Dynamic Tomorrow Ajou

웹 시스템 설계

Web System Design

17. JavaScript: Asynchronous Programming 2/2

References

- Learning JavaScript by Ethan Brown
- Mozilla Developer Network: JavaScript tutorials (https://developer.mozilla.org/ko/docs/Learn/JavaScript)

아주대학교 AJOU UNIVERSITY | 정보통신대학 College of Information Technology

Events

Events

- An (이벤트 발생을 담당) event emitter broadcasts events, and anyone who wishes to listen (or “subscribe”) to those events may do so.
- Using a callback to subscribe to an event.
- Node provides built-in support for the event system.
  - `EventEmitter`

Events (Countdown with Event)

```
const EventEmitter = require('events').EventEmitter;

class Countdown extends EventEmitter {          // 객체와 사용하는 것이 유리
   constructor(seconds, superstitious) {
      super();
      this.seconds = seconds;
      this.superstitious = !!superstitious;    // convert (falsey: 0, null, undefined) to boolean
   }
   go() {
      const countdown = this;
      return new Promise(function(resolve, reject) {
         for(let i=countdown.seconds; i>=0; i--) {
            setTimeout(function() {
               if(countdown.superstitious && i===13)
                  return reject(new Error("DEFINITELY NOT COUNTING THAT"));
               countdown.emit('tick', i);    // 이 이벤트에 대해 on('이벤트 이름', callback)으로 받음
               if(i===0) resolve();
            }, (countdown.seconds-i)*1000);
         }
      });
   }
}
```

Events (Countdown with Event)

```
const c = new Countdown(5);
c.on('tick', function(i) {
   if(i>0) console.log(i + '...');
});
c.go()
   .then(function() {
      console.log('GO!');
   })
   .catch(function(err) {
      console.error(err.message);
   })
```

```
const c = new Countdown(15, true);
c.on('tick', function(i) {
   if(i>0) console.log(i + '...');
});

c.go()
   .then(function() {
      console.log('GO!');
   })
   .catch(function(err) {
      console.error(err.message);
   })
```

Events (Countdown with Event -- fix)

```
const EventEmitter = require('events').EventEmitter;

class Countdown extends EventEmitter {
   constructor(seconds, superstitious) {
      super();
      this.seconds = seconds;
      this.superstitious = !!superstitious;
   }
   go() {
      const countdown = this;
      const timeoutIds = [];
      return new Promise(function(resolve, reject) {
         for(let i=countdown.seconds; i>=0; i--) {
            timeoutIds.push(setTimeout(function() {
               if(countdown.superstitious && i===13) {
                  // clear all pending timeouts
                  timeoutIds.forEach(clearTimeout);
                  return reject(new Error("DEFINITELY NOT COUNTING THAT"));
               }
               countdown.emit('tick', i);
               if(i===0) resolve();
            }, (countdown.seconds-i)*1000));
         }
      });
   }}
```

Promise Chain

- Promises can be chained → when one promise is fulfilled, you can have it immediately invoke another function that returns a promise…and so on.

```
function launch() {
   return new Promise(function(resolve, reject) {
         console.log("Lift off!");
         setTimeout(function() {
            resolve("In orbit!");
         }, 2*1000);
         // a very fast rocket indeed
   });
}
```

```
const c = new Countdown(5)
         .on('tick', i => console.log(i + '...'));      // event

c.go()
   .then(launch)
   .then(function(msg) {
         console.log(msg);
   })
   .catch(function(err) {
         console.error("Houston, we have a problem....");
})
```

Example: Coffee order with Callback (from meetup.toast.com)

```
function getId(phoneNumber, callback) { /* … */ }
function getEmail(id, callback) { /* … */ }
function getName(email, callback) { /* … */ }
function order(name, menu, callback) { /* … */ }

function orderCoffee(phoneNumber, function() {
    getId(phoneNumber, function(id) {
        getEmail(id, function(email) {
            getName(email, function(name) {
                order(name, 'coffee', function(result) {
                    callback(result);
                });
            });
        });
    });
});
```

Example: Coffee order with Promise (from meetup.toast.com)

```
function getId(phoneNumber) { /* … */ }
function getEmail(id) { /* … */ }
function getName(email) { /* … */ }
function order(name, menu) { /* … */ }

function orderCoffee(phoneNumber) {
    return getId(phoneNumber).then(function(id) {
        return getEmail(id);
    }).then(function(email) {
        return getName(email);
    }).then(function(name) {
        return order(name, 'coffee');
    });
}
```

Generator

Generator

- Generators are functions that use an iterator to control their execution.
  - A regular function takes arguments and returns a value, but otherwise the caller has no control of it.
  - Generators allow you to control the execution of the function.

- Generators are capable of
  - Controlling the execution of a function, having it execute in discrete steps.
  - Communicating with the function as it executes.

- Generators are like regular functions with two exceptions:
  - The function can *yield* control back to the caller at any point.
  - When you call a generator, it doesn't run right away. Instead, you get back an iterator. The function runs as you call the iterator's next method.

Generator (ECMA 6)

```
function* idMaker() {
    var index = 0;
    while(index < 3)
        yield index++;
}

var gen = idMaker(); // "Generator { }"

console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
console.log(gen.next().value); // undefined
// ...
```

function* 선언 (끝에 별표가 있는 function keyword) 은 generator function 을 정의하는데, 이 함수는 Generator 객체를 반환합니다.

Generator (ECMA 6)

```
function* anotherGenerator(i) {
  yield i + 1;
  yield i + 2;
  yield i + 3;
}
function* generator(i){
  yield i;
  yield* anotherGenerator(i);
  yield i + 10;
}
var gen = generator(10);
console.log(gen.next().value); // 10
console.log(gen.next().value); // 11
console.log(gen.next().value); // 12
console.log(gen.next().value); // 13
console.log(gen.next().value); // 20
```

Generator Example

```
function* rainbow() { // the asterisk marks this as a generator
   yield 'red';
   yield 'orange';
   yield 'yellow';
   yield 'green';
   yield 'blue';
   yield 'indigo';
   yield 'violet';
}
```

```
const it = rainbow();
it.next();          // { value: "red", done: false }
it.next();          // { value: "orange", done: false }
it.next();          // { value: "yellow", done: false }
it.next();          // { value: "green", done: false }
it.next();          // { value: "blue", done: false }
it.next();          // { value: "indigo", done: false }
it.next();          // { value: "violet", done: false }
it.next();          // { value: undefined, done: true }
// returns an object with two properties: value (which holds the
// “color” you’re now on) and done (true after the last one)
```

```
for(let color of rainbow()) {
   console.log(color):
}
```

yield Expressions and Two-Way Communication

- *two-way* communication, between a generator and its caller, happens through the **yield** expression
  - Yield is an expression that must evaluate to something (arguments provided by the caller)

```
function* interrogate() {
   const name = yield "What is your name?";
   const color = yield "What is your favorite color?";
   return `${name}'s favorite color is ${color}.`;
}
```

```
const it = interrogate();
it.next();
it.next('Ethan');
it.next('orange');
```

```
{ value: "What is your name?", done: false }
{ value: "What is your favorite color?", done: false }
{ value: "Ethan's favorite color is orange.", done: true }
```

Generators and return

- The yield statement by itself doesn't end a generator, even if it's the last statement in the generator.
- Calling return from anywhere in the generator will result in done being true, with the value property being whatever you returned.

```
function* abc() {
   yield 'a';
   yield 'b';
   return 'c';
}
```

```
const it = count();
it.next(); // { value: 'a', done: false }
it.next(); // { value: 'b', done: false }
it.next(); // { value: 'c', done: true }
```

Generator

- Generators are synchronous in nature, but when combined with promises, they offer a powerful technique for managing async code in JavaScript.

- Asynchronous Programming Dilemma
  - Asynchronous programming 을 통해 높은 성능을 달성할 수 있음
  - 그러나, 프로그래머들은 "순차적 synchronous"으로 생각함

- Question: Wouldn't it be nice if you could have the performance benefits of async without the additional conceptual difficulty?
- Since generators allow you to effectively pause code in the middle of execution, they open many possibilities related to asynchronous processing

Generator for Callback Hell Example (1/2)

```
Example was like:

dataA = read contents of 'a.txt'
dataB = read contents of 'b.txt'
dataC = read contents of 'c.txt'
wait 60 seconds
write dataA + dataB + dataC to 'd.txt'
```

```
1) Turn error-first callbacks into promises
function nfcall(f, ...args) {
   return new Promise(function(resolve, reject) {
      f.call(null, ...args, function(err, ...args) {
         if(err) return reject(err);
         resolve(args.length<2 ? args[0] : args);
      });
   });
}
```

```
2) Create timeout function for setTimeout promise
function ptimeout(delay) {
   return new Promise(function(resolve, reject) {
      setTimeout(resolve, delay);
   });
}
```

Generator for Callback Hell Example (2/2)

```
3) Create a generate runner
function grun(g) {
   const it = g();
   (function iterate(val) {
      const x = it.next(val);
      if(!x.done) {
         if(x.value instanceof Promise) {
            x.value.then(iterate).catch(err => it.throw(err));
         } else {
            setTimeout(iterate, 0, x.value);
         }
      }
   })();
}
```

```
4) Build a generator and execute!
function* theFutureIsNow() {
   const dataA = yield nfcall(fs.readFile, 'a.txt');
   const dataB = yield nfcall(fs.readFile, 'b.txt');
   const dataC = yield nfcall(fs.readFile, 'c.txt');
   yield ptimeout(60*1000);
   yield nfcall(fs.writeFile, 'd.txt', dataA+dataB+dataC);
}
grun(theFutureIsNow);
```

async / await

async function

- The async function (and await) is added from ES8(ECMAScript2017)
- The async function declaration defines an asynchronous function, which returns an AsyncFunction object. (async function 선언은 AsyncFunction객체를 반환하는 하나의 비동기 함수를 정의한다)
- An asynchronous function is a function which operates asynchronously via the event loop to return its result.
  - using an implicit Promise
- But the syntax and structure using async functions is much more like using standard synchronous functions.

async function

- An async function can contain an await expression
- An await expression 1) pauses the execution of the async function and 2) waits for the passed Promise's resolution, and then 3) resumes the async function's execution and 4) evaluates as the resolved value.
  - Note that while the async function is paused, the calling function continues running (having received the implicit Promise returned by the async function).

- 주의점: the await keyword is only valid inside async functions. If you use it outside of an async function's body, you will get a SyntaxError (i.e., await keyword는 반드시 async 함수 안에서만 쓰여야 함)
  - async 함수 안에 또 다른 일반 함수가 있고, 그 안에 await이 있으면 안됨

```
async function name([param[, param[, ... param]]]) {
   statements
}
```

```
let resolveAfter2Seconds = function() {
  console.log("starting slow promise");
  return new Promise(resolve => {
    setTimeout(function() {
      resolve(20);
      console.log("slow promise is done");
    }, 2000);
  });
};
let resolveAfter1Second = function() {
  console.log("starting fast promise");
  return new Promise(resolve => {
    setTimeout(function() {
      resolve(10);
      console.log("fast promise is done");
    }, 1000);
  });
};
let sequentialStart = async function() {
  const slow = await resolveAfter2Seconds();
  console.log(slow);
  const fast = await resolveAfter1Second();
  console.log(fast);
}
let concurrentStart = async function() {
  const slow = resolveAfter2Seconds(); // starts timer immediately
  const fast = resolveAfter1Second();
  console.log(await slow);
  console.log(await fast); // waits for slow to finish, even though fast is already done!
}
```

Example: Coffee order with Callback (from meetup.toast.com)

```
function getId(phoneNumber, callback) { /* … */ }
function getEmail(id, callback) { /* … */ }
function getName(email, callback) { /* … */ }
function order(name, menu, callback) { /* … */ }

function orderCoffee(phoneNumber, function() {
    getId(phoneNumber, function(id) {
        getEmail(id, function(email) {
            getName(email, function(name) {
                order(name, 'coffee', function(result) {
                    callback(result);
                });
            });
        });
    });
});
```

Example: Coffee order with Promise (from meetup.toast.com)

```
function getId(phoneNumber) { /* … */ }
function getEmail(id) { /* … */ }
function getName(email) { /* … */ }
function order(name, menu) { /* … */ }

function orderCoffee(phoneNumber) {
    return getId(phoneNumber).then(function(id) {
        return getEmail(id);
    }).then(function(email) {
        return getName(email);
    }).then(function(name) {
        return order(name, 'coffee');
    });
}
```

Example: Coffee order with Arrow (from meetup.toast.com)

```
function orderCoffee(phoneNumber) {
    return getId(phoneNumber)
        .then(id => getEmail(id))
        .then(email => getName(email))
        .then(name => order(name, 'coffee'));
}
```

Example: Coffee order with Generator (from meetup.toast.com)

```
function* orderCoffee(phoneNumber) {
    const id = yield getId(phoneNumber);
    const email = yield getEmail(id);
    const name = yield getName(email);
    const result = yield order(name, 'coffee');
    return result;
}
```

Example: Coffee order with Generator (from meetup.toast.com)

```
const iterator = orderCoffee('010-1234-1234');
iterator.next();

function getId(phoneNumber) {
    // …
    iterator.next(result); }
function getEmail(id) {
    // …
    iterator.next(result); }
function getName(email) {
    // …
    iterator.next(result); }
function order(name, menu) {
    // …
    iterator.next(result); }
```

Example: Coffee order with async (from meetup.toast.com)

```
async function orderCoffee(phoneNumber) {
    const id = await getId(phoneNumber);
    const email = await getEmail(id);
    const name = await getName(email);
    return await order(name, 'coffee')
}

orderCoffee('011-1234-5678').then(result => {
    console.log(result);
});
```