

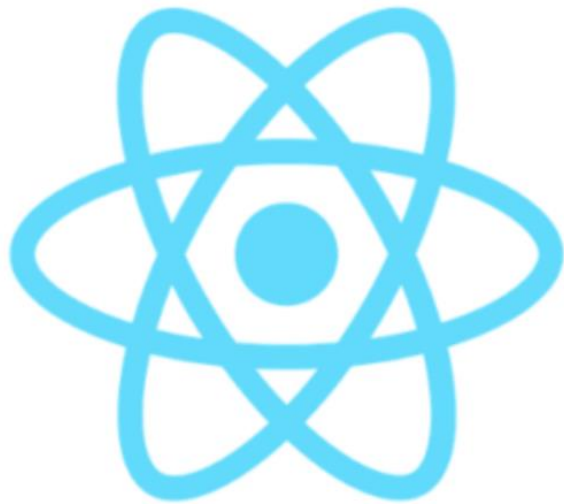


Full-stack Application Development

Functional and Object Oriented JavaScript. Design Patterns. Event-Driven Programming

Where to Find The Code and Materials?

<https://github.com/iproduct/fullstack-typescript-react>



Brief History of JavaScript™

- JavaScript™ created by Brendan Eich from Netscape for less than 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 properties

Ex.: `MyObject.name = 'Scene 1';`

`MyObject['num-elements'] = 5;`

`MyObject.prototype.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 properties and methods).
- **Prototypes** are objects (which also can have their prototypes) → inheritance = 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'

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 *searchElement* element in the array
- **lastIndexOf (searchElement[, fromIndex])** – returns the index of **last** occurrence of the *searchElement* 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 inner
function which 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];  
    }  
}
```

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

Object Literals. Using this

- Object literals – example:

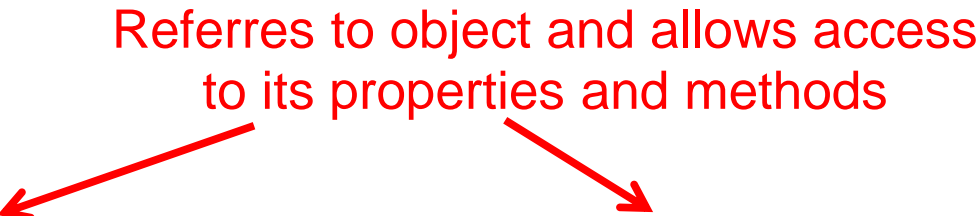
```
var point1 = { x: 50, y: 100 }
```

```
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'
```

Refers to object and allows access
to its properties and methods

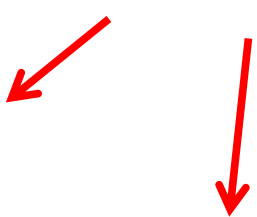


Accessing this in Inner Functions

- 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()

```
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.clone()`)
- 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

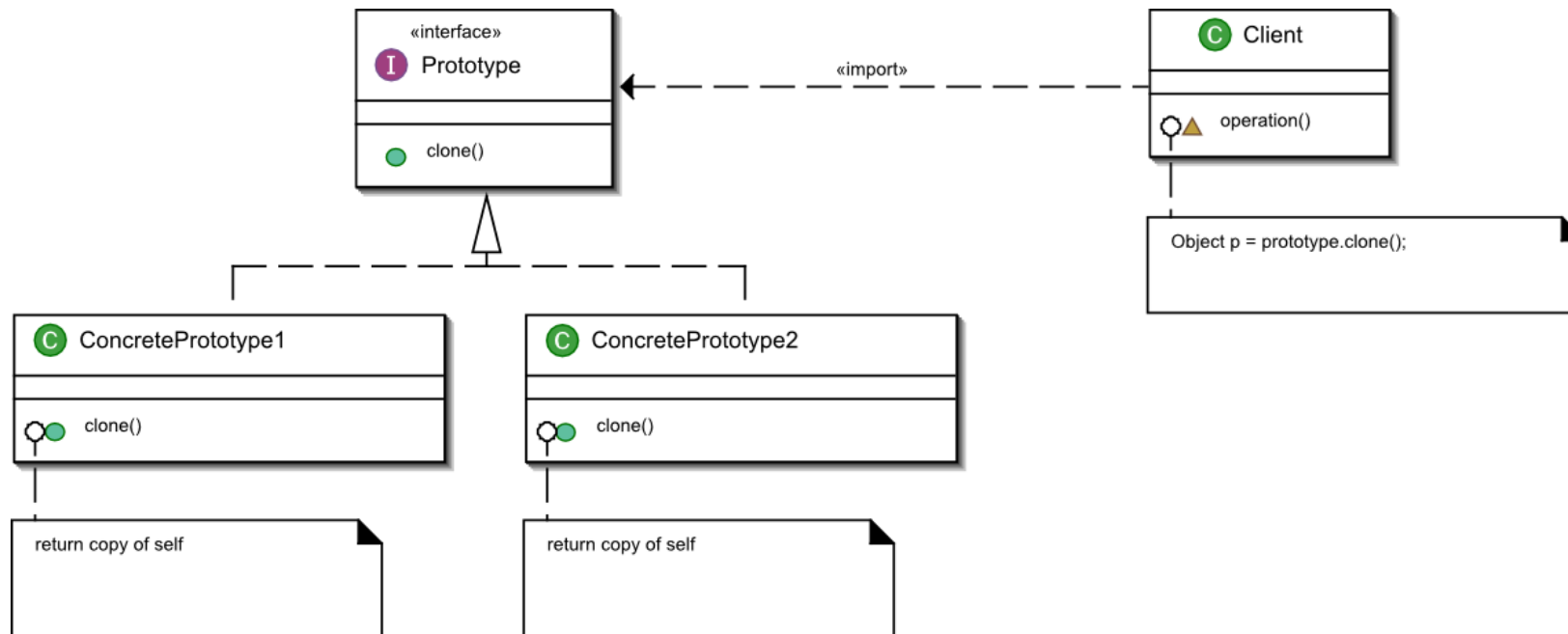
Learning JavaScript Design Patterns

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])`




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 \(ES Modules\)](#)

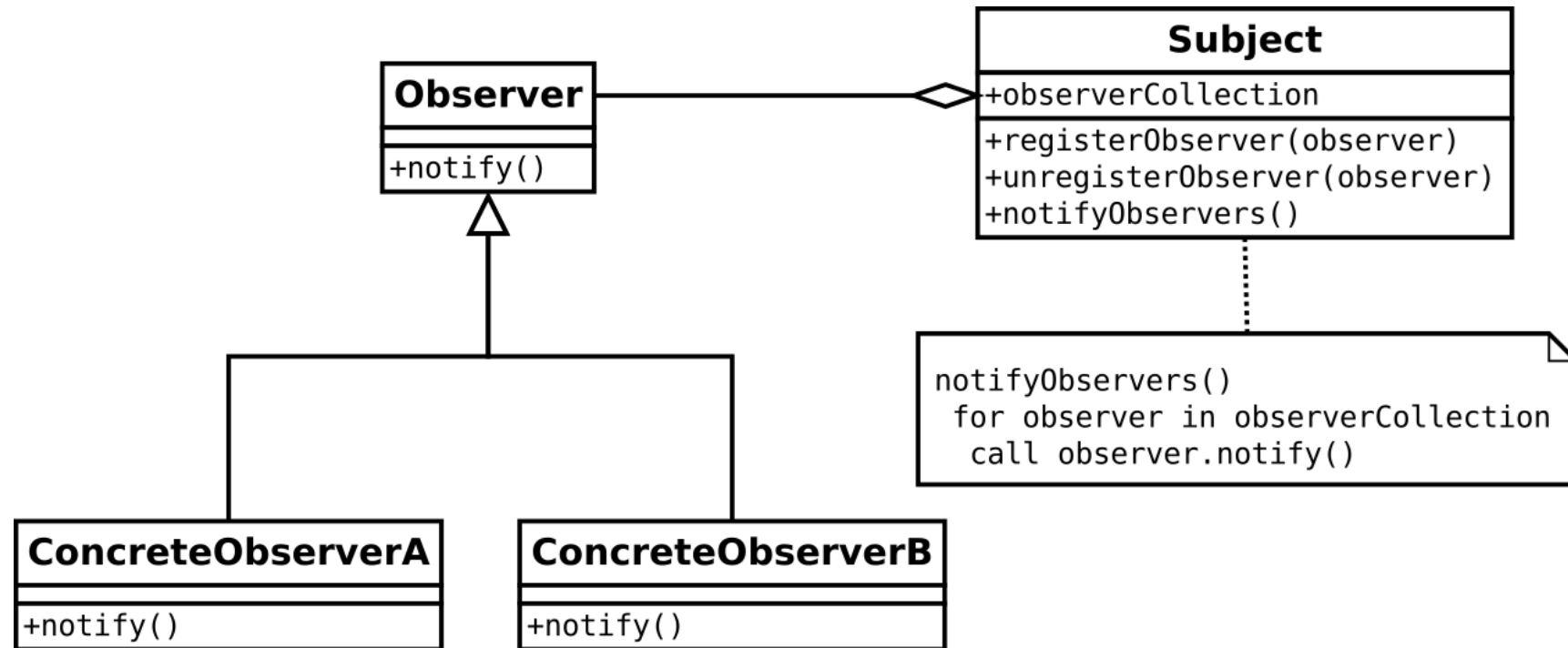
JS Design Patterns: Singleton

- 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 };
```

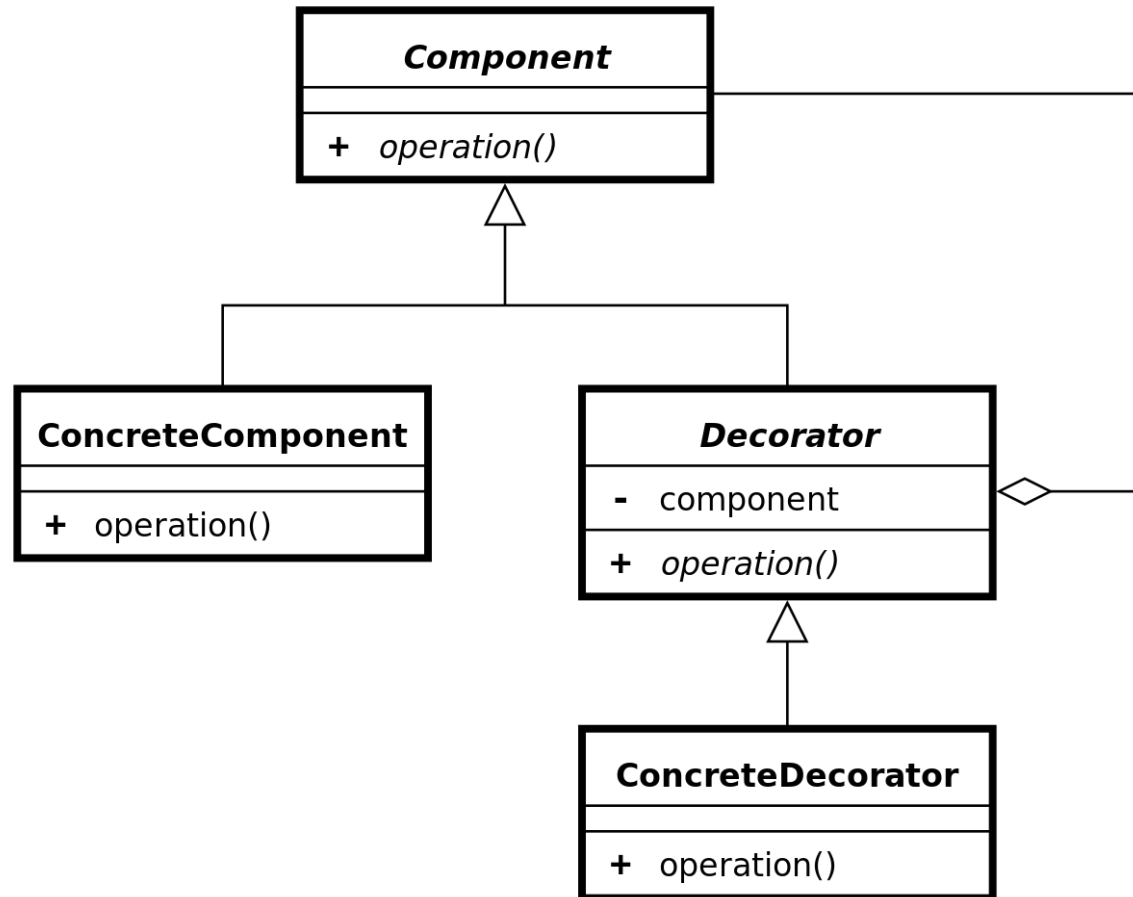
```
var obj = Object.assign({}, o1, o2, o3);
```

```
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)

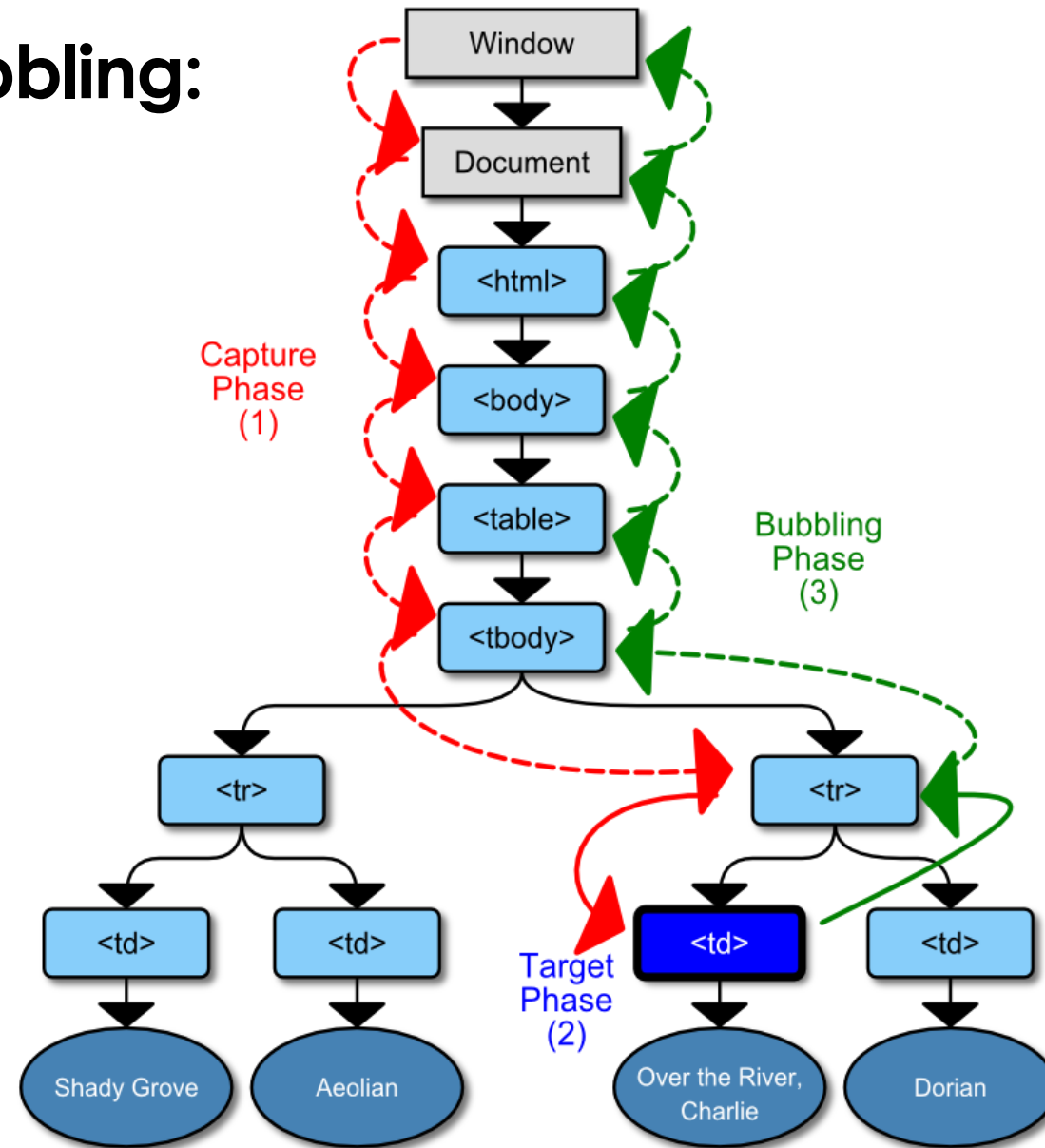
```
anElem.onclick = function() { this.style.color = 'red'; }
```

- 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 – processing 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

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>