**JAvascript design patterns**

# Section 1: INTRODUCTION to the course

## 0101 Course Introduction

### Welcome & Instructor Intro

Hello and welcome to the JavaScript Design Pattern course, where you will acquire the skills to produce software more efficiently, which is both adaptable and decipherable many for years to come.

My name is Jeff and I have been coding since 1983. I have been a professional developer for the past 7 years, working in several different industries, using a variety of languages and frameworks. I am currently a senior developer at a company which produces software for the health and social care industry.

### Benefits

This course has been designed for developers wishing to improve their coding standards.

By the end of this course you will be able to identify patterns at the backlog refinement stage, improve communication with colleagues utilising pattern references and then implement the patterns into your JavaScript code. This will facilitate development speed and more importantly future adaption and readability.

### List of Major Components

We’ll be exploring 2 main sections of design pattern; the classic Gang of Four collection of 23 patterns and then a further selection application level and code organisation structural patterns.

### Ideal Student Description

The ideal student for this course is any JavaScript developer that wishes to improve and formalise their coding standards. It is expected that you already have a foundation understanding of JavaScript development.

### Thank You & Call to Action

Feel free to look through the course description and I look forward to lighting your path to cleaner code.

# Section 2: Cleaner code

## 0201 Introduction

### Code durability

How durable is your code? How long will it survive before someone, maybe you, probably you, will replace it with a newer, shinier version? How much of this shiny new code actually does something new? If you think about your past projects, you’ll probably find that the majority of the time an updated, refactored or reengineered version of a solution was only requested due to the necessity of interfacing with a more modern framework or application. Wouldn’t it great if the original code had been written future proof, or at least in a way that made it easily adaptable?

### Section Overview

In this section we will explore the potential scruffiness of JavaScript code and the role of the design pattern in offering a proposed solution. Imagine discovering a world where developed solutions only have to be designed and coded once and any adaptions to these solutions could easily be implemented due to the formal design and simplified readability of the code?

Prepare to enter the JavaScript design pattern zone.

## 0202 Clean Code

### unclean code

So before we can investigate methods for cleaning up our JavaScript code, we need to identify what exactly is unclean code. Think of the number of times you’ve refined a set of requirements, pondered over a resolution, searching for a solution which you can develop swift enough to keep your bosses contented with sufficient complexity and maybe new technologies to ensure ample satisfaction. Did you stop to think who will read your code? Did you contemplate the durability of the pending coded solution? If not, it may surprise you to realise that over time more people will read a line of you code than has ever read it. Put plainly, each year, more and more developers, and probably not you, will have to read and reread your lines of code as adaptions are produced; and many of these developers, as you most certainly have done previously, will curse the codes architects.

### refactoring

But still we are only identifying the necessity for clean code, not its definition. So imagine a room which you use for storage. You have an item to store, the door is opened, the item is placed into the room and the door is closed. This process is repeated for a year, with new items added daily. Imagine the mess of the room. Probably not, as you just open the door and add a new item; in fact the messiness of the room would only be noticed when adding new items became troublesome and then maybe you would schedule a date for a clean-up, a time to refactor the room. Have you ever asked your boss for time to refactor some code and when they ask why, you answer because it has been coded in a messy way. Why was it coded it this messy way they ask? You mostly answer, a previous developer did it. Meanwhile the development time has increased, incurring more costs, suffering detrimental opinions on the quality of the code and its developers.

So we could say that messy code is code that constantly requires refactoring for adaption, or code that is very difficult to read, or even worse code that has been developed in such a way that adaption is impossible without a complete rewrite.

Going back to the messy room example; we have to consider what if the items had been placed in a suitable position within the room from the outset and what if each person adding an item understood this cataloguing system. This sort of organisation involves both a simple and concreate cataloguing system that everyone agrees with and respects and any new persons can easily pickup. This is the basis of a design pattern.

## 0203 Patterns and Anti Patterns

### Design Patterns

The purpose of a design pattern is not to define exactly how to develop a solution, but instead to guide on methods of resolving common problems. They are more a schema for a solution rather than a set of coding answers. By providing simple, reusable, commonly used, formal designs; groups of patterns can be efficiently identified within a proposed solution. This greatly facilitates communication within the team and simplifies future developers interaction with the code due to them quickly identify the patterns used. It is important to reiterate that the design of the solution, comprising of carefully selected, appropriate patterns is more important than the code that formulates each pattern, although the way in which patterns are coded must aim to be as clean and simple as possible which will be demonstrated later in this course. Choosing inappropriate patterns for a design is worse than not using patterns at all; resulting in messy code that other developers interpret incorrectly. Although incorrectly coding of a pattern can result in the need to refactor; the time taken to refactor small isolated sections of pattern code is far smaller than the time taken to redesign the entire solution.

### Gang Of Four

The 23 most common design patterns come from a book called; Design Patterns: Elements of Reusable Object-Oriented Software from Addison-Wesley. These patterns are often referred to as the Gang of Four, due to the book being written by 4 authors. These 23 patterns are divided in 3 main categories, which will be explored in detail in later sections of this course.

Creational Patterns: These illustrate methods of creating objects

Structural Patterns: These illustrate methods of adapting existing objects

Behavioural Patterns: These illustrate interactions between objects

### Other Patterns

This course will also explore other design patterns including application and data structure patterns, as well as JavaScript specific patterns.

### Anti Patterns

Just as there are patterns that improve software design, there exists a collection of anti-patterns which are detrimental to a design. Although this course concentrates on design patterns, in a later section a few of these anti-patterns will be explored in an attempt to highlight common mistakes. Many anti-patterns are introduced during the refactoring of code and although it would be wonderful to design and create software that didn’t require refactoring, we rarely work on green field projects and so interfacing with aging messy code will probably be a daily chore for all of us.

## 0204 JavaScript Patterns and Frameworks

### Object-Oriented JavaScript

As stated earlier, the 23 most common design patterns come from a book called; Design Patterns: Elements of Reusable Object-Oriented Software from Addison-Wesley. A key phase from the title of this book, when referring to JavaScript is object-oriented software, due to the fact that JavaScript does not adhere to the general object-oriented design structure. This does not mean that JavaScript is not an object-oriented language, more that it is structured differently than say languages like VB, C# and Java.

Many examples of JavaScript design patterns found on the internet and some modern frameworks attempt to emulate the traditional object-oriented structure within JavaScript by means of mocking classes and inheritance, etc. But this is working against the fundamentals of JavaScript’s structure. Throughout this course and in the next section in particular, the structure of JavaScript will be explored and utilised in the implementation of the design patterns. Each pattern will not just be converted to JavaScript, more each pattern will be analysed and a new, revised JavaScript version will be created. Of cause the basic premise of the design will remain, or else it would no longer be a formal universal pattern design.

### Fashionable Frameworks

Another major obstacle, contributor, saviour, of the messy code is the JavaScript framework. Their main goal is to facilitate the development of a solution by reducing the amount of code required. The ability to write 1 line of framework code to do the same work as 20 lines of JavaScript is very appealing; which is why we all use some sort of JavaScript library or framework in our daily development.

The problems arise when we choose a fashionable framework to develop a solution and then 5 years later have to refactor the entire project due to either a new fashionable framework arriving or limitations arising from the original framework selection. This doesn’t mean that frameworks always make messy code or should be avoided; more that they should be selected and implemented carefully. As highlighted earlier, a framework which adheres to emulate classic object-oriented design in JavaScript, is ultimately adding more code than is required just to make it easier for developers to transit between languages. This is more code to be processed when the code is execute, making the application run slower, just because it is easier to learn that using JavaScript correctly.

Choose your JavaScript frameworks and libraries appropriately, so that they not only solve a particular problem, but that they could easily be replaced later without having to rewrite and redesign the entire solution.

## 0205 Exercise

Design for course project (NodeJS dynamic API with AngularJS frontend)

## 0206 Exercise Solution

## 0207 Section Outro

It is hoped that you now understand the role of the design pattern and in particular the JavaScript design pattern within your daily development cycle. Maybe you are already thinking about your recent projects with regards to patterns. Although you may now be tempted to jump over to the sections of this course that mention the patterns that look the most interesting, you are advised to follow this course sequentially. Unlike other design pattern documentation which acts more like a reference guide, in this course each section has specifically been organised to evolve your understanding.

In the next section we will explore the structuring of your JavaScript code.

## 0208 Quiz

Which of the following is **not** one of the three main ‘Gang of Four’ design pattern categories?

1. Creational Patterns
2. Functional Patterns
3. Structural Patterns
4. Behavioural Patterns

Which of the following is **not** an accurate definition of messy code?

1. Code which constantly requires refactoring for adaption
2. Code which is very difficult to read
3. Code which has been created by some other developer
4. Code which has been developed in such a way that adaption is impossible without a complete rewrite

Why must JavaScript design patterns be structured differently to the many examples of patterns found on the internet?

1. JavaScript does not adhere to the general object-oriented design structure
2. JavaScript requires a framework to implement design patterns
3. JavaScript is not an object-oriented language
4. JavaScript can only implement design patterns by means of mocking classes and inheritance

# Section 3: Javascript structure

## 0301 Introduction

The aim of this section is to provide a basic structural over view of JavaScript code whilst highlighting a few simple but effective structural design patterns. An initial requirement of this course was foundation JavaScript awareness. This section will add to or revise your bass JavaScript knowledge, dependent upon your current skill set, by exploring the wonderful world of Scope and in particular Global Scope. Additionally dependency injection will be identified as a means of facilitating effective unit testing. Finally JavaScript objects will be observed with respect to object inheritance.

## 0302 scope

One of the main characteristics of JavaScript which can significantly improve or pollute its cleanliness is the understanding and usage of scope. JavaScript scope denotes the context in which variables, objects and functions are accessed or executed.

**function** *cave*() {  
 **var** lion = 10;  
 **console**.log(lion);}  
*cave*();

We define a function **cave** and within this function we define a variable **lion**; initiated with the value of 10. The value of lion will then output to the console. Now we call the function **cave** and as expected, when the code is executed, we can see the value of 10 output to the console.

**console**.log(lion);

**'use strict'**;

Now we add another console log after the call to the function **cave** and add a **use strict** statement to the head of our example; the reason for this **use strict** statement will be explored in the global scope lecture. The code is then re-executed. Now we still get the value of 10 output as seen previously, but additionally we get a **lion is not defined** exception. This is because the variable **lion** is defined within the scope of the function **cave** and so cannot be observed from outside this function.

**var *lion*** = 20;

To emphasise this we can now add another declaration of the variable **lion** with a different initiated value of say 20. Now when executing the code we get 2 output values, the original 10 and a new output of 20. But how can the variable **lion** be equal to both 10 and 20 at the same time? This is because they are totally different variables, even though they have the same name; as highlighted earlier the **lion** with a value of 10 is scoped to the function **cave** and the **lion** with the value of 20 is scoped to outside the function cave. In its simplest form, scope could be described as an encapsulated section of code. Here the **lion** variable with a value of 10 is encapsulated within the section of code forming the body of the **cave** function; whereas the **lion** variable with a value of 20 is encapsulated within the main body of the file.

Now just to add a little confusion to the code, we’ll remove the declaration of the variable **lion** from within the **cave** function. Now when executed we still get 2 outputs but this time both of them are the value 20. But shouldn’t the first one be a **lion is not defined** exception? No, because when JavaScript requests the use of a variable or object, it first searches the current scope looking for a variable defined with the requested name. But if it cannot find one then it proceeds to the parent scope and has a look there. This process is repeated until either the requested variable definition is found or the search runs out of parent scopes to look in and results in a **not defined** exception.

Although JavaScript is designed to search parent scopes looking for variable definitions, it doesn’t mean it’s correct to develop code reliant on this. The problem with code like this; where JavaScript has to search parent scopes looking for definitions is that so does the developer when reading or debugging the code. It is imperative that a developer can quickly identify what a section of code is doing; and this is extremely difficult when variables are declared miles away from where they are actually used.

## 0303 GLOBAL SCOPE

In the previous lecture we saw that if a variable definition is not found within a scope, then successive parent scopes are searched until no more scopes are found. This final, top level scope is known as the global scope. It is important to note that global scope only really exists in browser based JavaScript and not server side JavaScript such as NodeJS. The reason for this is based around how each separate JavaScript file is loaded. In this lecture we will concentrate on developers of browser based JavaScript and join back up with the server side developers in the modules lecture.

So we create a simple HTML web page which loads 2 JavaScript files. In the first file we define the variable **lion** with an initiated value of 10 and then in the second file we double the value of **lion** and log this to the console. Now you may not be too surprised to see that the value of 20 is output to the console. But how is this possible with the code being split across 2 files? Simple; when the files are loaded they are placed sequentially in memory and effectively act as 1 large file sharing a single global scope.

Now if we swap around the order in which the JavaScript files are loaded, we receive a **not defined** exception for the **lion** variable. This is to be expected due to the fact that we are trying to use the variable before it is defined. But more importantly, this highlights the issue around relying on loading all your JavaScript files in the correct order. But is this the real issue here? Isn’t the real issue that fact that one file relies on variables initiated in another, even though they exist in the same single global scope.

Well this is the major problem with global scope; not only reliance on correct sequential loading of code but also the difficulty in reading and debugging the code due to not knowing which file functions or variables are defined or used in. This is known as contaminating the global scope and should be avoided at all costs to ensure clean and readable code. How to avoid contaminating the global scope will be explored in the next lecture on modules.

**'use strict'**;

**console**.log(lion);

Another thing to point out; as highlighted previously; failure to locate a variable definition results in a **not defined** exception being thrown. But this exception is not thrown by default. In fact the default behaviour is to return an undefined value instead of throwing an error, meaning that the code would continue to run. At first this may seem like a better idea to keep the code running instead of throwing errors. But this method of developing can lead to very messy error prone code. The main reason for this is because, as highlighted previously, all variables should be declared near where they used and by not declaring a variable at all before use is effectively leaving it open to all sorts of potential miss use in the future via code adaption by unsuspecting developers. So it is advised that the **use strict** statement is added to the head of each of your JavaScript files to ensure that you do forget to declare a variable before use.

## 0304 Modules

### Modular JavaScript

Modules are a design pattern which attempt to resolve the global scope contamination issue. They do this by utilising the structure of JavaScript scope, in particular its ability to encapsulate and separate sections of code.

This can be demonstrated by creating a simple formula calculation example. We start by having file 1, which is loaded first, containing 4 methods which preform the basic calculation actions; add, subtract, multiply and divide. Then we add a second file, loaded second, which contains 1 method designed to add 2 values together and then multiply the result by 2. Finally we add a 3rd file, which will be loaded last, containing a call to the formula calculation and a log of the result. So in this small example, it can be seen that all 5 methods that we have created exist in the global scope and can be seen and called directly from any of the files. This is total contamination of the global scope.

Let’s refactor this example to reduce the contamination. First, all of the basic calculation methods can be encapsulated within a single object named **calculator**. This now means that instead of all the calculation methods existing in the global scope, only the new object **calculator** is in the global scope. Then we refactor file 2 in a similar way to file 1. Okay this file still has 1 global scope object as before, but it is free from future contamination. This is due to any new calculation methods being added; exist within the **calculations** object and not the global scope. We are now starting to develop thinking about future adaptions.

We have now refactored the original code to so that each file only creates 1 global scope object, meaning that each file has become a separated section of code with a single entry point. Effectively a modular system; where each file is a separate module.

### Utilising Frameworks

Browser based JavaScript frameworks such as AngularJS are actually built on a modular system, ensuring that all functions are associated with modules and sub module sections known as services.

NodeJS is also based upon a modular system. Here instead of each JavaScript file being loaded and placed sequentially; each NodeJS file is automatically treated as a separate module, with the name of the file being used as the module name. This actual removes the existence of global scope altogether, in fact you actually have to specify which functions are exposed from a NodeJS module.

## 0305 Dependency Injection and Unit Testing

### Testing

The only way we can prove that our developed code does what it is required to; is to test it. Some teams have dedicated testers who not only test the new code but also the areas around the new code which could have been affected. But all this testing takes time and can all the scenarios really be identified by the tester? It would be ideal if a large quantity of this testing was done automatically.

### Unit Testing

Unit testing is the process of automating the testing of each individual method in your code to ensure that it does what is expected, without influence from other methods. There are even methodologies such as TDD, Test Driven Development, which requires the tests to be created prior to the code. So effectively the tests are written, they fail and then code is developed to make them pass. Whichever method you employ, it is important that your code is written in a way that permits adequate testing.

If we take our previous calculator object; we can very easily check if each of the basic operations do what they are meant to. We can test the add method by asserting that 2 known values added together result in the correct result. This can be repeated for all 4 methods.

### Mocking

If we now take the calculations object and attempt to test this, we can, as described before, check that calling the calulateFormula function with 2 known values returns the correct result. But there is a problem here. Even though the correct result is returned, the calulateFormula function is itself calling 2 other functions; the add and multiple functions. Which means that our test is not testing 1 function, but 3. This is against unit testing rules, due to if 1 of the 3 functions fails then the main test will fail, even if that test isn’t at fault. Say someone was to accidently change the add function to subtract. Then the calulateFormula function test would fail due to an incorrect result. Yes it is incorrect and should fail, but no the logic for the calulateFormula function is correct; and as we are testing if logically the function is accurate then the test should not fail.

So how do we test the calulateFormula function without calling either of the add or multiple functions? We can do this by using something called mocking or faking.

### Dependency Injection

Mocking or faking as some people like to call it, is the act of replacing a method with a pretend version. Now the aim of this lecture is not to explore mocking frameworks, of which there are many, but to make the code ready to be mocked.

We do this by ensuring that any function is never directly dependent on any other function external to itself. Okay how do we do this as it can clearly be seen that the calulateFormula function directly calls 2 other methods and it need to too work. Easy, we inject the calculator object into the calulateFormula function and let this function access the calculator functions on this injected object instead of the main object directly. This process is called dependency injection; as it is the act of injecting all required dependencies. Now that the calulateFormula function doesn’t call the calculator object directly but instead accesses the injected object we can use our selected mocking framework to create a fake version of the calculator object to ensure that only the calulateFormula function itself is tested.

## 0306 Inheritance & linked objects

### Classic Inheritance

As stated in the introduction, JavaScript is an object orientated language, but it is structured differently to languages like Java or C#. If you’ve ever come into contact with a traditional object orientated language you will relate to the ideals of interfaces, abstractions and inheritance. Methods belong to classes; when methods are required to be shared between different classes, they are placed into a separate class and then other classes inherit from this separate class. This effectively forms vertical towers of classes which can be large, cumbersome and difficult to unit test. But classic inheritance is very popular due to the simplified structure. Each class specifies exactly what base class it inherits from and a class can only directly inherit from 1 other class. Although JavaScript doesn’t directly support this structure, there are many frameworks which attempt to emulate it and a simulation can be set up utilising prototypes. I’ll quickly run through inheritance emulation in JavaScript just so that afterwards you can compare the code to a proposed alternative.

Say we wish to create a series of shape objects and each requires an area function to calculate the shapes area. First we start with a **baseShapeClass** function which stores the shapes width and height. Then we add an **area** function to this which simply outputs the width multiplied by the height. Now we create our first shape function, a square. The square function only needs to know a width due to both width and height being the same. We now need to ensure that when the **square** function is called that the **baseShapeClass** function is also called as this initiates the width and height. Now currently the **square** function doesn’t know about the existence of the **baseShapeClass** function; so we rectify this by linking them via prototype. Now the **square** function requires its own implementation of the **area** function, which basically routes straight through to the **baseShapeClass** area function. Now we can create a new instance of a square and request to know its area. Wow, that was complicated.

### Composition

Composition is an alternative method of linking classes in traditional object orientated languages. The main premise is that instead of linked classes being inherited from, a property within the class acts as a reference to this linked class. This effectively forms horizontal chains of classes which through the implementation of dependency injection ensure that it can be unit tested. Composition also permits classes to be linked to many other classes in contrast to the 1 on 1 relationship of inheritance, which can confuse developers.

### Linked Objects

JavaScript offers a 3rd method for linking objects which although still utilises prototypal inheritance; it does it in a much simpler and cleaner way, resembling more of a composition like structure.

We’ll start by defining our **baseShape**, but this time it is structured as an object and not a function. As before this includes the area function. Now we define our **square** object, which just consists of a setter width property. Now we require a clean way of linking the **square** object to the **baseShape** object. This is done using Object assign, which effectively merges objects together. In this case the **baseShape** object is merged into the **square** object. This means that the **baseShape** object remains unaffected, but the **square** object is now both of these objects joined together. We can now create an instance of this new square object which prototypal links the width property and area method. The width property of this instance can now be set and the area method called.

Using this pattern it would now be possible to create a rectangle shape. This shape would obviously require both width and height properties. As can be seen the **baseShape** object is simply reused producing clean and readable code.

This process of linking objects will be employed through all design pattern examples throughout this course.

## 0307 Exercise

## 0308 Exercise Solution

## 0309 Section Outro

The aim of this section was to provide a basic structural over view of JavaScript code. It was identified that a modular design can greatly aid with reducing contamination of the global scope; thus rendering the code more organised. Additionally dependency injection was identified as being the liberator of unit tests by ensuring that dependent functions could be mocked. Finally methods of linking objects were explored and a solution identified which greatly improved readability over traditional class based structures.

So now that some basic but fundamental structural design patterns have been discovered, it is time to take our first steps into the world of tradition design patterns. We will start with the Gang of Four creational patterns.

## 0310 Quiz

If a variable definition cannot be found in the current scope and the current scope is not global then

1. An error is thrown
2. The parent scope is searched for the variable definition
3. An undefined value is returned
4. The global scope is searched for the variable definition

The main objective of modular JavaScript design is

1. To separate functions into different files.
2. To create a more readably structure.
3. To reduce the contamination of the global scope.
4. To ensure that functions have their own scope.

The JavaScript command to merge objects together is

1. Object.assign
2. Object.create
3. Object.join
4. Object.link

# Section 4: Creational Patterns

## 0401 Introduction

## 0402 Factory

## 0403 Builder

## 0404 Abstract Factory

## 0405 Singleton

## 0406 Exercise

## 0407 Exercise Solution

## 0408 Section Outro

## 0409 Quiz

# Section 5: Structural Patterns

## 0501 Introduction

## 0502 Adapter

## 0503 Bridge

## 0504 Composite

## 0505 Decorator

## 0506 Façade

## 0507 Flyweight

## 0508 Proxy

## 0509 Exercise

## 0510 Exercise Solution

## 0511 Section Outro

## 0512 Quiz

# Section 6: Behavioral Patterns

## 0601 Introduction

## 0602 State

## 0603 Strategy

## 0604 Template Method

## 0605 Memento

## 0606 Iterator

## 0607 Exercise

## 0608 Exercise Solution

## 0609 Section Outro

## 0610 Quiz

# Section 7: more Behavioral Patterns

## 0701 Introduction

## 0702 Observer

## 0703 Mediator

## 0704 Command

## 0705 Visitor

## 0706 Chain of Responsibility

## 0707 Interpreter

## 0708 Exercise

## 0709 Exercise Solution

## 0710 Section Outro

## 0711 Quiz

# Section 8: Advanced JavaScript Structure

## 0801 Introduction

## 0802 Asynchronous Patterns

## 0803 Multi Thread Patterns

## 0804 Messaging Patterns

## 0805 Exercise

## 0806 Exercise Solution

## 0807 Section Outro

## 0808 Quiz

# Section 9: Data Architecture

## 0901 Introduction

## 0902 Repository

## 0903 Data Mapper

## 0904 Unit of Work

## 0905 Exercise

## 0906 Exercise Solution

## 0907 Section Outro

## 0908 Quiz

# Section 10: Application Architecture

## 1001 Introduction

## 1002 MVC

## 1003 MVP

## 1004 MVVM

## 1005 SOA

## 1006 Exercise

## 1007 Exercise Solution

## 1008 Section Outro

## 1009 Quiz

## Conclusion

# Section 11

1101 Resources

1102 Closing message from instructor