Error!

SD540 Server-Side Programming

Maharishi University of Management

Masters of Software Development

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

Defensive coding

- While error handling is crucial, practicing defensive coding techniques and 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.

Errors

- **Syntax Error**: These errors occur when there is a mistake in the code's grammar or structure.
- **Reference Error**: These errors occur when you try to access a variable or function that has not been declared or is out of scope.
- **Range Error**: When a value falls outside the allowable range.
- **Type Error**: It shows up when you perform an operation on incompatible data types.
- **Custom Error**: JavaScript also allows you to create your own custom errors.

The Try-Catch Statement

Wrap a block of code in a try block and catch potential errors in the catch block, you can prevent your program from crashing when an error occurs. Inside catch blocks: Access message, name, stack trace, etc.

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

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("An error occurred:", error.message);
} finally {
  console.log("The finally block is executed.");
}

// Output: Inside the try block
// Output: The finally block is executed.
```

Error Object

The **`Error`** object in JavaScript plays a crucial role in error handling.

- It is an object created when an error occurs during runtime.
- We can also create it manually using the `Error` constructor:
  `new Error("message")`

It contains information about the error, including its message, name, stack trace, and other properties.

Throwing an Error

```
try {
  // Throw an error
  throw new Error("Something went wrong")
} catch (error) {
   // Output: An error occurred: Something went wrong
   console.log("An error occurred:", error.message);
}
```

Creating Custom Errors

Extend **Error** to create specific error types with informative messages. This empowers you to define your own error types and provide more meaningful error messages to aid in debugging.

```
class CustomError extends Error {
    details: string = '';

    constructor(message: string, details: string) {
        super(message);
        this.details = details;
    }
}
```

Catching Specific Errors

Use multiple `if-statement` blocks with **`instanceof`** operator.

```
try {
  // Code that may throw an error
  throw new CustomError("Width error", "Width must not be 0");
} catch (error) {
  if (error instanceof CustomError) {
    console.log("A custom error occurred:", error.details);
    // Output: A custom error occurred: Width must not be 0
  } else {
    console.log("A generic error occurred:", error.message);
  }
}
```

Asynchronous Errors

```
// Promise.reject() is equivalent to throw
const calculate = () => Promise.reject(new Error("Something went wrong"));

try {
   calculate(); // Does not work
} catch (error) {
   console.log("An error occurred:", error.message);
}
```

Asynchronous Errors

```
const calculate = () => Promise.reject(new Error("Something went wrong"));

try {
   calculate().catch((error) => console.log(error.message));
} catch (error) {
   console.log("An error occurred:", error.message);
}
```

await for Asynchronous Calls

```
const calculate = () => Promise.reject(new Error("Something went wrong"));

const run = async () => {
   try {
      await calculate();
   } catch (error) {
      console.log("An error occurred:", error.message);
   }
};

run();
```

Best Practices for Error Handling

- Use try...catch blocks.
- Log errors to the console.
- Throw custom errors.
- Validate user input.
- Test your code thoroughly.

Fixing Type Errors

- X Is Not Assignable To Y
- Property Does Not Exist On Type
- X is Possibly Null Or Undefined
- Types Of Property Are Incompatible
- X Is Of Type Unknown
- Expression Can't Be Used To Index Type X
- Property X Has No Initializer