Functions in JavaScript

Week 3

Adib Sakhawat
IUT CSE ‘21

Functions are one of the core components of JavaScript, allowing you to encapsulate code for reuse and organization. They can be declared in various ways, and understanding their nuances is crucial for effective programming.

Function Declarations

A function declaration defines a named function using the `function` keyword.

```
function greet(name) {
  return `Hello, ${name}!`;
}
```

Usage:

```
console.log(greet('Alice')); // Output: Hello, Alice!
```

Edge Case: Hoisting

Function declarations are hoisted, meaning they are moved to the top of their scope at compile time.

```
console.log(greet('Bob')); // Output: Hello, Bob!

function greet(name) {
  return `Hello, ${name}!`;
}
```

Function Expressions

A function expression defines a function as part of an expression. It can be named or anonymous.

```
const greet = function(name) {
  return `Hello, ${name}!`;
};
```

Usage:

```
console.log(greet('Charlie')); // Output: Hello, Charlie!
```

Edge Case: Hoisting

Function expressions are **not** hoisted. Using them before declaration results in an error.

```
console.log(greet('Dave')); // Error: greet is not defined

const greet = function(name) {
  return `Hello, ${name}!`;
};
```

Arrow Functions (`=>` Operation)

Arrow functions provide a shorter syntax for writing functions and lexically bind the `this` value.

```
const greet = (name) => {
  return `Hello, ${name}!`;
};
```

Simplified Syntax:

- **Single Parameter (no parentheses needed):**

  ```
  const greet = name => {
    return `Hello, ${name}!`;
  };
  ```

- **Implicit Return (no braces or `return` keyword):**

```
const greet = name => `Hello, ${name}!`;
```

Usage:

```
console.log(greet('Eve')); // Output: Hello, Eve!
```

Edge Cases with Arrow Functions

1. `this` Binding

Arrow functions do not have their own `this` context; they inherit it from the enclosing scope.

Example:

```
function Counter() {
  this.count = 0;
  setInterval(() => {
    this.count++;
    console.log(this.count);
  }, 1000);
}

const counter = new Counter();
```

Edge Case:

Using a regular function would result in `this` being `undefined` (in strict mode) or pointing to the global object.

```
function Counter() {
  this.count = 0;
  setInterval(function() {
    this.count++;
    console.log(this.count);
  }, 1000);
}
```

```
const counter = new Counter(); // NaN, because `this.count` is undefined
```

2. No `arguments` Object

Arrow functions do not have an `arguments` object.

Solution: Use Rest Parameters

```
const sum = (...args) => args.reduce((total, num) => total + num, 0);

console.log(sum(1, 2, 3)); // Output: 6
```

3. Cannot be Used as Constructors

Arrow functions cannot be used with the `new` keyword.

```
const Person = (name) => {
  this.name = name;
};

const person = new Person('Frank'); // Error: Person is not a constructor
```

4. No `prototype` Property

Arrow functions do not have a `prototype` property.

The `>` Operation

In JavaScript, the `->` operator does not exist. This operator is commonly associated with other languages:

- **PHP:** Used for accessing object properties/methods.
- **CoffeeScript:** Used for function definitions, which compile to JavaScript functions.

Note:

If you're seeing `->` in JavaScript code, it's likely either a typo or code from a transpiled language like CoffeeScript.

Lambdas (Anonymous Functions)

In JavaScript, lambdas refer to anonymous functions, which can be defined without a name and assigned to variables or passed as arguments.

Function Expression (Anonymous):

```
const greet = function(name) {
  return `Hello, ${name}!`;
};
```

Arrow Function (Anonymous):

```
const greet = name => `Hello, ${name}!`;
```

Usage in Higher-Order Functions:

```
const numbers = [1, 2, 3];

const squares = numbers.map(function(num) {
  return num * num;
});

console.log(squares); // Output: [1, 4, 9]
```

Using Arrow Functions:

```
const squares = numbers.map(num => num * num);
```

Parameter Passing

Understanding how parameters are passed to functions is vital for controlling data flow and avoiding unintended side effects.

Positional Parameters

Arguments are matched to parameters based on their order.

```
function createPoint(x, y) {
  return { x, y };
}

console.log(createPoint(2, 3)); // Output: { x: 2, y: 3 }
```

Default Parameters

Allows parameters to have default values if no argument is provided.

```
function greet(name = 'Guest') {
  return `Hello, ${name}!`;
}

console.log(greet()); // Output: Hello, Guest!
```

Edge Case: Falsy Values

Be cautious with falsy values like `0` or `''`.

```
function greet(name = 'Guest') {
  return `Hello, ${name}!`;
}

console.log(greet('')); // Output: Hello, !
```

Rest Parameters

Collects all remaining arguments into an array.

```
function multiply(factor, ...numbers) {
  return numbers.map(num => num * factor);
}

console.log(multiply(2, 1, 2, 3)); // Output: [2, 4, 6]
```

Parameter Destructuring

Allows unpacking values from arrays or properties from objects.

```
function display({ name, age }) {
  console.log(`Name: ${name}, Age: ${age}`);
}

const person = { name: 'Grace', age: 30 };

display(person); // Output: Name: Grace, Age: 30
```

Passing with Parameter Name

JavaScript does not support named parameters in function calls like some other languages (e.g., Python). However, you can simulate this behavior using objects.

Using Object Arguments:

```
function configure({ width = 100, height = 100, color = 'bl
ue' } = {}) {
  console.log(`Width: ${width}, Height: ${height}, Color:
${color}`);
}

configure({ height: 200, color: 'red' });
// Output: Width: 100, Height: 200, Color: red
```

Edge Case: Missing Argument Object

Ensure to provide a default empty object to avoid errors when no argument is passed.

```
function configure({ width = 100, height = 100 } = {}) {
  // Function body
}
```

Pass by Value and Pass by Reference

Understanding how values are passed in JavaScript functions is key to predicting how changes within functions affect variables outside them.

Pass by Value

Primitive types (`string`, `number`, `boolean`, `null`, `undefined`, `symbol`, `bigint`) are passed by value.

Example:

```
let num = 10;

function increment(value) {
  value++;
  console.log(`Inside function: ${value}`); // Output: 11
}

increment(num);
console.log(`Outside function: ${num}`); // Output: 10
```

Edge Case: Immutable Primitives

Primitives are immutable; any operation inside the function does not affect the original variable.

Pass by Reference (Technically Pass by Value of Reference)

Objects, arrays, and functions are passed by reference. The reference to the object is passed by value.

Example:

```
let arr = [1, 2, 3];

function addElement(array) {
  array.push(4);
  console.log(`Inside function: ${array}`); // Output: [1,
2, 3, 4]
}

addElement(arr);
console.log(`Outside function: ${arr}`); // Output: [1, 2,
3, 4]
```

Edge Case: Reassigning the Parameter

Reassigning the parameter to a new object does not affect the original reference.

```
function replaceArray(array) {
  array = [4, 5, 6];
  console.log(`Inside function: ${array}`); // Output: [4,
5, 6]
}

replaceArray(arr);
console.log(`Outside function: ${arr}`); // Output: [1, 2,
3, 4]
```

Handling Edge Cases

Mutating vs. Reassigning

- **Mutating an Object:** Changes affect the original object.

```
function mutate(obj) {
  obj.prop = 'changed';
}

const originalObj = { prop: 'original' };
mutate(originalObj);
console.log(originalObj.prop); // Output: changed
```

- **Reassigning an Object:** Does not affect the original object.

```
function reassign(obj) {
  obj = { prop: 'new' };
}

reassign(originalObj);
console.log(originalObj.prop); // Output: changed
```

Passing Functions as Parameters

Functions can be passed and invoked inside other functions.

```
function operate(a, b, operation) {
  return operation(a, b);
}

const sum = (x, y) => x + y;

console.log(operate(5, 3, sum)); // Output: 8
```

Edge Case: Callback Context

Ensure the correct `this` context when passing methods as callbacks.

```
const calculator = {
  value: 0,
  add(a) {
    this.value += a;
  }
};

function execute(fn) {
  fn(5);
}

execute(calculator.add); // Error: Cannot read property 'va
lue' of undefined
```

Solution: Bind the Context

```
execute(calculator.add.bind(calculator));
```