Async Generators

Composable Async Programming in ES7

ES6 has Generator Functions

```
function *numbers() {
   let file = new FileReader(“numbers.txt”);
   try {
      while(!file.eof) {
         yield parseInt(file.readLine(), 10);
      }
   }
   finally {
      file.close();
   }
}
```

Async Functions are proposed for ES7

```
async function getStockPrice(symbol, currency) {
   let price = await getStockPrice(symbol);
   return convert(price, currency);
}
```

A Question that needs Answering

If an async function returns a Promise,

and a generator function returns an Iterator…

A Question that needs Answering

…what does an async generator function return?

```
async function*() -> ?
```

A Question that needs Answering

	Synchronous	Asynchronous
function	T	Promise
function*	Iterator	???

async function*() -> ?

An async function sends a value using a callback.

A generator functions yields multiple values, and terminates with a return value *or* an error.

async function*

An async generator function sends multiple values using callbacks, and terminates with a return value *or* an error.

async function*() -> ?

Promise<Iterable<T>> ?

async function*() -> ?

Iterable<Promise<T>> ?

Why?

- Absence creates refactoring hazard
- Support asynchronous stream composition
  - Events for web developers
  - Async IO for server developers
- Support pending features in ES7 and web platform
- Validate generator and async function design

Event Composition

```
async function *getDrags(element) {
   for(let mouseDown on element.mouseDowns) {
      for(let mouseMove on
            document.mouseMoves.
               takeUntil(document.mouseUps)) {
         yield mouseMove;
      }
   }
}
```

Sync IO with function*

```
function* getStocks() {
     let reader = new FileReader(“stocks.txt”);
     try {
          while(!reader.eof) {
               let line = reader.readLine();
               yield JSON.parse(line);
          }
     }
     finally {
          reader.close();
     }
}

function writeStockInfos() {
     let writer = new FileWriter(“stocksAndPrices.txt”);
     try {
          for(let name of getStocks()) {
               let price = getStockPrice(name);
               writer.writeLine(JSON.stringify({name, price}));
          }
     }
     finally {
          writer.close();
     }
}
```

Async IO with async function*

```
async function* getStocks() {
     let reader = new AsyncFileReader(“stocks.txt”);
     try {
          while(!reader.eof) {
               let line = await reader.readLine();
               await yield JSON.parse(line);
          }
     }
     finally {
          reader.close();
     }
}

async function writeStockInfos() {
     let writer = new AsyncFileWriter(“stocksAndPrices.txt”);
     try {
          for(let name on getStocks()) {
               let price = await getStockPrice(name);
               await writer.writeLine(JSON.stringify({name, price}));
          }
     }
     finally {
          writer.close();
     }
}
```

How does this work?

Iteration and Observation are symmetrical.

Iteration and Observation

- Share the same semantics
- Can be created/consumed using same syntax and control structures
- Can be composed using the same operators

Top-rated Movies Collection

```
let getTopRatedFilms = user =>
   user.genreLists.
      flatMap(genreList =>
         genreList.videos.
            filter(video => video.rating === 5.0));

getTopRatedFilms(user).
   forEach(film => console.log(film));
```

Mouse Drags Collection

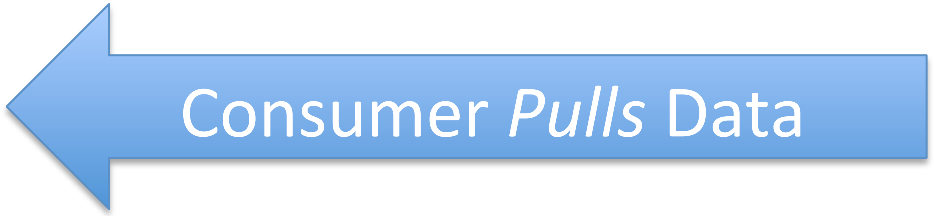
```
let getElementDrags  = elmt =>
   elmt.mouseDowns.
      flatMap(mouseDown =>
         elmt.mouseMoves.
            ~~filter~~ takeUntil(elmt.mouseUps));

getElementDrags(image).
   forEach(pos => moveTo(image, pos));
```

Iteration and Observation

The only difference is which party is in control, the consumer or the producer.

Generator Function is Iteration

Producer sends the consumer a generator, and the consumer uses it as a data source.



A Generator is a Data Source

- Can yield data
  - `generator.next().value;`
- Can throw an error and terminate
  - `try { generator.next(); } catch(e) { log(‘error’,e); }`
- Can return a value and terminate
  - `if ((pair = generator.next()).done === true) log(pair.value);`

Async Generator Function is Observation

Producer receives generator *from* consumer, and the producer uses it as a data sink.

Data *Push*ed To Consumer

A Generator is a Data Sink

- Can receive data
  - `generator.next(44);`
- Can receive an error and terminate
  - `generator.throw(“The operation did not succeed”);`
- Can receive a return value and terminate
  - `generator.return(5);`

What does an async generator function return?

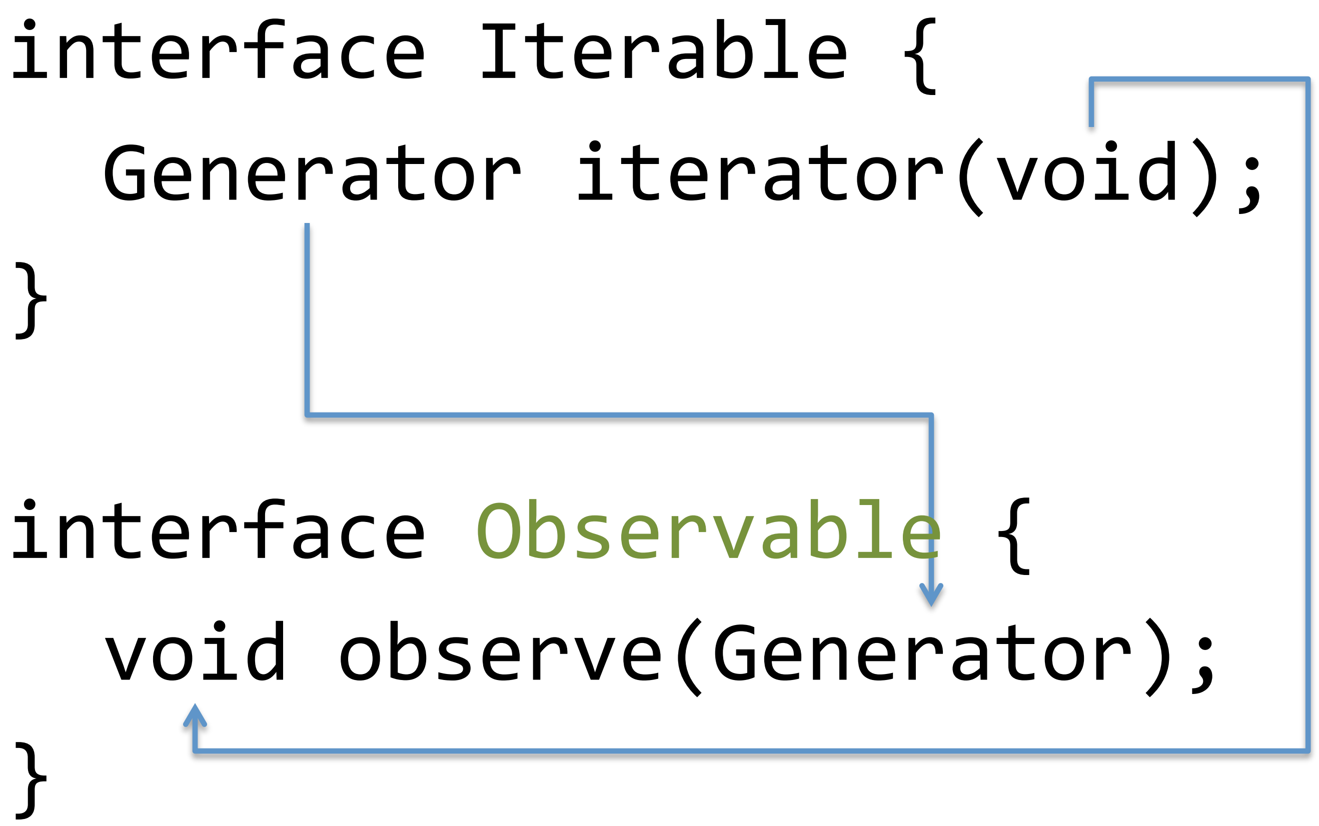
Introducing Observable

```
let nums = async function*() {
     yield 1;
     yield 2;
     return 3;
}

let numbers = nums(); // returns Observable

numbers.
     observe({
          next(v) {
               if (v === 1) {
                    return {done: true};
               }
          },
          throw(e) {
               log.error(e);
          },
          return(v) {
               log(v);
          }
     });
```

“Pushes” data to consumer

Consumer can short-circuit

Introducing Observable



```
interface Iterable {
   Generator iterator(void);
}

interface Observable {
   void observe(Generator);
}
```

How to short-circuit?

```
let nums = async function*() {
    yield 1;
    yield 2;
    return 3;
}

let numbers = nums(); // returns Observable

numbers.
    observe({
        next(v) {
            if (v === 1) {
                return {done: true};
            }
        },
        throw(e) {
            log.error(e);
        }
        return(v) {
            log(v);
        }
    });
```

Question

If the producer is in control, how can the consumer short-circuit without first getting a notification?

Short-circuiting Async Functions

```
interface Observable {
  ~~void~~ Generator observe(Generator);
}
```

Generator observe(Generator);

Detecting consumer short-circuit

- Producer adds new object to generator prototype chain
- Decorates throw and return methods
- Intercepts calls to detect consumer short-circuit

$decorateGenerator

```
function $decorateGenerator(generator, onDone) {
     let  throwFn = generator.throw,
          returnFn = generator.return;

     return Object.create(
          generator,
          {
               throw: {
                    value: function(e) {
                         onDone();
                         if (throwFn) {
                              throwFn.call(generator, e);
                         }
                    }
               },
               return {
                    value: function(v) {
                         onDone();
                         if (returnFn) {
                              returnFn.call(generator, v);
                         }
                    }
               }
          });
}
```

Short-circuiting Async Gen Function

```
let nums = async function*() {
    yield 1;
    yield 2;
    return 3;
}

let decoratedIterator =
    nums().
        observe({
            next(v) {
                log(v);
            },
            throw(e) {
                log.error(e);
            }
            return(v) {
                log(v);
            }
        });

// consumer can asynchronously short-circuit
decoratedIterator.return();
```

Async Generators Desugared

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

```
function nums() {
    return new Observable(generator => {
        let done = false,
            decoratedIterator =
              $decorateGenerator(
                    generator,
                    function onDone() { done = true; }),
           next = generator.next;

        async function() {
            try {
                if (done) return;
                decoratedIterator.next(1);
                if (done) return;
                decoratedIterator.next(2);
                if (done) return;
                decoratedIterator.return(3);
            } catch(e) {
                if (decoratedIterator.throw)
                    decoratedIterator.throw(e)
            }
        }();

        return decoratedIterator;
    });
}
```

Object.observe as an async generator

```
Object.observations = function(obj) {
    return new Observable(generator => {
        let next = generator.next,
        decoratedIterator = $decorateGenerator(generator, unobserve),
        handler = ev => { if (next) { next.call(decoratedIterator, ev); } },
        unobserve = () => Object.unobserve(obj, handler);

        Object.observe(obj, handler);

        return decoratedIterator;
    });
};
```

Generator Function

```
let numbers = function*() {
   let file = new FileReader(“numbers.txt”);
   try {
      while(!file.eof) {
         yield parseInt(file.readLine(), 10);
      }
   }
   finally {
      file.close();
   }
}
```

Async Generator Function

```
let numbers = async function*() {
   let file = new AsyncFileReader(“numbers.txt”);
   try {
      while(!file.eof) {
         yield parseInt(await file.readLine(), 10);
      }
   }
   finally {
      file.close();
   }
}
```

for…of

```
function writeNums() {
  let sum = 0;
  for(let x of numbers()) {
    sum += x;
  }
  return sum;
}
```

for…on

```
async function writeNums() {
  let sum = 0;
  for(let x on numbers()) {
    sum += x;
  }
  return sum;
}
```

for…on desugared

```
async function writeNums() {
    try {
        for(let x on numbers()) {
            log(x);
        }
        log(“completed”);
    }
    catch(e) {
        log(“error:”, e);
    }
}
```

```
function writeNums()
    return numbers().
        forEach(x => log(x)).
        then(
            () => log(“completed”),
            e => {
                log(“error:, e);
            });
}
```

Observable.forEach

```
Observable.prototype.forEach = function(next) {
   return new Promise((accept, reject) => {
      return this.observe({
         next,
         throw: reject,
         return: accept
      })
   });
}
```

An Answer to the Question

	Synchronous	Asynchronous
function	T	Promise<T>
Generator function	Iterator<T>	Observable<T>

Observable Methods

- All applicable array methods
- retry()
- takeUntil()
- Variations of flatMap
  - mergeMap()
  - concatMap()
  - switchMap()

Auto-complete

```
var searchResultSets =
   keyPresses.
      mergeMap(key =>
         getJSON("/searchResults?q=" + input.value).
            retry(3).
            takeUntil(keyPresses));

searchResultSets.forEach(
   resultSet => updateSearchResults(resultSet));
```

Nesting await expression in for…on

Should observation pause while awaiting promise?

```
async function *getStockInfos() {
    for(let stock on stocks()) {
        let price = await getPrice(stock);
        yield {name: stock.name, price};
    }
}
```

Should observation pause for await?

Yes.

"Wait" is the operative word.

Question

How do we *pause* Observation until an async operation completes?

Question

How do we *pause* ~~Observation~~ Iteration until an async operation completes?

Task.js

```
spawn(function*() {
   var data = yield $.ajax(url);
   $('#result').html(data);
   var status =
      $('#status').html('Download complete.');
   yield status.fadeIn().promise();
   yield sleep(2000);
   status.fadeOut();
});
```

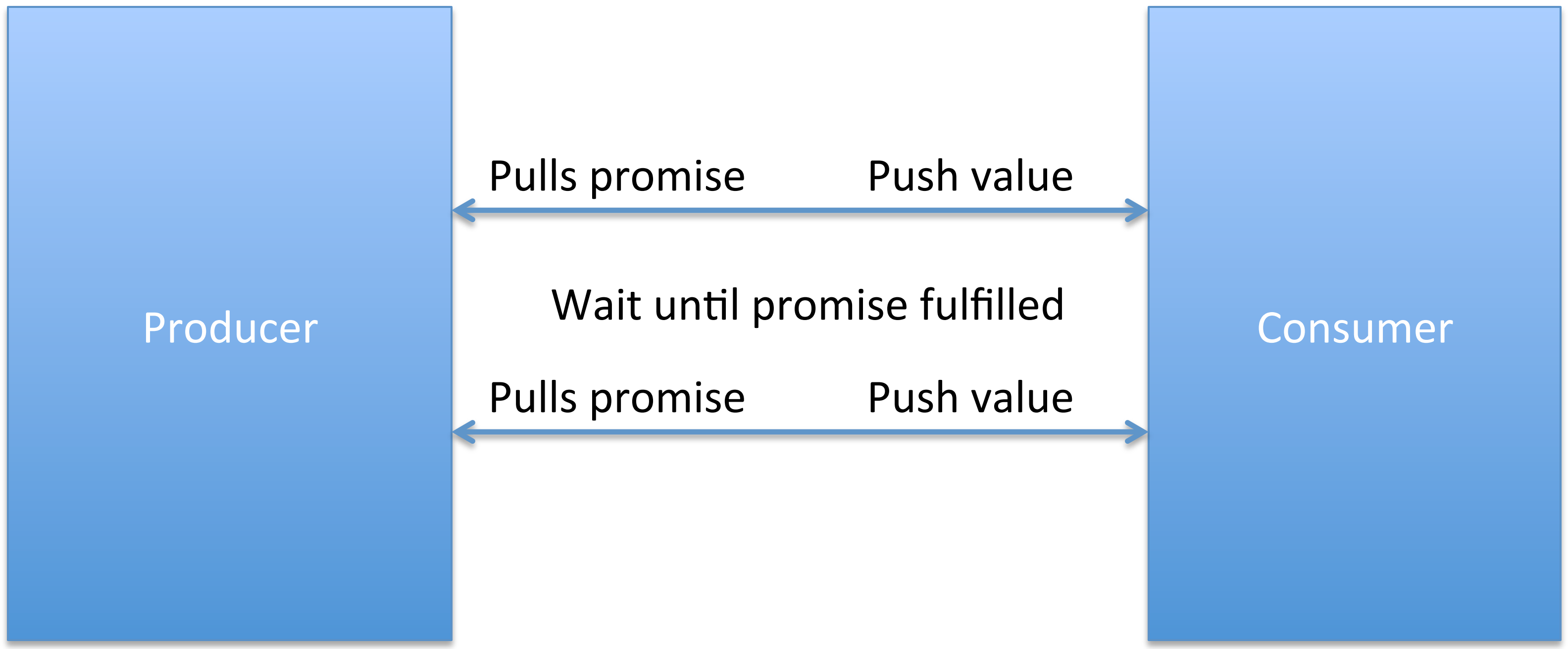
Question

How do we pause Observation until an async operation completes?

Nesting await and for…on

```
async function *getStockInfos() {
    for(let stock on stocks()) {
        let price = await getPrice(stock);
        yield {name: stock.name, price};
    }
}
```

Pausing Observation



Pause Observation while Awaiting

```
async function *getStockInfos() {
    return new Observable(generator => {
        let done,
            decoratedIterator =
                $decorateGenerator(
                    generator,
                    () => {done = true;});

        (async function() {
            try {
                await stocks().
                    forEach(async function(name) {
                        let price = await getPrice(name);
                        if (!done)
                            decoratedIterator.next({name, price});
                    });
            } catch(e) { decoratedIterator.throw(e); }
        }());

        return decoratedIterator;
    });
}
```

forEach block returns promise sub-expression from next(). Producer yield expression replaced by promise.

Async Generator

```
async function *getStocks() {
   let file = new AsyncFileReader(“stocks.txt”);
   try {
      while(!file.eof) {
         let line = await file.readLine();
         yield line;
      }
   }
   finally {
      file.close();
   }
}
```

Async Generator that Pauses

```
async function *getStocks() {
    let file = new AsyncFileReader(“stocks.txt”);
    try {
        while(!file.eof) {
            let line = await file.readLine();
            await yield line;
        }
    }
    finally {
        file.close();
    }
}
```

Result of yield is a promise!
Await result before continuing
Iteration.

Async IO with async function*

```
async function* getStocks() {
     let reader = new AsyncFileReader(“stocks.txt”);
     try {
          while(!reader.eof) {
               let line = await reader.readLine();
               await yield JSON.parse(line);
          }
     }
     finally {
          reader.close();
     }
}

async function writeStockInfos() {
     let writer = new AsyncFileWriter(“stocksAndPrices.txt”);
     try {
          for(let name on getStocks()) {
               let price = await getStockPrice(name);
               await writer.writeLine(JSON.stringify({name, price}));
          }
     }
     finally {
          writer.close();
     }
}
```

ES7 Comprehensions

```
async function writeStockInfos() {
    var stocks =
        (for (stock from PausableObservable.from(getStocks()))
            for (price from PausableObservable.from(getPrice(stock)))
                { stock, price });

    let writer = new AsyncFileWriter(“stocksAndPrices.txt”);
    try {
        for(let name on stocks) {
            let price = await getStockPrice(name);
            await writer.writeLine(JSON.stringify({name, price}));
        }
    }
    finally {
        writer.close();
    }
}
```

ES7 Comprehensions

```
var stocks =
        (for (stock from PausableObservable.from(getStocks()))
            for (price from PausableObservable.from(getPrice(stock)))
                { stock, price });

var stocks =
    PausableObservable.
        from(getStocks()).
        mergeMap(stock =>
            PausableObservable.
                from(getPrice(stock)).
                map(price => { stock, price }));
```

Validating Promises, async/await

- `await` should mirror `then` and conditionally unwrap
- Inability to cancel Promises causes friction
- Turning async generator into Promise loses cancellation semantics