

Javascript Error Handling

JavaScript Errors

JavaScript **errors** happen when something **unexpected** takes place while your code is running.

Let's take a look at some **common JavaScript errors** you might come across:

- **SyntaxError**: This occurs when your code **violates** JavaScript's **syntax** rules.
- **ReferenceError**: When you try to access a variable or function that **doesn't exist**.
- **TypeError**: It shows up when you perform an operation on incompatible data types.
- **RangeError**: If a value falls **outside** the allowable **range**, this error is triggered.
- **Custom Errors**: JavaScript also allows you to **create your own custom errors**.

The Try-Catch Statement

By wrapping a block of code in a **try block** and catching potential **errors in the catch block**, you can prevent your program from **crashing when an error** occurs.

```
try {  
  // Code that may throw an error  
  const result = someUndefinedVariable + 10;  
} catch (error) {  
  console.log("Oops! An error occurred:", error.message);  
  // Output: Oops! An error occurred:  
  //           someUndefinedVariable is not defined  
}
```

Catching Specific Errors

Besides the generic catch block, you can **catch specific types of errors** by using multiple catch blocks.

```
try {
  // Code that may throw an error
  const result = someUndefinedVariable + 10;
} catch (error) {
  if (error instanceof ReferenceError) {
    console.log("Oh no! A reference error occurred:", error.message);
    // Output: Oh no! A reference error occurred:
    //                               someUndefinedVariable is not defined
  } else {
    console.log("Oops! A generic error occurred:", error.message);
    // Output: Oops! A generic error occurred:
    //                               Cannot read property '10' of undefined
  }
}
```

The Finally Block

The **finally block** is incredibly useful as it gets executed regardless of whether an **error occurs or not**.

- It's commonly used to perform **cleanup operations** or release resources,

```
try {
  // Code that may throw an error
  console.log("Inside the try block");
} catch (error) {
  console.log("Oops! An error occurred:", error.message);
} finally {
  console.log("The finally block is executed.");
}
// Output: Inside the try block
// Output: The finally block is executed.
```

Throwing Custom Errors

JavaScript allows you to create your own custom errors by extending the Error object

- This empowers you to define your own error types and provide more meaningful error messages to aid in debugging.

```
class MyCustomError extends Error {
  constructor(message) {
    super(message);
    this.name = 'MyCustomError';
  }
}

try {
  throw new MyCustomError('Uh-oh! This is a custom error.');
```

} catch (error) {
 console.log("Oops! An error occurred:", error.name, error.message);
 // Output: Oops! An error occurred:
 // MyCustomError Uh-oh! This is a custom error.
}

Defensive Coding

- While error handling is crucial, practicing defensive coding techniques and error prevention is equally important.
- This involves validating user input, checking for null or undefined values, and implementing error checks to handle potential edge cases.
- By incorporating defensive coding practices, you can minimize the occurrence of errors and enhance the overall stability of your code.