

# Generator functions and yield

Absolutely! Let's explore **Generator Functions** and the `yield` keyword in JavaScript — powerful tools for writing **lazy-evaluated, pause-and-resume** functions.

## What is a Generator Function?

A **Generator Function** is a special type of function that can be **paused and resumed**, producing a **sequence of values over time**, instead of returning a single value.

### Syntax:

```
function* generatorFunction() {  
  yield value1;  
  yield value2;  
  // ...  
}
```

- `function*` defines a generator.
- `yield` is used to **pause** execution and **send a value** out.
- The generator returns an **iterator** object.

## Example: Basic Generator

```
function* greet() {  
  yield "Hello";  
  yield "Hi";  
  yield "Hey";  
}  
  
const greeter = greet();
```

```

console.log(greeter.next()); // { value: "Hello", done: false }
console.log(greeter.next()); // { value: "Hi", done: false }
console.log(greeter.next()); // { value: "Hey", done: false }
console.log(greeter.next()); // { value: undefined, done: true }

```

- Each `next()` call resumes the function from the last `yield`.
- When done, `value` becomes `undefined` and `done: true`.

## Use Case: Infinite Sequence

```

function* infiniteCounter() {
  let count = 1;
  while (true) {
    yield count++;
  }
}

const counter = infiniteCounter();
console.log(counter.next().value); // 1
console.log(counter.next().value); // 2
console.log(counter.next().value); // 3
// ...continues forever

```

This **lazy evaluation** helps in scenarios where infinite or huge data streams are involved.

## Generator vs Regular Function

Feature	Regular Function	Generator Function
Executes once	✓	✗ Can pause and resume
Returns	Single value	Many values using <code>yield</code>
State Preservation	✗	✓ Maintains internal state
Use Case	Immediate calculation	Lazy iteration, async ops

## Iterating Over a Generator

```
function* colors() {  
  yield "red";  
  yield "green";  
  yield "blue";  
}  
  
for (const color of colors()) {  
  console.log(color);  
}  
// Output: red, green, blue
```

## Passing Data Into Generator

```
function* ask() {  
  const name = yield "What is your name?";  
  console.log(`Hello, ${name}!`);  
}  
  
const gen = ask();  
console.log(gen.next().value); // "What is your name?"  
gen.next("Abhi");             // "Hello, Abhi!"
```

- You can pass data **into** a generator using `next(value)`.

## Use Case: Asynchronous Flow (with `co`, Redux Saga, etc.)

Generators were widely used **before** `async/await` for handling asynchronous flows.

Example:

```
function* asyncFlow() {
  const result = yield fetchData();
  console.log(result);
}
```

Libraries like **Redux Saga** still use generator functions to handle async side effects in a manageable way.

## ⚠ Common Mistakes

Mistake	Why It's Wrong
Using <code>return</code> instead of <code>yield</code>	<code>return</code> ends the generator
Not using <code>function*</code>	Generator won't work without the asterisk
Using <code>yield</code> outside generator	Causes syntax error



## Summary

Concept	Description
<code>function*</code>	Defines a generator function
<code>yield</code>	Pauses execution and outputs a value
<code>.next()</code>	Resumes execution until the next <code>yield</code> or <code>return</code>
State preservation	Remembers variables between yields
Use cases	Lazy evaluation, custom iterators, async control flows

Would you like to continue with **Function Composition** next or want a mini project using generators (like a step-based tutorial)?