SNS COLLEGE OF ENGINEERING

Kurumbapalayam(Po), Coimbatore – 641 107

An Autonomous Institution

Accredited by NAAC-UGC with 'A' Grade

Approved by AICTE, Recognized by UGC & Affiliated to Anna University, Chennai

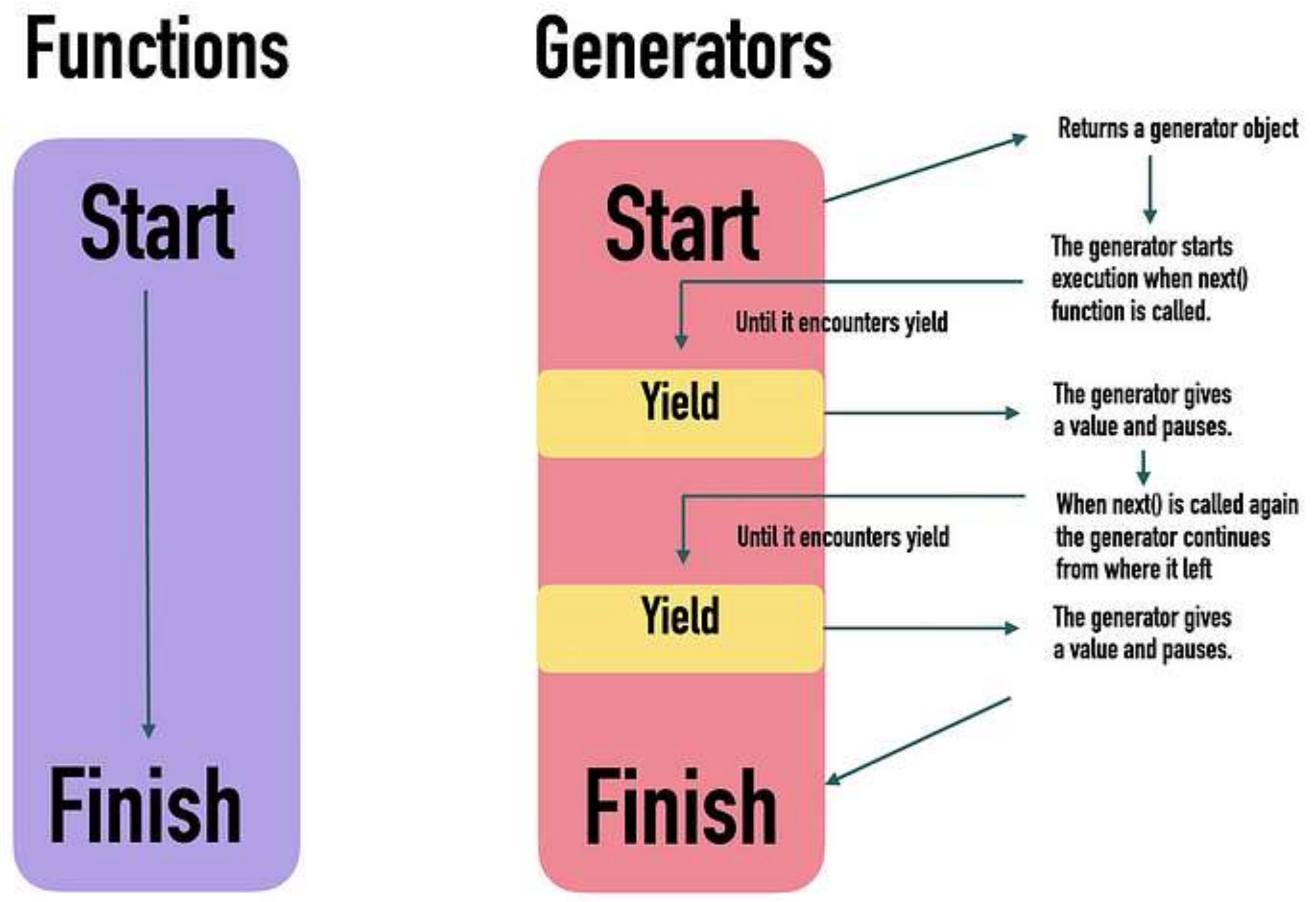
DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY

Course Code and Name : 19TS601 FULL STACK DEVELOPMENT

Unit 1 : JAVASCRIPT AND BASICS OF MERN STACK

Topic : Generators

Generators


Functions
Start
Finish
Generators
Start
Until it encounters yield
Yield
Until it encounters yield
Yield
Finish
Returns a generator object
The generator starts execution when next() function is called.
The generator gives a value and pauses.
When next() is called again the generator continues from where it left
The generator gives a value and pauses.

Generators

- A **generator function** is a special type of function that can pause its execution at any point and resume later.
- They are defined using the function* syntax and use the **yield keyword** to pause execution and return a value.
- A special syntax construct: function*, so-called “generator function”.

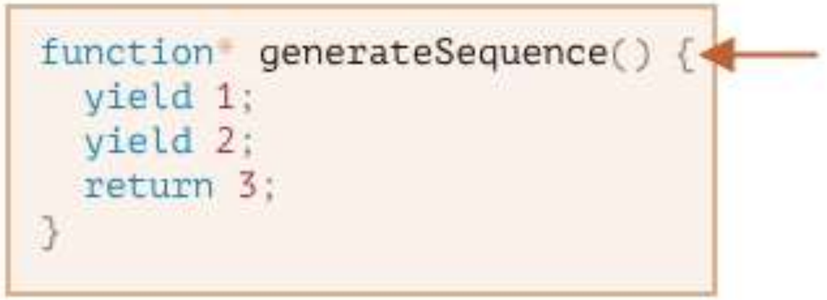
Syntax

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

Generator functions behave differently from regular ones. When such function is called, it doesn’t run its code. Instead it returns a special object, called “generator object”, to manage the execution.

```
function* generateSequence()
{ yield 1;
  yield 2;
  return 3;
 } // "generator function" creates "generator object"
let generator = generateSequence();
alert(generator); // [object Generator]
```

The function code execution hasn’t started yet:



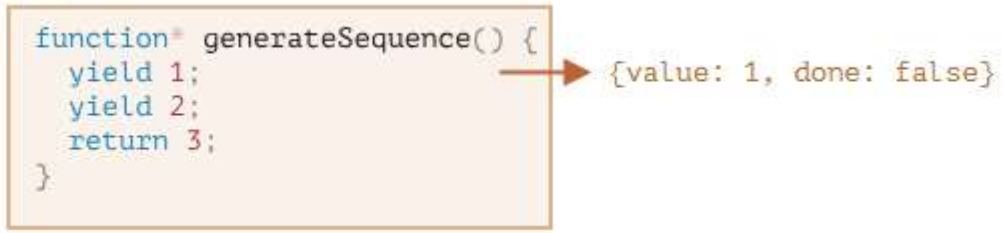
```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

The main method of a generator is `next()`. When called, it runs the execution until the nearest `yield <value>` statement (`value` can be omitted, then it’s `undefined`). Then the function execution pauses, and the yielded `value` is returned to the outer code.
The result of `next()` is always an object with two properties:
- `value`: the yielded value.
- done: `true` if the function code has finished, otherwise `false`.

we create the generator and get its first yielded value:

```
function* generateSequence()
{ yield 1;
  yield 2;
  return 3;
}
 let generator = generateSequence();
 let one = generator.next();
alert(JSON.stringify(one)); // {value: 1, done: false}
```

As of now, we got the first value only, and the function execution is on the second line:

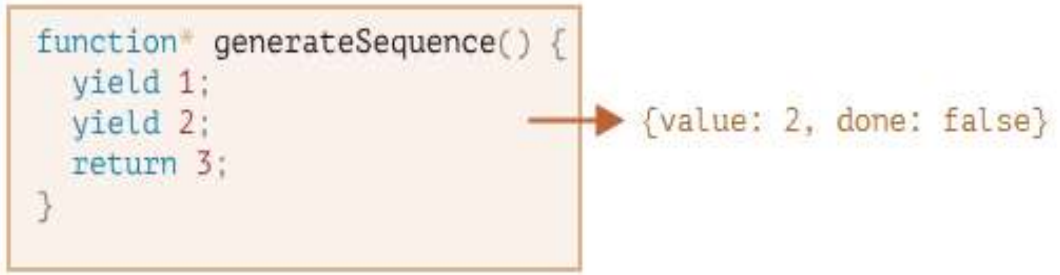


```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

Let’s call `generator.next()` again. It resumes the code execution and returns the next `yield`:

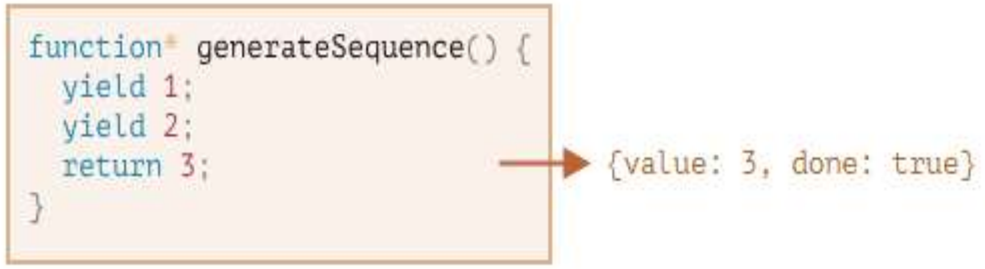
let two = generator.next();

alert(JSON.stringify(two)); // {value: 2, done: false}



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

- And, if we call it a third time, the execution reaches the return statement that finishes the function:
- let three = generator.next();
- alert(JSON.stringify(three)); // {value: 3, done: true}



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

{value: 3, done: true}

- Now the generator is done.
- We should see it from `done:true` and process `value:3` as the final result.
- New calls to `generator.next()` don't make sense any more. If we do them, they return the same object: `{done: true}`.
- **`function* f(…)` or `function *f(…)`?**
- Both syntaxes are correct.
- But usually the first syntax is preferred, as the star `*` denotes that it's a generator function, it describes the kind, not the name, so it should stick with the `function` keyword.

Generators are iterable

The next() method, generators are iterable.

We can loop over their values using for..of:

```
function* generateSequence()
{ yield 1;
  yield 2;
  return 3;
}
let generator = generateSequence();
for(let value of generator)
{
alert(value); // 1, then 2
}
```

- The example above shows `1`, then `2`, and that’s all.
- It doesn’t show `3`!
- It’s because `for..of` iteration ignores the last `value`, when `done: true`. So, if we want all results to be shown by `for..of`, we must return them with `yield`:

```
function* generateSequence()
{
 yield 1;
 yield 2;
 yield 3;
 }
 let generator = generateSequence();
 for(let value of generator)
 {
 alert(value); // 1, then 2, then 3
 }
```

As generators are iterable, we can call all related functionality, e.g. the spread syntax ...:

```
function* generateSequence()
{ yield 1;
  yield 2;
  yield 3;
 }
let sequence = [0, ...generateSequence()];
alert(sequence); // 0, 1, 2, 3
```

In the code above, `...generateSequence()` turns the iterable generator object into an array of items

Modules

JavaScript modules allow you to break up your code int
separate files.
This makes it easier to maintain a code-base.
Modules are imported from external files with
the `import` statement.
Modules also rely on `type="module"` in the <script> tag.

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Modules</h1>
<p id="demo"></p>
<script type="module">
import message from "./message.js";
document.getElementById("demo").innerHTML = message();
</script>
</body>
</html>
```

Output

- **JavaScript Modules**
- Jesse is 40 years old.

Export

- Modules with **functions** or **variables** can be stored in any external file.
- There are two types of exports: **Named Exports** and **Default Exports**.

Named Exports

Create a file named `person.js`, and fill it with the things we want to export.

You can create named exports two ways. In-line individually, or all at once at the bottom

In-line individually:

```
person.js
export const name = "Jesse";
export const age = 40;
```

- All at once at the bottom:

```
person.js
const name = "Jesse";
const age = 40;
export {name, age};
```

Default Exports

create another file, named `message.js`, and use it for demonstrating default export.
You can only have one default export in a file

Example

message.js

- const message = () => {
  const name = "Jesse";
  const age = 40;
  return name + ' is ' + age + 'years old.';
  };

  export default message;

Import

- Import modules into a file in two ways, based on if they are named exports or default exports.
- Named exports are constructed using curly braces. Default exports are not.

- Import from named exports
- Import named exports from the file person.js:
- import { name, age } from "./person.js";

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Modules</h1>
<p id="demo"></p>
<script type="module">
import { name, age } from "./person.js";
let text = "My name is " + name + ", I am " + age + ".";
document.getElementById("demo").innerHTML = text;
</script>
</body>
</html>
```

OUTPUT

- **JavaScript Modules**
- My name is Jesse, I am 40.

- Import from default exports
- Import a default export from the file message.js:

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Modules</h1>
<p id="demo"></p>
<script type="module">
import message from "./message.js";
document.getElementById("demo").innerHTML = message();
</script>
</body>
</html>
```

Output

- **JavaScript Modules**
- Jesse is 40 years old.

Note

- Modules only work with the HTTP(s) protocol.
- A web-page opened via the file:// protocol cannot use import / export.

1.Define Generators.

Text Book:

1.Pro MERN Stack, Full Stack Web App Development with Mongo, Express, React, and Node, Vasan Subramanian, A Press Publisher, 2019.

Reference:

David Flanagan, "Java Script: The Definitive Guide", O'Reilly Media, Inc, 7 th Edition, 2020

2. Matt Frisbie, "Professional JavaScript for Web Developers" Wiley Publishing, Inc, 4th Edition, ISBN: 978-1-119-36656-0, 2019

Thank
You!
dreamstime