Generators

JavaScript Developer Series

Peter Munro

Generators

Generators

Introducing Generators

Generators:

- are special functions, defined as `function* myGenerator() { … }`
- are like 'processes': they can be called, paused and resumed at different stages of their execution

Generator Function

To write a generator function, we use the `function*` keyword:

```
function* makeGenerator() {
  console.log('First');
  console.log('Second');
}
```

When we call a generator function, it returns a new generator instance:

```
let gen = makeGenerator();
```

Using a Generator Instance

Now we have an instance, we can call `.next()` on it:

```
let result = gen.next();
```

Output:

```
First
Second
```

Return value:

```
console.log(result);    // {value: undefined, done: true}
```

`yield` to "pause" a generator

- Within a generator function, we can call `yield`
- `yield` pauses execution and returns to the caller:

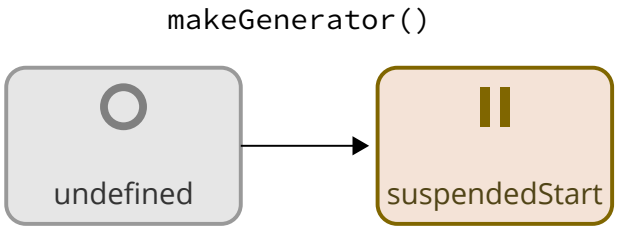
Caller

```
> let g = makeGenerator();
> g.next();  ▶️
First
☛ {value: undefined, done: false}
> g.next();
Second
☛ {value: undefined, done: true}
```

Generator

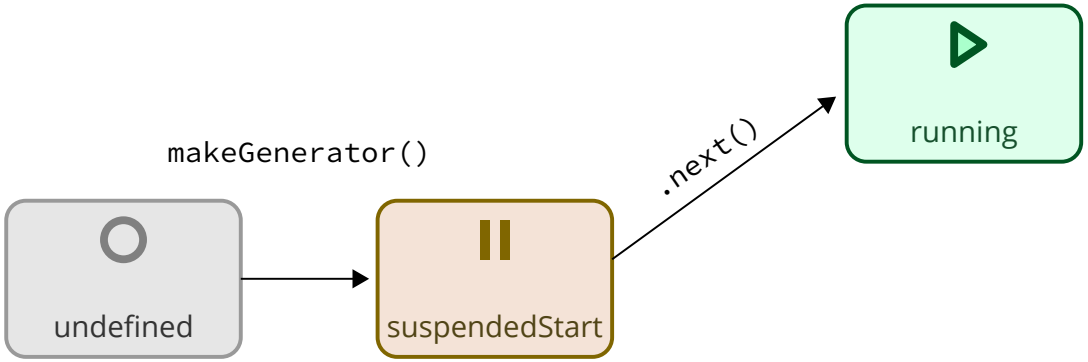
```
function* makeGenerator() {
  console.log('First');
  yield;  ⏸️
  console.log('Second');
}
```

Generator States



- At any given moment, the generator is in one 'state'
- Once instantiated by calling the generator function, the generator moves from the `undefined` state to `suspendedStart`
  - It's now paused

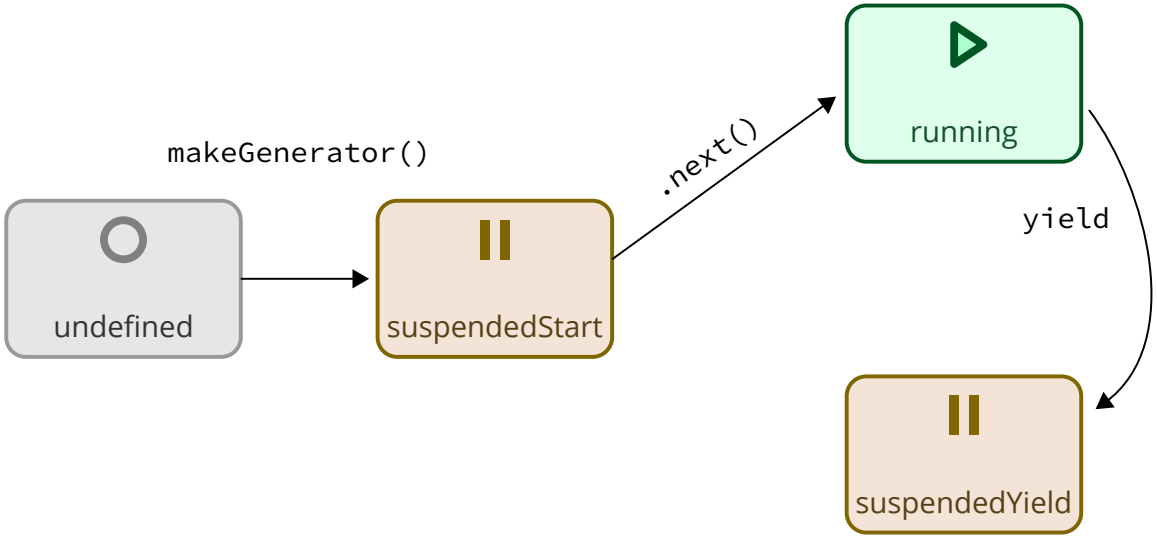
Starting the Generator



- Calling `.next()` on the generator moves it to the `running` state

```
gen.next();
```

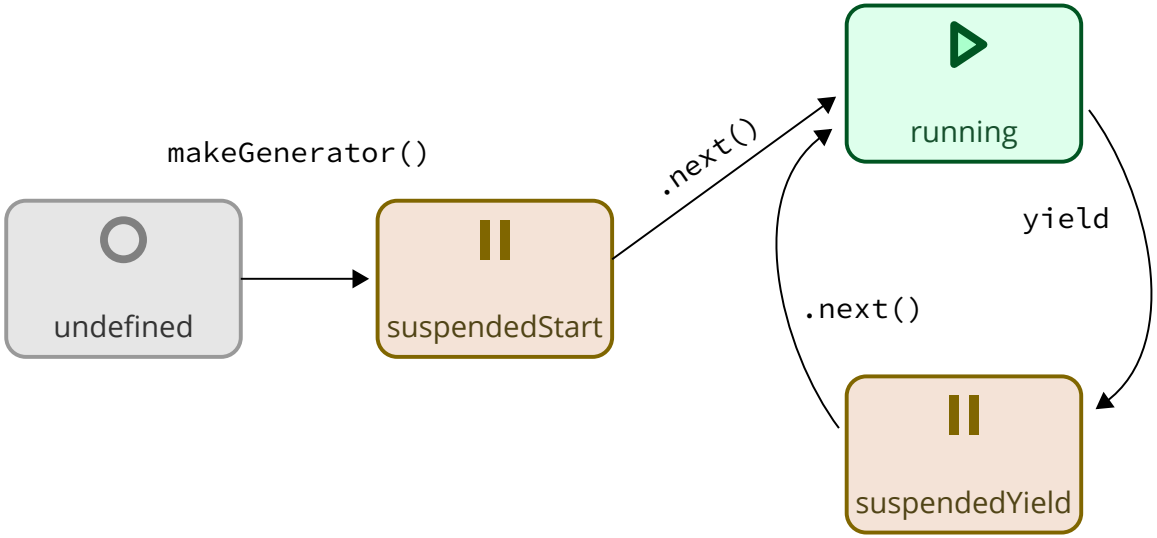
Pausing



- Calling `yield` within the generator function moves it to the `suspendedYield` state

```
yield;
```

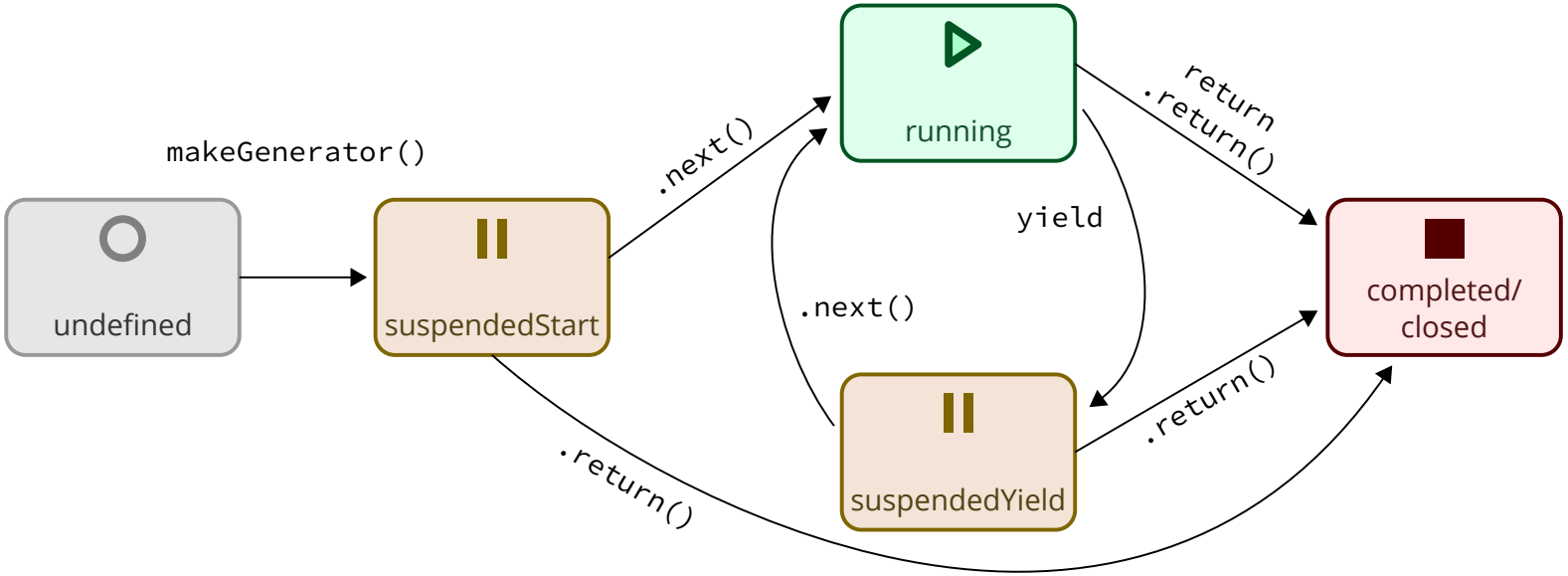
Resuming



- To resume execution, call `.next()` on the generator instance

```
gen.next();
```

Completing Execution



- When the generator function finishes, it moves to the `completed` state automatically
- You can also move it here yourself:
  - Within the generator function, use `return`; or
  - From the generator instance, call `.return()`

Passing Data to and from the Generator

Data can be passed:

- from generator to the calling code
  - using `yield` *expression*
- from the calling code to the generator
  - by passing a parameter into `.next(param)`

Passing data from the Generator

- `yield` can take an expression
- the `yield`-ed expression is returned from the caller's `.next()`

Caller

```
> let g2 = makeGenerator();
> g2.next();  ▶️
First
☛ {value: "a result", done: false}
> g2.next();
Second
☛ {value: undefined, done: true}
```

Generator

```
function* makeGenerator() {
  console.log('First');
  yield 'a result';  ⏸️
  console.log('Second');
}
```

Generating a Sequence

We can use this to generate a sequence of values:

```
var p = makePlanets();
p.next();       // {value: "Mercury", done: false}
p.next();       // {value: "Venus", done: false}
p.next();       // {value: "Earth", done: false}
p.next();       // {value: "Mars", done: false}
p.next();       // {value: undefined, done: true}
p.next();       // {value: undefined, done: true}
```

```
function* makePlanets() {
  yield 'Mercury';
  yield 'Venus';
  yield 'Earth';
  yield 'Mars';
  // etc
}
```

- Once there is no more code to run (or the generator otherwise moves to the `completed` state), any further calls to `.next()` will always return with `done: true`

Generators are Iterable

- This means you can use any syntax that expects iterables to access data produced by a generator

Destructuring example:

```
[first,, third] = makePlanets();
```

Output

```
console.log(first);      // "Mercury"
console.log(third);      // "Earth"
```

Array spread example:

```
[...makePlanets()]      // ["Mercury", "Venus", "Earth", "Mars"]
```

Consuming a Generator with the `for...of` Loop

You can also use the `for...of` loop:

```
for (let planet of makePlanets()) {
  console.log(`Next planet is: ${planet}`);
}
```

Output

```
Next planet is: Mercury
Next planet is: Venus
Next planet is: Earth
Next planet is: Mars
```

Exercise

Write a generator to generate numbers of the Fibonacci Sequence

Passing data to the Generator

- the caller's `.next(param)` can take a parameter value
- the `yield` expression evaluates to this value

Caller

```
> let g3 = makeGenerator();
> g3.next();  ▶️
First
☛ {value: "a result", done: false}
> g3.next('some data');
Second: some data
☛ {value: undefined, done: true}
```

Generator

```
function* makeGenerator() {
  console.log('First');
  let data = yield 'a result';  ⏸️
  console.log('Second:', data);
}
```