Week1

- *JAMStack* is a technology stack comprising of JavaScript, API and Markup. The term is coined by *Mathias Billmann*.
- JavaScript can be used on the client/browser or server (node) side. It helps reduce the load on the server.
- Most JavaScript engines at the browser use UTF-16 for encoding.
- Comparison using "===" operator checks the equality of value and type of data. This helps avoid implicit "coercion".
- *var* has a function-level scope, while *let* and *const* has a block-level scope. It implies variables declared using *let* and *const* don't exist outside the block. However *var* can change a global variable. To create a global variable, don't use any keyword.

```
function quiz() {
    a = 10  // Global
    var b = 20 //Function scope.
    console.log(b)
    let c = 20 //Block scope
    console.log(c)
}
quiz()
console.log(a) //prints 10
console.log(b) //ReferenceError
console.log(c)  //ReferenceError
```

- While defining *const* variables, initialization is mandatory. Else, *syntax error* ensues.
- AJAX stands for *Asynchronous JavaScript and XML*

Week2

- To find length of an array x, use x.length
- To delete elements of an array x, use x = []
- To add element into an array x, use x.push()
- *splice* is used to remove elements from an array. It returns the removed elements.
- Assigning x.length creates holes in the array.
- In a C-style for loop, one cannot use a *const* to declare the variable, since *const* cannot be incremented. Use *let* or *var*.
- C-style for loop loops over all indices of the array. for…in also loops over array indices. for…of loops over values instead.
- C-style for loop loops over holes; Similarly, for…of also loops over holes. for…in does not loop over holes.
- Object.keys(x) returns a list with the indices of the array x. Note that all indices are of type string, not number. This is unlike Python.
- Object.entries(x) returns a list containing lists, each containing a key (as a string) and its corresponding value.
- Values in an object can be accessed as x['first'] or x.first, assuming *first* is a key in x. console.log(x) will print all its entries.

- C-style for loop loops over all indices of the object. for…of cannot be used with objects directly (because objects are not iterable), unless used with its entries like Object.entries(x).
- new Array() defines an array, all of whose elements are holes initially. Array() can also be used instead.
- Use …x to spread out the array x inside another data structure.
- y = x.find(t => t < 0) finds the first value in array x that's less than 0.
- y = x.filter(t => t < 0) finds all values in array x that's less than 0, and returns them in an array format.
- x.map(t => t > 0 ? '+' : '-') returns an array containing '+' for all positive values in array x, and '-' for all negative values.
- x.reduce((a, i) => a + i, 0) returns the cumulative sum of all elements in *x*. *a* gets initial value of 0, and is accumulated as it marches through each element in *x*(represented as *i*).
- x.sort() sorts all elements in the array, but does a lexical sort. If you want to sort it numerically, use x.sort((a, b) => a - b)
- Array de-structuring

```
let x = [1, 2, 3];
let [a, b] = x;
```

a gets 1 and *b* gets 2, after the above code is run. Value 3 is ignored.
NOTE: If the second line was a, b = x;, it would give *ReferenceError*.

- Object de-structuring

```
const person = {
  firstName: 'Albert',
  lastName: 'Pinto',
  age: 25,
  city: 'Mumbai'
};
const {firstName: fn, city: c} = person;
```

fn is assigned 'Albert' and *c* is assigned 'Mumbai', after executing the above code.

- Object.keys(person).length returns the number of keys in the object *person*. Note that person.length will return *undefined*.
- Another example of destructuring

```
<script>
    fruit = {
        name: 'Apple',
        color: 'red',
    }
    let description = ({name, color, shape = 'Spherical'}) => {
        console.log(`${name} is ${color} and ${shape}`)
    }
    description (fruit) //Apple is red and Spherical
</script>
```

- type=" module" is mandatory when using the script inside an html file, if there're import/export statements in the script.

- import {c, energy} from './module1.js'; imports variables *c* and *energy* exported from module1.js. In order to export, use export c; in module1.js. Note that it's not necessary to export *internal* functions of module1.js.
- variables/functions can be renamed during *export* or *import* process.
- When importing, only a read-only view of the export variable is available. Hence, it's not possible to change the value of *c*, after importing.
- Implementation of class in JS involves setting *prototype* of one object to another (which will then act as its parent)
- Multiple inheritance is non-existent in JS. In the case of inheritance, the constructor must explicitly call super().
- *this* refers to the current object.

```
<script>
    obj = {
        first_name: "Anand",
        last_name: "Iyer",
    }

    obj.full_name = function () {
        return (this.first_name + " "+ this.last_name)
    }

    console.log(obj.full_name())  // ANand Iyer
</script>
```

NOTE: Instead of defining *full_name* as a function, it's possible to define it as a property by using *get* keyword. Similarly, *set* keyword can be used to set one or more attributes of the object.

```
<script>
    obj = {
        first_name: "Anand",
        last_name: "Iyer",
        get full_name() {
            return (this.first_name + " "+ this.last_name)
        },
        set full_name(full) {
            let [first, last] = full.split(' ')
            this.first_name = first
            this.last_name = last
        }
    }

    console.log(obj.full_name)  // ANand Iyer
    obj.full_name = "Aravind Krishnan"
    console.log(obj.first_name) // Aravind
</script>
```

- When executed in the browser console, console.log(this) outputs Window object. However, on a node.js environment, output will be a global object.
- this refers to the calling object when used in a normal function, but since arrow functions work on the parent scope (global object), it will not work. Note that, in the following example, in order to add *name* to global object, it must be defined using *var* outside of the object definition *o*.

```
<script>
    o = {
        name: 'Anand',
        a: function() {
            console.log('hello', this.name)
        },

        b: () => {
            console.log('hello', this.name)
        }
    }

    o.a() //hello Anand
    o.b() //hello
</script>
```

NOTE: In the case of execution from a *browser console/node environment*, the variable shows up in global object only when it's declared using *var* keyword, and doesn't when declared using *let* or *const*; however, In the case of a *script* execution, the variable shows up in global object, irrespective of the declaration mode (*var, let* or *const*)

- To make a copy of a variable, use the spread operator like z = ...x
- In JavaScript, a function has a method named *call*, which when called will in turn invoke the function. The first parameter for the *call* method is the context in which you're invoking it.
- *apply* method of the function allows to call it in the context of another object, while also passing a list of arguments as a list.
- *bind* method of the function allows to create a closure and creates another short-cut function with a pre-defined argument.

```
<script>
    let obj = {
        first_name: "Anand",
        last_name: "Iyer",
        set_first_name: function (first) {
            this.first_name = first
        },
    }

    let obj2 = {
        first_name: "Bhaskar",
        last_name: "Banerjee",
    }

    f = obj.set_first_name
    f.call(obj, 'Aravind')
    console.log(obj.first_name) //Aravind

    console.log(obj2.first_name) //Bhaskar
    f = obj.set_first_name
    f.call(obj2, 'Aravind')
    console.log(obj2.first_name) //Aravind

    f.apply(obj, ['Krishna'])
    console.log(obj.first_name) //Krishna

    f1 = f.bind(obj, 'Ramesh') //Creating a closure - a new function
    f1.call(obj)
    console.log(obj.first_name) //Ramesh
</script>
```

- When used with arrow functions, the first argument gets ignored in *call*, *apply* and *bind* and always refer to the enclosing context.
- Implementation of class in JavaScript uses *__proto__* keyword.

```
const x = {a: 1, inc: function() {this.a++;}};
console.log(x);
const y = {__proto__: x, b: 2};
console.log(y);
console.log(y.a);
y.inc();
console.log(y.a);
```

Here, *x* is a prototype of *y*, which means that *y* inherits all properties and functions defined by *x*. Thus, the last line in the above code will print *2* (increments value of a = 1)

- Here's an example of class implementation and inheritance.

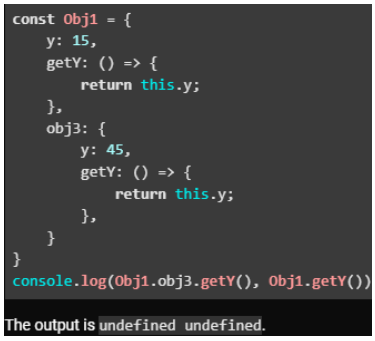
```
class Animal {
  constructor(name) {
    this.name = name;
  }
  describe() {
    return `${this.name} makes a sound $
    {this.sound}`
  }
}
let x = new Animal('Jerry');
console.log(x.describe());

class Dog extends Animal {
  constructor(name) {
    super(name);
    this.sound = 'Woof';
  }
}
let d = new Dog('Spike');
console.log(d.describe());
```

The parent class *Animal describes* the object using its *name* and its *sound*. Although the *name* is defined, the *sound* is not defined until *Animal* class has been extended to create a *Dog* class.

Thus, in the above code, the first x.describe() prints "Jerry makes a sound undefined" whereas the last x.describe() prints "Spike makes a sound Woof"

- In order to get the name of the class from its object O, use it like O.constructor.name
- When arrow functions are used, it's not mandatory to use flower brackets if there's a single statement in the function block. No need to use *return* keyword in this case.
- Hoisting allows a user to use a named normal function before it is defined. Note that arrow functions aren't hoisted.
- All variables including those declared using *var*, *let* or *const* are hoisted. However, *let* and *const* variables are not accessible before they're defined, since they're "hoisted" into a temporary dead zone, unlike those declared with *var*. Accessing such variables (before they're defined) will result in *ReferenceError*.
- Whenever a property is accessed via an object, the output always comes as *undefined*, if the property accessed does not exist. However, if the variable is accessed directly (without referencing via an object), it causes *ReferenceError*, if the variable doesn't exist.



- To redefine an existing variable *xyz* as a function, it's not sufficient to define *xyz* as a function, but must also be explicitly assigned to the variable *xyz.* This is unlike in Python.
- A task from task queue will be pushed to call stack, only if the call stack is empty. This is the run-to-completion semantics in JS.

Week4

- @click can be used as a shorthand for v-on:click during event binding.
- Computed property is generally derived from reactive properties of Vue instances.
- Computed properties are reactive.
- Computed properties are by default *getters*.
- Computed properties are cached based on their reactive dependencies.
- Computed properties should not be directly mutated.
- Watcher is a function which is triggered when the property it refers to changes.
- An element with *v-show* directive is always rendered irrespective of the condition's truth value.
- When *v-if* and *v-for* is used on the same element, *v-for* is evaluated first.
- *ref* in Vue allows us to directly reference the DOM element, *and* can be accessed only after the element is mounted.
- Vue components must be named when defining it using new Vue() construct, or need to be registered with the parent object.
- *v-bind* directive is used for one way data binding.
- *v-model* directive is used for two-way data binding.

Week5

- *BeforeCreate* event gets called before the data, methods and watchers are setup.
- *Created* event gets called after the above are setup, but before the instance is mounted. *el* will not be available during this event.
- *BeforeMount* event gets called after the *el* is available, but before mounting process is completed.
- Lifecycle hooks can't be handled using arrow functions, but full functions.
- It's typical to *fetch* data from backend during *Mounted* event, but it can also be used during *Created* event. These events can be handled in async.
- The lifecycle hooks are triggered implicitly, depending on the state of the component.
- All events like mouse or button clicks are asynchronous. Apart from these, there're certain JavaScript functions like *setTimeOut* that're inherently asynchronous.
- Calling a function with *async* keyword before definition delivers a *Promise* object

- Promise is a proxy for some value not necessarily known during creation. Promise always runs asynchronously.
- It can take two callback functions as argument.
- The first argument is called when the promise is resolved, and second is called when the promise is rejected
- *Promise* can be in any of 3 states = accepted, rejected or pending
- Using a *then* keyword on a Promise object, will resolve it (using the resolve_handler function). But, the timing of resolution isn't predictable. Optionally, it can also take a reject_handler function as its second parameter.
- Catching an error from an async function (Promise object) must be done using *.catch* syntax.
- The keyword *await* can only be used inside async functions, except Chrome/Firefox browser console.
- The promises in JavaScript can be resolved synchronously, but are designed to work asynchronously.

```
<script>
    const promise = Promise.resolve(42);
    promise.then(value => console.log(value)); // This will be executed synchronously
    console.log("This will be logged before the value of the promise");
</script>
```

- Here is a general technique to solve problems related to promises and async/await.
    - Follow path of execution. Start with the main program. Ignore function definitions.
    - The default path of execution inside a function or even a promise is always synchronous. This is true even when the function is defined as *async*.
    - When execution reaches *resolve/reject* phase of a promise, it skips and executes the next line. If *resolve/reject* occurs inside a function, it skips all remaining lines in the function.
    - setTimeOut inside an async function makes it asynchronous.
- *Fetch* method returns a promise object that's guaranteed to resolve, unless there's a network error. Resolution can be HTTP errors though.
- *Though fetch* API supports the use of the async/await syntax, which allows you to write asynchronous code in a synchronous-looking style, the underlying network request is still processed asynchronously.
- *Fetch* method has two parameters = *URL*, and an *init* object.
- The option "credentials : 'omit'" ensures that no cookies are sent with the request.
- A static method cannot be invoked by a class object in JavaScript.
- A child class constructor must call the parent class constructor to instantiate the child class.
- A function that accepts another function as its parameter, or one that returns another function is called a higher-order function
- *XMLHttpRequest*, *Fetch*, *SetTimeOut* are all examples of asynchronous APIs
- *Finally* block doesn't take an input, or return anything.
- *Catch* block catches exception raised from the previous block. If the previous block doesn't raise an exception, it'll be skipped.

Week6

- *LocalStorage* as well as *SessionStorage* are client-side storage mechanisms and local to a specific browser. Data is stored as key-value pairs in string format in both cases.

- The data stored by a specific domain in local storage of the browser cannot be directly accessed by its subdomain.
- Data stored in *LocalStorage* persists across app refreshes. Data stored in *SessionStorage* expires when page session ends.
- Use *novalidate* to prevent form validation by the browser
- SFC allows *scoped* CSS(<style>), component modularization (<script>)and *pre-compiled* templates.
- Jest, Chai, Mocha are some of the tools to test Vue.
- Vuelidate and VeeValidate are libraries to perform Form validation with Vue.
- Server side validation must be performed, though client-side validation exists. Browser provides simple validations for certain type of input elements.
- Client side storage can be used to store web-generated documents for offline use, or personalized site preferences.

Week7

- *Props* are used to pass information from parent to child.
- *Events* are emitted to pass information from parent to child.
- Child can directly invoke parent function and modify parent data using *$parent*
- An SPA fetch and update parts of the DOM asynchronously as and when needed. It helps to improve speed of transaction on page, page loading speed etc.
- When the server is "thin", all states and updates are handled on the browser using JavaScript, and gives pure data API service.
- *Flux, NgRx and Redux* are state management frameworks. *NgRx* is used by Angular applications, and *Redux* is used by React applications.
- *Vuex* is a state management framework, and is reactive to changes.
- *State* in a *Vuex* should be changed only through committing mutations, from within a method. *Mutations* are always synchronous.
- When state access/changes need to be asynchronous, use *Actions*.
- *Getters* in a *Vuex* is similar to computed properties, but limited to the Vuex store.
- Store state can be accessed as *this.$store.state*
- Target path in router-link can be set using *to* attribute, which will render it as a hyperlink tag. To render it as any other tag, use *tag* attribute.
- *router-view* renders the component matched by the path in router-enabled app? *router-link* enables user navigation in router-enabled app
- A Web worker is a script started by a web content that runs in the background, and can perform computation and fetch requests and communicate back messages to the origin web content.

Week8

- *PUT* APIs update the entire resource is updated, whereas *PATCH* APIs update only the necessary fields
- Token Based, OAuthA and JSON Web Token (JWT) are the common methods of API authentication.
- Following are two issues with REST APIs, which are solved using *GraphQL*
    - Construction of complex queries requires client side handling.
    - When data is from multiple data sources, the fusion of data becomes an issue.

- *GraphQL* can execute a complex query to return a complex data set with a single request, whereas, REST APIs may require to make multiple requests for the same.
- *GraphQL* is a query language for APIs with a single-entry point. It allows a type system to describe the schema for backend data. It is not tied to any specific database or storage engine.
- *GraphQL* resolvers help resolve queries in any programming language, including Python, Java and C++
- *GraphQL* supports a *ContentType* called *application/graphql*, which is well suited to query the server for multiple resources. *GraphQL* also allows to make POST requests using JSON body.
- *GraphQL* helps to fetch exactly the same data which is needed, and avoids over fetching as
- well as under fetching.
- All the responses to the *GraphQL* requests should return 200 as the status code in general.
- In REST architecture, GET requests are generally considered cacheable.
- The response to a *GraphQL* request is a JSON response, with the result stored in the "data" field.
- *JAM Stack* stands for JavaScript, API and Markup. It takes storage, logic, and presentation into the consideration
- *JAM Stack* can't guarantee high performance and stability of the application, even it improves them.
- *Jekyll, Hugo and MkDocs* are a few common static site generators.
- A permalink is a permanent link which is not expected to change throughout the lifetime of a resource, and identifies a resource uniquely.