Agenda :

- What is ES6 ?
- ES6 Features
- Introduction to ES Modules
- Why is understanding the JS Module System important?
- Different Types of JS Module Systems ?
- JavaScript "use strict"
- Lodash - Overview
- Luxon - Overview

What is ES6?

JavaScript ES6 (also known as ECMAScript 2015 or ECMAScript 6) is the newer version of JavaScript that was introduced in 2015. ECMAScript is th standard that JavaScript programming language uses. ECMAScript provides the specification on how JavaScript programming language should work.

ES6 Features

JavaScript let

JavaScript `let` is used to declare variables. Previously, variables were declared using the `var` keyword.

The variables declared using `let` are **block-scoped**. This means they are only accessible within a particular block. For example,

```
// variable declared using let
let name = 'Sara';
{
    // can be accessed only inside
    let name = 'Peter';

    console.log(name); // Peter
}
console.log(name); // Sara
```

JavaScript const

The `const` statement is used to declare constants in JavaScript. For example,

```
// name declared with const cannot be changed
const name = 'Sara';
```

Once declared, you cannot change the value of a const variable.

JavaScript Arrow Function

In the **ES6** version, you can use arrow functions to create function expressions. For example,This function

```
// function expression
let x = function(x, y) {
   return x * y;
}
```

can be written as

```
// function expression using arrow function
let x = (x, y) => x * y;
```

JavaScript Classes

JavaScript class is used to create an object. Class is similar to a constructor function. For example,

```
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

Keyword `class` is used to create a class. The properties are assigned in a constructor function.

Now you can create an object. For example,

```
class Person {
  constructor(name) {
    this.name = name;
  }
}

const person1 = new Person('John');

console.log(person1.name); // John
```

Default Parameter Values

In the ES6 version, you can pass default values in the function parameters. For example,

```
function sum(x, y = 5) {

    // take sum
    // the value of y is 5 if not passed
    console.log(x + y);
}

sum(5); // 10
sum(5, 15); // 20
```

In the above example, if you don't pass the parameter for `y`, it will take **5** by default.

JavaScript Template Literals

The template literal has made it easier to include variables inside a string. For example, before you had to do:

```
const first_name = "Jack";
const last_name = "Sparrow";

console.log('Hello ' + first_name + ' ' + last_name);
```

This can be achieved using template literal by:

```
const first_name = "Jack";
const last_name = "Sparrow";

console.log(`Hello ${first_name} ${last_name}`);
```

JavaScript Destructuring

The destructuring syntax makes it easier to assign values to a new variable. For example,

```
// before you would do something like this
```

```
const person = {
    name: 'Sara',
    age: 25,
    gender: 'female'
}

let name = person.name;
let age = person.age;
let gender = person.gender;

console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

Using ES6 Destructuring syntax, the above code can be written as:

```
const person = {
    name: 'Sara',
    age: 25,
    gender: 'female'
}

let { name, age, gender } = person;

console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

JavaScript import and export

You could export a function or a program and use it in another program by importing it. This helps to make reusable components. For example, if yc have two JavaScript files named contact.js and home.js.

In contact.js file, you can **export** the `contact()` function:

```
// export
export default function contact(name, age) {
    console.log(`The name is ${name}. And age is ${age}.`);
}
```

Then when you want to use the contact() function in another file, you can simply import the function. For example, in home.js file:

```
import contact from './contact.js';

contact('Sara', 25);
// The name is Sara. And age is 25
```

JavaScript Promises

Promises are used to handle asynchronous tasks. For example,

```
// returns a promise
let countValue = new Promise(function (resolve, reject) {
   reject('Promise rejected');
});

// executes when promise is resolved successfully
countValue.then(
    function successValue(result) {
        console.log(result); // Promise resolved
    },
 )
```

JavaScript Rest Parameter and Spread Operator

You can use the **rest parameter** to represent an indefinite number of arguments as an array. For example,

```
function show(a, b, ...args) {
  console.log(a); // one
  console.log(b); // two
  console.log(args); // ["three", "four", "five", "six"]
}

show('one', 'two', 'three', 'four', 'five', 'six')
```

You pass the remaining arguments using `...` syntax. Hence, the name **rest parameter**.

You use the **spread syntax** `...` to copy the items into a single array. For example,

```
let arr1 = ['one', 'two'];
let arr2 = [...arr1, 'three', 'four', 'five'];
console.log(arr2); // ["one", "two", "three", "four", "five"]
```

Both the rest parameter and the spread operator use the same syntax. However, the spread operator is used with arrays (iterable values).

Introduction to ES Modules

ES Modules is the ECMAScript standard for working with modules. While Node.js has been using the CommonJS standard since years, the brows
never had a module system, as every major decision such as a module system must be first standardized by ECMAScript and then implemented

- ES Modules is the ECMAScript standard for working with modules.
- While Node.js has been using the CommonJS standard for years, the browser never had a module system, as every major decision such as a module system must be first standardized by ECMAScript and then implemented by the browser.
- This standardization process completed with ES6 and browsers started implementing this standard trying to keep everything well aligned, working all in the same way, and now ES Modules are supported in Chrome, Safari, Edge and Firefox (since version 60).
- Modules are very cool, because they let you encapsulate all sorts of functionality, and expose this functionality to other JavaScript files, as libraries.

The ES Modules Syntax

The syntax to import a module is :

```
import package from 'module-name'
```

while CommonJS uses

```
const package = require('module-name')
```

Module is a file that contains code to perform a specific task. A module may contain variables, functions, classes etc. Let's see an example,

Suppose, a file named **greet.js** contains the following code:

```
// exporting a function
export function greetPerson(name) {
    return `Hello ${name}`;
}
```

Now, to use the code of **greet.js** in another file, you can use the following code:

```
// importing greetPerson from greet.js file
import { greetPerson } from './greet.js';

// using greetPerson() defined in greet.js
let displayName = greetPerson('Jack');

console.log(displayName); // Hello Jack
```

Here,

- The `greetPerson()` function in the **greet.js** is exported using the `export` keyword

```
// importing greetPerson from greet.js file
import { greetPerson } from './greet.js';

// using greetPerson() defined in greet.js
let displayName = greetPerson('Jack');

console.log(displayName); // Hello Jack
```

Then, we imported greetPerson() in another file using the import keyword. To import functions, objects, etc., you need to wrap them around { }.

```
import { greet } from '/.greet.js';
```

> Note: You can only access exported functions, objects, etc. from the module. You need to use the export keyword for the particular function, objects, etc. to import them and use them in other files.

Export Multiple Objects

It is also possible to export multiple objects from a module. For example,

In the file **module.js**

```
// exporting the variable
export const name = 'JavaScript Program';

// exporting the function
export function sum(x, y) {
    return x + y;
}
```

In main file,

```
import { name, sum } from './module.js';

console.log(name);
```

```
let add = sum(4, 9);
console.log(add); // 13
```

Here,

```
import { name, sum } from './module.js';
```

This imports both the name variable and the sum() function from the module.js file.

Renaming imports and exports

If the objects (variables, functions etc.) that you want to import are already present in your main file, the program may not behave as you want. In th case, the program takes value from the main file instead of the imported file.

To avoid naming conflicts, you can rename these functions, objects, etc. during the export or during the import.

1. Rename in the module (export file)

```
// renaming import inside module.js
export {
    function1 as newName1,
    function2 as newName2
};

// when you want to use the module
// import in the main file
import { newName1, newName2 } from './module.js';
```

Here, while exporting the function from module.js file, new names (here, newName1 & newName2) are given to the function. Hence, when importing th function, the new name is used to reference that function.

2. Rename in the import file

```
// inside module.js
export {
    function1,
    function2
};

// when you want to use the module
// import in the required file with different name

import { function1 as newName1, function2 as newName2 } from './module.js';
```

Here, while importing the function, the new names (here, newName1 & newName2) are used for the function name. Now you use the new names reference these functions.

Default Export

You can also perform default export of the module. For example,

In the file **greet.js**:

```
// default export
export default function greet(name) {
    return `Hello ${name}`;
}
```

```
export const age = 23;
```

Then when importing, you can use:

```
import random_name from './greet.js';
```

While performing default export,

- random_name is imported from `greet.js` . Since, `random_name` is not in `greet.js` , the default export (`greet()` in this case) is exported as `random_name` .
- You can directly use the default export without enclosing curly brackets `{}` .

> **Note**: A file can contain multiple exports. However, you can only have one default export in a file.

Modules Always use Strict Mode

By default, modules are in strict mode. For example,

```
// in greet.js
function greet() {
    // strict by default
}

export greet();
```

Benefit of Using Module

- The code base is easier to maintain because different code having different functionalities are in different files.
- Makes code reusable. You can define a module and use it numerous times as per your needs.

What about browsers that do not support modules?

Use a combination of `type="module"` and `nomodule` :

```
<script type="module" src="module.js"></script><script nomodule src="fallback.js"></script>
```

Why is understanding the JS Module System important?

> Telling stories is as basic to human beings as eating. More so, in fact, for while food makes us live, stories are what make our lives worth living - Richard Kearney.

There were many common functionalities required across projects. We always end up copy-pasting those functionalities to new projects over and ov
again.

The problem was that whenever one piece of the code changed, We needed to manually sync those changes across all my projects. To avoid all thes
tedious manual tasks, We planned to extract the common functionalities and compose an npm package out of them. This way, others on the team wou
be able to re-use them as dependencies and simply update them whenever a new release was rolled out.

This approach had some advantages:

- If there was some change in the core library, then a change only had to be made in one place without refactoring all the applications' code for the same thing.
- All the applications remained in sync. Whenever a change was made, all the applications just needed to run the "npm update" command.

So, the next step was to publish the library

This was the toughest part, because there were a bunch of things bouncing around our heads, like:

1. How do we make the tree shakeable?
2. What JS module systems should I target (commonjs, amd, harmony).
3. Should we transpile the source?
4. Should we bundle the source?
5. What files should we publish?

Everyone of us has had these kind of questions bubbling in our heads while creating a library. Right?

Different Types of JS Module Systems ?

1. CommonJS

- Implemented by **node**
- Used for the **server side** when you have modules installed
- No runtime/async module loading
- import via “**require**”
- export via “**module.exports**”
- When you import you get back an object
- No **tree shaking,** because when you import you get an object
- No static analyzing, as you get an object, so property lookup is at runtime
- You always get a copy of an object, so **no live changes** in the module itself
- Poor cyclic dependency management
- Simple Syntax

2. AMD: Async Module Definition

- Implemented by **RequireJs**
- Used for the **client side (browser)** when you want dynamic loading of modules
- Import via “require”
- Complex Syntax

3. UMD: Universal Module Definition

- Combination of **Commonjs + AMD** (that is, Syntax of CommonJs + async loading of AMD)
- Can be used for both **AMD/CommonJs** environments
- UMD essentially creates a way to use either of the two, while also supporting the global variable definition. As a result, UMD modules are capable of working on both **client and server**.

4. ECMAScript Harmony (ES6)

- Used for both **server/client** side
- **Runtime/static loading** of modules supported
- When you **import,** you get back **bindings value** (actual value)
- Import via “import” and export via “export”
- **Static analyzing** — You can determine imports and exports at compile time (statically) — you only have to look at the source code, you don’t have to execute it
- **Tree shakeable,** because of **static analyzing** supported by ES6
- Always get an **actual value** so live changes in the module itself
- Better cyclic dependency management than CommonJS

Conclusion

ES Modules are one of the biggest features introduced in modern browsers. They are part of ES6 but the road to implement them has been long.

We can now use them! But we must also remember that having more than a few modules is going to have a performance hit on our pages, as it's or more step that the browser must perform at runtime.

Prior to the introduction of the ES6 standard, there wasn't any native implementation for organizing source code in server-side JavaScript. Th community relied heavily on CommonJS module format.

Nowadays, with the introduction and API stabilization of ES modules, developers can enjoy the many benefits associated with the release specificatio This article has highlighted the transition between both module systems and their interoperability.

JavaScript "use strict"

In this tutorial, you will learn about the JavaScript 'use strict' syntax with the help of examples.

In JavaScript, `'use strict';` states that the code should be executed in '**strict mode**'. This makes it easier to write good and secure JS code. F example,

```
myVariable = 9;
```

Here, myVariable is created without declaring. This works as a global variable in JavaScript. However, if you use this in strict mode, the program w throw an error. For example,

```
'use strict';

// Error
myVariable = 9;
```

The above code throws an error because myVariable is not declared. In strict mode, you cannot use the variable without declaring them.

To indicate this program is in the strict mode, we have used

```
'use strict';
```

at the top of the program.

You can declare the strict mode by adding `'use strict';` or `"use strict";` at the beginning of a program.

When you declare strict mode at the beginning of a program, it will have global scope and all the code in the program will execute in strict mode.

Strict Mode in Variable

In strict mode, using a variable without declaring it throws an error.

> **Note**: You need to declare strict mode at the **beginning** of the program. If you declare strict mode below some code, it won't work.

For example,

```
console.log("some code");

// 'use strict' is ignored
// must be at the top
"use strict";

x = 21; // does not throw an error
```

Strict Mode in Function

You can also use strict mode inside a function. For example,

```
myVariable = 9;
console.log(myVariable); // 9

function hello() {

    // applicable only for this function
    'use strict';

    string = 'hello'; // throws an error
}

hello();
```

If you use `'use strict';` inside a function, the code inside the function will be in strict mode.

In the above program, `'use strict';` is used inside the `hello()` function. Hence, the strict mode is applicable only inside the function.

As you can see, in the beginning of the program, `myVariable` is used without declaring.

If you declare `'use strict';` at the top of the program, you cannot use a variable without declaring it inside the function as well. For example,

```
// applicable to whole program
'use strict';

function hello() {
    string = 'hello'; // throws an error
}

hello();
```

> Note : Strict mode doesn't apply to block statements with {} braces.

Things Not Allowed in Strict Mode

1. Undeclared variable is not allowed.

```
'use strict';

a = 'hello'; // throws an error
```

2. Undeclared objects are not allowed.

```
'use strict';

person = {name: 'Carla', age: 25}; // throws an error
```

3. Deleting an object is not allowed.

```
'use strict';

let person = {name: 'Carla'};

delete person; // throws an error
```

4. Duplicating a parameter name is not allowed.

```
"use strict";

function hello(p1, p1) { console.log('hello')}; // throws an error

hello();
```

5. **Assigning to a non-writable property is not allowed.**

```
'use strict';

let obj1 = {};

Object.defineProperty(obj1, 'x', { value: 42, writable: false });

// assignment to a non-writable property
obj1.x = 9; // throws an error
```

6. Assigning to a getter-only property is not allowed.

```
'use strict';

let obj2 = { get x() { return 17; } };

// assignment to a getter-only property
obj2.x = 5; // throws an error
```

7. Assigning to a new property on a non-extensible object is not allowed.

```
'use strict';

let obj = {};
Object.preventExtensions(obj);

// Assignment to a new property on a non-extensible object
obj.newValue = 'new value'; // throws an error
```

8. Octal syntax is not allowed.

```
'use strict';

let a = 010; // throws an error
```

9. The variable name arguments and eval are not allowed.

```
'use strict';

let arguments = 'hello'; // throws an error
let eval = 44;
```

10. You cannot also use these reserved keywords in strict mode.

`implements` `interface` `let` `package` `private` `protected` `public` `static` `yield`

Benefits of Strict Mode

The use of strict mode:

- helps to write a cleaner code
- changes previously accepted silent errors (bad syntax) into real errors and throws an error message
- makes it easier to write "secure" JavaScript

What is Lodash?

A JavaScript utility library delivering consistency, modularity, performance, & extras. It provides utility functions for common programming tasks using th functional programming paradigm.

Features of Lodash

Let us understand in detail all the important features available with Lodash :

Collections

Lodash provides various functions for collections like each, map, reduce which are used to apply an operation on each item of a collection. It provide method like groupBy, countBy, max, min which processes collections and ease lot of tasks.

Arrays

Lodash provides various functions for arrays like to iterate and process arrays like first, initial, lastIndexOf, intersection, difference etc.

Functions

Lodash provides functions such as bind, delay, before, after etc.

Objects

Lodash provides functions to manipulate objects, to map objects and comparing objects. For example, keys, values, extends, extendsOwn, isEqua isEmpty etc.

Utilities

Lodash provides various utilities methods like noConflict, random, iteratee, escape etc.

Chaining

Lodash provides chaining methods as well like chain, value.

In subsequent chapters, we'll cover important functions of Lodash

Method 1: Using Lodash File in Browser

Step 1

As a first step, go to the official website of Lodash https://lodash.com/.

Observe that there is a download option available which gives you the latest lodash.min.js file CDN Copies available. Click on the link and select late link for lodash.min.js.

Step 2

Now, include **lodash.min.js** inside the **script** tag and start working with Lodash. For this, you can use the code given below −

```
<script type = "text/JavaScript"
   src = "https://cdn.jsdelivr.net/npm/lodash@4.17.20/lodash.min.js">
</script>
```

Method 2: Using Node.js

If you are opting for this method, make sure you have **Node.js** and **npm** installed on your system. You can use the following command to install Lodas
–

```
npm install lodash
```

You can observe the following output once Lodash is successfully installed −

```
+ lodash@4.17.20
added 1 package from 2 contributors and audited 1 package in 2.54s
found 0 vulnerabilities
```

Now, to test if Lodash works fine with Node.js, create the file tester.js and add the following code to it −

```
var _ = require('lodash');
var numbers = [1, 2, 3, 4];
var listOfNumbers = '';
_.each(numbers, function(x) { listOfNumbers += x + ' ' });
console.log(listOfNumbers);
```

Save the above program in **tester.js**. The following commands are used to compile and execute this program.

Command

```
\>node tester.js
```

Output

```
1 2 3 4
```

Overview of Lodash :

Lodash - Array

Lodash - Collection

Lodash - Date

Lodash - Function

Lodash - Lang

Lodash - Math

Lodash - Number

Lodash - Object

Lodash - Seq

Lodash - String

Lodash - Util

Lodash - Properties

Lodash - Methods

Ref Link : https://www.tutorialspoint.com/lodash/lodash_quick_guide.htm

[Lodash](https://lodash.com/docs/4.17.15](https://lodash.com/docs/4.17.15)

What is Luxon?

It is a library that makes it easier to work with dates and times in JavaScript. If you want, add and subtract them, format and parse them, ask them ha questions, and so on, it provides a much easier and comprehensive interface than the native types it wraps.

Luxon is a library for dealing with dates and times in JavaScript.

```
DateTime.now().setZone('America/New_York').minus({weeks:1}).endOf('day').toISO();
```

Features

- DateTime, Duration, and Interval types.
- Immutable, chainable, unambiguous API.
- Parsing and formatting for common and custom formats.
- Native time zone and Intl support (no locale or tz files).

Installation Guide :

https://moment.github.io/luxon/#/install

- Note : Installation Guide won't be covered by the Instructors in the live session, students can go through the guidelines.

A quick tour

Luxon is a library that makes it easier to work with dates and times in JavaScript. If you want, add and subtract them, format and parse them, ask the hard questions, and so on, Luxon provides a much easier and comprehensive interface than the native types it wraps. We're going to talk about the mo immediately useful subset of that interface.

This is going to be a bit brisk, but keep in mind that the API docs are comprehensive, so if you want to know more, feel free to **dive into them**.

Your First Date & Time :

The most important class in Luxon is `DateTime`. A DateTime represents a specific millisecond in time, along with a time zone and a locale. Here's or that represents May 15, 2017 at 8:30 in the morning:

```
const dt = DateTime.local(2017, 5, 15, 8, 30);
```

DateTime.local takes any number of arguments, all the way out to milliseconds (months are 1-indexed). Underneath, this is similar to a JavaScript Da object. But we've decorated it with lots of useful methods.

Creating a DateTime

There are lots of ways to create a DateTime by parsing strings or constructing them out of parts. You've already seen one, `DateTime.local()`, b let's talk about three more.

Get the current date and time

To get the current time, just do this:

```
const now = DateTime.now();
```

This is really the equivalent to calling `DateTime.local()` with no arguments, but it's a little clearer.

Create from an object

The most powerful way to create a DateTime instance is to provide an object containing all the information:

```
dt = DateTime.fromObject({day: 22, hour: 12 }, { zone: 'America/Los_Angeles', numberingSystem: 'beng'})
```

Don't worry too much about the properties you don't understand yet; the point is that you can set every attribute of a DateTime when you create it. Or thing to notice from the example is that we just set the day and hour; the year and month get defaulted to the current one and the minutes, seconds, ar milliseconds get defaulted to 0. So `DateTime.fromObject` is sort of the power user interface.

Parse from ISO 8601

Luxon has lots of parsing capabilities, but the most important one is parsing **ISO 8601** strings, because they're more-or-less the standard wire format f dates and times. Use `DateTime.fromISO` .

```
DateTime.fromISO("2017-05-15")          //=> May 15, 2017 at midnight
DateTime.fromISO("2017-05-15T08:30:00") //=> May 15, 2017 at 8:30
```

You can parse a bunch of other formats, including **your own custom ones**.

Getting to know your DateTime instance

Now that we've made some DateTimes, let's see what we can ask of it.

toString

The first thing we want to see is the DateTime as a string. Luxon returns ISO 8601 strings:

```
DateTime.now().toString(); //=> '2017-09-14T03:20:34.091-04:00'
```

Getting at components

We can get at the components of the time individually through getters. For example:

```
dt = DateTime.now();
dt.year     //=> 2017
dt.month    //=> 9
dt.day      //=> 14
dt.second   //=> 47
dt.weekday  //=> 4
```

Other fun accessors

```
dt.zoneName     //=> 'America/New_York'
dt.offset       //=> -240
dt.daysInMonth  //=> 30
```

There are lots more!

Formatting your DateTime

You may want to output your DateTime to a string for a machine or a human to read. Luxon has lots of tools for this, but two of them are most importar If you want to format a human-readable string, use `toLocaleString` :

```
dt.toLocaleString()      //=> '9/14/2017'
dt.toLocaleString(DateTime.DATETIME_MED) //=> 'September 14, 3:21 AM'
```

This works well across different locales (languages) by letting the browser figure out what order the different parts go in and how to punctuate them.

If you want the string read by another program, you almost certainly want to use `toISO` :

```
dt.toISO() //=> '2017-09-14T03:21:47.070-04:00'
```

Custom formats are also supported. See **formatting**.

Transforming your DateTime

Immutability

Luxon objects are immutable. That means that you can't alter them in place, just create altered copies. Throughout the documentation, we use terms lil
"alter", "change", and "set" loosely, but rest assured we mean "create a new instance with different properties".

Math

This is easier to show than to tell. All of these calls return new DateTime instances:

```
var dt = DateTime.now();
dt.plus({ hours: 3, minutes: 2 });
dt.minus({ days: 7 });
dt.startOf('day');
dt.endOf('hour');
```

Set

You can create new instances by overriding specific properties:

```
var dt = DateTime.now();
dt.set({hour: 3}).hour   //=> 3
```

Intl

Luxon provides several different Intl capabilities, but the most important one is in formatting:

```
var dt = DateTime.now();
var f = {month: 'long', day: 'numeric'};
dt.setLocale('fr').toLocaleString(f)      //=> '14 septembre'
dt.setLocale('en-GB').toLocaleString(f)   //=> '14 September'
dt.setLocale('en-US').toLocaleString(f)  //=> 'September 14'
```

Luxon's Info class can also list months or weekdays for different locales:

```
Info.months('long', {locale: 'fr'}) //=> [ 'janvier', 'février', 'mars', 'avril', ... ]
```

Time zones

Luxon supports time zones. There's a whole **big section** about it. But briefly, you can create DateTimes in specific zones and change their zones:

```
DateTime.fromObject({}, {zone: 'America/Los_Angeles'}); // now, but expressed in LA's local time
DateTime.now().setZone("America/Los_Angeles"); // same
```

Luxon also supports UTC directly:

```
DateTime.utc(2017, 5, 15);
DateTime.utc(); // now, in UTC time zone
DateTime.now().toUTC();
DateTime.utc().toLocal();
```

Durations

The Duration class represents a quantity of time such as "2 hours and 7 minutes". You create them like this:

```
var dur = Duration.fromObject({ hours: 2, minutes: 7 });
```

They can be added or subtracted from DateTimes like this:

```
dt.plus(dur);
```

They have getters just like DateTime:

```
dur.hours   //=> 2
dur.minutes //=> 7
dur.seconds //=> 0
```

And some other useful stuff:

```
dur.as('seconds') //=> 7620
dur.toObject()    //=> { hours: 2, minutes: 7 }
dur.toISO()       //=> 'PT2H7M'
```

You can also format, negate, and normalize them. See it all in the `Duration` API docs.

Intervals

Intervals are a specific period of time, such as "between now and midnight". They're really a wrapper for two DateTimes that form its endpoints. Here what you can do with them:

```
now = DateTime.now();
later = DateTime.local(2020, 10, 12);
i = Interval.fromDateTimes(now, later);

i.length()                             //=> 97098768468
i.length('years')                //=> 3.0762420239726027
i.contains(DateTime.local(2019))       //=> true

i.toISO()       //=> '2017-09-14T04:07:11.532-04:00/2020-10-12T00:00:00.000-04:00'
i.toString()    //=> '[2017-09-14T04:07:11.532-04:00 – 2020-10-12T00:00:00.000-04:00)
```

Intervals can be split up into smaller intervals, perform set-like operations with other intervals, and few other handy features. See the `Interval` AI docs.

Interview Questions

> What do you mean by strict mode in JavaScript and characteristics of JavaScript strict-mode?

In ECMAScript 5, a new feature called JavaScript Strict Mode allows you to write a code or a function in a "strict" operational environment. In most case this language is 'not particularly severe' when it comes to throwing errors. In 'Strict mode,' however, all forms of errors, including silent errors, will b thrown. As a result, debugging becomes a lot simpler. Thus programmer's chances of making an error are lowered.

Characteristics of strict mode in JavaScript

1. Duplicate arguments are not allowed by developers.
2. In strict mode, you won't be able to use the JavaScript keyword as a parameter or function name.
3. The 'use strict' keyword is used to define strict mode at the start of the script. Strict mode is supported by all browsers.
4. Engineers will not be allowed to create global variables in 'Strict Mode.

> What are Imports and Exports in JavaScript?

Imports and exports help in writing modular code for our JavaScript applications. With the help of imports and exports, we can split a JavaScript code in multiple files in a project. This greatly simplifies the application source code and encourages code readability.

```
export const sqrt = Math.sqrt;

export function square(x) {

  return x * x;
}

export function diag(x, y) {

  return sqrt(square(x) + square(y));

}
```

This file exports two functions that calculate the squares and diagonal of the input respectively.

```
import { square, diag } from "calc";

console.log(square(4)); // 16

console.log(diag(4, 3)); // 5
```

Therefore, here we import those functions and pass input to those functions to calculate square and diagonal.