OOPS IN JAVASCRIPT

1. **Object**–

An Object is a **unique** entity which contains **property** and **methods**. For example "car" is a real life Object, which have some characteristics like color, type, model, horsepower and performs certain action like drive. The characteristics of an Object are called as Property, in Object Oriented Programming and the actions are called methods. An Object is an **instance** of a class. Objects are everywhere in JavaScript almost every element is an Object whether it is a function,arrays and string.
Note: A Method in javascript is a property of an object whose value is a function.
Object can be created in two ways in JavaScript:

==> Using an **Object Literal**

```
//Defining object
let person = {
    first_name:'Rajeev',
    last_name: 'Kumar',

    //method
    getFunction : function(){
        return (`The name of the person is
          ${person.first_name} ${person.last_name}`)
    },
    //object within object
    phone_number : {
        mobile:'12345',
        landline:'6789'
    }
}
console.log(person.getFunction());
console.log(person.phone_number.landline);
```

==> Using an **Object Constructor:**

```
//using a constructor
function person(first_name,last_name){
   this.first_name = first_name;
   this.last_name = last_name;
}
//creating new instances of person object
let person1 = new person('Rajeev','Kumar');
let person2 = new person('Rahul','Anand');

console.log(person1.first_name);
console.log(`${person2.first_name} ${person2.last_name}`);
```

==>Using Object.create() method: The Object.create() method creates a new object, using an existing object as the prototype of the newly created object.

```
// Object.create() example a

// simple object with some properties
const coder = {
    isStudying : false,
    printIntroduction : function(){
        console.log(`My name is ${this.name}. Am I
          studying?: ${this.isStudying}.`)
    }
}
// Object.create() method
const me = Object.create(coder);

// "name" is a property set on "me", but not on "coder"
me.name = 'Rajeev';

// Inherited properties can be overwritten
me.isStudying = 'True';

me.printIntroduction();
```

2. **Classes–**
Classes are **blueprint** of an Object. A class can have many Object, because class is a **template** while Object are **instances** of the class or the concrete implementation.
Before we move further into implementation, we should know unlike other Object Oriented Language there is **no classes in JavaScript** we have only Object. To be more precise, JavaScript is a prototype based object oriented language, which means it doesn't have classes rather it define behaviors using constructor function and then reuse it using the prototype.

```
// Defining class using es6
class Vehicle {
  constructor(name, maker, engine) {
    this.name = name;
    this.maker =  maker;
    this.engine = engine;
  }
  getDetails(){
      return (`The name of the bike is ${this.name}.`)
  }
}
// Making object with the help of the constructor
let bike1 = new Vehicle('Hayabusa', 'Suzuki', '1340cc');
let bike2 = new Vehicle('Ninja', 'Kawasaki', '998cc');

console.log(bike1.name);    // Hayabusa
console.log(bike2.maker);   // Kawasaki
console.log(bike1.getDetails());

// Defining class in a Traditional Way.
function Vehicle(name,maker,engine){
    this.name = name,
    this.maker = maker,
    this.engine = engine
};

Vehicle.prototype.getDetails = function(){
   console.log('The name of the bike is '+ this.name);
}

let bike1 = new Vehicle('Hayabusa','Suzuki','1340cc');
let bike2 = new Vehicle('Ninja','Kawasaki','998cc');

console.log(bike1.name);
console.log(bike2.maker);
console.log(bike1.getDetails());
```

3. **Encapsulation –**
The process of **wrapping property and function** within a **single unit** is known as encapsulation.
Let’s understand encapsulation with an example.

```
//encapsulation example
class person{
   constructor(name,id){
      this.name = name;
      this.id = id;
   }
   add_Address(add){
      this.add = add;
   }
   getDetails(){
      console.log(`Name is ${this.name},Address is: ${this.add}`);
   }
}

let person1 = new person('Rajeev',21);
person1.add_Address('Delhi');
person1.getDetails();
```

In the above example we simply create an *person* Object using the constructor and Initialize it property and use it functions we are not bother about the implementation details. We are working with an Objects interface without considering the implementation details.
Sometimes encapsulation refers to **hiding of data** or **data Abstraction** which means representing essential features hiding the background detail. Most of the OOP languages provide access modifiers to restrict the scope of a variable, but their are no such access modifiers in JavaScript but their are certain way by which we can restrict the scope of variable within the Class/Object.

```
// Abstraction example
function person(fname,lname){
   let firstname = fname;
   let lastname = lname;

   let getDetails_noaccess = function(){
      return (`First name is: ${firstname} Last
            name is: ${lastname}`);
    }

    this.getDetails_access = function(){
        return (`First name is: ${firstname}, Last
            name is: ${lastname}`);
    }
}
let person1 = new person('Rajeev','Kumar');
console.log(person1.firstname);
console.log(person1.getDetails_noaccess);
console.log(person1.getDetails_access());
```

In the above example we try to access some property(*person1.firstname*) and functions(*person1.getDetails_noaccess*) but it returns *undefine* while their is a method which we can access from the *person* object(*person1.getDetails_access()*), by changing the way to define a function we can restrict its scope.

4. **Inheritance –**
It is a concept in which some property and methods of an Object is being used by another Object. Unlike most of the OOP languages where classes inherit classes, JavaScript Object inherits Object i.e. certain features (property and methods)of one object can be reused by other Objects.
Lets's understand inheritance with example:

```
//Inhertiance example

class person{
    constructor(name){
        this.name = name;
    }
    //method to return the string
    toString(){
        return (`Name of person: ${this.name}`);
    }
}
class student extends person{
    constructor(name,id){
        //super keyword to for calling above class constructor
        super(name);
        this.id = id;
    }
    toString(){
        return (`${super.toString()},Student ID: ${this.id}`);
    }
}
let student1 = new student('Rajeev',22);
console.log(student1.toString());
```

In the above example we define an *Person* Object with certain property and method and then we *inherit* the *Person* Object in the *Student* Object and use all the property and method of person Object as

well define certain property and methods for *Student*.
Note: The Person and Student object both have same method i.e toString(), this is called as **Method Overriding**. Method Overriding allows method in a child class to have the same name and method signature as that of a parent class.
In the above code, super keyword is used to refer immediate parent class instance variable.

Another example:

```
class Animal {
constructor(name, weight) {
this.name = name;
this.weight = weight;
}

eat() {
return `${this.name} is eating!`;
}

sleep() {
return `${this.name} is going to sleep!`;
}

wakeUp() {
return `${this.name} is waking up!`;
}

}

class Gorilla extends Animal {
constructor(name, weight) {
super(name, weight);
}

climbTrees() {
return `${this.name} is climbing trees!`;
}

poundChest() {
return `${this.name} is pounding its chest!`;
}

showVigour() {
```

```
return `${super.eat()} ${this.poundChest()}`;
}

dailyRoutine() {
return `${super.wakeUp()} ${this.poundChest()} ${super.eat()} ${super.sleep()}`;
}

}

function display(content) {
console.log(content);
}

const gorilla = new Gorilla('George', '160Kg');
display(gorilla.poundChest());
display(gorilla.sleep());
display(gorilla.showVigour());
display(gorilla.dailyRoutine());
```

Traditional JavaScript Classes

```
function Animal(name, weight) {
this.name = name;
this.weight = weight;
}

Animal.prototype.eat = function() {
return `${this.name} is eating!`;
}

Animal.prototype.sleep = function() {
return `${this.name} is going to sleep!`;
}

Animal.prototype.wakeUp = function() {
return `${this.name} is waking up!`;
}
```

```
function Gorilla(name, weight) {
Animal.call(this, name, weight);
}

Gorilla.prototype = Object.create(Animal.prototype);
Gorilla.prototype.constructor = Gorilla;

Gorilla.prototype.climbTrees = function () {
return `${this.name} is climbing trees!`;
}

Gorilla.prototype.poundChest = function() {
return `${this.name} is pounding its chest!`;
}

Gorilla.prototype.showVigour = function () {
return `${Animal.prototype.eat.call(this)} ${this.poundChest()}`;
}

Gorilla.prototype.dailyRoutine = function() {
return `${Animal.prototype.wakeUp.call(this)} ${this.poundChest()} $
{Animal.prototype.eat.call(this)} ${Animal.prototype.sleep.call(this)}`;
}

function display(content) {
console.log(content);
}

var gorilla = new Gorilla('George', '160Kg');
display(gorilla.poundChest());
display(gorilla.sleep());
display(gorilla.showVigour());
display(gorilla.dailyRoutine());
```

What is the drawback of creating true private methods in JavaScript?

One of the drawbacks of creating true private methods in JavaScript is that they are very memory-inefficient, as a new copy of the method would be created for each instance.

```
var Employee = function (name, company, salary) {
    this.name = name || "";       //Public attribute default value is null
    this.company = company || ""; //Public attribute default value is null
    this.salary = salary || 5000; //Public attribute default value is null

    // Private method
    var increaseSalary = function () {
        this.salary = this.salary + 1000;
    };

    // Public method
    this.dispalyIncreasedSalary = function() {
        increaseSlary();
        console.log(this.salary);
    };
};

// Create Employee class object
var emp1 = new Employee("John","Pluto",3000);
// Create Employee class object
var emp2 = new Employee("Merry","Pluto",2000);
// Create Employee class object
var emp3 = new Employee("Ren","Pluto",2500);
```

Here each instance variable `emp1`, `emp2`, `emp3` has its own copy of the `increaseSalary` private method.

So, as a recommendation, don't use private methods unless it's necessary.

What is a "closure" in JavaScript? Provide an example

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
```

Write a `mul` function which will produce the following outputs when invoked:

```
console.log(mul(2)(3)(4)); // output : 24
console.log(mul(4)(3)(4)); // output : 48
```

Below is the answer followed by an explanation to how it works:

```
function mul (x) {
    return function (y) { // anonymous function
        return function (z) { // anonymous function
            return x * y * z;
        };
    };
}
```

Here the `mul` function accepts the first argument and returns an anonymous function, which takes the second parameter and returns another anonymous function that will take the third parameter and return the multiplication of the arguments that have been passed.

In JavaScript, a function defined inside another one has access to the outer function's variables. Therefore, a function is a first-class object that can be returned by other functions as well and be passed as an argument in another function.

- A function is an instance of the Object type
- A function can have properties and has a link back to its constructor method
- A function can be stored as a variable
- A function can be pass as a parameter to another function
- A function can be returned from another function

ow 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** `arrayList` anywhere 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.

How do you check if an object is an array or not?

```
if($.isArray(arrayList)){
    console.log('Array');
}else{
        console.log('Not an array');
}
```

FYI, jQuery uses `Object.prototype.toString.call` internally to check whether an object is an array or not.

In modern browsers, you can also use

```
Array.isArray(arrayList);
```

What will be the output of the following code?

```
var output = (function(x){
    delete x;
    return x;
  })(0);

  console.log(output);
```

The output would be `0`. The `delete` operator is used to delete properties from an object. Here `x` is not an object but a **local variable**. `delete` operators don't affect local variables.

What will be the output of the code below?

```
var Employee = {
  company: 'xyz'
}
var emp1 = Object.create(Employee);
delete emp1.company
console.log(emp1.company);
```

The output would be `xyz`. Here, `emp1` object has `company` as its **prototype** property. The `delete` operator doesn't delete prototype property.

`emp1` object doesn't have **company** as its own property. You can test it `console.log(emp1.hasOwnProperty('company')); //output : false`. However, we can delete the `company` property directly from the`Employee` object using `delete Employee.company`. Or, we can also delete the `emp1` object using the `__proto__` property `delete emp1.__proto__.company`.

What will be the output of the code below?

```
var trees = ["xyz","xxxx","test","ryan","apple"];
delete trees[3];
```

```
  console.log(trees.length);
```

The output would be `5`. When we use the `delete` operator to delete an array element, the array length is not affected from this. This holds even if you deleted all elements of an array using the `delete` operator.

In other words, when the `delete` operator removes an array element, that deleted element is not longer present in array. In place of value at deleted index `undefined x 1` in **chrome** and `undefined` is placed at the index. If you do `console.log(trees)` output `["xyz", "xxxx", "test", undefined × 1, "apple"]` in Chrome and in Firefox `["xyz", "xxxx", "test", undefined, "apple"]`.

What will be the output of the code below?

```
var bar = true;
console.log(bar + 0);
console.log(bar + "xyz");
console.log(bar + true);
console.log(bar + false);
```

The code will output `1, "truexyz", 2, 1`. Here's a general guideline for addition operators:

- Number + Number -> Addition
- Boolean + Number -> Addition
- Boolean + Number -> Addition
- Number + String -> Concatenation
- String + Boolean -> Concatenation
- String + String -> Concatenation

What will be the output of the code below?

```
var z = 1, y = z = typeof y;
console.log(y);
```

The output would be `undefined`. According to the `associativity` rule, operators with the same precedence are processed based on the associativity property of the operator. Here, the associativity of the assignment operator is `Right to Left`, so `typeof y` will evaluate first , which is `undefined`. It will be assigned to `z`, and then `y` would be assigned the value of z and then `z` would be assigned the value `1`.

What will be the output of the code below?

```
// NFE (Named Function Expression
var foo = function bar(){ return 12; };
typeof bar();
```

The output would be `Reference Error`. To make the code above work, you can re-write it as follows:

Sample 1

```
var bar = function(){ return 12; };
typeof bar();
```

or

Sample 2

```
function bar(){ return 12; };
typeof bar();
```

A function definition can have only one reference variable as its function name. In **sample 1**, `bar`'s reference variable points to `anonymous function`. In **sample 2**, the function's definition is the name function.

```
var foo = function bar(){
    // foo is visible here
    // bar is visible here
        console.log(typeof bar()); // Work here :)
 };
// foo is visible here
// bar is undefined here
```

What is the difference between the function declarations below?

```
var foo = function(){
    // Some code
};

function bar(){
    // Some code
};
```

The main difference is the function `foo` is defined at `run-time` whereas function `bar` is defined at parse time. To understand this in better way, let's take a look at the code below:

```
Run-Time function declaration
<script>
foo(); // Calling foo function here will give an Error
  var foo = function(){
    console.log("Hi I am inside Foo");
 };
 </script>

<script>
Parse-Time function declaration
bar(); // Calling foo function will not give an Error
 function bar(){
  console.log("Hi I am inside Foo");
 };
</script>
```

Another advantage of this first-one way of declaration is that you can declare functions based on certain conditions. For example:

```
<script>
if(testCondition) {// If testCondition is true then
   var foo = function(){
    console.log("inside Foo with testCondition True value");
   };
 }else{
         var foo = function(){
    console.log("inside Foo with testCondition false value");
   };
}
</script>
```

However, if you try to run similar code using the format below, you'd get an error:

```
<script>
if(testCondition) {// If testCondition is true then
   function foo(){
    console.log("inside Foo with testCondition True value");
   };
 }else{
         function foo(){
    console.log("inside Foo with testCondition false value");
   };
}
</script>
```

What is function 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 will be the output of code below?

```
var salary = "1000$";

 (function () {
     console.log("Original salary was " + salary);

     var salary = "5000$";

     console.log("My New Salary " + salary);
})();
```

The output would be `undefined, 5000$`. Newbies often get tricked by JavaScript's hoisting concept. In the code above, you might be expecting `salary` to retain its value from the outer scope until the point that `salary` gets re-declared in the inner scope. However, due to `hoisting`, the salary value was `undefined` instead. To understand this better, have a look of the code below:

```
var salary = "1000$";

(function () {
    var salary = undefined;
    console.log("Original salary was " + salary);

    salary = "5000$";

    console.log("My New Salary " + salary);
})();
```

`salary` variable is hoisted and declared at the top in the function's scope. The `console.log` inside returns `undefined`. After the `console.log`, `salary` is redeclared and assigned `5000$`.

What is the `instanceof` operator in JavaScript? What would be the output of the code below?

```
function foo(){
  return foo;
}
new foo() instanceof foo;
```

Here, `instanceof` operator checks the current object and returns true if the object is of the specified type.

For Example:

```
var dog = new Animal();
dog instanceof Animal // Output : true
```

Here `dog instanceof Animal` is true since `dog` inherits from `Animal.prototype`.

```
var name = new String("xyz");
name instanceof String // Output : true
```

Here `name instanceof String` is true since `dog` inherits from `String.prototype`. Now let's understand the code below:

```
function foo(){
  return foo;
}
new foo() instanceof foo;
```

Here function `foo` is returning `foo`, which again points to function `foo`.

```
function foo(){
  return foo;
}
var bar = new foo();
// here bar is pointer to function foo(){return foo}.
```

So the `new foo() instanceof foo` return `false`;

How can we calculate the length of the above associative array's `counterArray`?

There are no in-built functions and properties available to calculate the length of associative array object here. However, there are other ways by which we can calculate the length of an associative array object. In addition to this, we can also extend an `Object` by adding a method or property to the prototype in order to calculate length. However, extending an object might break enumeration in various libraries or might create cross-browser issues, so it's not recommended unless it's necessary. Again, there are various ways by which we can calculate length.

`Object` has the `keys` method which can be used to calculate the length of an object:

~~~
We can also calculate the length of an object by iterating through an object and by
counting the object's own property.

```javascript
function getSize(object){
  var count = 0;
  for(key in object){
    // hasOwnProperty method check own property of object
    if(object.hasOwnProperty(key)) count++;
  }
  return count;
}
~~~

We can also add a `length` method directly on `Object`:

```
Object.length = function(){
        var count = 0;
  for(key in object){
    // hasOwnProperty method check own property of object
    if(object.hasOwnProperty(key)) count++;
  }
  return count;
}
//Get the size of any object using
```

```
console.log(Object.length(counterArray))
```

Bonus: We can also use `Underscore` (recommended, As it's lightweight) to calculate object length.

What is difference between Array.splice() and Array.slice() method in JavaScript?

- The array.slice() removes items from the array and then return those removed items as an array whereas array.slice() method is selected items from an array and then those elements as a new array object.
- The splice() method affects the original array whereas slice() method doesn’t affect the original array.
- Splice() method takes n number of arguments whereas slice() can take only two arguments.

Syntax of splice(): array.splice(index, howmany, item1, ..., itemX)

Syntax of slice(): array.slice(start, end)

Explain Promise in JavaScript?

A promise is an object in JavaScript which is used to produce a value that may give result in the future. The value can be resolved value or it can be a reason which tells why the value is not resolved.

A promise can be of three states:

- Fulfilled: The operation is completed and the promise has a specific value.
- Rejected: The operation is failed and promise has a reason which shows why the operation failed.
- Pending: Th operation is not fulfilled or rejected, means it has not completed yet.

How to remove duplicates from JavaScript Array?

There are many ways to remove duplicates from JavaScript array. These are described below with examples:

1. By using Set: It is the simplest approach to remove duplicates. Set is an inbuilt object to store unique values in an array. Here's how we use set:

```
    function uniquearray(array) {
       let unique_array= Array.from(set(array))
       return unique_array;}
```

As in the above code, you created a set of an array which automatically eliminates the duplicate values.

2. By using Filter: Another approach to remove duplicates from an array is applying filter on an array. To call filter method, it requires three arguments: array, current element, index of current element. Here’s how we use filter:

```
function unque_array (arr){
 let unique_array = arr.filter(function(elem, index, self) {
 return index == self.indexOf(elem); }
return unique_array }
 console.log(unique_array(array_with_duplicates));
```

3. By using for loop: In this, we can use for loop to remove duplicates. In this we make an empty array in which those elements will be added from the duplicate array which are not present in this before. Thus, finally we will get an array which has unique elements. Code to implement this:

```
Array dups_names = ['Ron', 'Pal', 'Fred', 'Rongo', 'Ron'];
function dups_array(dups_names) {
 let unique = {};
 names.forEach(function(i) {
    If (!unique[i]) {
      unique[i] = true;    }
  });
return Object.keys(unique);}   // Ron, Pal, Fred, Rongo
Dups_array(names);
```

These are the main three methods used in JavaScript to get a unique array.

Explain few difference between null, undefined or undeclared JavaScript variable?

Null is a value that can be assigned to a variable or an object.

Undefined means a variable has been declared but no value is assigned to it. This type of variable is declared itself to be undefined.

Undeclared means the variable has declared without any datatype.

Null, Undefined are primitive data types whereas Undeclared is not a primitive data type.

Explain spread operator in JavaScript?

The spread operator expands an expression in places where multiple argument/variables/elements are needed to present. It represents with three dots (…).

For example:

var mid = [3, 4];

var newarray = [1, 2, ...mid, 5, 6];

console.log(newarray);

// [1, 2, 3, 4, 5, 6]

In above example, instead of appending mid array, it rather expands in the newarray with the help of spread operator. This is how spread operator works in JavaScript.

Explain Arrow functions?

An arrow function is a consise and short way to write function expressions in Es6 or above.A rrow functions cannot be used as constructors and also does not supports this, arguments, super, or new.target keywords. It is best suited for non-method functions. In general an arrow function looks like `const function_name= ()=>{}`

```
 const greet=()=>{console.log('hello');}
 greet();
```

Explain function hoisting in JavaScript?

JavaScript's default behavior that allows moving declarations to the top is called Hoisting. The 2 ways of creating functions in JavaScript are **Function Declaration** and **Function Expression**. Let's find out more about these:

Function Declaration

A function with the specific parameters is known as function declarations. To create a variable in JavaScript is called declarations.

e.g:

```
hoisted(); // logs "foo"

function hoisted() {

  console.log('foo');

}
```

Function Expression

When a function is created by using an expression it is called function expression.

e.g:

```
notHoisted(); // TypeError: notHoisted is not a function

var notHoisted = function() {

   console.log('bar');

};
```

How to remove duplicate values from a JavaScript array?

We can use array.indexOf method to check a value exists or not. See below example to remove duplicate values.

let duplicates = ['delhi','kanpur','kanpur','goa','delhi','new york'];

```
function removeDuplicatesValues(arr){
    let unique_array = [];
    for(let i = 0;i < arr.length; i++){
        if(unique_array.indexOf(arr[i]) == -1){
            unique_array.push(arr[i])
        }
    }
    return unique_array
}

console.log(removeDuplicatesValues(duplicates));
```

What is the difference between let and var?

Both var and let are used for variable/ method declaration in javascript but the main difference between let and var is that **var** is function scoped whereas **let** is block scoped.

How to clone an object in Javascript?

Object.assign() method is used for cloning an object in Javascript.Here is sample usage

```
        var x = {myProp: "value"};
        var y = Object.assign({}, x);
```

Explain Typecasting in Javascript?

In Programming whenever we need to convert a variable from one data type to another Typecasting is used. In Javascript, we can do this via library functions. There are basically 3 typecasts are available in Javascript Programming, they are:

- Boolean(value): Casts the inputted value to a Boolean
- Number(value): Casts the inputted value to an Integer or Floating point Number.
- String(value) : Casts the inputted value value a string

List different ways of empty an array in Javascript?

In Javascript, there are many ways to empty an array in Javascript, below we have listed 4 major

- By assigning an empty array.

  ```
  var arr1 =[1,4,5,6];
  arr1=[];
  ```

- By assigning array length to 0.

  ```
  var arr2 =[1,4,5,6];
  arr2.length=0;
  ```

- By poping the elements of the array.

  ```
  var arr2 =[1,4,5,6];
  while(arr.length > 0) {
      arr.pop();
  }
  ```

- By using .splice() .

  ```
  var arr =[1,4,5,6];
  ```

```
    arr.splice(0,arr.length)
```

What does the instanceof operator do?

In Javascript **instanceof** operator checks whether the object is an instance of a class or not:

Example Usage

```
Square.prototype = new Square();
console.log(sq instanceof Square); // true
```

How to get the primitive value of a string in Javascript?

In Javascript **valueOf()** method is used to get the primitive value of a string.

Example Usage:

```
var myVar= "Hi!"
console.log(myVar.valueOf())
```

What are different types of Inheritence? Which Inheritance is followed in Javascript.

There are two types of Inherientence in OOPS Classic and Prototypical Inheritance. Javascript follows Prototypical Inheritance.

What is output of undefined * 2 in Javascript?

nan is output of undefined * 2.

###Difference between constructor and ngOninit:

==> The `Constructor` is a default method of the class that is executed when the class is instantiated and ensures proper initialization of fields in the class and its subclasses. Angular or better Dependency Injector (DI) analyzes the constructor parameters and when it creates a new instance by calling `new MyClass()` it tries to find providers that match the types of the constructor parameters, resolves them and passes them to the constructor like.

`ngOnInit` is a life cycle hook called by Angular2 to indicate that Angular is done creating the component.

Mostly we use `ngOnInit` for all the initialization/declaration and avoid stuff to work in the constructor. The constructor should only be used to initialize class members but shouldn't do actual "work".

So you should use `constructor()` to setup Dependency Injection and not much else. ngOnInit() is better place to "start" - it's where/when components' bindings are resolved.

###ActivateRouting and Routestate:

Since `ActivatedRoute` can be reused, `ActivatedRouteSnapshot` is an immutable object representing **a particular version** of `ActivatedRoute`. It exposes all the same properties as `ActivatedRoute` as plain values, while `ActivatedRoute` exposes them as observables.

Here is the comment in the implementation:

```
export class ActivatedRoute {
  /** The current snapshot of this route */
  snapshot: ActivatedRouteSnapshot;
```

If a router reuses a component and doesn't create a new activated route, you will have two versions of `ActivatedRouteSnapshot` for the same `ActivatedRoute`. Suppose you have the following routing configuration:

```
path: /segment1/:id,
component: AComponent
```

Now you navigate to:

```
/segment1/1
```

You will have the param in the `activatedRoute.snapshot.params.id` as `1`.

Now you navigate to:

```
/segment1/2
```

You will have the param in the `activatedRoute.snapshot.params.id` as `2`.

You can see it by implementing the following:

```
export class AComponent {
  constructor(r: ActivatedRoute) {
    r.url.subscribe((u) => {
      console.log(r.snapshot.params.id);
    });
```

What is directives?

==> Angular 2 categorizes directives into 3 parts:

1. Directives with templates known as **Components**
2. Directives that creates and destroys DOM elements known as **Structural Directives**
3. Directives that manipulate DOM by changing behavior and appearance known as **Attribute Directives**

Attribute Directives

Attribute directives, as the name goes, are applied as attributes to elements. They are used to manipulate the DOM in all kinds of different ways except creating or destroying them. I like to call them DOM-friendly directives.

Directives in this categories can help us achieve one of the following tasks:

- **Apply conditional styles and classes to elements**

```
<p [style.color]="'blue'">Directives are awesome</p>
```

- **Hide and show elements, conditionally (different from creating and destroying elements)**

```
<p [hidden]="shouldHide">Directives are awesome</p>
```

- **Dynamically changing the behavior of a component based on a changing property**

Structural Directives

Structural directives are not DOM-friendly in the sense that they create, destroy, or re-create DOM elements based on certain conditions.

This is a huge difference from what `hidden` attribute directive does. This is because `hidden` retains the DOM element but hides it from the user, whereas structural directives like `*ngIf` destroy the elements.

`*ngFor` and `[ngSwitch]` are also common structural directives and you can relate them to the common programming flow tasks.

Custome directive:

myHidden: Case Study

Our first directive is going to be a case study of the existing Angular 2 `hidden` directive. Let's implement that and it would serve as an eye opener of how these things work internally:

```
// ./app/shared/hidden.directive.ts
import { Directive, ElementRef, Renderer } from '@angular/core';

// Directive decorator
@Directive({ selector: '[myHidden]' })
// Directive class
export class HiddenDirective {
    constructor(el: ElementRef, renderer: Renderer) {
     // Use renderer to render the element with styles
       renderer.setElementStyle(el.nativeElement, 'display', 'none');
    }
}
```

then call directive into html

```
<h1 myHidden>Hidden Welcome</h1>
```

###Dependency Injection:

Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies.

The AngularJS injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested.

###Compilation type in angular:

***Just-in-Time* (JIT)**, which compiles your app in the browser at runtime.

***Ahead-of-Time* (AOT)**, which compiles your app at build time.

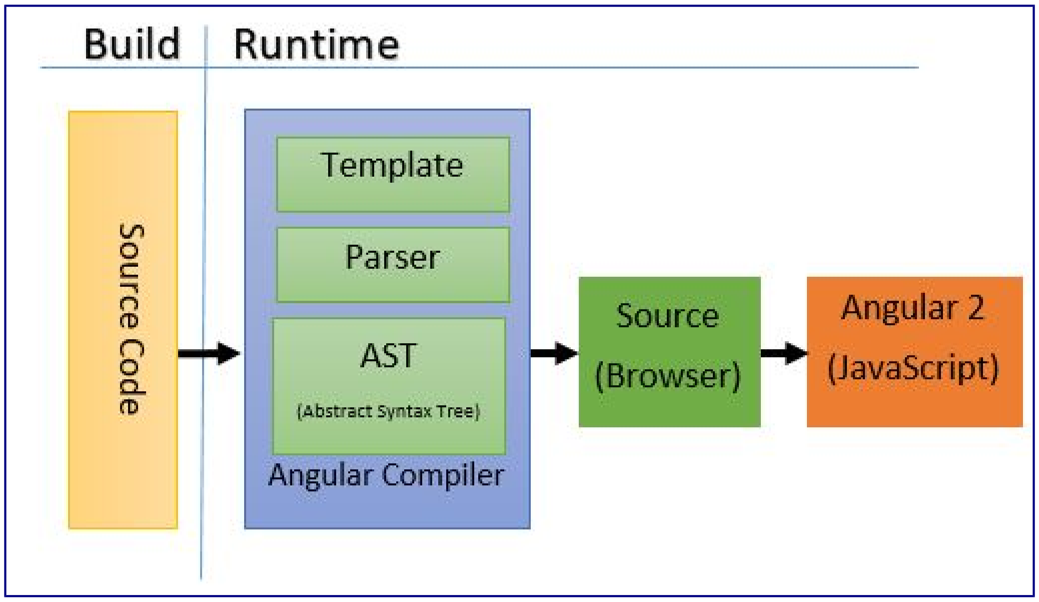
JIT - Compile TypeScript just in time for executing it.

- Compiled in the browser.
- Each file compiled separately.
- No need to build after changing your code and before reloading the browser page.
- Suitable for local development.

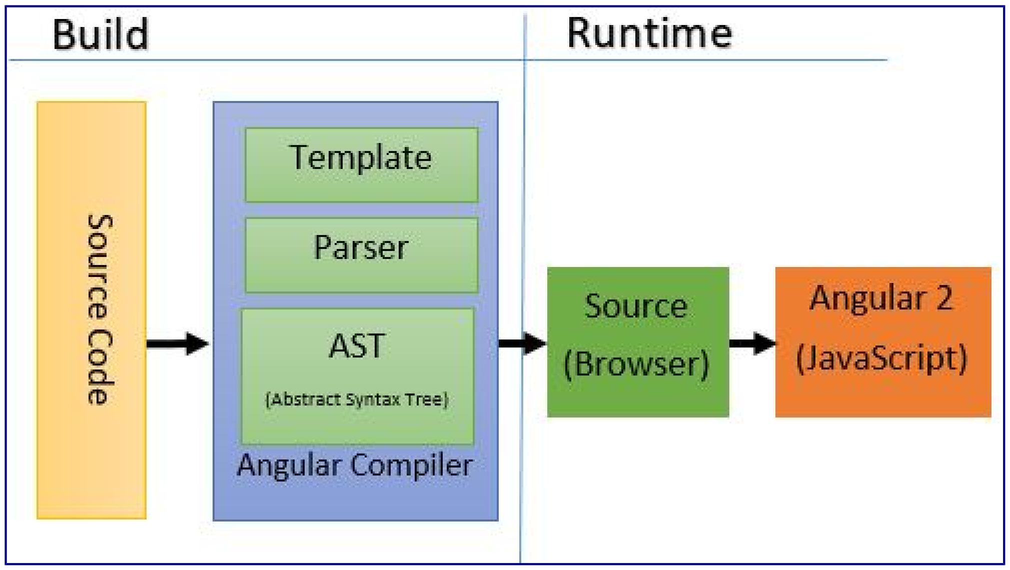
AOT - Compile TypeScript during build phase.

- Compiled by the machine itself, via the command line (Faster).
- All code compiled together, inlining HTML/CSS in the scripts.
- No need to deploy the compiler (Half of Angular size).
- More secure, original source not disclosed.
- Suitable for production builds.

JIT (Just-in-Time Compilation)



AOT (Ahead-of-Time Compilation)


Build
Runtime
Source Code
Template
Parser
AST
(Abstract Syntax Tree)
Angular Compiler
Source (Browser)
Angular 2 (JavaScript)