##### 

[JavaScript Async/Await](#_w5vfremur1qb)

[When would you use var in your declaration and when you wouldn't?](#_77v3vf7p2kpb)

[What is the difference between == and ===? Which one would you use?](#_f2o7knb137zf)

[How do you check if a variable is an object](#_7dqzfyoa4wh9)

[Discuss scoping in JavaScript.](#_6zvygbd0wbi6)

[Explain hoisting in JavaScript.](#_xg3v0ech97g6)

[What are closures?](#_yh8koj32zuu7)

[Explain prototypal/differential inheritance](#_uwmu1pmwlf2u)

[What is Strict Mode in JavaScript](#_iibst2feztwz)

[THIS](#_wzbl0jcy1b42)

[How to empty an array in JavaScript?](#_e1a9f5kremdu)

[Difference between undefined and not defined in JavaScript](#_lrlhkwn5w0og)

[Why Should You Not Prefer To Use Global Variables In JavaScript And How Can You Prevent It](#_mxbjhhz86bjn)

[Use Cases for JavaScript's IIFEs](#_yplkggm7xsbx)

[anonymous functions uses for three reasons: Advantages /Disadvantages](#_rmlt3je5flfq)

[What Are Event Handlers In JavaScript And How To Use Them?](#_myfhfz53r7nq)

[Difference between Anonymous vs. referenced vs. declared functions](#_k7yj67za887n)

[What is a Promise?](#_d6sai3mshs4q)

[Why Promises and When to Use Them?](#_opy901y6q5lg)

[Const vs let vs var](#_yjhefz5au1p1)

[Callback hell](#_vn98un26y8bx)

##### 

##### 

##### 

##### 

##### JavaScript Async/Await

For the longest of time JavaScript developers had to rely on callbacks for working with asynchronous code. As a result, many of us have experienced callback hell and the horror one goes through when faced with functions looking like [this](https://tutorialzine.com/media/2017/07/callback-hell.jpg).

Thankfully, then (or should we say .then()) came Promises. They offered a much more organized alternative to callbacks and most of the community quickly moved on to using them instead.

Now, with the most recent addition of Async/Await, writing JavaScript code is about to get even better!

**What is Async/Await?**

Async/Await is a long anticipated JavaScript feature that makes working with asynchronous functions much more enjoyable and easier to understand. It is build on top of Promises and is compatible with all existing Promise-based APIs.

The name comes from async and await - the two keywords that will help us clean up our asynchronous code:

**Async - declares an asynchronous function** (async function someName(){...}).

* Automatically transforms a regular function into a Promise.
* When called async functions resolve with whatever is returned in their body.
* Async functions enable the use of await.

**Await - pauses the execution of async functions**. (var result = await someAsyncCall();).

* When placed in front of a Promise call, await forces the rest of the code to wait until that Promise finishes and returns a result.
* Await works only with Promises, it does not work with callbacks.
* Await can only be used inside async functions.

Here is a simple example that will hopefully clear things up:

Let's say we want to get some JSON file from our server. We will write a function that uses the [axios](https://github.com/mzabriskie/axios) library and sends a HTTP GET request to <https://tutorialzine.com/misc/files/example.json>. We have to wait for the server to respond, so naturally this HTTP request will be asynchronous.

Below we can see the same function implemented twice. First with Promises, then a second time using Async/Await.

// Promise approach  
  
function getJSON(){  
  
 // To make the function blocking we manually create a Promise.  
 return new Promise( function(resolve) {  
 axios.get('https://tutorialzine.com/misc/files/example.json')  
 .then( function(json) {  
  
 // The data from the request is available in a .then block  
 // We return the result using resolve.  
 resolve(json);  
 });  
 });  
  
}  
  
// Async/Await approach  
  
// The async keyword will automatically create a new Promise and return it.  
async function getJSONAsync(){  
  
 // The await keyword saves us from having to write a .then() block.  
 let json = await axios.get('https://tutorialzine.com/misc/files/example.json');  
  
 // The result of the GET request is available in the json variable.  
 // We return it just like in a regular synchronous function.  
 return json;  
}

It's pretty clear that the Async/Await version of the code is much shorter and easier to read. Other than the syntax used, both functions are completely identical - they both return Promises and resolve with the JSON response from axios. We can call our async function like this:

getJSONAsync().then( function(result) {  
 // Do something with result.  
});

**So, does Async/Await make promises obsolete?**

No, not at all. When working with Async/Await we are still using Promises under the hood. A good understanding of Promises will actually help you in the long run and is highly recommended.

There are even use cases where Async/Await doesn't cut it and we have to go back to Promises for help. One such scenario is when we need to make multiple independent asynchronous calls and wait for all of them to finish.

If we try and do this with async and await, the following will happen:

async function getABC() {  
 let A = await getValueA(); // getValueA takes 2 second to finish  
 let B = await getValueB(); // getValueB takes 4 second to finish  
 let C = await getValueC(); // getValueC takes 3 second to finish  
  
 return A\*B\*C;  
}

Each await call will wait for the previous one to return a result. Since we are doing one call at a time the entire function will take 9 seconds from start to finish (2+4+3).

This is not an optimal solution, since the three variables A, B, and C aren't dependent on each other. In other words we don't need to know the value of A before we get B. We can get them at the same time and shave off a few seconds of waiting.

To send all requests at the same time a Promise.all() is required. This will make sure we still have all the results before continuing, but the asynchronous calls will be firing in parallel, not one after another.

async function getABC() {  
 // Promise.all() allows us to send all requests at the same time.   
 let results = await Promise.all([ getValueA, getValueB, getValueC ]);   
  
 return results.reduce((total,value) => total \* value);  
}

This way the function will take much less time. The getValueA and getValueC calls will have already finished by the time getValueB ends. Instead of a sum of the times, we will effectively reduce the execution to the time of the slowest request (getValueB - 4 seconds).

**Handling Errors in Async/Await**

Another great thing about Async/Await is that it allows us to catch any unexpected errors in a good old try/catch block. We just need to wrap our await calls like this:

async function doSomethingAsync(){  
 try {  
 // This async call may fail.  
 let result = await someAsyncCall();  
 }  
 catch(error) {  
 // If it does we will catch the error here.  
 }   
}

The catch clause will handle errors provoked by the awaited asynchronous calls or any other failing code we may have written inside the try block.

If the situation requires it, we can also catch errors upon executing the async function. Since all async functions return Promises we can simply include a .catch() event handler when calling them.

// Async function without a try/catch block.  
async function doSomethingAsync(){  
 // This async call may fail.  
 let result = await someAsyncCall();  
 return result;   
}  
  
// We catch the error upon calling the function.  
doSomethingAsync().  
 .then(successHandler)  
 .catch(errorHandler);

It's important to choose which method of error handling you prefer and stick to it. Using both try/catch and .catch() at the same time will most probably lead to problems.

**Browser Support**

Async/Await is already available in most major browsers. This excludes only IE11 - all other vendors will recognize your async/await code without the need of external libraries.

##### **When would you use var in your declaration and when you wouldn't?**

Always use var. Not using var for variable declaration will traverse scopes all the way up till the global scope. If variable with that name is not found it will declare it in the global scope. Therefore not using var implicitly declares variable in the global scope (which, let me remind you, is a bad practice).

(function() {  
 baz = 5;  
 var bar = 10;  
})();  
  
console.log(baz); *// outputs 5*  
*//console.log(bar); // error: bar is not defined*

Try it: <http://jsfiddle.net/tnajdek/AKxn9/>

A common mistake is to not use var in loops (e.g. for(i=0; i>10; i++) when i has not been previously declared) which will pollute the global scope and in some cases might bear unexpected results.

What does the attribute defer/async do when added to the script tag?

The defer attribute will cause browser to execute script after the document has been parsed. This attribute was first implemented in Internet Explorer 4, then [added to HTML 4](http://www.w3.org/TR/html401/interact/scripts.html#adef-defer) and more recently[HTML 5 spec](http://dev.w3.org/html5/spec/Overview.html#attr-script-defer). You might not have heard of it as it has not been supported by other browsers (Firefox support came in version 3.5 - Gecko 1.9.2). Async is another attribute that can affect how a script is loaded and executed, here is a quote from HTML 5 spec on how this is expected to work:

There are three possible modes that can be selected using these attributes. If the async attribute is present, then the script will be executed asynchronously, as soon as it is available. If the async attribute is not present but the deferattribute is present, then the script is executed when the page has finished parsing. If neither attribute is present, then the script is fetched and executed immediately, before the user agent continues parsing the page.

Note: A somewhat (but not exactly) similar defer behavior can be achieved by placing your script tags at the end of the body tag and that's what is considered to be modern 'best practice'

##### What is the difference between == and ===? Which one would you use?

The equality (==) operator will compare for equality after doing necessary type casting, the identity operator (===) doesn't do any conversions. A good practice suggested by Douglas Crockford is to always use strict equality, These are the operators provided by JavaScript – strict equality and Type converting equality.

Strict equality (===) returns true if the values which it is going to compare have the same data type. Taking an example, “2” will not be equal to 2 i.e. (“2″===2) will return false.

Secondly, Type converting equality (==), automatically converts the variable to value irrespective of the data type. Taking an example, here “2” will be equal to 2 i.e. (“2″===2) will return true.

Summarizing it, double equal (==) is an autotype converting equality operator while three equals (===) is a strict equality operator, i.e. it will not convert values automatically.

'' == '0' *// false*  
0 == '' *// true*  
0 == '0' *// true*  
  
false == 'false' *// false*  
false == '0' *// true*  
  
false == undefined *// false*  
false == null *// false*  
null == undefined *// true*

How would you check if a variable is null/undefined?

*//check if bar is null*  
bar === null

*//check if bar is undefined*  
typeof bar === "undefined"

##### How do you check if a variable is an object

You can use typeof to determine if variable is an object, however bear in mind that null is actually an object! However null object is 'falsy' thus the following will work:

if(bar && typeof bar === "object") {  
 console.log('bar is object and is not null');  
}

##### **Discuss scoping in JavaScript**.

JavaScript has lexical scoping based on functions but not blocks. Therefore:

*//global scope*  
(function() {  
 *//anonymous function scope*  
 var foo = 1;  
 function bar() {  
 *//bar function scope*  
 var foo = 2;  
 }  
 bar();  
 console.log(foo); *//outputs 1*  
 if(true) {  
 var foo = 3; *//redeclares foo*  
 }  
 console.log(foo); *//outputs 3*  
})();

Try it: <http://jsfiddle.net/tnajdek/8y3XC/>. Note: from within function scope everything in above scope(s) is available (see closures below)

##### Explain hoisting in JavaScript.

Function Expression

**var** foo = **function** **foo**(){   
 **return** 12;   
};

In JavaScript, variable and functions are hoisted. Let's take function hoisting first. Basically, the JavaScript interpreter looks ahead to find all variable declarations and then hoists them to the top of the function where they're declared. For example:

foo(); *// Here foo is still undefined*   
**var** foo = **function** **foo**(){   
 **return** 12;   
};

Behind the scene of the code above looks like this:

**var** foo = undefined;  
 foo(); *// Here foo is undefined*   
 foo = **function** **foo**(){  
 / Some code stuff  
 }

**var** foo = undefined;  
 foo = **function** **foo**(){  
 / Some code stuff  
 }  
 foo(); *// Now foo is defined here*

##### What are closures?

A closure is a function defined inside another function (called the parent function), and has access to variables that are declared and defined in the parent function scope.

The closure has access to variables in three scopes:

* Variables declared in their own scope
* Variables declared in a parent function scope
* Variables declared in the global namespace

**var** globalVar = "abc";   
  
*// Parent self invoking function*   
(**function** **outerFunction** (outerArg) { *// begin of scope outerFunction*  
 *// Variable declared in outerFunction function scope*   
 **var** outerFuncVar = 'x';   
 *// Closure self-invoking function*   
 (**function** **innerFunction** (innerArg) { *// begin of scope innerFunction*  
 *// variable declared in innerFunction function scope*  
 **var** innerFuncVar = "y";   
 console.log(   
 "outerArg = " + outerArg + "\n" +  
 "outerFuncVar = " + outerFuncVar + "\n" +  
 "innerArg = " + innerArg + "\n" +  
 "innerFuncVar = " + innerFuncVar + "\n" +  
 "globalVar = " + globalVar);  
   
 }*// end of scope innerFunction)(5); // Pass 5 as parameter*   
}*// end of scope outerFunction )(7); // Pass 7 as parameter*

innerFunction is closure that is defined inside outerFunction and has access to all variables declared and defined in the outerFunction scope. In addition, the function defined inside another function as a closure will have access to variables declared in the global namespace.

Thus, the output of the code above would be:

outerArg = 7  
outerFuncVar = x  
innerArg = 5  
innerFuncVar = y  
globalVar = abc

##### Explain prototypal/differential inheritance

Most of the Object Oriented languages support classes and objects. Here, Classes inherit from other classes.

In JavaScript, the inheritance is prototype-based. This means that there are no classes. Instead, there is an object that inherits from another object.

JavaScript provides three different types of Prototypal Inheritance.

**1. Delegation (I.E. The Prototype Chain).**

A delegate prototype is an object that serves as a base for another object. When you inherit from a delegate prototype, the new object gets a reference to the prototype.

When we try to access any property, it first checks in the properties owned by the object. If that property does not exist there, it checks in the **‘[[Prototype]]’** and so on. If that property does not exist there, it checks in the **‘[[Prototype]]’** and so on. Gradually, it moves up the prototype chain, until it reaches the **<Object.prototype>** i.e. the root delegate for most of the objects.

**2. Concatenative Inheritance (I.E. Mixins, Object.Assign()).**

It is the process of inheriting the features of one object to another by copying the source objects properties. JavaScript calls these source prototypes by the name mixins. This process makes use of the JavaScript method Object.assign(). However, before ES6, the <.extend()> method was used.

**3. Functional (Not To Be Confused With Functional Programming).**

In JavaScript, a function can create an object. It’s not necessary to be a constructor(or a class). It is called a factory function. Functional inheritance produces an object from a factory and also extends it, by assigning properties.

Every type of Prototypal Inheritance supports a separate set of use-cases, applicable to it. All of them are equally useful in their ability to enable composition. It provides has-a, uses-a, or can-do relationship as compared to the is-a relationship created with class inheritance.

##### What is Strict Mode in JavaScript

Strict Mode has been introduced as part of ECMAScript 5 and introduces new, restricted variant of JavaScript

You can declare strict mode by adding "use strict"; at the beginning of a file, a program, or a function. This kind of declaration is known as a *directive prologue*.

* Throws errors for actions that are rather silly but previously didn't throw an error
* Throws errors for potentially unsafe actions
* Disables functions that are poorly thought out
* Potentially code in strict mode could run faster by eliminating mistakes that would make it difficult for JavaScript engines to perform optimizations

Strict mode can be enabled for the entire source file or on per function basis by adding a string literal "use strict" on top of the file/function i.e.

function foo(){  
 "use strict";  
 *// ... your code ...*  
}**Restrictions on Code in Strict Mode**

The following table lists the most important restrictions that apply in strict mode.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| Language element | Restriction | Error | Example |
| Variable | Using a variable without declaring it. | SCRIPT5042: Variable undefined in strict mode | testvar = 4; |
| Read-only property | Writing to a read-only property. | SCRIPT5045: Assignment to read-only properties is not allowed in strict mode | var testObj = Object.defineProperties({}, { prop1: { value: 10, writable: false // by default }, prop2: { get: function () { } } }); testObj.prop1 = 20; testObj.prop2 = 30; |
| Non-extensible property | Adding a property to an object whose extensibleattribute is set to false. | SCRIPT5046: Cannot create property for a non-extensible object | var testObj = new Object(); Object.preventExtensions(testObj); testObj.name = "Bob"; |
| delete | Deleting a variable, a function, or an argument.  Deleting a property whose configurableattribute is set to false. | SCRIPT1045: Calling delete on <expression>is not allowed in strict mode | var testvar = 15; function testFunc() {}; delete testvar; delete testFunc; Object.defineProperty(testObj, "testvar", { value: 10, configurable: false }); delete testObj.testvar; |
| Duplicating a property | Defining a property more than once in an object literal. | SCRIPT1046: Multiple definitions of a property not allowed in strict mode | var testObj = { prop1: 10, prop2: 15, prop1: 20 }; |
| Duplicating a parameter name | Using a parameter name more than once in a function. | SCRIPT1038: Duplicate formal parameter names not allowed in strict mode | function testFunc(param1, param1) { return 1; }; |
| Future reserved keywords | Using a future reserved keyword as a variable or function name. | SCRIPT1050: The use of a future reserved word for an identifier is invalid. The identifier name is reserved in strict mode. | - implements  - interface  - package  - private  - protected  - public  - static  - yield |
| Octals | Assigning an octal value to a numeric literal, or attempting to use an escape on an octal value. | SCRIPT1039: Octal numeric literals and escape characters not allowed in strict mode | var testoctal = 010; var testescape = \010; |
| this | The value of this is not converted to the global object when it is null or undefined. |  | function testFunc() { return this; } var testvar = testFunc();  In non-strict mode, the value of testvaris the global object, but in strict mode the value is undefined. |
| eval as an identifier | The string "eval" cannot be used as an identifier (variable or function name, parameter name, and so on). |  | var eval = 10; |
| Function declared inside a statement or a block | You cannot declare a function inside a statement or a block. | SCRIPT1047: In strict mode, function declarations cannot be nested inside a statement or block. They may only appear at the top level or directly inside a function body. | var arr = [1, 2, 3, 4, 5]; var index = null; for (index in arr) { function myFunc() {}; } |
| Variable declared inside an evalfunction | If a variable is declared inside an evalfunction, it cannot be used outside that function. | SCRIPT1041: Invalid usage of 'eval' in strict mode | eval("var testvar = 10"); testvar = 15;  Indirect evaluation is possible, but you still cannot use a variable declared outside the eval function.  var indirectEval = eval; indirectEval("var testvar = 10;"); document.write(testVar);  This code causes an error SCRIPT5009: 'testVar' is undefined. |
| Arguments as an identifier | The string "arguments" cannot be used as an identifier (variable or function name, parameter name, and so on). | SCRIPT1042: Invalid usage of 'arguments' in strict mode | var arguments = 10; |
| arguments inside a function | You cannot change the values of members of the local argumentsobject. |  | function testArgs(oneArg) { arguments[0] = 20; }  In non-strict mode, you can change the value of the oneArg parameter by changing the value of arguments[0], so that the value of both oneArg and arguments[0] is 20. In strict mode, changing the value of arguments[0]does not affect the value of oneArg, because the arguments object is merely a local copy. |
| arguments.callee | Not allowed. |  | function (testInt) { if (testInt-- == 0) return; arguments.callee(testInt--); } |
| with | Not allowed. | SCRIPT1037: 'with' statements are not allowed in strict mode | with (Math){ x = cos(3); y = tan(7); } |

##### THIS

At the time of execution of every function, JavaScript engine sets a property to the function called this which refer to the current execution context. this is always refer to an object and depends on how function is called. There are 7 different cases where the value of this varies.

1. In the global context or inside a function this refers to the window object.
2. Inside IIFE (immediate invoking function) if you use "use strict", value of this is undefined. To pass access window inside IIFE with "use strict", you have to pass this.
3. While executing a function in the context of an object, the object becomes the value of this
4. Inside a setTimeout function, the value of this is the window object.
5. If you use a constructor (by using new keyword) to create an object, the value of this will refer to the newly created object.
6. You can set the value of this to any arbitrary object by passing the object as the first parameter of bind, call or apply
7. For dom event handler, value of this would be the element that fired the event

##### How to empty an array in JavaScript?

For instance,

**var** arrayList = ['a','b','c','d','e','f'];

How can we empty the array above?

There are a couple ways we can use to empty an array, so let's discuss them all.

**Method 1**

arrayList = []

Above code will set the variable arrayList to a new empty array. This is recommended if you don't have references to the original array arrayListanywhere else, because it will actually create a new, empty array. You should be careful with this method of emptying the array, because if you have referenced this array from another variable, then the original reference array will remain unchanged.

For Instance,

**var** arrayList = ['a','b','c','d','e','f']; *// Created array*   
**var** anotherArrayList = arrayList; *// Referenced arrayList by another variable*   
arrayList = []; *// Empty the array*   
console.log(anotherArrayList); *// Output ['a','b','c','d','e','f']*

**Method 2**

arrayList.length = 0;

The code above will clear the existing array by setting its length to 0. This way of emptying the array also updates all the reference variables that point to the original array. Therefore, this method is useful when you want to update all reference variables pointing to arrayList.

For Instance,

**var** arrayList = ['a','b','c','d','e','f']; *// Created array*   
**var** anotherArrayList = arrayList; *// Referenced arrayList by another variable*   
arrayList.length = 0; *// Empty the array by setting length to 0*  
console.log(anotherArrayList); *// Output []*

**Method 3**

arrayList.splice(0, arrayList.length);

The implementation above will also work perfectly. This way of emptying the array will also update all the references to the original array.

**var** arrayList = ['a','b','c','d','e','f']; *// Created array*   
**var** anotherArrayList = arrayList; *// Referenced arrayList by another variable*   
arrayList.splice(0, arrayList.length); *// Empty the array by setting length to 0*  
console.log(anotherArrayList); *// Output []*

**Method 4**

**while**(arrayList.length){  
 arrayList.pop();  
}

The implementation above can also empty arrays, but it is usually not recommended to use this method often.

##### Difference between undefined and not defined in JavaScript

In JavaScript if you try to use a variable that doesn't exist and has not been declared, then JavaScript will throw an error var name is not defined and the script will stop execute thereafter. But If you use typeof undeclared\_variable then it will return undefined.

Before starting further discussion let's understand the difference between declaration and definition.

var x is a declaration because you are not defining what value it holds yet, but you are declaring its existence and the need of memory allocation.

var x; // declaring x  
console.log(x); //output: undefined

var x = 1 is both declaration and definition (also we can say we are doing initialisation), Here declaration and assignment of value happen inline for variable x, In JavaScript every variable declaration and function declaration brings to the top of its current scope in which it's declared then assignment happen in order this term is called hoisting.

A variable that is declared but not define and when we try to access it, It will result undefined.

var x; // Declaration  
if(typeof x === 'undefined') // Will return true

A variable that neither declared nor defined when we try to reference such variable then It result not defined.

console.log(y); // Output: ReferenceError: y is not defined

##### Why Should You Not Prefer To Use Global Variables In JavaScript And How Can You Prevent It

The principal issue in using a global variable is that someone else can create another variable with the same name. And you may not know it until the duplicate could overwrite the value of your variable.

To avoid using globals, follow any of the following approaches.

**1.** Create a single global variable that holds all your other variables.

var myGlobalList = {};  
  
myGlobalList.first = "test";

**2.** Enclose all of your code in a self-executing method/function so that any variable declared inside remain in the function scope.

(function(){  
 var test = "myvar";  
})();

##### Use Cases for JavaScript's IIFEs

**Function Scoping vs. Block Scoping**

Local variables declared using the var keyword are scoped to the enclosing function. If no such function exists, the variables will be created as global variables instead, thus polluting the global scope. To prevent this, we can use an IIFE to create a function wrapper for local variables:

(*function()* {  
 *var* foo = "bar";  
 console.log(foo);  
})();  
  
foo; // ReferenceError: foo is not defined

The argument now is that instead of using an IIFE, we can use block-scoped variables to achieve the same result. Introduced by ECMAScript 2015, the let and const keywords declare local variables that are scoped to the enclosing *block* rather than the enclosing *function*:

{  
 let foo = "bar";  
 console.log(foo);  
}  
  
foo; // ReferenceError: foo is not defined

However, block-scoped variables are not a replacement for immediately invoked function expressions. Yes, let and const can be used to restrict the visibility of local variables to the surrounding block — if ECMAScript 2015 is supported, that is!

If, however, you're running your JavaScript code in an environment that doesn't support ECMAScript 2015 yet (such as older browsers, for example), you can't use the new letand const keywords for creating block-scoped local variables. You'll have to resort to classic function scoping in this case.

**Closures and Private Data**

Another use case for an IIFE is to provide a wrapping scope around a local variable that is accessed by a function returned from the IIFE. This way, [a *closure* is created](https://github.com/getify/You-Dont-Know-JS/blob/master/scope%20%26%20closures/ch5.md) that enables the function to access the local variable even when that function is executed *outside* of the IIFE's lexical scope.

Assume we want to create a function uniqueId that returns a unique identifier (like "id\_1", "id\_2", and so on) every time it's called. Within the IIFE, we'll be keeping track of a private counter variable that is incremented every time the counter function is called. We return from the IIFE another function that returns a new identifier string when called:

const uniqueId = (*function()* {  
 let count = 0;  
 return *function()* {  
 ++count;  
 return `id\_${count}`;  
 };  
})();  
  
console.log(uniqueId()); // "id\_1"  
console.log(uniqueId()); // "id\_2"  
console.log(uniqueId()); // "id\_3"

Note that the count variable is inaccessible from outside of the IIFE. Except for the function that's being returned, nobody can read or modify the count variable. This allows for the creation of truly private state that can only be modified in a controlled fashion. The [revealing module pattern](https://addyosmani.com/resources/essentialjsdesignpatterns/book/#revealingmodulepatternjavascript) relies heavily on this mechanism:

const counter = (*function()* {  
 let counterValue = 0;   
  
 return {  
 increment() {  
 ++counterValue;   
 },   
  
 get value() {  
 return counterValue;  
 }  
 };  
})();  
  
counter.increment();  
console.log(counter.value); // 1  
  
counter.increment();  
counter.increment();  
console.log(counter.value); // 3

Neither let nor const is a replacement for an IIFE returning a function that closes over some local variables to manage private data.

**Aliasing Variables**

Sometimes, you might be in the situation that you're using two different libraries that expose a global variable with the same name. For instance, consider that you're using jQuery and another library that also assigns to the $ global variable.

To resolve this naming conflict, you can wrap a piece of code with an IIFE that passes one of the global variables (e.g. jQuery) as an argument. Within the function, you can then refer to the value by a parameter name (e.g. $) of your choosing:

*window*.$ = *function* somethingElse*()* {  
 // ...  
};  
  
(*function($)* {  
 // ...  
})(jQuery);

Within the IIFE, the $ parameter refers to the jQuery function and shadows whatever value has been assigned to $ in the outer scope.

**Capturing the Global Object**

Depending on where your JavaScript code runs, you'll have a different global object. When running in the browser, the global object is window. Node.js, on the other hand, uses the global object. Since you don't want to hardcode either one of those names when writing universal JavaScript code, you can use a wrapper like this:

(*function(global)* {  
 // ...  
})(this);

The global parameter will refer to the correct global object in both a browser and a Node.js environment. Check out [this post by Todd Motto](https://toddmotto.com/what-function-window-document-undefined-iife-really-means/) for more details on capturing the global object using this technique.

**Optimization for Minification**

The approach of aliasing variable names can also be used to optimize code such that it can be minified more efficiently. Take this common wrapper, for example:

(*function(window, document, undefined)* {  
 // ...  
})(*window*, *document*);

A JavaScript minifier like [UglifyJS](https://github.com/mishoo/UglifyJS2) can now shorten the function's parameter names to single-letter identifiers:

(*function(w, d, u)* {  
 // ...  
})(*window*, *document*);

The idea is that shorter identifier names result in a smaller file size. However, if HTTP responses are compressed using Gzip or Deflate, the file size is reduced very effectively anyway. Hence, the marginal gains of this minification technique are lower if used in conjunction with compression algorithms. The shorter names might still pay off, though, so measure and compare the response sizes for yourself.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| voteaccepted | anonymous functions uses for three reasons: Advantages /Disadvantages **PROS**   1. If no name is needed because the function is only ever called in one place, then why add a name to whatever namespace you're in. 2. Anonymous functions are declared inline and inline functions have advantages in that they can access variables in the parent scopes. Yes, you can put a name on an anonymous function, but that's usually pointless if it's declared inline. So inline has a significant advantage and if you're doing inline, there's little reason to put a name on it. 3. The code seems more self-contained and readable when handlers are defined right inside the code that's calling them. You can read the code in almost sequential fashion rather than having to go find the function with that name.   I do try to avoid deep nesting of anonymous functions because that can be hairy to understand and read. Usually when that happens, there's a better way to structure the code (sometimes with a loop, sometimes with a data table, etc...) and named functions isn't usually the solution there either.  I guess I'd add that if a callback starts to get more than about 15-20 lines long and it doesn't need direct access to variables in the parent scope, I would be tempted to give it a name and break it out into it's own named function declared elsewhere. There is definitely a readability point here where a non-trivial function that gets long is just more maintainable if it's put in its own named unit. But, most callbacks I end up with are not that long and I find it more readable to keep them inline.  **CONS**   * Anon functions can't take advantage of function hoisting. This is a major difference. I tend to take heavy advantage of hoisting to define my own explicitly named funcs and object constructors towards the bottom and get to the object definition and main-loop type stuff right up at the top. I find it makes the code easier to read when you name your vars well and get a broad view of what's going on before ctrl-Fing for details only when they matter to you. Hoisting can also be a huge benefit in heavily event-driven interfaces where imposing a strict order of what's available when can bite you in the butt. Hoisting has its own caveats (like circular reference potential) but it is a very useful tool for organizing and making code legible when used right. * Legibility/Debug. Absolutely they get used way too heavily at times and it can make debug and code legibility a hassle. Codebases that rely heavily on JQ, for instance, can be a serious PITA to read and debug if you don't encapsulate the near-inevitable anon-heavy and massively overloaded args of the $ soup in a sensible way. JQuery's hover method for instance, is a classic example of over-use of anon funcs when you drop two anon funcs into it, since it's easy for a first-timer to assume it's a standard event listener assignment method rather than one method overloaded to assign handlers for one or two events. $(this).hover(onMouseOver, onMouseOut) is a lot more clear than two anon funcs.  What Are Event Handlers In JavaScript And How To Use Them? Answer.  JavaScript event handlers are functions that bind to a specific HTML DOM event. And events are the part of HTML document object model (DOM). An event can take place in one of the following cases.  Due to user actions on a web page.  <onclick()>, <onmouseover()>, <onkeydown()>  Some events are callbacks triggered by the browser when a page changes its state.  <onload()>, <onunload()>, <onresize()>  Whenever the DOM receives an event, it calls the JavaScript event handler function bind to the event. To use a handler function, we can assign it to the desired event as an attribute of the target HTML element. Please follow the below example.  <HTML>  <BODY>  <FORM>  <INPUT TYPE=”button” VALUE=”Test” onClick=”window.alert ('Event Handler')”>  </FORM>  </BODY>  </HTML> **Difference between Anonymous vs. referenced v**s. declared functions **Anonymous**  Anonymous functions are typically used as callbacks.  *So what’s a callback?*  I’m glad you asked!   |  |  | | --- | --- | | 1 2 3 4 5 6 7 8 | function takesACallback(callback) {  // do some interesting things here  return "The callback says: " + callback(); }  takesACallback(function() {  return "I'm the callback!"; }); // returns "The callback says: I'm the callback!" |     See the function that’s a parameter to the takesACallback call? Notice that it doesn’t have a name? That’s an anonymous function.  **Referenced**  However, you can still have a reference to an anonymous function, just because it is unnamed it doesn’t mean it’s unreferenced.   |  |  | | --- | --- | | 1 2 3 4 5 | var reference = function() {  return "I'm still an anonymous function"; }  reference(); // returns "I'm still an anonymous function" |     The function itself is still anonymous (it doesn’t have a name directly attached to it), but you can call the function by the named reference.  **Declared**  Declared functions are not anonymous. They have a name directly attached to the function, with no need for a named reference.   |  |  | | --- | --- | | 1 2 3 4 5 | function declared() {  return "I'm not anonymous function"; }  declared(); // returns "I'm not anonymous function" |     Writing a named function like this results in a function declaration.  According to [Helen](http://helephant.com/2008/08/23/javascript-anonymous-functions/):  Anonymous functions are created at runtime ... Function declarations are different. They are run before any of the other code is executed so the functions do not have to be declared before the code that calls them.  Here’s an example of the referenced anonymous function and a declared function in action:   |  |  | | --- | --- | | 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | function tester() {  declaredFunction();  referencedFunction();   function declaredFunction() {  console.log("I'm a declared function");  }   var referencedFunction = function () {  console.log("I'm an anonymous function called by a reference");  }; }  tester(); // logs "I'm a declared function" // Uncaught TypeError: referencedFunction is not a function |     *Huh? Why did only one of those work?*  This is directly related to [hoisting](http://lucybain.com/blog/2014/js-anonymous-referenced-declared-functions/#) - if you’re not familiar with that term you should read up about it. Go ahead, I’ll wait. All caught up? Good!  Hopefully after reading about hoisting you’ve got an idea of what’s going on. Let’s look in a bit more detail. Although you wrote the code above the JS interpreter converts it to:   |  |  | | --- | --- | | 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | function tester() {  function declaredFunction() {  console.log("I'm a declared function");  }   var referencedFunction;   declaredFunction();  referencedFunction();   referencedFunction = function () {  console.log("I'm an anonymous function called by a reference");  }; } |     Now Helen’s quote should make more sense. The anonymous function (which is referenced by referencedFunction) isn’t created until it’s position in the code (at runtime). However the declared function is hoisted to the top of the scope and is immediately available for use. What is a Promise? A Promise in short:  "Imagine you are a kid. Your mom promises you that she'll get you a new phone next week."  You *don't know* if you will get that phone until next week. Your mom can either *really buy*you a brand new phone, or *stand you up* and withhold the phone if she is not happy :(.  That is a promise. A promise has 3 states. They are:   1. Promise is pending: You don't know if you will get that phone until next week. 2. Promise is resolved: Your mom really buy you a brand new phone. 3. Promise is rejected: You don't get a new phone because your mom is not happy.     **Chaining Promises**  Promises are chainable.  Let's say, you, the kid, promise your friend that you will show them the new phone when your mom buy you one.  That is another promise. Let's write it!   [Why Promises and When to Use Them?](https://scotch.io/tutorials/javascript-promises-for-dummies#toc-why-promises-and-when-to-use-them) Why do we need promises? How's the world look like before promise? Before answering these questions, let's go back to the fundamental.  **Normal Function vs Async Function**  Let's take a look at these two example, both example perform addition of two number, one add using normal function, the other add remotely.  **Normal Function to Add Two Numbers**  // add two numbers normally  function add (num1, num2) {  return num1 + num2; }  const result = add(1, 2); // you get result = 3 immediately  **Async Function to Add Two numbers**  // add two numbers remotely  // get the result by calling an API const result = getAddResultFromServer('http://www.example.com?num1=1&num2=2'); // you get result = "undefined"  If you add the numbers with normal function, you get the result immediately. However when you issue a remote call to get result, you need to wait, you can't get the result immediately.  Or put it this way, you don't know if you will get the result because the server might be down, slow in response, etc. You don't want your entire process to be blocked while waiting for the result.  Calling APIs, downloading files, reading files are among some of the usual async operations that you'll perform.  **World Before Promises: Callback**  Must we use promise for asynchronous call? Nope. Prior to Promise, we use callback. Callback is just a function you call when you get the return result. Let's modify the previous example to accept a callback.  // add two numbers remotely // get the result by calling an API  function addAsync (num1, num2, callback) {  // use the famous jQuery getJSON callback API  return $.getJSON('http://www.example.com', {  num1: num1,  num2: num2  }, callback); }  addAsync(1, 2, success => {  // callback  const result = success; // you get result = 3 here });  The syntax looks ok, why do we need promises then?  **What if You Want to Perform Subsequent Async Action?**  Let's say, instead of just add the numbers one time, we want to add 3 times. In a normal function, we do this:-  // add two numbers normally  let resultA, resultB, resultC;   function add (num1, num2) {  return num1 + num2; }  resultA = add(1, 2); // you get resultA = 3 immediately resultB = add(resultA, 3); // you get resultB = 6 immediately resultC = add(resultB, 4); // you get resultC = 10 immediately  console.log('total' + resultC); console.log(resultA, resultB, resultC);  How it looks like with callbacks?  // add two numbers remotely // get the result by calling an API  let resultA, resultB, resultC;  function addAsync (num1, num2, callback) {  // use the famous jQuery getJSON callback API  return $.getJSON('http://www.example.com', {  num1: num1,  num2: num2  }, callback); }  addAsync(1, 2, success => {  // callback 1  resultA = success; // you get result = 3 here   addAsync(resultA, 3, success => {  // callback 2  resultB = success; // you get result = 6 here   addAsync(resultB, 4, success => {  // callback 3  resultC = success; // you get result = 10 here   console.log('total' + resultC);  console.log(resultA, resultB, resultC);  });  }); });  The syntax is less user friendly. In a nicer term, It looks like a pyramid, but people usually refer this as "callback hell", because the callback nested into another callback. Imagine you have 10 callbacks, your code will nested 10 times!  **Escape From Callback Hell**  Promises come in to rescue. Let's look at the promise version of the same example.  // add two numbers remotely using observable  let resultA, resultB, resultC;  function addAsync(num1, num2) {  // use ES6 fetch API, which return a promise  return fetch(`http://www.example.com?num1=${num1}&num2=${num2}`)  .then(x => x.json()); }  addAsync(1, 2)  .then(success => {  resultA = success;  return resultA;  })  .then(success => addAsync(success, 3))  .then(success => {  resultB = success;  return resultB;  })  .then(success => addAsync(success, 4))  .then(success => {  resultC = success;  return resultC;  })  .then(success => {  console.log('total: ' + success)  console.log(resultA, resultB, resultC)  });  With promises, we flatten the callback with .then. In a way, it looks cleaner because of no callback nesting. Of course, with ES7 async syntax, we can even further enhance this example, but I leave that to you. :)  [#](https://scotch.io/tutorials/javascript-promises-for-dummies#toc-new-kid-on-the-block-observables)**[New Kid On the Block: Observables](https://scotch.io/tutorials/javascript-promises-for-dummies#toc-new-kid-on-the-block-observables)**  Before you settle down with promises, there is something that has come about to make it even easier to deal with async data called Observables.  Observables are lazy event streams which can emit zero or more events, and may or may not finish.   * [source](https://cycle.js.org/streams.html)   Some key differences between promises and observable are:   * Observables are cancellable * Observable are lazy   Fear not, let look at the same demo written with Observables. In this example, I am using [RxJS](http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html) for the observables.  let Observable = Rx.Observable; let resultA, resultB, resultC;  function addAsync(num1, num2) {  // use ES6 fetch API, which return a promise  const promise = fetch(`http://www.example.com?num1=${num1}&num2=${num2}`)  .then(x => x.json());   return Observable.fromPromise(promise); }  addAsync(1,2)  .do(x => resultA = x)  .flatMap(x => addAsync(x, 3))  .do(x => resultB = x)  .flatMap(x => addAsync(x, 4))  .do(x => resultC = x)  .subscribe(x => {  console.log('total: ' + x)  console.log(resultA, resultB, resultC)  });  Demo: <https://jsbin.com/dosaviwalu/edit?js,console>  Notes:   * Observable.fromPromise converts a promise to observable stream. * .do and .flatMap are among some of the operators available for Observables * Streams are lazy. Our addAsync runs when we .subscribe to it.   Observables can do more funky stuff easily. For example, delay add function by 3 secondswith just one line of code or retry so you can retry a call a certain number of times.  ...  addAsync(1,2)  .delay(3000) // delay 3 seconds  .do(x => resultA = x) Const vs let vs var In JavaScript, `**const**` means that the identifier can’t be reassigned. (Not to be confused with immutable values. Unlike true immutable datatypes such as those produced by Immutable.js and Mori, a `const` object can have properties mutated.)  If I don’t need to reassign, `const` is my default choice over `let` because I want the usage to be as clear as possible in the code.  I use **`let`** when I need to reassign a variable. Because I use one variable to represent one thing, the use case for `let` tends to be for loops or mathematical algorithms.  I don’t use `**var`** in ES6. There is value in block scope for loops, but I can’t think of a situation where I’d prefer `var` over `let`. `const` is a signal that the identifier won’t be reassigned. `let`, is a signal that the variable may be reassigned, such as a counter in a loop, or a value swap in an algorithm. It also signals that the variable will be used only in the block it’s defined in, which is not always the entire containing function. `var` is now the weakest signal available when you define a variable in JavaScript. The variable may or may not be reassigned, and the variable may or may not be used for an entire function, or just for the purpose of a block or loop. Warning: With `let` and `const` in ES6, it’s no longer safe to check for an identifier’s existence using `typeof`:  function foo () {  typeof bar;  let bar = ‘baz’; } foo(); // ReferenceError: can't access lexical declaration  // `bar' before initialization Callback hell See the pyramid shape and all the }) at the end? Eek! This is affectionately known as **callback hell**.  The cause of callback hell is when people try to write JavaScript in a way where execution happens visually from top to bottom  fs.readdir(source, **function** (err, files) {  **if** (err) {  console.log('Error finding files: ' **+** err)  } **else** {  files.**forEach**(**function** (filename, fileIndex) {  console.log(filename)  gm(source **+** filename).size(**function** (err, values) {  **if** (err) {  console.log('Error identifying file size: ' **+** err)  } **else** {  console.log(filename **+** ' : ' **+** values)  aspect **=** (values.width / values.height)  widths.**forEach**(**function** (width, widthIndex) {  height **=** Math.round(width / aspect)  console.log('resizing ' **+** filename **+** 'to ' **+** height **+** 'x' **+** height)  **this**.resize(width, height).write(dest **+** 'w' **+** width **+** '\_' **+** filename, **function**(err) {  **if** (err) console.log('Error writing file: ' **+** err)  })  }.bind(**this**))  }  })  })  } })  **How do I fix callback hell?**  Callback hell is caused by poor coding practices. Luckily writing better code isn't that hard!  You only need to follow **three rules**:  **1. Keep your code shallow**  Here is some messy browser JavaScript that uses [browser-request](https://github.com/iriscouch/browser-request) to make an AJAX request to a server:  **var** form **=** document.querySelector('form') form.onsubmit **=** **function** (submitEvent) {  **var** name **=** document.querySelector('input').value  request({  uri: "http://example.com/upload",  body: name,  method: "POST"  }, **function** (err, response, body) {  **var** statusMessage **=** document.querySelector('.status')  **if** (err) **return** statusMessage.value **=** err  statusMessage.value **=** body  }) }  This code has two anonymous functions. Let's give em names!  **var** form **=** document.querySelector('form') form.onsubmit **=** **function** **formSubmit** (submitEvent) {  **var** name **=** document.querySelector('input').value  request({  uri: "http://example.com/upload",  body: name,  method: "POST"  }, **function** **postResponse** (err, response, body) {  **var** statusMessage **=** document.querySelector('.status')  **if** (err) **return** statusMessage.value **=** err  statusMessage.value **=** body  }) }  As you can see naming functions is super easy and has some immediate benefits:   * makes code easier to read thanks to the descriptive function names * when exceptions happen you will get stacktraces that reference actual function names instead of "anonymous" * allows you to move the functions and reference them by their names   Now we can move the functions to the top level of our program:  document.querySelector('form').onsubmit **=** formSubmit  **function** **formSubmit** (submitEvent) {  **var** name **=** document.querySelector('input').value  request({  uri: "http://example.com/upload",  body: name,  method: "POST"  }, postResponse) }  **function** **postResponse** (err, response, body) {  **var** statusMessage **=** document.querySelector('.status')  **if** (err) **return** statusMessage.value **=** err  statusMessage.value **=** body }  Note that the function declarations here are defined at the bottom of the file. This is thanks to [function hoisting](https://gist.github.com/maxogden/4bed247d9852de93c94c).  **2. Modularize**  This is the most important part: **Anyone is capable of creating modules** (aka libraries). To quote [Isaac Schlueter](http://twitter.com/izs) (of the node.js project): *"Write small modules that each do one thing, and assemble them into other modules that do a bigger thing. You can't get into callback hell if you don't go there."*  Let's take out the boilerplate code from above and turn it into a module by splitting it up into a couple of files. I'll show a module pattern that works for either browser code or server code (or code that works in both):  Here is a new file called formuploader.js that contains our two functions from before:  module.exports.submit **=** formSubmit  **function** **formSubmit** (submitEvent) {  **var** name **=** document.querySelector('input').value  request({  uri: "http://example.com/upload",  body: name,  method: "POST"  }, postResponse) }  **function** **postResponse** (err, response, body) {  **var** statusMessage **=** document.querySelector('.status')  **if** (err) **return** statusMessage.value **=** err  statusMessage.value **=** body }  The module.exports bit is an example of the node.js module system which works in node, Electron and the browser using [browserify](https://github.com/substack/node-browserify). I quite like this style of modules because it works everywhere, is very simple to understand and doesn't require complex configuration files or scripts.  Now that we have formuploader.js (and it is loaded in the page as a script tag after being browserified) we just need to require it and use it! Here is how our application specific code looks now:  **var** formUploader **=** require('formuploader') document.querySelector('form').onsubmit **=** formUploader.submit  Now our application is only two lines of code and has the following benefits:   * easier for new developers to understand -- they won't get bogged down by having to read through all of the formuploader functions * formuploader can get used in other places without duplicating code and can easily be shared on github or npm   **3. Handle every single error**  There are different types of errors: syntax errors caused by the programmer (usually caught when you try to first run the program), runtime errors caused by the programmer (the code ran but had a bug that caused something to mess up), platform errors caused by things like invalid file permissions, hard drive failure, no network connection etc. This section is only meant to address this last class of errors.  The first two rules are primarily about making your code readable, but this one is about making your code stable. When dealing with callbacks you are by definition dealing with tasks that get dispatched, go off and do something in the background, and then complete successfully or abort due to failure. Any experienced developer will tell you that you can never know when these errors happen, so you have to plan on them always happening.  With callbacks the most popular way to handle errors is the Node.js style where the first argument to the callback is always reserved for an error.  **var** fs **=** require('fs')   fs.readFile('/Does/not/exist', handleFile)   **function** **handleFile** (error, file) {  **if** (error) **return** console.error('Uhoh, there was an error', error)  // otherwise, continue on and use `file` in your code  }  Having the first argument be the error is a simple convention that encourages you to remember to handle your errors. If it was the second argument you could write code like function handleFile (file) { } and more easily ignore the error.  Code linters can also be configured to help you remember to handle callback errors. The simplest one to use is called [standard](http://standardjs.com/). All you have to do is run $ standard in your code folder and it will show you every callback in your code with an unhandled error.  **Summary**   1. Don't nest functions. Give them names and place them at the top level of your program 2. Use [function hoisting](https://gist.github.com/maxogden/4bed247d9852de93c94c) to your advantage to move functions 'below the fold' 3. Handle **every single error** in every one of your callbacks. Use a linter like [standard](http://standardjs.com/) to help you with this. 4. Create reusable functions and place them in a module to reduce the cognitive load required to understand your code. Splitting your code into small pieces like this also helps you handle errors, write tests, forces you to create a stable and documented public API for your code, and helps with refactoring.   The most important aspect of avoiding callback hell is **moving functions out of the way** so that the programs flow can be more easily understood without newcomers having to wade through all the detail of the functions to get to the meat of what the program is trying to do.  You can start by moving the functions to the bottom of the file, then graduate to moving them into another file that you load in using a relative require like require('./photo-helpers.js') and then finally move them into a standalone module like require('image-resize').  Here are some rules of thumb when creating a module:   * Start by moving repeatedly used code into a function * When your function (or a group of functions related to the same theme) get big enough, move them into another file and expose them using module.exports. You can load this using a relative require * If you have some code that can be used across multiple projects give it it's own readme, tests and package.json and publish it to github and npm. There are too many awesome benefits to this specific approach to list here! * A good module is small and focuses on one problem * Individual files in a module should not be longer than around 150 lines of JavaScript * A module shouldn't have more than one level of nested folders full of JavaScript files. If it does, it is probably doing too many things * Ask more experienced coders you know to show you examples of good modules until you have a good idea of what they look like. If it takes more than a few minutes to understand what is happening, it probably isn't a very good module.   **What about promises/generators/ES6 etc?**  Before looking at more advanced solutions, remember that callbacks are a fundamental part of JavaScript (since they are just functions) and you should learn how to read and write them before moving on to more advanced language features, since they all depend on an understanding of callbacks. If you can't yet write maintainable callback code, keep working at it!  If you *really* want your async code to read top-to-bottom, there are some fancy things you can try. Note that **these may introduce performance and/or cross platform runtime compatibility issues**, so make sure to do your research.  **Promises** are a way to write async code that still appears as though it is executing in a top-down way, and handles more types of errors due to encouraged use of try/catch style error handling.  **Generators** let you 'pause' individual functions without pausing the state of the whole program, which at the cost of slightly more complex to understand code lets your async code appear to execute in a top-down fashion. Check out [watt](https://github.com/mappum/watt) for an example of this approach.  **Async functions** are a proposed ES7 feature that will further wrap generators and promises in a higher level syntax. Check them out if that sounds interesting to you.  Personally I use callbacks for 90% of the async code I write and when things get complicated I bring in something like [run-parallel](https://github.com/feross/run-parallel) or [run-series](https://github.com/feross/run-series). I don't think callbacks vs promises vs whatever else really make a difference for me, the biggest impact comes from keeping code simple, not nested and split up into small modules.  Regardless of the method you choose, always **handle every error** and **keep your code simple**.  … |