Are singletons like `null` pre-allocated in JavaScript memory?

✔️ Yes. According to the PDF you shared, values like `true`, `false`, `null`, and `undefined` are pre-allocated in the Heap when the JavaScript engine starts.

From Part 9 of the PDF:

> **Heap Address 0x0001: true**
> **Heap Address 0x0002: false**
> **Heap Address 0x0003: null**
> **Heap Address 0x0004: undefined**

And then all variables that use these values simply point to these same locations.

Example from PDF:

> **let x = true → points to 0x0001**
> **let y = true → also points to 0x0001**

🔍 Meaning for `null`

So yes:

✔️ `null` is a singleton

✔️ It is pre-allocated in the heap

✔️ Every `null` in your code simply points to that one memory location

✔️ No new memory allocations for every `null`

This is exactly the same behavior as:

- `undefined`
- `true`
- `false`

🧠 Why engines do this (from the PDF logic)

1. **These values are used extremely often** → no point allocating new memory each time.
2. **They have only one possible value** → reusing is safe.
3. **It reduces heap usage and speeds up assignments.**

⚠️ Important note (real V8 nuance)

The PDF is conceptually accurate for learning:
JS engines treat these values as singletons.

Internally, some engines represent them as **tagged immediate values**, not full heap objects — but conceptually the behavior is the same:

✔️ One instance
✔️ No new allocations
✔️ Shared everywhere

The PDF's explanation is correct for your learning level.

Great question — and now that you've read the PDF, you're thinking at the *right* low-level memory level.
Let's break it down **exactly how `Boolean(null)` becomes `false` inside the JS engine**.

✅ Step 1 — `NullValue = null`

When you write:

let NullValue = null

Memory-wise (according to your PDF):

- `null` is a **pre-allocated singleton** at a fixed heap address (say 0x0003).
- `NullValue` just stores that pointer.

No new memory is allocated.

✅ Step 2 — What happens when you do:

Boolean(NullValue)

This does **not** look at memory to check "is there false stored there?"

It uses a **built-in conversion table** defined by the ECMAScript spec.

JavaScript has a specific mechanism for converting values to boolean:

⚡ ToBoolean Conversion (ECMAScript §7.1.2)

JavaScript doesn't inspect memory content —
it *checks the type of the value*, then applies the rule for that type.

The conversion rules are fixed and hardcoded:

Value	**ToBoolean result**
`undefined`	`false`
`null`	`false`
`0`, `-0`, `NaN`	`false`
`""` (empty string)	`false`
`false`	`false`
Everything else	`true`

So, the engine checks:

What type is this value? → Null

Rule for null? → Always `false`

It does *not* check the memory address.
It does *not* check the contents of heap.
It does *not* compare values.

🔍 Internally, how does V8 check this?

Let's simplify V8's process:

1️⃣ The engine sees you called:

Boolean(NullValue)

2️⃣ It reads `NullValue` → gets pointer/tag that means "null"

In V8, `null` is represented by a special **tagged immediate value** (not a normal heap pointer).
Example: something like:

0x00...05 (a specific pattern meaning "null")

3️⃣ V8 checks:

> “Is this value’s tag == NULL_TAG?”

If yes → return **false** immediately.

🧠 Key point

The engine never checks “what is stored in that heap address”

Instead, it checks the **TAG / TYPE** of the value.

`null` has its own unique tag.
That tag → maps to `false`.

🔥 Final Answer

`Boolean(null)` returns `false` because:

1. **null has a unique internal tag** (like a code)
2. The ToBoolean algorithm has a hardcoded rule:
   null → false
3. The engine simply reads the tag and applies the rule.

It **does NOT** check the memory content or compare values.

If you want, I can also explain **how V8 represents primitives using NaN-boxing / tagged pointers**, which shows exactly how `null` is identified using bits.

You're **very** close — but the internal logic is slightly different.

Let me correct it cleanly:

❌ Your assumption

> “Boolean({}) checks if heap reference is given or not, then true.”

This sounds logical, but it's not how JavaScript actually works.

✅ Correct Explanation

✔️ `{}` (an empty object) is ALWAYS truthy

✔️ NOT because the engine checks if heap reference exists

✔️ BUT because of the ToBoolean rules for objects

🔥 Important: In JavaScript, ALL objects are truthy — regardless of contents.

Even this is truthy:

```
Boolean(new Boolean(false))  // true
Boolean([])                  // true
Boolean({})                  // true
Boolean(function(){})        // true
```

Reason:

👉 Objects → always true

The built-in ToBoolean algorithm has a fixed rule:

- **If the value is of type "Object" → return true**

It does **not** check:

- whether heap memory exists
- whether it's empty
- whether properties exist
- whether reference is valid
- whether it's a new object or reused

Nothing like that.

🧠 V8's internal process (simplified)

When you do:

Boolean({})

Steps:

1️⃣ Engine sees an object literal `{}`

It allocates:

- A new heap object at address, say `0x100050`
- The variable stores this pointer

2️⃣ Calling `Boolean(value)` triggers the ToBoolean algorithm

V8 internally checks:

Is value an object type?

This is done by checking the *tag bits* of the pointer.

3️⃣ If it's an object type → Return true immediately

No heap lookup, no property check, no content examination.

🧩 Why objects are always truthy?

Because objects are **complex references**, and JavaScript chooses not to treat them as falsey to avoid ambiguity.

If JS tried to check object emptiness for truthiness:

```
if ({}) {}   // what should this mean?
```

This would become unpredictable.

So JS takes the simplest consistent rule:

> All objects → truthy.

✔️ Final Clear Summary

`Boolean({})` returns true because:

- The value is of **type Object**
- ToBoolean rules say: **Objects** → **true**
- Engine does NOT check:
    - heap reference validity
    - whether object is empty

    - whether object has any keys

It's purely type-based, not memory-based.
