JavaScript Concepts: Parameter Validation, Recursion, Functions, Callbacks, setTimeout, setInterval, and Arrow Functions

1. Parameter Validation (5.1.1)

Parameter validation ensures that function arguments meet expected types and requirements, preventing errors and improving function predictability.

Benefits of Parameter Validation

- **Robustness:** Prevents unexpected behavior.
- **Security:** Reduces the chance of errors and vulnerabilities.
- **Maintainability:** Helps in early error detection.

Example – Basic Type Checking:

```
function add(a, b) {
  if (typeof a !== 'number' || typeof b !== 'number') {
    throw new Error('Both arguments must be numbers.');
  }
  return a + b;
}

try {
  console.log(add(2, '3')); // Error: Both arguments must be numbers.
} catch (error) {
  console.error(error.message);
}
```

Example – Type and Range Validation:

```
function calculatePercentage(value, total) {
  if (typeof value !== 'number' || typeof total !== 'number') {
    throw new Error('Both arguments must be numbers.');
  }
  if (value < 0 || total <= 0) {
    throw new Error('Value must be positive and total must be greater than 0.');
  }
  return (value / total) * 100;
}

try {
  console.log(calculatePercentage(50, 200)); // 25
  console.log(calculatePercentage(-5, 100)); // Error: Value must be positive.
} catch (error) {
  console.error(error.message);
}
```

Exercises:

1. Implement `isValidEmail(email)`, which checks if the input is a valid email address.
2. Create `calculateBMI(weight, height)` to ensure inputs are positive numbers and calculate BMI.

2. Recursion (5.1.2)

Recursion is a method where a function calls itself to solve a problem. A **base case** is crucial to prevent infinite recursion.

Characteristics and Advantages

- **Efficiency:** Solves complex problems elegantly.
- **Elegance:** Reduces the need for loops and saves code.
- **Use Cases:** Useful for tree traversal and mathematical problems.

Example – Factorial Calculation:

```
function factorial(n) {
  if (n <= 1) return 1; // Base case
  return n * factorial(n - 1);
}

console.log(factorial(5)); // 120
```

Example – Fibonacci Sequence:

```
function fibonacci(n) {
  if (n <= 1) return n; // Base case
  return fibonacci(n - 1) + fibonacci(n - 2);
}

console.log(fibonacci(6)); // 8
```

Exercises:

1. Implement `sumTo(n)` that calculates the sum of numbers from 1 to `n` recursively.
2. Write a recursive function to calculate the depth of a nested array, e.g., `[1, [2, [3]]]`.

3. Functions as First-Class Objects (5.1.3)

In JavaScript, functions are **first-class objects**, meaning they can be stored, passed as arguments, and returned from other functions.

Characteristics

- **Functionality:** Functions can be stored in variables.
- **Flexibility:** They can be passed as arguments to other functions.
- **Reusability:** Functions can be used anywhere in the code.

Example – Functions as Arguments:

```
const greet = function(name) {
  return "Hello " + name + "!";
};

const processGreeting = function(greetFunction, name) {
  return greetFunction(name);
};

console.log(processGreeting(greet, 'Alice')); // "Hello Alice!"
```

Exercises:

1. Write `calculate` that accepts two numbers and a callback function, returning the result.
2. Create `generateGreeting` that produces different greetings based on the time of day.

4. Function Expressions (5.1.4)

A function expression is a function assigned to a variable. Function expressions can be anonymous and are only executable once defined.

Characteristics

- **Anonymity:** Function expressions are often anonymous, adding flexibility.
- **Scoping:** They are only available within the current scope.
- **Utility:** Ideal for inline functions or callbacks.

Example – Anonymous Function Expression:

```
const square = function(num) {
  return num * num;
};

console.log(square(4)); // 16
```

Example – Used as a Callback:

```
setTimeout(function() {
  console.log("This message appears after 2 seconds.");
}, 2000);
```

Exercises:

1. Define an anonymous function to return the character count of a string.
2. Use a function expression to create a `power(x, n)` function.

5. Callbacks (5.1.5)

A callback is a function passed as an argument to another function, executed after the other function completes. Callbacks are useful in asynchronous code like event handling and timing functions.

Characteristics and Benefits

- **Asynchronicity:** Enables reaction to events after they occur.
- **Flexibility:** Callbacks can modify a function's flow afterward.
- **Modularity:** Functions become extensible through callbacks.

Example – Simple Callback:

```
function doHomework(subject, callback) {
  console.log(`Starting my ${subject} homework.`);
  callback();
}

doHomework('math', function() {
  console.log('Finished my homework.');
});
```

Example – API Simulation Callback:

```
function getUserData(callback) {
  setTimeout(() => {
    const data = { name: 'John', age: 25 };
    callback(data);
  }, 1000);
}

getUserData((user) => {
  console.log(`User Name: ${user.name}, Age: ${user.age}`);
});
```

Exercises:

1. Implement `getWeatherData(city, callback)` that simulates weather data retrieval and passes it to a callback.
2. Write `runTasks(task1, task2, callback)` to execute two tasks in sequence and then call the callback.

6. Asynchronous Callbacks (5.1.6)

Asynchronous callbacks execute functions only after long-running processes finish, like data fetching or timed events.

Characteristics

- **Time-Independence:** Allows non-blocking execution.
- **Efficiency:** Resources aren't blocked during lengthy processes.
- **Queues:** Tasks follow the event-loop principle.

Example – Data Processing with `setTimeout`:

```
function fetchData(callback) {
  setTimeout(() => {
    const data = "Fetched data";
    callback(data);
  }, 3000);
}

fetchData((data) => {
  console.log(data); // Fetched data after 3 seconds
});
```

Exercises:

1. Write `asyncAddition(a, b, callback)` that returns the sum after a delay.
2. Implement a function simulating an API that returns a data object.

7. `setTimeout` and `setInterval` Functions (5.1.7)

These functions control task timing. `setTimeout` executes a function once after a delay, while `setInterval` repeats it at intervals.

Example – Countdown with `setTimeout`:

```
function countdown(seconds) {
  if (seconds > 0) {
    console.log(seconds);
    setTimeout(() => countdown(seconds - 1), 1000);
  } else {
    console.log("Happy New Year!");
  }
}

countdown(5);
```

Exercises:

1. Create a blinking element that stops after 5 seconds using intervals.
2. Implement a function that increments a counter every second and stops after a specified time.

8. Arrow Functions (5.1.8)

Arrow functions provide a modern, concise way to write functions and inherit the surrounding `this` context.

Characteristics and Benefits

- **Conciseness:** Less typing and increased readability.
- **Lexical `this`:** Inherits `this` from the surrounding context.
- **Inline Use:** Perfect for callback functions.

Example – Simple Arrow Function:

```
const multiply = (a, b) => a * b;
console.log(multiply(4, 5)); // 20
```

Exercises:

1. Write an arrow function that calculates the square of a value.
2. Use an arrow function to check if a number is even or odd.

With this comprehensive guide, you should be well-prepared for your lessons!