1. THIS KEYWORD

this keyword in JavaScript ***refers to an object that is executing in the current code***. Like 'window' object executes in the global scope. Similarly, ***every function while executing has a reference to its current execution context***, which can be ***referenced by*** **'this'**.

In most of the cases, the ***value of 'this' depends by how a function is called***. It may be different each time the function is called.

The value of 'this' also depends whether we are working in the strict mode or not.

1.1. In normal/sloppy mode

In ***global execution context***, 'this' refers to the global object as the function call is bound to window object by default.

Inside a function, if the ***value of 'this' is not set by the call***, then it will also ***default to 'window' object***. Eg.,

```
function abc() {
    console.log(this);
}
abc();              // Prints Window object
```

When an object has a function defined as a property to it, its 'this' refers to the object the method is called on. Eg.,

```
var obj = {
    a: 25,
    abc: function() {
        return this.a;
    }
}
console.log(obj.abc());          // Prints - 25
```

The result of the output won't change even if the function is attached to the object afterwards the creation of the object.
Let's say that the object defined above has a nested object as -

```
obj.b = { bcd: function() { console.log(this.a); }, a: 35 };
```

Then, inside the nested object - 'b', 'this' will refer to the object 'b'. So, if we do something like this -

```
console.log(obj.b.bcd());              // Prints - 35
```

Only the most immediate reference to the function call matters.

1.2. In strict mode

Strict mode puts a restriction on value of this, it will be undefined if in global context function is not bound to any object. Whereas in sloppy mode it was set to 'window' object.

Inside a function, if the ***value of 'this' is not set by the call, then it will be 'undefined'***. This happens because the function is called directly and not as a method(eg. window.abc()). Here is an example below -

Eg.,

```
function abc() {
    console.log(this);
}
abc();              // Prints - undefined
window.abc()        // Prints window object
```

The function of an object behaves in same way whether in the strict mode and or not.

1.3. Using call() or apply() methods

As we know now that in strict mode if you call a function staightaway it not bound to any object. So you can bound a function to a particular object by using **'call()'** or **'apply()'** methods.

Eg.,

```
var obj = {a: 12, b: 13};

function sum() {
    return this.a + this.b;
}

sum.call(obj);
sum.apply(obj);
```

The difference in **'call()'** and **'apply()'** functions lies in the way arguments are passed to the function.
In call() method, you ***pass individual parameters*** after passing the object parameter.

In apply() method, you pass the rest of the ***parameters as an array***.

If the ***first argument passed is some other value than object***, then ***attempt is made to convert it to an objec***t. So, if you write code like this -

```
function abc() {
    console.log(this);
}
abc.call(7);              // Prints - Number { 7 }
abc.apply(true);          // Prints -Boolean { true }
```

The number is internally converted to object as - ***new Number(7)***

EXTRA:

You can read an article on this keyword from the link below -
https://codeburst.io/all-about-this-and-new-keywords-in-javascript-38039f71780c

2. CONSTRUCTOR

You can ***create objects*** in JavaScript ***using curly braces { … }*** syntax. But what if you need to create multiple similar objects. You can either write the same syntax for every object or you can use constructor to create objects.

Using { … } syntax to create multiple objects can create certain inconsistencies in code - there can be spelling mistakes, the code can become difficult to maintain, changes to all the objects will be difficult.

To overcome all the above inconsistencies, ***JavaScript provides a function constructor***. The ***constructor provides a blueprint/structure for objects***. You use this same structure to create multiple objects.

Constructor technically are regular functions. There is one convention to constructors -

- The ***first letter of the function is capital***.

Objects can be created by ***calling the constructor function with the 'new' keyword***.

Using a constructor means that -

- all of these objects will be created with the same basic structure.
- we are less likely to accidentally make a mistake than if we were creating a whole bunch of generic objects by hand.

It is important to ***always use the new keyword*** when invoking the constructor.
If ***new is not used***, the constructor may clobber the 'this' which was accidentally passed to it. In most cases that is the global object (window in the browser or global in Node.). Without 'new' function will not work as a constructor.

```
function Student(first, last, age) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
}

var stu1 = new Student("John", "Doe", 50);
var stu2 = new Student("Sally", "Rally", 48);
```

The ***new keyword is the one that is converting the function call into constructor call*** and the following things happen -

1. A brand new empty object gets created
2. The empty object gets linked to prototype property of that function. (We will read about prototype later in this lecture.)
3. The empty object gets bound as ***this*** keyword for execution context of that function call
4. If that function does not return anything then it implicitly returns ***this*** object.

NOTE: *The 'this' referred in the constructor bound to the new object being constructed.*

EXTRA:

You can get a good read from the link below -
https://javascript.info/constructor-new

3. PROTOTYPES

Prototypes is a simple way to share behaviour and data between multiple objects. The prototypes are property of the Object constructor and can be seen from the code below -

```
Object.prototype
```

All the objects created in JavaScript are instances of 'Object'. Therefore, all of them share the 'Object.prototype' method. We can also override these properties, i.e. we can change them.

Why do you need prototypes? - Sometimes you want to add new properties (or methods) to all the existing objects of a given type. This is possible only by adding the new method to the prototype function.

Since, all the objects share the same prototype chain, the changes done to the Object prototype is seen by all the instances.

Eg.,

```
function Student(name, age) {
    this.name = name;
    this.age = age;
}

var stu1 = new Student("John", 50);
var stu2 = new Student("Sally", 48);

Student.prototype.getName = function() {
    return this.name;
}
```

The below statement will work and produce the name of the object like -

```
stu1.getName();.        // Return "John"
```

3.1. _proto_ AKA Dunder Proto

proto points to the prototype object of the constructor function.

When an object is created in JavaScript, JavaScript engine adds a __proto__ property to the newly created object. This _proto_ property is equal to the Object's prototype property.

Let's see an example -

```
function Student(name,age) {
    this.name = name;
    this.age = age;
}

var stu1 = new Student("John", 50);
```

If you do check the `stu1._proto_` and `Student.prototype`, then you will see that both of them are same. Also if you apply strict equal to check if they point at the same location then it will return true.

```
Student.prototype === stu1._proto_              // Return true
```

object.getPrototypeof(---); ALTERNATIVE OF DUNDER PROTO(NOT ENCORAGE TO USE

vehicle.prototype.isPrototypeOf(vehicle1) check wether it is a prototype of vehicle or not

vehicle.hasOwnProperty('propertyname') check if it is its own property or it inherit from prototype

4. OBJECT

4.1. undefined

'undefined' is the value assigned to the variable that has not

5. CLASSES

Classes are introduced in ECMAScript 2015. These are ***special functions that allows one to define prototype-based classes*** with a clean, nice-looking syntax. It also introduces great new features which are useful for object-oriented programming.

An example of creating a class is -

```
class Person {
    constructor(first, last) {
        this.firstName = first;
        this.lastName = last;
    }
}
```

The way you have defined class above is known as class declaration. In order to ***create a new instance of class Person***, we do this -

```
let p1 = new Person("Rakesh", "Kumar");
```

Some points you need to remember -

- You define class members inside the class, such as methods or constructor.
- By default, the body of class is ***executed in strict mode***.
- The ***constructor method is a special method*** for creating and initializing an object.
- You cannot use constructor method more than once, else SyntaxError is thrown.
- Just like constructor function, ***'new' keyword is required to create a new object***.

NOTE: *The type of the class is 'function', i.e. typeof(Person) will print 'function' on the console.*

5.1. Class Expression

A **class expression** is another way to define a class, which is similar to function expression. They can be named and unnamed both, like -

```
let Person = class {};
```
OR
```
let Person = class Person2 {};
```

The name given to the class expression is local to the class's body.

5.2. Hoisting

Class declarations are not hoisted. If you try to use hoisting, it will return 'not defined' error.

5.3. Inheritance

The JavaScript class also supports inheriting like other languages such as Java and C++. To inherit a class you have to use **'extends'** keyword. Eg.,

```
class Vehicle {
    constructor(make, model, color) {
        this.make = make;
        this.model = model;
        this.color = color;
    }
    getName() {
        return this.make + " " + this.model;
    }
}

class Car extends Vehicle{
    getName(){
        return this.make + " " + this.model +" in child class.";
    }
}

let car = new Car("Honda", "Accord", "Purple");
car.getName(); // "Honda Accord in child class."
```

If you want to call the function of the base class. We use 'super' keyword in order to call the base class methods from within the methods of the child class. Eg.,

```
class Car extends Vehicle{
    getName(){
        return super.getName() +"  - called base class function
        from child class.";
    }
}
```

5.4. Getter and Setter

You can also have **getter/setter** to ***get the property value*** or to ***set the property values*** respectively**.** You have to use **'get'** and **'set'** method to create a getter and setter respectively.

Eg.,

```
class Vehicle {
    constructor(model) {
        this.model = model;
    }

    get model() {
        return this._model;
    }

    set model(value) {
        this._model = value;
    }
}

Vehicle v = new Vehicle("dummy");
console.log(v.model); // get is invoked
v.model = "demo";     // set is invoked
```

EXTRA:

Read from the link below to read about classes -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes