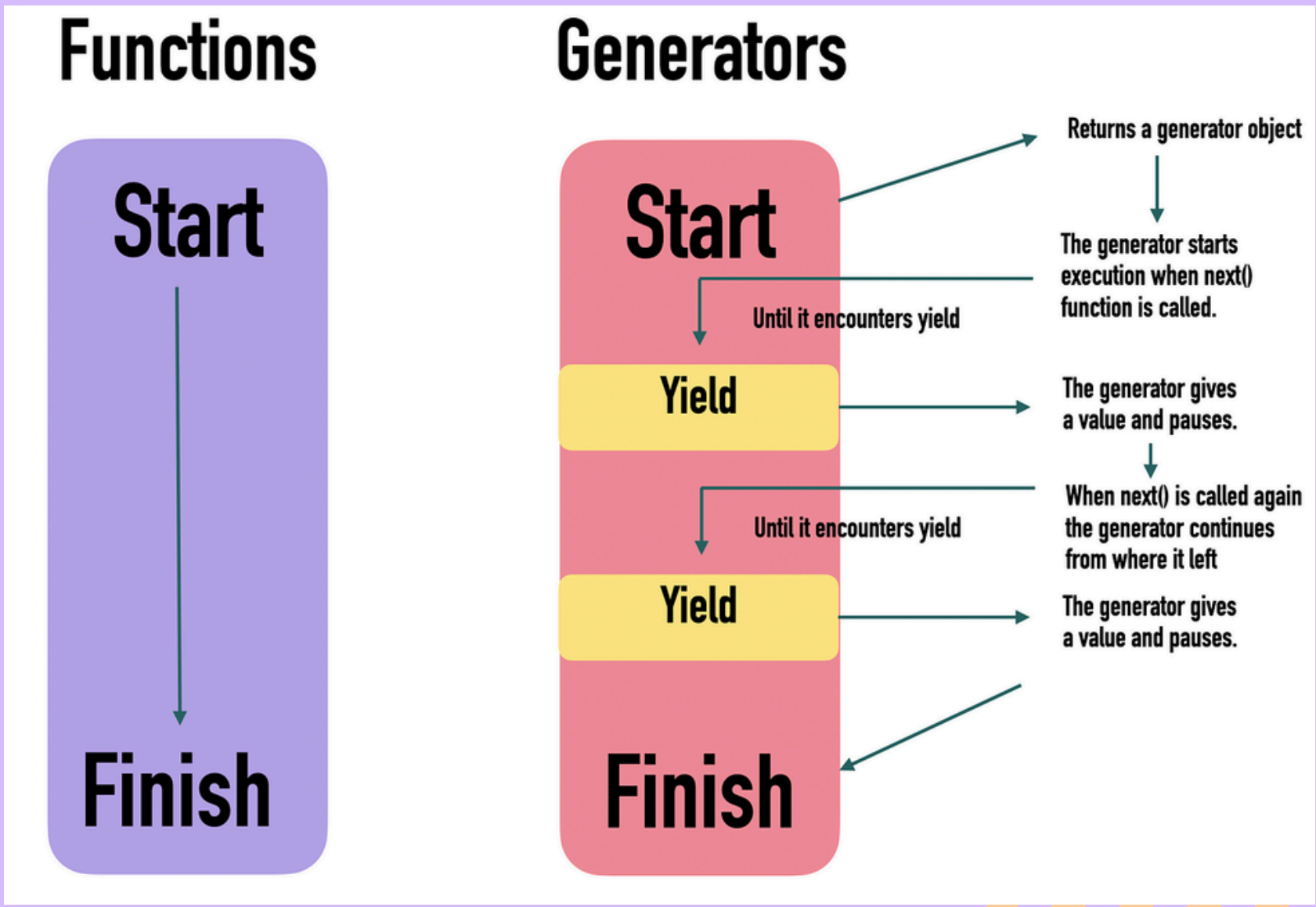
What are Generators in JS?

Vladyslav Demirov
@vladyslav-demirov

What are Generators in JavaScript?

Generators are functions that can pause and resume execution, producing a sequence of values over time. They are perfect for handling data streams, lazy evaluations, and asynchronous code.



Vladyslav Demirov

Basic Generator Syntax

Generators use the function* syntax. Use the yield keyword to produce values.

JS

```
function* basicGenerator() {
  yield 1;
  yield 2;
  yield 3;
}
const generator = basicGenerator();
console.log(generator.next().value); // 1
console.log(generator.next().value); // 2
console.log(generator.next().value); // 3
```

Vladyslav Demirov

Yielding Values

Use the yield keyword to pause the generator and return a value. Execution can be resumed with the next() method.

JS

```
function* countUp() {
  let i = 0;
  while (true) {
    yield i++;
  }
}
const counter = countUp();
console.log(counter.next().value); // 0
console.log(counter.next().value); // 1
console.log(counter.next().value); // 2
```

Vladyslav Demirov

Infinite Sequences

Generators can create infinite sequences, making them ideal for scenarios where you don’t know the length of the sequence in advance.

JS

```
function* incrementingGenerator() {
  let value = 0;
  while (true) {
    yield value += 2;
  }
}
const generator = incrementingGenerator();
console.log(generator.next().value); // 2
console.log(generator.next().value); // 4
console.log(generator.next().value); // 6
```

Vladyslav Demirov

Practical Usage - Lazy Evaluation

Generators can generate values on demand, making them suitable for large datasets or infinite sequences.

JS

```
function* fibonacci() {
  let [a, b] = [0, 1];
  while (true) {
    [a, b] = [b, a + b];
    yield a;
  }
}
const sequence = fibonacci();
console.log(sequence.next().value); // 1
console.log(sequence.next().value); // 1
console.log(sequence.next().value); // 2
console.log(sequence.next().value); // 3
console.log(sequence.next().value); // 5
```

Vladyslav Demirov

Practical Usage - Handling Data Streams

Generators can generate values on demand, making them suitable for large datasets or infinite sequences.

JS

```js
async function* fetchData() {
  const urls = ['/api/data/1', '/api/data/2', '/api/data/3'];
  for (const url of urls) {
    const response = await fetch(url);
    const data = await response.json();
    yield data;
  }
}
const dataGenerator = fetchData();
(async () => {
  for await (const data of dataGenerator) {
    console.log(data);
  }
})();
```

Vladyslav Demirov

Practical Usage - Implementing Iterators

Generators can be used to create custom iterators.

JS

```js
const myIterable = {
  *[Symbol.iterator]() {
    yield 1;
    yield 2;
    yield 3;
  }
};
for (const value of myIterable) {
  console.log(value); // 1, 2, 3
}
```

Vladyslav Demirov

HAPPY CODING

Vladyslav Demirov
@vladyslav-demirov