JavaScript Array Operations

Web/Mobile Application Development

Franklin Mendez
BSCEN, Engineering Mgmt. MSc

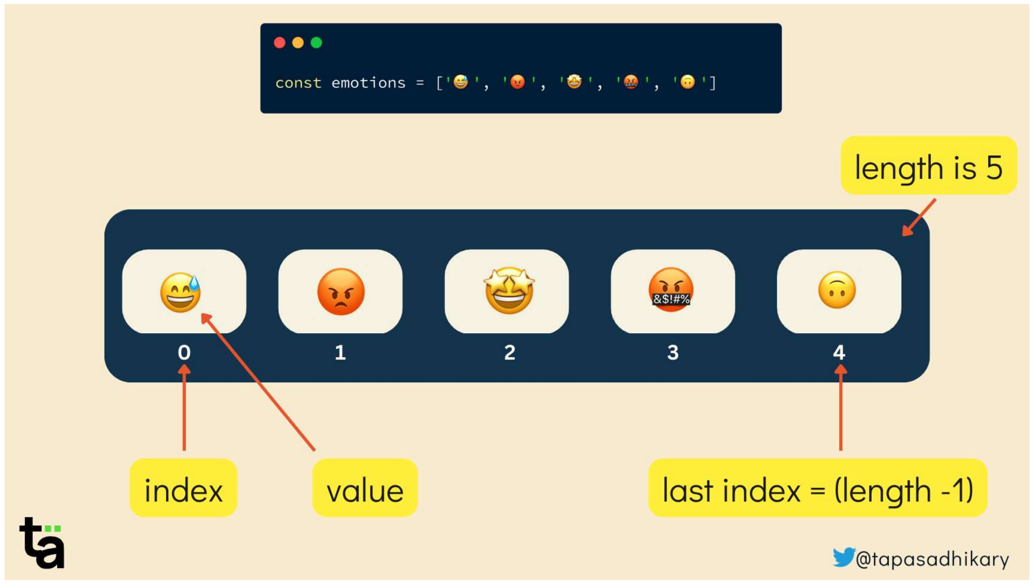
Autumn 2024

Introduction to Arrays

Definition: Arrays are ordered collections of elements, typically of the same data type, accessible by their index.

Syntax: How to declare an array using `[]` (e.g., `let arr = [1, 2, 3];`).

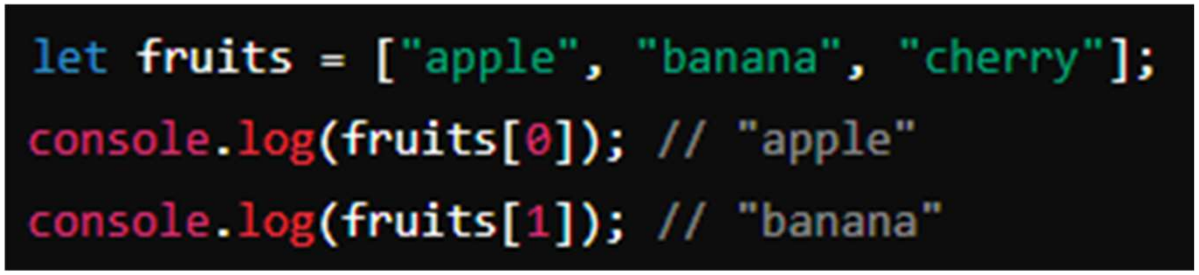
Accessing Elements: Access using `array[index]`, e.g., `arr[0]` for the first element.



```
const emotions = ['😅', '😡', '🤩', '🤬', '🙃']
```

Accessing Elements by Index

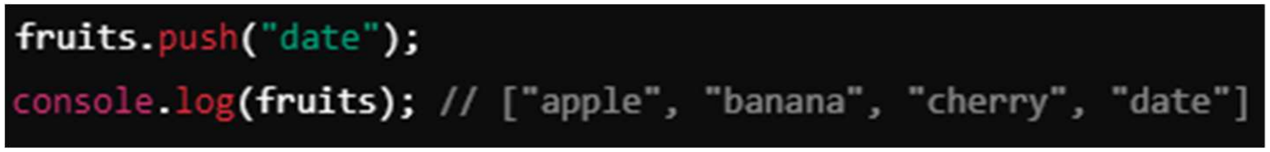
- Arrays can contain any data type, including strings, numbers, objects, or even other arrays (nested arrays).
- Accessing Elements by Index



```
let fruits = ["apple", "banana", "cherry"];
console.log(fruits[0]); // "apple"
console.log(fruits[1]); // "banana"
```

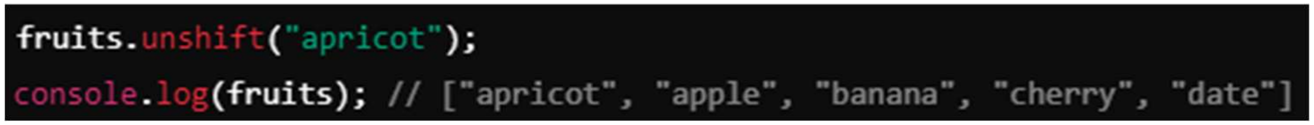
Basic Array Adding Elements

- push(): Adds elements to the end.



```
fruits.push("date");
console.log(fruits); // ["apple", "banana", "cherry", "date"]
```

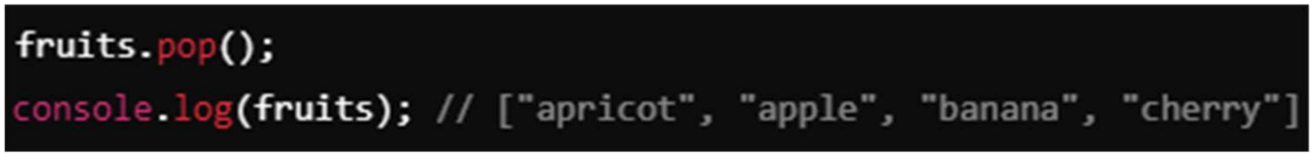
- unshift(): Adds elements to the beginning.



```
fruits.unshift("apricot");
console.log(fruits); // ["apricot", "apple", "banana", "cherry", "date"]
```

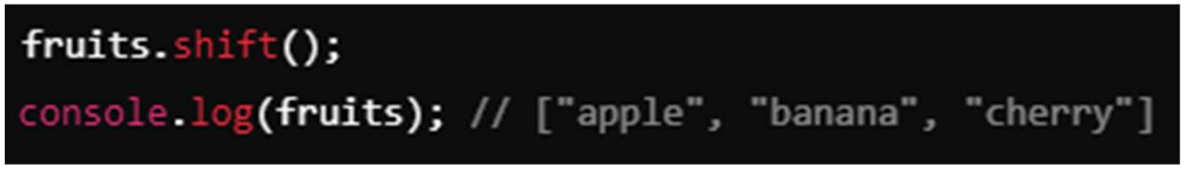
Basic Array Removing Elements

- pop(): Removes the last element.



```
fruits.pop();
console.log(fruits); // ["apricot", "apple", "banana", "cherry"]
```

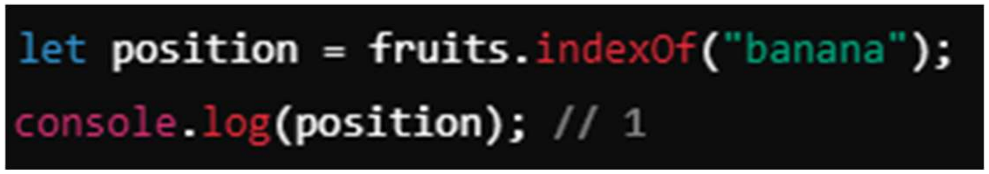
- shift(): Removes the first element.



```
fruits.shift();
console.log(fruits); // ["apple", "banana", "cherry"]
```

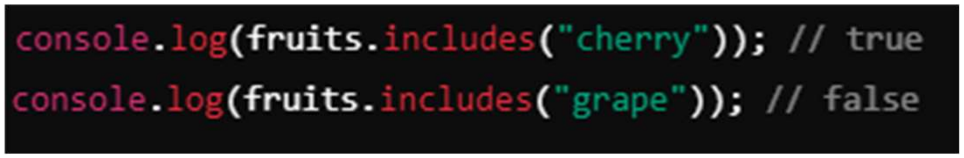
Basic Array Finding Elements

- indexOf():



```
let position = fruits.indexOf("banana");
console.log(position); // 1
```

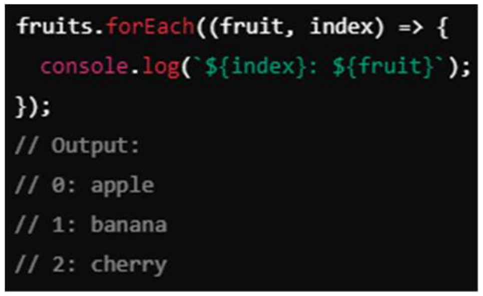
- Includes():



```
console.log(fruits.includes("cherry")); // true
console.log(fruits.includes("grape")); // false
```

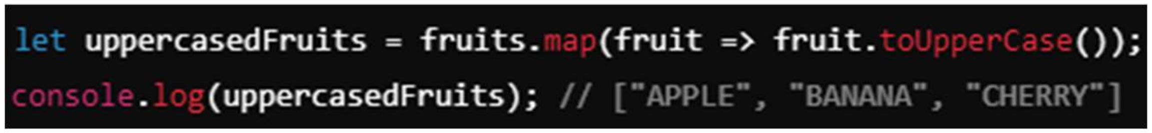
Basic Array Looping and Iteration

- ForEach():



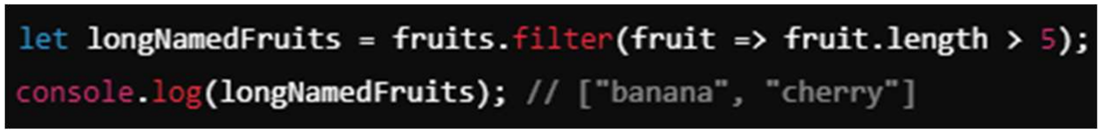
```
fruits.forEach((fruit, index) => {
  console.log(`${index}: ${fruit}`);
});
// Output:
// 0: apple
// 1: banana
// 2: cherry
```

- Map(): Creates a new array by transforming each element.



```
let uppercasedFruits = fruits.map(fruit => fruit.toUpperCase());
console.log(uppercasedFruits); // ["APPLE", "BANANA", "CHERRY"]
```

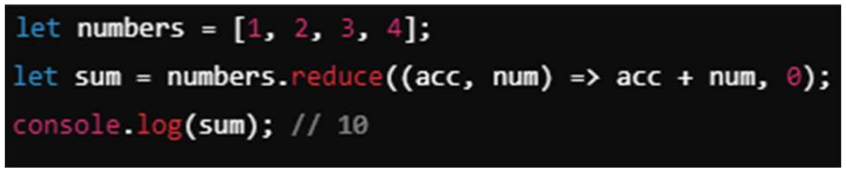
- Filter(): Filters elements based on a condition.



```
let longNamedFruits = fruits.filter(fruit => fruit.length > 5);
console.log(longNamedFruits); // ["banana", "cherry"]
```

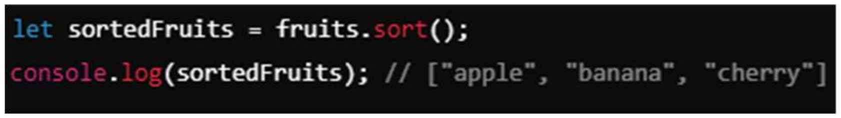
Transforming Arrays with reduce()

- reduce(): Reduces the array to a single value by applying a function.



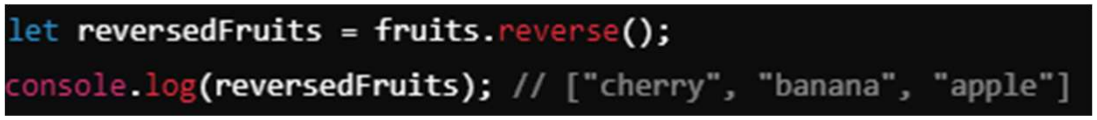
```
let numbers = [1, 2, 3, 4];
let sum = numbers.reduce((acc, num) => acc + num, 0);
console.log(sum); // 10
```

- sort(): Sorts an array.



```
let sortedFruits = fruits.sort();
console.log(sortedFruits); // ["apple", "banana", "cherry"]
```

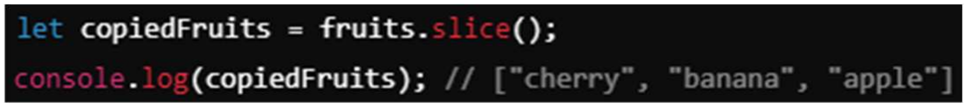
- reverse(): Reverses the array's order.



```
let reversedFruits = fruits.reverse();
console.log(reversedFruits); // ["cherry", "banana", "apple"]
```

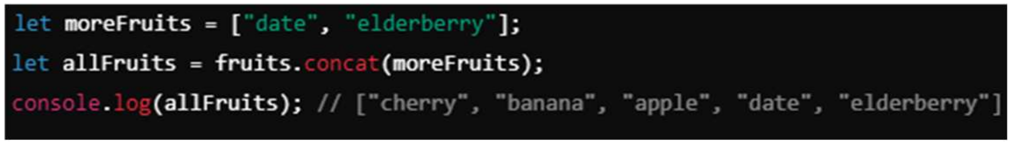
Copying and Combining Arrays

- slice(): Creates a shallow copy of an array.



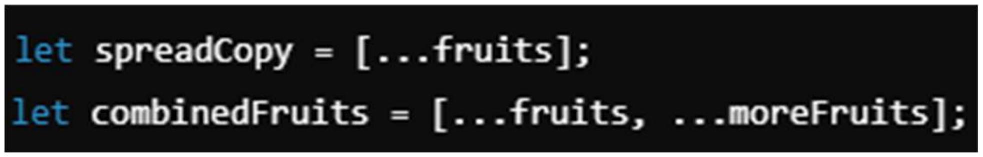
```javascript
let copiedFruits = fruits.slice();
console.log(copiedFruits); // ["cherry", "banana", "apple"]
```

- concat():



```javascript
let moreFruits = ["date", "elderberry"];
let allFruits = fruits.concat(moreFruits);
console.log(allFruits); // ["cherry", "banana", "apple", "date", "elderberry"]
```

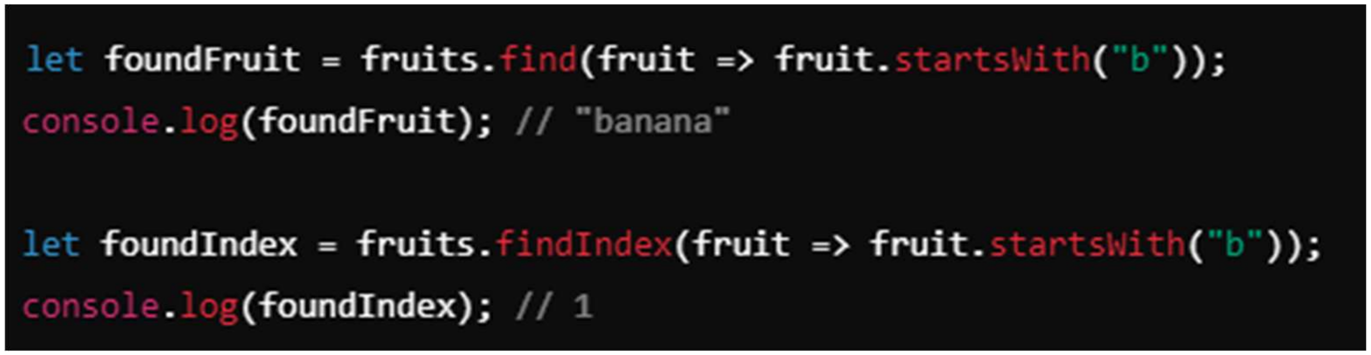
- Spread Operator **...** :



```javascript
let spreadCopy = [...fruits];
let combinedFruits = [...fruits, ...moreFruits];
```

Advanced Array Methods

- find() and findIndex():



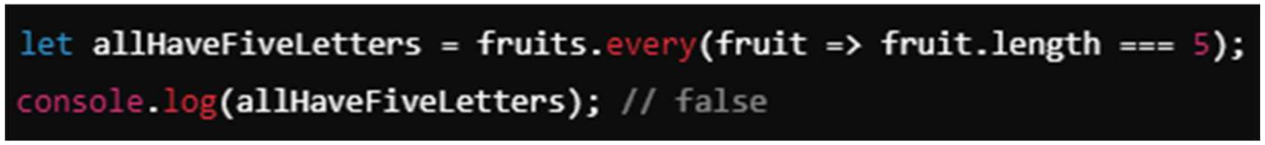
```
let foundFruit = fruits.find(fruit => fruit.startsWith("b"));
console.log(foundFruit); // "banana"

let foundIndex = fruits.findIndex(fruit => fruit.startsWith("b"));
console.log(foundIndex); // 1
```

Advanced Array Methods(2)

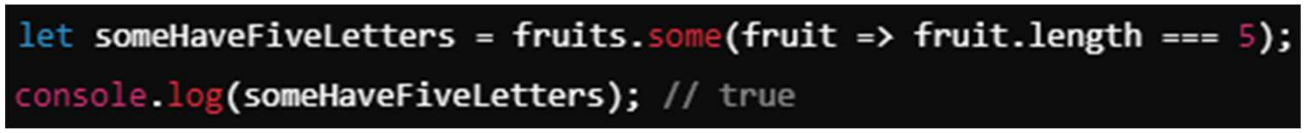
every() and some():

- every() checks if all elements meet a condition



```
let allHaveFiveLetters = fruits.every(fruit => fruit.length === 5);
console.log(allHaveFiveLetters); // false
```

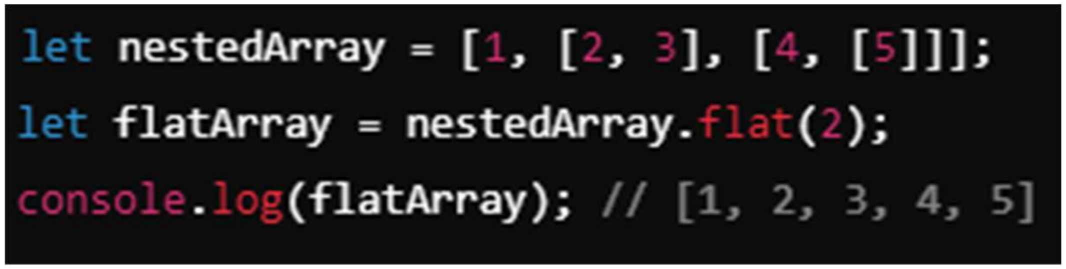
- **some()** checks if at least one element meets a condition:



```
let someHaveFiveLetters = fruits.some(fruit => fruit.length === 5);
console.log(someHaveFiveLetters); // true
```

Advanced Array Methods(3)

- **flat()**: Flattens nested arrays into a single array



```
let nestedArray = [1, [2, 3], [4, [5]]];
let flatArray = nestedArray.flat(2);
console.log(flatArray); // [1, 2, 3, 4, 5]
```

JavaScript Loops Overview

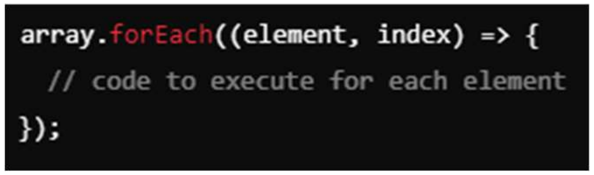
Loops allow you to repeat actions multiple times and are fundamental for working with arrays and lists.

Common types:

- **forEach**: Best for iterating over array elements.
- **for**: Traditional loop, often used when you need a loop counter.
- **while**: Runs as long as a specified condition is true.

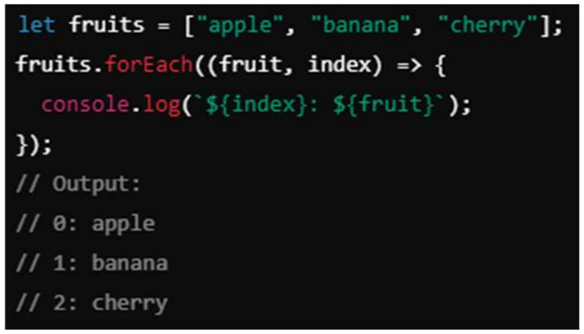
forEach Loop

The forEach method iterates over each element in an array, calling a function for each element.



```
array.forEach((element, index) => {
  // code to execute for each element
});
```

Example:



```
let fruits = ["apple", "banana", "cherry"];
fruits.forEach((fruit, index) => {
  console.log(`${index}: ${fruit}`);
});
// Output:
// 0: apple
// 1: banana
// 2: cherry
```

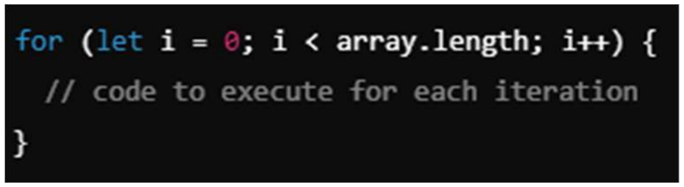
Pros: Cleaner and ideal for working with arrays. Automatically stops at the end of the array.

Cons: Cannot use break or continue within forEach to alter the loop flow directly.

for Loop

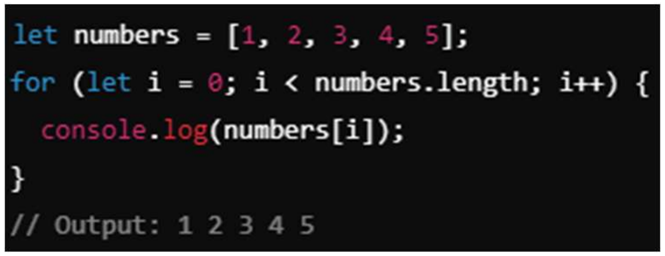
The for loop is more flexible and allows you to control the loop counter, start, stop, and step conditions.

- Syntax:



```
for (let i = 0; i < array.length; i++) {
  // code to execute for each iteration
}
```

Example:



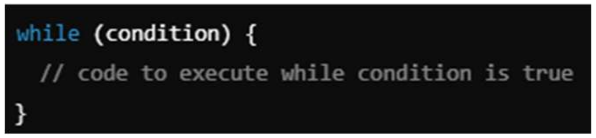
```
let numbers = [1, 2, 3, 4, 5];
for (let i = 0; i < numbers.length; i++) {
  console.log(numbers[i]);
}
// Output: 1 2 3 4 5
```

Pros: Flexible and allows breaking out of the loop with break or continue.

Cons: Slightly more verbose; easy to make mistakes with the loop conditions.

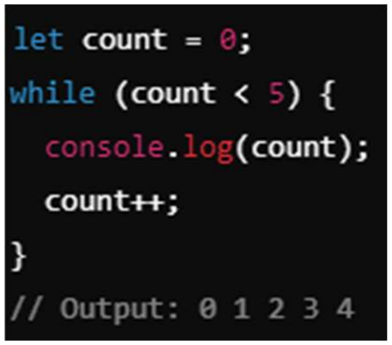
while Loop

- The while loop runs as long as a specified condition is true.



```
while (condition) {
  // code to execute while condition is true
}
```

Example:



```
let count = 0;
while (count < 5) {
  console.log(count);
  count++;
}
// Output: 0 1 2 3 4
```

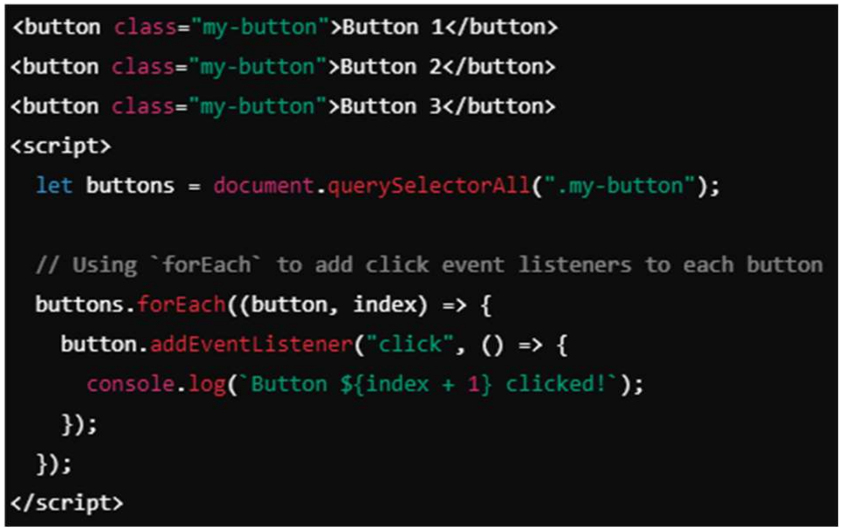
Pros: Ideal for loops with uncertain end conditions.

Cons: Risk of infinite loops if the condition never becomes false.

Events in Loops

- Problem: Directly adding event listeners in a loop can cause issues, especially if using var in for loops (due to scoping).
- Solution: Use let in for loops or pass the loop variable into a function to handle each event separately.

Example with Button Click Events in a Loop:



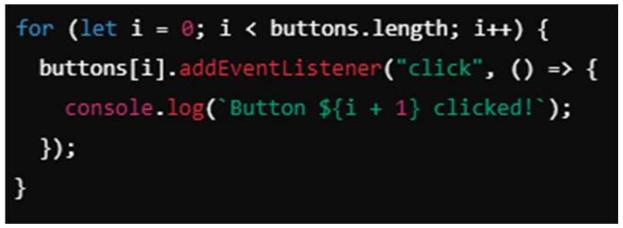
```
<button class="my-button">Button 1</button>
<button class="my-button">Button 2</button>
<button class="my-button">Button 3</button>
<script>
  let buttons = document.querySelectorAll(".my-button");

  // Using `forEach` to add click event listeners to each button
  buttons.forEach((button, index) => {
    button.addEventListener("click", () => {
      console.log(`Button ${index + 1} clicked!`);
    });
  });
</script>
```

This method ensures each button has its own event listener, and the index is correctly scoped within each event.

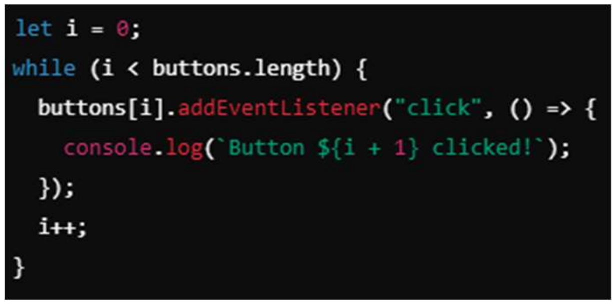
Using for and while with Events

for loop with Events:



```
for (let i = 0; i < buttons.length; i++) {
  buttons[i].addEventListener("click", () => {
    console.log(`Button ${i + 1} clicked!`);
  });
}
```

while loop with Events:



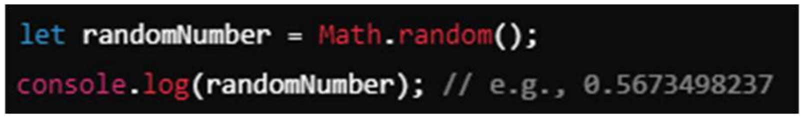
```
let i = 0;
while (i < buttons.length) {
  buttons[i].addEventListener("click", () => {
    console.log(`Button ${i + 1} clicked!`);
  });
  i++;
}
```

Best Practices

- Use let or const: Helps to avoid scoping issues in loops.
- Avoid Modifying Array Length: Modifying the array inside a loop (e.g., removing items) can cause unexpected behavior.
- Optimize for Performance: For large arrays or complex operations, for may be faster than forEach.

The Math.random() Function for Generating Random Numbers

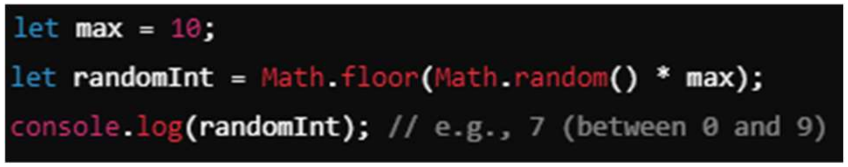
- Definition: Math.random() generates a random floating-point number between 0 (inclusive) and 1 (exclusive).
- Basic Example:



```javascript
let randomNumber = Math.random();
console.log(randomNumber); // e.g., 0.5673498237
```

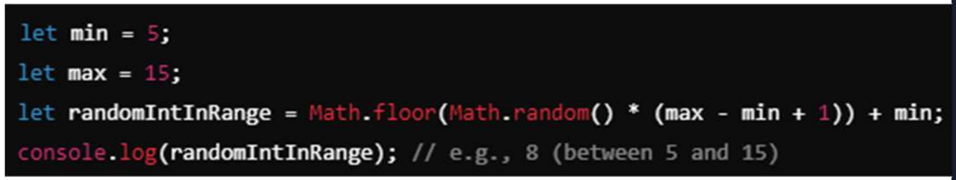
Generating Random Integers:

Between 0 and a Max Value



```javascript
let max = 10;
let randomInt = Math.floor(Math.random() * max);
console.log(randomInt); // e.g., 7 (between 0 and 9)
```

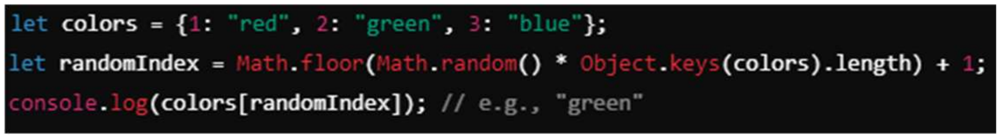
Between a Min and Max Range:



```javascript
let min = 5;
let max = 15;
let randomIntInRange = Math.floor(Math.random() * (max - min + 1)) + min;
console.log(randomIntInRange); // e.g., 8 (between 5 and 15)
```

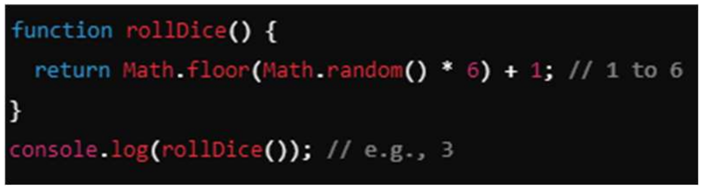
Practical Applications of Math.random() with Objects

- Randomly Selecting a Property



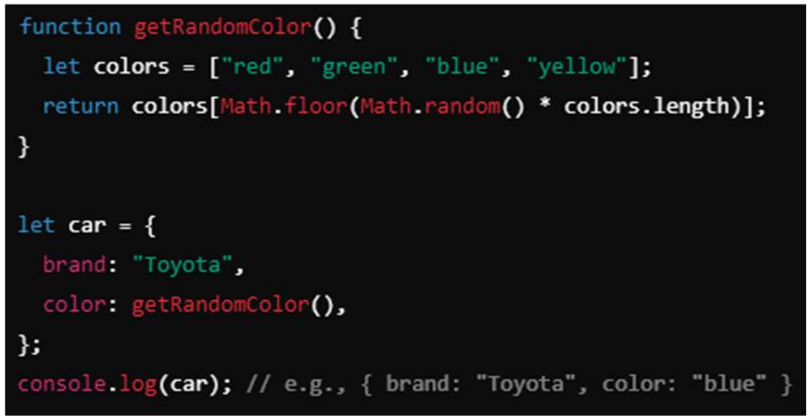
```
let colors = {1: "red", 2: "green", 3: "blue"};
let randomIndex = Math.floor(Math.random() * Object.keys(colors).length) + 1;
console.log(colors[randomIndex]); // e.g., "green"
```

- Simulating Dice Rolls:



```
function rollDice() {
  return Math.floor(Math.random() * 6) + 1; // 1 to 6
}
console.log(rollDice()); // e.g., 3
```

Creating Random Objects:



```
function getRandomColor() {
  let colors = ["red", "green", "blue", "yellow"];
  return colors[Math.floor(Math.random() * colors.length)];
}

let car = {
  brand: "Toyota",
  color: getRandomColor(),
};
console.log(car); // e.g., { brand: "Toyota", color: "blue" }
```