Debugging JavaScript Errors

Introduction

Debugging is an essential part of the development process. JavaScript is a dynamic language that can throw a variety of errors, and knowing how to identify and fix these issues is crucial for writing robust code. In this short lesson, we'll cover common types of errors, how to read error messages, and effective debugging techniques.

Common JavaScript Errors

1. **Syntax Errors**

   These occur when the code is not written correctly according to the language's rules. They can prevent your code from running entirely.

   Example:

```
function greet(name {
  return "Hello, " + name; // Missing closing parenthesis
}
```

   How to Spot:

   The console will show a message like "Unexpected token" and point to the location of the error.

2. Reference Errors

These happen when you try to use a variable that hasn't been declared.

Example:

```
console.log(user); // ReferenceError: user is not defined
```

How to Spot:

- The error message will indicate that a variable is not defined.

3. Type Errors

- These occur when a value is not of the expected type. For instance, calling a method that doesn't exist on a value.

- **Example:**

```
let str = 123;
console.log(str.toUpperCase()); // TypeError: str.toUpperCase is not a function
```

- **How to Spot:**
  - The error will specify that a method cannot be called on the variable.

4. **Range Errors**

   - Occur when a value is not within a valid range, like when trying to create an array with a negative length.
   - **Example:**

```
let arr = new Array(-1); // RangeError: Invalid array length
```

5. **Logical Errors**

   - These do not throw errors but result in incorrect behavior of the code, making them harder to spot.
   - **Example:**

```
function isEven(num) {
  return num % 2 = 0; // Should use '==' or '===' for comparison
}
```

   - **How to Spot:**
     - You may need to verify the output of your functions against expected results.

Debugging Techniques

1. **Using console.log()**
    - The simplest way to debug is to insert console.log() statements throughout your code. This will help you track variable values and understand the flow of execution.
    - **Example:**

```
function divide(a, b) {
  console.log(`Dividing ${a} by ${b}`);
  return a / b;
}
divide(10, 2);
```

    - **Best Practices:**
        - Use descriptive messages in console.log() to clarify what you are logging.
        - Remove or comment out console.log() statements in production code.

2. **Browser Developer Tools**
    - Every modern browser has built-in developer tools (press F12 or right-click -> Inspect).
    - **Features to Use:**
        - **Console:** Check for error messages and run JavaScript snippets directly.
        - **Sources:** Set breakpoints in your code to pause execution and inspect variables.
        - **Network:** Monitor network requests and responses, which is useful for debugging asynchronous code.

3. **Breakpoints**

Breakpoints allow you to pause code execution at a specific line, enabling you to inspect the current state of your application.

- **How to Use:**
  - Open the Sources tab in the developer tools, find your script, and click on the line number where you want to set a breakpoint.
- **Example:**

```
function calculate() {
  let total = 0;
  for (let i = 0; i < 5; i++) {
    total += i;
    debugger; // Use debugger; to pause execution here
  }
  return total;
}
console.log(calculate());
```