object creation in bulk

Scope

MDN

In JavaScript, scope determines the accessibility of variables within a specific part of your code. Understanding scopes is crucial for writing clean, efficient, and bug-free code.

Types of Scopes in JavaScript:

1. **Global Scope:**
   - Variables declared outside of any function or block have global scope.
   - They are accessible from anywhere within the script.
   - **Caution:** Overusing global variables can lead to naming conflicts and make code harder to maintain.
2. **Local Scope (Function Scope):**
   - Variables declared inside a function have local scope.
   - They are only accessible within that function.
   - When the function execution ends, the local variables are destroyed.
3. **Block Scope (Introduced in ES6):**
   - Variables declared with `let` and `const` within a block (e.g., `if` statement, `for` loop) have block scope.
   - They are only accessible within that block.
   - This helps prevent accidental variable declarations and unintended side effects.

Key Points to Remember:

- **Variable Hoisting:**
  - In JavaScript, variable declarations are hoisted to the top of their scope.
  - This means that you can use a variable before it's declared, but its value will be `undefined` until the declaration is reached.
  - However, the actual assignment of a value to the variable happens at the point of declaration.

- **`var` vs. `let` vs. `const`:**
  - **`var`:** Declares a function-scoped or globally scoped variable. Avoid using `var` in modern JavaScript as it can lead to unexpected behavior due to hoisting.
  - **`let`:** Declares a block-scoped variable. It's the preferred way to declare variables in most cases.
  - **`const`:** Declares a block-scoped constant. Its value cannot be reassigned after initialization.

Example:

```
function myFunction() {
  // function scope
  let x = 10;
  const y = 20;

  if (true) {
    // Block scope
    let z = 30;
    console.log(x, y, z); // Output: 10 20 30
  }

  console.log(x, y); // Output: 10 20
  // z is not accessible here
}

// Global scope
let a = 5;

myFunction();

console.log(a); // Output: 5
console.log(x, y, z); // ReferenceError: x, y, z is not defined
```

Best Practices:

- Use `let` and `const` to declare variables whenever possible.
- Limit the scope of variables to the smallest possible block.
- Avoid using global variables unless absolutely necessary.
- Be mindful of hoisting and its potential side effects.

By understanding and applying these concepts, you can write more reliable and maintainable JavaScript code.

IIFE - Immediately invoked function expression

MDN

We keep our code inside a function, in order to prevent pollution of the global scope. We execute it immediately to make sure that our code actually runs.

```
(function () {
  // our code here
})();
```

Very important note: Make sure that the statement above it ends with a semicolon else javascript engine might teat it as a continuation of that statement and throw weird errors.

Student Task

▼ Hoisting with `'var'`

```
javascriptCopy code
function hoistingVar() {
  console.log(x);
  var x = 10;
}

hoistingVar();

//Guess the Output: __________
```

▼ Block Scope with `'var'`

```


function blockScopeVar() {
  if (true) {
    var y = 20;
  }
```

```
  console.log(y);
}

blockScopeVar();

//Guess the Output: __________
```

▼ Re-declaration with `'let'`

```
javascriptCopy code
let z = 30;
{
  let z = 40;
  console.log(z);
}
console.log(z);

//Guess the Output: __________
```

▼ Function Scope with `'const'`

```
javascriptCopy code
function functionScopeConst() {
  if (true) {
    const a = 50;
  }
  // console.log(a); // Uncomment this line to check the result
}

functionScopeConst();

//
```

▼ Global Scope

```
javascriptCopy code
const b = 60;

function globalScope() {
  console.log(b);
}
```

```
globalScope();

//Guess the Output: ___________
```

▼ IIFE (Immediately Invoked Function Expression)

```
javascriptCopy code
const c = 70;
(function () {
  const c = 80;
  console.log(c);
})();
console.log(c);

//Guess the Output: ___________
```

▼ Block Scope and for-loop with `'let'`

```
javascriptCopy code
function blockScopeLoopLet() {
  for (let i = 0; i < 3; i++) {
    setTimeout(function () {
      console.log(i);
    }, 100);
  }
}

blockScopeLoopLet();

//Guess the Output: ___________
```

▼ Block Scope and for-loop with `'var'`

```
javascriptCopy code
function blockScopeLoopVar() {
  for (var j = 0; j < 3; j++) {
    setTimeout(function () {
      console.log(j);
    }, 100);
  }
```

```
}

blockScopeLoopVar();

//Guess the Output: __________
```

Declaration, Declaration with Initialization, Assignment, Reassignment

```
//declare or define a variable called firstName
let firstName;

// initialize or assign value to a variable
firstName = 'John'

// declare & initialize | define a variable and assign a value
let lastName = 'Smith';

// re-assign value to a variable
firstName = 'Will'

// Access or showing or logging or looking up the variable content
console.log(firstName, lastName)
```

What all types of values can be assigned to variables?

MDN- primitive

Primitives (value types)

string

number

boolean

undefined

null

bigInt

symbol

Non Primitives (reference types)

object

array

function

Hoisting

MDN

In the compilation/parsing phase (phase-1), We have all the tokens/identifiers available before the execution phase starts. Of course, the assignment happens in the execution phase, but yes, all the variables are available to us, they exist in the memory, immediately after the parsing phase (phase-1)

- `let`s and `const`s hoist to a block, whereas `var`s hoist to a function.
- In case of `var`, in the compilation/parsing phase, it is initialized to `undefined`
- In case of `let`, in the compilation/parsing phase, a token is created, but it is not initialized at all (TDZ error - uninitialized)

MDN Definition (best one): JavaScript **Hoisting** refers to the process whereby the interpreter appears to move the *declaration* of functions, variables or classes to the top of their "**scope"**, prior to execution of the code.

This means that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local. Of note however, is the fact that the hoisting mechanism only moves the **declaration**. The assignment in case of expressions are left in place.

var VS let VS const

	var	let	const
Scope	function	block	block
re-declaration	allowed ✅	not-allowed❌	not-allowed❌
accesibility before declaration	accessible ✅	not accessible ❌	not accessible ❌

Temporal Dead Zone:

Unlike `var`, `let` and `const` are not initialized with undefined while hoisting. That is why let and const are not accessible till the point in the block where it is initialized with some value. the area in the block where let and const are not accessible know as `Temporal Dead Zone`.

this keyword and object creation

Student Tasks:https://codepen.io/Pavan-Ambulkar/pen/ZEPMQzq?editors=0010 [problem - create animal factory function]

▼ Solution

```
// Animal
//     ├── noOfLegs <number>
//     ├── vegetarian <boolean>
//     └── eat() <function that logs `eating...` >
//     └── greet() <function that logs `Hi, I have <noOfLegs> legs.` >
// Convert this


function animal(noOfLegs, vegetarian) {
  let obj = {};

  obj.noOfLegs = noOfLegs;
  obj.vegetarian = vegetarian;
  obj.eat = function() {
    console.log(`eating...`)
```

```
  }
  obj.greet = function() {
    console.log(`Hi, I have ${this.noOfLegs} legs.`)
  }



  return obj;
}

// example invocation
let a1 = animal(4,true);
// a1.eat() // eating...
// a1.greet() // Hi, I have 4 legs.
```

Student Tasks:https://codepen.io/Pavan-Ambulkar/pen/xxBaZxo?editors=0012 [problem - create animal using constructor function]

▼ Solution

```

```

What is object and Why do we use objects ?

Contain properties (key-value pairs). Value can be of any type including primitives, objects, functions and arrays.

Why Objects ?

#1 **Grouping related variables**

```
// grouping related variables
let account = {
  accountNumber : 12092903490,
  name: 'Vivek',
  type: 'Simple Saving',
  balance: 100000,
```

```
  active: true,
}
```

#2 **Adding & Removing Properties and Methods after creating an object [Dynamic objects]**

```
account.reference = 'Some one';
account['new_property'] = 'Yet some value';
console.log(account);

delete account.reference;
console.log(account);
```

#3 **Passing into a function as an Argument**

```
// can be passes to a function as an argument
function printAccountDetails(obj) {
  console.log('Name: ', obj.name, 'Type: ' ,obj.type, 'Active: ' , obj.act
}

printAccountDetails(account);
```

#4 **Related functions can be a part of the object itself, so wherever we have this object we have access to its functions (methods). In other words, Objects can store functions with their associated data.**

```
let account = {
  name: 'Vivek',
    accountNumber : 12092903490,
  type: 'Preffered Savings',
  balance: 100000,
  active: true,
  printAccountDetails: function () {
    console.log('Name: ', this.name, 'Type: ', this.type, 'Active: ', this
  }
};
```

```
account.printAccountDetails()
```

The `this` keyword :

1. **Global Scope :**
    a. In global scope, `this` refers to global object;

```
console.log(this); // Window
```

b. Even in strict mode. in global scope, this refers to window object;

```
"use strict";
 console.log(this);
```

▼ **Strict Mode**

Strict Mode :

The `"use strict"` flag does a number of things:

- Helps prevent accidentally setting global variables (meaning every variable needs a var declaration)

```
"use strict"
age = 10
console.log(age) // Uncaught ReferenceError: age is not defined
```

- Ensures that function arguments are named uniquely.

```
"use strict"
function printHello(a, b, c, a) {
  console.log(a, b, c, a)
}

printHello(1, 2, 3, 4) // throws an error since duplicate param
eter
```

- Basically Strict mode primarily enforces a stricter set of rules for writing JavaScript code to catch common mistakes and prevent potentially problematic behavior.

Strict mode - JavaScript | MDN

JavaScript's strict mode is a way to opt in to a restricted variant of JavaScript, thereby implicitly opting-out of "sloppy mode". Strict mode isn't just a subset: it intentionally has different semantics from

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode

In my opinion, `"use strict"` wherever possible. It will cause your Javascript to throw errors instead of silently trying to remedy what could be a serious problem!

Please be aware, however, that "use strict" has not yet been fully implemented in all browsers. By all means, take advantage of it, but don't rely heavily on it just yet!

2. **In Functions** :

   a. If a function is called without an object, "this" will refer to the global object (usually "window" in a web browser).

```
function test() {
  console.log(this); // Window
}

test();
```

   b. If a function is called as a method of an object, "this" points to the object itself.

```
const person = {
  name: 'John',
  sayHello: function() {
    console.log(`Hello, ${this.name}!`);
  }
};

person.sayHello(); // This will print "Hello, John!" because "this"
refers to the "person" object.
```

   c. In a function, in strict mode, this is undefined.

```
"use strict"
function test() {
  console.log(this) // undefined
}

test()
```

Some observations (Quick Review)

- this will always point to an object.
- this is a pointer that we use in javascript to represent owner object.
- this is flexible; value of this changes
- this can point to different owner objects through few methods that we are going to learn (`call` , `apply` , `bind`).

Call, Apply & Bind Methods :

All JavaScript functions has access to some very special methods which we can use to control where 'this' should refer. Such methods are `call()` , `bind()` & `apply()` .

CALL

```
// .call & .apply are used to invoke a function & set the value of the
this keyword
// inside of the function

// .bind is used to create a function & set the value of the this keywo
rd
// inside of the newly created function

var person1 = {
    name: "John",
  };

  var person2 = {
    name: "Jill",
  };

  function myName() {
```

```
    console.log(this.name);
  }

  myName.call(person1); // takes in the value of this keyword as the fi
rst argument
// prints John
  myName.call(person2); // Prints Jill
```

In case there are multiple arguments to function

```
var person1 = {
  name: "John",
};

var person2 = {
  name: "Jill",
};

function printDetails(age,city) {
  console.log(this.name + " is " + age + " years old"+" and lives in "+
city);
}

printDetails.call(person1,28,"New york"); // John is 28 years old and l
ives in New york
printDetails.call(person2,32,"Bengaluru"); // Jill is 32 years old and
lives in Bengaluru
```

the above code is similar to

```
 var person1 = {
    name: "John",
    printDetails(age,city) {
          console.log(this.name + " is " + age + " years old"+" and liv
es in "+ city);
        }
  };
```

APPLY

The apply() method is literally the same as call() method. They just both take arguments differently.

example3:

```
  var person1 = {
    name: "John",
  };

  var person2 = {
    name: "Jill",
  };

  function printDetails(age,city) {
      console.log(this.name + " is " + age + " years old"+" and lives i
n "+ city);
    }
  myName.apply(person1, [28, "bangalore"]); // takes in the value of th
is keyword as the first argument; rest of the arguments to the function
can be sent a an array of values
```

BIND

You can bind a particular object as 'this' to a function and use it later. You cannot use call() or apply() later, they run immediately.

example4:

```
  var person1 = {
    name: "John",
  };

  var person2 = {
    name: "Jill",
  };

  function myName(age,city) {
    this.age = age;
    this.city = city;
  }
  let myBindFunc = myName.bind(person1, 28, "bangalore"); // bind gives
you a new function in which the this keyword is preset for us; it retur
ns a function which can be invoked later; this "this" value being point
```

```
ed to person1 will be remembered;

  myBindFunc();
```

As our application grows, we need different ways to create Objects. How to create multiple accounts? for example.

Let's say a bank has to create accounts of 90 Lakh customers.

Is it feasible to keep writing like this ? Is is scalable ?

```
// grouping related variables
let account1 = {
  accountNumber : 12092903490,
  name: 'Vivek',
  type: 'Simple Saving',
  balance: 100000,
  active: true,
}
let account2 = {
  accountNumber : 120923208900,
  name: 'Samuel',
  type: 'Current',
  balance: 300000,
  active: true,
}
let account3 = {
  accountNumber : 1209382692610,
  name: 'Aslam',
  type: 'Current',
  balance: 150000,
  active: true,
}
....
```

The answer is no..

So what could be done ?

You will basically create a process - a system - a function - that would take in some arguments and give you a new account!!

```
function account(accountNumber,name,type,balance,active){
  let accountDetails = {};

  accountDetails.accountNumber = accountNumber;
  accountDetails.name = name;
  accountDetails.type = type;
  accountDetails.balance = balance;
  accountDetails.active = active;

    accountDetails.printAccountDetails = function () {
    console.log('Name: ', this.name, 'Type: ', this.type, 'Active: ', t
his.active, 'Bal: ', this.balance);
  }

  return accountDetails;
}

let vivekAccount = account(1290337843, 'Vivek', 'Savings', 90000, true)

console.log(vivekAccount)
/*
{
  accountNumber: 1290337843,
  name: 'Vivek',
  type: 'Savings',
  balance: 90000,
  active: true
}
*/
```

Another example:

What if you need to draw this exact same drawing in 1,00,000 documents?

You would probably create a system - a process - a stamp maybe?

> **If we need to create similar kind of a thing in quantity - its good to have a system - a structure - a process**

Can you think of more real world examples?

- **Molds** (sancha) for clay toys / utensils

"Creating objects in Bulk Quantity"

vs

"Inheritance"

This the concept of creating or manufacturing an object in bulk quantity. Creation of object is different of inheritance.

If you need to manufacture 1,00,000 iPhone 1st Gen in 2007 - we understand that we need a process to manufacture them... create them... only the serial number of the device would change, rest all properties & methods remains the same...

The all have some common properties & methods

Now when iPhone-2 (3G) was planned for 2008, obviously - first thing is that we need a system/structure/process/blueprint

But wait can we inherit/reuse of the properties from the older version of the phone.

Do you see the difference between creating & inheriting

I Phone Example

Using factory functions :

Factory Function:

A factory function is a function that returns an object when called. It is a way to create and initialize objects by encapsulating the object creation process within a function. Factory functions are useful when you want to create multiple instances of objects with similar properties or when you want to abstract the object creation process.

Problem: https://codepen.io/abduljabbarpeer/pen/KKbNwdv

Solution:

```
// write a factory function iPhone1 to create iPhone objects in bulk quant
// iPhone1 takes in ASIN, color, display, camera
// the object it creates has the following
// properties: ASIN, color, display, camera
// methods:
// dial - console logs "tring.. tring..."
// sendMessage - console logs "Sending message..."
// cameraClick - "Camera clicked"

function iPhoneGen1(ASIN, color, display, camera) {
  let obj = {}

  obj.ASIN = ASIN
  obj.color = color
  obj.display = display
  obj.camera = camera

  obj.dial = function () {
    console.log("tring.. tring...")
  }

  obj.sendMessage = function () {
    console.log("Sending message...")
```

```
  }

  obj.cameraClick = function () {
    console.log("Camera clicked")
  }

  return obj
}

let iphone1 = iPhoneGen1("B09X67JBQV", "Gray", "90mm", "2.0 MP")
iphone1.dial() // "tring.. tring..."
iphone1.sendMessage() // "Sending message..."
iphone1.cameraClick() // "Camera clicked"
```

Using constructor function :

Constructor Function :

A constructor function is a special type of function that is used with the `new` keyword to create instances of objects. Constructor functions are typically used when you want to create multiple objects with the same structure and behavior.

Problem : https://codepen.io/abduljabbarpeer/pen/YzdpPqK

Solution :

```
// write a constructor function IphoneGen1 to create iPhone objects in
bulk quantiy
// IphoneGen1 takes in ASIN, color, display, camera
// the object it creates has the following
// properties: ASIN, color, display, camera
// methods:
// dial - console logs "tring.. tring..."
// sendMessage - console logs "Sending message..."
// cameraClick - "Camera clicked"

function IphoneGen1(ASIN, color, display, camera) {
  this.ASIN = ASIN; //  // the new keyword helps in the initialization
of the object and pointing "this" to newly created object;
  this.color = color;
  this.display = display;
  this.camera = camera;
```

```
  this.dial = function() {
    console.log("tring.. tring...")
  }

  this.sendMessage = function() {
    console.log("Sending message...")
  }

  this.cameraClick = function() {
    console.log("Camera clicked")
  }
}


let iphone1 = new IphoneGen1('B09X67JBQV','Gray','90mm','2.0 MP') // th
e new keyword helps in returning the product
console.log(iphone1)
iphone1.dial(); // "tring.. tring..."
iphone1.sendMessage(); // "Sending message..."
iphone1.cameraClick(); // "Camera clicked"
```

Student Tasks: https://codepen.io/drupalastic/pen/wvEdeMr?editors=0010 [problem - create animal factory function]

▼ Solution

```
function animal(noOfLegs, vegetarian) {
  let obj = {};

  obj.noOfLegs = noOfLegs;
  obj.vegetarian = vegetarian;

  obj.eat = function() {
    console.log('eating...')
  }

  obj.greet = function() {
    console.log(`Hi, I have ${obj.noOfLegs} legs.`)
  }
```

```
  return obj;
}

// example invocation
let a1 = animal(4,true);
console.log(a1)
a1.eat() // eating...
a1.greet() // Hi, I have 4 legs.
```

Student Tasks: https://codepen.io/drupalastic/pen/KKxmqvG?editors=0012 [problem - create animal using constructor function]

▼ Solution

```
function Animal(noOfLegs, vegetarian) {
  this.noOfLegs = noOfLegs;
  this.vegetarian = vegetarian;

  this.eat = function() {
    console.log('eating...')
  }

  this.greet = function() {
    console.log(`Hi, I have ${this.noOfLegs} legs.`)
  }
}

// example invocation
let a1 = new Animal(4,true);
console.log(a1)
a1.eat() // eating...
a1.greet() // Hi, I have 4 legs.
```

Addition Tasks :

Try to construct `vehicles` using all three ways:

```
Vehicle
    ├── brand <string>
    ├── tyres <number>
    ├── doors <number>
```

```
    ├── driver <string>
    ├── accelerate <function that logs `accelerating`>
    └── honking <function that logs `honking`>
```

ES6-class

Using constructor function :

Constructor Function :

A constructor function is a special type of function that is used with the `new` keyword to create instances of objects. Constructor functions are typically used when you want to create multiple objects with the same structure and behavior.

Problem :
https://codepen.io/abduljabbarpeer/pen/YzdpPqK

Solution :

```
// write a constructor function
IphoneGen1 to create iPhone obje
cts in bulk quantiy
// IphoneGen1 takes in ASIN, col
or, display, camera
// the object it creates has the
following
// properties: ASIN, color, disp
lay, camera
// methods:
// dial - console logs "tring..
tring..."
// sendMessage - console logs "S
ending message..."
// cameraClick - "Camera clicke
d"

function IphoneGen1(ASIN, color,
```

Using ES6 Class :

ES6 Class :

ES6 introduced a more class-oriented syntax to JavaScript, known as ES6 Classes. ES6 Classes provide a way to create constructor functions and define methods in a more structured and familiar manner, similar to class-based languages like Java or C++. While under the hood, ES6 Classes are still using constructor functions and prototypes, they offer a cleaner and more intuitive syntax for object-oriented programming.

Problem :
https://codepen.io/abduljabbarpeer/pen/MWZ

Solution :

```
// write a ES6 class IphoneGen1
to create iPhone objects in bulk
quantiy
// IphoneGen1 takes in ASIN, col
or, display, camera
// the object it creates has the
following
// properties: ASIN, color, disp
lay, camera
// methods:
// dial - console logs "tring..
tring..."
// sendMessage - console logs "S
```

```
display, camera) {
  this.ASIN = ASIN; //  // the n
ew keyword helps in the initiali
zation of the object and pointin
g "this" to newly created objec
t;
  this.color = color;
  this.display = display;
  this.camera = camera;

  this.dial = function() {
    console.log("tring.. trin
g...")
  }

  this.sendMessage = function()
{
    console.log("Sending messag
e...")
  }

  this.cameraClick = function()
{
    console.log("Camera clicke
d")
  }
}



let iphone1 = new IphoneGen1('B0
9X67JBQV','Gray','90mm','2.0 M
P') // the new keyword helps in
returning the product
console.log(iphone1)
iphone1.dial(); // "tring.. trin
g..."
iphone1.sendMessage(); // "Sendi
ng message..."
iphone1.cameraClick(); // "Camer
a clicked"
```

```
ending message..."
// cameraClick - "Camera clicke
d"

class IphoneGen1 {
    constructor(ASIN, color, dis
play, camera){
        this.ASIN = ASIN;
        this.color = color;
        this.display = display;
        this.camera = camera;

        this.dial = function() {
          console.log("tring.. t
ring...")
        }

        this.sendMessage = funct
ion() {
          console.log("Sending m
essage...")
        }

        this.cameraClick = funct
ion() {
          console.log("Camera cl
icked")
        }
  }
}



let iphone1 =  new IphoneGen1('A
90KX67JBQV','Gray','90mm','2.0 M
P')
console.log(iphone1)
iphone1.dial(); // "tring.. trin
g..."
iphone1.sendMessage(); // "Sendi
ng message..."
```

```
iphone1.cameraClick(); // "Camer
a clicked"
```

In this ES6 Class example:

1. The `class` keyword is used to define the class `IphoneGen1`.
2. The `constructor` method is used to initialize object properties when a new instance is created.
3. Methods like `dial`, `sendMessage`, `cameraClick` can be defined directly within the class body.