

1) Is Generators

→ They are special functions that can be paused and resumed during execution.

→ They use keyword yield → To produce values & control flow of iteration.

→ Regular functions return only one single value
Generator return → multiple values.

→ To create a generator → Define generator function → (function)

→ When called → Doesn't immediately run its code. → It returns → Special object
↓
Generator object.

→ Syntax: function* functionname()

→ Yield: operator is used to pause & resume generator function asynchronously.
→ Only used within generator function that contain it

→ Syntax: yield expression;

→ Pauses execution → generator function → return its operand
Generator caller. (expression) to

2) Working

→ When generator function is called → returns generator object

→ When next() is called on generator object → resumes execution
↓
Generator object

→ When yield expression is encountered

→ Pauses execution

→ Returns expression after yield keyword to object caller
(next())

→ next() → Returns Iterator object & Properties (Value, done)

→ Value → actual value of expression

→ done → Boolean value

→ true → If generator function is executed completely
→ false →

Eg 1) Generator function Demo

```

<script>
function* generator() {
  yield 1;
  yield 2;
  yield 2;
}

const gen = generator();
console.log(gen.next());
console.log(gen.next());
console.log(gen.next());
console.log(gen.next());
</script>

```

Eg 2) They are not only great producers they are also great listeners.

(1) Receive values from outside world & send them back too.

```

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

const gen = twoWayGenerator();
console.log(gen.next());
console.log(gen.next(45));
console.log(gen.next());
</script>

```

Eg 3) Iterator with Generator

(1) Create custom Iterators - giving them
(2) Loop -> Over their values
Using `for...of` loop
power to control flow of iteration

(1) `for in` (Used for iterating over enumerable properties of object)
(2) Iterating over Iterable objects
Not Arrays
Set
Generators

Working code

INo	Value	done	Step/Curr
1	1	false	2 CUR = 1
2	3	false	1 < 50
			CUR = 1
			Step

<body>

```

<script>
function* myRange(start, end, step) {
  let current = start;
  while (current <= end) {
    yield current;
    current += step;
  }
}

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

for (const num of numbers) {
  console.log(num);
}

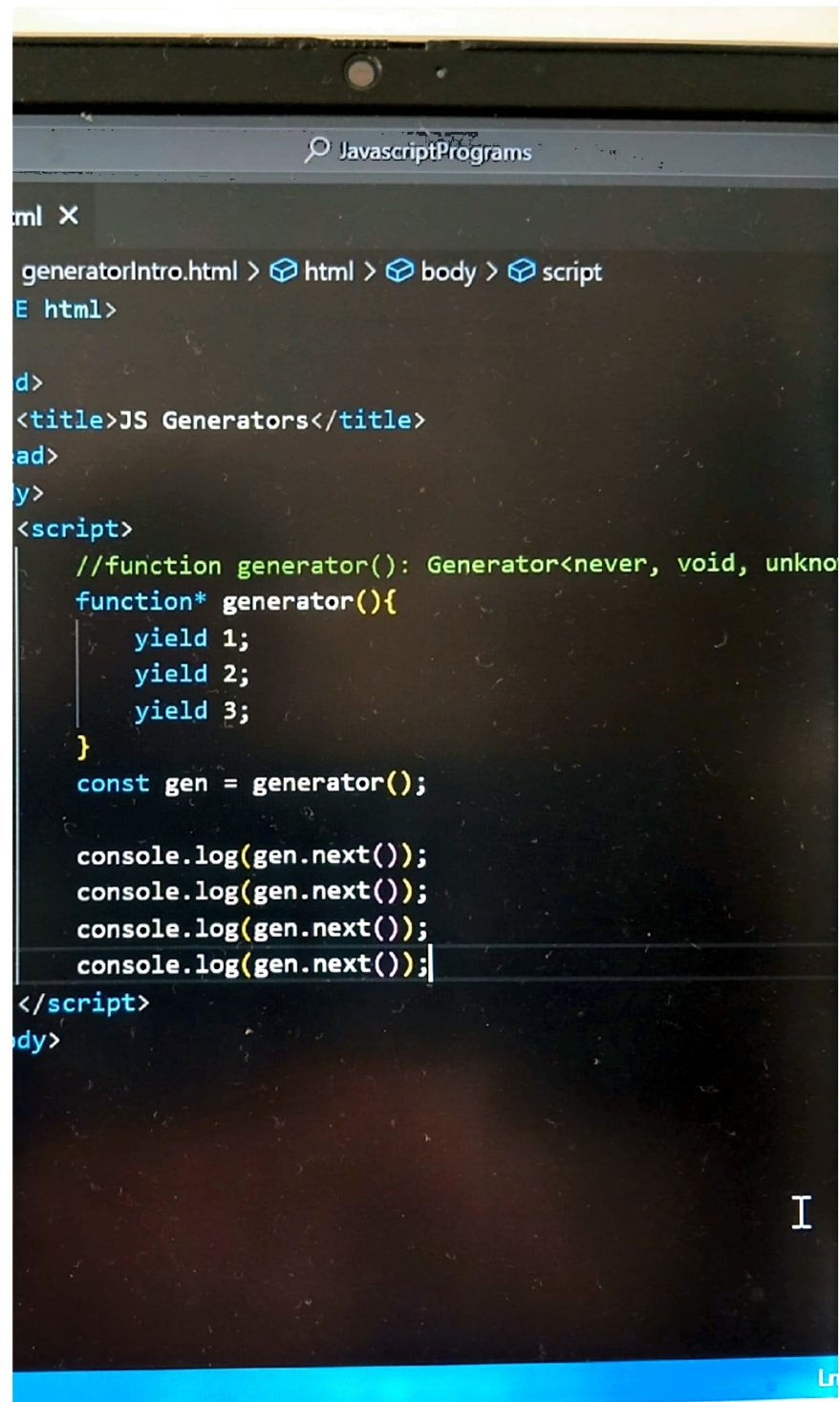
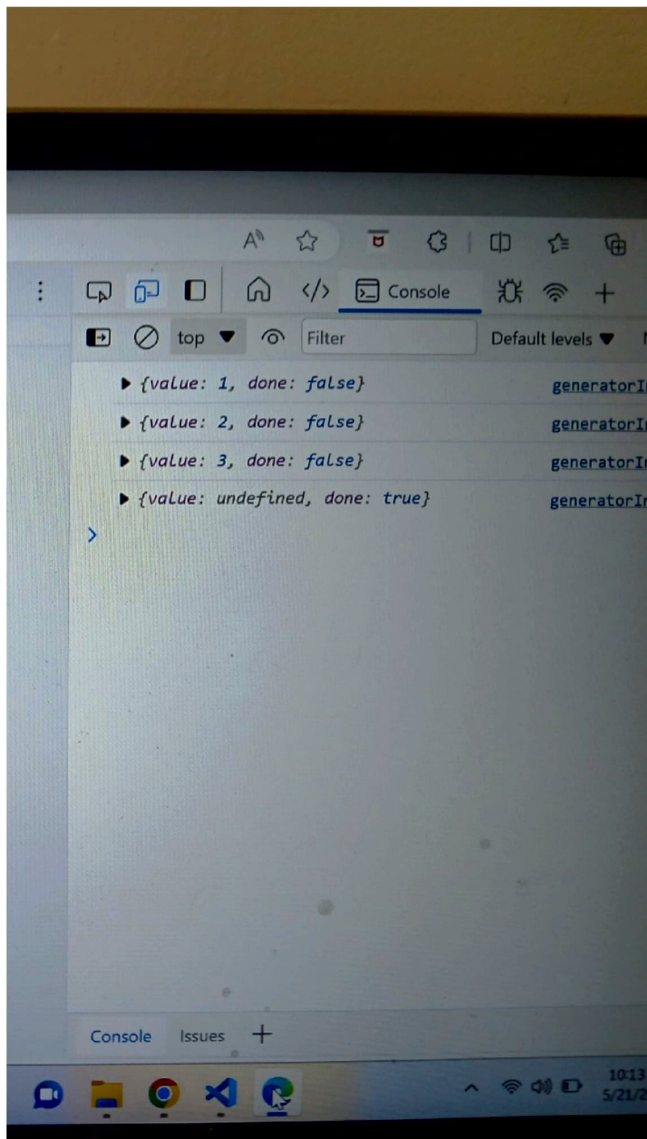
</script>
</body>

```

Eg 4) Generators Delegate

They have talent for teamwork too.

They are join force and delegate tasks among themselves.



generator_Delegation.html X

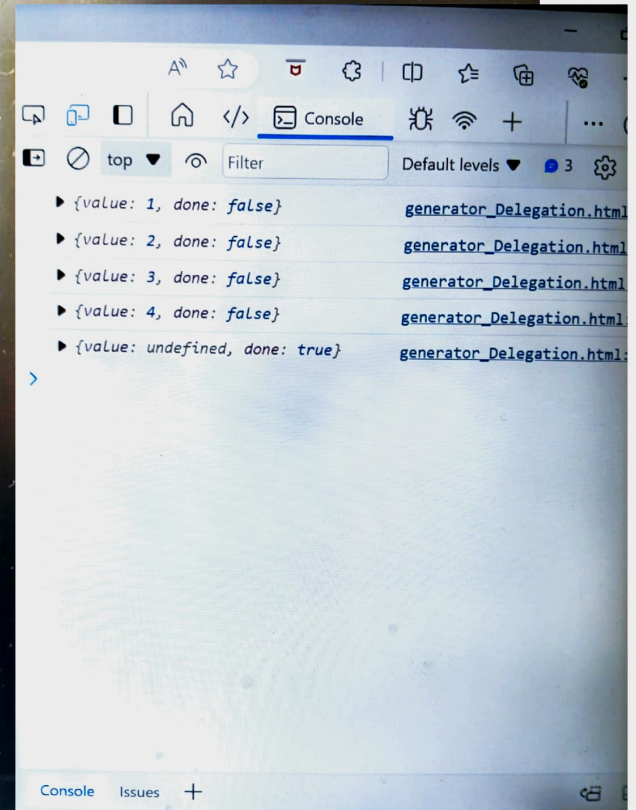
Generators > generator_Delegation.html > html > body > script

```
<html>
<body>
  <script>
    //function generator(): Generator<never, void, unknown>
    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());
    console.log(gen.next());
    console.log(gen.next());
    console.log(gen.next());
    console.log(gen.next());

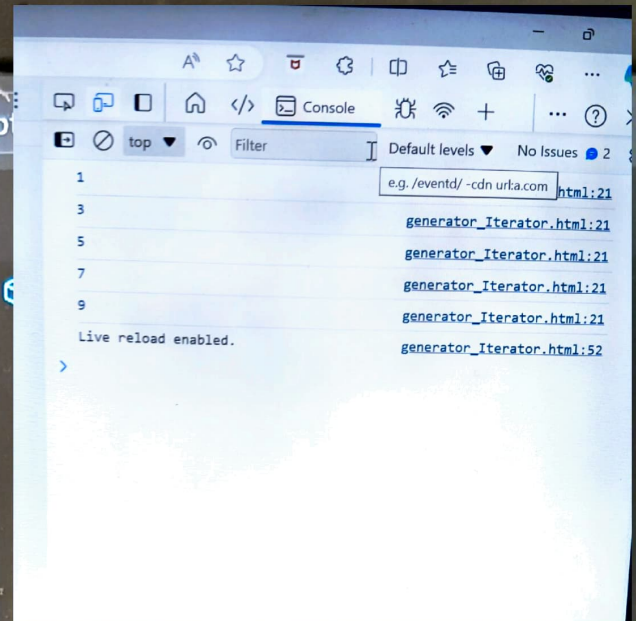
  </script>
</body>
```



generator_iterator.html X

JS_Generators > generator_iterator.html > html >

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5    <title>JS Generators</title>
6  </head>
7
8  <body>
9    <script>
10      //function generator(): Generator<never, void, unknown>
11      function* myRange(start, end, step) {
12        let current = start;
13        while (current <= end) {
14          yield current;
15          current += step;
16        }
17      }
18      const numbers = myRange(1, 10, 2);
19
20      for(const num of numbers) {
21        console.log(num);
22      }
23    </script>
24  </body>
```



Error Handling

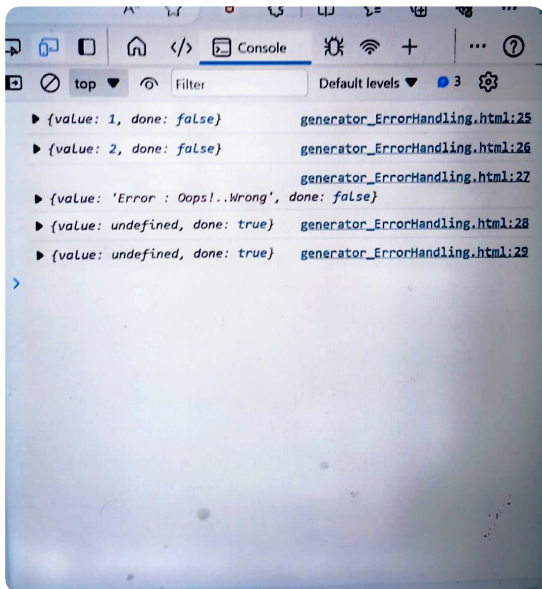
Error handling is essential in any application, and generators provide mechanisms to handle errors gracefully.

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

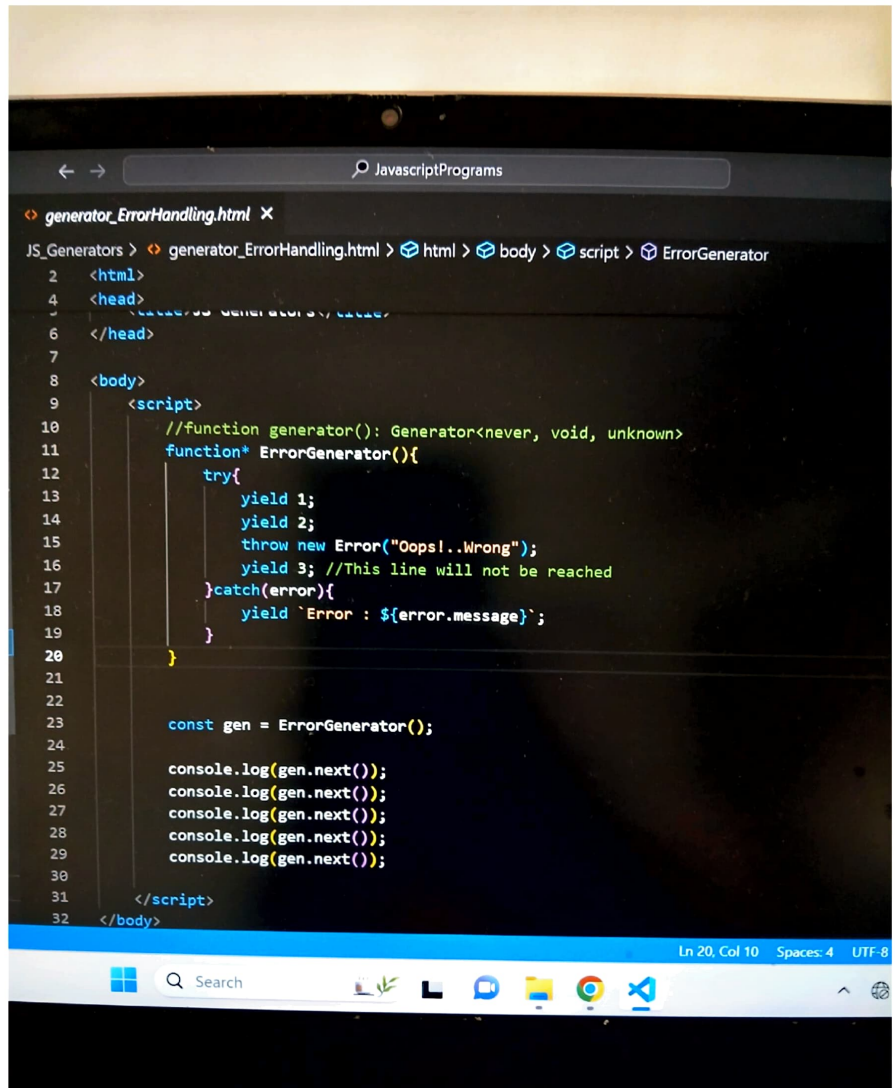
```
function* errorGenerator() {
  try {
    yield 1;
    yield 2;
    throw new Error("Oops, something went wrong!");
    yield 3; // This line will not be reached
  } catch (error) {
    yield `Error: ${error.message}`;
  }
}

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 }
```



```
{value: 1, done: false}      generator_ErrorHandling.html:25
{value: 2, done: false}      generator_ErrorHandling.html:26
{value: 'Error : Oops!..Wrong', done: false} generator_ErrorHandling.html:27
{value: undefined, done: true} generator_ErrorHandling.html:28
{value: undefined, done: true} generator_ErrorHandling.html:29
```



```
generator_ErrorHandling.html X
JS_Generators > generator_ErrorHandling.html > html > body > script > ErrorGenerator
2  <html>
4  <head>
6  </head>
7
8  <body>
9    <script>
10      //function generator(): Generator<never, void, unknown>
11      function* ErrorGenerator(){
12        try{
13          yield 1;
14          yield 2;
15          throw new Error("Oops!..Wrong");
16          yield 3; //This line will not be reached
17        }catch(error){
18          yield `Error : ${error.message}`;
19        }
20      }
21
22
23      const gen = ErrorGenerator();
24
25      console.log(gen.next());
26      console.log(gen.next());
27      console.log(gen.next());
28      console.log(gen.next());
29      console.log(gen.next());
30
31    </script>
32  </body>
```

Ln 20, Col 10 Spaces: 4 UTF-8

rator_PassingArguments.html X

rators > generator_PassingArguments.html > html > body > script > twoWaygenerator

<!DOCTYPE html>

<html>

<head>

<title>JS Generators</title>

</head>

<body>

<script>

//function generator(): Generator<never, void, unknown>

function* twoWaygenerator(){

const value = yield "Please Provide a value";

yield `You Provided : "\${value}"`;

}

const gen = twoWaygenerator();

console.log(gen.next());

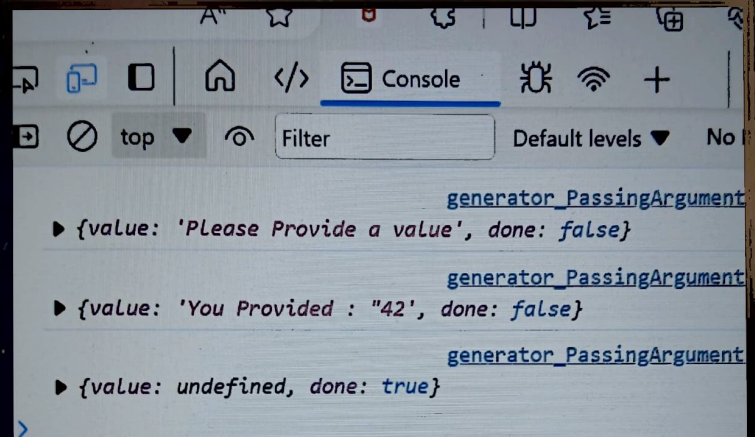
console.log(gen.next(42));

console.log(gen.next());

</script>

</body>

</html>



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.