Iteration Protocols

PROGRAMMING WITH JAVASCRIPT

Iterables

- Allows us to customise the iteration behavior of objects
- Some built-ins are iterable (Array) some are not (Object)
- To be iterable the object must have [Symbol.iterator] as a key

```
var myIterable = {
    [Symbol.iterator]: function() {
        //return object must conform to the iterator protocol
        return {...}
    }
}
```

Iterators

- The iterator is the returned object from the iterable when it's iterated, such as when it's looped over in a "for of" construct
- An iterator implements a next() method such that it returns an object with two properties:
  - done: a Boolean which returns true when it is beyond the end of the iterated sequence
  - value: Any value returned by the iterator

```
var myIterable = {
    [Symbol.iterator]: function() {
        //return object must conform to the iterator protocol
        return {
            next: function(){
                …
                if (…) {return {value: XXX,done:false } }
                return {done: true}
            }
        }
    }
}
```

3

Generators

- Generators are returned by the function* declaration
- Generators are iterators and iterables
- Generators pause execution before the first line of code runs
- Generators pause execution at each occurrence of keyword 'yield'
- 'yield' behaves much like return, but saving the position of execution

```
function* myGen() {
    let index = 0;
    while (index < 10) {yield index++;}
}
```

```
for (let num of myGen()) {
    console.log(num);
}
```

Generators: .return()

- What happens if calculate() throws an error?

```
function* myGen() {
    try {
        //yield somethings…
    } finally {
        //do some cleanup
    }
}
```

```
for (let something of myGen()) {
    calculate(something);
}
```

- The finally block runs anyway!
- Thanks to: myGen.return() being called

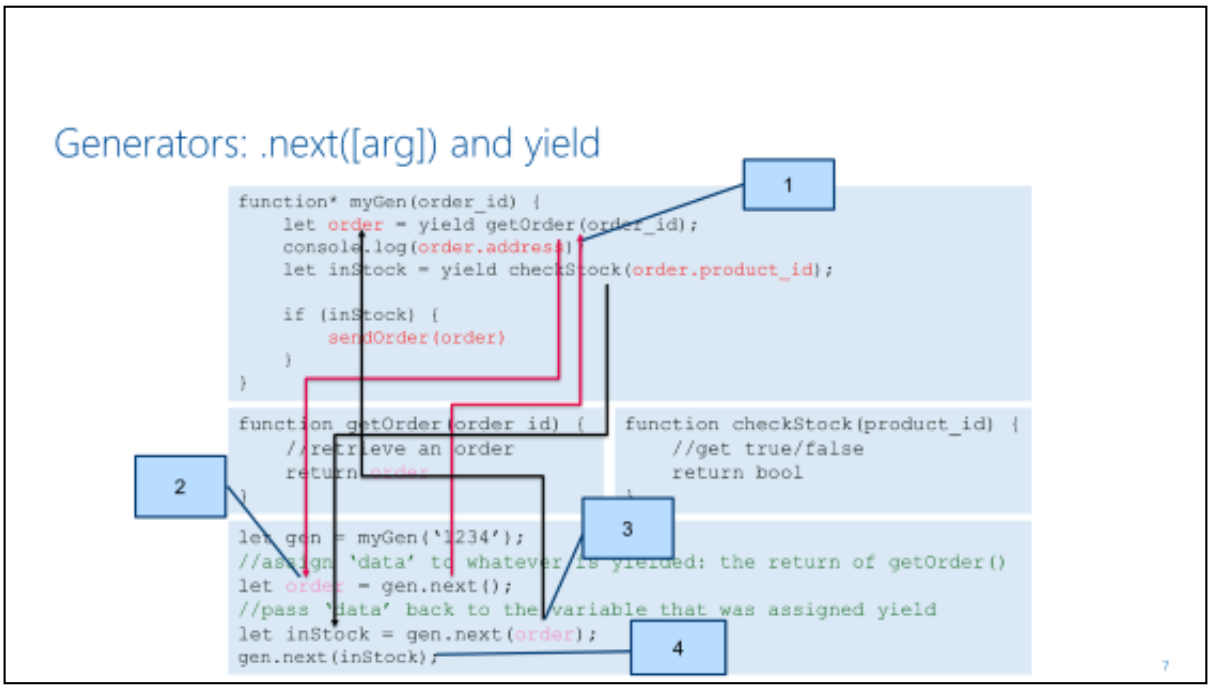
Generators: .next([arg]) and yield

- .next() takes an optional parameter which becomes the return value of the previous yield

```
function* myGen(order_id) {
    let order = yield getOrder(order_id);
    console.log(order.address);
    let inStock = yield   checkStock(order.product_id);
    if (inStock) {
        sendOrder(order)
    }
}
```

```
let gen = myGen('1234');
let order = gen.next();
let inStock = gen.next(order);
gen.next(inStock);
```

- In this case, the first .next() call passes nothing
- The second .next(order) call passes 'order' back to the 'order' variable in the generator



1) Execute getOrder(order_id) and yield its return value to (2)
2) Variable holds the return of getOrder(order_id)
3) Use .next() for the next yield of myGen() - passing in the optional parameter 'order' which becomes the value assigned to 'order' in the generator
4) Use the data to yield another result