Debugging JavaScript Review

Common Types of Error Messages

- **SyntaxError**: These errors happen when you write something incorrectly in your code, like missing a parenthesis, or a bracket. Think of it like a grammar mistake in a sentence.

Example Code

```
const arr = ["Beau", "Quincy" "Tom"]
```

- **ReferenceError**: There are several types of Reference Errors, triggered in different ways. The first type of reference error would be not defined variables. Another example of a ReferenceError is trying to access a variable, declared with let or const, before it has been defined.

Example Code

```
console.log(num);

const num = 50;
```

- **TypeError**: These errors occur when you try to perform an operation on the wrong type.

Example Code

```
const developerObj = {
  name: "Jessica",
  country: "USA",
  isEmployed: true
};


developerObj.map()
```

- **RangeError**: These errors happen when your code tries to use a value that's outside the range of what JavaScript can handle.

Example Code

```
const arr = [];

arr.length = -1;
```

The throw Statement

- **Definition**: The throw statement in JavaScript is used to throw a user defined exception. An exception in programming, is when an unexpected event happens and disrupts the normal flow of the program.

Example Code

```
function validateNumber(input) {
 if (typeof input !== "number") {
  throw new TypeError("Expected a number, but received " + typeof input);
 }
 return input * 2;
}
```

try...catch...finally

- **Definition**: The try block is used to wrap code that might throw an error. It acts as a safe space to try something that could fail. The catch block captures and handles errors that occur in the try block. You can use the error object inside catch to inspect what went wrong. The finally block runs after the try and catch blocks, regardless of whether an error occurred. It's commonly used for cleanup tasks, such as closing files or releasing resources.

Example Code

```
function processInput(input) {
 if (typeof input !== "string") {
  throw new TypeError("Input must be a string.");
 }


 return input.toUpperCase();
}
```

```
try {
  console.log("Starting to process input...");
  const result = processInput(9);
  console.log("Processed result:", result);
} catch (error) {
  console.error("Error occurred:", error.message);
}
```

Debugging Techniques

- **debugger Statement**: This statement lets you pause your code at a specific line to investigate what's going on in the program.

Example Code

```
let firstNumber = 5;
let secondNumber = 10;

debugger; // Code execution pauses here
let sum = firstNumber + secondNumber;

console.log(sum);
```

- **Breakpoints**: Breakpoints let you pause the execution of your code at a specific line of your choice. After the pause, you can inspect variables, evaluate expressions, and examine the call stack.
- **Watchers**: Watch expressions lets you monitor the values of variables or expressions as the code runs even if they are out of the current scope.
- **Profiling**: Profiling helps you identify performance bottlenecks by letting you capture screenshots and record CPU usage, function calls, and execution time.

- **console.dir()**: This method is used to display an interactive list of the properties of a specified JavaScript object. It outputs a hierarchical listing that can be expanded to see all nested properties.

Example Code

```
console.dir(document);
```

- **console.table()**: This method displays tabular data as a table in the console. It takes one mandatory argument, which must be an array or an object, and one optional argument to specify which properties (columns) to display.