Full-stack Application Development
using ReactJS, Express and TypeScript

About me

Trayan Iliev

– CEO of **IPT – Intellectual Products & Technologies**
http://www.iproduct.org

– Oracle® certified programmer 15+ Y

– end-to-end reactive fullstack apps with Java, ES6+,
TypeScript, Angular, React and Vue.js

– 12+ years IT trainer: Spring, Java EE, Node.js, Express,
GraphQL, SOA, REST, DDD & Reactive Microservices

– Voxxed Days, jPrime, Java2Days, jProfessionals,
BGOUG, BGJUG, DEV.BG speaker

– Organizer RoboLearn hackathons and IoT enthusiast

Why to Attend This Training?

What will you learn?

The Course Main Goals

The course provides in-depth study of state-of-the-art *TypeScript, ReactJS, NodeJS,* and *Express* platform for rapid development of modern, component-based, mobile-first, responsive full-stack web applications, that are easy to extend and maintain.

You Will Learn How To:

- Set up different kinds of TypeScript projects – front-end with React, and back-end with NodeJS and Express;

- Build reusable React components using elegant functional syntax and TSX

- Integrate them in Single Page Applications with routing using React Router

- Code splitting and lazy loading with React.lazy()

- Performance optimization using immutability and memoising

- Implement form handling and validation using Material UI, react-hook-form and Yup

- Handle cross-cutting concerns using React Context

You Will Learn How To:

- Build type-safe components, single page apps, and backend services;

- Integrate them in full-stack projects using REST/JSON APIs using Material UI, React Router, Express and MySQL

- Practice development of type safe React components, single page apps, and backend services;

- Implement real-time server-push event streaming using Socket.IO;

- Understand how to boost development performance with TypeScript, and different tooling alternatives;

- Debug full-stack applications written in TypeScript

Course Topics and Homeworks - I

1. Quick recap of HTML5, CSS and JavaScript - 4 h

2. HTTP Client and Fetch APIs - 4 h - Homework 1: Building a Google Books API search simple JavaScript application

3. Novelties in ES6+ - 5 h

4. TypeScript - 6 h - Homework 2: Refactoring Google Books API search application using TypeScript

5. Building React components and Single Page Applications (SPA) with TypeScript – 6 h - Homework 3 : Building React Comments client side application using TypeScript and react-markdown

6. TSX in depth – 6 h

Course Topics and Homeworks - II

7. New lifecycle methods – 2h

8. Building purely functional components using new Hooks API – 4 h - Homework 4 : Building Google Books SPA to use functional components and hooks. Using effects and rendering performance optimization with react hooks

9. Working with forms. Using component libraries - 6 h - Homework 5 : Refactoring Google Books API search SPA using React, react-hook-form, Yup and Material-UI. Adding bookmarks and book annotation features

10. Components composition in depth – 4h

11. Dependency Injection (DI) of custom application services using React Context – 2 h - Homework 6 : Choosing a fullstack course project using React, Express and MySQL project specification

Course Topics and Homeworks - III

12. SPA routing using React Router v6 – 8 h - Homework 7 : Adding routing to Google Books SPA using React Router v6, Homework 8 : Developing initial structure with routing and navigation between features, for the chosen course project using React and React Router v6

13. Server side application development with Node.js and TypeScript - 4 h - Homework 9 : Building a custom Users REST API using Node.js and TypeScript

14. REST service API development using Express.js -14h - Homework 10 / 1 1 : Developing the initia l prototype / final course project using React, Express and MySQL
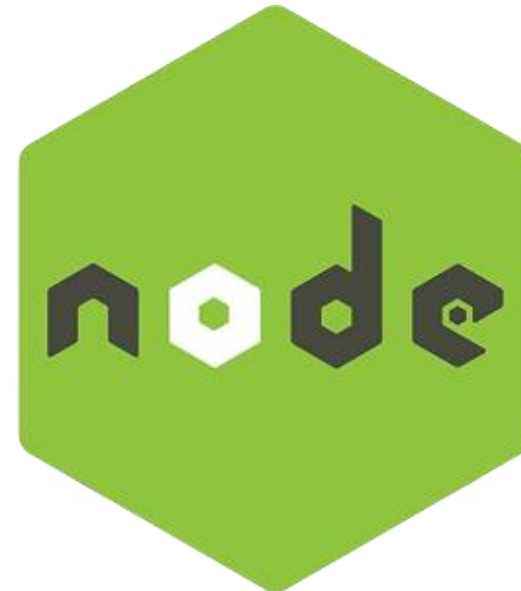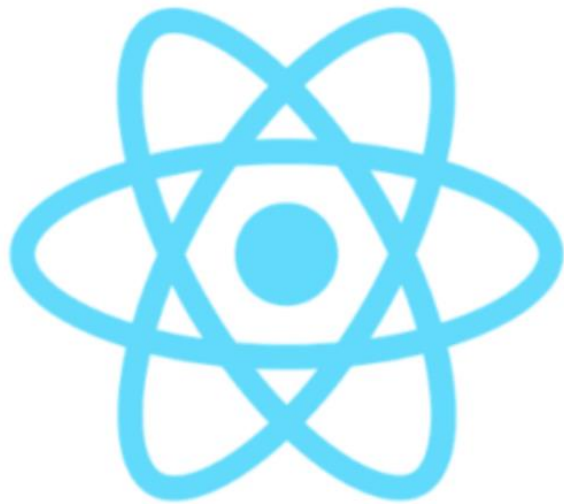
Course Dates:

- 24, 27, 28, 31 October 2022,

- 3, 4, 7, 10, 11, 14, 17, 18, 21, 24, 25, 28 November 2022,

- 1, 2, 5, 8, 9, 12, 15, 16, 19, 22, 23 December 2022,

- 5, 6, 9 January 2023

Time Schedule

- 18:30 – 19:45 – Block 1

- 19:45 – 20:00 – Break

- 20.00 – 21.15 – Block 2

# Where to Find The Code and Materials?

https://github.com/iproduct/react-typescript-academy-2022

Software to Install

- NodeJS - https://nodejs.org/en/download/current/

- Visual Studio Code - https://code.visualstudio.com/download

- MySQL Community Server - https://dev.mysql.com/downloads/mysql/
  Don't forget to install MySQL Workbench too -
  https://dev.mysql.com/downloads/workbench/

- MongoDB Community Server -
   https://www.mongodb.com/try/download/community
  Don't forget to install MongoDB Compass (GUI) too.

- Yarn package manager for NodeJS -
   https://classic.yarnpkg.com/en/docs/install/#windows-stable

- Chrome Web Browser

Visual Studio Code
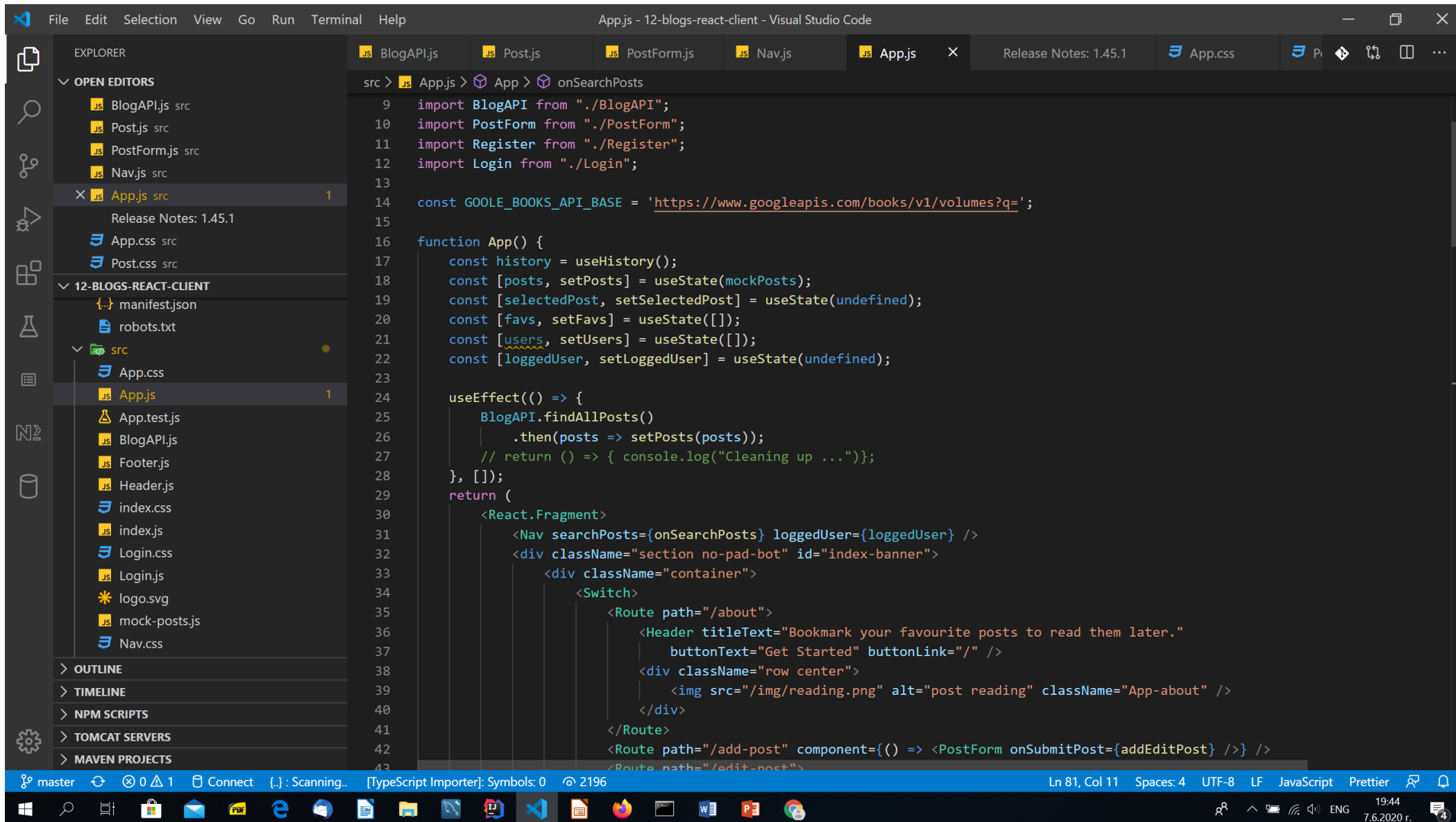
Introduction to HTML5, CSS & JavaScript

Web design technologies recap
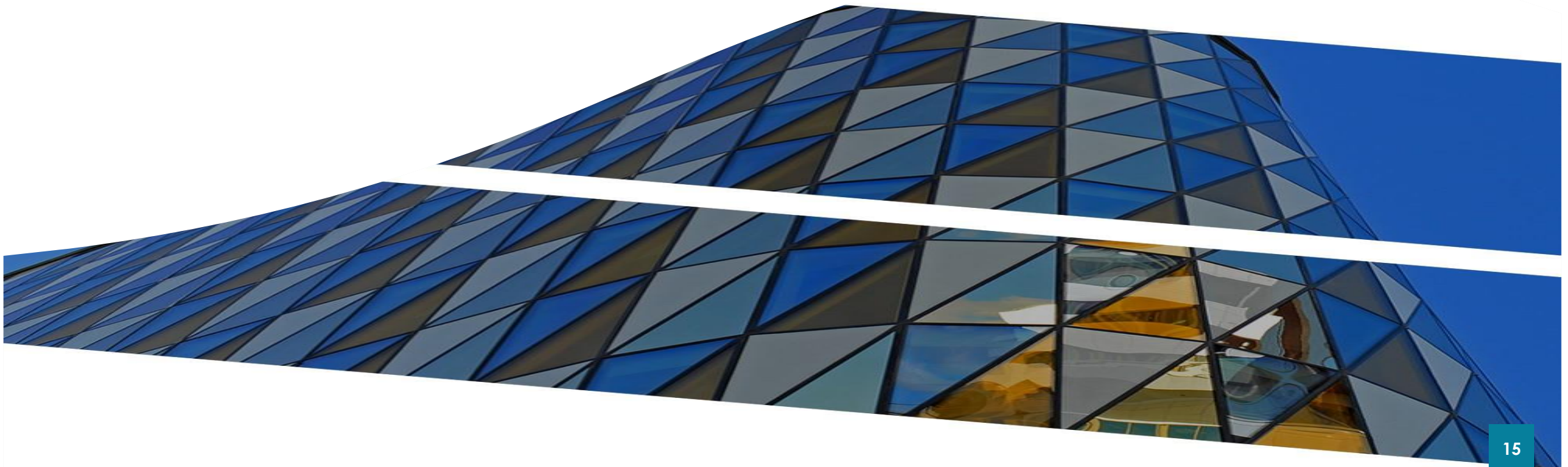
Introduction to HTML 5

Base tags, semantic markup, styling

We will follow the [HTML 5 Tutorials at W3Schools](#):

- HTML Elements

- Styles and formatting

- Integration with CSS

- Basic building blocks: links, images, tables, lists

- Block and inline elements

- Ids and classes. Basic CSS selectors

- Adding behaviour with JavaScript

- Basic layouts with HTML and CSS

- Semantic markup

We will follow the [HTML 5 Tutorials at W3Schools](#):

- Charsets and URL encoding

- HTML forms and validation

- HTML Graphics with Canvas and SVG

- HTML 5 Media API - <audio>, <video>

- HTML 5 Geolocation API

- Drag and drop support

- Web Storage API – localStorage and sessionStorage

Quiz Time

https://www.w3schools.com/quiztest/quiztest.asp?qtest=HTML

Exercises

https://www.w3schools.com/html/exercise.asp

Introduction to CSS

Selectors, positioning, backgrounds, box model and layout, grid system, Flexbox

We will follow the CSS Tutorials at W3Schools:

- Selectors

- Positioning

- Colours

- Gradients

- Backgrounds

- Borders

- Box model

- Text and fonts

- Horizontal and vertical alignment

We will follow the CSS Tutorials at W3Schools:

- Complex selectors

- Pseudo-classes and pseudo-elements

- How-to's: navigation bar, dropdowns, gallery

- Forms styling

- Counters

- CSS units

- Selectors specificity

- Web page layout using CSS floating

- Animations and transforms. Styling images

We will follow the [CSS Tutorials at W3Schools](#):

- CSS variables (custom properties) - https://developer.mozilla.org/en-US/docs/Web/CSS/Using_CSS_custom_properties

- calc() Function - https://developer.mozilla.org/en-US/docs/Web/CSS/calc, https://css-tricks.com/creating-color-themes-with-custom-properties-hsl-and-a-little-calc/

- Responsive web design (RWD) and CSS *@media* rule and media queries

- Flexbox - https://css-tricks.com/snippets/css/a-guide-to-flexbox/

https://css-tricks.com/almanac/properties/f/flex/

https://css-tricks.com/understanding-flex-grow-flex-shrink-and-flex-basis/

- Responsive page layout using Flexbox

- CSS Grid Layout - https://css-tricks.com/snippets/css/complete-guide-grid/, https://css-tricks.com/auto-sizing-columns-css-grid-auto-fill-vs-auto-fit/

Quiz Time

https://www.w3schools.com/css/css_quiz.asp

Exercises

https://www.w3schools.com/css/exercise.asp

Where to Find The Code and Materials?

https://github.com/iproduct/react-typescript-academy-2022
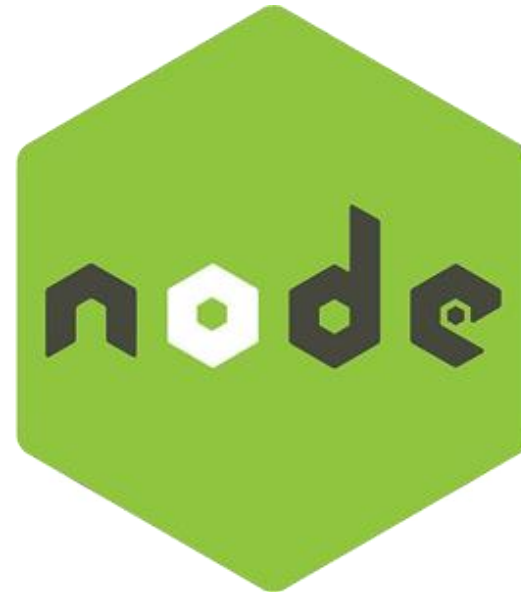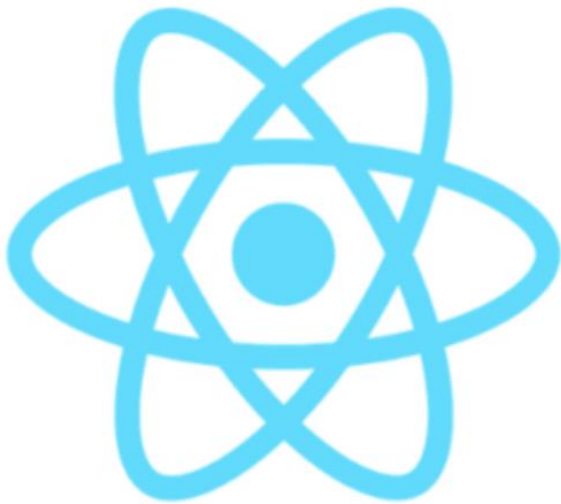
Brief History of JavaScript™

- JavaScript™ created by Brendan Eich from Netscape for less then 10 days!

- Initially was called Mocha, later LiveScript – Netscape Navigator 2.0 - 1995

- December 1995 Netscape® и Sun® agree to call the new language JavaScript™

- "JS had to 'look like Java' only less so, be Java's dumb kid brother or boy-hostage sidekick. Plus, I had to be done in ten days or something worse than JS would have happened."

B. E. (http://www.jwz.org/blog/2010/10/every-day-i-learn-something-new-and-stupid/#comment-1021)

The Language of Web

- JavaScript™ success comes fast. Microsoft® create own implementation called JScript to overcome trademark problems. JScript was included in Internet Explorer 3.0, in August 1996.

- In November 1996 Netscape announced their proposal to Ecma International to standardize JavaScript → ECMAScript

- JavaScript – most popular client-side (in the browser) web programming language („de facto" standard) and one of most popular programming languages in general.

- Highly efficient server-side platform called Node.js based on Google V8 JS engine, compiles JS to executable code Just In Time (JIT) during execution (used at the client-side also).

Object-Oriented JavaScript

Three standard ways to create objects in JavaScript:

- Using object literal:

  var newObject = { };

- Using Object.create(prototype[, propertiesObject]) (prototypal)

  var newObject = Object.create(Object.prototype);

- Using constructor function (pseudo-classical)

  var newObject = new Object();

Object Properties

- Object-Oriented (OO) – object literals and constructor functions

- Objects can have named properites

Ex.: **myObject.name = 'Scene 1';**
myObject ['num-elements'] = 5;
MyObject.protopype.toString = function() {
return "Name: " + this.name + ": " + this['num-elements'] }

- Configurable object properties – e.g. read only, get/set, etc.

Ex.: **Object.defineProperty(newObject, "someKey", {**
value: "fine grained control on property's behavior",
writable: true, enumerable: true, configurable: true
});

Property Getters and Setters

Ex.:

```
function PositionLogger() {
    var position = null,   positionsLog = [];
    Object.defineProperty(this, 'position', {
        get: function() {
            console.log('get position called');
            return position;
        },
        set: function(val) {
            position = val;
            positionsLog.push({ val: position });
        }
    });
    this.getLog = function() { return positionsLog; };
}
```

JavaScript Features

- The state of objects could be changed using JS functions stored in object's **prototype**, called **methods**.

- Actually in JavaScript **there were no real classes**, - only objects and constructor functions before ES6 (ES 2015, Harmony).

- JS is **dynamically typed language** – new properties and methods can be added runtime.

- JS supports object inheritance using prototypes and mixins (adding dynamically new properies and methods).

- **Prototypes** are objects (which also can have their prototypes) → inhreritance = traversing prototype chain

- Main resource: **Introduction to OO JS YouTube video** https://www.youtube.com/watch?v=PMfcsYzj-9M

JavaScript Features

- Supports **for ... in** operator for iterating object's properties, including inherited ones from the prototype chain.

- Provides a number of predefined datatypes such as:
  Object, Number, String, Array, Function, Date etc.

- Dynamically typed – variables are universal containers, no variable type declaration.

- Allows dynamic script evaluation, parsing and execution using **eval()** – **discouraged as a bad practice**.

Datatypes in JavaScript

- Primitive datatypes:
  - **boolean** – values **true** и **false**
  - **number** – floating point numbers (no real integers in JS)
  - **string** – strings (no char type –> string of 1 character)

- Abstract datatypes:
  - **Object** – predefined, used as default prototype for other objects (defines some common properties and methods for all objects: **constructor, prototype**; methods: **toString(), valueOf(), hasOwnProperty(), propertyIsEnumerable(), isPrototypeOf()**;)
  - **Array** – array of data (really dictionary type, resizable)
  - **Function** – function or object method (defines some common properties: **length, arguments, caller, callee, prototype**)

Datatypes in JavaScript

- Special datatypes:
  - **null** – special values of object type that does not point anywhere
  - **undefined** – a value of variable or argument that have not been initialized
  - **NaN** – Not-a-Number – when the arithmetic operation should return numeric value, but result is not valid number
  - **Infinity** – special numeric value designating infinity ∞

- Operator **typeOf**

  Example: **typeOf myObject.toString** //-->'function'

Problem 1: OOP in JavaScript

Create a JS file **employees.js** that implements the following functionality:

- Create constructor **Employee(name, experienceInYears, qualifications)**. The **qualifications** should be an **array of strings**.

- Override the method **toString()** from base class Object in order to return as string all provided employee object attributes.

- Create **two employees** using the implemented constructor, and print their state to the **console** (using **console.log()**)

New Array Methods in ECMAScript 5 (1)

- Introduces in JavaScript 1.6 (ECMAScript Language Specification 5.1th Edition - ECMA-262) – November 2005

- **indexOf (searchElement[, fromIndex])** – returns the index of **first** occurrence of the *searchEleement* element in the array

- **lastIndexOf (searchElement[, fromIndex])** – returns the index of **last** occurrence of the *searchEleement* element in the array

- **every(callback[, thisObject]))** – calls the boolean result *callback function* for each element in the array till callback returns false, if callback returns true for each element => every returns true

- *Ex:* **function isYoung(value, index, array) { return value < 45; }**

    var areAllYoung = [41, 20, 17, 52, 39].every(isYoung);

New Array Methods in ECMAScript 5 (2)

- **some(callback[, thisObject]))** – calls the boolean result *callback function* for each element in the array till callback returns true, if callback returns false for each element => some returns false

- *Ex:* **function isYoung(value, index, array) { return value < 45; }**

    var isSomebodyYoung = [41, 20, 17, 52, 39].some(isYoung);

- **filter(callback[, thisObject])** – calls the boolean result *callback function* for each element in the array, and returns new array of **only** these elements, for which the predicate (callback) is true

- *Ex:* **function isYoung(value, index, array) { return value < 45; }**

    var young = [41, 20, 17, 52, 39].filter(isYoung);

    // returns [41, 20, 17, 39]

New Array Methods in ECMAScript 5 (3)

- **map(callback[, thisObject]))** – calls the *callback function* for **each** element of the array, and returns new array with containing the **results** returned by *callback function*

- *Ex:* **function nextYear(value, index, array) { return value + 1;}**
  var newYearAges = [41, 20, 17, 52, 39].map(nextYear);
  // returns [42, 21, 18, 53, 40]

- **forEach(callback[, thisObject])** – executes the *callback function* for each element in the array

- *Ex:* **function print(value, index, array) { console.log(value) }**
  [41, 20, 17, 52, 39].filter(isYoung).map(ageNextYear).forEach(print);
  // prints in console: 42, 21, 18 и 40

New Array Methods in ECMAScript 5 (4)

- **reduce(callback[, initialValue])** – applies *callback function* for an *accumulator variable* and for *each of the array elements* (left-to-right) – reducing this way the array to a *single value (the final accumulator value)*, returned as a result.

- **reduceRight(callback[, initialValue])** – the same but right-to-left

- *Ex:* **function sum(previousValue, currentValue, index, array) {**

      return previousValue + currentValue; }

    var result = [41, 20, 17, 52, 39]

      .filter(isYoung).map(ageNextYear).reduce(sum, 0);

    console.log("Sum = ", result); *//* prints: Sum = 121

Functional JavaScript

- Functional language – functions are "first class citizens"

- Functions can have own properties and methods, can be assigned to variables, pass as arguments and returned as a result of other function's execution.

- Can be called by reference using operator **()**.

- Functions can have embedded inner functions at arbitrary depth

- All arguments and variables of outer function are accessible to inner functions – even after call of outer function completes

- Outer function = enclosing context (Scope) for inner functions → **Closure**

Closures

Example:

```
function countWithClosure() {
    var count = 0;
    return function() {
        return count ++;
    }
}


var count = countWithClosure(); <-- Function call – returns innner
        function wich keeps reference to count variable from the outer scope
console.log( count() );        <-- Prints 0;
console.log( count() );        <-- Prints 1;
console.log( count() );        <-- Prints 2;
```

Default Values & RegEx

- Functions can be called with different number of arguments. It is possible to define default values – Example:

```
function Polygon(strokeColor, fillColor) {
    this.strokeColor = strokeColor || "#000000";
    this.fillColor = fillColor || "#ff0000";
    this.points = [];
    for (i=2;i < arguments.length; i++) {
        this.points[i] = arguments[i];
}
```

- Regullar expressions – Example: **/a*/**.match(str)

Object Literals. Using this
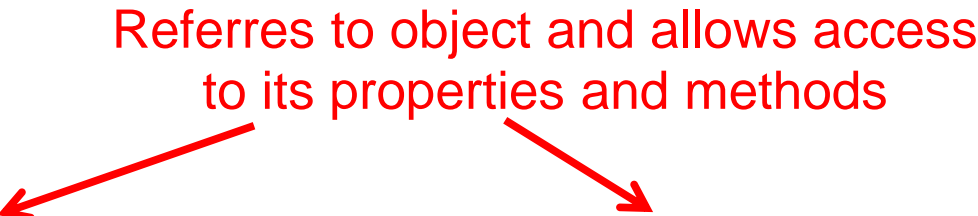
- Object literals – example:

var point1 = { x: 50, y: 100 }

var rectangle1 = { x: 200, y: 100, width: 300, height: 200 }

- Using **this** calling a function /D. Crockford/ - „Method Call":

```
var scene1 = {
    name: 'Scene 1',
    numElements: 5,
    toString: function() {
        return "Name: " + this.name + ", Elements: " + this['numElements'] }
}
console.log(scene1.toString())   // --> 'Name: Scene 1, Elements: 5'
```

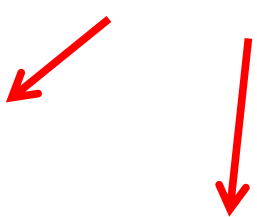Referres to object and allows access to its properties and methods

Accessing this in Inner Functions

- Using **this** calling a function /D. Crockford/ - „Function Call":

```
var scene1 = {
  ...
  log: function(str) {
    var self = this;
    var createMessage = function(message) {
        return "Log for '" + self.name +"' („ + Date() + "): "
  + message;
    }
    console.log( createMessage(str) );
}
```

It's necessary to use additional variable, because *this* points to global object (window) *undefined* in strict mode

„Classical" Inheritance, call() apply() & bind()

- Pattern „Calling a function using special method"
  Function.prototype.apply(thisArg, [argsArray])
  Function.prototype.call(thisArg[, arg1, arg2, ...])
  Function.prototype.bind(thisArg[, arg1, arg2, ...])

```
function Point(x, y, color){
    Shape.apply(this, [x, y, 1, 1, color, color]);
}
extend(Point, Shape);
function extend(Child, Parent) {
    Child.prototype = new Parent;
    Child.prototype.constructor = Child;
    Child.prototype.supper = Parent.prototype;
}
```

„Classical" Inheritance. Using call() & apply()

```
Point.prototype.toString = function() {

    return "Point [" + this.supper.toString.call( this ) + "]";

}

Point.prototype.draw = function(ctx) {

    ctx.fillStyle = this.fillColor;

    ctx.fillRect(this.x, this.y, 1, 1);

}

point1 = new Point(200,150, „blue");

console.log(point1.toString() );
```

„Classical" Inheritance. Using call() & apply()

```javascript
Point.prototype.toString = function() {
    return "Point [" + this.supper.toString.apply( this, [] ) + "]";
}
Point.prototype.draw = function(ctx) {
    ctx.fillStyle = this.fillColor;
    ctx.fillRect(this.x, this.y, 1, 1);
}
point1 = new Point(200,150, „blue");
console.log(point1.toString() );
```

JavaScript Design Patterns

- **Software design patterns** gained popularity after the book Design Patterns: Elements of Reusable Object-Oriented Software [1994], GoF: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

- **Def: Software design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design

- **Proven solutions** – proven techniques that reflect the experience and insights the developers

- **Easily reused** – out of the box solutions to common problems

- **Expressiveness** – define common vocabulary and structure

JavaScript Design Patterns

- Prototype (Object.create() / Object.assign())

- Constructor (using prototypes)

- Singleton (literals, lazy instantiation)

- Module

- Observer (publish/subscribe events)

- Dynamic loading of JS modules

- DRY (Don't Repeat Yourself)

- Command

- Facade

- Factory

- Mixin

- Decorator

- Function Chaining

Examples Using JavaScript Design Patterns

Learning JavaScript Design Patterns

A book by Addy Osmani:

https://addyosmani.com/resources/essentialjsdesignpatterns/book/

JS Design Patterns: Prototype

- Intent: creates objects based on a template of an existing object through cloning: Object.create(prototype[, propertiesObject])

```
function Person(first, last, age, gender, interests) {
        // property and method definitions
        this.name = { 'first': first, 'last' : last };
        this.age = age; this.gender = gender;
        //... more properties here
}
let person1 = new Person('Bob', 'Smith', 32, 'male', ['music', 'skiing']);
```

JS Design Patterns: Constructor

- Intent: **constructor** is a special function used to initialize properties of a new object once memory allocated

```
function Vehicle( model, year, kilometers ) {
  this.model = model;
  this.year = year;
  this.kilometers = kilometers;
  this.toString = function () {
    return this.model + " (" + this.year + ") has travelled "
          + this.kilometers + " kilometers";
  };
}
var focus = new Vehicle( "Ford Focus", 2010, 90000 );
var jazz = new Vehicle( "Honda Jazz", 2005, 170000 );
```

Better solution is to place the object methods in the prototype instead of making copies for each instance

JS Design Patterns: Module

- Intent: Group several related elements, such as singletons, properties and methods, into a single conceptual entity.

- A portion of the code must have global or public access and be designed for use as global/public code. Additional private or protected code can be executed by the main public code.

- A module must have an initializer/finalizer functions that are equivalents to, or complementary to object constructor/ destructor methods

- In JavaScript, there are several options for implementing modules: Module pattern, as Object literal, AMD modules, CommonJS modules, ECMAScript Harmony modules (ES Modules)

JS Design Patterns: Singleton

- Intent: Ensure a class has only one instance, and provide a global point of access to it.

- Object literals **{ }** in JavaScript are a natural way to implement Singletons

- Often Singletons are lazily initialized, like:

```
getInstance: function( myOptions ) {

    if( instance === undefined ) {

        instance = new MySingleton( myOptions );

    }

    return instance;

}
```

JS Design Patterns: Observer (Publish/Subscribe)

- Intent: Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.
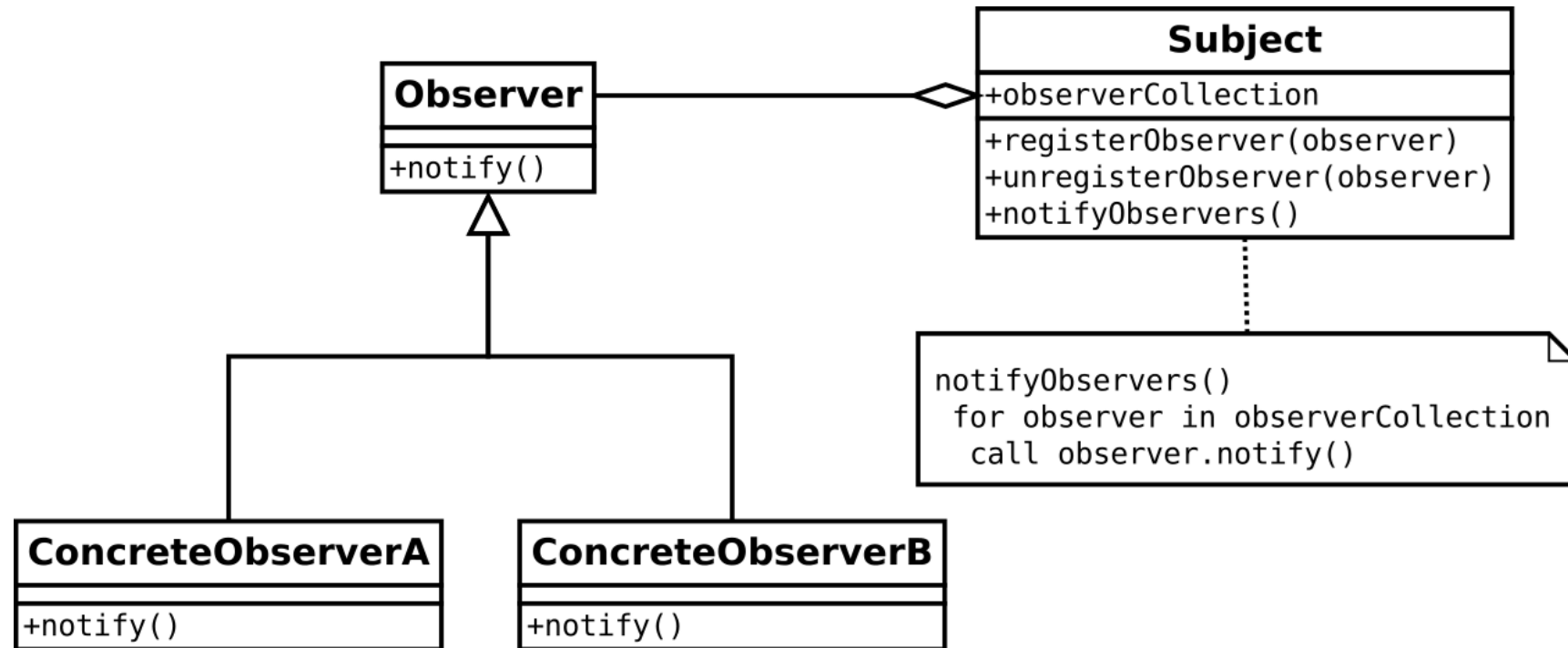
JS Design Patterns: Mixin

- Intent: Mixins as a means of collecting functionality through extension – simple alternative to multiple inheritance

- Example:

var o1 = { a: 1, b: 1, c: 1 };
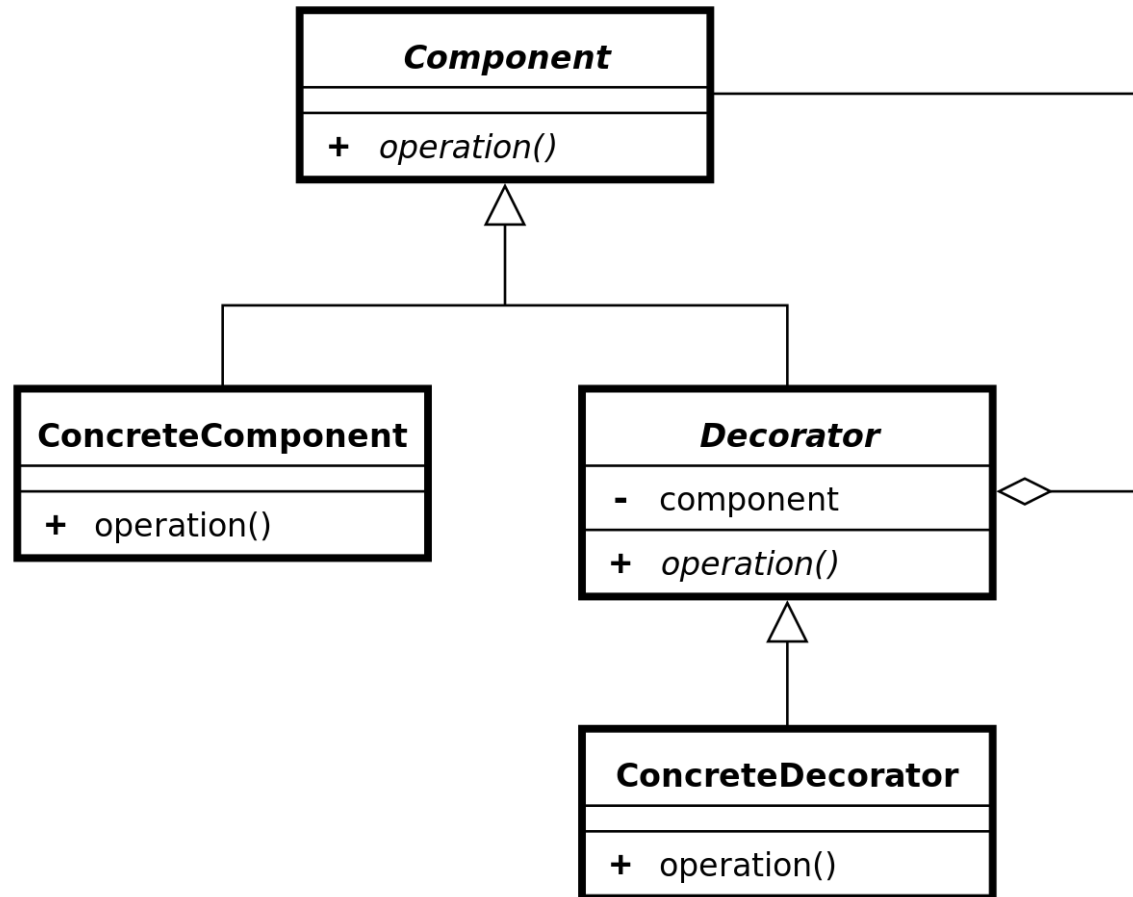
var o2 = { b: 2, c: 2 };

var o3 = { c: 3 };

var obj = Object.assign({ }, o1, o2, o3);

console.log(obj); // { a: 1, b: 2, c: 3 }

- In ECMAScript 6 there is **Object.assign(target, ...sources)**

JS Design Patterns: Decorator

- Intent: Attach additional responsibilities to an object dynamically keeping the same interface.

- Decorators provide a flexible alternative to subclassing for extending functionality.

Conclusions – OO JavaScript Development

JavaScript™ provides everything needed for contemporary object-oriented and functional software development. JavaScript supports:

- **Data encapsulation** (separation of public and private parts) – How**?**: Using design patterns **Module**

- **Inheritance** – before ES 6 there were no classes but several choices for constructing new objects using object templates ("pseudo-classical" using **new**, OR **using functions**, OR **Object.create(baseObject)**, OR **Mixin**)

- **Polimorphism** supported – there are methods with the same name and different implementations – **duck typing**

Event Handling Models in JavaScript

- DOM Level 0 (original Netscape model)

<a href="#" onclick= "alert('I\'m clicked!'); return false;" />

- Traditional model (as properties)

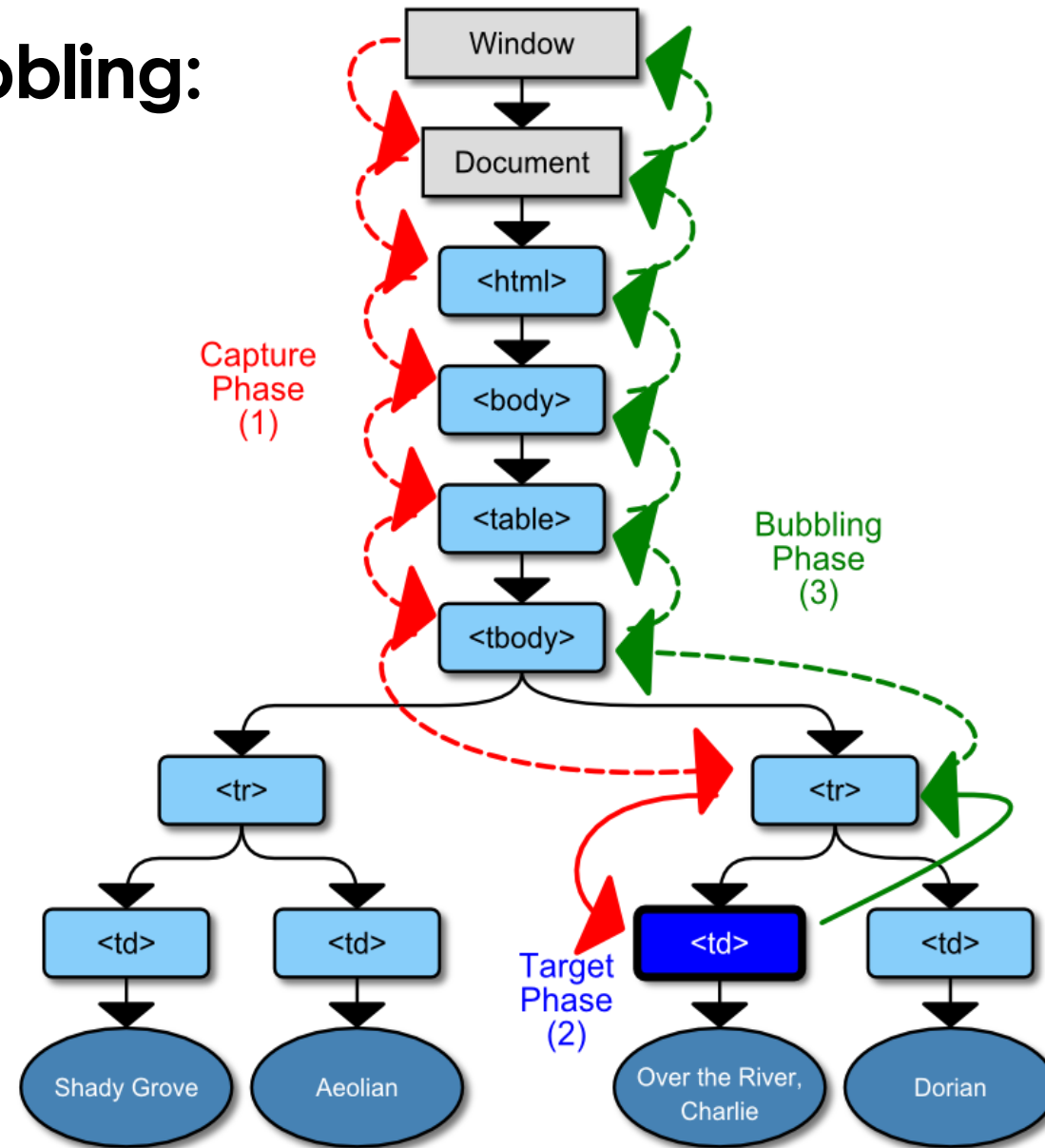**anElem.onclick = function() { this.style.color = 'red'; }**

- can register multiple event handlers:

var oldHandler = (anElem.onclick) ? anElem.onclick : function (){ };

anElem.onclick = function () {oldHandler(); this.style.color = 'red'; };

- Microsoft Event Handling Model

- DOM Level 2 Event Handling Model

- DOM Level 3 Event Handling Model

Events Capturing and Bubbling:

W3C DOM Level 2 Event Handling Model

- Three phases in event handling life-cycle:
  - **Capturing** phase – from document to target element
  - **At Target** phase – processsing in the target element
  - **Bubbling** phase – returns back from target to document
- All events go through Capturing phase, but not all through Bubbling phase – only low level (raw) events
- **event.stopPropagation()** - stops further processing
- **event.preventDefault()** - prevents standards event processing
- Register/deregister event handlers:

anElement.addEventListener('click', eventListener, false)

anElement.removeEventListener('click', eventListener, false)

Microsoft Event Handling Model (Older IE)

- Register/deregister event handlers:

anElement.attachEvent('onclick', eventListener)

anElement.detachEvent('onclick', eventListener)

- Callback function *eventListener* does not receive *event* object:

function crossBrowserEventHandler(event) {

   if(!event) event = window.event; ... // processing follows ... }

- No Capturing phase – every element has methods **setCapture()** and **releaseCapture()**

- from document towards target element

- **window.event.cancelBubble = true;** // stops bubbling -a

- **window.event.returnValue=false;** // prevents default action

W3C DOM Level 2 Events and APIs – Sample Events

https://www.w3schools.com/jsref/dom_obj_event.asp
https://developer.mozilla.org/en-US/docs/Web/API/Event

Interface Name	Events
Event	abort, blur, change, error, focus, load, reset, resize, scroll, select, submit, unload
MouseEvent	click, mousedown, mousemove, mouseout, mouseover, mouseup
UIEvent	DOMActivate, DOMFocusIn, DOMFocusOut

Resources

- Crockford, D., JavaScript: The Good Parts. O'Reilly, 2008.

- Douglas Crockford: JavaScript: The Good Parts video at YouTube – http://www.youtube.com/watch?v=_DKkVvOt6dk

- Douglas Crockford: JavaScript: The Good Parts presentation at http://crockford.com/onjs/2.pptx

- Koss, M., Object Oriented Programming in JavaScript – http://mckoss.com/jscript/object.htm

- Osmani, A., Essential JavaScript Design Patterns for Beginners http://addyosmani.com/resources/essentialjsdesignpatterns/book/

- Fielding's REST blog – http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD