Generators & Iterators

> **WARNING:** We looked at iteration in a previous section, so if you're rusty on it, better check it out again because they're resurfacing here with generators!

When a generator is invoked, it doesn't actually run any of the code inside the function. Instead, it creates and returns an iterator. This iterator can then be used to execute the actual generator's inner code.

```
const generatorIterator = getEmployee();
generatorIterator.next();
```

Produces the code we expect:

```
the function has started
Amanda
Diego
Farrin
James
Kagure
Kavita
Orit
Richard
the function has ended
```

Now if you tried the code out for yourself, the first time the iterator's `.next()` method was called it ran all of the code inside the generator. Did you notice anything? The code never paused! So how do we get this magical, pausing functionality?

The Yield Keyword

The `yield` keyword is new and was introduced with ES6. It can only be used inside generator functions. `yield` is what causes the generator to pause. Let's add `yield` to our generator and give it a try:

```
function* getEmployee() {
    console.log('the function has started');

    const names = ['Amanda', 'Diego', 'Farrin', 'James', 'Kagure', 'Kavita', 'Orit', 'Richard'];

    for (const name of names) {
        console.log(name);
        yield;
    }

    console.log('the function has ended');
}
```

Notice that there's now a `yield` inside the `for...of` loop. If we invoke the generator (which produces an iterator) and then call `.next()`, we'll get the following output:

```
const generatorIterator = getEmployee();
generatorIterator.next();
```

Logs the following to the console:

It's paused! But to really be sure, let's check out the next iteration:

```
generatorIterator.next();
```

Logs the following to the console:

```
Diego
```

So it remembered exactly where we left off! It took the next item in the array (Diego), logged it, and then hit the `yield` again, so it paused again.

Now pausing is all well and good, but what if we could send data from the generator back to the "outside" world? We can do this with `yield`.

Yielding Data to the "Outside" World

Instead of logging the names to the console and then pausing, let's have the code "return" the name and then pause.

```
function* getEmployee() {
    console.log('the function has started');

    const names = ['Amanda', 'Diego', 'Farrin', 'James', 'Kagure', 'Kavita', 'Orit', 'Richard'];

    for (const name of names) {
        yield name;
    }

    console.log('the function has ended');
}
```

Notice that now instead of `console.log(name);` that it's been switched to `yield name;`. With this change, when the generator is run, it will "yield" the name back out to the function *and then pause its execution*. Let's see this in action:

```
const generatorIterator = getEmployee();
let result = generatorIterator.next();
result.value // is "Amanda"

generatorIterator.next().value // is "Diego"
generatorIterator.next().value // is "Farrin"
```

QUIZ QUESTION

How many times will the iterator's `.next()` method need to be called to fully complete/"use up" the `udacity` generator function below:

```
function* udacity() {
    yield 'Richard';
    yield 'James'
}
```

- 0 times

2 times

3 times

SUBMIT

NEXT