COESIONE ITALIA 21-27 PIEMONTE
Cofinanziato dall'Unione europea
REGIONE PIEMONTE

Web Developer

Programmazione - Javascript e Typescript

Docente: Shadi Lahham

Intervento realizzato da ITS TECNOLOGIE DELL'INFORMAZIONE E DELLA COMUNICAZIONE

Error handling

In Javascript

Shadi Lahham - Web development

Errors

Every program is susceptible to runtime errors

The choice of handling depends on the requirements of the application and the type of error

The approach:

- based on the team's preferences
- allows for easy problem diagnosis and resolution
- efficient and maintainable in the long run

01. Ogni programma può incontrare errori durante l'esecuzione.

02. La scelta di come gestire un errore dipende dalle esigenze dell'applicazione e dal tipo di errore.

03. L'approccio adottato può dipendere dalle preferenze del team, ma dovrebbe sempre rendere più facile identificare e risolvere i problemi, garantendo efficienza e manutenzione nel tempo.

Strategies

Error handling

Strategies

When a line of code produces a runtime error, you can handle it in several ways:

1. **Retry** - try to run an operation that failed again
2. **Fallback** - use a value that is a default fallback
3. **Error value** - return an error value but keep executing
4. **Propagate** - pass the error upwards
5. **Log** - write a warning or error
6. **Terminate** - end the application
7. **Ignore** - ignore the error and continue execution

1. Retry

Ritenta un'operazione fallita, utile per problemi di rete come chiamate API fallite.

Retry an operation that failed:
This approach is commonly used for dealing with network-related errors, such as failed API requests or database connections
It allows the application to automatically retry the operation a certain number of times before giving up

Retry

```
let retries = 0;

function doSomething() {
  try {
    // Code that might fail
    throw new Error('oops');
  } catch (error) {
    if (retries < 3) {
      retries++;
      doSomething();
    } else {
      console.log('Operation failed after 3 retries');
    }
  }
}

doSomething();
```

2. Fallback

Utilizza un valore di default se un'operazione fallisce (es., geolocalizzazione utente).

Using a default fallback value:
This approach is often used when the result of an operation is not critical and there is a reasonable fallback value that can be used in case of an error
For example, if a user's geolocation cannot be determined, the application might default to using the user's IP address

Fallback

```
function getUsername() {
  try {
    // Code that retrieves data from a server
    throw new Error('oops');
  } catch (error) {
    return 'Default User';
  }
}

console.log(`Username: ${getUsername()}`);
```

Error value

Ritorna un valore di errore ma continua l'esecuzione, utile per notificare l'errore senza interrompere il flusso.

Returning an error value:

Error value return is commonly used in lower-level code to notify calling code of an error

For instance, a file-reading function may return null if the file is not found

JavaScript's indexOf method returns -1 to indicate the absence of a specified element in an array, allowing the caller to handle the situation gracefully

Error value

```
// function must document error values that are returned
function calculateAverage(numbers) { SERVONO + CONTROLLI PER GLI ELMENTI NELL'ARRAY [ ]
  if (numbers.length === 0) {
    return null;
  }
  const sum = numbers.reduce((acc, num) => acc + num, 0);
  return sum / numbers.length;
}

// code that uses the function must handle  error values
const average = calculateAverage([]);
if (average === null) {
  console.error('Cannot calculate average of empty array');
}
```

4. Propagate

Passa l'errore a livelli superiori. (passiamo la responsabilità a chi invoca quella funzione)

Throwing a new Error:

This approach is commonly used for more critical errors that require the application to stop execution or for errors that cannot be handled by the current code block

It allows the error to be caught by higher-level error handlers or to bubble up to the top-level error handler

Propagate

```
function doSomething() {
  try {
    // Code that might fail
    throw new Error('oops');
  } catch (error) {
    throw new Error('Something went wrong');
  }
}

try {
  doSomething();
} catch (error) {
  console.log(error.message);
}
```

Propagate

```
function convertToNumber(str) {
  const num = Number(str);
  if (Number.isNaN(num)) {
    throw new Error('Invalid number format');
  }
  return num;
}

try {
  const num = convertToNumber('abc');
  console.log(`num: ${num}`);
} catch (error) {
  console.error(error.message);
}
```

5. # Log

Registra un avviso o un errore per monitoraggio e debugging.

Logging a console warning or error:
This approach is commonly used for non-critical errors or warnings that do not require the application to stop execution
It can be used for debugging purposes or to alert developers to potential issues in the code
Logging is useful for tracking errors and debugging issues in a production environment

Log

```
function doSomething() {
  try {
    // Code that might fail - non critical
    throw new Error('oops');
  } catch (error) {
    console.warn('Something went wrong:', error.message);
    // can also use console.error if the error is more serious
  }
}

doSomething();
```

messaggio di warining in arancione! (non blocca il codice)

6. Terminate

Termina l'applicazione in caso di errore critico (più comune in back-end).
non ha senso usarlo nel frontend (mostriamo una pagina bianca come errore critico!)

Terminating the application:
This approach is rarely used in modern front-end web development as it can lead to a poor user experience
However, in the case of a back-end Node.js server application, it may be necessary to terminate the server in the event of a critical error that renders it incapable of functioning, in order to mitigate any further harm

Terminate

```
function doSomething() {
  try {
    // Code that might fail
    throw new Error('oops');
  } catch (error) {
    console.error('Critical error:', error.message);

    // never a good reason to do this in a browser
    window.stop(); // stop the loading of the page

    // never a good reason to do this either
    window.close(); // close the page
  }
}

doSomething();
```

NON HANNO SENSO, se non va qualcosa noi non stoppiamo il collegamento!

HANNO SENSO nel contesto questa strategia se lavoriamo con la parte backend

7. Ignore

Continua l'esecuzione ignorando l'errore (sconsigliato perché può portare a risultati inaspettati o corruzione dei dati).

Ignoring the error:

This approach is not recommended as it can lead to unexpected behavior or data corruption

Ignoring errors means that the application continues running as if nothing happened, which can result in incorrect or incomplete results

It is better to handle errors in some way, even if it means using a default value or logging a warning

Ignore

```
function divide(a, b) {
  return a / b;
}

const result = divide(10, 0);
console.log(result); // Output: Infinity
```

Ignore

```
function doSomething() {
  try {
    // Code that might fail
    throw new Error('oops');
  } catch (error) {
    // do nothing
    // intentionally ignore the error - never a good practice
  }
}

doSomething();
```

Mechanisms

Error handling

Try catch

Try-catch blocks are a fundamental method of handling errors in JavaScript
When you use a try-catch block, you put the code that might cause an error inside a try block
If the code causes an error, the catch block is executed, allowing you to handle the error in some way

Try catch

```
try {
  // Code that might cause an error
} catch (error) {
  // Handle the error here
}
```

Finally

Finally Block:
A finally block is a block of code that is always executed, regardless of whether or not an error occurred in the try block
This is useful for releasing resources, such as closing a file or database connection or doing a UI update such as hiding an on-screen loading spinner

Finally

```
try {
  // Code that might cause an error
} catch (error) {
  // Handle the error here
} finally {
  // Code that is always executed
}
```

Throw

Throw Statements:

In JavaScript, you can throw an error by using a throw statement

When you throw an error, it stops the execution of the current function and looks for the nearest catch block to handle the error

Throw

```
if (someConditionIsNotMet) {
  throw new Error('Some descriptive error message');
}
```

Promise Rejection

Promise Rejection:
Promises are a common way of handling asynchronous code in JavaScript
When a Promise is rejected, it means that an error occurred. You can handle Promise rejections by attaching a catch block to the Promise chain

Promise Rejection

```
const myPromise = new Promise((resolve, reject) => {
  // do some async work here
  // resolve or reject the Promise based on the result
});

myPromise
  .then(result => {
    // handle the result
  })
  .catch(error => {
    // handle the error
  })
  .finally(() => {
    // do some final cleanup or UI update
  });
```

Window error handling

Window error handling:
The window object is a global object in the browser environment that represents the browser window or tab
Both window.onerror and window.addEventListener('error', ...) can be used to prevent default error handling and provide more information about the error but the latter provides greater flexibility and control over error handling

Window error handling

alternativa all'event listener

```
window.onerror = (message, url, line) => {
  console.error(`An error occurred at line ${line} of ${url}: ${message}`);
  return true; // Prevent default error handling
};
```

Window error handling

```
window.addEventListener('error', event => {
  console.error(`An error occurred at line ${event.lineno} of ${event.filename}: ${event.message}`);
  event.preventDefault(); // Prevent default error handling
});
```

Built-in error objects

Built-in error objects:
JavaScript provides built-in error objects, such as Error, TypeError, RangeError, and SyntaxError, that you can use to create custom error messages and handle different types of errors in your code

Built-in error objects

```
function multiply(a, b) {
  if (typeof a !== 'number' || typeof b !== 'number') {
    throw new TypeError('Both arguments must be numbers!');
  }
  return a * b;
}

console.log(multiply(10, 2)); // Output: 20
console.log(multiply('10', 2)); // Throws a TypeError: "Both arguments must be numbers!"
```

Custom Error Objects

Custom Error Objects:
You can create custom Error objects in JavaScript by extending the built-in Error object
This allows you to add additional properties or methods to the Error object that are specific to your application

Custom Error Objects

```
class CustomError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
  }

  logError() {
    console.log(`[${this.statusCode}] ${this.message}`);
  }
}

throw new CustomError('Something went wrong', 500);

// how would you catch and log this type of error?
```

Your turn

1.You're joking

Create a function called fetchRandomJoke() that fetches one random joke from a Random joke API and returns a promise that resolves with the text of the joke

Create a page that uses the function and displays the joke on the page or an error message if the promise rejects

The function

- Should use error handling to handle errors that may occur during the fetching
- Should return a Promise that resolves with the joke text, not the joke text itself
- If the fetch operation fails, the function should retry the operation up to 3 times before giving up
- If the fetch operation fails after 3 attempts, the function should reject the promise

Bonus
Create variants that can fetch jokes by number and by type

2.Validate me

Write a function validatePassword() that returns true if a password meets the following requirements:

- Must be at least 8 characters long
- Must contain at least one uppercase letter, one lowercase letter, one digit and one symbol

If the password is invalid, the function should throw a custom error object with the message "Invalid password format" and the reason the password is not valid

This code tests the function. Add more cases to it

```
try {
  const validPassword = 'Abcdefg$1';
  const invalidPassword = 'abcdefg1';

  console.log(validatePassword(validPassword)); // true
  console.log(validatePassword(invalidPassword)); // throws error
} catch (error) {
  console.error(error.message); // "Invalid password format - no uppercase"
}
```

References

Control flow and error handling
Error handling, "try...catch"
Custom errors, extending Error

References

Error
RangeError
TypeError
console: warn() method
console: error() method