@CODE.CLASH

Generators

JAVASCRIPT

Introduction

- JavaScript generators are special functions that can be paused and resumed during execution.
- They use the `yield` keyword to produce values and control the flow of iteration.
- Regular functions return only one, single value but Generators can return ("yield") multiple values.
- We'll discover what makes them unique, how they differ from regular functions, and why they are valuable additions to JavaScript.

Example

To create a generator, you need to first define a generator function with `function*` symbol.

- When a it's called, it doesn't immediately run its code. Instead, it returns a special object known as a "generator object."

```
function* myGenerator() {
  yield 1;
  yield 2;
  yield 3;
}

const gen = myGenerator();

console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: 3, done: false }
console.log(gen.next()); // { value: undefined, done: true }
```

Passing Arguments

Generators are not only great at producing values, but they're also excellent listeners!

- They can receive values from the outside world and send them back too.

```
function* twoWayGenerator() {
  const value = yield "Please provide a value";
  yield `You provided: ${value}`;
}

const gen = twoWayGenerator();

console.log(gen.next());
// { value: "Please provide a value", done: false }
console.log(gen.next(42));
// { value: "You provided: 42", done: false }
console.log(gen.next());
// { value: undefined, done: true }
```

Iterating With Generators

Generators have a hidden talent — they can create custom iterators! It's like giving them the power to control the flow of iteration.

- Here's is the example how you can build custom iterators using generators.
- We can loop over their values using `for..of`.

```
function* range(start, end, step) {
  let current = start;
  while (current <= end) {
    yield current;
    current += step;
  }
}

const numbers = range(1, 10, 2);

for (const num of numbers) {
  console.log(num); // Output: 1, 3, 5, 7, 9
}
```

Generator Delegation

Generators have a talent for teamwork too! They can join forces and delegate tasks among themselves.

- It's like assembling a dream team of generators.

```
function* generatorOne() {
  yield 1;
  yield 2;
}
function* generatorTwo() {
  yield 3;
  yield 4;
}
function* composedGenerator() {
  yield* generatorOne();
  yield* generatorTwo();
}
const gen = composedGenerator();

console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: 3, done: false }
console.log(gen.next()); // { value: 4, done: false }
console.log(gen.next()); // { value: undefined, done: true }
```

Error Handling

Error handling is essential in any application, and generators provide mechanisms to handle errors gracefully.

- We'll learn how to handle errors within generator functions using `try...catch` blocks.

```
function* errorGenerator() {
  try {
    yield 1;
    yield 2;
    throw new Error("Oops, something went wrong!");
    yield 3; // This line will not be reached
  } catch (error) {
    yield `Error: ${error.message}`;
  }
}

const gen = errorGenerator();

console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next());
// { value: "Error: Oops, something went wrong!", done: false }
console.log(gen.next()); // { value: undefined, done: true }
```

Conclusion

- Generators are created by generator functions `function* f(…) {…}`.
- Inside generators (only) there exists a `yield` operator.
- The outer code and the generator may exchange results via `next`/`yield` calls.
- As always, I hope you enjoyed the post and learned something new.
- If you have any queries then let me know in the comment box.