**Call-Stack**

In JavaScript, a **call stack** is a mechanism for managing the function calls in your program. It's a stack data structure (LIFO: Last In, First Out) that tracks function invocations and helps manage execution contexts.

**How the Call Stack Works:**

1. **Push to the Stack**: When a function is called, its execution context is pushed onto the stack.
2. **Execute Function**: The function executes, performing its tasks.
3. **Pop from the Stack**: Once the function completes, its execution context is removed from the stack.
4. **Repeat**: This continues until there are no more functions to execute.

If an error occurs, like a **stack overflow**, it's because the stack exceeds its maximum size (usually due to infinite recursion).

**Common Scenarios Involving the Call Stack:**

1. **Synchronous Execution**: JavaScript executes code in a single-threaded, synchronous manner, which means only one function runs at a time. The call stack ensures this order.
2. **Stack Overflow**: A stack overflow occurs when the call stack exceeds its maximum size. This typically happens in scenarios where functions call themselves (recursion) without a termination condition, or there are excessive nested function calls.

function overflow() {

overflow(); // No base case

}

overflow(); // Error: Maximum call stack size exceeded

**Why Does This Happen?**

* Each function call adds a new frame to the call stack.
* Without termination, the stack keeps growing until it exceeds the available memory.

**Fix:**

**function overflow(count) {**

**If (count =0)**

**return “done”;**

**return overflow(count-1);**

**}**

**overflow(100);**

**Excessive Nested Function Calls**

**Occurs when deeply nested function calls fill the stack.**

**function nestedCalls(n) {**

**if (n === 0) return "Done";**

**return nestedCalls(n - 1);**

**}**

**nestedCalls(100000); // Stack overflow**

**Reduce depth by using iteration instead of recursion**

**function iterativeApproach(n) {**

**while (n > 0) {**

**n--;**

**}**

**return "Done";**

**}**

**iterativeApproach(100000);**

1. **Asynchronous Code**: Asynchronous operations (e.g., setTimeout, fetch) are handled differently. They don’t block the stack; instead, they are moved to the **Event Loop** and executed later.

A screen shot of a computer program

Description automatically generated

The approximate stack size in JavaScript varies depending on the environment and the JavaScript engine being used. Typical estimates are:

1. **Google Chrome (V8 Engine)**: Around 10,000 to 20,000 function calls for standard recursion depth .
2. **Mozilla Firefox**: Similar to Chrome, with stack sizes often accommodating up to 10,000 function calls .
3. **Internet Explorer (Older Versions)**: Significantly smaller, with limits around 3,200 calls .

These values can vary based on the complexity of each function call, the memory available on the system, and the specific version of the browser or Node.js.

**Primitive Types**

Primitive types in JavaScript are the most basic building blocks of data. They are immutable, meaning their values cannot be changed once created. JavaScript has **7 primitive types**:

1. **String**
2. **Number**
3. **BigInt**: Represents integers larger than Number.MAX\_SAFE\_INTEGER. Example: 123456789012345678901n.
4. **Undefined**
5. **Null**
6. **Boolean**
7. **Symbol**: Represents unique identifiers. Example: Symbol('id').

Each primitive type is immutable, meaning any operation on them creates a new value rather than modifying the original value.

In JavaScript, **primitives** are not objects and, therefore, do not inherently have methods or properties. However, JavaScript provides a mechanism called **auto-boxing** to allow primitives to temporarily act as if they are objects with methods.

**1. Auto-boxing**

When you try to access a method or property on a primitive value, JavaScript:

* **Wraps** the primitive in a corresponding wrapper object (e.g., String, Number, Boolean, etc.).
* **Calls** the method or accesses the property on the wrapper object.
* **Discards** the wrapper object after the operation is complete.

For example:

let result = "foo".includes("f");

Here’s what happens under the hood:

1. The primitive "foo" is wrapped in a temporary String object.
2. The String.prototype.includes method is called on this temporary object.
3. The wrapper object is discarded, and the result (true) is returned.

**2. Ephemeral Wrapper Objects**

Since the wrapper object is temporary and not stored anywhere:

* You can't mutate it or attach properties to it.
* Any "mutations" (like setting properties) happen on the temporary object and are immediately lost.

Example:

let str = "hello";

str.foo = 42; // Assigns `foo` to a temporary String object.

console.log(str.foo); // Undefined, because the wrapper is discarded.

What happens here:

1. "hello" is auto-boxed into a temporary String object.
2. A property foo is assigned to this object.
3. The object is discarded immediately after the assignment.
4. Accessing str.foo returns undefined because no such property exists on the primitive itself.

**3. Examples of Wrapper Objects**

* A string is wrapped by a String object.
* A number is wrapped by a Number object.
* A boolean is wrapped by a Boolean object.

For instance:

// Primitive

let num = 42;

// Auto-boxing in action

console.log(num.toFixed(2)); // "42.00"

// Explicit boxing

let numObj = new Number(42);

console.log(numObj.toFixed(2)); // "42.00"

In summary, auto-boxing is a behind-the-scenes mechanism in JavaScript that makes primitives behave like objects for property access and method invocation. It’s important to remember that these wrapper objects are ephemeral and cannot be directly interacted with in a persistent way.

**Example of Value Type Behavior:**

let a = 5; // a holds the value 5

let b = a; // b is a copy of a, so b also holds the value 5

b = 10; // modifying b does not affect a

console.log(a); // Output: 5

console.log(b); // Output: 10

Here, a and b are independent of each other.

Reference Types

**Definition:**

* Reference types store a reference (or address) to the memory location where the actual data resides, not the data itself.
* When you assign a reference type to another, both variables point to the same memory location. Changes made through one variable reflect in the other.

**Example of Reference Type Behavior:**

let arr1 = [1, 2, 3]; // arr1 holds a reference to the array in memory

let arr2 = arr1; // arr2 also points to the same array

arr2.push(4); // modifies the array through arr2

console.log(arr1); // Output: [1, 2, 3, 4] (affected by arr2)

Here, arr1 and arr2 share the same reference, so modifying one affects the other.

**Arrays as Reference Types**

**Why Are Arrays Reference Types?**

1. Arrays are objects in JavaScript.
2. When an array is assigned to a variable, that variable stores a reference to the memory where the array's elements are located, not the elements themselves.

**Strings as Value Types**

**Why Are Strings Not Reference Types?**

* Strings are immutable in JavaScript. When you manipulate a string, a new string is created in memory instead of altering the original.
* They behave like value types because every change creates a distinct copy.

**Primitive Types vs. Reference Types: Key Differences**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Primitive Types (Value Types)** | **Reference Types** |
| **Memory Allocation** | Stores data directly | Stores a reference(address) |
| **Mutability** | Immutable | Mutable (usually) |
| **Assignment** | Creates a new copy | Shares the reference |
| **Examples** | number, boolean, string | array, object, function |

**Object.freeze(obj) makes the object immutable. Any attempts to modify it directly or via reference are ignored.**

**const obj = { key: "original" };**

**Object.freeze(obj);**

**obj.key = "modified"; // Attempt to modify a frozen object**

**console.log(obj.key); // original**

**const ref = obj;**

**ref.key = "attempted change"; // Attempt to modify through reference**

**console.log(ref.key); // original**

**Explanation of the Code with Circular Reference**

**let obj = {};**

**obj.self = obj; // Circular reference**

**try {**

**console.log(JSON.stringify(obj));**

**}**

**catch (err) {**

**console.error("Error:", err.message);**

**}**

* **The JSON.stringify method is used to convert a JavaScript object into a JSON string.**
* **JSON serialization works by traversing the entire object structure.**
* **However, when it encounters a circular reference, it gets stuck in an infinite loop as it tries to serialize self, which points back to obj, creating a never-ending cycle.**

**Explicit and Implicit**

**Implicit Type Conversion — automatically done during code execution by JavaScript engine. It is usually done when some operation is done on operands of different data types. Explicit Type Conversion — done manually by humans.**

**let num = "10"; // Explicit Conversion let explicitResult = Number(num) + 5; // Convert "10" to 10 console.log(explicitResult); // Output: 15**

**// Implicit Conversion let implicitResult = num + 5; // "10" is implicitly converted to "10" (string concatenation) console.log(implicitResult); // Output: '105'**

**Nominal, Structural, and Duck Typing**

**1. Nominal Typing**

**Nominal typing depends on explicit declarations of types, where compatibility between types is determined by their names or explicit definitions rather than their structure.**

**JavaScript does not inherently support Nominal Typing since it lacks traditional static typing. However, tools like TypeScript introduce it.**

**Example in TypeScript (Nominal Typing Simulation):**

**type UserId = { id: string };**

**type OrderId = { id: string };**

**function getUser(user: UserId) {**

**console.log(`Fetching user with ID: ${user.id}`);**

**}**

**// Even though both types have the same structure, TypeScript will enforce that only `UserId` is passed.**

**const user: UserId = { id: "user123" };**

**const order: OrderId = { id: "order123" };**

**getUser(user); // ✅ Works**

**// getUser(order); // ❌ Error in TypeScript: Type 'OrderId' is not assignable to type 'UserId'.**

**2. Structural Typing**

**Structural typing means types are compatible if their structures match, regardless of their names. In JavaScript, this is the default typing behavior.**

**function printName(obj) {**

**console.log(`Name: ${obj.name}`);**

**}**

**const person = { name: "Alice", age: 30 };**

**const dog = { name: "Buddy", breed: "Golden Retriever" };**

**printName(person); // ✅ Works**

**printName(dog); // ✅ Works because the structure matches (both have `name`).**

**3. Duck Typing**

**Duck typing is a subset of structural typing where an object's type is determined by its behavior (methods and properties) rather than its explicit type. If an object has the required methods or properties, it is treated as a specific type.**

**function quack(obj) {**

**if (obj.quack && typeof obj.quack === "function") {**

**obj.quack();**

**} else {**

**console.log("Not a duck!");**

**}**

**}**

**const duck = {**

**quack: () => console.log("Quack!"),**

**};**

**const toyDuck = {**

**quack: () => console.log("Squeak!"),**

**};**

**const cat = {**

**meow: () => console.log("Meow!"),**

**};**

**quack(duck); // Quack!**

**quack(toyDuck); // Squeak!**

**quack(cat); // Not a duck!**

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |