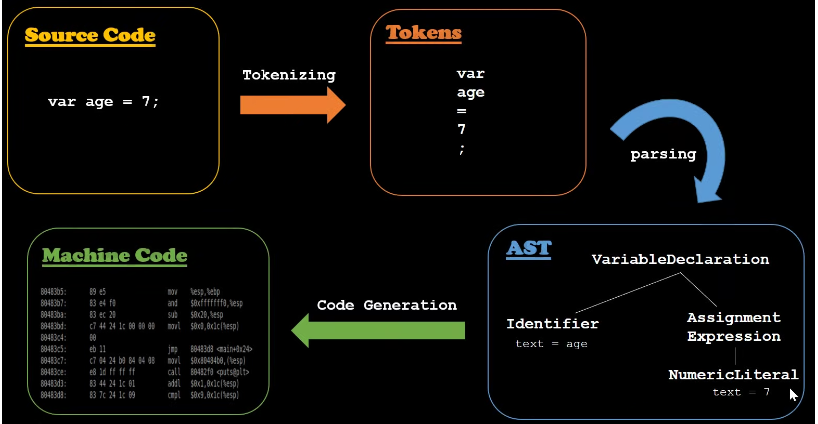
1. **JavaScript Interpreted or Compiled?**
2. An interpreter (مترجم فورى) is a computer program that directly executes instructions written in a programming or scripting language without requiring them previously to have been compiled into a machine language program. It translates one Statement at a time.
3. A compiler is computer software that transforms computer code written in one programming language (the source language, like JavaScript, etc.) into another programming language (the target language, like machine code).

In JavaScript, the source code typically goes through the following phases before it is executed:

**Tokenizing**: Breaking up a source code string into meaningful chunks called, Tokens. For example, the source code var age = 7; can be tokenize as, var, age, =, 7 and, ;.

**Parsing**: Parsing is a methodology to take the array of Tokens as input and turn it into a tree of nested elements understood by the grammar of the programming language. This tree is called Abstract Syntax Tree(AST).

**Code Generation**: In this phase, the AST is used as input, and an executable byte-code is generated that is understood by the environment(or platform) where the executable code will be running. The executable byte-code is then refined/converted even further by the optimizing JIT (Just-In-Time) compiler.



**Does JavaScript really Compiles?** The answer is a loud YES. After the compilation process produces a binary byte code, the JS virtual machine executes it.

**Lexical Environment**: The word Lexical means related to something. Lexical Environment means how and where your code is physically placed.

**Context** سياق: is a set of circumstances (ظروف) or facts that surround (تدور)a particular event, situation, etc.

**Execution Context:** It means the code currently running and everything surrounding it helps run it. There can be lots of Lexical Environment available but, the one Currently running is managed by the Execution Context.

Whenever a JavaScript Code first runs, it creates something **called Global Execution Context(GEC).** Even when you do not have a single line of code in a .js file and load it, you will have the Global Execution Context created.

**Global Execution Context i**s also called the base execution context. It creates two special things for you,

* the creation phase and the execution phase
* A global object called window for the browser.
* A global variable called this.
* The window object and the this variable both are equal in the Global Execution Context.

The this keyword is a pointer which refers to the object it belongs to. In other words, it references the object that is executing the current function. JavaScript this keyword has different values depending on how the code is being executed. When dealing with a regular function, this by default references the window object in browsers and the global object in a nodeJs environment.

   console.log(this)  // Refers to the window object

function sayHi() {

    console.log(this);

}

sayHi();  //Refers to the window object

**this - Inside Object's method:**

// this when used in a method

const test={

    name:"kero",

    age:30,

    sayHello:function () {

        console.log(this)

    }

}

test.sayHello() // {name: 'kero', age: 30, sayHello: ƒ}

const test={

    name:"kero",

    age:30,

    sayHello: () =>{

        console.log(this)

    }

}

test.sayHello() // Refers to window object

/\* Declaring a method with arrow function refers to the window

object and not the current object containing the method\*/

JavaScript strict mode does not allow default binding. So in a strict mode, this is undefined. In the code below, adding a strict mode - "**use strict**" will return undefined

function sayHi() {

  "use strict";

  console.log(this);

}

sayHi();  // undefined

**The bind, call and apply method:**

**Functions are special objects in JavaScript. Every function has a bind, call and apply method, and these methods can be used to assign a value to this in the execution phase of the function.**

const user = {

      firstName: "Code with",

      lastName: "kisha",

      fullName: function(){

          console.log(this.firstName + " " + this.lastName);

      }

  };

 function getAllNames(){

     console.log(this);}

getAllNames(); //window object

**we want this to refer to the object user. To do this we save the getAllNames to a variable and add the method bind to the getAllNames function as seen below and viola!! we have our this referencing the object user.**

 const register = getAllNames.bind(user);

register(); // {firstName: "Code with", lastName: "kisha", fullName: ƒ}

const user = {

      firstName: "Code with",

      lastName: "kisha",

      fullName: function(){

          console.log(this.firstName + " " + this.lastName);

      }

  };

  function getAllNames(){

   /\*   console.log(this); \*/

   this.fullName()

 }

 const register = getAllNames.bind(user);

register(); // Code with kisha

In JavaScript, Scope is the mechanism to determine where the variables exist to use. The variable may exist inside or outside of a function call.

**When NOT to use Arrow functions?**

1. **Constructors:**

Arrow functions do not create their own scope when executed. They are bound to their enclosing scope context, which means unlike regular functions, arrow functions do not have their own this.

const Person = (name) => {

  this.name= name;

  console.log(this);

};

const roger = new Person('Roger');

//Output: TypeError Person is not a constructor

On the other hand, using a regular function will work properly.

const Someone = function(name){

    this.name= name;

    console.log(this);

}

const james = new Someone('James');

//Output: Someone {name: 'James'}

1. **Object Methods:**

When we create an object method in JavaScript with a regular function, it creates its own scope so that this refers to the object that has the method stored as its property. However, arrow functions do not have their own scope so this would refer to the window object. This results in this being a window object and any of its property being undefined.

var obj = {

  i: 10,

  b: () => console.log(this.i, this), // does not create a new scope

  c: function() {

    console.log(this.i, this);

  }

}

obj.b(); // prints undefined, Window {...} (or the global object)

obj.c(); // prints 10, Object {...}

1. **Callback Functions with 'this' :**

The fact that arrow functions do not have their own scope and this also makes them unsuitable for callback functions which includes 'this'. Consider the following code:

const myButton = document.getElementById('example');

/\* when use arrow function \*/

myButton.addEventListener('click', ()=> {

  console.log(this); //refers to window not the button itself

  this.innerHTML = 'Clicked'; //so this won't work

});

/\* when use anonymous function \*/

myButton.addEventListener('click', function() {

  console.log(this); //refers to <button id="example">Clicked</button>

  this.innerHTML = 'Clicked'; //so this refers to button and change the innerHTML

});

**How to Freeze an Object:**

it freezes the Object. It means once frozen, it is not possible to:

* extend or add a property to the object.
* delete a property from the object.
* modify the value of property of the object unless, the value is an unfrozen object.
* const person = {
* name: 'Tapas Adhikary',
* age: 36,
* };
* // person and frozenPerson are the reference
* // to the same object.
* const frozenPerson = Object.freeze(person);
* person["salary"] = 68761182;
* console.log('Person after adding property, Salary', person); // {name: 'Tapas Adhikary', age: 36}
* // or you can also do as,
* frozenPerson["salary"] = 68761182;
* console.log('frozenPerson after adding property, Salary', frozenPerson); // {name: 'Tapas Adhikary', age: 36}
* delete person.name;
* console.log('Person after deleting property, name', person); // {name: 'Tapas Adhikary', age: 36}

**Promises vs. Callbacks:**

1. Promises reduces the amount of nested code
2. Promises allow you to visualize the execution flow easily
3. Promises let you handle all errors at once at the end of the chain.