**JavaScript ss**

1. **Data Types in JavaScript:**

* JavaScript has seven primitive data types: **string, number, boolean, undefined, null, symbol**, and bigint. Non-primitive data types include objects like arrays, functions, and objects themselves.

**let** str = "Hello"; // string

**let** num = 42; // number

**let** bool = true; // boolean

**let** undef; // undefined (declared but no value assigned)

**let** n = null; // null (explicitly set to "no value")

**let** symbol = Symbol(); // unique symbol

**let** bigInt = 123n; // bigint for very large integers

**let** obj = { name: "Alice" }; // non-primitive, object

 let str = "Hello"; defines a string variable holding text. Strings store characters or words and are enclosed in quotes.

 let num = 42; defines a number variable. JavaScript uses the same data type for integers and floats.

 let bool = true; defines a boolean variable, which can only be true or false.

 let undef; declares a variable without assigning a value, so it’s undefined.

 let n = null; assigns the value null to represent “no value” or an intentionally empty value.

 let symbol = Symbol(); creates a unique Symbol, a primitive used as a unique identifier.

 let bigInt = 123n; defines a bigint for very large integers, denoted by an n suffix.

 let obj = { name: "Alice" }; defines an object, a non-primitive data structure that holds key-value pairs, here with a property name set to "Alice".

1. **Undefined and Null:**

Undefined means a variable has been declared but not assigned a value, while null is an intentional absence of any object value.

**let** a;

console.**log**(a); // undefined

**let** b = null;

console.**log**(b); // null

 let a; declares a variable a without assigning a value, so it’s undefined.

 let b = null; explicitly assigns null to b, meaning "no value" or an empty object reference.

1. **Var, Let, Const (Scope and Differences):**

var has function or global scope and is hoisted, meaning it can be used before declaration (though its value will be undefined). let and const have block scope (limited to the block in which they are defined), and const cannot be reassigned.

**if** (true) {

**var** x = 10; // function-scoped

**let** y = 20; // block-scoped

**const** z = 30; // block-scoped and cannot be reassigned

}

console.**log**(x); // 10 (var is function-scoped, accessible here)

// console.log(y); // Error: y is not defined (block-scoped)

// console.log(z); // Error: z is not defined (block-scoped)

 var x = 10; is accessible outside the if block because var is function-scoped.

 let y = 20; and const z = 30; are block-scoped and can’t be accessed outside of the if block.

 Attempting to access y or z outside the block will result in an error.

1. **Hoisting:**

Variables declared with var and function declarations are hoisted to the top of their scope, meaning they can be used before they are declared. let and const are hoisted but not initialized until their line of code is reached or you can say for the time being it is temporal deadzone.

console.**log**(a); // undefined (var is hoisted)

**var** a = 10;

// console.log(b); // ReferenceError (let is hoisted but not initialized)

**let** b = 20;

 var a is hoisted, so console.log(a); runs without an error but logs undefined.

 let b is hoisted but not initialized, so accessing b before the declaration causes a ReferenceError.

1. **Scope and Scope Chain:**

 **Scope** refers to the accessibility of variables and functions in different parts of your code. There are mainly two types of scope in JavaScript:

* **Global Scope**: Variables defined outside of any function or block can be accessed from anywhere in the code.
* **Local Scope**: Variables defined within a function or block are only accessible within that function or block.

 **Scope Chain**: This is the hierarchy of scopes that the JavaScript engine uses to resolve variable names. When a variable is not found in the local scope, the engine looks up the chain to the outer scope until it reaches the global scope or finds the variable.

**let** globalVar = "I am global";

**function** outerFunction() {

**let** outerVar = "I am outer";

**function** innerFunction() {

**let** innerVar = "I am inner";

console.**log**(innerVar); // I am inner

console.**log**(outerVar); // I am outer

console.**log**(globalVar); // I am global

}

innerFunction();

// Uncommenting the next line would cause an error

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

}

outerFunction();

console.**log**(globalVar); // I am global

// Uncommenting the next line would cause an error

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

 In this code, globalVar is declared in the global scope, making it accessible from anywhere in the code.

 outerVar is defined inside outerFunction, making it accessible within outerFunction and any inner functions (like innerFunction), but not outside of outerFunction.

 innerVar is defined inside innerFunction, so it is only accessible within that function.

 When innerFunction is called, it logs innerVar, outerVar, and globalVar successfully. However, trying to access innerVar or outerVar outside their respective functions will result in a ReferenceError, demonstrating the concept of scope and the scope chain.

This structure illustrates how JavaScript determines variable accessibility through scope and the chain of scopes it traverses to resolve variables.

1. **Shallow Copy and Deep Copy**

A shallow copy duplicates an object’s top-level properties, but nested objects or arrays are still references to the original. A deep copy creates independent copies of nested objects or arrays.

**let** original = { name: "Alice", address: { city: "Wonderland" } };

**let** shallowCopy = { ...original };

**let** deepCopy = JSON.**parse**(JSON.**stringify**(original));

shallowCopy.address.city = "Nowhere";

console.**log**(original.address.city); // "Nowhere" (affected by shallow copy)

console.**log**(deepCopy.address.city); // "Wonderland" (deep copy unaffected)

* shallowCopy is a shallow copy, so address.city changes also affect original.
* deepCopy is a deep copy using JSON.parse(JSON.stringify()), so changes to deepCopy do not affect original.

1. **Pass by Value and Pass by Reference:**

In JavaScript, primitive types are passed by value, meaning a copy of the value is passed. Objects are passed by reference, meaning a reference to the object in memory is passed.

**let** num = 10;

**function** modifyValue(x) {

x = 20;

}

modifyValue(num);

console.**log**(num); // 10 (unchanged, passed by value)

**let** obj = { value: 10 };

**function** modifyObject(o) {

o.value = 20;

}

modifyObject(obj);

console.**log**(obj.value); // 20 (changed, passed by reference)

 num is a primitive, passed by value, so modifyValue does not change num.

 obj is passed by reference, so changes in modifyObject affect obj.

1. **Map, Filter, and Reduce Functions:**

map creates a new array by applying a function to each element in an array. filter creates a new array with elements that pass a specified test. reduce accumulates values from an array into a single result.

**let** numbers = [1, 2, 3, 4, 5];

**let** doubled = numbers.**map**(n => n \* 2);

console.**log**(doubled); // [2, 4, 6, 8, 10]

**let** even = numbers.**filter**(n => n % 2 === 0);

console.**log**(even); // [2, 4]

**let** sum = numbers.**reduce**((acc, curr) => acc + curr, 0);

console.**log**(sum); // 15

 map creates doubled, where each element is multiplied by 2.

 filter creates even, with only even numbers.

 reduce sums up all elements, resulting in 15.

1. **Event Loop:**

JavaScript has a single-threaded event loop, allowing asynchronous operations like setTimeout, Promises, and event listeners to be handled efficiently. The event loop checks the call stack and callback queue to determine what to execute next.

console.**log**("Start");

**setTimeout**(() => {

console.**log**("Timeout");

}, 0);

console.**log**("End");

* "Start" is logged first.
* setTimeout is added to the callback queue, and "End" is logged next.
* "Timeout" is logged last when the call stack is clear.

1. **Promise.all:**

Promise.all takes an array of Promises and returns a new Promise that resolves when all input Promises have resolved or rejects if any of the Promises reject.

**let** p1 = Promise.resolve(1);

**let** p2 = Promise.resolve(2);

**let** p3 = Promise.resolve(3);

Promise.all([p1, p2, p3]).then(values => {

console.**log**(values); // [1, 2, 3]

});

* Promise.all waits for all promises (p1, p2, and p3) to resolve and then logs [1, 2, 3].

1. **Closure in JavaScript:**

A closure is a function that retains access to its lexical scope, even when the function is executed outside that scope.

**function** createCounter() {

**let** count = 0;

**return** **function**() {

count += 1;

**return** count;

};

}

**let** counter = createCounter();

console.**log**(counter()); // 1

console.**log**(counter()); // 2

 createCounter returns an inner function that accesses count.

 Each call to counter() increases and retains count, demonstrating closure.

1. **This Keyword:**

The this keyword refers to the context in which a function is called. Its value can vary depending on how a function is invoked.

**const** obj = {

name: "Alice",

greet: **function**() {

console.**log**(`Hello, my name is ${**this**.name}`);

}

};

obj.greet(); // Hello, my name is Alice

**const** greetFunc = obj.greet;

greetFunc(); // Hello, my name is undefined (in non-strict mode)

* here, this.name refers to the name property of the obj object. However, when greetFunc is called without the object context, this becomes undefined in non-strict mode.

1. **Coercion in JavaScript:**

Coercion is the automatic or implicit conversion of values from one data type to another, such as converting a string to a number.

console.**log**('5' + 2); // "52" (string concatenation)

console.**log**('5' - 2); // 3 (string is coerced to number)

console.**log**(true + 1); // 2 (true is coerced to 1)

console.**log**(null + 1); // 1 (null is coerced to 0)

console.**log**(undefined + 1); // NaN (undefined cannot be coerced)

* Here, JavaScript performs automatic type conversion, leading to different results based on the operation.

1. **NaN (Not a Number):**

NaN represents a value that is not a legal number. It is returned when a mathematical operation fails.

console.**log**(0 / 0); // NaN

console.**log**(Math.**sqrt**(-1)); // NaN

console.**log**(Number("Hello")); // NaN

console.**log**(NaN === NaN); // false (NaN is not equal to itself)

* In the example, NaN results from invalid mathematical operations, and it's unique in that it does not equal itself.

1. **Strict Mode:**

Strict mode is a way to opt-in to a restricted variant of JavaScript, which helps in catching common coding mistakes and "unsafe" actions.

"use strict";

**function** example() {

x = 3.14; // ReferenceError: x is not defined

}

example();

* In strict mode, variables must be declared with var, let, or const. The code above throws an error because x is not declared.

1. **First-Class Functions:**

Functions in JavaScript are first-class citizens, meaning they can be treated like any other value. They can be assigned to variables, passed as arguments, and returned from other functions.

**function** greet(name) {

**return** `Hello, ${name}`;

}

**const** greetFunction = greet; // Assigning function to a variable

console.**log**(greetFunction("Bob")); // Hello, Bob

**function** executeFunction(func, arg) {

**return** func(arg);

}

console.**log**(executeFunction(greet, "Alice")); // Hello, Alice

* In the example, functions can be stored in variables and passed as arguments.

1. **Anonymous Functions:**

An anonymous function is a function without a name. It is often used as a callback or immediately invoked function expression (IIFE).

**setTimeout**(**function**() {

console.**log**("This is an anonymous function!");

}, 1000);

* In this example, the anonymous function is passed directly to setTimeout, and it executes after one second.

1. **Named Function Expressions:**

A named function expression is a function expression that has a name. It helps with recursion and debugging.

**const** factorial = **function** fact(n) {

**if** (n <= 1) **return** 1;

**return** n \* fact(n - 1);

};

console.**log**(factorial(5)); // 120

* In this example, fact is the name of the function expression, allowing it to be called recursively.

1. **Function Expressions vs. Function Declarations:**

Function declarations are hoisted, while function expressions are not. Declarations can be called before they are defined, but expressions cannot.

console.**log**(declaredFunc()); // "I'm a declaration!"

**function** declaredFunc() {

**return** "I'm a declaration!";

}

// console.log(exprFunc()); // TypeError: exprFunc is not a function

**const** exprFunc = **function**() {

**return** "I'm an expression!";

};

* In this example, declaredFunc can be called before its definition due to hoisting, while exprFunc cannot be called before its definition.

1. **Block Scope and Shadowing:**

Block scope is the area within a set of curly braces, where variables declared with let and const are limited to that block. Shadowing occurs when a variable in a nested scope has the same name as a variable in an outer scope.

**let** outerVar = "I am outside!";

**if** (true) {

**let** innerVar = "I am inside!";

console.**log**(innerVar); // I am inside!

console.**log**(outerVar); // I am outside!

}

console.**log**(innerVar); // ReferenceError: innerVar is not defined

**function** shadowingExample() {

**let** x = "outer";

**function** inner() {

**let** x = "inner"; // shadows outer x

console.**log**(x); // inner

}

inner();

console.**log**(x); // outer

}

shadowingExample();

* In this example, innerVar is block-scoped, and x is shadowed in the inner function.

1. **Parameters vs. Arguments:**

Parameters are the names listed in the function definition, while arguments are the values passed to the function when it is called.

**function** add(a, b) { // a and b are parameters

**return** a + b;

}

console.**log**(add(5, 3)); // 8 (5 and 3 are arguments)

* In this example, a and b are parameters, and 5 and 3 are the corresponding arguments.

1. **Pure and Impure Functions:**

A pure function always produces the same output for the same input and does not have side effects, while an impure function can produce different outputs or modify external state.

// Pure function

**function** square(x) {

**return** x \* x; // always returns the same output for the same input

}

console.**log**(square(4)); // 16

// Impure function

**let** counter = 0;

**function** incrementCounter() {

counter++; // modifies external state

**return** counter;

}

console.**log**(incrementCounter()); // 1

console.**log**(incrementCounter()); // 2

* In this example, square is a pure function, while incrementCounter is impure due to its effect on the external counter variable.