var fish = {

type : "steelhead",

water : "fresh",

catch : function(){

if(fish.type === "bass"){

alert("this is not a steelhead");

} else if (fish.water === "salt"){

alert("this is not a river");

}

}

};

//fish.type = "bass";

fish["water"] = "salt"

fish.catch();

When an object has a property with function as the value, it is referred to as a method of that object.

var mark = {

name : "mark",

tshirtcolor : "navy blue"

};

function Friend (name, tshirtcolor) {

this.name = name;

this.tshirtcolor;

}

var denny = new Friend ("Denny", "blue");

alert(denny.name);

**Javascript Objects and Prototypes**

'use strict';

var Cat = {

name : {first:'fluffy',last: 'lebough'},

color : 'tabby'

};

for (var propertyName in Cat)

display(propertyName); //displays name and color...the properties of the object.

**Getters and Setters:**

'use strict';

var cat = {

name : {first:'fluffy',last: 'lebough'},

color : 'tabby'

}

Object.defineProperty(cat, "fullName", {

get: function() {

return this.name.first + " " + this.name.last

},

set: function(value) {

var nameParts = value.split(' ')

this.name.first = nameParts[0]

this.name.last = nameParts[1]

}

})

cat.fullName = 'Molly wright';

display(cat.fullName)

display(cat.name.first)

display(cat.name.last)

**Javascript Prototypes and Inheritance**

var arr = ['bass', 'pike', 'steelhead']

Object.defineProperty (arr, 'last', {get: function() {

return this[this.length-1]

}})

var last = arr.last

display(last) //displays steelhead

var arr = ['bass', 'pike', 'steelhead']

Object.defineProperty (Array.prototype, 'last', {get: function() {

return this[this.length-1]

}})

var last = arr.last

var arr2 = ['one','two','three']

display(arr.last)

display(arr2.last) //displays steelhead and three

**A prototype is a function that exists on every object in JavaScript.**

**A function's prototype is the object instance that will become the prototype for all objects created using this function as a constructor.**

**An object's prototype is the object instance from which the object is inherited.**

function Animal(voice){

this.voice = voice || 'grunt'

}

Animal.prototype.speak = function() {

display(this.voice)

}

function Cat(name, color) {

Animal.call(this, 'Meow')

this.name = name

this.color = color

}

Cat.prototype = Object.create(Animal.prototype)

Cat.prototype.constructor = Cat

var myCat = new Cat('jill', 'blend')

display (myCat.voice)

**Class**

class Animal {

constructor(voice) {

this.voice = voice || 'grunt'

}

speak() {

display(this.voice)

}

}

class Cat extends Animal{

constructor(name, color){

super('meow')

this.name = name

this.color = color

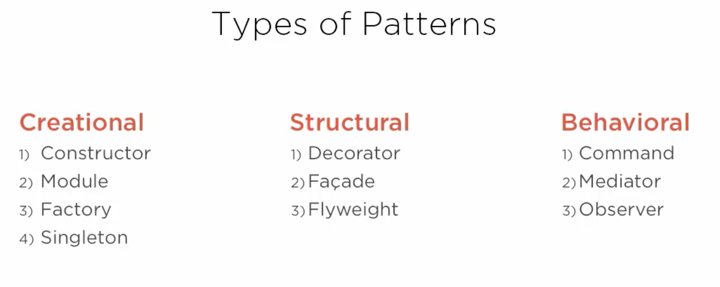
}

}

var myCat = new Cat('jill', 'blend')

myCat.speak()

**JavaScript Design Patterns**



**3 ways to create new objects**

Var obj ={};

Var nextObj = Object.create(object.prototype);

Var lastObj = new Object();

**Assigning Keys and Values (Dot Notation)**

var obj ={};

obj.param = ‘new value’;

console.log(obj.param); //new value

**Assigning Keys and Values (Bracket Notation)**

var obj ={};

obj[‘param’] = ‘new value’;

console.log(obj[‘param’]); //new value

**Can use variables in Bracket Notation**

var obj ={};

Var val = ‘value’;

obj[val] = ‘new value’;

console.log(obj[val]); //new value

**Define Property** <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty>

Object.defineProperty (objectName, property, descriptor)

var task = {

title : 'My task',

description : 'My Description'

};

Object.defineProperty (task, 'toString', {

value: function () {

return this.title + ' ' + this.description;

},

writeable: true,

enumerable: true,

configurable: true

});

console.log(task.toString()); // My task my description

**Inheritance**

var urgentTask = Object.create(task);

Object.defineProperty (urgentTask, 'toString', {

value: function () {

return this.title + ' ' + 'is urgent';

},

writeable: true,

enumerable: true,

configurable: true

});

console.log(task.toString()); // My task is urgent

**Creational Design Patterns**

**Constructor Pattern (Node)**

function ObjectName(param1, param2) {

this.param1 = param1;

this.param2 = param2;

this.toString = function() {

return this.param1 + ' '+ this.param2;

}

}

var task = function (name) {

this.name = name;

this.completed = false;

this.complete = function() {

console.log('completing task: ' + this.name);

this.complete = true;

}

this.save = function() {

console.log('saving Task: ' + this.name);

}

}

var task1 = new task('create demo for constructors');

var task2 = new task('create demo for modules');

var task3 = new task('create demo for singletons');

var task4 = new task('create demo for prototypes');

task1.complete();

task2.save();

task3.save();

task4.save();

**Prototype: An encapsulation of properties that an object links to.**

**Candidates = save() and complete() methods**

ClassName.prototype.methodname = function (arguments) {

};

var task = function (name) {

this.name = name;

this.completed = false;

/\* this.complete = function() {

console.log('completing task: ' + this.name);

this.complete = true;

}

this.save = function() {

console.log('saving Task: ' + this.name);

}

\*/

}

task.prototype.complete = function () {

console.log('completing task: ' + this.name);

this.complete = true;

};

task.prototype.save = function () {

console.log('saving Task: ' + this.name);

};

var task1 = new task('create demo for constructors');

var task2 = new task('create demo for modules');

var task3 = new task('create demo for singletons');

var task4 = new task('create demo for prototypes');

task1.complete();

task2.save();

task3.save();

task4.save();

**Constructor Pattern (Class) Must use ‘use strict’**

'use strict'

class task {

constructor (name) {

this.name = name;

this.completed = false;

};

complete () {

console.log('completing task: ' + this.name);

this.complete = true;

};

save () {

console.log('saving Task: ' + this.name);

};

}

var task1 = new task('create demo for constructors');

var task2 = new task('create demo for modules');

var task3 = new task('create demo for singletons');

var task4 = new task('create demo for prototypes');

task1.complete();

task2.save();

task3.save();

task4.save();

**Module Pattern (typically 1 of something, like calling a service)**

**Collection of functions**

var Module = {

method: function() {...},

nextMethod: function() {...}

}

**Revealing Module Pattern**

var repo = function () {

var get = function(id) {

console.log('getting task ' + id);

return {

name: 'new task from db'

}

}

var save = function(task) {

console.log('Saving ' + task.name + ' to the db');

}

return {

get: get,

save: save

}

}

module.exports = repo();

**Factory Pattern (A pattern used to simplify object creation)**

* **Simplifies object creation**
* **Creating different object based on need**
* **Repository creation**

var repoFactory = function() {

var repos = this;

var repoList = [{name: 'task', sources: './taskRepository'},

{name: 'user', source: './userRepository'},

{name: 'project', source: "./projectRepository"}];

repoList.forEach(function(repo) {

repos[repo.name] = require(repo.source)()

});

};

module.exports = new repoFactory;

**Singleton Pattern (Used to restrict an object to one instance of that object across the application)**

* **Remembers the last time you used it**
* **Hands the same instance back**
* **Node uses CommonJS**
* **Angular uses singletons for services**

**Structural Design Patterns**

**Concerned with how objects are made up and simplify relationships between objects.**

* **Deal with the relationship of objects**
* **Extend functionality**
* **Simplify functionality**

var task = function (name) {

this.name = name;

this.completed = false;

}

task.prototype.complete = function () {

console.log('completing task: ' + this.name);

this.complete = true;

};

task.prototype.save = function () {

console.log('saving Task: ' + this.name);

};

var myTask = new task('Legacy Task');

myTask.complete();

myTask.save();

var UrgentTask = function(name, priority) {

task.call(this, name);

this.priority = priority;

};

UrgentTask.prototype = Object.create(task.prototype);

UrgentTask.prototype.notify = function() {

console.log('notifying important people');

};

UrgentTask.prototype.save = function () {

this.notify();

console.log('do special stuff before saving');

task.prototype.save.call(this)

};

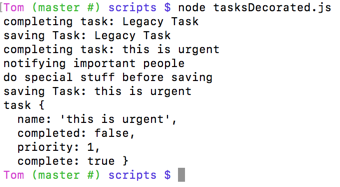
var ut = new UrgentTask('this is urgent', 1);

ut.complete();

ut.save();

console.log(ut);

//module.exports = task;



**Angular Decorator**

(function(){

var app = angular.module('taskManager');

//$delegate is an exact copy of the object, that is then decorated.

app.decorator('TaskRepository',function($delegate){

var oldSave = $delegate.save;

$delegate.save = function(task){

console.log('Special logging for task ' + task.name);

oldSave(task);

}

return $delegate;

})

}())

**Facade Pattern**

**Used to provide a simplified interface to a complicated system**

* **Think about the front of a building**
* **Facade hides the chaos from us**
* **Simplifies the interface**
* **Think JQuery**

<http://plnkr.co/edit/2CH3Qy?p=preview>

**Flyweight Pattern**

**Conserves memory by sharing portions of an object between objects.**

* **Lots of Non-unique data**
* **Flyweight shares data across objects**
* **Results in lower memory footprint**
* **Like a boxer --- flyweight**
* **Only useful when you have a large number of objects**

**Behavioral Design Patterns**

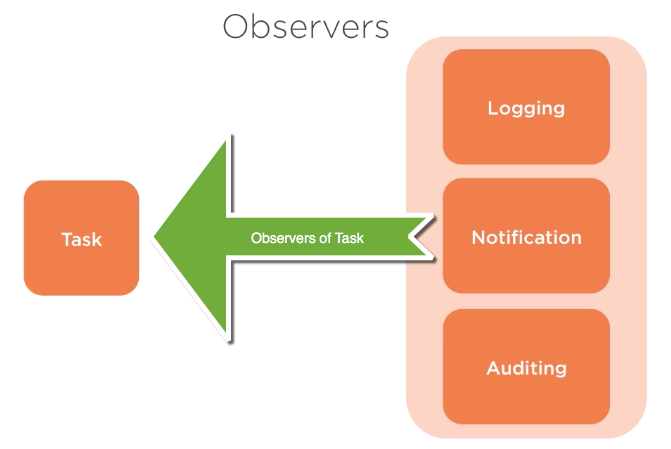
**Concerned with the assignment of responsibilities between objects and how they communicate.**

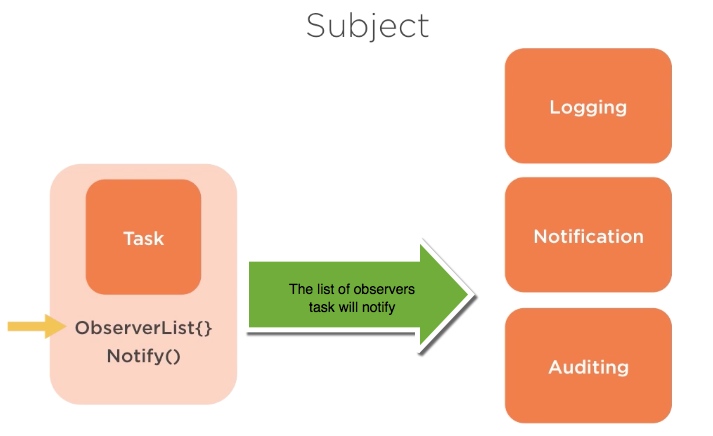
* **Deals with the responsibilities of objects**
* **Helps objects cooperate**
* **Assign clear hierarchy**
* **Can encapsulate requests**

**Observer Pattern**

**Allows a collection of objects to watch an object and no notified of changes**

* **Allows for loosely coupled system**
* **One object is the focal point**
* **Group of objects watch for changes**





var Task = require('./taskObserver');

var notificationService = function() {

var message = 'Notifying ';

this.update = function (task) {

console.log(message + task.user + ' for task ' + task.name);

}

};

var loggingService = function() {

var message = 'Logging ';

this.update = function(task){

console.log(message + task.user + ' for task ' + task.name);

}

};

var auditService = function() {

var message = 'Auditing ';

this.update = function(task){

console.log(message + task.user + ' for task ' + task.name);

}

};

var ObserverList = function(){

this.observerList = [];

}

ObserverList.prototype.add = function(obj){

return this.observerList.push(obj);

};

ObserverList.prototype.get = function(index){

if (index > -1 && index < this.observerList.length){

return this.observerList[index];

}

};

ObserverList.prototype.count=function() {

return this.observerList.length;

}

var ObservableTask = function (data) {

Task.call(this, data);

this.observers = new ObserverList();

};

ObservableTask.prototype.addObserver = function(observer) {

this.observers.add(observer);

};

ObservableTask.prototype.notify = function (context) {

var observerCount = this.observers.count();

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

this.observers.get(i)(context);

}

}

ObservableTask.prototype.save = function() {

this.notify(this);

Task.prototype.save.call(this);

};

var task1 = new ObservableTask({name: 'create a demo for constructor', user:'Jon'})

var not = new notificationService();

var ls = new loggingService();

var audit = new auditService();

task1.addObserver(not.update);

task1.addObserver(ls.update);

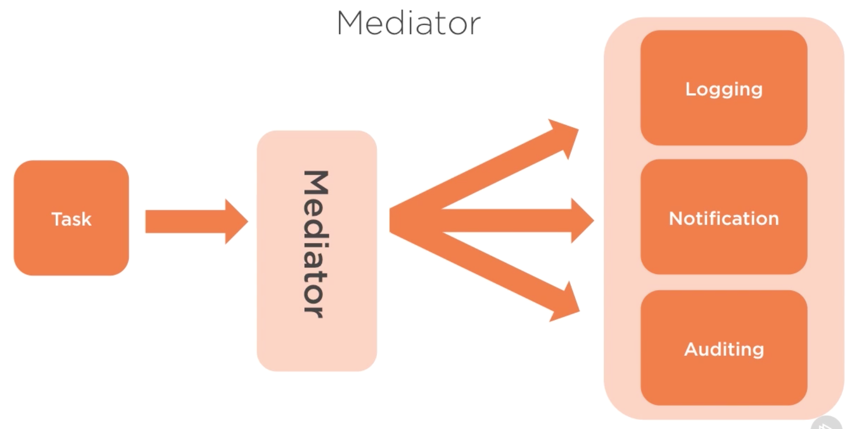
task1.addObserver(audit.update);

task1.save();

**Mediator Pattern**

**Controls communication between objects so neither object has to be coupled to the others**

* **Allows for loosely coupled system**
* **One object manages all communication**
* **Allows for many to many relationships**



var Task = require('./taskObserver');

var notificationService = function() {

var message = 'Notifying ';

this.update = function (task) {

console.log(message + task.user + ' for task ' + task.name);

}

};

var loggingService = function() {

var message = 'Logging ';

this.update = function(task){

console.log(message + task.user + ' for task ' + task.name);

}

};

var auditService = function() {

var message = 'Auditing ';

this.update = function(task){

console.log(message + task.user + ' for task ' + task.name);

}

};

//wrap the mediator in in iife

//**Immediately Invoked Function Expression (IIFE)**

/\*

Why?: An IIFE removes variables from the global scope. This helps prevent variables and function declarations from living longer than expected in the global scope, which also helps avoid variable collisions.

Why?: When your code is minified and bundled into a single file for deployment to a production server, you could have collisions of variables and many global variables. An IIFE protects you against both of these by providing variable scope for each file.

\*/

/\*

The most simple way to think of a **closure** is a **function** that can be stored as a variable (referred to as a "first-class **function**"), that has a special ability to access other variables local to the scope it was created in. Example (JavaScript)

\*/

var mediator = (function(){

var channnels = {};

var subscribe = function(channel, context, func){

if(!mediator.channels[channel]) {

mediator.channels[channel]=[]

}

mediator.channels[channel].push({

context: context,

func: func

});

}

var publish = function(channel){

if(!this.channels[channel]){

return false;

}

var args = Array.prototype.slice.call(arguments, 1);

for (var i = 0; i < mediator.channels[channel].length; i++){

var sub = mediator.channels[channel][i];

sub.func.apply(sub.context, args);

}

}

return{

channels: {},

subscribe: subscribe,

publish: publish

};

}());

var ObserverList = function(){

this.observerList = [];

}

ObserverList.prototype.add = function(obj){

return this.observerList.push(obj);

};

ObserverList.prototype.get = function(index){

if (index > -1 && index < this.observerList.length){

return this.observerList[index];

}

};

ObserverList.prototype.count=function() {

return this.observerList.length;

}

var ObservableTask = function (data) {

Task.call(this, data);

this.observers = new ObserverList();

};

ObservableTask.prototype.addObserver = function(observer) {

this.observers.add(observer);

};

ObservableTask.prototype.notify = function (context) {

var observerCount = this.observers.count();

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

this.observers.get(i)(context);

}

}

ObservableTask.prototype.save = function() {

this.notify(this);

Task.prototype.save.call(this);

};

var task1 = new ObservableTask({name: 'create a demo for constructor', user:'Jon'})

var not = new notificationService();

var ls = new loggingService();

var audit = new auditService();

//task1.addObserver(not.update);

//task1.addObserver(ls.update);

//task1.addObserver(audit.update);

mediator.subscribe('complete', not, not.update);

mediator.subscribe('complete', ls, ls.update);

mediator.subscribe('complete', audit, audit.update);

task1.complete = function() {

mediator.publish('complete', this);

Task.prototype.complete.call(this);

}

task1.complete();

**Command Pattern**

**Encapsulates the calling of a method as an object**

* **Fully decouples the execution from the implementation**
* **Allows for less fragile implementations**
* **Supports undo operations**
* **Supports auditing and logging of operations**

var repo = {

tasks: {},

commands:[],

get: function(id) {

console.log('getting task ' + id);

return {

name: 'new task from db'

}

},

save: function(task) {

repo.tasks[task.id]= task;

console.log('Saving ' + task.name + ' to the db');

},

replay: function () {

for(var i = 0; i , repo.commands.length; i++){

var command = repo.commands[i];

repo.executeNoLog(command.name, command.obj);

}

}

}

repo.executeNoLog = function(name){

var args = Array.prototype.slice.call(arguments, 1);

if(repo[name]) {

return repo[name].apply(repo,args) //apply rather than call

} //allows passing in of an array

}

repo.execute = function(name){

var args = Array.prototype.slice.call(arguments, 1);

repo.commands.push({

name: name,

obj: args[0]

});

if(repo[name]) {

return repo[name].apply(repo,args) //apply rather than call

} //allows passing in of an array

return false;

};

repo.execute('save', {

id: 1,

name: 'Task 1',

completed: false

});

repo.execute('save', {

id: 2,

name: 'Task 2',

completed: false

})

repo.execute('save', {

id: 3,

name: 'Task 3',

completed: false

})

repo.execute('save', {

id: 4,

name: 'Task 4',

completed: false

})

console.log(repo.tasks);

repo.tasks = {};

console.log(repo.tasks);

repo.replay();

console.log(repo.tasks);

**JQuery: Ge**

**tting Started**

**$ == JQuery**

var txt = $('h2').text();

**Id attributes: Values for Id’s must be unique**

Example: <h2 id=”content-container” />

$(‘#content-container’) //make use of the # (pound) before the used to identify id attribute of html tag

**Classes: Value for class be repeated and have multiple space-separated values which represent Sets or collections.**

Example: <h2 class=”title” />

$(‘.title’) //make use of a . (dot) before the class name

**Pseudo Class: reflect a specific state of an element**

Example: <style> a:hover {...} </style>

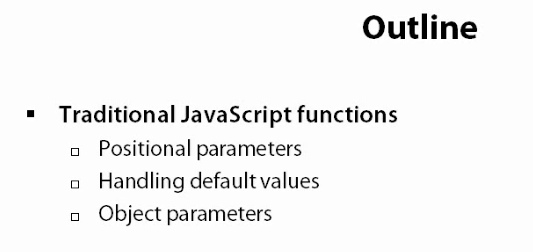
$(‘div:first-child’) //make use of a : (colon) right after the element name

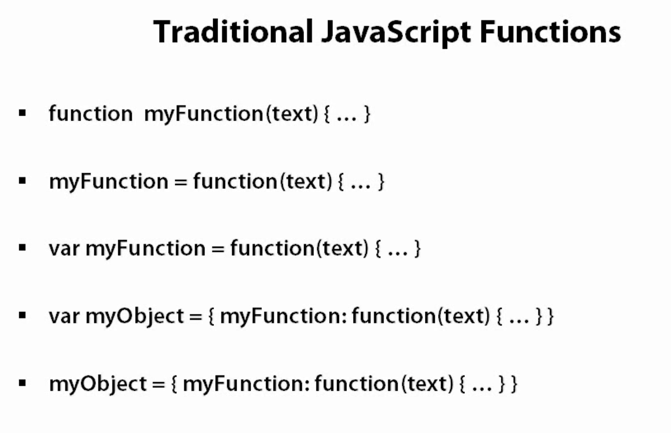
**The most simple way to think of a closure is a function that can be stored as a variable (referred to as a "first-class function"), that has a special ability to access other variables local to the scope it was created in. Example (JavaScript):**





**Advanced Techniques in JavaScript and JQuery**





**Events-1.html**

<!DOCTYPE html>

<html lang="en">

<head>

<title>Events 1</title>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>

<link type="text/css" rel="stylesheet" href="styles/eventStyles.css" />

</head>

<body>

<div id="WorkArea">

<h5>Work Area</h5>

<hr />

<input type="button" class="clickable" value="Click Me" />

</div>

<div id="Messages">

<h5>Messages</h5>

<hr />

</div>

</body>

<script>

$(function () {

$('.clickable').click(function () {

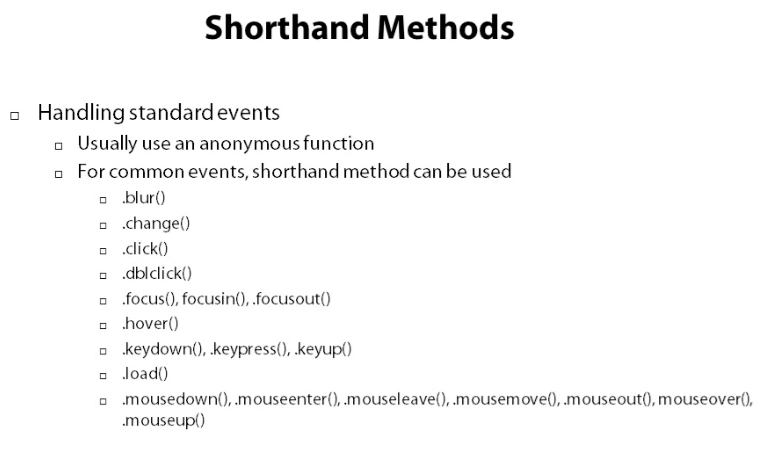
$('#Messages').append('click occured <br/>');

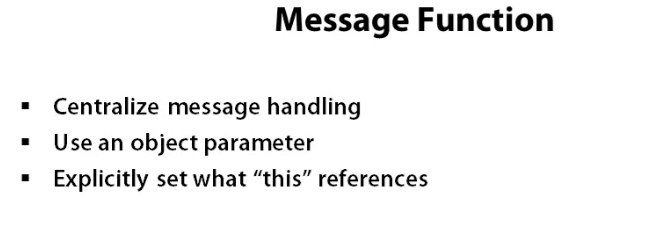
});

});

</script>

</html>





<!DOCTYPE html>

<html lang="en">

<head>

<title>Events 1</title>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>

<link type="text/css" rel="stylesheet" href="styles/eventStyles.css" />

</head>

<body>

<div id="WorkArea">

<h5 class="clickable">Work Area</h5>

<hr />

<input type="button" class="clickable" value="Click Me" />

</div>

<div id="Messages">

<h5>Messages</h5>

<hr />

</div>

</body>

<script>

$(function () {

var showEventMessage = function(options) {

options = $.extend (

{ eventType: 'CLICK',

eventTarget: this,

suffix: '<br/>'

}, options);

var message = options.eventType + ': ' +

(options.eventTarget.nodeName || 'unknown') +

options.suffix;

$('#Messages').append(message);

};

$('.clickable').click(function () {

//showEventMessage.call(this, {suffix: '!<br/>'});

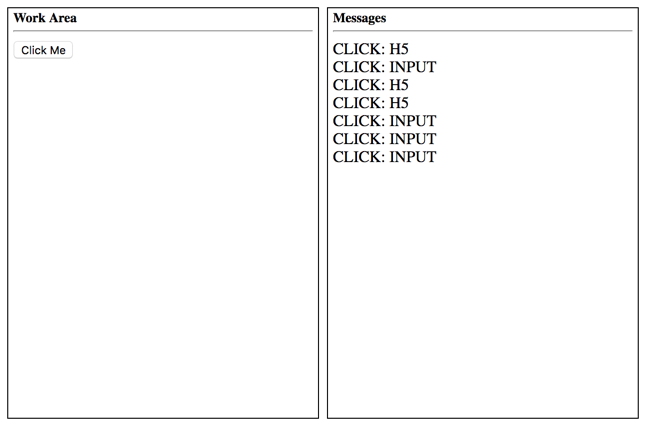
showEventMessage.call(this);

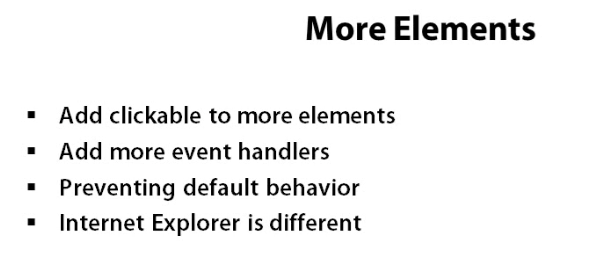
});

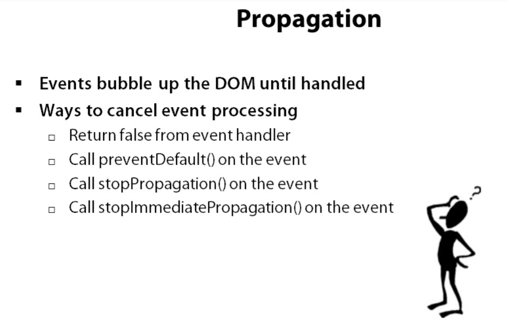
});

</script>

</html>







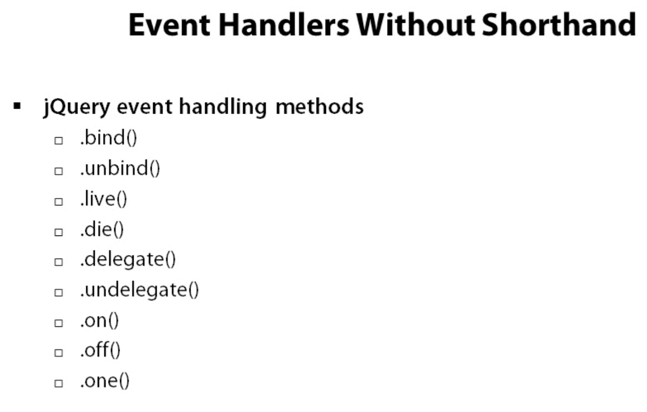
$('.clickable')

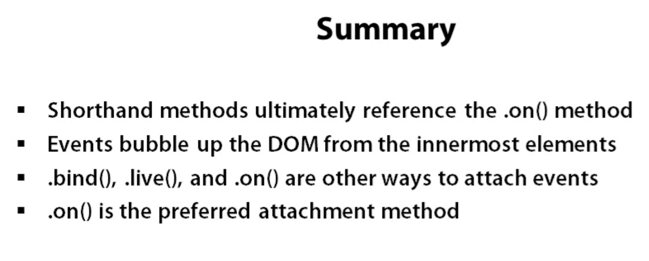
.click(function (event) {

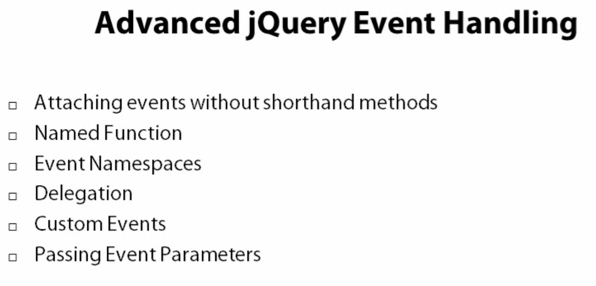
showEventMessage.call(this, {eventType: event.type});

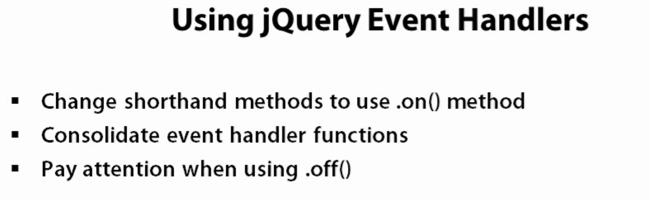
event.stopImmediatePropagation();

**stopImmediatePropagation()** => prevents the same event from being fired at levels above it in the DOM.









<!DOCTYPE html>

<html lang="en">

<head>

<title>Events 1</title>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>

<link type="text/css" rel="stylesheet" href="styles/eventStyles.css" /> </head>

<body>

<div id="WorkArea" class="clickable">

<h5 class="clickable">Work Area</h5>

<hr />

<input type="button" class="clickable" value="Click Me" />

<p class="clickable">This is clickable too!</p>

<p class="clickable"><span class="clickable">Click me too</span></p>

</div>

<div id="Messages">

<h5>Messages</h5>

<hr /> </div>

</body>

<script>

$(function () {

var showEventMessage = function(options) {

options = $.extend (

{ eventType: 'CLICK',

eventTarget: this,

suffix: '<br/>'

}, options);

var message = options.eventType + ': ' +

(options.eventTarget.nodeName || 'unknown') +

options.suffix;

$('#Messages').append(message);

};

$('.clickable')

.off('click')

.on('click dblclick mousedown', function (event) {

if (!event.isPropagationStopped() &&

!event.isImmediatePropagationStopped() &&

!event.isDefaultPrevented()) {

showEventMessage.call(this, {eventType: event.type});

event.stopPropagation();

if (event.type === 'mousedown')

event.preventDefault();

}

})

.off('click mousedown')

.on('mousedown', function(event) {

if (!event.isPropagationStopped() &&

!event.isImmediatePropagationStopped() &&

!event.isDefaultPrevented()) {

showEventMessage.call(this, {eventType: event.type, suffix: '#2<br/>'});

}

})

.on('selectstart', function(event) {

return false

})

});

</script>

</html>

