JavaScript Lecture Notes

The `this` Keyword in JavaScript Objects

The `this` keyword in JavaScript refers to the object that is executing the current function. Its behavior depends on how a function is called:

- Inside object methods, `this` refers to the object the method belongs to
- When used alone, `this` refers to the global object (window in browsers)
- In event handlers, `this` refers to the element that received the event

Example:

```
const newObjectEC = {
  ...objectEc,
  getEmail: function() {
    console.log({ thisValue: this }); // 'this' refers to newObjectEC
    return `${this.full_name}@gmail.com`;
  }
};

newObjectEC.getEmail(); // Correctly accesses this.full_name from the object
```

Types of Functions in JavaScript

JavaScript supports several ways to define functions:

1. **Function declarations**:

   ```
   function myFunction(parameters) {
     // code
   }
   ```

2. **Function expressions**:

   ```
   const myFunction = function(parameters) {
     // code
   };
   ```

3. **Arrow functions**:

```
const myFunction = (parameters) => {
  // code
};
```

Note: Arrow functions don't have their own `this` binding - they inherit `this` from the parent scope

4. **Immediately Invoked Function Expressions (IIFE)**:

```
(function() {
  // code that executes immediately
  console.log("This runs right away!");
})();

// With parameters
(function(name) {
  console.log(`Hello, ${name}!`);
})("JavaScript");

// Arrow function IIFE
(() => {
  console.log("Arrow function IIFE");
})();
```

Benefits of IIFEs:

- Creates a private scope for variables
- Avoids polluting the global namespace
- Executes code immediately without needing a separate function call
- Useful for initialization code or creating modules

5. **Method definition in objects**:

```
const obj = {
  myMethod() {
    // code
  },
  myMethod2: function() {
    // code
  }
};
```

Spread Operator vs Rest Operator

Both use the same syntax (`...`) but serve different purposes:

Spread Operator

- **Expands** an iterable (array, object, string) into individual elements
- Used to create copies or merge iterables

Array Example:

```
let arr = [1, 2, 3, 4, 5, 6];
arr = [...arr, 112]; // Adds 112 to the end: [1, 2, 3, 4, 5, 6, 112]
```

Object Example:

```
const objectEc = {
  first_name: "houssam",
  last_name: "largou",
  full_name: "houssam.largou",
  gender: "female",
};

const newObjectEC = {
  ...objectEc,  // Copies all properties from objectEc
  email: "example@gmail.com"  // Adds a new property
};
```

Rest Operator

- **Collects** multiple elements into a single array
- Used in function parameters to handle variable numbers of arguments

Example:

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

sum(1, 2, 3, 4); // 10
```

Common Array Methods

Method	Description	Mutates Original?	Example
`push()`	Adds element(s) to the end	Yes	`arr.push(7)`
`pop()`	Removes last element and returns it	Yes	`arr.pop()`
`shift()`	Removes first element and returns it	Yes	`arr.shift()`

`unshift()`	Adds element(s) to the beginning	Yes	`arr.unshift(213)`
`splice()`	Changes array by removing/replacing elements	Yes	`arr.splice(2, 1, 'new')`
`concat()`	Combines arrays, returns new array	No	`arr.concat([8, 9])`
`slice()`	Returns portion of array as new array	No	`arr.slice(1, 3)`
`map()`	Creates new array with results of callback	No	`arr.map(x => x * 2)`
`filter()`	Creates new array with elements passing test	No	`arr.filter(x => x > 3)`
`reduce()`	Reduces array to single value	No	`arr.reduce((a, b) => a + b)`
`forEach()`	Executes callback for each element	No	`arr.forEach(x => console.log(x))`
`find()`	Returns first element that passes test	No	`arr.find(x => x > 3)`
`some()`	Tests if at least one element passes test	No	`arr.some(x => x > 3)`
`every()`	Tests if all elements pass test	No	`arr.every(x => x > 0)`

Object Manipulation

Creating Objects

```
const objectEc = {
  first_name: "houssam",
  last_name: "largou",
  full_name: "houssam.largou",
  gender: "female",
};
```

Cloning and Modifying Objects

```
// Create clone with additional properties
const newObjectEC = {
  ...objectEc,
  getEmail: function() {
    return `${this.full_name}@gmail.com`;
  },
  oldObject: objectEc  // Reference to original object
};

// Delete properties
delete newObjectEC.first_name;
delete newObjectEC.last_name;

// Add properties to existing object
objectEc.newProperty = "property";
```

Key Takeaways

- The spread operator (`...`) creates a shallow copy
- Modifying the original object after copying will not affect the clone
- Adding properties to the original object after copying will not add them to the clone
- Nested objects are still referenced (not deeply cloned)