

Advanced JavaScript

Vinod

Agenda

- Basics
 - Dynamic Typing
 - Globals, Closures
 - Variable scope in JavaScript
 - JavaScript Context
 - JavaScript Objects and Prototypes
- Patterns in JavaScript
 - Need design patterns in JavaScript
 - Commonly used object patterns
 - Asynchronous Module Pattern
 - Pub/Sub design pattern
 - Promises

(Data)Typing in JavaScript

- Languages that have static typing or dynamic typing are said to be "static typed" or "dynamic typed".

Static Typing

- Static typed programming languages are those in which variables need not be defined before they're used.
- This implies that static typing has to do with the explicit declaration (or initialization) of variables before they're employed.
 - C, C++ and Java are examples of a static typed languages.
 - Note that, variables can be cast into other types, but they don't get converted; you just read them assuming they are another type.

Static Typing

- Static typing does not imply that you have to declare all the variables first, before you use them.
- Variables maybe be initialized anywhere, but developers have to do so before they use those variables anywhere.

Dynamic Typing

- Dynamic typed programming languages are those languages in which variables must necessarily be defined before they are used.
- This implies that dynamic typed languages do not require the explicit declaration of the variables before they're used.
 - JavaScript, PHP, Python are examples of a dynamic typed programming language.

Primitives Vs Objects

- JavaScript is object based
- Not everything in JavaScript is object

Primitive (value) types

- String
- Number
- Boolean
- Undefined
- None

Object (reference) types

- Object
- Array
- Function
- Date
- RegExp

Primitive or value types

- These do not have properties or methods
- For example, string values do not have any properties.
 - If you try to access a property or method using a string, an Object is created, used and then destroyed

```
name = "Vinod";  
console.log(name.length)
```

```
var name = new String("Vinod");  
console.log(name.length)
```

```
typeof name
```


Primitives Vs Objects

```
var num = 10;
```

```
console.log(typeof num);
```

number

```
num.createdBy = "Vinod";
```

No error, but will not be assigned

```
console.log(num.createdBy);
```

undefined

```
num = new Number(10);
```

```
console.log(typeof num);
```

number

```
num.createdBy = "Vinod";
```

successfully creates property

```
console.log(num.createdBy);
```

Object

Creating objects - 1

- Use the `Object()` constructor to create an object
- Assign properties and methods

Creating objects - 1

```
var person = new Object();  
person.firstName = "Vinod";  
person.lastName = "Kumar";  
person.city = "Bangalore";  
person.sayHi = function(){  
    return "Hi";  
}
```

Creating Objects - 2

- Use the object literal method
- {}
- define property name/value pairs in a:b format

Creating Objects - 2

```
var person = {  
    firstName : "Vinod",  
    lastName  : "Kumar",  
    city      : "Bangalore",  
    sayHi: function(){  
        return "Hi";  
    }  
};
```

Which method to use?

- Method 2 has the following advantages:
 - Performance
 - JavaScript can execute a single complex statement much faster than multiple simple statements
 - Organization of code
 - Lesser code

Factory method for creating objects

- Use a function that accepts parameters and return an object

```
function createPerson(firstName, lastName, city){  
    return {  
        firstName : firstName,  
        lastName : lastName,  
        city : city,  
        sayHi: function(){  
            return "Hi";  
        }  
    };  
}
```

```
var p1 = createPerson("Vinod", "Kumar", "Bangalore");  
var p2 = createPerson("Ross", "Scott", "Dallas");
```

Factory method for creating objects

- Factory methods for creating objects should be used only if you multiple objects to be created
 - Using the factory method for creating a single object is an overkill

“this” keyword

- Consider the following code:

```
var person = {  
  name: "Vinod",  
  location: "Bangalore",  
  info: function(){  
    return "I am from " + location  
  }  
}
```

- Calling person.info() method will result in an error or unexpected output

```
> person.location  
« "Bangalore"  


---

  
> person.info()  
« "I am from http://jsbin.com/eyozat/2/edit"
```

“this” keyword

- JavaScript will look for the variables “name” and “city” in the global scope
 - The “window” object
- Solution to this would be to use the variable name to reference the members of that object in context.

```
var person = {  
  name: "Vinod",  
  location: "Bangalore",  
  info: function(){  
    return "I am from " + person.location  
  }  
}
```

“this” keyword

- There are few problems with this approach
 - If we decide to change the name of the variable person, then it should be done manually for all the methods that use the members
 - What if we have an anonymous object?
 - How do we address the factory method?
- The solution is the use of “this”

The “global” object

- Whenever you create a global variable or global methods, they are created as members of the “global” (window) object.

“this” and “global”

- Consider this code:

```
window.name = "Vinod";  
info = function(){  
    return "My name is " + this.name;  
}
```

```
var p1 = {  
    name : "Kumar",  
    info : window.info  
};
```

```
> info()
```

```
<< "My name is Vinod"
```

```
> p1.info()
```

```
<< "My name is Kumar"
```

The value of “this” depends on the function that it is attached to.

Even though, we have assigned the window.info method, JavaScript copies the function object from window and assigns it to the member p1.info

“this” - Context Switching

```
var shyam = {  
  name : "Shyam",  
  info : function(){  
    return this.name;  
  }  
}  
  
var vinod = {  
  name : "Vinod",  
  info : function(){  
    return "My name is " + this.name  
      + " and my friend's name is " + this.friendInfo();  
  },  
  friendInfo: shyam.info  
}
```

Console

```
> vinod.info()
```

```
<< "My name is Vinod and my friend's name is Vinod"
```

“this” - Context Switching

```
> vinod
<< {"friendInfo": function () {
    return this.name;
}, "info": function () {
    return "My name is " + this.name + " and my friend's name is " +
    this.friendInfo();
}, "name": "Vinod"}
```

Notice that the method “vinod.friendInfo” no longer has any reference to the function object “shaym.info”.

“this” - Context Switching

- We can associate the “this” keyword with another object during the method execution of another object’s context
- This is done using the “bind” method of the function object.

```
friendInfo: shyam.info.bind(shyam)
```


“this” - Context Switching

Console

```
> vinod.info()
```

```
<< "My name is Vinod and my friend's name is Shyam"
```

```
> vinod
```

```
<< {"friendInfo": function () {  
    [native code]  
  }, "info": function () {  
    return "My name is " + this.name + " and my  
    friend's name is " + this.friendInfo();  
  }, "name": "Vinod"}
```

“this” - Context Switching

- Very useful when
 - you need to assign a callback function that should be executed in its own context
 - if you want to handle an event, but you don't want the event handler to execute within the context of the event target

A more practical requirement

```
var makeRequest = function(url, callback){
    var data = 10;
    callback(data);
}

var obj = {
    someValue : 20,
    loadData: function(data){
        var sum = this.someValue + data;
        alert("sum = " + sum);
    },
    prepareRequest: function(){
        var url = "http://vinod.co/number-service";
        makeRequest(url, this.loadData.bind(this));
    }
}
```

Closure

- A "closure" is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).

Closure

- Closures are one of the most powerful features of JavaScript.
- They are relatively easy to create, even accidentally, and their creation has potentially harmful consequences, particularly in some relatively common web browser environments.
- This depends heavily on the role of scope chains in identifier resolution and so on the resolution of property names on objects.

Closure

- JavaScript allows us to define a function inside another function (inner functions)
- These inner functions have access to all of the local variables, parameters and declared inner functions within their outer function(s).
- A closure is formed when one of those inner functions is made accessible outside of the function in which it was contained, so that it may be executed after the outer function has returned.

Closure

- At which point it still has access to the local variables, parameters and inner function declarations of its outer function.
- Those local variables, parameter and function declarations (initially) have the values that they had when the outer function returned and may be interacted with by the inner function.

Closure

- A closure is formed by returning a function object that was created within an execution context of a function call from that function call and assigning a reference to that inner function to a property of another object.

Closure

```
function exampleClosureForm(arg1, arg2) {  
    var localVar = 8;  
    function exampleReturned(innerArg) {  
        return ((arg1 + arg2) / (innerArg + localVar));  
    }  
    return exampleReturned;  
}  
  
var globalVar = exampleClosureForm(2, 4);
```

What can be done with Closures?

- setTimeout with function references
- Associating functions with object instance Methods
- Encapsulating related functionality

Accidental Closures

- Rendering any inner function accessible outside of the body of the function in which it was created will form a closure.
- That makes closures very easy to create
 - javascript authors (who do not appreciate closures as a language feature), can observe the use of inner functions for various tasks and employ inner functions, with no apparent consequences, not realising that closures are being created or what the implications of doing that are.

Closure

- Accidentally creating closures can have harmful side effects
 - Memory leak
 - Impact on the efficiency of code.
 - It is not the closures themselves, but the use of inner functions that can impact on efficiency.
 - Carefully used, they can contribute significantly towards the creation of efficient code.

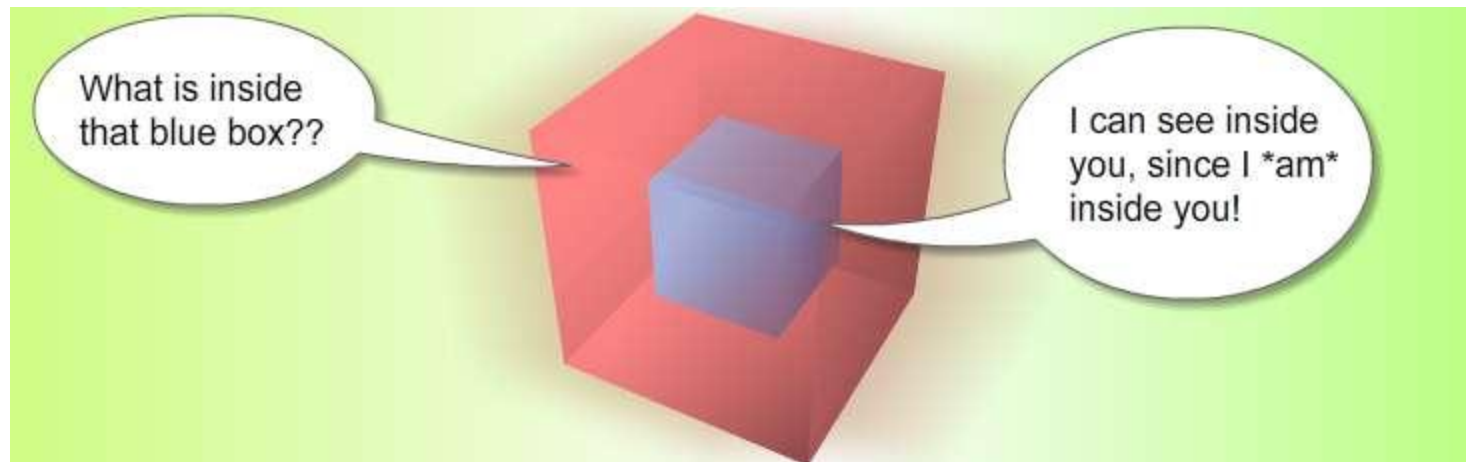
Scopes and Context

Scope

- Applies to the variable and functional access of a function,
- Scope is function-based,

Context

- Is the property and method access of a function.
- Context is object-based.



Scope

- Here is a simple example to explain the “Scope”
- funcA() has access to variable “a” and funcB() has access to both “a” and “b”

```
var funcA = function(){  
    var a = 1;  
    var funcB = function(){  
        var b = 2;  
        console.log(a, b); // outputs: 1, 2  
    }  
    console.log(a, b); // Error! b is not defined  
}  
funcA();
```

Scope

- The second log fails because a function captures its scope, and keeps it all to itself.
- Nobody else has access to funcB's scope, not even funcA.
- However, when funcA captured its scope, it included funcB and therefore funcB has access to all of funcA's goodies.

Context

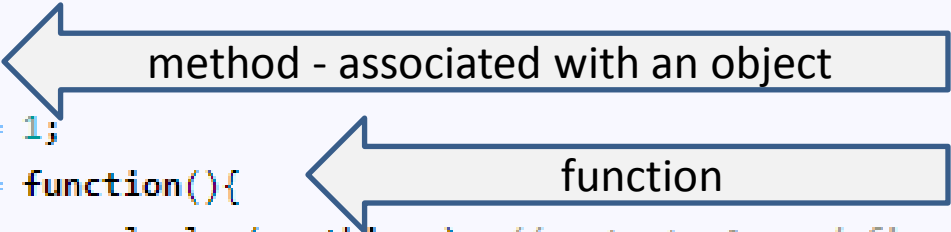
- The term context can loosely be related to usage of the keyword “this”.

```
var o = {  
  x:10,  
  m: function(){  
    var x = 1;  
    console.log(x, this.x); // outputs 1, 10  
  }  
}  
o.m();
```

Both x's are able to be accessed because **this.x** is a property within the m() method's context and **var x** is a variable within its scope.

Broken context

```
var o = {  
  x:10,  
  m: function(){  
    var x = 1;  
    var f = function(){  
      console.log(x, this.x); // outputs 1, undefined  
    }  
    f();  
  }  
}  
o.m();
```



method - associated with an object

function

The reason for the error is that the function `f()` was treated as a method.

`x:10` → defines a property of an object

`var x = 1` → creates a variable with a local scope

Solution

```
var o = {  
  x:10,  
  m: function(){  
    var x = 1;  
    this.f = function(){  
      console.log(x, this.x); // outputs 1, 10  
    }  
    this.f();  
  }  
}
```

Scopes and Context

- Knowing the difference between scope and context will not only help you understand the JavaScript language better, but you'll be better able to communicate with other developers and ask the proper questions.
- Likewise, knowing the difference between methods and functions, and properties and variables is important to communicate exactly what it is you're referring to.
- Remember, `var x` and `this.x` are not the same thing!

Patterns in JavaScript

Agenda

- Need of design patterns in JavaScript
- Common Object Pattern
- Asynchronous Module Pattern
- Observer Pattern
- Publisher / Subscriber Pattern
- Promises

What is a design pattern?

- A reusable software solution to a specific type of problem that occurs frequently when developing software.
- Over the many years of practicing software development, experts have figured out ways of solving similar problems.
- These solutions have been encapsulated into design patterns.
- Patterns are not an exact solution.

Benefits of design pattern

- Patterns are proven solutions to software development problems
- Patterns are scalable as they usually are structured and have rules that you should follow
- Patterns are reusable for similar problems

Benefits of design pattern

- Reusing patterns assists in preventing minor issues that can cause major problems in the application development process.
- Patterns can provide generalized solutions which are documented in a fashion that doesn't require them to be tied to a specific problem.

Benefits of design pattern

- Certain patterns can actually decrease the overall file-size footprint of our code by avoiding repetition.
- Patterns add to a developers vocabulary, which makes communication faster.

Types of Design Patterns

- Creational patterns
- Structural design patterns
- Behavioral patterns

Creational patterns

- Focus on ways to create objects or classes.
- This may sound simple (and it is in some cases), but large applications need to control the object creation process.
 - Builder Pattern
 - Prototype Pattern

Structural design patterns

- Focus on ways to manage relationships between objects so that your application is architected in a scalable way.
- A key aspect of structural patterns is to ensure that a change in one part of your application does not affect all other parts.
 - Composite Pattern
 - Facade Pattern

Behavioral patterns

- Focus on communication between objects.
- Not organizing that communication can lead to bugs that are difficult to find and fix.
- Behavioral design patterns prescribe different methods of organizing the communication between objects.
 - Observer Pattern
 - Mediator Pattern

Builder Pattern

- Allows us to construct objects by only specifying the type and the content of the object.
- We don't have to explicitly create the object.

```
var myDiv = $('<div id="myDiv">This is a div.</div>');  
//myDiv now represents a jQuery object referencing a DOM node.  
var someText = $('<p/>');  
//someText is a jQuery object referencing an HTMLParagraphElement  
var input = $('<input />');
```

Prototype Pattern

- The prototype pattern is a pattern where objects are created based on a template of an existing object through cloning.

```
var Person = {  
    numFeet: 2,  
    numHeads: 1,  
    numHands: 2  
};  
  
//Object.create takes its first argument and  
// applies it to the prototype of your new object.  
  
var tilo = Object.create(Person);  
  
console.log(tilo.numHeads); //outputs 1  
tilo.numHeads = 2;  
console.log(tilo.numHeads) //outputs 2
```

Composite Pattern

- The composite pattern says that a group of objects can be treated in the same manner as an individual object of the group.

```
$('.myList').addClass('selected');  
$('#myItem').addClass('selected');  
//dont do this on large tables, it's just an example.  
$("#dataTable tbody tr").on("click", function(event){  
    alert($(this).text());  
});  
$('#myButton').on("click", function(event) {  
    alert("Clicked.");  
});
```


Facade Pattern

- The Facade Pattern provides the user with a simple interface, while hiding it's underlying complexity.
- The Facade pattern almost always improves usability of a piece of software.

```
$(document).ready(function() {  
    //all your code goes here...  
});
```

ready() is not simple, it makes our work simpler

```
ready: (function() {  
    ...  
    //Mozilla, Opera, and Webkit  
    if (document.addEventListener) {  
        document.addEventListener("DOMContentLoaded", idempotent_fn, false);  
        ...  
    }  
    //IE event model  
    else if (document.attachEvent) {  
        // ensure firing before onload; maybe late but safe also for iframes  
        document.attachEvent("onreadystatechange", idempotent_fn);  
        // A fallback to window.onload, that will always work  
        window.attachEvent("onload", idempotent_fn);  
        ...  
    }  
})
```

jQuery normalizes the browser inconsistencies to ensure that ready() is fired at the appropriate time.

Observer Pattern

- In the Observer Pattern, a subject can have a list of observers that are interested in it's lifecycle.
- Anytime the subject does something interesting, it sends a notification to its observers.
- If an observer is no longer interested in listening to the subject, the subject can remove it from its list.

Observer Pattern

- We need three methods to describe this pattern:
 - `publish(data)`
 - Called by the subject when it has a notification to make. Some data may be passed by this method.
 - `subscribe(observer)`
 - Called by the subject to add an observer to its list of observers.
 - `unsubscribe(observer)`
 - Called by the subject to remove an observer from its list of observers.

Observer Pattern

- The Observer pattern is one of the simpler patterns to implement, but it is very powerful.
- JavaScript is well suited to adopt this pattern as it's naturally event-based.
- Applications should be developed based on modules that are loosely-coupled with each other and adopt the Observer pattern as a means of communication.

Observer Pattern

- The observer pattern can become problematic, if there are too many subjects and observers involved.
- This can happen in large-scale systems.

Observer Pattern

```
var o = $( {} );
$.subscribe = o.on.bind(o);
$.unsubscribe = o.off.bind(o);
$.publish = o.trigger.bind(o);
// Usage
document.on( 'tweetsReceived', function(tweets) {
    //perform some actions, then fire an event
    $.publish('tweetsShow', tweets);
});
//We can subscribe to this event and then fire our own event.
$.subscribe( 'tweetsShow', function() {
    //display the tweets somehow
    ..
    //publish an action after they are shown.
    $.publish('tweetsDisplayed');
});
$.subscribe('tweetsDisplayed', function() {
    ...
});
```

Publish/Subscribe

- This is a variation of the observer pattern
- Dependency between the subject and the observer is removed
 - The Observer pattern requires that the observer (or object) wishing to receive topic notifications must subscribe this interest to the object firing the event (the subject).

Publish/Subscribe pattern

- The Pub/Sub pattern however uses a topic/event channel which sits between the objects wishing to receive notifications (subscribers) and the object firing the event (the publisher).
- This event system allows code to define application specific events which can pass custom arguments containing values needed by the subscriber.

Observer Vs Pub/Sub pattern

- The publish/subscribe pattern allows any subscriber implementing an appropriate event handler to register for and receive topic notifications broadcast by the publisher.

Mediator Pattern

- The Mediator Pattern promotes the use of a single shared subject that handles communication with multiple objects.
- All objects communicate with each other through the mediator.
- Mediator pattern is often used as a system gets overly complicated.
- By placing mediators, communication can be handled through a single object, rather than having multiple objects communicating with each other.

Mediator Pattern

```
$('#album').on('click', function(e) {
    e.preventDefault();
    var albumId = $(this).id();
    mediator.publish("playAlbum", albumId);
});

var playAlbum = function(id) {
    //...
    mediator.publish("albumStartedPlaying", {
        songList: [...], currentSong: "Without You"
    });
};

var logAlbumPlayed = function(id) {
    //Log the album in the backend
};

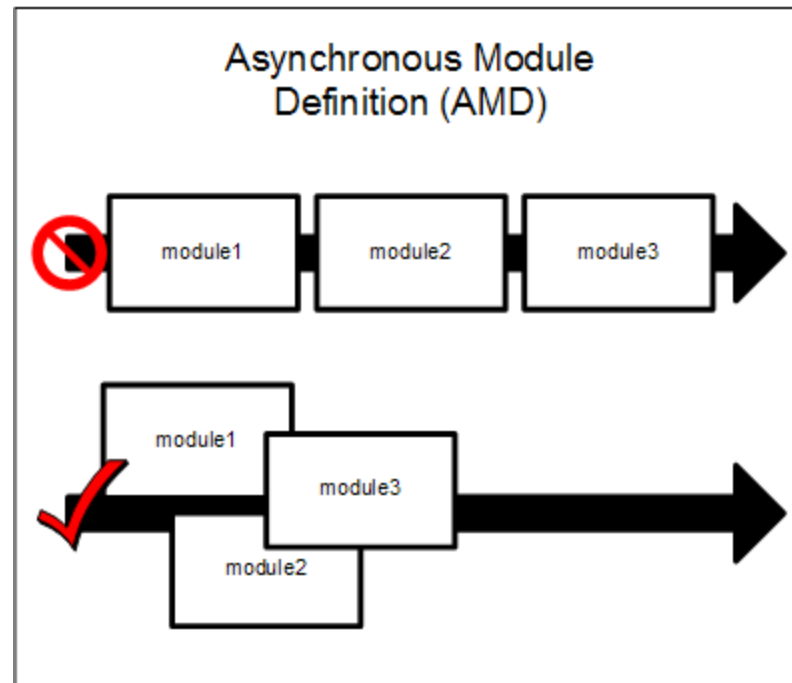
var updateUserInterface = function(album) {
    //Update UI to reflect what's being played
};

//Mediator subscriptions
mediator.subscribe("playAlbum", playAlbum);
mediator.subscribe("playAlbum", logAlbumPlayed);
mediator.subscribe("albumStartedPlaying", updateUserInterface);
```

Other patterns

- Asynchronous Module Pattern
- Publisher / Subscriber Pattern
- Promises

Asynchronous Module Definition



Instead of loading files one after the other, AMD can load them all separately, even when dependent on each other

Asynchronous Module Definition

- Asynchronous module definition (AMD) is a JavaScript API for defining modules such that the module and its dependencies can be asynchronously loaded.
- It is useful in improving the performance of websites by bypassing synchronous loading of modules along with the rest of the site content.

Promises

- Asynchronous patterns are becoming more common and more important to moving web programming forward.
- To make asynchronous patterns easier, JavaScript libraries (like jQuery and Dojo) have added an abstraction called promises (or sometimes deferreds).
- With these libraries, developers can use promises in any browser with good ECMAScript 5 support.

Promises

- When you make an asynchronous call, you need to handle both successful completion of the work as well as any potential errors that may arise during execution.
- Upon the successful completion of one asynchronous call, you may want to pass the result into make another Ajax request.
- This can introduce complexity through nested callbacks.

Promises

- The nested callbacks make the code hard to understand
 - what code is business logic specific to the app and
 - what is the boilerplate code required to deal with the asynchronous call?
- In addition, due to the nested callbacks, the error handling becomes fragmented.
 - We must check several places to see if an error occurred.

Promises

- Represents the result of a potentially long running and not necessarily complete operation.
- Instead of blocking and waiting for the long-running computation to complete, the pattern returns an object which represents the promised result.

Promises

- An example of this might be making a request to a third-party system where network latency is uncertain.
- Instead of blocking the entire application while waiting, the application is free to do other things until the value is needed.
- A promise implements a method for registering callbacks for state change notifications, commonly named the “then” method:

Promises

- At any moment in time, promises can be in one of three states:
 - unfulfilled
 - resolved
 - rejected (failed)


CommonJS Promise/A proposal

- Has several derivatives in popular libraries.
- The then method on the promise object adds handlers for the resolved and rejected states.
- This function returns another promise object to allow for promise-pipelining, enabling the developer to chain together async operations where the result of the first operation will get passed to the second.

```
then(resolvedHandler, rejectedHandler);
```

- The resolvedHandler callback function is invoked when the promise enters the fulfilled state, passing in the result from the computation.
- The rejectedHandler is invoked when the promise goes into the failed state.

Me.

- Vinod Kumar
- vinod@knowledgeworksindia.com
-  97-314-24784