So we can get data out of a generator by using the yield keyword. We can also send data back *into* the generator, too. We do this using the `.next()` method:

```
function* displayResponse() {
    const response = yield;
    console.log(`Your response is "${response}"!`);
}

const iterator = displayResponse();

iterator.next(); // starts running the generator function
iterator.next('Hello Udacity Student'); // send data into the generator
// the line above logs to the console: Your response is "Hello Udacity Student"!
```

Calling `.next()` with data (i.e. `.next('Richard')`) will send data into the generator function where it last left off. It will "replace" the yield keyword with the data that you provided.

So the `yield` keyword is used to pause a generator *and* used to send data outside of the generator, and then the `.next()` method is used to pass data *into* the generator. Here's an example that makes use of both of these to cycle through a list of names one at a time:

```
function* getEmployee() {
    const names = ['Amanda', 'Diego', 'Farrin', 'James', 'Kagure', 'Kavita', 'Orit', 'Richard'];
    const facts = [];

    for (const name of names) {
        // yield *out* each name AND store the returned data into the facts array
        facts.push(yield name);
    }

    return facts;
}

const generatorIterator = getEmployee();

// get the first name out of the generator
let name = generatorIterator.next().value;

// pass data in *and* get the next name
name = generatorIterator.next(`${name} is cool!`).value;

// pass data in *and* get the next name
name = generatorIterator.next(`${name} is awesome!`).value;

// pass data in *and* get the next name
name = generatorIterator.next(`${name} is stupendous!`).value;

// you get the idea
name = generatorIterator.next(`${name} is rad!`).value;
name = generatorIterator.next(`${name} is impressive!`).value;
name = generatorIterator.next(`${name} is stunning!`).value;
name = generatorIterator.next(`${name} is awe-inspiring!`).value;

// pass the last data in, generator ends and returns the array
const positions = generatorIterator.next(`${name} is magnificent!`).value;

// displays each name with description on its own line
positions.join('\n');
```

QUIZ QUESTION

What will happen if the following code is run?

```
        toppings.push(yield);
        toppings.push(yield);
        toppings.push(yield);

        return toppings;
    }

    var it = createSundae();
    it.next('hot fudge');
    it.next('sprinkles');
    it.next('whipped cream');
    it.next();
```

- The `toppings` array will have `undefined` as its last item
- An error will occur
- The generator will be paused, waiting for it's last call to `.next()`

SUBMIT

Generators are a powerful new kind of function that is able to pause its execution while also maintaining its own state. Generators are great for iterating over a list of items one at a time so you can handle each item on its own before moving on to the next one. You can also use generators to handle nested callbacks. For example, let's say that an app needs to get a list of all repositories *and* the number of times they've been starred. Well, before you can get the number of stars for each repository, you'd need to get the user's information. Then after retrieving the user's profile the code can then take that information to find all of the repositories.

Generators will also be used heavily in upcoming additions to the JavaScript language. One upcoming feature that will make use of them is async functions.

NEXT