**Difference between var and let keyword**

|  |  |
| --- | --- |
| **var** | **let** |
| “var” is there in js from the beginning | “let” was introduced in ES2015 / ES6 |
| “var” has functional scope | “let” variable has block scope |
| “var” gets hoisted  Variable declared with var gets hoisted on top of its functions | “let” not gets hoisted |

**Difference between “==” and “===”**

|  |  |
| --- | --- |
| == | === |
| Comparison operator | Comparison operator |
| Compare values only | Compare value and type |
|  |  |
|  |  |

**Difference between let and const keyword**

|  |  |
| --- | --- |
| let | const |
| With let variable we can change value | Define Constant value i.e. can be change  With array /object const allow to modify value but not allow to reassign value |
|  |  |
|  |  |
|  |  |

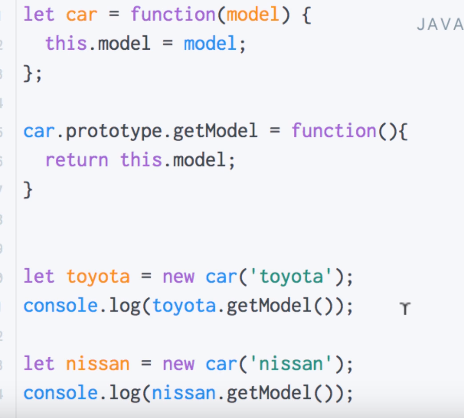
**Difference between undefined and null keyword**

|  |  |
| --- | --- |
| undefined | null |
| Represent empty value | Represent empty value |
| typeof(undefined) => undefined | typeof(null) => object |
|  |  |
|  |  |

**Use of Arrow Functions?**

**What is prototypal inheritance?**

Every object has property called prototype. We can add methods and property to it and when you create other object from this object, the newly created object automatically inherited the property of parents.



**Difference between function declaration and function expression?**

**function expression - Anonymous function saved into the variable.**

**//**Execute function before its declaration

Console.log (funcE ()); //error

let funcE = function () {

console.log(‘function expression’);

}

**function declaration**

//Execute function before its declaration

Console.log (funcD ());

// available before declaration

function funcD () {

console.log(‘function declartion’);

}

**What is promises and why do we use it?**

**setTimeout () ?**

setTimeOut (function () {

console.log(‘a’);

},0);

Console.log(‘b’);

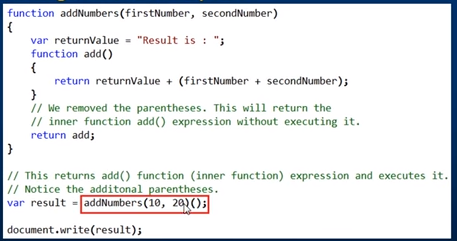
Console.log (‘c’);

**What is a closure and how do you use it?**

Closures are frequently used in JavaScript for object data privacy, in event handlers and callback functions, and in partial applications, currying, and other functional programming patterns.

Objects are not the only way to produce data privacy. Closures can also be used to create stateful functions whose return values may be influenced by their internal state.

A closure is an inner function that has access to the outer functions variable in addition to its own variable and global variable. In simple terms a closure is function inside function. These function, that is the inner and outer functions could be named functions or anonymous functions.



**4 JavaScript Design Patterns You Should Know**

The design patterns:

* Module design patterns
* Prototype design patterns
* Observer design patterns
* Singleton design patterns

**Module design patterns**

JavaScript modules are the most prevalently used design patterns for keeping particular pieces of code independent of other components. This provides loose coupling to support well-structured code.

Modules should be Immediately-Invoked-Function-Expressions (IIFE) to allow for private scopes - that is, a closure that protect variables and methods (however, it will return an object instead of a function). This is what it looks like:

(function () {

// declare private variables and/or functions

return {

// declare public variables and/or functions

}

} )();

var HTMLChanger = (function () {

var contents = 'contents'

var changeHTML = function() {

var element = document.getElementById('attribute-to-change');

element.innerHTML = contents;

}

return {

callChangeHTML: function () {

changeHTML();

console.log(contents);

}

};

})();

HTMLChanger.callChangeHTML(); // Outputs: 'contents'

console.log(HTMLChanger.contents); // undefined

Notice that callChangeHTML binds to the returned object and can be referenced within the HTMLChanger namespace. However, when outside the module, contents are unable to be referenced.

**Revealing Module Pattern**

A variation of the module pattern is called the Revealing Module Pattern. The purpose is to maintain encapsulation and reveal certain variables and methods returned in an object literal. The direct implementation looks like this:

var Exposer = (function() {

var privateVariable = 10;

var privateMethod = function() {

console.log('Inside a private method!');

privateVariable++;

}

var methodToExpose = function() {

console.log('This is a method I want to expose!');

}

var otherMethodIWantToExpose = function() {

privateMethod();

}

return {

first: methodToExpose,

second: otherMethodIWantToExpose

};

})();

Exposer.first(); // Output: This is a method I want to expose!

Exposer.second(); // Output: Inside a private method!

Exposer.methodToExpose; // undefined

**Prototype design patterns**

Any JavaScript developer has either seen the keyword prototype, confused by the prototypical inheritance, or implemented prototypes in their code. The Prototype design pattern relies on the JavaScript prototypical inheritance. The prototype model is used mainly for creating objects in performance-intensive situations.

var TeslaModelS = function() {

this.numWheels = 4;

this.manufacturer = 'Tesla';

this.make = 'Model S';

}

TeslaModelS.prototype = {

go: function() {

// Rotate wheels

},

stop: function() {

// Apply brake pads

}

}

var TeslaModelS = function() {

this.numWheels = 4;

this.manufacturer = 'Tesla';

this.make = 'Model S';

}

TeslaModelS.prototype.go = function() {

// Rotate wheels

}

TeslaModelS.prototype.stop = function() {

// Apply brake pads

}

**Revealing Prototype Pattern**

Like Module pattern, the Prototype pattern also has a revealing variation. The Revealing Prototype Pattern provides encapsulation with public and private members since it returns an object literal.

var TeslaModelS = function() {

this.numWheels = 4;

this.manufacturer = 'Tesla';

this.make = 'Model S';

}

TeslaModelS.prototype = function() {

var go = function() {

// Rotate wheels

};

var stop = function() {

// Apply brake pads

};

return {

pressBrakePedal: stop,

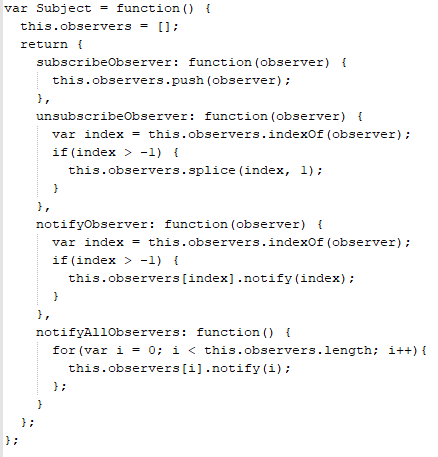
pressGasPedal: go

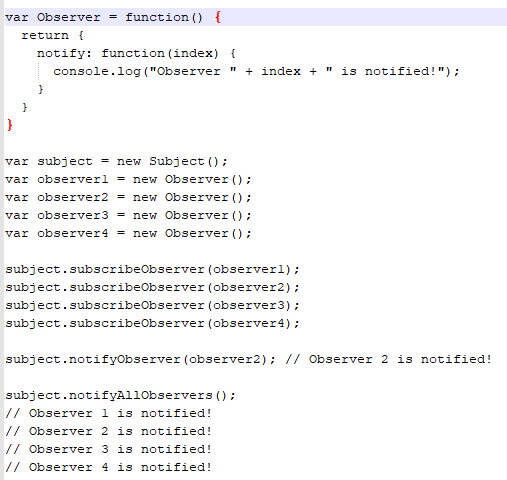
}

}();

**Observer design patterns**

We can create our own Subjects and Observers in JavaScript. Let's see how this is implemented:





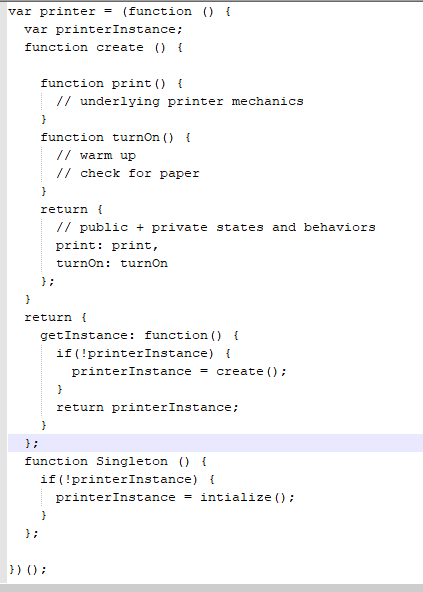
**Publish/Subscribe**

The Publish/Subscribe pattern, however, uses a topic/event channel that sits between the objects wishing to receive notifications (subscribers) and the object firing the event (the publisher). This event system allows code to define application-specific events that can pass custom arguments containing values needed by the subscriber. The idea here is to avoid dependencies between the subscriber and publisher.

Many developers choose to aggregate the publish/subscribe design pattern with the observer though there is a distinction. Subscribers in the publish/subscribe pattern are notified through some messaging medium, but observers are notified by implementing a handler like the subject.

**Singleton design patterns**

A Singleton only allows for a single instantiation, but many instances of the same object. The Singleton restricts clients from creating multiple objects, after the first object created, it will return instances of itself.



var officePrinter = printer.getInstance();

The create method is private because we do not want the client to access this, however, notice that the getInstance method is public. Each officer worker can generate a printer instance by interacting with the getInstance method, like so: