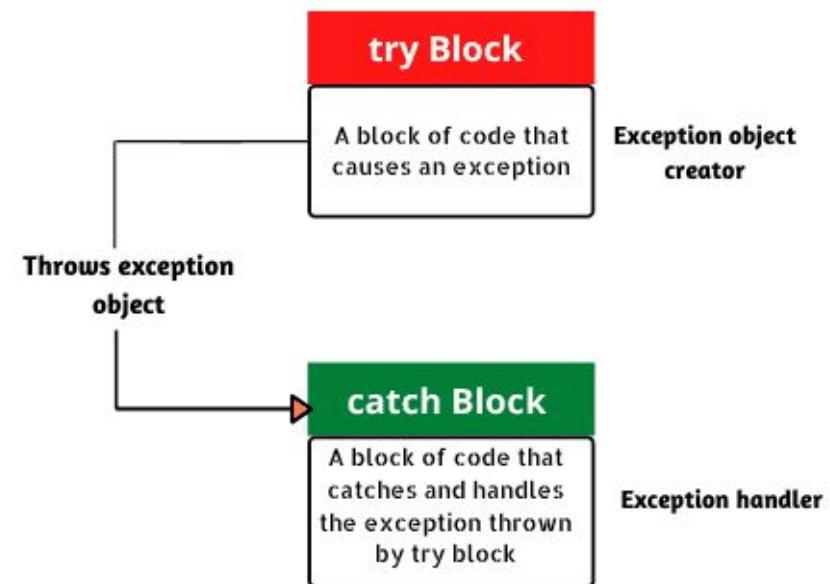
Handling Errors using try/catch block

JavaScript provides the `try...catch` statement to handle errors gracefully.

Syntax:



```
1 try {  
2     // Code that might throw an error  
3 } catch (error) {  
4     // Code to handle the error  
5 }
```



Dummy Example: try/catch

try/catch for fetching data, but for simplicity we have replaced fetch with Math.random()

```
1  async function fetchData() {
2      try {
3          // Simulate an error by using a random number
4          let response = Math.random() > 0.5;
5
6          if (!response) {
7              throw new Error("Failed to fetch data!");
8          }
9
10         console.log("Data fetched successfully!");
11     } catch (error) {
12         console.log("Error:", error.message);
13     }
14 }
```

Output:

// if response === false

Error: Failed to fetch data!

We used *async* keyword to make function asynchronous.

Real Example: try/catch

Let's use proper fetch function to fetch the data from the server

```
1  async function fetchData() {  
2      try {  
3          let response = await fetch("https://api.example.com/data");  
4  
5          if (!response.ok) {  
6              throw new Error(`HTTP error! Status: ${response.status}`);  
7          }  
8  
9          let data = await response.json();  
10         console.log(data);  
11     } catch (error) {  
12         console.error("Error occurred:", error.message);  
13         alert("Failed to fetch data. Please try again.");  
14     }  
15 }  
16 }
```

Don't stress about fetch()
api, we will learn it later

Output:

// if response === false

Error: Failed to fetch data!

In Class Questions

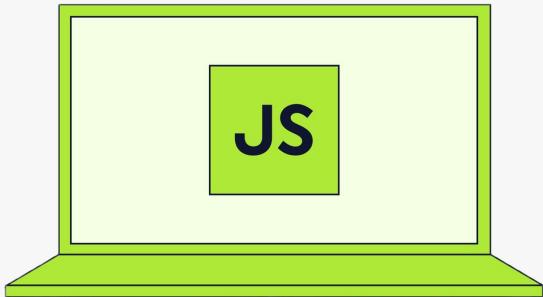
References

1. **MDN Web Docs: JavaScript:** Comprehensive and beginner-friendly documentation for JavaScript.
<https://developer.mozilla.org/en-US/docs/Web/JavaScript>
2. **Eloquent JavaScript:** A free online book covering JavaScript fundamentals and advanced topics.
<https://eloquentjavascript.net/>
3. **JavaScript.info:** A modern guide with interactive tutorials and examples for JavaScript learners.
<https://javascript.info/>
4. **freeCodeCamp JavaScript Tutorials:** Free interactive lessons and coding challenges to learn JavaScript.
<https://www.freecodecamp.org/learn/>

**Thanks
for
watching!**



The Complete Javascript Course



@newtonschool

Lecture 10: Introduction to Promises

-Bhavesh Bansal



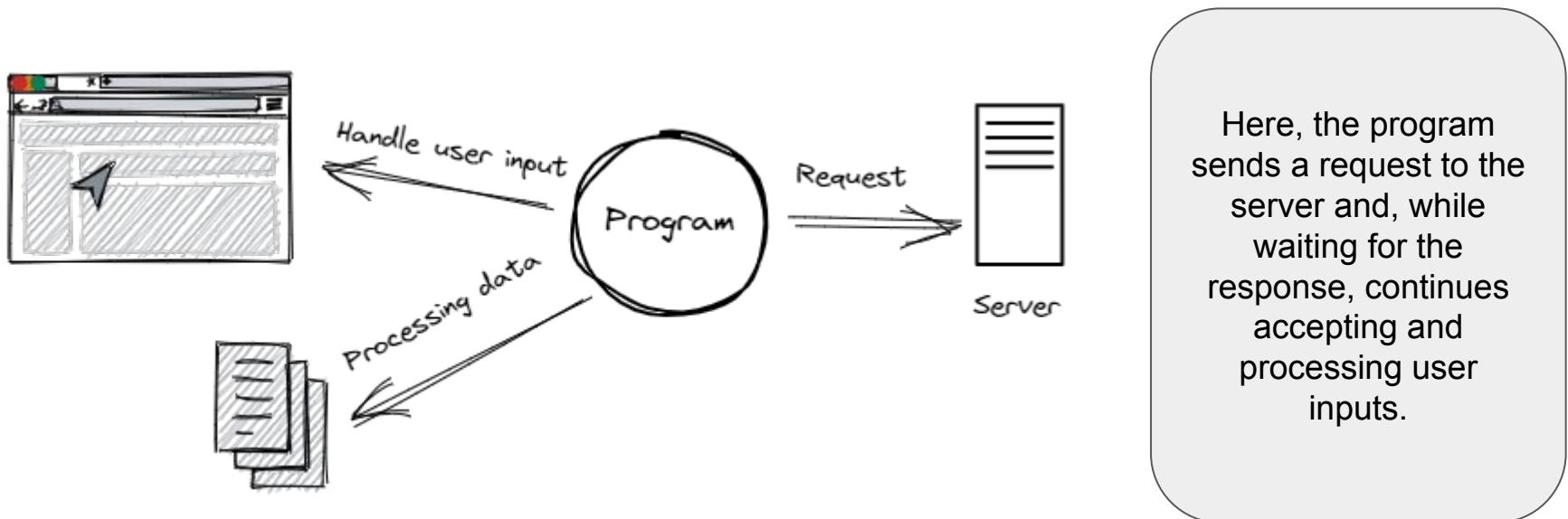
Table of Contents

- Callbacks vs Promise: Need for promise
- Understanding Promises
 - Promises in Javascript
 - Basic Promise Usage
- Creating Promises
 - Promise Constructor
 - Promise Methods: then(), catch() and finally()
- Handling Promise States
 - resolve
 - reject
 - pending
- Error Handling

Why do we need Promises?

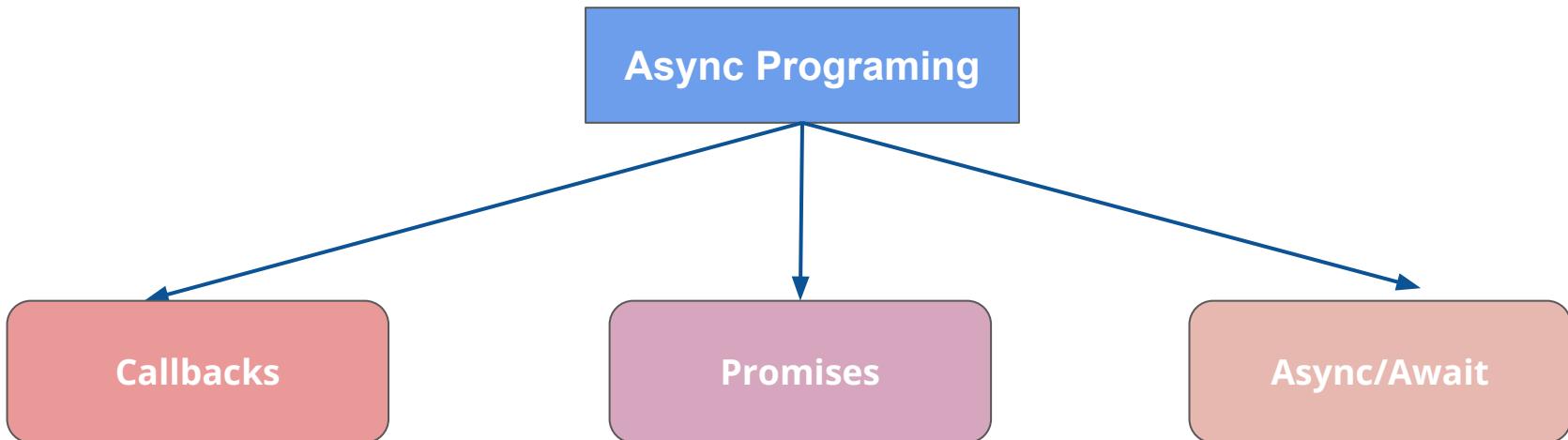
Pre-discussed: Async Programming

Asynchronous programming lets tasks run independently, allowing your program to continue other tasks while waiting for long operations to finish.



Pre-discussed: Ways to implement

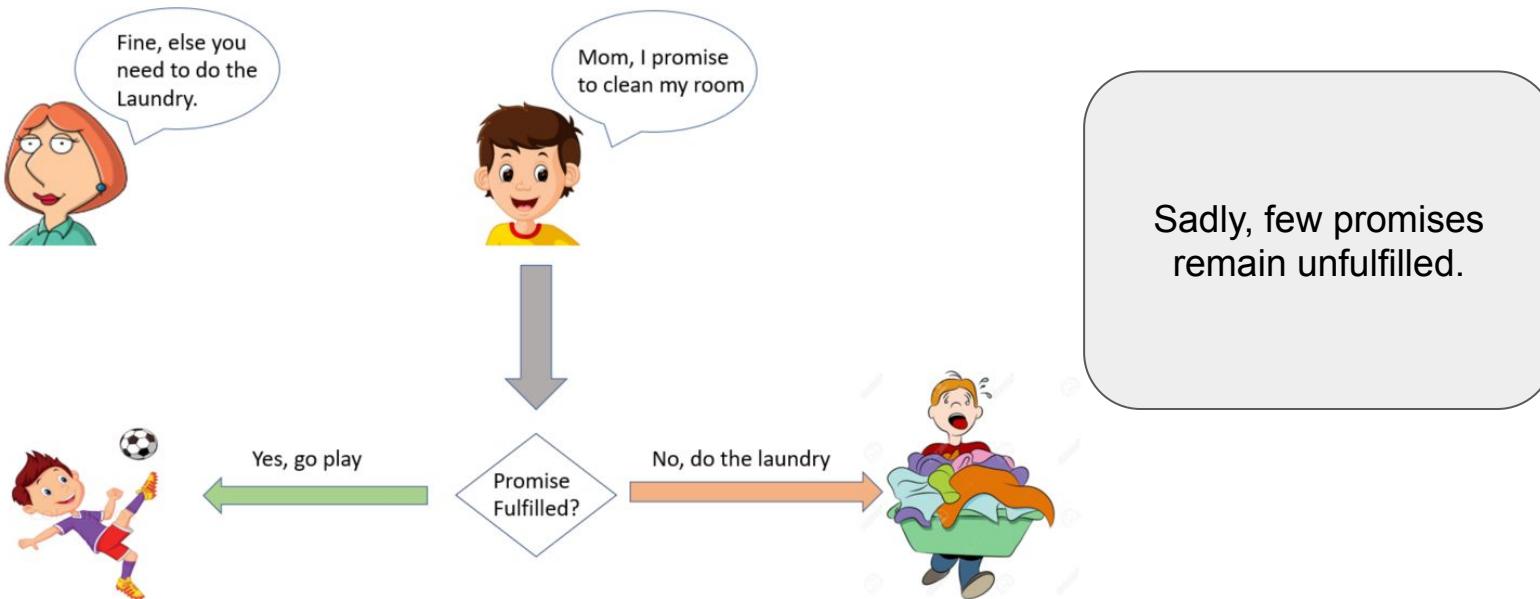
There are three main ways to implement asynchronous programming in JavaScript:



Understanding the Promises

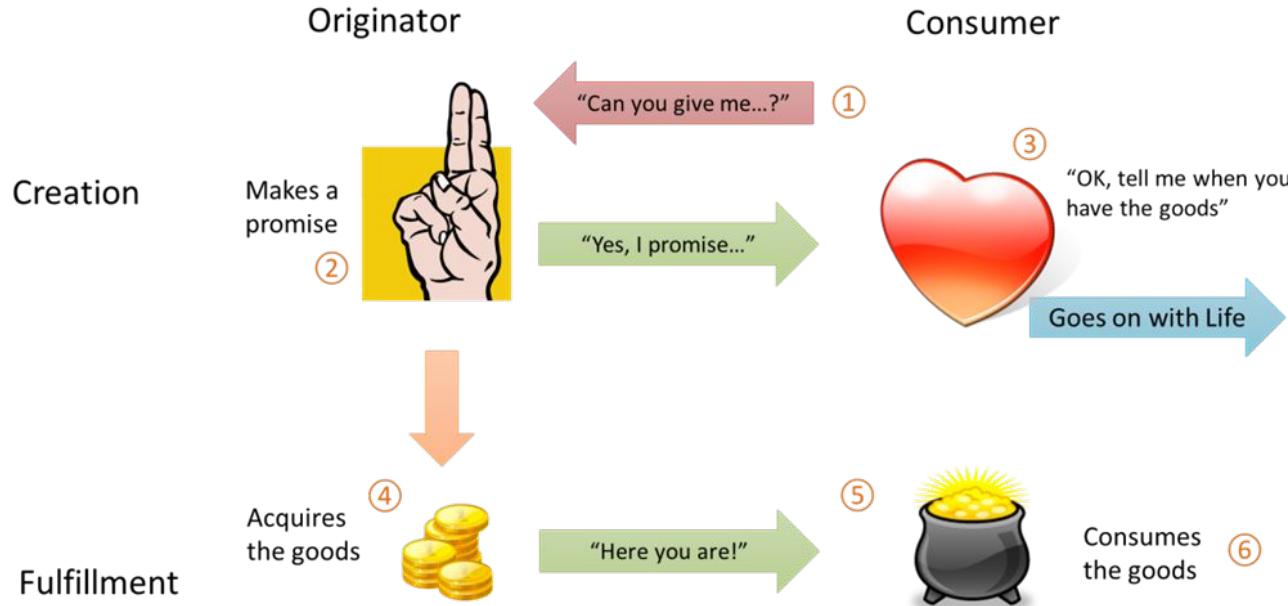
Have you ever made a promise?

If you did, then you must know how valuable promises are. A promise in real life is a commitment or guarantee made by someone.



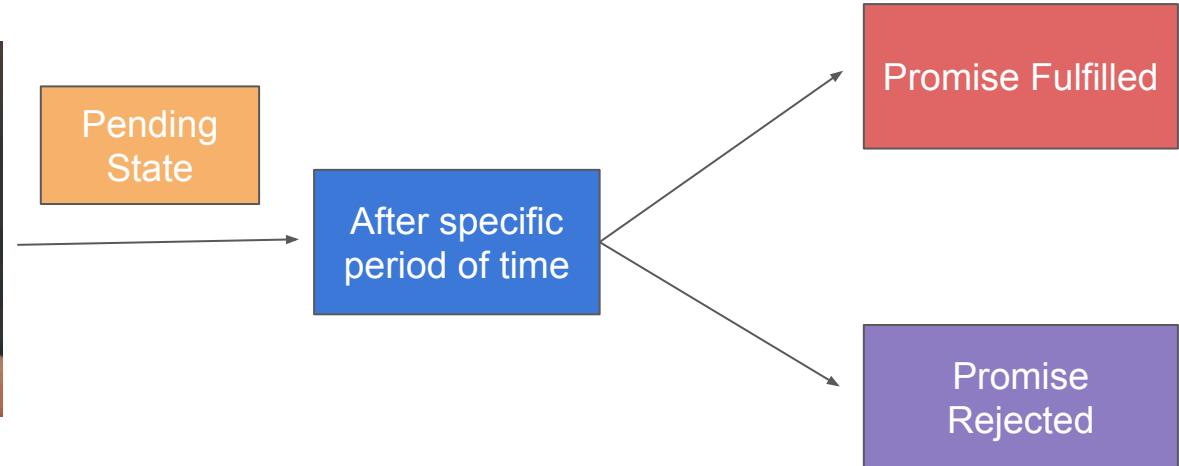
Stages of promise

If you did, then you must know how valuable promises are. A promise in real life is a commitment or guarantee made by someone.



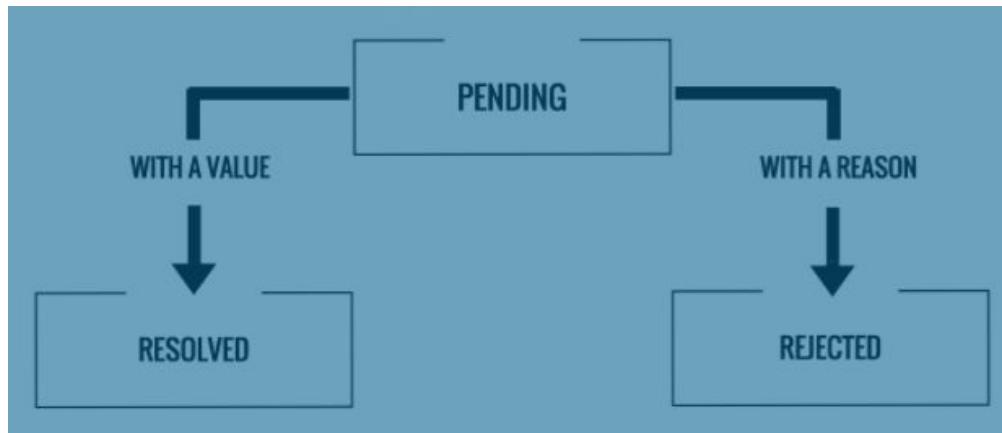
Promises in Javascript

A promise is a commitment in programming—a guarantee that something will happen in the future but not an immediate action. It's a placeholder for future work.



Different promise states

Promises in Javascript exist in three states: pending, fulfilled, and rejected.

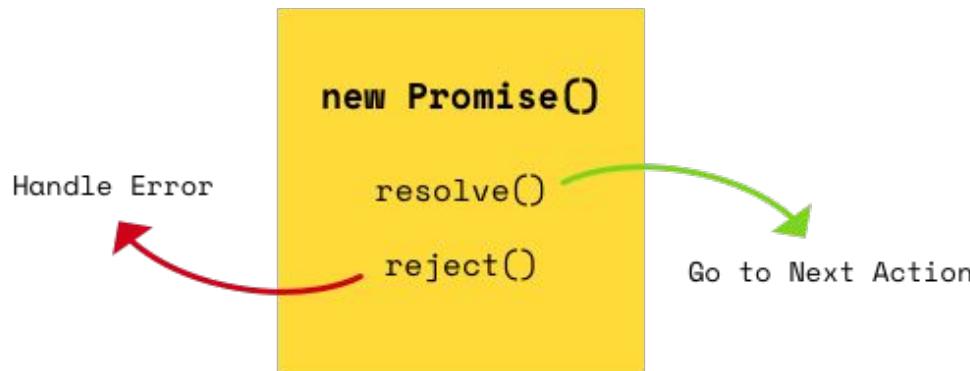


Initially, a promise remains in a pending state, and after some time it either gets resolved/fulfilled or rejected.

Creating Promises

Promise: constructing a promise

To create a promise in JavaScript, you use the `Promise` constructor. The promise constructor receives two functions as arguments: `resolve()` and `reject()`



The `resolve` function is called when the promise is successfully fulfilled.

The `reject` function is called when the promise is rejected.

Promise: Using Promise Constructor

To create a promise in JavaScript, you use the `Promise` constructor. The promise constructor receives two functions as arguments: `resolve()` and `reject()`



```
1 // Constructor to build a promise
2 const myPromise = new Promise([resolve, reject) => {
3     // Body of Promise
4
5     // Here we can either resolve
6     // Or reject promise conditionally
7
8});
```

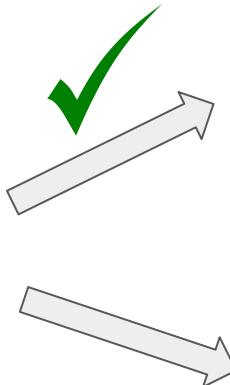
Received resolve and
reject functions as
arguments

Promise: resolve()

The `resolve` method in JavaScript is a static method of the `Promise` object. It is used to create a promise that is resolved with a given value.



Promised to complete the task in meeting



Resolved the task and provided the work data as return value

`resolve()` function's work is to finish the task and provide a return value.



Fail/rejected and return fail/error message

`reject()` function's work is to return an appropriate error message

Promise: resolve()

Let's have a look at a dummy example:



```
1 // Constructor to build a promise
2 const myPromise = new Promise((resolve, reject) => {
3     // Simulation of success in resolving promise
4     const success = true;
5
6     if(success === true){
7         resolve("Promise got resolved");
8     }
9 });

```

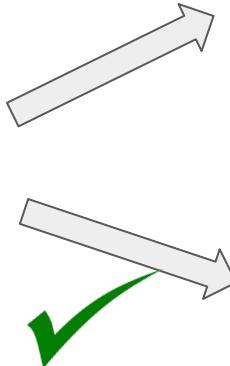
On successful resolution of promise, resolve() returns a value. For simplicity, we are returning a success message.

Promise: reject()

It might be possible that a promise can't be delivered and you need to provide an explanation and, in JavaScript, an error message.



Promised to complete the task
in meeting



Resolved the task and provide
the work data as return value

resolve() function
work is to finish the
task and provide a
return value.



Fail/rejected
and return
fail/error
message

reject() function work
is to return an
appropriate error
message

Promise: reject()

Let's have a look at a dummy example:



```
1 const myPromise = new Promise((resolve, reject) => {
2     // Simulation of success in resolving promise
3     const success = false;
4
5     if(success === true){
6         resolve("Promise got resolved");
7     } else {
8         reject("Promise got rejected");
9     }
10});
```

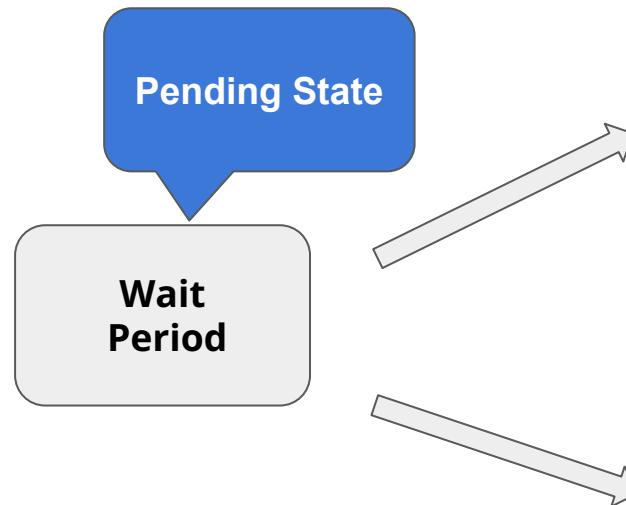
On failure to resolve a promise reject() returns an error message. For simplicity we are returning a failure message..

Promise: pending()

Before either promise gets resolved or rejected, there is some wait period. This wait period is called pending state of promise



Promised to complete the task
in meeting



Resolved the task and provided
the work data as return value



Fail/rejected
and return
fail/error
message

Promise Creation: Example 1

Let's understand promises with a simple example:

```
1 // Constructor to build a promise
2 const myPromise = new Promise((resolve, reject) => {
3     // Asynchronous Operation
4     const success = true; // Simulating a condition
5
6     if(success === true){
7         resolve("Promise got resolved!");
8     } else{
9         reject("Promise got rejected!");
10    }
11});
```



Using `resolve()` to
fulfill the promise



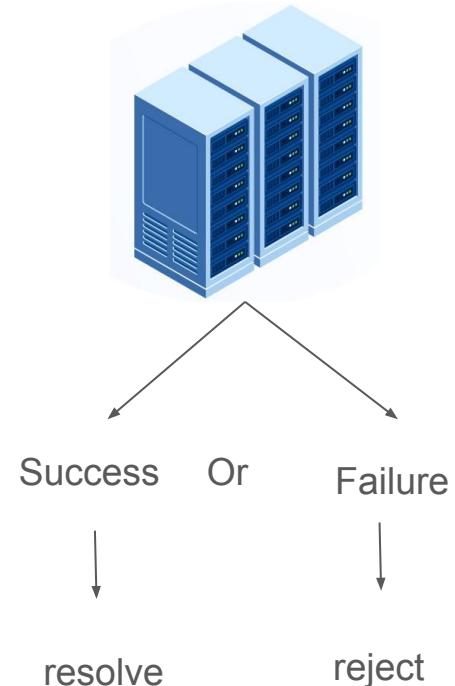
Using `reject()` to
reject the promise

Promise Creation: Example 2

Let's take a more practical example:



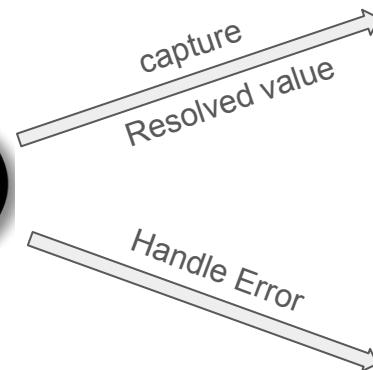
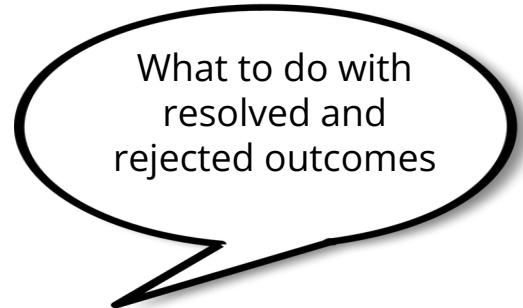
```
1 // Simulating a function that fetches data from a server
2 function fetchDataFromServer() {
3     return new Promise((resolve, reject) => {
4         const serverStatus = false;
5         // Simulate server failure (false means failed)
6
7         if (serverStatus) {
8             resolve('Data fetched successfully');
9         } else {
10             reject('Error: Server is down');
11         }
12     });
13 }
```



Handling the Promise

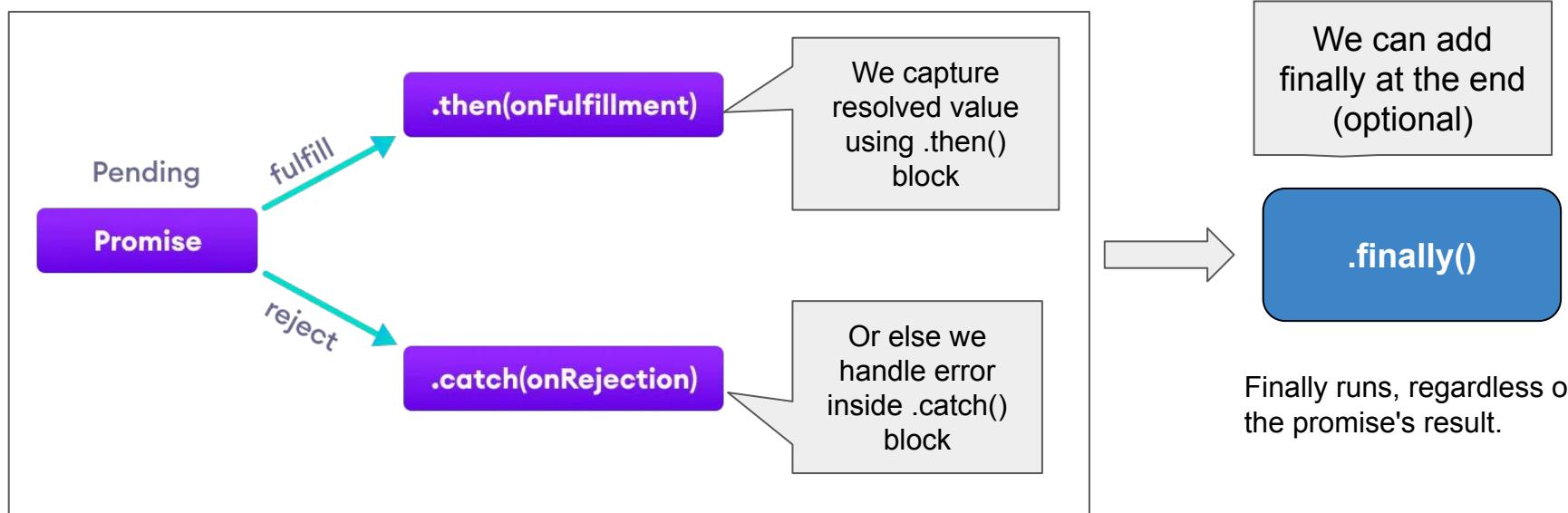
Promises: Handling Promise Outcomes

As we have learned, a promise is initially in a pending state, and then after a while it either gets resolved or rejected.



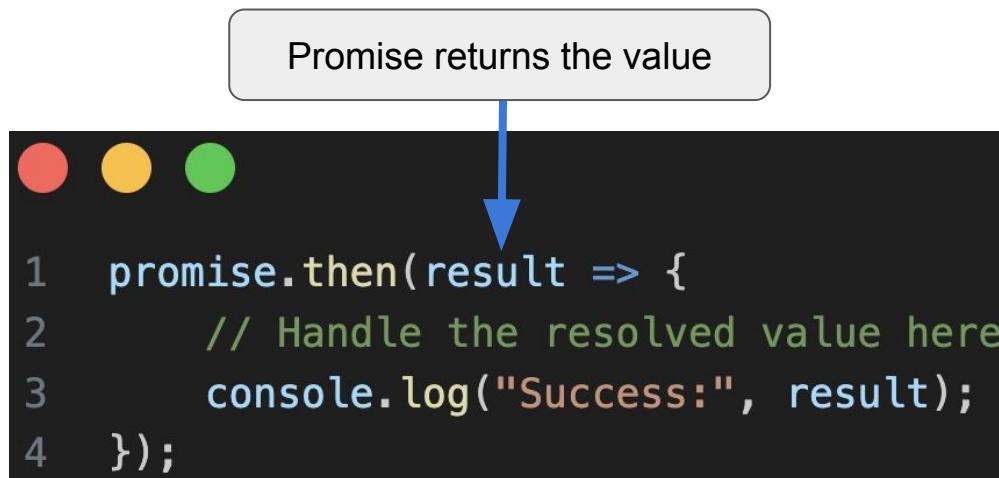
Promises: Bridging Hope and Reality

We hope a promise resolves/fulfilled and gives us a value to work with in the “`then`” block. However, if it gets rejected, we handle it in the “`catch`” block.



Promise: Using the .then() block

The `.then()` block handles data after resolution in a promise chain by capturing it and allowing further processing.



Promise returns the value

```
1 promise.then(result => {
2     // Handle the resolved value here
3     console.log("Success:", result);
4});
```

We can use the returned value from Promise inside `.then()` block

.then(): Example

The `.then()` block is used to handle the successful resolution of a promise, capturing the returned value and allowing further processing or actions.

```
1 // Simulating a function that fetches data
2 function fetchDataFromServer() {
3     return new Promise((resolve, reject) => {
4         const serverStatus = true;
5         // Simulate server failure (false means failed)
6
7         if (serverStatus) {
8             resolve('Data fetched successfully');
9         } else {
10            reject('Error: Server is down');
11        }
12    });
13 }
```



```
1 // Using .then()
2 fetchDataFromServer()
3     .then(result => {
4         console.log("Success:", result);
5         // Handling the resolved value
6     })

```

Here we are using `.then()` block to capture the resolved value

Promise: Using .catch() block

The `.catch()` block handles errors or rejections in a promise chain, ensuring that any issues during the operation are caught and managed gracefully.

```
1 promise.then(()=>{
2     // then block code
3 }
4     .catch((error)=>{
5         // catch block code
6         console.log("Error: ", error);
7 });


```

`.catch()` is indeed added after the last `.then()` block in most cases to handle any errors that occur while consuming the promise.

.catch(): Example

It might be possible that error occurred while consuming the promise which can be caught using the .catch() block.

```
1 // Using .then()  
2 fetchDataFromServer()  
3   .then(result => {  
4     console.log("Success:", result);  
5     // Handling the resolved value  
6   })  
7   .catch((error) => {  
8     console.log("Error: ", error);  
9   });
```

Here we are capturing
error while consuming
the promise.

Promise: Using .finally() block

It is possible to chain a block that runs regardless of whether the promise is resolved or rejected using .finally() block.

```
1 promise.then(()=>{
2     // then block code
3 }
4     .catch((error)=>{
5         // catch block code
6         console.log("Error: ", error);
7     })
8     .finally(()=>{
9         // finally block code
10});
```

.finally(): Example

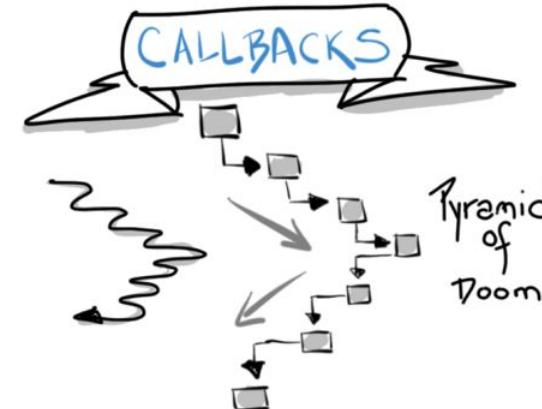
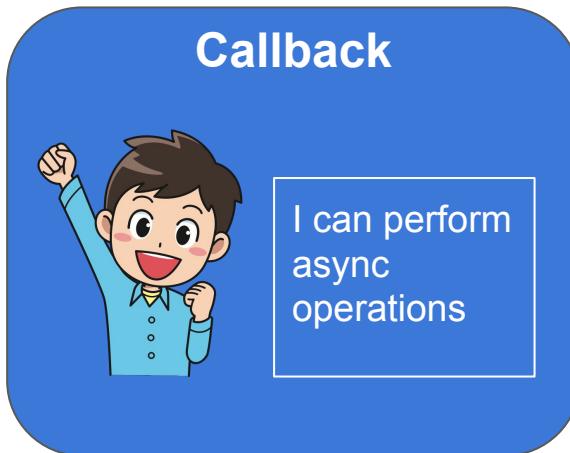
It is possible to chain a block that runs regardless of whether the promise is resolved or rejected using .finally() block.

```
1 // Using .then()
2 fetchDataFromServer()
3   .then(result => {
4     console.log("Success:", result);
5     // Handling the resolved value
6   })
7   .catch((error) => {
8     console.log("Error: ", error);
9   })
10  .finally(() => {
11    console.log("Fetch operation completed.");
12    // Executes regardless of success or failure
13});
```

**finally() block**

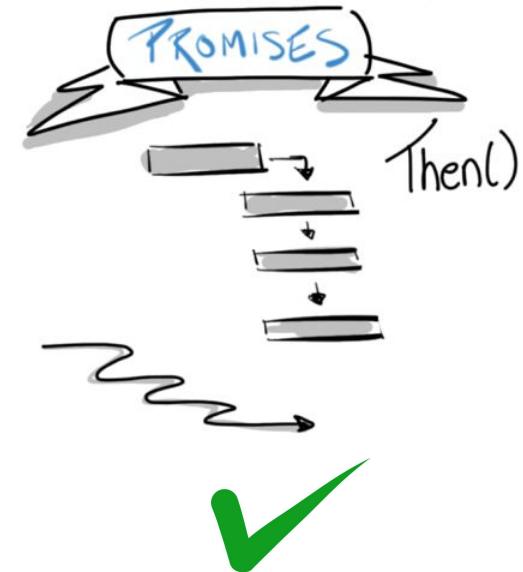
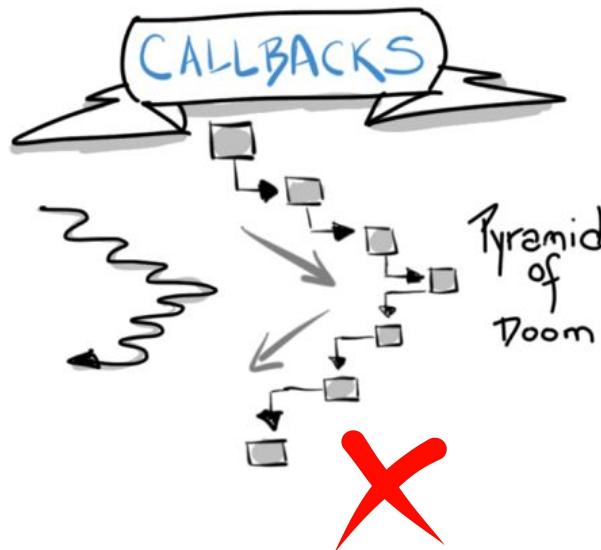
Why do we need Promises??

In the previous lecture we learned that callbacks can be used to implement async programming; then why do we need to make promises?



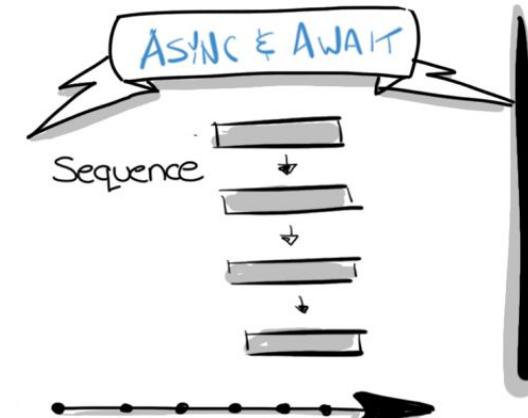
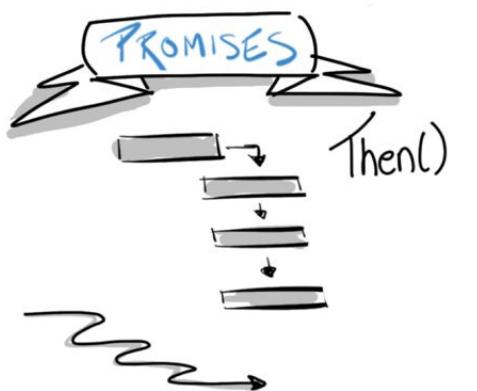
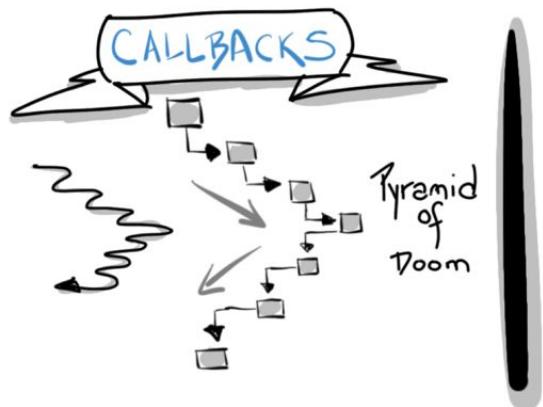
Why do we need Promises??

Callbacks can often lead to cluttered and unreliable code due to nested functions. It is better to chain tasks, where one task starts after the previous one ends, and this is achieved using promises.



Callbacks vs Promises vs Async/await

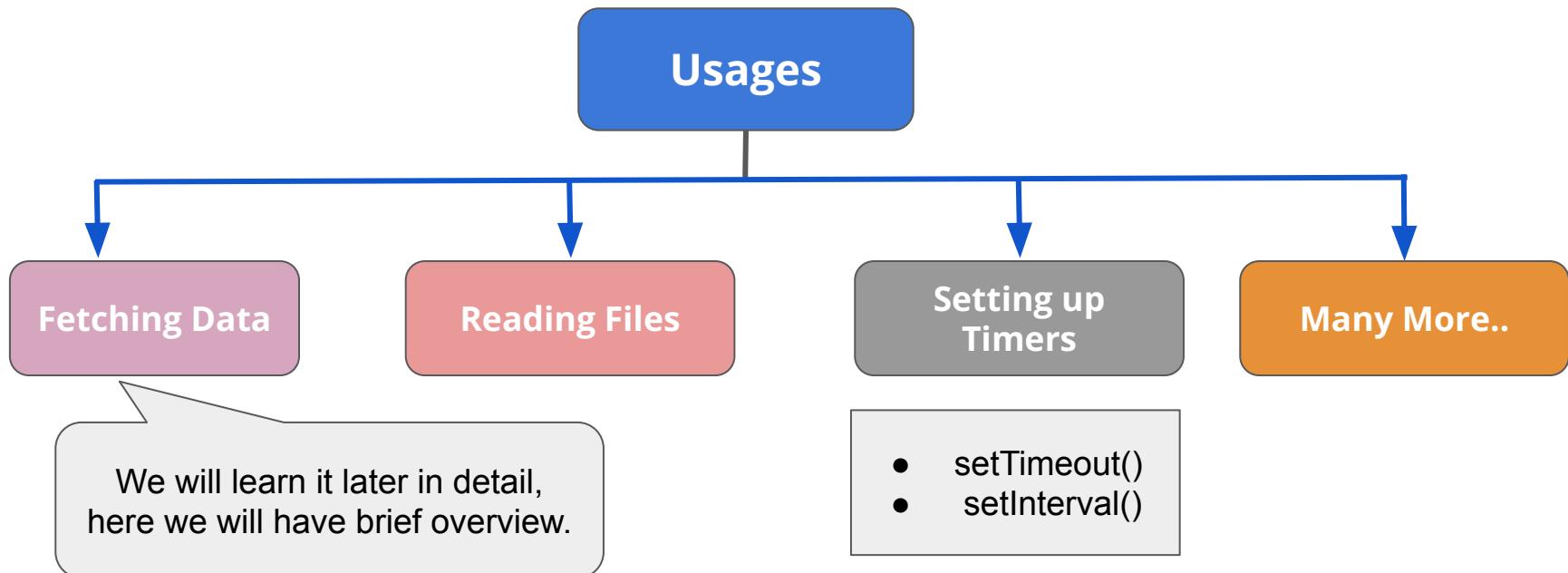
We use callbacks for simple asynchronous tasks, promises for chaining more complex tasks, and `async/await` for writing asynchronous code that appears sequential.



Promise Basic Usages

Basic Promise Usage

Promises in JavaScript simplify handling asynchronous operations. Here are common use cases:



Usage: Fetching Data

We use fetch api to get data from the server.

```
1 // Function to fetch data
2 function fetchData(url) {
3     return new Promise((resolve, reject) => {
4         fetch(url)
5             .then((response) => {
6                 if (!response.ok) {
7                     // On failure, reject the promise
8                     [reject(`HTTP error!: ${response.status}`)];
9                 }
10                return response.json();
11                // Parse JSON from the response
12            })
13            .then((data) => {
14                [resolve(data); // On success return data using resolve]
15            })
16            .catch((error) => {
17                [reject(`Network error: ${error}`)]; // Reject the promise
18            });
19    });
20 }
```

Fetching data from server is an async operation so we used promise for it.

Here we conditionally rejected/resolved the promise

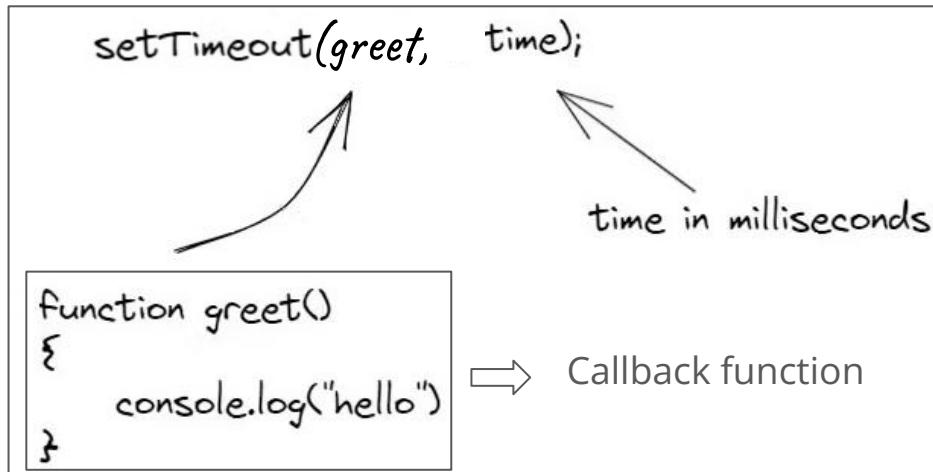
setTimeout and setInterval

with Promises

Pre-discussed: setTimeout

`setTimeout` is a JavaScript function that executes a callback function after a specified delay (in milliseconds).

Syntax:



Pre-discussed: setTimeout Example

We can print specific message after a certain period of time using setTimeout.



```
1 function greet() {  
2     console.log("Hello");  
3 }  
4  
5 // greet function runs after 2 seconds  
6 // i.e. (2000 milliseconds)  
7 setTimeout(greet, 2000);
```

Here we get output after 2000 milliseconds i.e. 2 secs

Output:

Hello

setTimeout wrapped in a Promise

There might be instances where you need to set up timers in JavaScript. In such cases, you can use `setTimeout` in combination with promises

```
1  function delay(ms) {  
2      return new Promise((resolve) => {  
3          setTimeout(resolve, ms);  
4          // Resolve the promise after a delay (in ms)  
5      });  
6  }  
7  
8  delay(2000).then(() => {  
9      console.log("Executed after 2 seconds");  
10 });
```



Creating and returning a promise

Calling the function returning the promise and consuming the returned promise

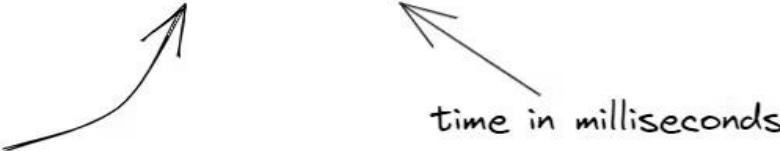
Pre-discussed: setInterval

`setInterval` is a JavaScript function that repeatedly executes a specified callback function at fixed time intervals (in milliseconds).

Syntax:

```
setInterval(greet,    time);  
  
function greet()  
{  
    console.log("hello")  
}
```

time in milliseconds



Pre-discussed: setInterval example

Here we are printing and updating message count every 1000 milliseconds. The `setInterval()` function returns a unique identifier, known as an interval ID, which is stored to later clear the interval

```
1 let count = 0;
2
3 const intervalId = setInterval(() => {
4     console.log(`Message ${++count}`);
5 }, 1000);
```

Output:

Message 1
Message 2
Message 3
..
..

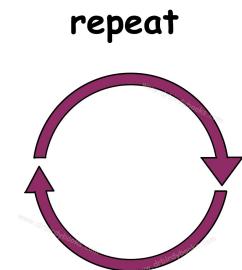
Usage: Setting up timers (SetInterval)

In similar fashion you can use setInterval if you want a task to be repeated after certain interval of time.

```
1  function repeatTask(ms) {  
2      return new Promise(() => {  
3          setInterval(() => {  
4              console.log("Repeating task...");  
5          }, ms);  
6      });  
7  }  
8  
9  console.log("Starting...");  
10  
11 repeatTask(2000);  
12  
13 console.log("Waiting...");
```

Creating and returning a promise

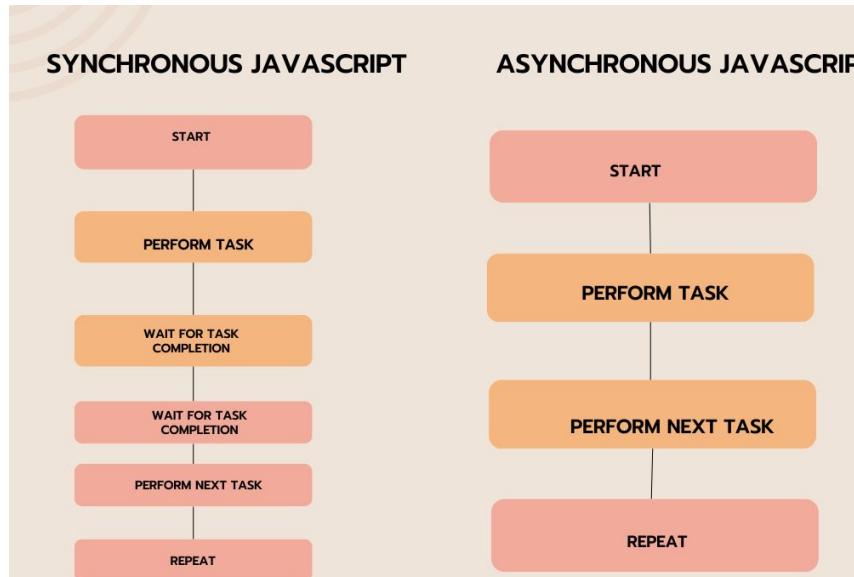
Calling the function returning the promise.



Understanding Asynchronous flow in Promises

What is Asynchronous Flow?

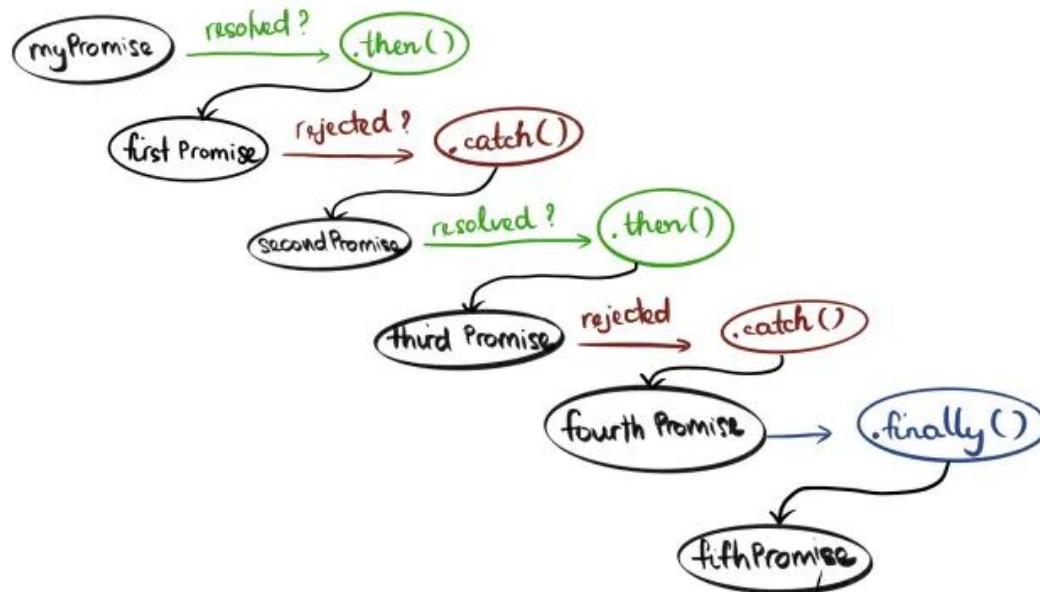
Asynchronous flow allows programs to perform tasks concurrently without blocking other tasks. It enables operations such as fetching data, reading files, or waiting for user input without freezing the program.



Unlike sync operations
async operations do not
need to wait till the other
task complete its
execution.

Asynchronous Flow in Promises

In promises we chain the tasks using then() and catch() blocks. Hence program is executed in chain like manner.

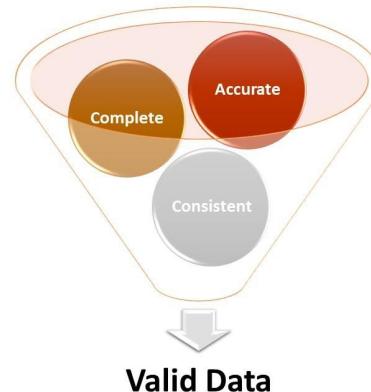


Corresponding block executes only after the execution of block previous block.

Workshop: Question

You're building a registration form where users submit their name, email, and age. The input data needs to be validated asynchronously, such as if name is valid, checking if the email is already registered (with a delay) and ensuring the age is valid.

How would you implement using promises?



A registration form enclosed in a light grey box. It contains three input fields:

- Name
- Email
- Age



Workshop: Solution

Let's write a function using promises for validating name asynchronously.

```
1 // Function to validate the name
2 function validateName(name) {
3     return new Promise((resolve, reject) => {
4         if (!name.trim()) {
5             reject('Name cannot be empty.');
6         } else if (name.length < 3) {
7             reject('Name must be at least 3 characters long.');
8         } else if (/[^a-zA-Z\s]/.test(name)) {
9             reject('Name must only contain letters and spaces.');
10        } else {
11            resolve('Name is valid.');
12        }
13    });
14 }
```

Write functions to validate email and age by yourself.

Workshop: Solution

Let's write validation functions for email and age too.



```
1 function validateEmail(email) {
2     return new Promise((resolve, reject) => {
3         const emailPattern =
4             /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;
5         if (!emailPattern.test(email)) {
6             reject('Invalid email format.');
7         } else {
8             resolve('Email is valid.');
9         }
10    });
11 }
```

Validating Email



```
1 function validateAge(age) {
2     return new Promise((resolve, reject) => {
3         if (isNaN(age) || age < 18) {
4             reject('Must be a number and at least 18.');
5         } else {
6             resolve('Age is valid.');
7         }
8     });
9 }
```

Validating Age

In Class Questions

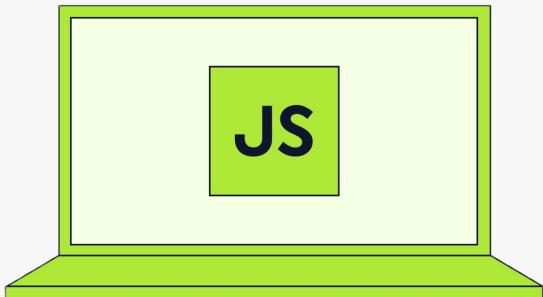
References

1. **MDN Web Docs - JavaScript**: Comprehensive and beginner-friendly documentation for JavaScript.
<https://developer.mozilla.org/en-US/docs/Web/JavaScript>
2. **Eloquent JavaScript**: A free online book covering JavaScript fundamentals and advanced topics.
<https://eloquentjavascript.net/>
3. **JavaScript.info**: A modern guide with interactive tutorials and examples for JavaScript learners.
<https://javascript.info/>
4. **freeCodeCamp JavaScript Tutorials**: Free interactive lessons and coding challenges to learn JavaScript.
<https://www.freecodecamp.org/learn/>

**Thanks
for
watching!**



The Complete Javascript Course



@newtonschool

Lecture 11: Understanding APIs

-Narendra Kumar



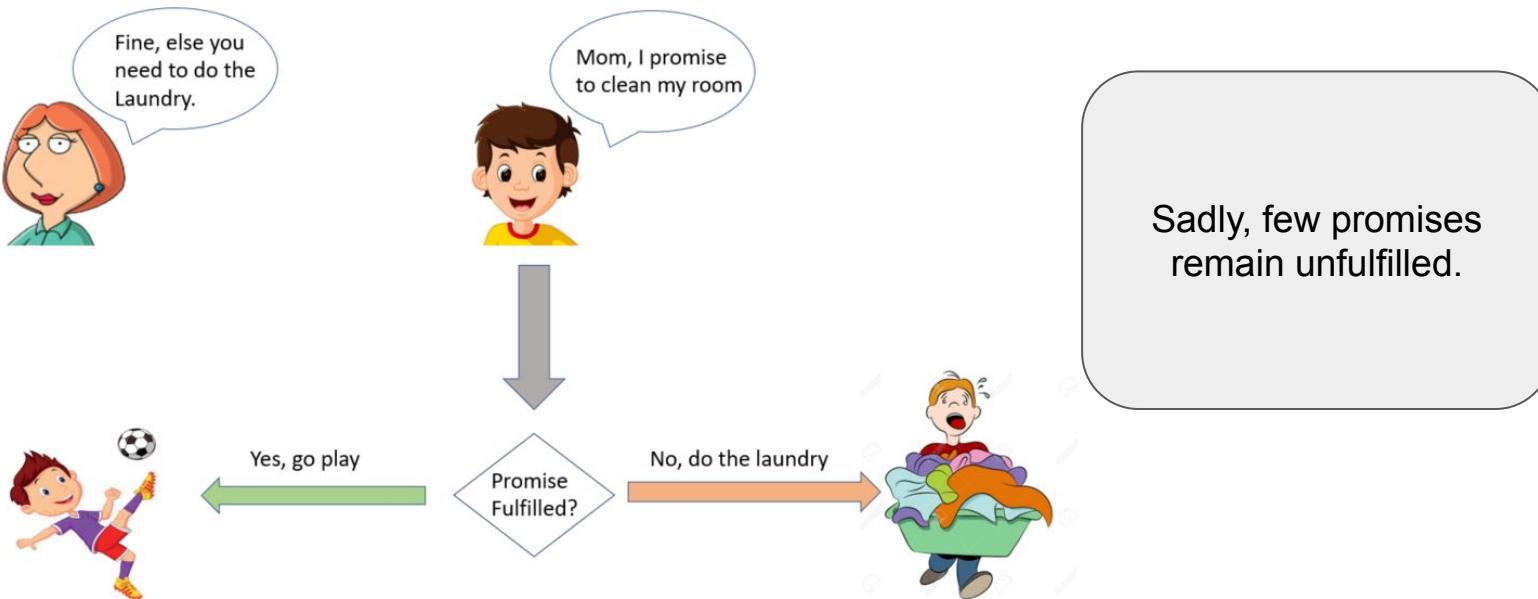
Table of Contents

- Quick Recap
- Understanding APIs
- Rest API: Understanding with fetch()
- Error Handling
 - Basic Error Catching
 - Multiple Error Handlers
 - Recovery Patterns
- Promise: Static Methods
- Status Codes

Quick Recap

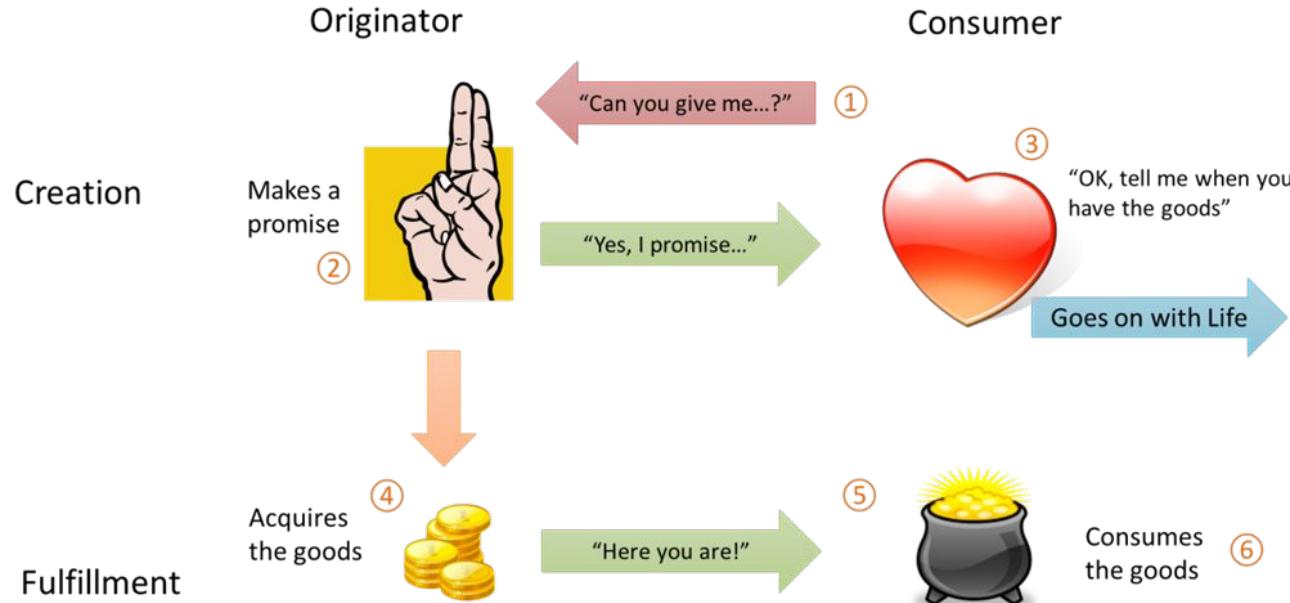
Pre-discussed: Making a Promise

If you did then you must know how valuable promises are. A promise in real life is a commitment or guarantee made by someone.



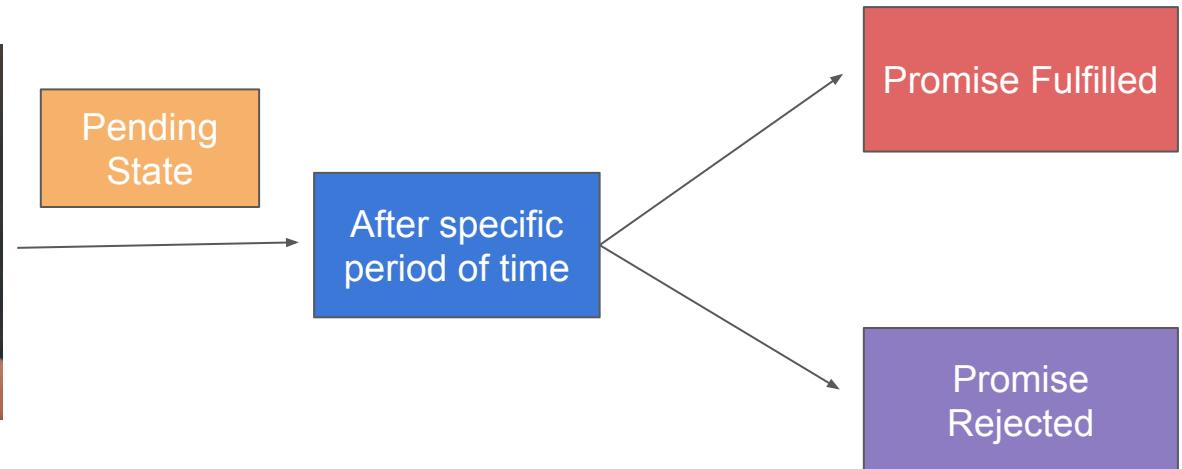
Pre-discussed: Stages of promise

If you did then you must know how valuable promises are. A promise in real life is a commitment or guarantee made by someone.



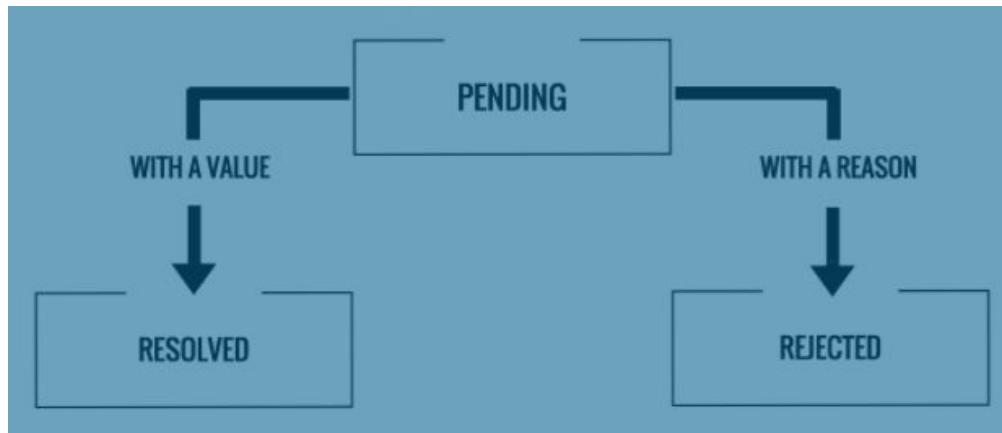
Pre-discussed: Promises in Javascript

A promise is a commitment in programming—a guarantee that something will happen in the future, but not an immediate action. It's a placeholder for future work.



Pre-discussed: Different promise states

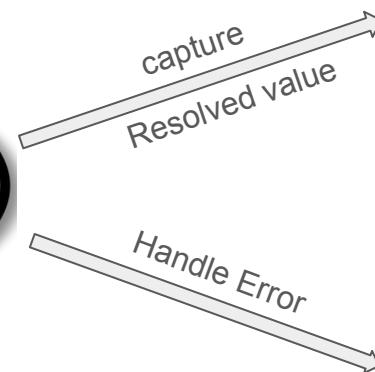
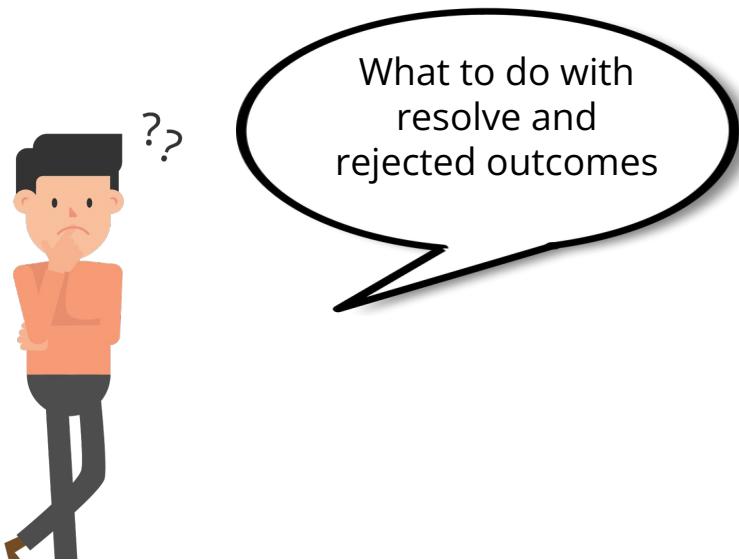
Promises in javascript exists in three states namely, pending, fulfilled, and rejected.



Initially promise remains in pending state, and after sometime it either gets resolved/fulfilled or rejected.

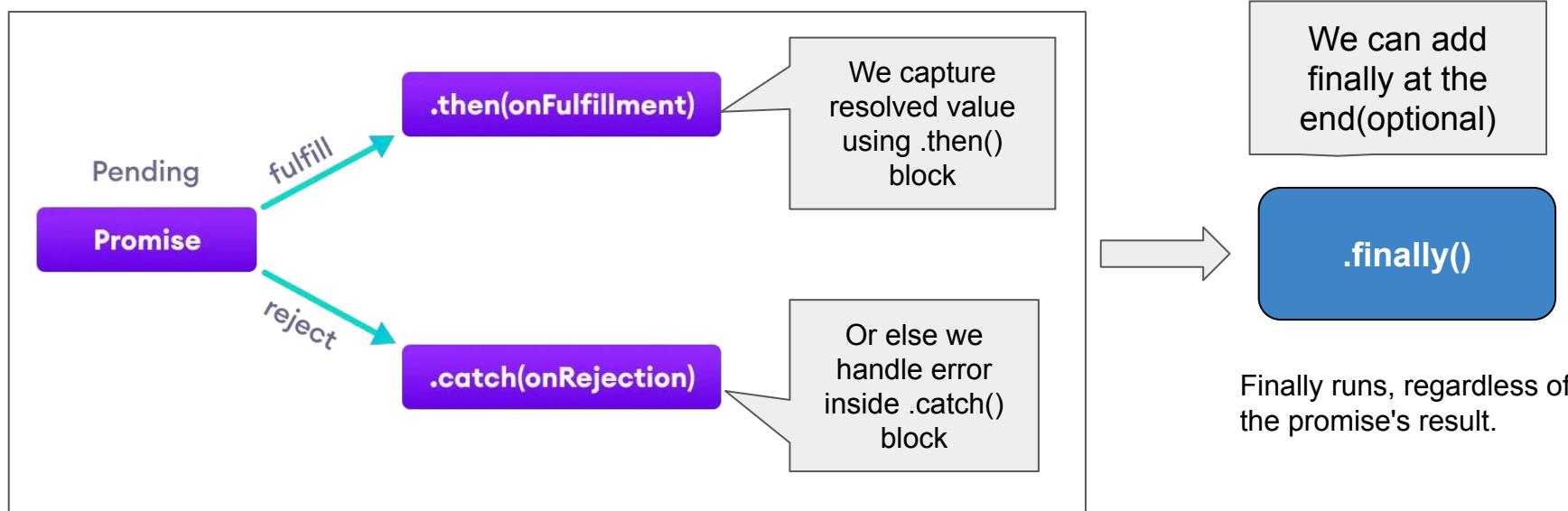
Pre-discussed: Handling Promise

As we have learned that promise is initially at pending state and then after a while it either get resolved or rejected.



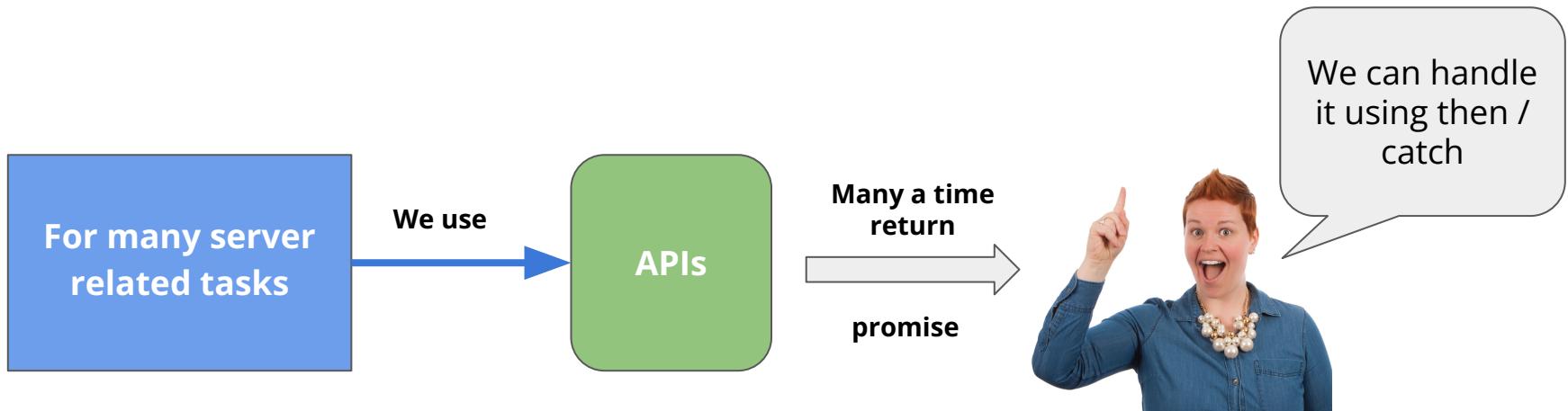
Pre-discussed: Handling Promise

We hope a promise resolves/fulfilled and gives us a value to work with in the `then` block. However, if it gets rejected, we handle it in the `catch` block.



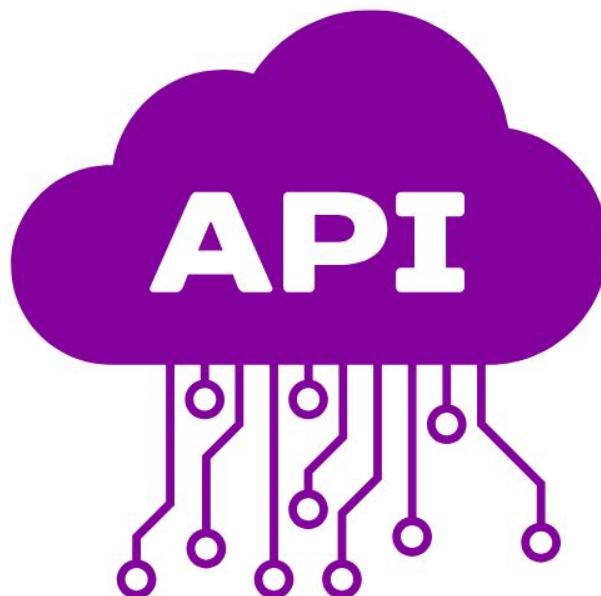
Promises in APIs

Applications often fetch external data using APIs. Since APIs return data asynchronously, they often return Promises.



But What they are exactly?

Before we jump into how to handle promises returned by APIs lets understand what they are.

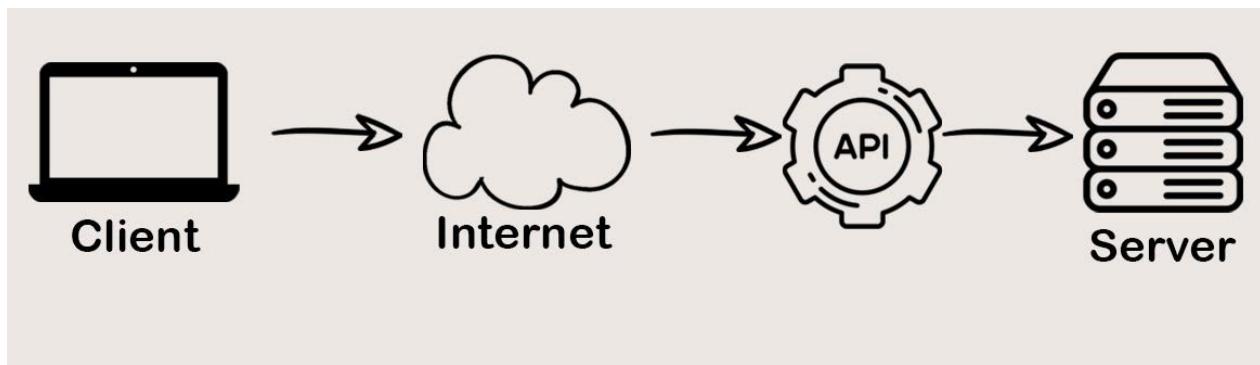


They seem to be some kind of interface, a kind of gateway to other places.

Understanding APIs

What is an API?

API stands for **Application Programming Interface**. An **API** is like a **messenger** that connects two different apps or programs talk to each other.

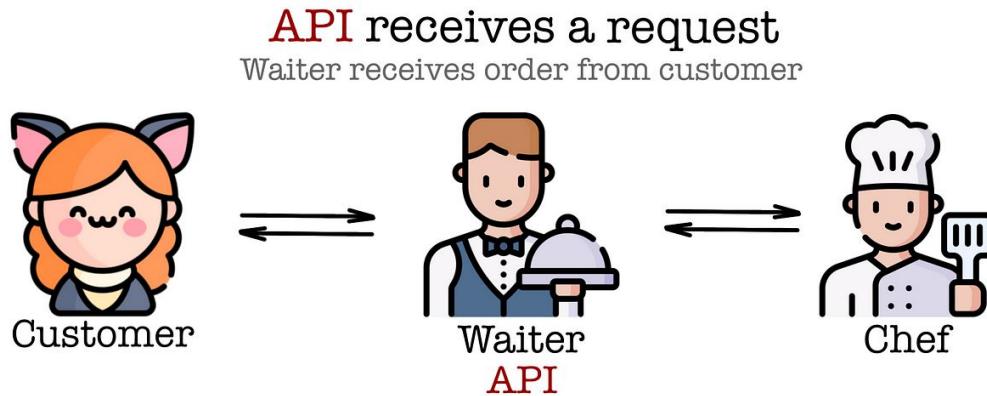


Understood??

Let's
understand it
with an analogy

Let's understand it with an analogy

In this analogy, the waiter is like an API—taking the guest's order (request) to the chef (server) and bringing back the dish (response),



API collects and processes a response, then returns with that response

As waiter would take order from customer, report it to chef and delivers the answer - completed meal from kitchen

The waiter is the entry point for communication with the chef, just like an API connects different software systems.

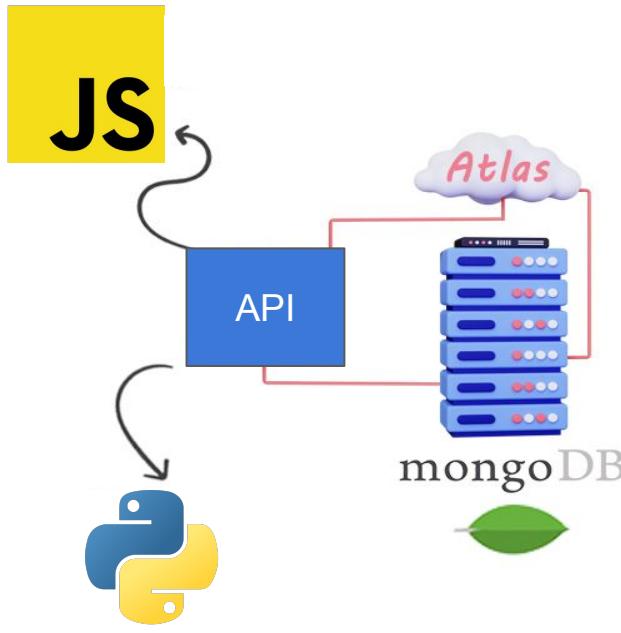
API: Bridge Between User and Server

Just like waiter in previous analogy, API **stands in between end user and server**, facilitating communication.



But Why Not Communicate Directly?

Software often uses different languages and environments, making direct communication difficult. APIs bridge that gap, allowing apps to understand each other.



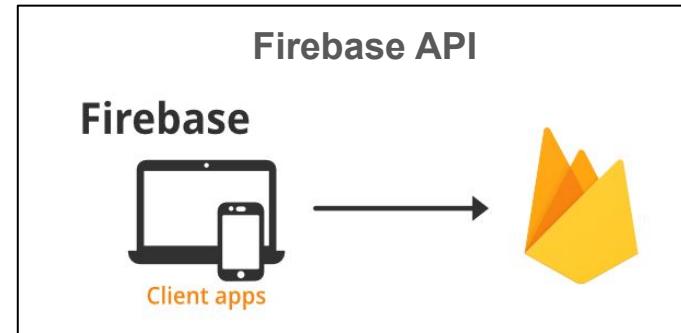
Services like MongoDB and software like ChatGPT provide APIs that allow different languages to communicate efficiently.

Popular Examples of APIs in Action

Let's have a look at few popular apis:-

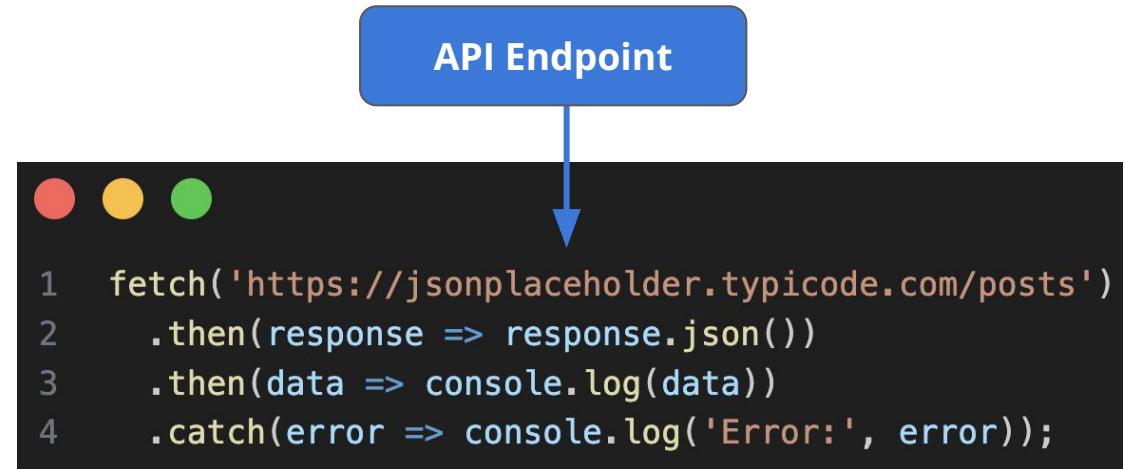
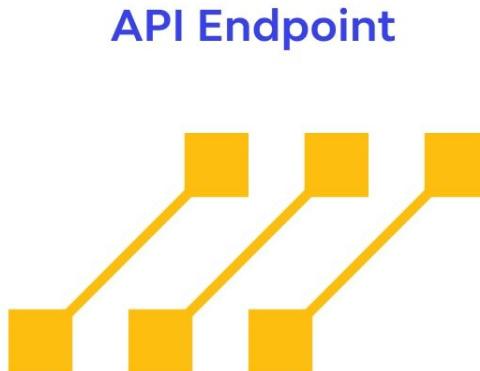


Remember how often we used the Fetch API in previous lectures



API Endpoints: How to Access APIs?

An API endpoint is a specific URL using which an API can be accessed by a client application.

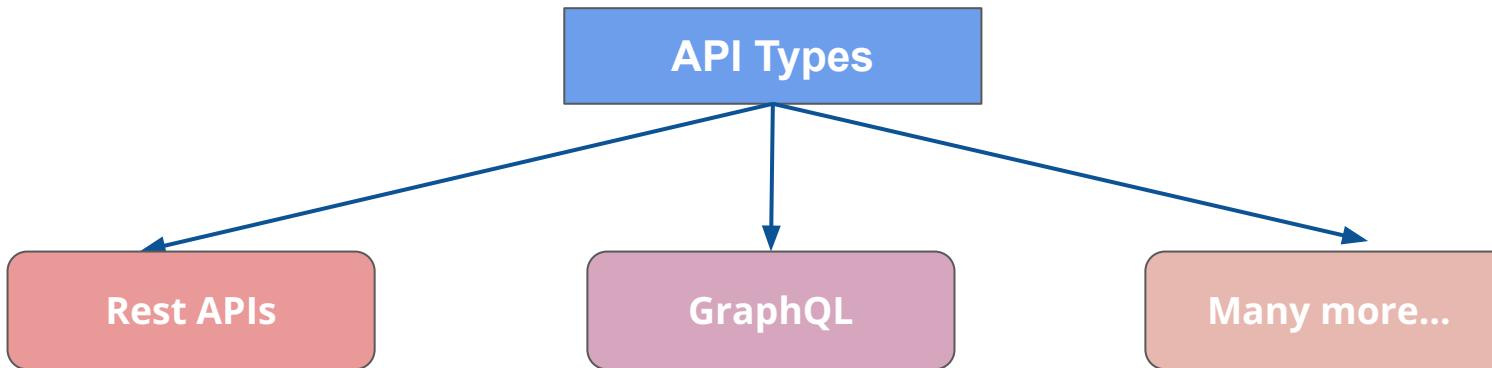


The diagram shows an arrow pointing from a blue rounded rectangle labeled "API Endpoint" to a code snippet. The code is written in JavaScript and demonstrates how to make a fetch request to an API endpoint:

```
1  fetch('https://jsonplaceholder.typicode.com/posts')  
2    .then(response => response.json())  
3    .then(data => console.log(data))  
4    .catch(error => console.log('Error:', error));
```

Types of APIs

There are different types of APIs. Here are few of them:-



Rest API

Understanding with fetch()

Rest API

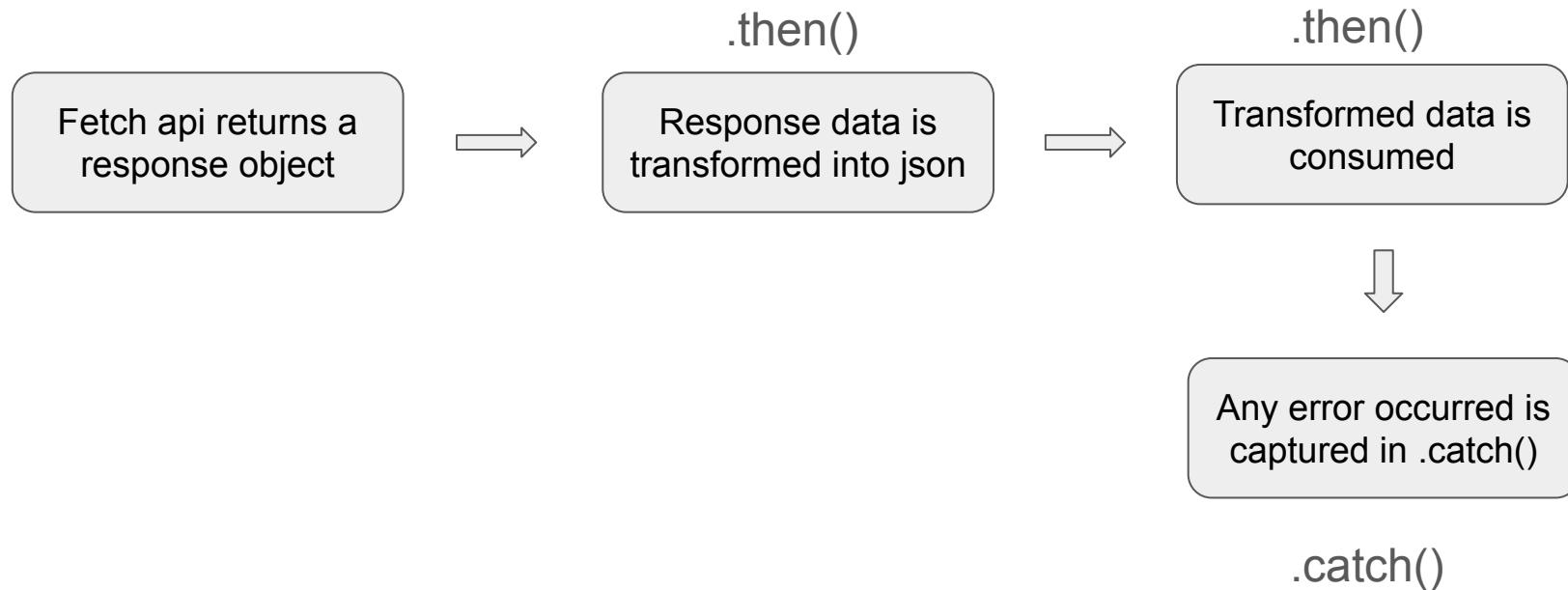
A REST API allows apps to communicate over the internet using simple commands like GET, POST, PUT, and DELETE. Each request is independent and doesn't remember the previous one.



We provide them simple http
request like GET, POST etc, and we
get data in response.

Rest API: fetch api example

The `fetch()` API returns a promise that resolves to a response. `.then()` handles and parses the data, while `.catch()` handles errors.



fetch API: then/catch

The `fetch()` API returns a promise that resolves to a response. `.then()` handles and parses the data, while `.catch()` handles errors.



Settling fetch API: then/catch

Any promise gets settled if it either gets resolved or gets rejected.



```
1  fetch('https://jsonplaceholder.typicode.com/posts/1')
2    .then(response => {
3      if (!response.ok) {
4        throw new Error(`HTTP error! Status: ${response.status}`);
5      }
6      return response.json(); ←
7    })
8    .then(data => {
9      console.log('Post:', data);
10     })
11   .catch(error => {
12     console.error('Error fetching data:', error);
13   });

```

Either returns a
response data or
throws an error

Based on results i.e.
response or error, it
gets passed either to
.then() or .catch()

fetch API: then/catch chaining

Here fetch api returns a promise, when promise gets resolved it returns a response object which is received by the first then block and values gets passed down in the chain.

.then() chaining,
values returned
from previous
gets captured in
next .then()

```
1  fetch('https://jsonplaceholder.typicode.com/posts/1')
2    .then(response => {
3      if (!response.ok) {
4        throw new Error(`HTTP error! Status: ${response.status}`);
5      }
6      return response.json();
7    })
8    .then(data => {
9      console.log('Post:', data);
10     })
11   .catch(error => {
12     console.error('Error fetching data:', error);
13   });

```

fetch: then/catch Data transformation

The received data from the fetch is in form of string which needs to be transformed into a json object for further processing.

```
1  fetch('https://jsonplaceholder.typicode.com/posts/1')
2    .then(response => {
3      if (!response.ok) {
4        throw new Error(`HTTP error! Status: ${response.status}`);
5      }
6      return response.json(); ←
7    })
8    .then(data => {
9      console.log('Post:', data);
10     })
11   .catch(error => {
12     console.error('Error fetching data:', error);
13   });

```

Response data is
transformed into json
object

fetch: then/catch Data transformation

The received data from the fetch is in form of string which needs to be transformed into a json object for further processing.

```
1  fetch('https://jsonplaceholder.typicode.com/posts/1')
2    .then(response => {
3      if (!response.ok) {
4        throw new Error(`HTTP error! Status: ${response.status}`);
5      }
6      return response.json();
7    })
8    .then(data => {
9      console.log('Post:', data); ←
10     })
11    .catch(error => {
12      console.error('Error fetching data:', error);
13    });

```

Transformed response
data returned by .then()
gets received by next
.then() in the chain

fetch: then/catch Catching the Error

Error occurred whether due to server or error thrown from anywhere inside our code gets caught in the .catch().

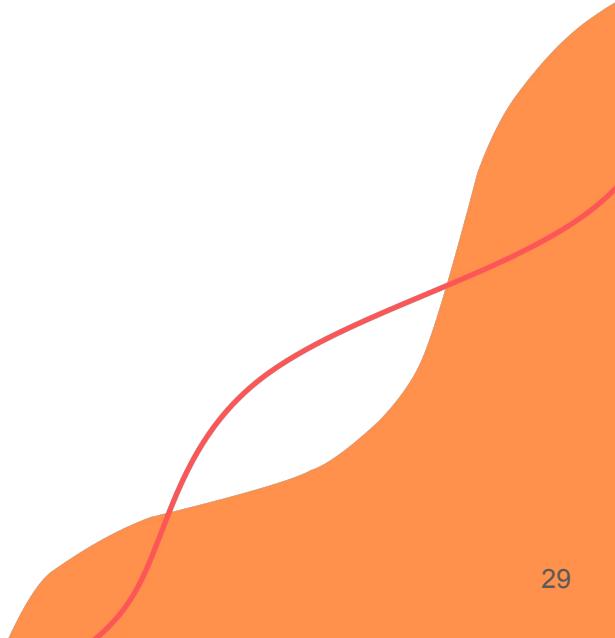


```
1  fetch('https://jsonplaceholder.typicode.com/posts/1')
2    .then(response => {
3      if (!response.ok) {
4        throw new Error(`HTTP error! Status: ${response.status}`);
5      }
6      return response.json();
7    })
8    .then(data => {
9      console.log('Post:', data);
10   })
11   .catch(error => {
12     console.error('Error fetching data:', error);
13   });

```

Error thrown from
inside our code

Error get captured in
.catch()



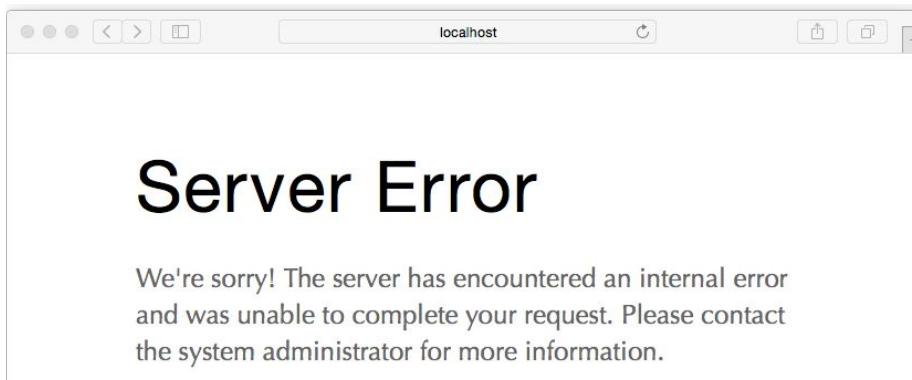
then/catch

Workshop

Workshop: Question 1

You are building a simple weather application that fetches data from a weather API. If the fetch operation fails, you want to display an error message to the user. How would you structure the promise to handle this error?

How you would handle error using catch()?

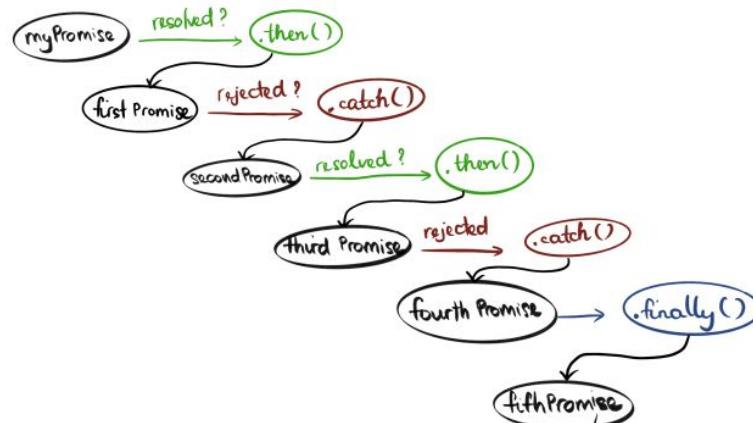


A screenshot of a web browser window. The address bar shows "localhost". The main content area displays the text "Server Error" in large, bold, black font. Below it, a smaller text message reads: "We're sorry! The server has encountered an internal error and was unable to complete your request. Please contact the system administrator for more information."

Workshop: Question 2

You need to fetch user data from an API and then fetch their favorite products from another API.

How would you chain these two promises and handle any error that may occur in either of the steps?



fetch API: HTTP methods

Fetch APIs use standard HTTP methods to perform actions on resources. Each method has a specific purpose and meaning.



GET

Retrieve a
resource



POST

Create a
resource



PUT

Replace a
resource



PATCH

Update a
resource



DELETE

Delete a
resource

fetch() - GET method

Let's understand REST API with an example of fetch api. If we don't specify anything in fetch then it does GET request by default. GET request retrieves a resource from the server.

URL for the GET request



```
1  fetch('https://jsonplaceholder.typicode.com/posts')
2    .then(response => response.json())
3    .then(data => console.log(data))
4    .catch(error => console.log('Error:', error));
```

It would retrieve data from the API endpoint.

The response is typically in JSON format, and we can process it using `.json()`.

fetch() - POST method

If we want to send some data instead of requesting we can do a post request with appropriate data specifying what data is being sent in the body.

```
● ● ●  
1  fetch('https://jsonplaceholder.typicode.com/posts', {  
2      method: 'POST', →  
3      headers: {  
4          'Content-Type': 'application/json' →  
5      },  
6      body: JSON.stringify({  
7          title: 'New Post',  
8          body: 'This is a new post', userId: 1 →  
9      })  
10 }  
11     .then(response => response.json())  
12     .then(data => console.log(data))  
13     .catch(error => console.log('Error:', error));
```

To send data we do POST request

Specifying data format is JSON

In the body, we convert the JSON data to a string so it can be sent over the network.

fetch() - PATCH method

PATCH request updates a resource or data stored in the server.



```
1  fetch('https://jsonplaceholder.typicode.com/posts/1', {  
2      method: 'PATCH',  
3      headers: {  
4          'Content-Type': 'application/json'  
5      },  
6      body: JSON.stringify({ title: 'Updated Post Title' })  
7  })  
8      .then(response => response.json())  
9      .then(data => console.log(data))  
10     .catch(error => console.log('Error:', error));
```

We need to pass an identifier to identify which item to patch in database

And need to supply the value to update.

fetch() - PUT method

Similarly we can perform PUT request:-



```
1  fetch('https://jsonplaceholder.typicode.com/posts/1', {  
2      method: 'PUT',  
3      headers: {  
4          'Content-Type': 'application/json',  
5      },  
6      body: JSON.stringify({  
7          id: 1,  
8          title: 'Updated Post',  
9          body: 'This is an updated post.',  
10         userId: 1,  
11     }),  
12 })  
13     .then(response => response.json())  
14     .then(data => console.log(data))  
15     .catch(error => console.log('Error:', error));
```

Instead of updating the resource, PUT request replaces the entire resource.

This new data would replace the data entry with identifier 1

fetch() - DELETE method

In the same manner as that of PATCH we can perform deletion.



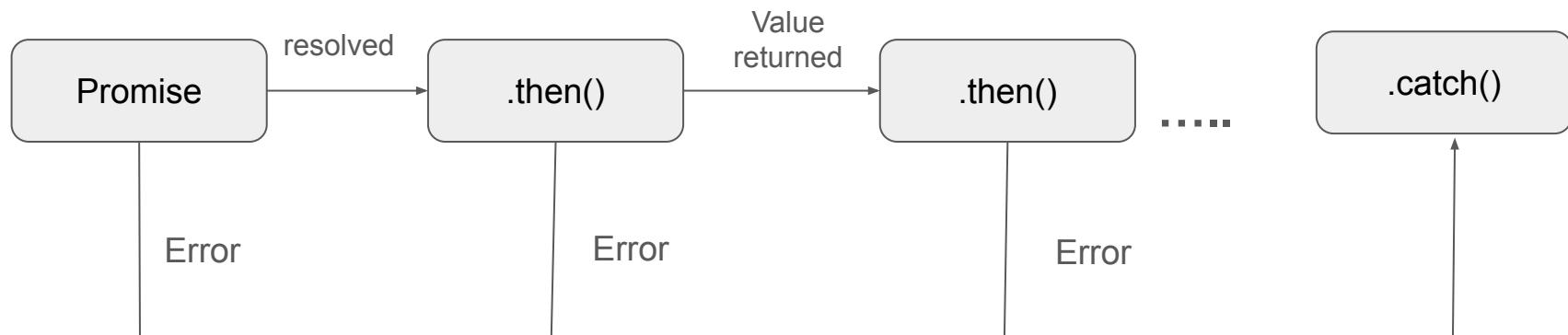
```
1 fetch('https://jsonplaceholder.typicode.com/posts/1', {  
2   method: 'DELETE'  
3 })  
4   .then(response => response.json())  
5   .then(data => console.log('Deleted:', data))  
6   .catch(error => console.log('Error:', error));
```

Unlike PATCH we
don't need to
pass body in
DELETE request.

Error Handling

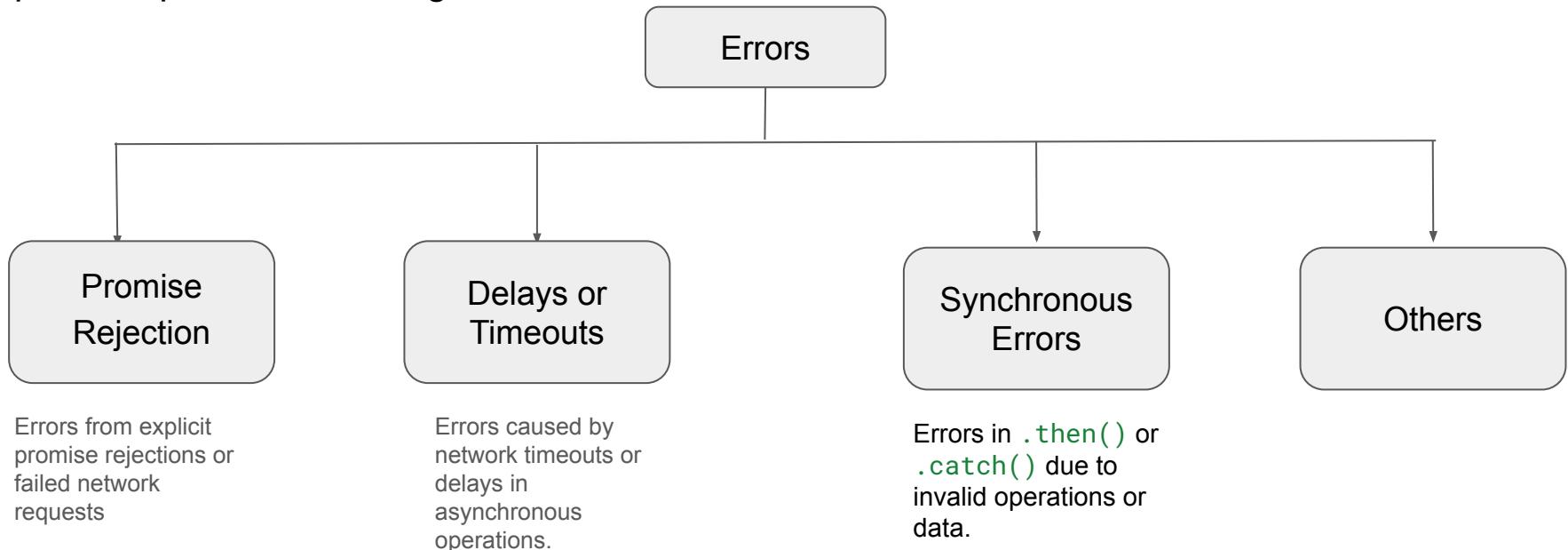
Error Handling: Catch block

The `.catch()` block is used to handle any errors that occur during the execution of a promise or any promise chain. It catches both rejections and exceptions that occur in the promise lifecycle.



Handling Errors

There are following types of errors which can be thrown during the execution of a promise/promise handling.



Promise Rejection

Promise rejection errors occur when a promise is explicitly rejected or if an operation inside the promise fails.

A promise can be manually rejected using `reject()`, or due to network/HTTP failures like 404 or 500 errors, especially in API requests.



Promise Rejection: Manual rejection

Promise rejection errors occur when a promise is explicitly rejected or if an operation inside the promise fails.

```
1 let myPromise = new Promise((resolve, reject) => {
2     reject('Something went wrong');
3 });
4
5 myPromise
6     .catch(error => {
7         console.error('Caught error:', error);
8     });

```

Here we simply caught error
and displayed using
console.error()

Promise Rejection: Network Http failures

For network or HTTP failures, check the response status and handle errors using `.catch()` or inside the `fetch()` response block.

```
1  fetch('https://invalidurl.com')
2    .then(response => {
3      if (!response.ok) {
4        throw new Error('Network error');
5      }
6      return response.json();
7    })
8    .catch(error => {
9      console.error('Network error:', error);
10 });

```

It might happened that you got a response but it is a failed response, in that case you need to throw an error from inside `.then()`

Delays or Timeouts

To handle timeouts, create a timeout promise and reject if the operation exceeds the allowed time.

```
1 let timeoutPromise = new Promise((resolve, reject) => {
2     setTimeout(() => reject('Operation timed out'), 5000);
3 });
4
5 timeoutPromise.catch(error => {
6     console.error('Timeout error:', error);
7 });
```

Most asynchronous APIs, especially network requests like HTTP fetch calls, do have a built-in timeout to prevent the request from hanging indefinitely

Synchronous Errors

Synchronous errors like invalid input/data, type errors etc, could occur for which we need to throw an error manually while creating the promise.

```
1 let processData = new Promise((resolve, reject) => {
2     let input = null;
3     if (!input) {
4         reject('Invalid data');
5     }
6 });
7
8 processData.catch(error => {
9     console.error('Data error:', error);
10});
```



Invalid data

Handling Errors at different stages

We can add error handlers at different points to manage errors at each stage.

```
1  fetch('https://jsonplaceholder.typicode.com/posts')
2  .then(response => {
3      if (!response.ok) {
4          throw new Error('Network error: Failed to fetch posts');
5      }
6      return response.json(); // Parse the JSON data
7  })
8  .then(posts => {
9      return fetch('https://jsonplaceholder.typicode.com/comments')
10     .then(response => {
11         if (!response.ok) {
12             throw new Error('Network error: Failed to fetch comments');
13         }
14         return response.json(); // Parse the comments
15     })
16     .catch(commentError => {
17         console.error('Error in fetching comments:', commentError);
18         throw new Error('Comments stage failed');
19     });
20 }
21 .catch(error => {
22     // General error handling for any unhandled errors in chain
23     console.error('General error occurred:', error);
24 })
```

Generally we place multiple .catch() to handle specific type of error at different places

Throws an error only if comment fetching fails.

Handles errors anywhere in the code.

Recovering from errors

Recovering from Errors refers to the process of handling errors in a way that allows the program to continue running or perform a fallback action instead of crashing.

```
1  fetch('https://jsonplaceholder.typicode.com/posts')
2  .then(response => {
3    if (!response.ok) {
4      throw new Error('Network error: Failed to fetch posts');
5    }
6    return response.json();
7  })
8  .then(data => {
9    console.log('Fetched posts:', data);
10 })
11 .catch(error => {
12   console.error('Error occurred:', error);
13   // Fallback: Provide default data or retry
14   const fallbackData = [{ id: 1, title: 'Fallback post' }];
15   console.log('Using fallback data:', fallbackData);
16 });


```

Let's say that fetch operation failed to retrieve data from the server, in that case we can provide some fallback data.

Recovering from errors

Or we can retry by sending one more fetch request to the server:-

```
1  function fetchWithRetry(url, retries = 3) {  
2      return fetch(url)  
3          .then(response => {  
4              if (!response.ok) {  
5                  throw new Error('Network error');  
6              }  
7              return response.json();  
8          })  
9          .catch(error => {  
10              if (retries > 0) {  
11                  console.log(`Retrying... Attempts left: ${retries}`);  
12                  return fetchWithRetry(url, retries - 1); // Retry  
13              } else {  
14                  console.error('Max retries reached');  
15                  throw error; // Fail after retries  
16              }  
17          });  
18      }  
19  
20  fetchWithRetry('https://jsonplaceholder.typicode.com/posts');
```

Here, we are recursively calling `fetchWithRetry()`, in case network error occurs along with limiting the number of retries.

Recovering from errors

Or else we can gracefully show minimal or different content

```
1  fetch('https://jsonplaceholder.typicode.com/posts')
2    .then(response => response.json())
3    .then(data => {
4      // Process data
5    })
6    .catch(error => {
7      console.error('Showing limited content');
8      // Show minimal or default content
9      displayMinimalContent();
10 });

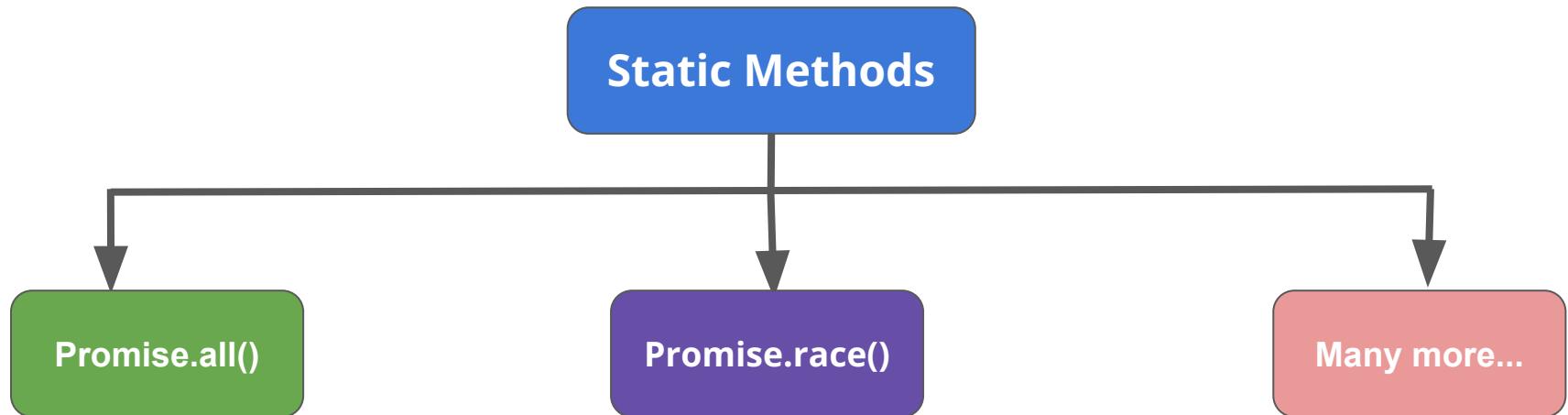
```

Or we can simply state
“Difficulty in fetching data, try
again later”.

Promise: Static Methods

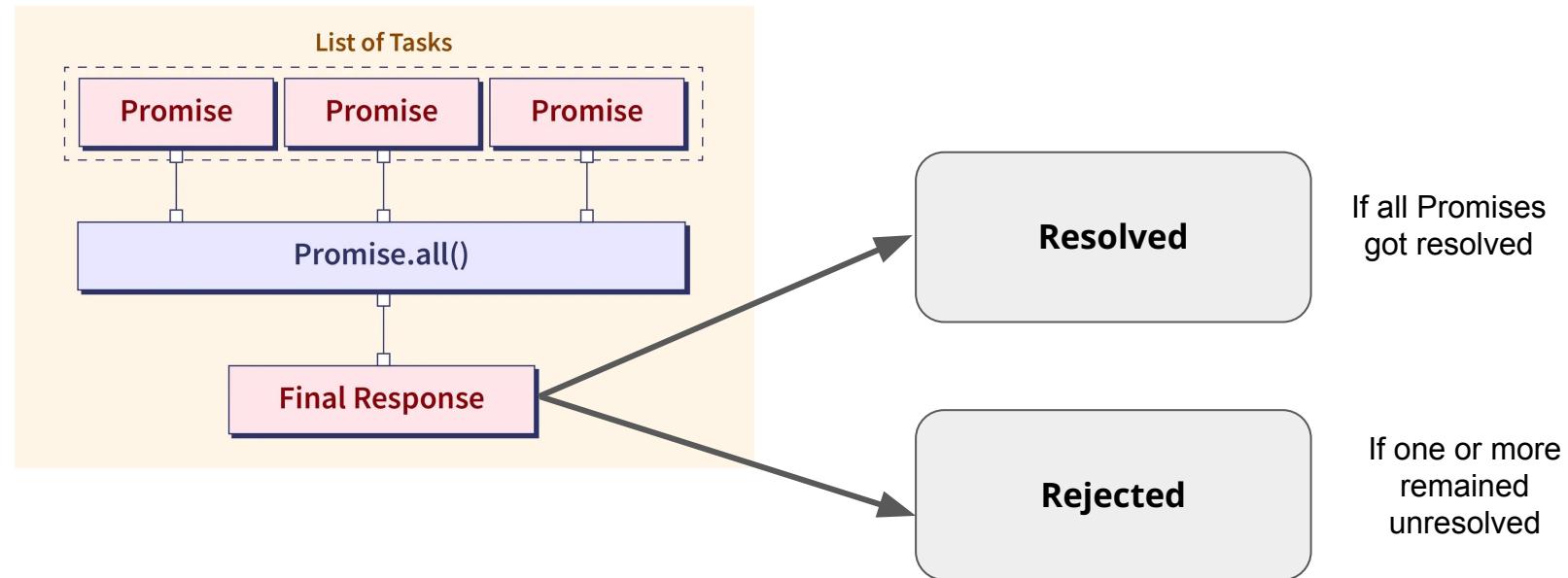
Promise: What are Static methods?

Static methods of the `Promise` class are called on the `Promise` constructor to manage multiple promises or handle resolution globally.



Static Method: Promise.all()

Promise.all() is a static method that takes an array of promises and returns a single promise. It resolves when all promises resolve or rejects as soon as any promise rejects.



Promise.all(): Example

Promise.all() is a static method that takes an array of promises and returns a single promise. It resolves when all promises resolve or rejects as soon as any promise rejects.

```
● ● ●  
1 const promise1 = Promise.resolve(10);  
2 const promise2 = Promise.resolve(20);  
3 const promise3 = Promise.resolve(30);  
4  
5 Promise.all([promise1, promise2, promise3])  
6   .then(values => {  
7     console.log(values);  
8     // Output: [10, 20, 30]  
9   })  
10  .catch(error => {  
11    console.log(error);  
12  });
```

Here we have three promises which returns 10, 20 and 30 respectively as resolved value

Since all of them are getting resolved, Promise.all() will return an array of values

Promise.all(): Example

Let's say one of the promise is not getting resolved.

```
1 const promise1 = Promise.resolve(10);
2 const promise2 = Promise.reject('Error occurred');
3 const promise3 = Promise.resolve(30);
4
5 Promise.all([promise1, promise2, promise3])
6   .then(values => {
7     console.log(values);
8     // This will not execute
9   })
10  .catch(error => {
11    console.log(error);
12    // Output: 'Error occurred'
13  });
```

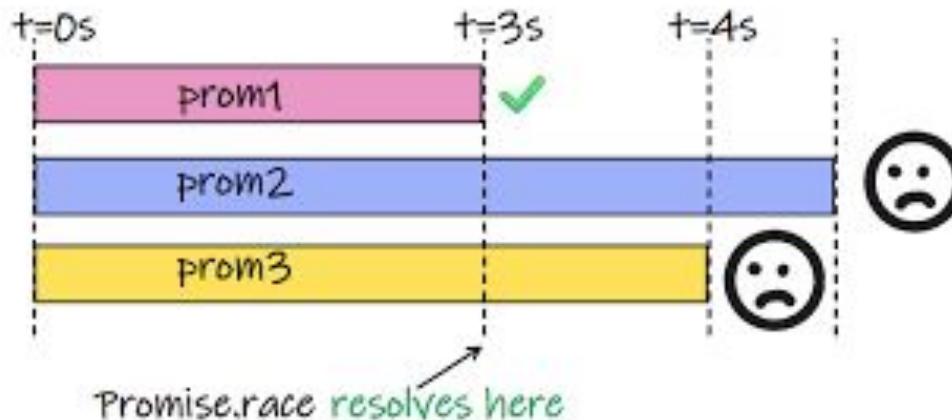
This promise is being rejected

So instead an error would be thrown

And error gets captured by catch block

Static Method: Promise.race()

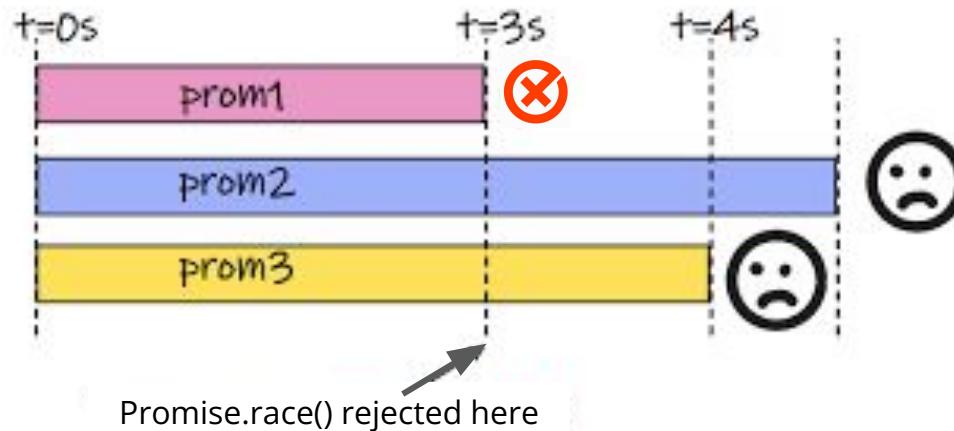
`Promise.race()` is a static method that takes an array of promises and returns a single promise, which resolves or rejects as soon as one of the promises settles.



Since the first promise in `Promise.race` is settled with state `resolve`, `Promise.race` will be *resolved*.

Static Method: Promise.race()

Let's say first settled promise gets rejected, Promise.race() would also be settled as rejected.



Since the first promise in Promise.race is settled with state reject, Promise.race will be *rejected*.

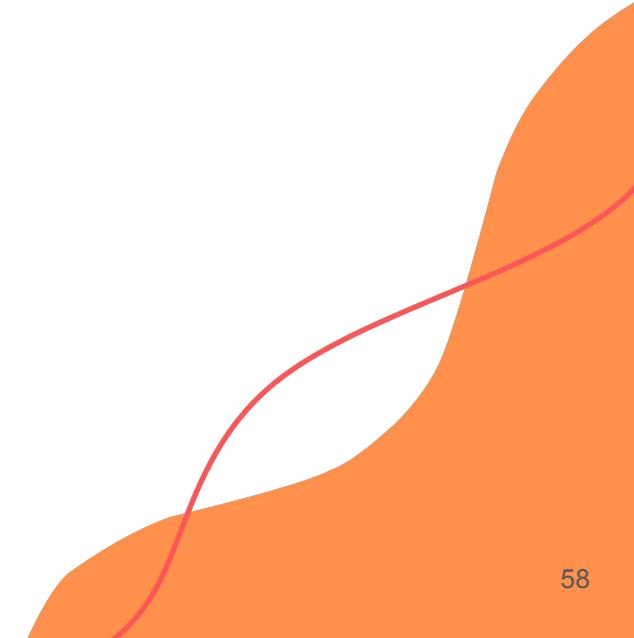
Promise.all(): Example

We are fetching data from a URL and setting a timeout. If the timeout occurs before the data is fetched, `Promise.race()` will reject; otherwise, it will resolve with the data.



```
1 const fetchData = fetch('https://api.example.com/data');
2 const timeout = new Promise(_, reject) =>
3   setTimeout(() => reject('Request timed out'), 3000)
4 );
5
6 Promise.race([fetchData, timeout])
7   .then(response => {
8     console.log('Data fetched:', response);
9   })
10  .catch(error => {
11    console.error('Error:', error);
12    // Output: 'Request timed out' if timeout occurs
13  });
```

If data is fetched before 3000 milliseconds then promise would be resolved else it would be rejected.



Promise.all() Workshop

Workshop: Question

You had a form where users submit their name, email, and age. You have validated it, now you need to consume the promises and process the data only if all validations pass. If any validation fails, an error message should be displayed to the user.

How can you consume the promise?

Consume the
Promise



The form consists of three horizontal input fields. The first field is labeled "Name", the second is labeled "Email", and the third is labeled "Age". Each label is positioned above its corresponding input field.

Consume it
using
then/catch

Workshop: Solution

Let's consume the promise:-

```
1 // Function to handle form submission
2 function handleFormSubmission(name, email, age) {
3     // Validate all fields using promises and consume them
4     Promise.all([
5         validateName(name),
6         validateEmail(email),
7         validateAge(age)
8     ])
9     .then((validationResults) => {
10         // All validations passed, process the form
11         console.log('Form submission successful!');
12         console.log(validationResults);
13         // You can log validation results if needed
14         // Proceed to save data or display success message
15     })
16     .catch((error) => {
17         // Handle the first validation error that occurs
18         console.log('Form submission failed:', error);
19     });
20 }
```

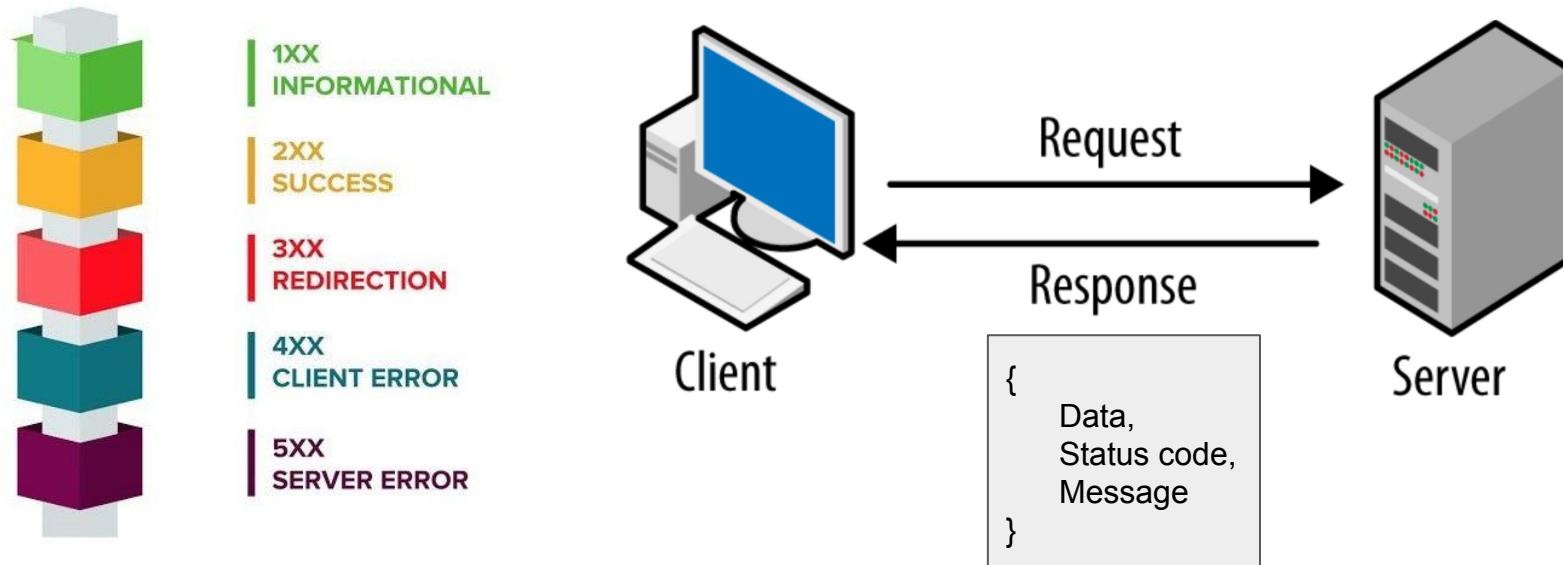
Here we used `Promise.all()` to consume all the promises at once.

Status Codes

Signalling between Client and Server

Status Codes: Signaling Success and Failure

When a request is processed, the server signals the outcome (success or failure) using status codes, providing clear communication about the operation's result.



Status Code: Retrieving Status Code

When you make a fetch request, it returns a response with a status code. You can access the status code using `response.status` to check the outcome of the request.

```
1  fetch('https://jsonplaceholder.typicode.com/posts')
2    .then(response => {
3      console.log('Status Code:', response.status);
4      // Retrieves and logs the status code
5      return response.json();
6      // Process the response body if needed
7    })
8    .then(data => {
9      console.log('Data:', data);
10    })
11   .catch(error => {
12     console.log('Error:', error);
13   });

```

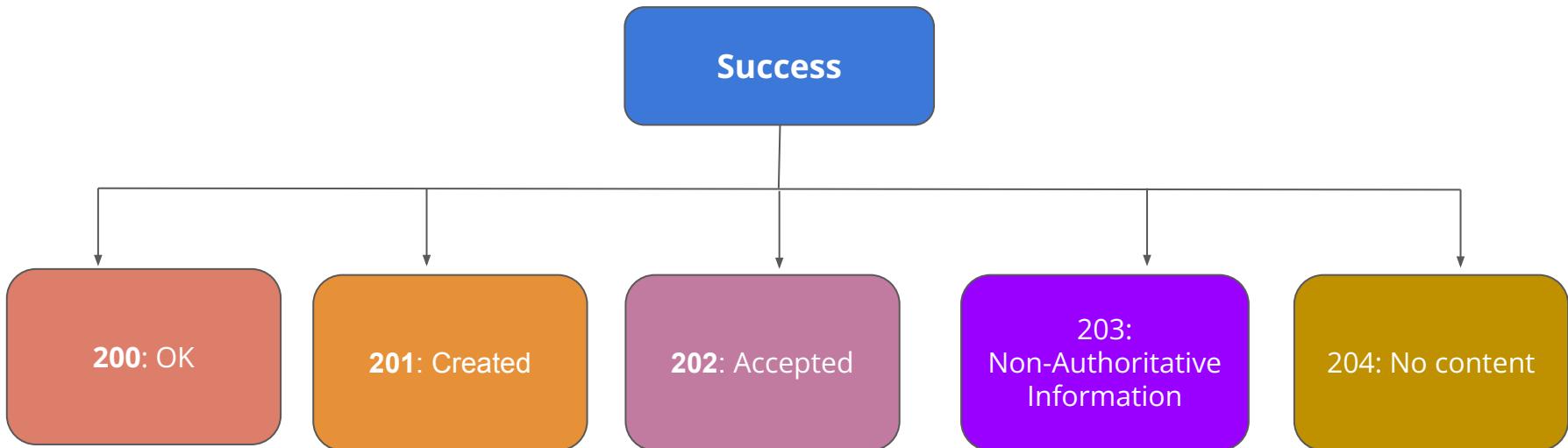
Here we are capturing status code

We can use some other variable let say *data*, then we get status code using:-

data.status

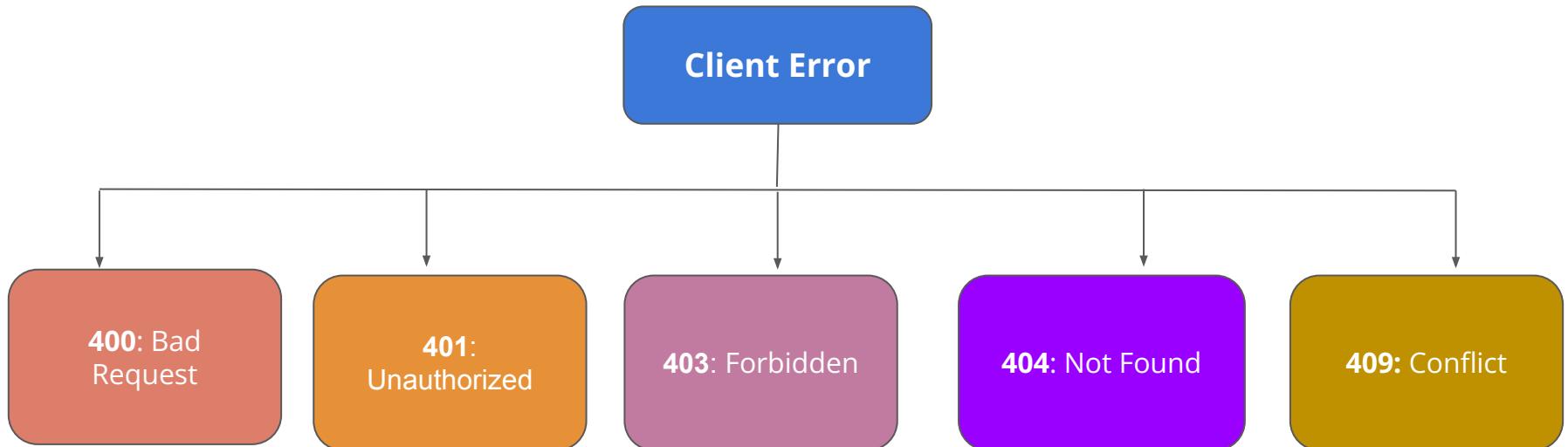
Status Code: Success

Success status codes (200-299) indicate that the request was successfully processed by the server.



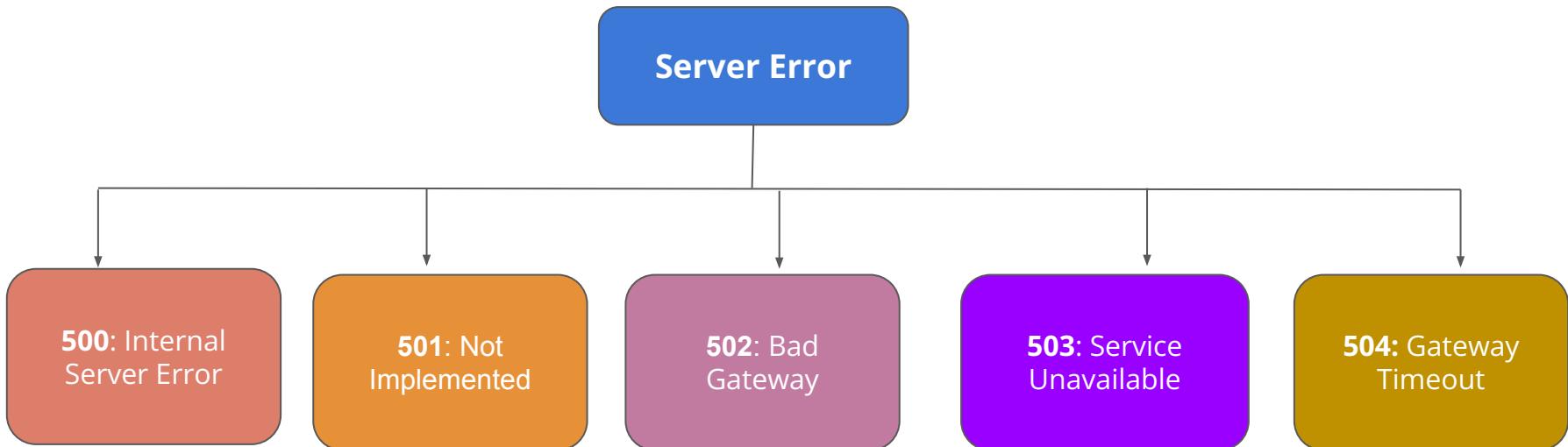
Status Codes: Client Error

Client error status codes (400-499) indicate that the request was invalid or cannot be processed due to issues on the client side.



Status Codes: Server Error

Client error status codes (400-499) indicate that the request was invalid or cannot be processed due to issues on the client side.



In Class Questions

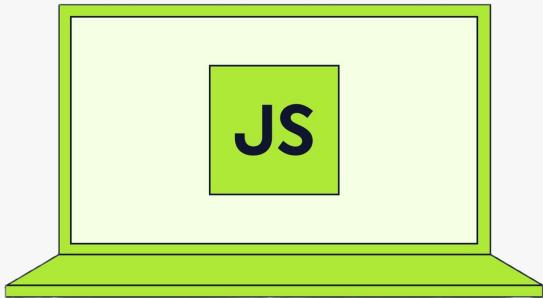
References

1. **MDN Web Docs - JavaScript:** Comprehensive and beginner-friendly documentation for JavaScript.
<https://developer.mozilla.org/en-US/docs/Web/JavaScript>
2. **Eloquent JavaScript:** A free online book covering JavaScript fundamentals and advanced topics.
<https://eloquentjavascript.net/>
3. **JavaScript.info:** A modern guide with interactive tutorials and examples for JavaScript learners.
<https://javascript.info/>
4. **freeCodeCamp JavaScript Tutorials:** Free interactive lessons and coding challenges to learn JavaScript.
<https://www.freecodecamp.org/learn/>

**Thanks
for
watching!**



The Complete Javascript Course



@newtonschool

Lecture 13: async / await

-Vishal Sharma



Table of Contents

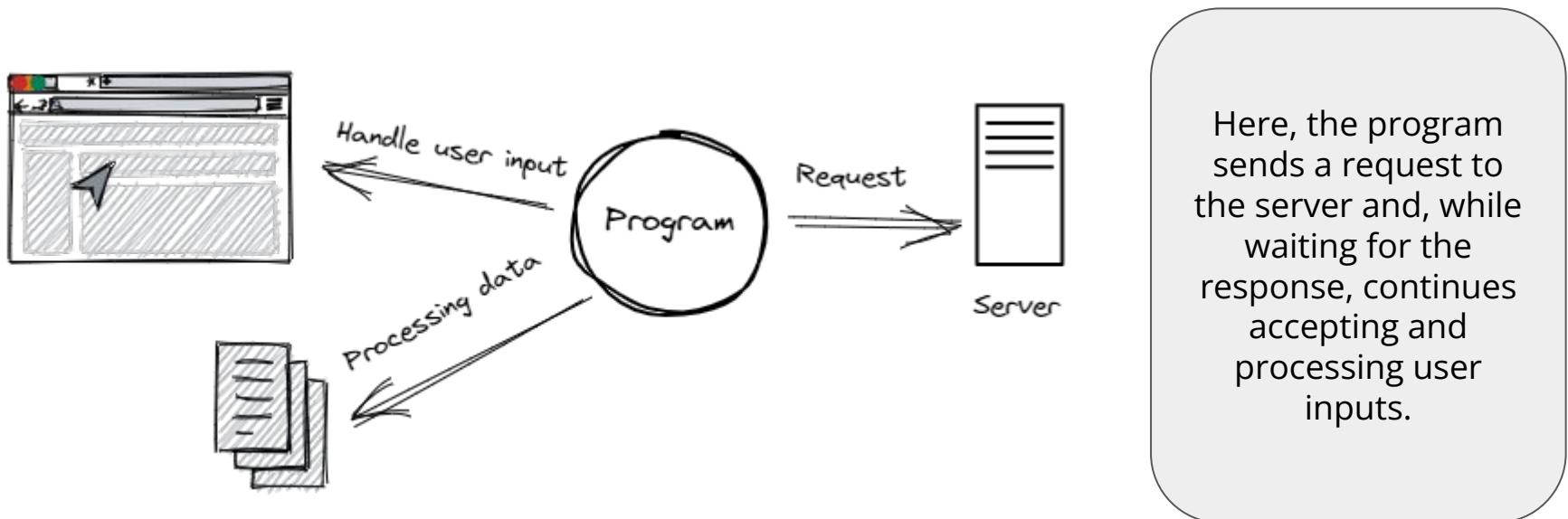
- Quick Revision
 - Asynchronous Programming
 - Callbacks
 - Promises
- async/await
 - Need for async/await
 - Syntax, usage and example
- Error Handling in async code
- Quiz

Quick Revision

Promises and Callbacks

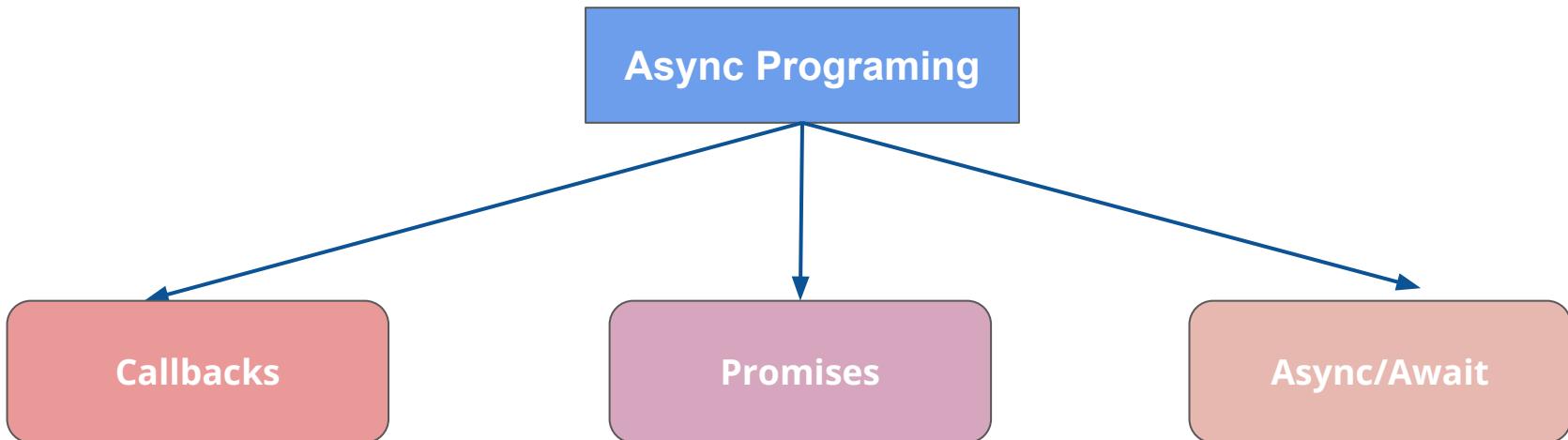
Pre-discussed: Async Programming

Asynchronous programming lets tasks run independently, allowing your program to continue other tasks while waiting for long operations to finish.



Pre-discussed: How to implement?

There are three main ways to implement asynchronous programming in JavaScript:



Pre-discussed: Callbacks

In simple terms, a callback is just a function that you give to another function. The receiving function will then decide when and how to execute.

```
1  function task1(callback) {  
2      console.log("Task 1 completed");  
3      callback();  
4      // Execute the callback after Task 1  
5  }  
6  
7  function task2() {  
8      console.log("Task 2 completed");  
9  }  
10  
11 task1(task2);  
12 // Passing task2 as a callback to task1
```

Here it is upto task1() function when and where it executes callback function. Here it is executing it immediately but that is not always the case.

Pre-discussed: Callbacks

Let's add a delay of 2000 milliseconds before callback executes using setTimeout().

```
1 function task1(callback) {  
2     setTimeout(() => {  
3         console.log("Task 1 completed");  
4         callback();  
5         // Execute the callback once Task 1 is done  
6     }, 2000); // Simulate a delay  
7 }  
8  
9 function task2() {  
10    console.log("Task 2 completed");  
11 }  
12  
13 task1(task2);  
14 // Passing task2 as a callback to task1
```

Here we are adding a delay inside task1 function using setTimeout().

Callbacks: Don't make function `async`

Callbacks don't make functions asynchronous!

```
1  function greet(name, callback) {  
2      console.log("Hello, " + name);  
3      callback();  
4  }  
5  
6  function sayGoodbye() {  
7      console.log("Goodbye!");  
8  }  
9  
10 greet("Alice", sayGoodbye);
```

Output:

Hello, Alice
Goodbye!

Here, `sayGoodbye()` executes immediately after `console.log()`. There's nothing asynchronous happening.

Callbacks: Handle async operations

Let's have a look at end to end comparison of performing async operations without and with callbacks.

```
1 function getUserData() {  
2     fetch("https://jsonplaceholder.typicode.com/users/1")  
3         .then(response => response.json());  
4 }  
5  
6 let user = getUserData();  
7 console.log(user); // ✗ Output: undefined
```

Can you tell why??

Why?



Callbacks: Handle async operations

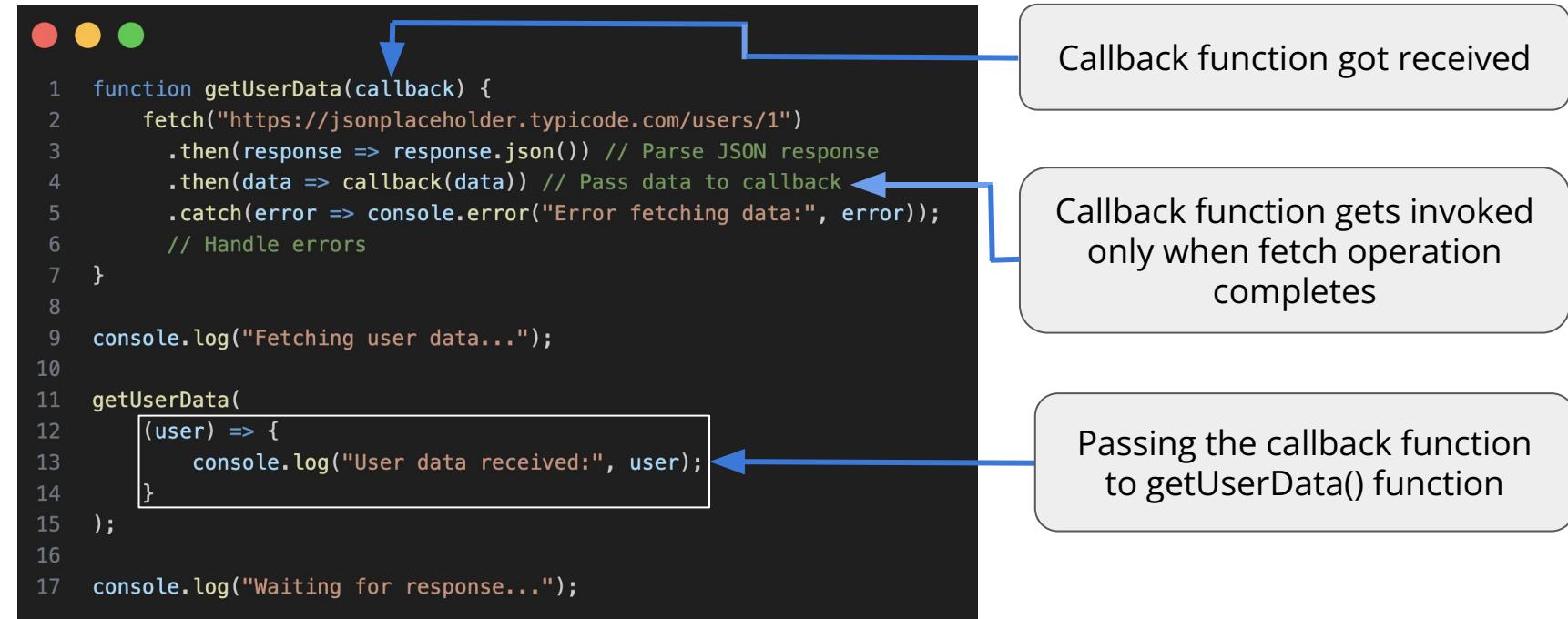
Here `fetch()` does return a value, but since we are not using callbacks, we can't add a task which only needs to happen after `fetch` operations completes.

```
1 function getUserData() {  
2     fetch("https://jsonplaceholder.typicode.com/users/1")  
3         .then(response => response.json());  
4 }  
5  
6 let user = getUserData();  
7 console.log(user); // ✗ Output: undefined
```

We expect to get a value immediately after calling `getUserData()`, but since `fetch()` is asynchronous, the data isn't available yet, so we get `undefined`.

Callbacks: Handle async operations

Instead we need to pass a function to which gets invoked only after `fetch()` completes its execution. And that function is called **Callback Function**.



```
● ● ●
1  function getUserData(callback) {
2    fetch("https://jsonplaceholder.typicode.com/users/1")
3      .then(response => response.json()) // Parse JSON response
4      .then(data => callback(data)) // Pass data to callback
5      .catch(error => console.error("Error fetching data:", error));
6    // Handle errors
7  }
8
9  console.log("Fetching user data...");
10
11 getUserData(
12   (user) => {
13     console.log("User data received:", user);
14   }
15 );
16
17 console.log("Waiting for response...");
```

Callback function got received

Callback function gets invoked only when fetch operation completes

Passing the callback function to `getUserData()` function

Challenge with Callbacks

Callbacks facilitate asynchronous operations, but when multiple operations need to be performed sequentially, they can lead to callback hell.

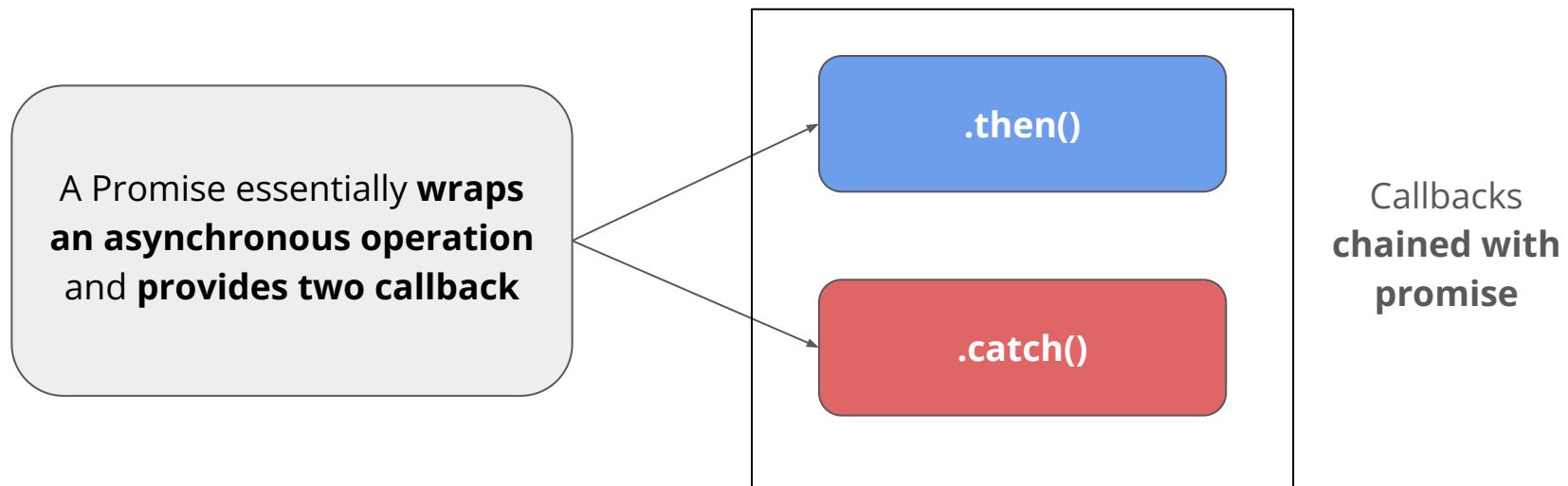
```
3
4 a(function (resultsFromA) {
5     b(resultsFromA, function (resultsFromB) {
6         c(resultsFromB, function (resultsFromC) {
7             d(resultsFromC, function (resultsFromD) {
8                 e(resultsFromD, function (resultsFromE) {
9                     f(resultsFromE, function (resultsFromF) {
10                        console.log(resultsFromF);
11                    })
12                })
13            })
14        })
15    })
16 });
17
```



Callback hell makes code hard to debug and update, leading to poor scalability.

Promises: a more structured way

Promises are built on callbacks, but they offer a more structured and manageable way to handle asynchronous operations compared to using callbacks directly.



Difficulty with promises

Promises are a huge improvement over callbacks but Similar to callback hell, if you nest too many `.then()` blocks, the code can become difficult to manage and read.

```
// Chaining multiple promises in a complex sequence
getUserInfo(1)
  .then(user => {
    return getUserOrders(user.userId);
  })
  .then(orders => {
    return getOrderDetails(orders[0]); // Details for the first order
  })
  .then(orderDetails => {
    return getItemReviews(orderDetails.items[0]); // Reviews for the first item
  })
  .then(reviews => {
    return processReview(reviews[0]); // Process the review for the first item
  })
  .then(processedReview => {
    return getItemReviews(102); // Reviews for the second item in the order
  })
  .then(reviews2 => {
    return processReview(reviews2[0]); // Process the second review
  })
  .then(processedReview2 => {
    return getOrderDetails(orders[1]); // Now for the second order
  })
```

As you keep chaining more `.then()` calls, the code starts getting increasingly indented. This makes it difficult to read, track, and debug.

How to improve code with `async/await`

The solution to this is to use `async/await`, which simplifies the syntax and eliminates the need for deep chaining.

```
1  async function processUserData() {  
2      try {  
3          const user = await getUserInfo(1);  
4          const orders = await getUserOrders(user.userId);  
5          const orderDetails = await getOrderDetails(orders[0]);  
6          const reviews = await getItemReviews(orderDetails.items[0]);  
7          const processedReview = await processReview(reviews[0]);  
8          const reviews2 = await getItemReviews(102);  
9          const processedReview2 = await processReview(reviews2[0]);  
10         const orderDetails2 = await getOrderDetails(orders[1]);  
11     } catch (error) {  
12         console.log('Error:', error);  
13     }  
14 }  
15  
16 processUserData();
```

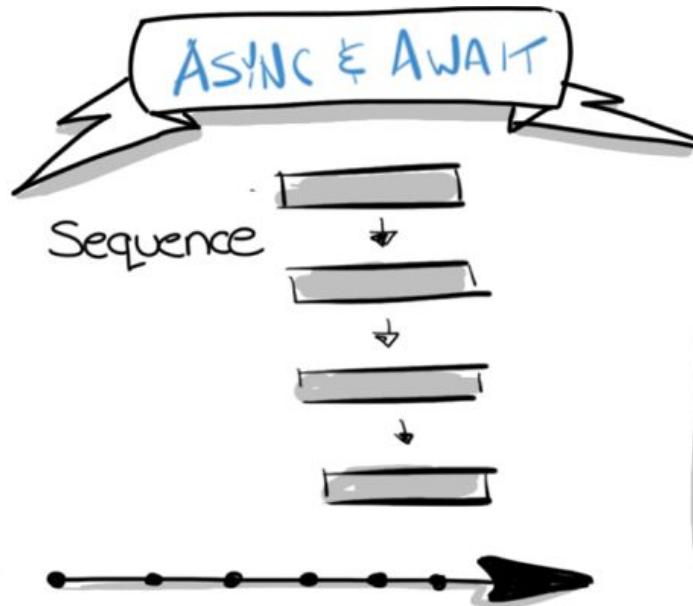
Notice how much easier it is to ***read, update, and debug*** our code now?

async / await

Solution to deep nesting/chaining

Understanding `async` / `await`

`async` and `await` are language features in JavaScript that simplify handling asynchronous operations, making the code look and behave more like synchronous code.



While the `async` function runs asynchronously, the operations inside it execute sequentially.

Basic structure of `async / await`

Async makes a function return a promise, and await makes the function wait for the promise to finish before moving to the next step.

```
async function() {  
    await ...  
}
```

We use the `await` keyword before any task that is asynchronous and takes time to complete, allowing the code to pause and wait for the result.

async / await: Example

Let's understand it with an example:-

Declares a function as asynchronous



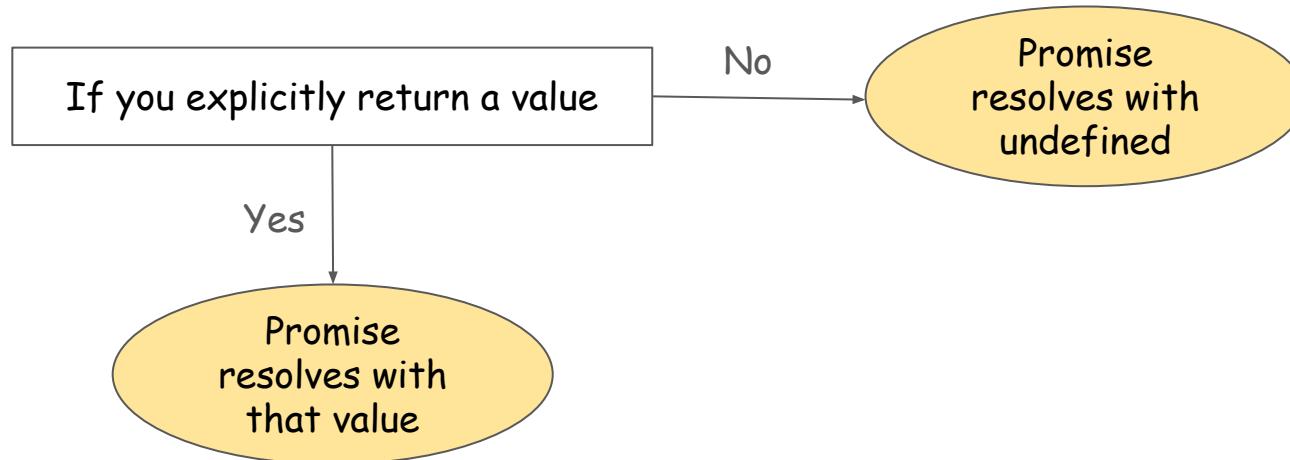
```
1 // Async function to fetch data from a shorter API URL
2 async function getDataFromAPI() {
3     const response = await fetch("https://jsonplaceholder.typicode.com/todos/1");
4     const data = await response.json(); // Parsing the response as JSON
5     console.log(data); // Logging the fetched data
6 }
7
8 getDataFromAPI();
```



Pauses the execution of next statement
until response.json() gets resolved

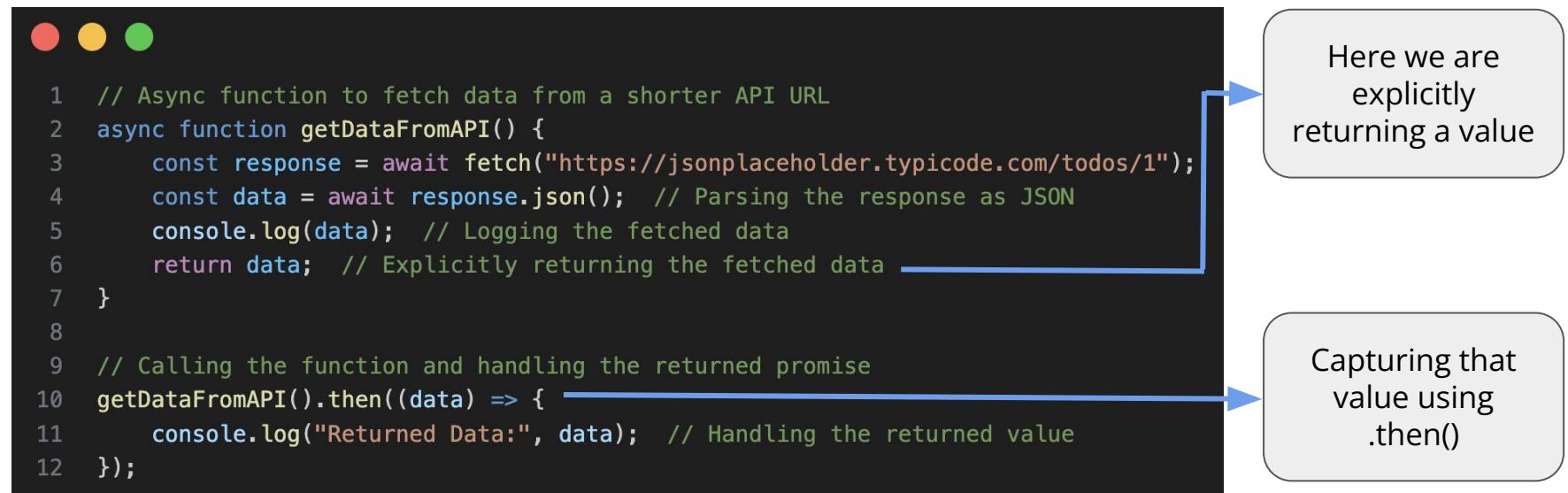
Return values in `async/await`

An `async` function always returns a promise. If you return a value, the promise resolves with it; otherwise, it resolves with `undefined`.



async/await: Explicitly return value

When an `async` function completes execution and reaches a `return` statement, the returned value is automatically wrapped in a Promise.



```
1 // Async function to fetch data from a shorter API URL
2 async function getDataFromAPI() {
3     const response = await fetch("https://jsonplaceholder.typicode.com/todos/1");
4     const data = await response.json(); // Parsing the response as JSON
5     console.log(data); // Logging the fetched data
6     return data; // Explicitly returning the fetched data
7 }
8
9 // Calling the function and handling the returned promise
10 getDataFromAPI().then((data) => {
11     console.log("Returned Data:", data); // Handling the returned value
12 });
```

Here we are explicitly returning a value

Capturing that value using `.then()`

async/await: No return value

If the function does not have an explicit `return`, it implicitly returns `undefined` wrapped in a Promise

```
1 // Async function to fetch data from a shorter API URL
2 async function getDataFromAPI() {
3     const response = await fetch("https://jsonplaceholder.typicode.com/todos/1");
4     const data = await response.json(); // Parsing the response as JSON
5     console.log(data); // Logging the fetched data
6 }
7
8 const apiReturnedVal = getDataFromAPI();
9
10 // Printing value
11 console.log(apiReturnedVal);
12 // Output: undefined
```

Here we got `undefined`
because we didn't returned
anything implicitly

Error Handling

Using `async / await`

Error Handling in `async / await`

When working with `async/await`, errors can occur due to network failures, invalid responses, or unexpected issues.

To handle errors, we use



Use `try/catch` in `async` block



Use `catch` on the returned promise

try / catch: Saves your day

Imagine a nice day, driving your car and suddenly met with an accident. But don't worry airbags are there to catch and save you.



Car/application running smoothly



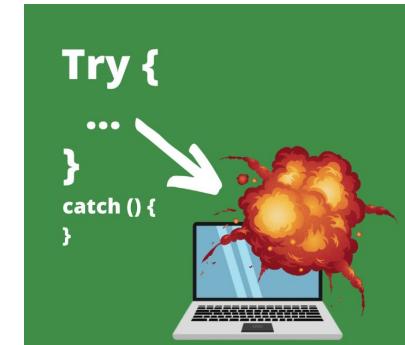
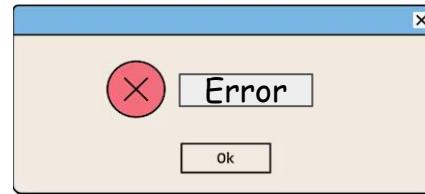
A crash occurred causing an injury/error



Thankfully error got caught not causing much damage

try / catch: Saves your day

Definitely we were not talking about car!!



Web application running smoothly

Error occurred

try...catch caught the error

Error Handling: try / catch

try...catch block in general is used to handle errors. We can use try...catch in the following way:-

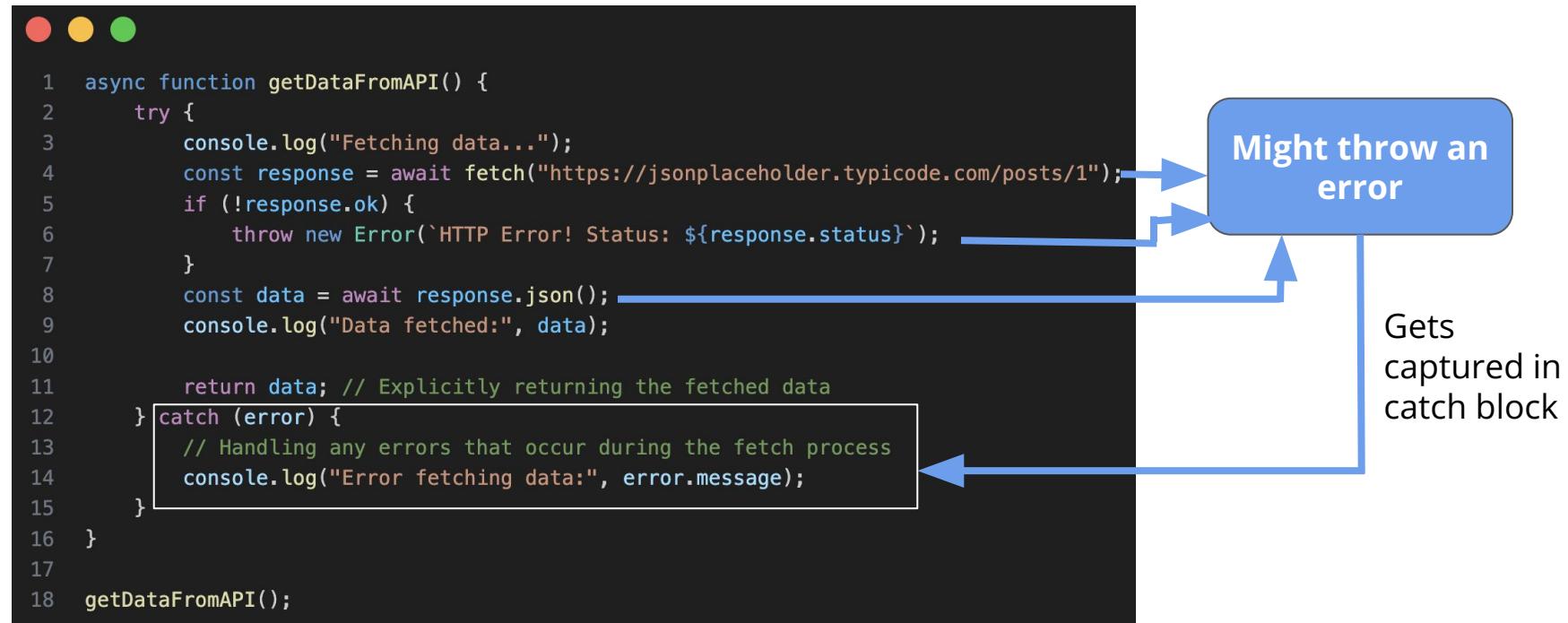
```
1  try {  
2      let result = 10 / 0;  
3      console.log("Result:", result);  
4  
5      let name = undefined;  
6      console.log(name.length); // This will cause an error  
7  } catch (error) {  
8      console.log("Oops! Something went wrong:", error.message);  
9  }
```

Code which might throw an error

Capturing the error

Error Handling: try / catch inside async

We can capture errors in similar fashion inside our async function:-



```
1  async function getDataFromAPI() {  
2      try {  
3          console.log("Fetching data...");  
4          const response = await fetch("https://jsonplaceholder.typicode.com/posts/1");  
5          if (!response.ok) {  
6              throw new Error(`HTTP Error! Status: ${response.status}`);  
7          }  
8          const data = await response.json();  
9          console.log("Data fetched:", data);  
10         return data; // Explicitly returning the fetched data  
11     } catch (error) {  
12         // Handling any errors that occur during the fetch process  
13         console.log("Error fetching data:", error.message);  
14     }  
15 }  
16  
17  
18 getDataFromAPI();
```

Might throw an error

Gets captured in catch block

Error Handling: catch in returned promise

Or else we can attach a catch block in returned promise and capture all the errors there.

```
1  async function getDataFromAPI() {  
2      console.log("Fetching data...");  
3      const response = await fetch("https://jsonplaceholder.typicode.com/posts/1");  
4      if (!response.ok) {  
5          throw new Error(`HTTP Error! Status: ${response.status}`);  
6      }  
7      const data = await response.json();  
8      console.log("Data fetched:", data);  
9  }  
10  
11  getDataFromAPI()  
12    .catch((error) => {  
13        console.log("Error fetching data:", error.message);  
14    });
```

Any error
occured

Gets
captured
here

Quiz

Test Your Understanding!

References

1. **MDN Web Docs - JavaScript:** Comprehensive and beginner-friendly documentation for JavaScript.
<https://developer.mozilla.org/en-US/docs/Web/JavaScript>
2. **Eloquent JavaScript:** A free online book covering JavaScript fundamentals and advanced topics.
<https://eloquentjavascript.net/>
3. **JavaScript.info:** A modern guide with interactive tutorials and examples for JavaScript learners.
<https://javascript.info/>
4. **freeCodeCamp JavaScript Tutorials:** Free interactive lessons and coding challenges to learn JavaScript.
<https://www.freecodecamp.org/learn/>

**Thanks
for
watching!**