Memory management

Low-level languages like C, have manual memory management primitives such as `malloc()` and `free()`. In contrast, JavaScript automatically allocates memory when objects are created and frees it when they are not used anymore (garbage collection). This automaticity is a potential source of confusion: it can give developers the false impression that they don't need to worry about memory management.

Memory life cycle

Regardless of the programming language, the memory life cycle is pretty much always the same:

1. Allocate the memory you need
2. Use the allocated memory (read, write)
3. Release the allocated memory when it is not needed anymore

The second part is explicit in all languages. The first and last parts are explicit in low-level languages but are mostly implicit in high-level languages like JavaScript.

Allocation in JavaScript

Value initialization

In order to not bother the programmer with allocations, JavaScript will automatically allocate memory when values are initially declared.

```
const n = 123; // allocates memory for a number
const s = "azerty"; // allocates memory for a string

const o = {
  a: 1,
  b: null,
}; // allocates memory for an object and contained values

// (like object) allocates memory for the array and
// contained values
const a = [1, null, "abra"];

function f(a) {
  return a + 2;
} // allocates a function (which is a callable object)
```

```
// function expressions also allocate an object
someElement.addEventListener(
  "click",
  () => {
    someElement.style.backgroundColor = "blue";
  },
  false,
);
```

Allocation via function calls

Some function calls result in object allocation.

```
const d = new Date(); // allocates a Date object
const e = document.createElement("div"); // allocates a DOM element
```

Some methods allocate new values or objects:

```
const s = "azerty";
const s2 = s.substr(0, 3); // s2 is a new string
// Since strings are immutable values,
// JavaScript may decide to not allocate memory,
// but just store the [0, 3] range.

const a = ["ouais ouais", "nan nan"];
const a2 = ["generation", "nan nan"];
const a3 = a.concat(a2);
// new array with 4 elements being
// the concatenation of a and a2 elements.
```

Using values

Using values basically means reading and writing in allocated memory. This can be done by reading or writing the value of a variable or an object property or even passing an argument to a function.

Release when the memory is not needed anymore

The majority of memory management issues occur at this phase. The most difficult aspect of this stage is determining when the allocated memory is no longer needed.

Low-level languages require the developer to manually determine at which point in the program the allocated memory is no longer needed and to release it.

Some high-level languages, such as JavaScript, utilize a form of automatic memory management known as garbage collection (GC). The purpose of a garbage collector is to monitor memory allocation and determine when a block of allocated memory is no longer needed and reclaim it. This is automatic process.

Garbage collection

As stated above, the general problem of automatically finding whether some memory "is not needed anymore" is undecidable. As a consequence, garbage collectors implement a restriction of a solution to the general problem.

References

The main concept that garbage collection algorithms rely on is the concept of *reference*. Within the context of memory management, an object is said to reference another object if the former has access to the latter (either implicitly or explicitly). For instance, a JavaScript object has a reference to its prototype (implicit reference) and to its properties values (explicit reference).

In this context, the notion of an "object" is extended to something broader than regular JavaScript objects and also contain function scopes (or the global lexical scope).

Reference-counting garbage collection

Note: no modern browser uses reference-counting for garbage collection anymore.

Reference-counting garbage collection is a type of automatic memory management technique used in some programming languages, including JavaScript.

In this approach, every object in memory has a reference count, which represents the number of references to that object in the program. When an object is created, its reference count is set to 1. Every time the object is assigned to a variable or used as a parameter in a function call, its reference count is incremented. Conversely, every time a reference to the object is removed (for example, when a variable goes out of scope), its reference count is decremented.

Once an object's reference count reaches 0, it means that no other part of the program is referencing that object. At this point, the garbage collector can safely remove the object from memory to free up space.

```
let obj1 = { name: "John" }; // reference count for obj1 is 1
let obj2 = obj1; // reference count for obj1 and obj2 is 2
obj1 = null; // reference count for obj2 is 1
obj2 = null; // reference count for obj2 is 0, so the object can be garbage collected
```

In this example, we create two objects `(obj1 and obj2)` and set `obj2` equal to `obj1`. This means that both `obj1` and `obj2` reference the same object in memory, so the reference count for that object is 2. We then set `obj1` to `null`, which decrements the reference count for the object to 1. Finally, we set `obj2` to `null`, which decrements the reference count to 0 and allows the garbage collector to remove the object from memory.

Reference-counting has some advantages, such as immediate reclamation of memory when an object's reference count drops to zero, but it also has some limitations. For example, it can't detect reference cycles, where objects reference each other in a circular manner, causing a memory leak. That's why modern JavaScript engines use more sophisticated garbage collection techniques, such as mark-and-sweep and generational garbage collection. Circular references are a common cause of memory leaks.

Mark-and-sweep algorithm

The Mark-and-sweep algorithm is a garbage collection technique used by many modern programming languages, including JavaScript. It works by identifying and collecting objects that are no longer being used by the program, freeing up memory for new objects to be created.
Here's how the Mark-and-sweep algorithm works:

1. Mark Phase: The first phase of the algorithm is the mark phase. The garbage collector starts at the root objects, which are usually global variables and objects referenced from the execution stack. The algorithm then recursively traverses all reachable objects, marking them as live or reachable. Objects that are not reachable are considered garbage.

2. Sweep Phase: Once all reachable objects are marked, the algorithm moves on to the sweep phase. During this phase, the garbage collector looks through the memory and frees up any unmarked objects. This frees up the memory used by the unmarked objects for use by the program.

3. Fragmentation Phase: Finally, there is the fragmentation phase, which is optional in some implementations. This phase compacts the memory and eliminates any fragmentation that may have occurred as a result of the sweep phase.

Here's an example of how the Mark-and-sweep algorithm works:

```
// Create some objects
let obj1 = { foo: 'bar' };
let obj2 = { baz: 'qux' };
let obj3 = { quux: 'corge' };

// Assign references to the objects
let arr = [obj1, obj2];
let obj4 = { arr: arr, obj: obj3 };

// Remove references to the objects
obj1 = null;
obj2 = null;
obj3 = null;
arr = null;
obj4 = null;

// Run the garbage collector
// Mark phase will mark all objects in the array and the object that references it
// The sweep phase will then free up the memory used by the unmarked objects
```

In this example, we create four objects and assign references to them. We then remove all references to the objects, making them garbage. Finally, we run the garbage collector, which will mark all objects that are reachable and free up any memory used by unmarked objects.

The Mark-and-sweep algorithm is a powerful technique for managing memory in programming languages. It allows programs to allocate and deallocate memory dynamically without worrying about memory leaks or other issues. While it can be slower than other garbage collection techniques, it is often more reliable and easier to use.

Configuring an engine's memory model

JavaScript engines typically offer flags that expose the memory model. For example, Node.js offers additional options and tools that expose the underlying V8 mechanisms for configuring and debugging memory issues. This configuration may not be available in browsers, and even less so for web pages (via HTTP headers, etc.).

The max amount of available heap memory can be increased with a flag:

```
node --max-old-space-size=6000 index.js
```

We can also expose the garbage collector for debugging memory issues using a flag and the Chrome Debugger:

```
node --expose-gc --inspect index.js
```

Data structures aiding memory management

Although JavaScript does not directly expose the garbage collector API, the language offers several data structures that indirectly observe garbage collection and can be used to manage memory usage.

WeakMaps and WeakSets

WeakMaps and WeakSets are special types of objects in JavaScript that allow you to associate data with objects without preventing the garbage collector from reclaiming memory when the object is no longer needed.

A WeakMap is a collection of key-value pairs where the keys must be objects and the values can be any type of value. The WeakMap object holds weak references to the keys, which means that if the key object has no other references, it can be garbage collected even if it's still in the WeakMap.

Here's an example of how to use WeakMap:

```
let myWeakMap = new WeakMap();
let obj1 = {};
let obj2 = {};

myWeakMap.set(obj1, "hello");
myWeakMap.set(obj2, "world");

console.log(myWeakMap.get(obj1)); // "hello"
console.log(myWeakMap.get(obj2)); // "world"

obj1 = null; // The key object is no longer referenced anywhere
console.log(myWeakMap.get(obj1)); // undefined
```

In this example, we create a WeakMap object `myWeakMap` and add two key-value pairs to it using the `set()` method. We then retrieve the values associated with the keys using the `get()` method. Finally, we set `obj1` to `null`, which means it's no longer referenced anywhere, and when we try to get the value associated with it, it returns undefined.

A WeakSet is a collection of objects where each object can only appear once. Like a WeakMap, a WeakSet holds weak references to the objects it contains. If an object has no other references, it can be garbage collected even if it's still in the WeakSet.

Here's an example of how to use WeakSet:

```
let myWeakSet = new WeakSet();
let obj1 = {};
let obj2 = {};

myWeakSet.add(obj1);
myWeakSet.add(obj2);

console.log(myWeakSet.has(obj1)); // true
console.log(myWeakSet.has(obj2)); // true

obj1 = null; // The object is no longer referenced anywhere
console.log(myWeakSet.has(obj1)); // false
```

In this example, we create a WeakSet object `myWeakSet` and add two objects to it using the `add()` method. We then check if the WeakSet contains each object using the `has()` method. Finally, we set `obj1` to `null`, which means it's no longer referenced anywhere, and when we check if the WeakSet contains it, it returns false.

WeakMaps and WeakSets are useful when you need to associate data with an object but don't want to prevent the object from being garbage collected. They can be used in situations where you need to track some data associated with an object, but the object itself can be destroyed at any time.

WeakRefs and FinalizationRegistry

WeakRefs and FinalizationRegistry are features introduced in ECMAScript 2021, which is the latest version of the JavaScript language. They provide additional tools for managing memory in JavaScript, especially in situations where you need to keep track of objects that may be garbage collected.

A WeakRef is a weak reference to an object that does not prevent that object from being garbage collected. This is different from a normal reference, which would keep the object alive as long as the reference exists. A WeakRef allows you to track an object without affecting its lifetime.

Here's an example of how to use WeakRef:

```
let obj = { foo: 'bar' };
let weakRef = new WeakRef(obj);

console.log(weakRef.deref()); // { foo: 'bar' }

obj = null; // The object is no longer referenced anywhere
console.log(weakRef.deref()); // null
```

In this example, we create an object `obj` and a WeakRef `weakRef` to it. We then use the `deref()` method to get the object associated with the WeakRef. Finally, we set obj to `null`, which means it's no longer referenced anywhere, and when we try to get the object associated with the WeakRef again, it returns `null`.

A FinalizationRegistry is a registry of functions that will be called when an object is garbage collected. The registry allows you to associate cleanup functions with objects, which can be useful for freeing resources or cleaning up after an object.

Here's an example of how to use FinalizationRegistry:

```
let registry = new FinalizationRegistry((value) => {
  console.log(`Cleaning up ${value}`);
});
let obj = { foo: 'bar' };
registry.register(obj, 'myValue');
console.log(registry.has(obj)); // true
obj = null; // The object is no longer referenced anywhere
console.log(registry.has(obj)); // false

// The cleanup function will be called when the object is garbage collected
```

In this example, we create a FinalizationRegistry object `registry` with a cleanup function that logs a message. We then create an object `obj` and register it with the registry using the `register()` method, along with a value `'myValue'`. We check if the registry has the object using the `has()` method, and then set `obj` to `null`. Finally, we see that the registry no longer has the object and expect the cleanup function to be called when the object is garbage collected.

WeakRefs and FinalizationRegistry are useful for managing memory in situations where you need to keep track of objects that may be garbage collected. They can be used to track resources associated with objects and ensure that those resources are properly freed when the object is no longer needed.