Synchronous single threaded language.

All the code execution happens in the Execution Context.

Even before 1st line of JS code executes, JS engine creates the Global Execution Context.

Execution Context has 2 components

1. Variable Environment Or Memory Component

All the variables and functions get stored in key value pair

1. Code Component Or Thread of Execution

Code executes line by line.

# Hoisting

In Javascript variables and functions can be invoked even before they are declared in code. This is possible for hoisting

console.log(x); //Undefined

Hello(); // Hello World

var x=5;

function Hello(){

    console.log("Hello World!")

}

console.log(x); //5

console.log(x); //Uncaught ReferenceError: Cannot access 'x' before initialization

Hello();

let x=5;

function Hello(){

    console.log("Hello World!")

}

console.log(x);

## Hoisting var

Before execution of code, All the vars (in Global scope) initialized with undefined.

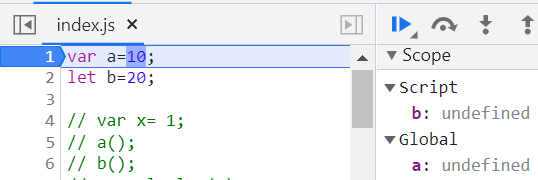
## Hoisting function

Before execution of code, JS engine initializes the function with it’s body as the value and function name as the key

|  |  |
| --- | --- |
| Function Hello(){  Console.log(“Hello World!”)} | Global / Window  Hello: f(){  Console.log(“Hello World!”)} |

## Hoisting let & const

Let & const declarations also get hoisted before code execution, they also get the special value undefined. But they don’t get attached to Global scope, in fact they get attached to Script and these can’t be invoked till some value is assigned. This period between hoisting to actual value assignment is known as Temporal Dead zone.



# Var vs let vs const

console.log(a); //10

console.log(b); //Uncaught ReferenceError: Cannot access 'b' before initialization

var a=10;

let b=20;

console.log(b);//20

console.log(c); //Uncaught ReferenceError: c is not defined

let is stricter than var. You can’t access let before assigning any value.

You can’t re-declare let. It will give syntax error and the app won’t run at all.

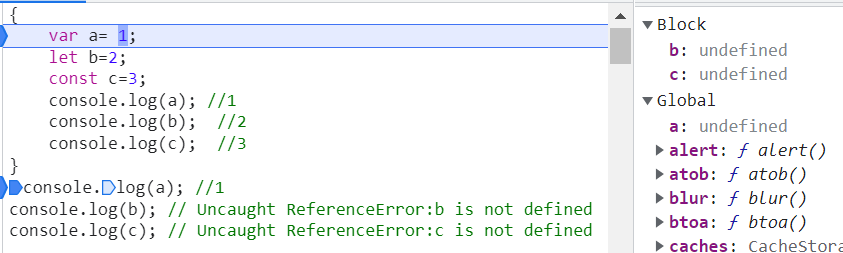
var a =5;

var a=50;

let b=1;

let b=10; //Uncaught SyntaxError: Identifier 'b' has already been declared

var is global scoped. Let & const are block scoped. Var can be accessed outside the defined block. Let and const are available on in the block.



|  |  |  |  |
| --- | --- | --- | --- |
|  | **Defined in function** | **Defined in block** | **Globally defined** |
| **var** | Local | Global | Global |
| **let** | Local | Block | Script |
| **const** | Local | Block | Script |

# Shadowing

Var Identifiers defined in a block (if, while switch etc. not function) shadow the declarations outside the block. Let & const also shadow(within the block) the declarations outside.

var a= 10;

let b=20;

const c= 30;

{

    var a= 1;

    let b=2;

    const c=3;

    console.log(a); //1

    console.log(b);  //2

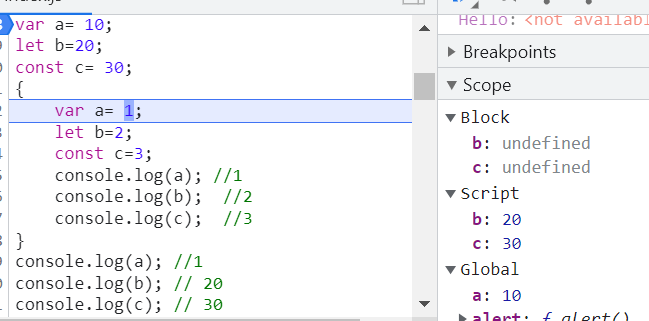
    console.log(c);  //3

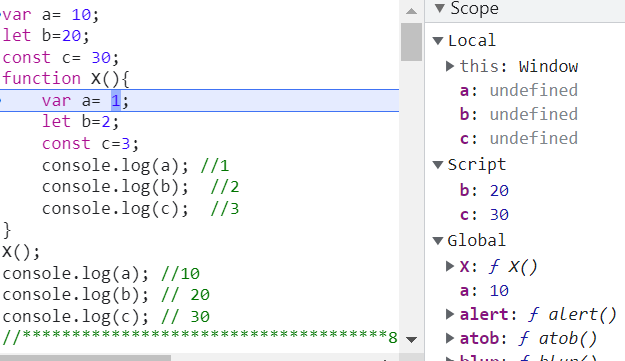
}

console.log(a); //1

console.log(b); // 20

console.log(c); // 30





The Shadowing identifier can’t have more scope than the original declaration. Var & function behave the same in terms of scope & shadowing.

Let defined outside block can’t be shadowed by a var declared in block since var in block has Global scope

|  |  |
| --- | --- |
| var a= 10;  {      var a= 1;      console.log(a); //1  }  console.log(a); //1 | var a= 10;  function X(){      var a= 1;      console.log(a); //1  }  X()  console.log(a); //10 |
| var a= 10;  {      let a= 1;      console.log(a); //1  }  console.log(a); //10 | var a= 10;  function Y(){      let a= 1;      console.log(a); //1  }  Y()  console.log(a); //10 |
| let a= 10;  {      var a= 1; // Uncaught SyntaxError: Identifier 'a' has already been declared      console.log(a);  }  console.log(a); | let a= 10;  function Z(){      var a= 1;      console.log(a); //1  }  Z()  console.log(a); //10 |
| const a= 10;  {      const a= 1;      console.log(a); //1  }  console.log(a); //10 | |
| var a= 10;  {      const a= 1;      console.log(a); //1  }  console.log(a); //10 | const a= 10;  {      var a= 1; // Uncaught SyntaxError: Identifier 'a' has already been declared      console.log(a);  }  console.log(a); |
| let a= 10;  {      const a= 1;      console.log(a); //1  }  console.log(a); //10 | const a= 10;  {      let a= 1;      console.log(a); //1  }  console.log(a); //10 |

# Closure

Function along with it’s Lexical environment.

function A(){

    var x= 5;

    function B(){

        console.log(x);

    }

    return B;  // Clolsure: Function along with it's Lexical scope

}

var c= A();

var x=50;

c(); //5

{

    var x= 5;

    function B(){

        console.log(x);

    }

    x=20;

}

B(); // 20

var x=50;

B(); //50

|  |  |
| --- | --- |
| setTimeout(function(){      console.log("Within setTimeout");  },2000);  console.log("Hello World!");  //Hello World!  //Within setTimeout | for(let i=0; i<5; i++){      setTimeout(function(){          console.log(i);      },2000);  }  console.log("Hello World!");  //Hello World!  // 0 1 2 3 4 |
| for(var i=0; i<5; i++){      setTimeout(function(){          console.log(i);      },2000);  }  console.log("Hello World!");  //Hello World!  // 5 5 5 5 5 | for(var i=0; i<5; i++){      function Close(x){          setTimeout(function(){              console.log(x);          },2000);      }      Close(i);  }  console.log("Hello World!");  //Hello World!  // 0 1 2 3 4 |

Closure helps in Data encapsulation. In the following example the “counter” variable can only be modified by the containing method through the closure incrementCounter()

function UpdateCounter(){

    var counter= 0;

    return function incrementCounter(){

        counter++;

        console.log(counter);

    }

}

var counter1 =UpdateCounter();

counter1(); counter1(); counter1(); // 1  2   3

var counter2 =UpdateCounter(); // WIll have a different instance

counter2(); counter2(); // 1   2

counter1(); // 4

counter2();//3

## Constructor function

function UpdateCounter(){

    var counter= 0; var test="";

    this.incrementCounter = function (){

        counter++;

        console.log(counter);

    }

    this.decrementCounter = function (){

        counter--;

        console.log(counter);

    }

}

var counter1 =new UpdateCounter(); // Constructor function

counter1.incrementCounter(); counter1.incrementCounter(); counter1.decrementCounter(); // 1  2   1

var counter2 =new UpdateCounter(); // Will have a different instance

counter2.decrementCounter(); counter2.incrementCounter(); // -1   0

counter1.incrementCounter(); // 2

counter2.incrementCounter();//1

closures consume more memory since the variables & functions defined in the lexical scope don’t get garbage collected till JS engine makes sure the closure is no longer needed. In the above example, by the modern browsers the variable “test” would be garbage collected once control moves out of the function scope since it’s not used in closure. But “counter” won’t get garbage collected since it’s used in closure. This may effectively lead to data leakage if not handled properly.

# Function Statement vs Expression vs Declaration

A(); //Function Statement or Function Declaration

B(); //Uncaught TypeError: B is not a function

function A(){

    console.log("Function Statement or Function Declaration")

}

var B = function(){

    console.log("Function Expression")

}

B(); //Function Expression

Function statement & expression differ in the way they hoisted.

Function expressions are treated normally like any other variable and hoisted same as var. Function expression will have undefined till the line of assignment is invoked. So a function expression can’t be invoked before it’s assigned.

# First Class Function