Modern JavaScript

Yakov Fain

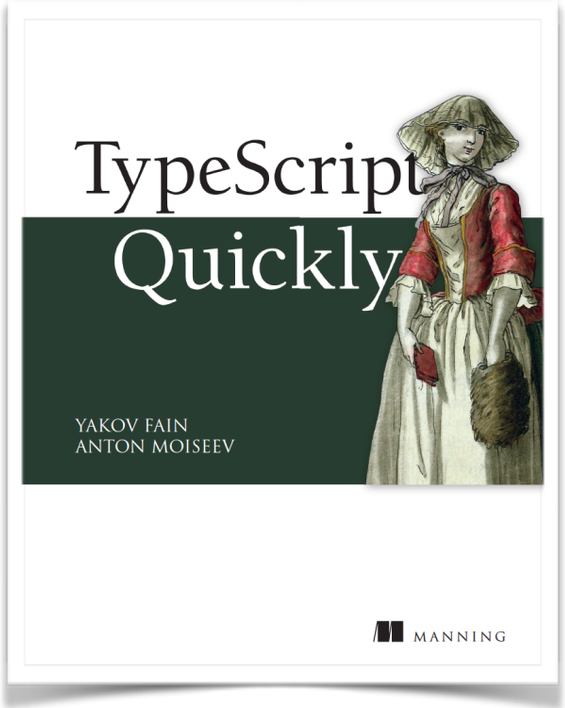
Farata Systems

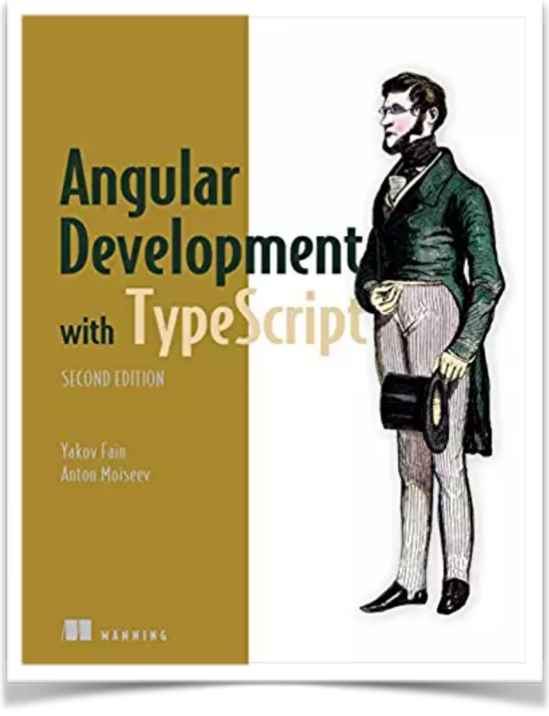
@yfain

About myself

- Work for Farata Systems
- Angular consulting and training
- Java Champion
- Recent books:

  “TypeScript Quickly”
  “Angular Development with TypeScript”,
  2nd edition





@yfain

ECMAScript

- ECMAScript (ES) is a standard for client-side scripting languages.
- ES5 was published in 2009
- ES6 a.k.a. ES2015
- ES7 a.k.a. ES2016
- ES8 a.k.a. ES2017
- ES9 a.k.a. ES2018
- ES10 a.k.a. ES2019
- ES Next

Language spec: https://tc39.github.io/ecma262

ES6 - the most significant changes

- Modules
- Classes
- arrow functions
- let and const
- destructuring
- spread and rest operators
- generator functions

How browsers support ES6

		Compilers/polyfills						Desktop browsers																	
	98%	56%	70%	70%	50%	68%	17%	5%	11%	96%	96%	96%	98%	98%	98%	98%	98%	98%	98%	98%	98%	99%	99%	99%	99%
Feature name	Current browser	Traceur	Babel 6 + core-js	Babel 7 + core-js	Closure 2018.10	TypeScript + core-js	es6-shim	Kong 4.14[1]	IE 11	Edge 17	Edge 18	Edge 19 Preview	FF 60 ESR	FF 62	FF 63	FF 64 Beta	FF 65 Nightly	CH 69, OP 56	CH 70, OP 57	CH 71, OP 58	CH 72, OP 59	SF 11.1	SF 12	SF TP	WK
Optimisation																									
proper tail calls (tail call optimisation)	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2
Syntax																									
default function parameters	7/7	4/7	4/7	4/7	5/7	5/7	0/7	0/7	0/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7
rest parameters	5/5	4/5	3/5	3/5	2/5	4/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
spread syntax for iterable objects	15/15	15/15	13/15	13/15	11/15	14/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15
object literal extensions	6/6	6/6	6/6	6/6	5/6	6/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6
for..of loops	9/9	9/9	9/9	9/9	6/9	9/9	0/9	0/9	0/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9
octal and binary literals	4/4	2/4	4/4	4/4	2/4	4/4	2/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
template literals	5/5	4/5	4/5	4/5	3/5	3/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
RegExp "y" and "u" flags	5/5	3/5	3/5	3/5	0/5	0/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
destructuring, declarations	22/22	20/22	21/22	21/22	20/22	21/22	0/22	0/22	0/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22
destructuring, assignment	24/24	23/24	24/24	24/24	22/24	24/24	0/24	0/24	0/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24
destructuring, parameters	24/24	19/24	21/24	21/24	20/24	21/24	0/24	0/24	0/24	23/24	23/24	23/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24
Unicode code point escapes	2/2	1/2	1/2	1/2	1/2	1/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
new.target	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
Bindings																									
const	18/18	16/18	14/18	14/18	16/18	14/18	0/18	2/18	14/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18
let	14/14	12/14	10/14	10/14	12/14	10/14	0/14	0/14	12/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14
block-level function declaration[17]	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Functions																									
arrow functions	13/13	11/13	9/13	9/13	10/13	9/13	0/13	0/13	0/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13
class	24/24	17/24	19/24	19/24	14/24	19/24	0/24	0/24	0/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24
super	8/8	7/8	4/8	4/8	7/8	7/8	0/8	0/8	0/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8
generators	27/27	24/27	24/27	24/27	19/27	22/27	0/27	0/27	0/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27

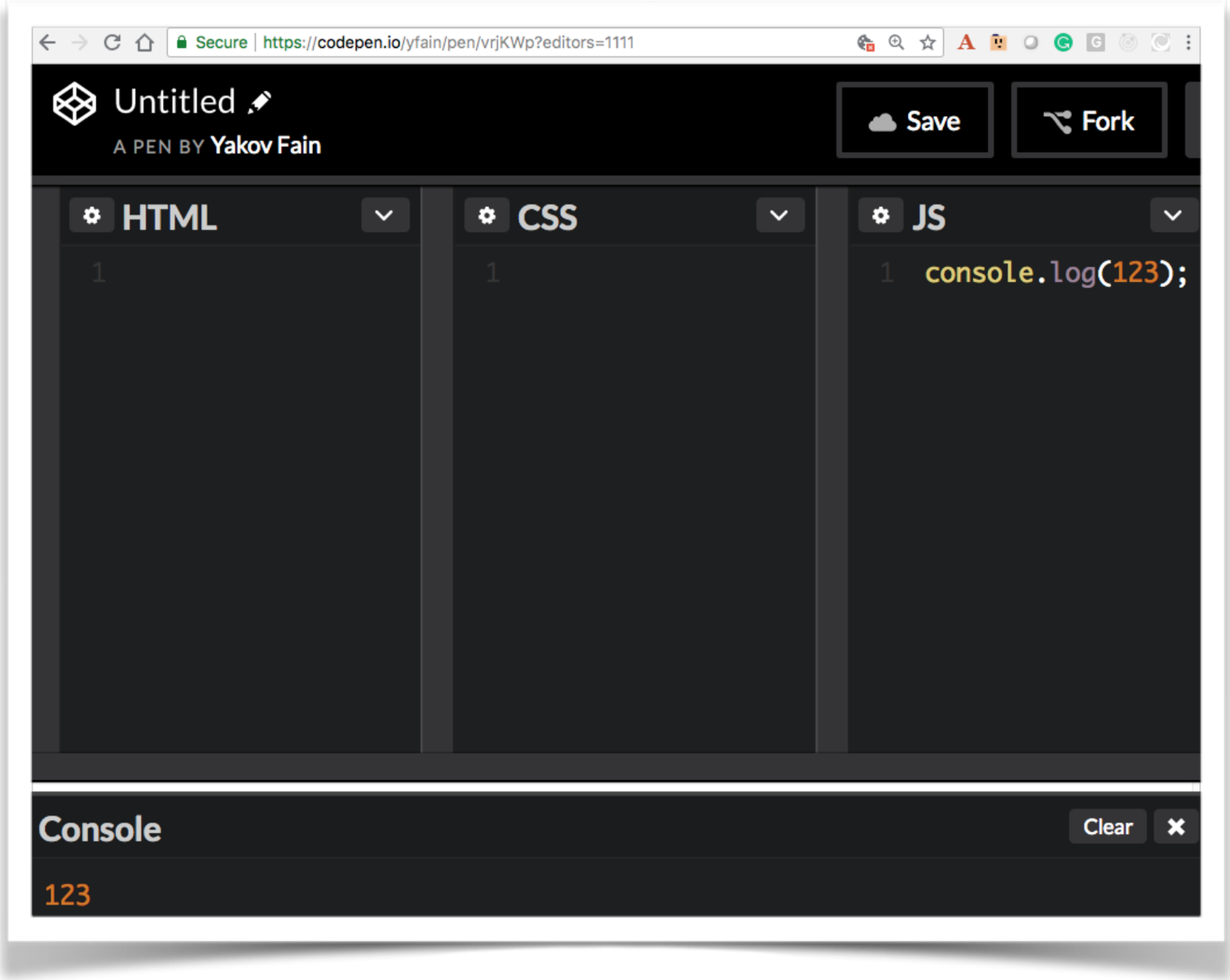
https://kangax.github.io/compat-table/es6/

For non-supporting browsers use compilers and polyfills

@yfain

How to run code samples

During this session, we'll use https://codepen.io



You can also find code samples embedded into HTML pages at https://github.com/yfain/modernJS

var, let, const

ES5: var declaration hoisting

```
function foo() {

   for (var i=0; i<10; i++) {

   }

  console.log("i=" + i);
}

foo();
```

More on hoisting: https://developer.mozilla.org/en-US/docs/Glossary/Hoisting

ES5: var declaration hoisting

```
function foo() {

    for (var i=0; i<10; i++) {

    }

  console.log("i=" + i);
}

foo();
```

Let's see
what this code will print.

@yfain

ES5: declaration hoisting (cont.)

```
var customer = "Joe";

(function () {
    console.log("The customer inside the function is "
                + customer);
 })();

console.log("The customer outside the function is "
                + customer);
```

What this code will print?

ES5: declaration hoisting

```
var customer = "Joe";

(function () {
    console.log("The customer inside the function is "
                + customer);
    if(true){
       var customer = "Mary";
    }
  })();

console.log("The customer outside the function is "
                + customer);
```

What this code will print?

ES5: declaration hoisting

```
var customer = "Joe";
(function (){
        console.log("1. The customer inside
                 the function:"  + customer);
         if (true) {
             var customer = "Mary";
         }

         console.log("2. The customer inside
                    the function: "  + customer);
    })();

    console.log("3. The customer outside
                    the function is "  + customer);
```

What this code will print?

See it in CodePen: http://bit.ly/2AwgYAC

@yfain

Block scoping with `const` and `let`

- Using **`let`** instead of **`var`** introduces a block scope
- **`const`** is like **`let`** but can be initialized only once

Block scoping

```
const customer = "Joe";

  (function () {
      console.log("The customer inside the function is " + customer);
      if (true) {
        const customer = “Mary";
        console.log("The customer inside the block is "  + customer);
      }
 })();

console.log("The customer in the global scope is "  + customer);
```

See it in CodePen: http://bit.ly/2Awwhtq

Optional params and default values

ES5

```
function calcTaxES5(income, state) {

    state = state || "Florida";

    console.log("Calculating tax for the resident of "
              + state + " with the income " + income);
}

calcTaxES5(50000);
```

Optional params and default values

ES5

```
function calcTaxES5(income, state) {

    state = state || "Florida";

    console.log("Calculating tax for the resident of "
                + state + " with the income " + income);
}

calcTaxES5(50000);
```

ES6

```
function calcTaxES6(income, state = "Florida") {

  console.log("Calculating tax for the resident of "
              + state + " with the income " + income);
}

calcTaxES6(50000);
```

See it in CodePen: http://mng.bz/U51z

Template literals

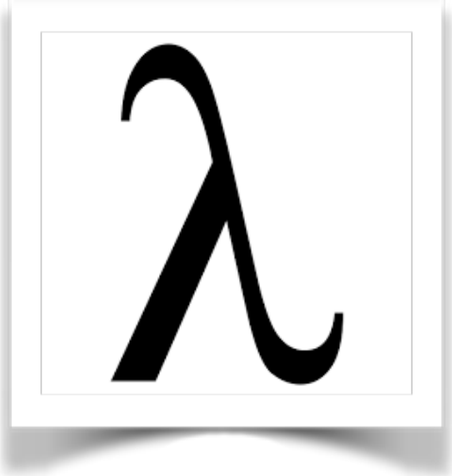
- A string can contain embedded expressions
- Use back ticks instead of quotes
- You can surround multi-line strings with back ticks

```
const customerName = "John Smith";
console.log(`Hello ${customerName}`);
```

See it in CodePen: http://bit.ly/2jlnmji

Arrow function expressions

- A short notation for anonymous functions
- Adds an intuitive lexical environment (scope) for `this`



Read about lexical environment at https://bit.ly/30r99oM

Arrow function expressions

ES6: `let sum = (arg1, arg2) => arg1 + arg2;`

ES5: `var sum = function (arg1, arg2) {return arg1 + arg2;};`

Using arrow functions as arguments

```
const myArray = [1, 2, 3, 4, 5];

console.log( "The sum of myArray elements is " +
                myArray.reduce((a,b) => a+b));


console.log( "The even numbers in myArray are " +
                myArray.filter( value => value % 2 === 0));
```

The `this` problem

```
function StockQuoteGenerator(symbol){

  this.symbol = symbol;

  setInterval(function getQuote(){
    console.log("The price of " + this.symbol
                    + " is " + Math.random());
  }, 1000);
}

const stockQuoteGenerator = new StockQuoteGenerator("IBM");
```

What this code will print?

This and that solution

```
function StockQuoteGenerator(symbol){

  const that = this;
  that.symbol = symbol;

  setInterval(function getQuote(){
    console.log("The price of " + that.symbol
                    + " is " + Math.random());
  }, 1000);
}

const stockQuoteGenerator = new StockQuoteGenerator("IBM");
```

See it in CodePen http://bit.ly/2A8PAbj

An arrow function solution

```
function StockQuoteGenerator(symbol) {
      this.symbol = symbol;

      setInterval( () => {
                console.log("The price of " + this.symbol
                        + " is " + Math.random());
            }, 1000);
}

const stockQuoteGenerator = new StockQuoteGenerator("IBM");
```

The rest operator ...

- Represents a variable number of arguments in a function
- Turns a variable number of arguments into an array
- Has to be the last in the arguments list

The rest operator ...

```
function calcTax(income, ...customers) {

    console.log(`Calculating tax for customers with the income
                ${income}`);

    customers.forEach(
            (customer) => console.log(`Processing ${customer}`));
}

calcTax(50000, "Smith", "Johnson", "McDonald");
calcTax(750000, "Olson", "Clinton");
```

See it in CodePen http://bit.ly/2hT3H9Y

The spread operator ...

Turns an array into a list of values

The spread operator ...

Adding elements of a second array to the first one:

```
array1.push(...array2);
```

The spread operator ...

Adding elements of a second array to the first one:

```
array1.push(...array2);
```

Creating a copy of an array

```
let arrayCopy = [...myArray];
```

The spread operator ...

Adding elements of a second array to the first one:

```
array1.push(...array2);
```

Creating a copy of an array

```
let arrayCopy = [...myArray];
```

Finding a maximum value in an array

```
const maxValue = Math.max(...myArray);
```

Cloning with `Object.assign()`

```
const myObject = {name: "Mary" , lastName: "Smith"};

// Cloning
const clone = Object.assign({}, myObject);

// Clone with modifying the lastName
const cloneModified = Object.assign(myObject, {lastName: "Lee"});
```

Cloning with the spread operator

```
const myObject = {name: "Mary" , lastName: "Smith"};

// Cloning
const cloneSpread = {...myObject};

// Clone with modifying the lastName
const cloneSpreadModified = {...myObject, lastName: "Lee"};
```

See it in CodePen http://bit.ly/2A60PBb

Destructuring

- Destructuring means taking an object apart
- An object may have more properties than you're interested in

Destructuring an object

```
function getStock(){

    return {
        symbol: "IBM",
        price: 100.00,
        shares: 1200000
    };
}

let {symbol, price} = getStock();

console.log(`The price of ${symbol} is ${price}`);
```

See it in CodePen http://bit.ly/2iTIEWU

Destructuring in the browser

```
 <button id="myButton">Click me</button>
...

document
  .getElementById("myButton")
  .addEventListener("click",
       ({target}) => console.log("The target value is", target));
```

See it in CodePen http://bit.ly/2n2JzYB

Destructuring of arrays

- Instead of curly braces use square brackets
- Specify arbitrary variable names to match array's indexes

```
let [name1, name2] = ["Smith", "Clinton"];

let [, name2] = ["Smith", "Clinton"];
```

Component's state in React.js:

```
const [userName, setUserName] = useState('John');
```

Combining destructuring and the rest operator

```
let customers = ["Smith", "Clinton", "Lou", "Gonzales"];

let [firstCust, secondCust, ...otherCust] = customers;

console.log(`The first customer is ${firstCust}
             and the second one is ${secondCust}`);

console.log(`Other customers are ${otherCust}`);
```

Classes and inheritance

ES5:

```
function Tax() {...}

function NJTax() {...}

NJTax.prototype = new Tax();

var njTax = new NJTax();
```

ES6:

```
class Tax {...}

class NJTax extends Tax {...}

let njTax = new NJTax();
```

Class constructors

- A class constructor is a method that's executed once during the object creation

- Use the keyword `constructor`

```
class Tax {

  constructor(income) {
    this.income = income;
  }
}

let myTax = new Tax(50000);
```

Subclasses and constructors

If a subclass doesn't declare its own constructor, the one from the superclass is invoked

```
class Tax {
    constructor(income) {
        this.income = income;
    }
}

class NJTax extends Tax {
    // The code specific to New Jersey tax goes here
}

let njTax = new NJTax(50000);

console.log(`The income in njTax instance is ${njTax.income}`);
```

super and super()

```
class Tax {
     constructor(income) {
        this.income = income;
     }
     calculateFederalTax() {
         console.log(`Calculating federal tax for income ${this.income}`);
     }
     calcMinTax() {
         console.log("In Tax. Calculating min tax");
         return 123;
     }
 }

 class NJTax extends Tax {
    constructor(income, stateTaxPercent) {
        super(income);
        this.stateTaxPercent=stateTaxPercent;
    }
    calculateStateTax() {
        console.log(`Calculating state tax for income ${this.income}`);
    }
    calcMinTax() {
         let minTax = super.calcMinTax();
         console.log(`In NJTax. Will adjust min tax of ${minTax}`);
    }
 }
```

@yfain

super and super()

```
class Tax {
     constructor(income) {
        this.income = income;
     }
     calculateFederalTax() {
         console.log(`Calculating federal tax for income ${this.income}`);
     }
     calcMinTax() {
         console.log("In Tax. Calculating min tax");
         return 123;
     }
 }

 class NJTax extends Tax {
    constructor(income, stateTaxPercent) {
        super(income);
        this.stateTaxPercent=stateTaxPercent;
    }
    calculateStateTax() {
        console.log(`Calculating state tax for income ${this.income}`);
    }
    calcMinTax() {
         let minTax = super.calcMinTax();
         console.log(`In NJTax. Will adjust min tax of ${minTax}`);
    }
 }
```

```
const theTax = new NJTax(50000, 6);

theTax.calculateFederalTax();
theTax.calculateStateTax();
theTax.calcMinTax();
```

See it in CodePen http://bit.ly/2joJkBW

@yfain

Class fields (TC39, stage 3)

BABEL

SETTINGS

- [ ] Evaluate
- [x] Line Wrap
- [ ] Prettify
- [ ] File Size
- [ ] Time Travel

Source Type

Module

PRESETS

- [ ] es2015
- [ ] es2015-loose
- [ ] es2016
- [ ] es2017
- [ ] stage-0
- [ ] stage-1
- [ ] stage-2
- [x] stage-3

v7.9.0

```
1  class Rectangle {
2    _height = 0; // before: naming convention
3    #width;      // private field
4
5    constructor(height, width) {
6      this._height = height;
7      this.#width = width;
8    }
9  }
10
11 const rec = new Rectangle(2,3);

console.log(rec._height);

15 console.log(rec.#weight);   // compile error
```

```
/repl.js: Private name #weight is not defined (15:16)

  13 | console.log(rec._height);
  14 |
> 15 | console.log(rec.#weight);   // compile error
     |                 ^
```

Try it: https://bit.ly/3cEWo0h

@yfain

Static class members

are shared between class instances

Static: sharing the value between instances

```
class A {
  static counter = 0;

  printCounter(){
    console.log("static counter=" + A.counter);
  };
}

const a1 = new A();
A.counter++;
a1.printCounter();   // prints 1

A.counter++;

const a2 = new A();
a2.printCounter();   // prints 2

console.log("On the a1 instance, counter is " + a1.counter);
console.log("On the a2 instance, counter is " + a2.counter);
```

See it in CodePen: https://codepen.io/yfain/pen/VGBjvR?editors=0012

Static methods

```
class Helper {

  static convertDollarsToEuros() {

    console.log("Converting dollars to euros");
  }

  static convertCelsiusToFahrenheit() {

    console.log("Converting Celsius to Fahrenheit");
  }

}

Helper.convertDollarsToEuros();
Helper.convertCelsiusToFahrenheit();
```

See it in CodePen: https://codepen.io/yfain/pen/VGBjRL?editors=0011

Asynchronous calls

From callbacks to promises to async/await

ES5: Callbacks

```
(function getProductDetails() {

    setTimeout(function () {
        console.log('Getting customers');
        setTimeout(function () {
            console.log('Getting orders');
            setTimeout(function () {
                console.log('Getting products');
                setTimeout(function () {
                    console.log('Getting product details')
                }, 1000);
            }, 1000);
        }, 1000);
    }, 1000);
})();
```

Asynchronous callbacks

See it in CodePen http://bit.ly/2Be8vyo

ES6: Promises

- The `Promise` object represents an **eventual completion or failure** of an async operation.
- If the operation ends successfully, the `then()` method is called
- If it fails, the `catch()` method is called
- Makes code more readable by eliminating callback nesting

ES6: Promises

```
function getCustomers(){

    return new Promise( function (resolve, reject) {

            console.log("Getting customers");
            setTimeout(function() {

                const success = true;

                if (success) {
                    resolve( "John Smith"); // got customer
                } else {
                    reject("Can't get customers");
                }
            },1000);

            });
}
```

ES6: Promises

```
function getCustomers(){

        return new Promise( function (resolve, reject) {

          console.log("Getting customers");
          setTimeout(function() {
               const success = true;
               if (success) {
                   resolve( "John Smith"); // got customer
               } else {
                   reject("Can't get customers");
               }
          },1000);

        });
}
```

```
getCustomers()
    .then(cust => console.log(cust))
    .catch(err => console.error(err));
```

@yfain

Chaining Promises

```
function getCustomers(){

    return new Promise(
        function (resolve, reject){

            console.log("Getting customers");
            setTimeout(function(){
                const success = true;
                if (success){
                    resolve( "John Smith");
                }else{
                    reject("Can't get customers");
                }
            },1000);
        }
    );
}
```

```
function getOrders(customer){

    return new Promise(
        function (resolve, reject){

        console.log("Getting orders");
        setTimeout(function(){
            const success = true;
            if (success){
              resolve( `Found the order 123 for ${customer}`);
            }else{
              reject("Can't get orders");
            }
        },1000);
        }
    );
}
```

```
getCustomers()
    .then(cust => getOrders(cust))
    .then(order => console.log(order))
    .catch(err => console.error(err));
```

See it in CodePen http://bit.ly/2ABGeWj

@yfain

Aggregating results of multiple promises

- If promises don't depend on each other, run them in parallel with `Promise.all()`
- You'll get back an array of results after all promises are resolved

```
Promise.all([getWeather(),
             getStockMarketNews(),
             getTraffic()])
.then( (results) => { /* render UI */ })
.catch(err => console.error(err)) ;
```

ES8: async-await

- Treat functions returning promises as if they're synchronous
- The next line is executed only when the previous one completes
- Waiting for the async code to complete doesn't block the rest of the program
- No callbacks; no `then()` either
- Error handling: `try-catch` blocks

ES8: async-await

- `async` - marks an asynchronous function
- `await` - wait until the asynchronous code completes

ES8: async-await

```
async function getCustomersOrders(){

  try {
      const customer = await getCustomers();

      console.log(`Got customer ${customer}`);

      const orders = await getOrders(customer);

      console.log(orders);

  } catch(err){
      console.log(err);
  }

}
```

Both `getCustomers()` and `getOrders()` return promises

See it in CodePen http://bit.ly/2k4wYmC

Parallel async calls

```
function add5(x) {
    return new Promise(resolve => {
      setTimeout(() => {
        resolve(x +5);
      }, 3000);
    });
  }

  async function calcInParallel(x) {
    const firstCall = add5(20);
    const secondCall = add5(40);

    return x + (await firstCall) + (await secondCall);
  }

  console.time('time spent');
  calcInParallel(10)
     .then(result => {console.log( result);
                      console.timeEnd('time spent')}
          );
```

@yfain

Parallel async calls

```
function add5(x) {
    return new Promise(resolve => {
      setTimeout(() => {
        resolve(x +5);
      }, 3000);
    });
  }

  async function calcInParallel(x) {
    const firstCall = add5(20);
    const secondCall = add5(40);

    return x + (await firstCall) + (await secondCall);
  }

  console.time('time spent');
  calcInParallel(10)
     .then(result => {console.log( result);
                      console.timeEnd('time spent')}
          );
```

Console output

```
80
time spent: 3004.9560546875ms
```

@yfain

ES6: generator functions

- A JS engine executes a regular function from start to end without interruptions
- The execution of a generator function can be paused and resumed multiple times
- A generator function can yield control to the calling script
- Useful for handling streams of data

yield and next

```
function* doSomething(){

  console.log("Started processing");

  yield;

  console.log("Resumed processing");
}

let iterator = doSomething(); // returns the Generator obj

iterator.next();  // Starts executing the function body
```

Generators: Handling data streams

Retrieve stock prices and buy the stock if the price falls below a limit price

```
function* getStockPrice(symbol){

  while(true){
    yield Math.random()*100; // an AJAX call could go here

    console.log(`resuming for ${symbol}`);
  }
}
let priceGenerator = getStockPrice("IBM");

const limitPrice = 15;
let price = 100;

while ( price > limitPrice){

   price = priceGenerator.next().value;
   console.log (`The generator returned ${price}`);
}

console.log(`buying at ${price} !!!`);
```

See it in CodePen: https://bit.ly/2tD2Pfd

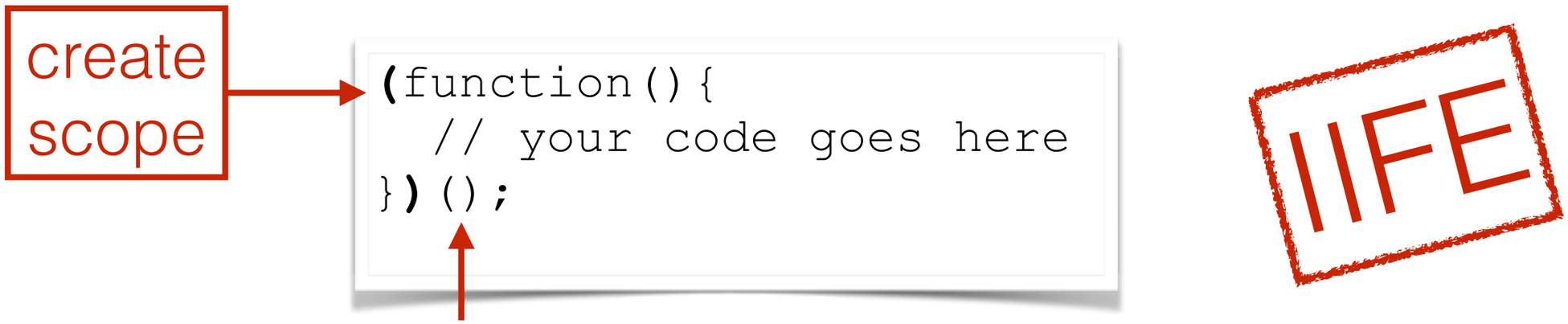
Code modularization

- Avoid monolithic apps
- Easier to maintain
- Easier to test
- Control which module members are exposed to other modules and which ones are hidden

Creating modules in ES5

- Implement module design pattern yourself
- Use third-party libraries that support module standards (e.g. AMD or CommonJS)

Immediately Invoked Function Expression



- A function that runs as soon as it's defined
- The external grouping operator **()** prevents accessing variables within IIFE as polluting global scope

Implementing Module design pattern

named scope

```
var shippingModule = (function(){

})();
```

Implementing Module design pattern

private

```
var shippingModule = (function(){

    var applyDiscount = function(custName, discount){
      console.log("Giving " + discount + "% off to " + custName);

    }})();


shippingModule.applyDiscount('Smith', 25);  // error
```

Implementing Module design pattern

```
var shippingModule = (function(){

    var applyDiscount = function(custName, discount){
      console.log("Giving " + discount + "% off to " + custName);
    }

    return {
       calcDiscount: function (custName) {
         if ("Soprano" === custName ){
           applyDiscount(custName, 50);
         } else {
           applyDiscount(custName, 10);
         }
       }
    }
})();

shippingModule.calcDiscount('Soprano');
shippingModule.calcDiscount('Smith');
```

Exposed to outer scope

See it in CodePen http://bit.ly/2ia7SSE

ES6 Modules

- A script becomes a module if it uses `import` and/or `export` keywords
- Export a constant, variable, function, or class
- A module can import other modules
- Only exported module members are visible outside of the module; other members remain encapsulated

Not a module

person.js

```
class Person {

}

let p = new Person(); // created on global scope
```

An ES module

person.js

```
export class Person {

}
```

ES6 modules don't pollute global scope

person.js

```
export class Person {

}
```

Modules have top-level lexical scope, e.g.
the variable p is created **not** on the global scope (even with `var`)

main.js

```
import {Person} from './person.js';
...
var p = new Person();
```

@yfain

Named exports

Allow you to export multiple module members

tax.js

```
export let taxCode = 1;
export function calcTaxes() { }
function doSomethingElse() { }   // private
export function fileTaxes() { }
```

Named exports

Allow you to export multiple module members

tax.js

```
export let taxCode = 1;
export function calcTaxes() { }
function doSomethingElse() { }   // private
export function fileTaxes() { }
```

main.js

```
import {taxCode, calcTaxes} from './tax.js';
...
if (taxCode === 1) { // do something }

calcTaxes();
```

Default exports

One of the exported module members can be marked as default.

Another module can give it any name in its `import` statement.

tax.js

```
export default function() { // do something }

export let taxCode;
```

main.js

```
import coolFunction, {taxCode} from './tax.js';

coolFunction();
```

Loading ES modules in the browsers

`<script type="module">` is supported by all major browsers

JavaScript modules via script tag - LS

Usage
Global

Loading JavaScript module scripts using `<script type="module">`
Includes support for the `nomodule` attribute.

Current aligned | Usage relative | Date relative | Apply filters | Show all | ?

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android
	12-14										
	16 15	2-53	4-59	3.1-10	10-46	3.2-10.2					
	6 16-18	2 54-59	1 60	4 5 10.1	1 47	4 5 10.3					
6-10	79	60-74	61-80	11-12.1	48-66	11-13.2		2.1-4.4.4	12-12.1		
11	80	75	81	13	67	13.3	all	80	46	80	68
		76-77	83-85	13.1-TP		13.4					

@yfain

A demo from GitHub dir modules/esmodules

index.html

```
<!DOCTYPE html>
<head>
    <title>My modules</title>
</head>
<body>
  <h1>Hello modules (see console)!</h1>

  <script type="module" src="./main.js"></script>
</body>
</html>
```

A demo from GitHub dir modules/esmodules

index.html

```
<!DOCTYPE html>
<head>
    <title>My modules</title>
</head>
<body>
  <h1>Hello modules (see console)!</h1>

  <script type="module" src="./main.js"></script>
</body>
</html>
```

main.js

```
import {ship} from './shipping.js';

ship();
```

shipping.js

```
export function ship() {
 console.log("Shipping products...");
}

function calculateShippingCost(){
 console.log("Calculating shipping cost");
}
```

https://plnkr.co/edit/TY9AQJhJPQKAigGu

@yfain

A fallback for `type="module"`

If a browser supports `type="module"`,
it ignores the `nomodule` attribute

```
<script type="module" src="./main.js"></script>
<script nomodule src="./main_fallback.js"></script>
```

What if my code has to run in older browsers?

http://babeljs.io

BABEL

Docs Setup Try it out Videos Blog Search Donate Team GitHub

SETTINGS

- [ ] Evaluate
- [x] Line Wrap
- [ ] Minify
- [ ] Prettify
- [ ] File Size
- [ ] Time Travel

Source Type

Module

PRESETS

- [x] es2015
- [ ] es2015-loose
- [ ] es2016
- [ ] es2017
- [ ] stage-0
- [ ] stage-1
- [x] stage-2
- [ ] stage-3
- [x] react
- [ ] flow
- [ ] typescript

OPTIONS

Decorators mode Current Proposal

Decorators before export

ENV PRESET

PLUGINS

```
1  class Tax {
2      constructor(income) {
3         this.income = income;
4      }
5
6      calculateFederalTax() {
7          console.log(`Calculating federal tax for income
   ${this.income}`);
8      }
9
10     calcMinTax() {
11         console.log("In Tax. Calculating min tax");
12         return 123;
13     }
14 }
15
16 class NJTax extends Tax {
17    constructor(income, stateTaxPercent) {
          super(income);
          this.stateTaxPercent=stateTaxPercent;
      }
21
22    calculateStateTax() {
23        console.log(`Calculating state tax for income
   ${this.income}`);
24    }
25
26     calcMinTax() {
27         let minTax = super.calcMinTax();
28         console.log(`In NJTax. Will adjust min tax of ${minTax}`);
29     }
30 }
31
32 const theTax = new NJTax(50000, 6);
33
34 theTax.calculateFederalTax();
```

```
1  "use strict";
2
3  function _typeof(obj) { if (typeof Symbol === "function" && typeof
   Symbol.iterator === "symbol") { _typeof = function _typeof(obj) {
   return typeof obj; }; } else { _typeof = function _typeof(obj) {
   return obj && typeof Symbol === "function" && obj.constructor ===
   Symbol && obj !== Symbol.prototype ? "symbol" : typeof obj; }; }
   return _typeof(obj); }
4
5  function _possibleConstructorReturn(self, call) { if (call &&
   (_typeof(call) === "object" || typeof call === "function")) {
   return call; } return _assertThisInitialized(self); }
6
7  function _assertThisInitialized(self) { if (self === void 0) {
   throw new ReferenceError("this hasn't been initialised - super()
   hasn't been called"); } return self; }
8
9  function _get(target, property, receiver) { if (typeof Reflect !==
   "undefined" && Reflect.get) { _get = Reflect.get; } else { _get =
   function _get(target, property, receiver) { var base =
   _superPropBase(target, property); if (!base) return; var desc =
   Object.getOwnPropertyDescriptor(base, property); if (desc.get) {
   return desc.get.call(receiver); } return desc.value; }; } return
   _get(target, property, receiver || target); }
10
11 function _superPropBase(object, property) { while
   (!Object.prototype.hasOwnProperty.call(object, property)) { object
   = _getPrototypeOf(object); if (object === null) break; } return
   object; }
12
13 function _getPrototypeOf(o) { _getPrototypeOf =
   Object.setPrototypeOf ? Object.getPrototypeOf : function
   _getPrototypeOf(o) { return o.__proto__ ||
   Object.getPrototypeOf(o); }; return _getPrototypeOf(o); }
14
15 function _inherits(subClass, superClass) { if (typeof superClass
```

@yfain

What TypeScript brings to the table?

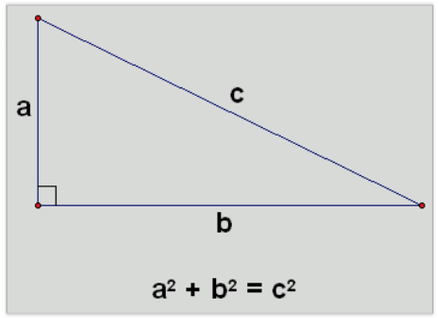
TypeScript

- It’s a programming language that compiles to JavaScript
- JavaScript + types + great IDE support
- It catches errors during compile-time
- It’s a superset of JavaScript

The Pythagorean Theorem

```
function distance(pointA, pointB) {
  return Math.sqrt(
    (pointA.x - pointB.x) ** 2 +
    (pointA.y - pointB.y) ** 2);
}

console.log(distance({ z: 3, y: -4 }, {x:6, y:0}));
```



Anything wrong with this code?

Playground: https://bit.ly/2vMUHKH

@yfain

Summary

- If you start a new project, don't use ten year old syntax; write it in ES.Next
- Introduce a transpiler in your workflow to use the modern JavaScript today
- To increase the productivity of writing JavaScript, consider writing in TypeScript

Links!

- Code samples: https://github.com/yfain/modernJS
- Blog: yakovfain.com
- email: yfain@faratasystems.com