1. Stack & Heap

JavaScript engines have two places to store data: Stack & Heap.

Stack	Heap
Primitive data types and references	Objects and functions
Size is known at compile time	Size is known at run time
Fixed memory allocated	No limit for object memory

Memory Management in JavaScript

Memory Life Cycle: Irrespective of programming language, the memory life cycle follows the following stages:

1. **Allocates the memory we need:** JavaScript allocates memory to the object created.
2. **Use the allocated memory.**
3. **Release the memory when not in use:** Once the allocated memory is released, it is used for other purposes. It is handled by a JavaScript engine.

Note: The second stage is the same for all the languages. However, the first and last stage is implicit in high-level languages like JavaScript.

JavaScript engines have two places to store data:

- **Stack:** It is a data structure used to store static data. Static data refers to data whose size is known by the engine during compile time. In JavaScript, static data includes primitive values like strings, numbers, boolean, null, and undefined. References that point to objects and functions are also included. A fixed amount of memory is allocated for static data. This process is known as **static memory allocation.**
- **Heap:** It is used to store objects and functions in JavaScript. The engine doesn't allocate a fixed amount of memory. Instead, it allocates more space as required.

Overview:

Stack	Heap
Primitive data types and references	Objects and functions
Size is known at compile time	Size is known at run time
Fixed memory allocated	No limit for object memory

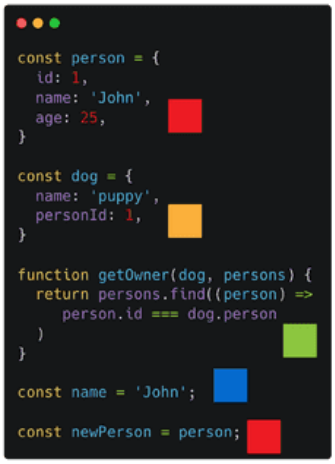
Garbage Collection: Garbage collectors are used in releasing memory. Once the engine recognizes that a variable, object, or function is not needed anymore, it releases the memory it occupied. The main issue here is that it is very difficult to predict accurately whether a particular variable, object, or function is needed anymore or not. Some algorithms help to find the moment when they become obsolete with great precision.

Reference-counting garbage collection: It frees the memory allocated to the objects that have no references pointing to them. However, the problem with this algorithm is that it doesn't understand cyclic references.

```
1. Stack
/*
    Memory Life Cycle: Allocate => Use => Release
    - Stack: + Static memory allocation - Cấp phát bộ nhớ tĩnh
    - stores primitive values (strings, numbers, booleans, undefined, and null)
*/

const male = true
const name = 'John Doe'
const age = 24
const adult = true

// All the values get stored in the stack since they all contain primitive values.
/*
    Stack:
    male = true
    name = 'John Doe'
    age = 24
    adult = true
*/

/*
    - Static data is data where the engine knows the size at compile time.
    - This includes primitive values (strings, numbers, booleans, undefined, and null)
    and references, which point to objects and functions.
    - Since the engine knows that the size won't change, it will allocate a fixed amount of memory for each value.
    - The process of allocating memory right before execution is known as static memory allocation.
    - Because the engine allocates a fixed amount of memory for these values, there is a limit to how large primitive values can be.
    - The limits of these values and the entire stack vary depending on the browser.
*/
```

```
2. Heap
/*
    Memory Life Cycle: Allocate => Use => Release
    - Heap: + Dynamic memory allocation - Cấp phát bộ nhớ động
    - stores objects and functions
*/

const person = {
    name: 'John',
    age: 24,
};

const hobbies = ['hiking', 'reading'];

function foo() {
    const a = 1;
    console.log('Function stored in heap!')
}

/*
    Stack:
    - person: reference to "person" object
    - hobbies: reference to "hobbies" array
    - foo: reference to "foo" function

    Heap:
    - {name: 'John', age: 24}
    - ['hiking', 'reading']
    - foo() {
        console.log('Function stored in heap!')
    }
*/

// stack
const a = 1;
const b = 1;

// heap
const c = [1];
const d = [1];

console.log(a === b) // true --- stack
console.log(c === d) // false --- heap
```

Explanation: In the above example object 'employee' is created in the heap and a reference to it in the stack.



```
const person = {
  id: 1,
  name: 'John',
  age: 25,
}

const dog = {
  name: 'puppy',
  personId: 1,
}

function getOwner(dog, persons) {
  return persons.find((person) =>
    person.id === dog.person
  )
}

const name = 'John';

const newPerson = person;
```