Hotels.com

ES6 and Functional Programming workshop

13/11/2018

Lakshmi Manasa Sai Tummala

& Joonas Tanner

What is happening?

We are at a major turning point in UI development globally.

- jQuery is dead. Direct DOM manipulation is dead. Writing unpredictable code full of “side effects”… is dead.
- Writing modern JavaScript and transpiling it down to support older browsers is the new norm for years to come. (ES6/7/8/etc)
- All about embracing new things early, instead of working off a solid “baseline” that old browsers support.
- Modern websites written with React etc (such as **Kes**) follow these principles → mastering these will help you to be productive there.
- SHP codebase in DUP will soon be fully on ES6 as well!

These are exciting times!

What is Functional Programming?

- **Functional programming** is the process of building software by composing **pure functions**, avoiding shared state, **mutable data** and side-effects.
- It is **declarative** rather than imperative (programming is done with expressions or declarations instead of statements)
- Application state flows through **pure functions**. Contrast with object oriented programming, where application state is usually shared and co-located with methods in objects.

Why Functional Programming?

- More readable code
- Encourage reusability and prevent code duplication
- Debugging is easy
- Better testability

Key concepts of Functional Programming

Data is Immutable

- You will almost never use "let" or "var" anymore; just "const"
- An object is never updated, but copied before changing it. Etc

Pure functions

- A function called multiple times with the same arguments will *always* return the same value. If you replace the function call by its return value, this doesn't cause any difference in your program; this is called *referential transparency*.
- No "side effects" occur, meaning we don't change state or rely on any variables outside a function's scope. Think about how much easier unit testing will be!

Higher-order functions

- A function that accepts other function as an argument or returns a function as a return value.

  Eg: add(4, multiply(2,3))

In simple terms…

- Avoid variable assignments

- Avoid array loops, instead
  - Transform each item – **map()**
  - Accumulate to a single value – **reduce()**
  - Conditionals – **filter()**

→ Everyone will need to master these! Practice makes perfect ☺

- Try to keep the function or block logic within itself.

- Make code concise and readable.

Example

Be *descriptive*; code is instantly understandable and no need for comments if your code is declarative enough:

```
const unavailableHotels = hotels.filter(hotel => hotel.unavailable);

// vs

var unavailableHotels = [];
for (var i=0; i < hotels.length; i++) {
    if (hotels[i].unavailable) {
        unavailableHotels.push(hotels[i]);
    }
}
```

ES6

- The 2015 version of JavaScript which is the most important change in the language, well, probably ever, or at least in a decade.

- Many great new features that support functional programming

- It’s only the beginning, there are already ES7 and ES8 specs etc. But ES6 is the most significant change.

- **Let’s embrace it to its full potential. ☺**

Variable Assignment

ES6 has made variable assignment a lot more flexible.

- Const and let
- Template Literals
- Enhanced object properties
- Spread operator
- Destructuring assignment

Constants

In ES6, we can maintain immutability by declaring variables using 'const'.

```
=> const a = 1;
   a = 2;
   output: error

=> const obj = {a:1, b:2};
   obj.b = 3;
   output: 3
```

“Const” and “let” are *block scoped*, instead of function scoped like “var” was.
Note: Variable hoisting does not work with let and const.

Template literals

Intuitive expression interpolation for single-line and multi-line strings.

Inside ${} can be any JS expression, not just a variable.

Examples

```
const color = ‘red’;
console.log(`Roses are ${color}`);

console.log(`We have ${5+ 2} roses`);

// Multiple lines
console.log(`1st line
2nd line`);
```

Enhanced Object Properties

With regards to working with objects, we have few options to assign values to properties

- Property shorthand

```
const x = 0, y = 0;
const obj = {
   x,
   y
}; // {x: 0, y: 0}
```

- Computed property names

```
const obj = {
   a: ‘bc’,
   [‘b’+1]: ‘c’
 };  // {a: "bc", b1: "c"}
```

- Method Properties

```
const obj = {
   display(a) {   // This will declare a property of name display and value as the function
      console.log(a);
   }
 };
```

Spread operator

Spreading of elements of an iterable collection (like an array or even a string) into both literal elements and individual function parameters.

- combine arrays

```
const array1 = [4,5];
const array2 = [1, 2, 3, …array1, 6, 7]; // [1,2,3,4,5,6,7]
```

- Helps us avoid loops (and maintain immutability)

```
const array1 = [1,2];
const array2 = [3,4];
const array3 = […array1, …array2]; // [1,2,3,4]
```

- Calling functions without apply

```
function showMax(a, b) { return a>b? a: c; }
const array1 = [1,2];
showMax(…array1);
```

Destructuring assignment

Intuitive and flexible destructuring of iterables into individual variables during assignment

- Array matching
  ```
  const array1 = [1,2,3];
  const [a,b,c] = array1;
  // a =1 , b=2, c=3
  ```

- Object Matching
  ```
  const obj = {a:1, b:2};
  const {a, b} = obj; // a=1, b=2
  const {a: c, b: d} = obj; // c=1, d=2
  ```

- Default values
  ```
  const obj = {a:1};
  const {a, b=2} = obj; //a=1, b=2
  const {a, b=2} = {a:1, b: 3}; //a=1,b=3
  ```

Functions

Functions in ES6 have got many options to make them more concise.

- Arrow functions
- Default parameters
- Rest parameters
- New built-in methods

Arrow functions

More expressive closure syntax to define a function.

```
// Instead of
var add = function(a, b) {
   return a + b;
}

// …we can do simply:
const add = (a, b) => a + b;

// If you need to do more complicated stuff inside the function you need to wrap with curly brackets
and have a return separately
const example = (what, ever) => {
   const bla;
   // bla, do whatever
   return bla;
}
```

Arrow functions continued

- Arrow functions do not have their own **this** value.

```
(function () {
    this.age = 0;
    setInterval(() => this.age++, 1000); // |this| properly refers to the Person object
}());
```

- No **arguments** object

```
const arr = () => console.log(arguments.length);
arr(1); // error: arguments is not defined.
```

- They cannot be used as constructors and will throw an error when used with **new**

```
const Foo = () => {};
const foo = new Foo(); // TypeError: Foo is not a constructor
```

- They do not have **prototype** property.

```
const Foo = () => {};
console.log(Foo.prototype); //undefined
```

Default parameters

Ability to have your functions initialize parameters with default values even if the function call doesn't include them.

- Static values

```
function add(a,b=1,c) {
  return a+b+c;
}
add(1,2,3); // 6
add(1, undefined, 3); // 5
add(1, null, 3); // 4
```

- Expressions

```
function addDate(day, month, year = new Date().getFullYear()) {
  return `${day}-${month}-${year}`;
}
addDate(1,2);  //1-2-2018
```

- Arguments object ignores the default values

```
console.log(arguments.length) // 2
```

Rest parameters

Aggregation of remaining arguments into single parameter of variadic functions.

- When the number of arguments are dynamic

```
function add(…numbersToAdd) {
   return numbersToAdd.reduce((sun, next) => sum + next);
}
add(1,2,3) // 6
```

- They are arrays so you can do any array operations on them like length, sort on them

```
function welcome(…people) {
   return `Welcome to ${people.sort()}`;
}
welcome('monica', 'nick', 'john’);  // Welcome to john,monica,nick
```

- They can be destructed

```
const tail = function([, ...xs]) {
   return xs;
}
tail([1,2]); // 2
```

New built-in methods

More expressive closure syntax to define a function.

- Object assignment to combine objects
```
  const dest = { a: 0};
  const src1 = {b: 1, c: 2};
  const src2 = {b: 3, d: 4};
  Object.assign(dest, src1, src2); // {a: 0, b: 3, c: 2, d: 4}
```

- Array filtering
```
   [ 1, 3, 4, 2].find(x => x > 3) // 4
   [ 1, 3, 4, 2].findIndex(x => x > 3) //2
```

- String searching
```
  “hello”.startsWith(“ello”, 1) // true
  “hello”.endsWith(“hell”, 4) // true
  “hello”.includes(“ell”) // true
  “hello”.includes(“ell”, 1) // true
  “hello”.includes(“ell”, 2) // false
```

- String repeating
```
   "foo".repeat(3) // foofoofoo
```

Iterators and Generators

A generator is function which may be paused in the middle, one or many times, and resumed later, allowing other code to run during these paused periods.

- The pause should happen from inside the function. (using '**yield**')
- The resume should happen from outside the function (using '**iterators**')
- You can have a generator function with an infinite loop
- It enables 2-way message passing into and out of the generator, as it progresses.

Use cases:

- Make asynchronous calls
- Make class iterable
- Generate UUID

Examples of Iterators and Generators

- Using next():

```
function *foo(x) {
    var y = 2 * (yield (x + 1));
    var z = yield (y / 3);
    return (x + y + z);
}
var it = foo( 5 );
console.log( it.next() );       // { value:6, done:false }
console.log( it.next( 12 ) );   // { value:8, done:false }
console.log( it.next( 13 ) );   // { value:42, done:true }
```

- Using for of:

```
function *foo() {
    yield 1;
    yield 2;
}
for (var v of foo()) {
    console.log( v );
} // 1 2
```

Classes

```
Class Parent {
    constructor(a) {
        this.value = a
     }
     displayValue() {
         console.log(‘Value displayed from parent: ’ + this.value);
      }
}

Class Child extends Parent {
     static displayStaticMethod() {
         console.log(‘static function called’);
     }
     displayValue() {
         super.displayValue();
          console.log(‘Value displayed from child: ’ + this.value);
      }
}
```

Modules

- Named Exports:
  Exporting/importing values from/to modules without global namespace pollution.
  import {sum, pi} from “lib/math”

- Default Exports:
  Marking a value as the default exported value.
  export default (x) => Math.exp(x)

Example:

```
// lib/mathplusplus.js
export * from “lib/math”
export var e = 2.71828182846
export default (x) => Math.exp(x)

// someApp.js
import exp, { pi, e } from “lib/mathplusplus”
console.log(“e^{π} =“ + exp(pi))
```

Promises

A Promise is an object that is used as a placeholder for the eventual results of a deferred (and possibly asynchronous) computation.

```
const delay = (ms) => new Promise((resolve, reject) => {
   if (true) {
       resolve(setTimeout(resolve, ms));
   }
   reject(new Error(‘rejected’));
});

delay(2)
   .then(() => {
      console.log('Resolved after 2 sec')
      return delay(1500);
   }, (error) => {
     console.log(error);
   }).catch(() => {
     console.log('caught an error')
})
```

Promise methods

- Promise.all() – resolves/rejects only after all the promises in the array are done.

```
const promise1 = new Promise((resolve, reject)  => {
    setTimeout(resolve, 500, 'one');
});
const promise2 = new Promise((resolve, reject) => {
   setTimeout(resolve, 100, 'two');
});
Promise.all([promise1, promise2])
   .then((value) => {
      console.log(value);
});
// [“one”, “two”]
```

- Promise.race() – resolves/rejects as soon as one of the promises in the array are done.

```
Promise.race([promise1, promise2])
   .then((value) => {
      console.log(value);
});
// two
```

There is SO MUCH MORE TO LEARN!
…and there is no way of covering it all in a presentation.

It is not simply learning new JS features.
Functional Programming is *a complete mindset change* and requires everyone to start thinking about and approach things differently.

…but these principles will help us be more aligned in how we approach writing JavaScript going forward!

Don’t be scared; it’s a great opportunity to embrace change and become a better UI developer - together!

Recap

- Write *pure* functions with a clear responsibility and *no side effects* to keep the code predictable and easily testable
- Keep data *immutable*
- Be *declarative* when writing JS code
- No more variables; aim to use const *always*!
- No more loops! (Use Array.map() .filter() .reduce() etc instead)
- Read on and utilise all ES6 features, such as spread, rest, destructuring etc etc
- Embrace asynchronous approach (Promises, async/await etc)
- **Keep learning, help each other, embrace change ☺**

Further reading

- A great O'Reilly book covering fundamentals of FP and also React, very well written and useful: **Learning React: Functional Web Development with React and Redux**
- https://en.wikipedia.org/wiki/Functional_programming
- https://flaviocopes.com/javascript-functional-programming
- https://opensource.com/article/17/6/functional-javascript
- https://snipcart.com/blog/functional-programming-paradigm-concepts
- https://medium.com/@kavisha.talsania/top-10-es6-features-every-javascript-developer-must-know-4c81ec54bbcd

… and the rest of the whole internet basically. :D

Thank you! ☺

Code challenge

Don’t just google it, try to think about what needs to be done, how it should be broken down to reusable functions and try to write the code yourself first! ☺

This is not an exam! It’s meant as a fun, small challenge for your own benefit. ☺

Challenge #1: Transform an array of objects

Fork this Codepen and give it a go:
https://codepen.io/anon/pen/xQELWL

Example approach - maybe it is not perfect? ☺ Discuss!
https://codepen.io/anon/pen/yQaoda