**Advanced JavaScript:**

**Event Bubbling and Capturing**

Event bubbling and capturing are two ways of event propagation in HTML DOM.

* Event bubbling
* In bubbling the event is first captured and handled by the inner most elements and then propagated to outer elements.

**Ex:**

<div class="div-1">1

<div class="div-2">2

<div class="div-3">3</div>

</div>

</div>

* Event Capturing
* In capturing the event is first captured by the outer most elements and propagated to the inner most elements.

Only event bubbling model is supported by all the major browsers.

So if you are going to use event capturing still you need to handle event bubbling for IE. So it will easier to use event bubbling instead of capturing.

**Callbacks and Anonymous functions**

**Callbacks**

Callback functions are the functions that are passed to another function as an argument.

Callback functions are probably the most widely used functional programming technique in JavaScript, and they are literally in just about every piece of JavaScript and Jquery code.

Ex1:

function getDivision(arg1, arg2, callback) {

var divisionValue = arg1/arg2;

callback(divisionValue);

}

getDivision(5, 15, function(num){

return Math.round(num);

});

Ex2:

$("button").click (function () {  
 $("p").hide (1000);  
  alert ("The paragraph is now hidden");  
});

Ex3:

$("button").click(function(){  
  $("p").hide("slow”, function(){  
    alert("The paragraph is now hidden");  
  });  
});

**Anonymous function**

<http://helephant.com/2008/08/23/javascript-anonymous-functions/>

Anonymous functions are functions that are dynamically declared at runtime. They’re called anonymous functions because they aren’t given a name in the same way as normal functions.

**Normal function declaration**

**function** hello() {

alert('world');

}

hello();

**Anonymous function declaration**

**var** anon = **function**() {

alert('I am anonymous');

};

anon();

setTimeout(**function**() {

alert('hello');

}, 1000);

(**function**() {

alert('foo');

}());

**Closures:**

**http://javascriptissexy.com/understand-javascript-closures-with-ease/**

A closure is an inner function that has access to the outer (enclosing) function’s variables—scope chain. The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function’s variables, and it has access to the global variables.

The inner function has access not only to the outer function’s variables, but also to the outer function’s parameters. Note that the inner function cannot call the outer function’s *arguments* object, however, even though it can call the outer function’s parameters directly.

EX1:

function showName (firstName, lastName) {

var nameIntro = "Your name is";

function makeFullName() {

return nameIntro + firstName + " " + lastName;

}

return makeFullName();

}

showName("Michael", "Jackson");

Ex2:

function celebrityName(firstName) {

var nameIntro = "This celebrity is";

function lastName(theLastName) {

return nameIntro + firstName + " " + theLastName;

}

return lastName;

}

var mjName = celebrityName("Michael");

mjName("Jackson");

**OOPS in JavaScript**

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction\_to\_Object-Oriented\_JavaScript

Object-oriented programming is a programming paradigm that uses abstraction to create models based on the real world.

It uses several techniques from previously established paradigms, including modularity, polymorphism, and encapsulation.

Object-oriented programming is intended to promote greater flexibility and maintainability in programming, and is widely popular in large-scale software engineering.

**Concepts:**

* Class
* Object
* Property
* Method
* Constructor
* Inheritance
* Encapsulation
* Abstraction
* Polymorphism

Class: JavaScript is a prototype-based language which contains no class statement, such as is found in C++ or Java.

Ex:

function Person() { }

Object:

An object is just a special kind of data, with **properties** and **methods**.

Ex:

function Person() { }

var person1 = new Person();

var person2 = new Person();

Property:

Properties are variables contained in the class; every instance of the object has those properties. Properties should be set in the prototype property of the class (function) so that inheritance works correctly.

Ex:

function Person(gender) {

this.gender = gender;

alert('Person instantiated');

}

var person1 = new Person('Male');

var person2 = new Person('Female');

alert('person1 is a ' + person1.gender);

Method:

Methods follow the same logic as properties; the difference is that they are functions and they are defined as functions. Calling a method is similar to accessing a property, but you add () at the end of the method name, possibly with arguments.

Ex:

function Person(gender) {

this.gender = gender;

this.sayHello = function()

{

alert ('hello');

};

}

var person1 = new Person('Male');

var person2 = new Person('Female');

person1.sayHello();

Constructor:

The constructor is called at the moment of instantiation (the moment when the object instance is created). The constructor is a method of the class. In JavaScript, the function serves as the constructor of the object; therefore, there is no need to explicitly define a constructor method. Every action declared in the class gets executed at the time of instantiation.

Ex:

function Person() {

alert('Person instantiated');

}

var person1 = new Person();

var person2 = new Person();

Inheritance:

Inheritance is a way to create a class as a specialized version of one or more classes (*JavaScript only supports single class inheritance*).

Ex:

// define the Person Class

function Person() {}

Person.prototype.walk = function(){

alert ('I am walking!');

};

Person.prototype.sayHello = function(){

alert ('hello');

};

// define the Student class

function Student() {

// Call the parent constructor

Person.call(this);

}

// inherit Person

Student.prototype = new Person();

// correct the constructor pointer because it points to Person

Student.prototype.constructor = Student;

// replace the sayHello method

Student.prototype.sayHello = function(){

alert('hi, I am a student');

}

// add sayGoodBye method

Student.prototype.sayGoodBye = function(){

alert('goodBye');

}

var student1 = new Student();

student1.sayHello();

student1.walk();

student1.sayGoodBye();

// check inheritance

alert(student1 instanceof Person); // true

alert(student1 instanceof Student); // true

Encapsulation:

In the previous example, Student does not need to know how the Person class's walk() method is implemented, but still can use that method; the Student class doesn't need to explicitly define that method unless we want to change it. This is called encapsulation, by which every class inherits the methods of its parent and only needs to define things it wishes to change.

Abstraction:

Abstraction is a mechanism that permits modeling the current part of the working problem. This can be achieved by inheritance (specialization), or composition. JavaScript achieves specialization by inheritance, and composition by letting instances of classes be the values of attributes of other objects.

Polymorphism:

Just like all methods and properties are defined inside the prototype property, different classes can define methods with the same name; methods are scoped to the class in which they're defined. This is only true when the two classes do not hold a parent-child relation (when one does not inherit from the other in a chain of inheritance).

Prototype:

<http://net.tutsplus.com/tutorials/javascript-ajax/prototypes-in-javascript-what-you-need-to-know/>

When you define a function within JavaScript, it comes with a few pre-defined properties. One of these is the illusive prototype.

Properties:

* caller
* length
* name

Methods:

* apply()
* bind()
* call()
* constructor()
* hasOwnProperty()
* isPrototypeOf()
* propertyIsEnumerable()
* toLocaleString()
* toString()
* valueOf()

Objects:

* arguments
* prototype

TODO-explain prototype in detail.

Prototype Inheritance:

<http://msdn.microsoft.com/en-us/library/hh924508(v=vs.94).aspx>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create>

TODO-Explain in detail

Class in JavaScript:

TODO-Explain class in JavaScript detail

Module Pattern:

Modules are an integral part of any robust application's architecture and typically help in keeping the units of code for a project both cleanly separated and organized.

In JavaScript, there are several options for implementing modules. These include:

* The Module pattern
* Object literal notation
* AMD modules

Object literal notation:

In object literal notation, an object is described as a set of comma-separated name/value pairs enclosed in curly braces ({}).

Ex:

var myObjectLiteral = {

    parent: value,

    child: function () {}

};

AMD modules:

TODO - need to prepare two lines for this.

The Module pattern:

<http://addyosmani.com/resources/essentialjsdesignpatterns/book/#modulepatternjavascript>

The Module pattern was originally defined as a way to provide both private and public encapsulation.

The Module pattern encapsulates "privacy", state and organization using closures.

It provides a way of wrapping a mix of public and private methods and variables, protecting pieces from leaking into the global scope and accidentally colliding with another developer's interface.

With this pattern, only a public API is returned, keeping everything else within the closure private.

Ex:

var testModule = (function () {

  var counter = 0;

  return {

    incrementCounter: function () {

      return counter++;

    },

    resetCounter: function () {

      console.log( "counter value prior to reset: " + counter );

      counter = 0;

    }

  };

})();

Usage:

testModule.incrementCounter();

testModule.resetCounter();

Jquery:

jQuery is a lightweight, "write less, do more", JavaScript library.

The purpose of jQuery is to make it much easier to use JavaScript on your website.

jQuery takes a lot of common tasks that require many lines of JavaScript code to accomplish, and wraps them into methods that you can call with a single line of code.

jQuery also simplifies a lot of the complicated things from JavaScript, like AJAX calls and DOM manipulation.

The jQuery library contains the following features:

* HTML/DOM manipulation
* CSS manipulation
* HTML event methods
* Effects and animations
* AJAX
* Utilities

Jquery Plug-in Implementation:

<http://www.sitepoint.com/how-to-develop-a-jquery-plugin/>

<http://coding.smashingmagazine.com/2011/10/11/essential-jquery-plugin-patterns/>

<http://www.learningjquery.com/2007/10/a-plugin-development-pattern/>

**Why do we need jQuery Plugin?**

In a word: re-use.

By extending jQuery, you create reusable components that can be used on any web page. Your code is encapsulated and there is less risk that you will use the same function names elsewhere.

**How it will Work:**

$("p").reverseText();

**The Plugin Declaration:**

jQuery.fn.reverseText = function(params) { ... };

Using ‘jQuery’ rather than ‘$’ ensures there are no conflicts with other JavaScript libraries. All our internal code should also refer to ‘jQuery’ rather than ‘$’.

**However, we can save a little typing and reduce the file size using an anonymous function:**

(function($) {

$.fn.reverseText = function(params) { ... };

})(jQuery);

This function runs immediately and is passed jQuery as a parameter named ‘$’. Since ‘$’ is a local variable, we can assume that it always refers to the jQuery library rather than another library that grabbed the global ‘$’ variable first.

## Plugin Parameters

We require two parameters for our plugin: minlength and maxlength. It is easiest to define these as function arguments, e.g.

(function($) {

$.fn.reverseText = function(minlength, maxlength) { ... };

})(jQuery);

// example

$("p").reverseText(0, 100);

But what if we decide to add further parameters later? Our plugin could have dozens of options — parameter handling would quickly become convoluted. As an alternative, we could pass a single JSON object, e.g.

(function($) {

$.fn.reverseText = function(params) { ... }

})(jQuery);

// example

$("p").reverseText( { minlength: 0, maxlength: 100 } );

The first line in our reverseText function should define a set of default parameters then *overwrite* these with any user-defined values. The jQuery extend function can handle this for us:

// merge default and user parameters

params = $.extend( {minlength: 0, maxlength: 99999}, params);

Therefore, **params.minlength** is 0 and **params.maxlength** is 99999 unless the calling code overrides those values.

## The Plugin Code

We can now write our main plugin code:

// traverse all nodes

this.each(function() {

// express a single node as a jQuery object

var $t = $(this);

// find text

var origText = $t.text(), newText = '';

// text length within defined limits?

if (origText.length >= params.minlength && origText.length <= params.maxlength) {

// reverse text

for (var i = origText.length-1; i >= 0; i--) newText += origText.substr(i, 1);

$t.text(newText);

}

});

Explanation:

1. The this.each function traverses all the jQuery DOM nodes and calls an anonymous function.
2. Within the function, ‘this’ contains a single node. A jQuery node collection is assigned to $t so we can run jQuery methods.
3. The variable origText is assigned the text string within the DOM node. newText is set to an empty string.
4. If the length of origText falls between params.minlength and params.maxlength, the loop creates a reversed text string in newText. The DOM node is then updated accordingly.

## Don’t Break the Chain!

Finally, we should remember to return the jQuery object so other methods can be chained:

return this;

## The Completed Code

Our plugin code is now complete:

(function($) {

// jQuery plugin definition

$.fn.reverseText = function(params) {

// merge default and user parameters

params = $.extend( {minlength: 0, maxlength: 99999}, params);

// traverse all nodes

this.each(function() {

// express a single node as a jQuery object

var $t = $(this);

// find text

var origText = $t.text(), newText = '';

// text length within defined limits?

if (origText.length >= params.minlength && origText.length <= params.maxlength) {

// reverse text

for (var i = origText.length-1; i >= 0; i--) newText += origText.substr(i, 1);

$t.text(newText);

}

});

// allow jQuery chaining

return this;

};

})(jQuery);

This file is saved as jquery.reversetext.js. We can then include it in any HTML page after the jQuery library has loaded, e.g.

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">

<html lang="en">

<head>

<title>jQuery plugin: reverseText demonstration</title>

</head>

<body>

<h1>jQuery plugin: reverseText</h1>

<p>This jQuery plugin reverses all the text in the selected nodes.</p>

<ul>

<li>This text will be reversed</li>

<li>This text will not be reversed</li>

<li>reversed</li>

<li>not reversed</li>

</ul>

<script type="text/javascript" src="jquery-1.3.2.min.js"></script>

<script type="text/javascript" src="jquery.reversetext.js"></script>

<script type="text/javascript">

// reverse even-numbered LI tag text

$("ul li:even").reverseText();

</script>

</body>

</html>

Jquery Widget Implementation:

<http://www.erichynds.com/blog/tips-for-developing-jquery-ui-widgets>

<http://learn.jquery.com/jquery-ui/widget-factory/extending-widgets/>

<http://net.tutsplus.com/tutorials/javascript-ajax/an-in-depth-review-of-jquerys-widget-factory/>

The widget factory is a simple function on the global jQuery object - jQuery.widget - that accepts 2 or 3 arguments.

jQuery.widget( "namespace.widgetname", optional - an existing widget prototype to inherit from , An object literal to become the widget's prototype{...});

## \_init() and \_create()

The widget factory automatically fires the \_create() and \_init() methods during initialization, in that order.

At first glance it appears that the effort is duplicated, but there is a sight difference between the two. Because the widget factory protects against multiple instantiations on the same element, \_create() will be called a maximum of one time for each widget instance, whereas \_init() will be called each time the widget is called without arguments:

Ex:

$(**function**(){

*// \_create() and \_init() fire on the first call.*

$("div").mywidget();

*// a widget has already been instantiated on the div, so this time*

*// only \_init will fire.*

$("div").mywidget();

*// however, once the widget is destroyed...*

$("div").mywidget("destroy");

*// both \_create() and \_init() will fire.*

$("div").mywidget();

});

Jquery Plug-In Vs. Jquery Widget: