https://www.notion.so/Lecture-03-V8-Engine-Memory-Allocation-27a3a78e0e228017963adec0533f0d78

Complete In-Depth Explanation of Memory Management in JavaScript

I'll explain this entire lecture comprehensively, covering every important concept without skipping anything.

Part 1: Foundation - How Programs Work

Understanding Basic Computer Architecture

When you have applications on your phone (PUBG, Instagram, WhatsApp), they're stored somewhere. Let me break this down:

1. Hard Disk/SSD (Permanent Storage)

- Your application code is stored here
- This is **permanent storage** - even when you switch off your device, data remains
- Example: Instagram's code lives here
- Speed: Relatively SLOW

2. RAM (Random Access Memory - Temporary Storage)

- When you want to RUN an application, the code is first loaded from Hard Disk → RAM
- RAM is **temporary storage** - when you close an app or switch off device, data is cleared
- Speed: Much FASTER than Hard Disk
- Why use RAM? Because transferring data from Hard Disk directly to CPU would be extremely slow

3. CPU (Central Processing Unit)

- The CPU executes the instructions from your code
- It reads instructions from RAM and performs operations

Flow Summary:

Hard Disk → RAM → CPU
(Code stored) (Code loaded) (Code executed)

Part 2: Understanding Memory Addressing

The Core Problem

Let's say you have:

a = 10
b = 20

Both need to be stored in RAM. But here's the problem:

Scenario: Later you write:

b = 25 // Change b from 20 to 25

How does the computer know WHICH 20 to change? If there are multiple 20s in memory, how do we identify the correct one?

Solution: Memory Addressing

Think of RAM as having **blocks**, each of specific size (1 byte = 8 bits).

Example: 16-byte RAM

Let's divide this 16-byte RAM into blocks:

[Block 0][Block 1][Block 2][Block 3]...[Block 15]

Each block = 1 byte

We give each block a **unique address**:

- Block 0 → Address: 0
- Block 1 → Address: 1
- Block 2 → Address: 2
- And so on...

This is called **Byte Addressable** - every byte gets an address.

How Variables Map to Addresses

```
a = 10  // Let's say stored at address 2
b = 20  // Let's say stored at address 8
```

Behind the scenes, your code is transformed:

```
// What you write:
a = 10

// What actually happens:
address_0010 = 10  // 'a' is replaced with its memory address
```

The Mapping:

- Variable a → Address 2 → Value 10
- Variable b → Address 8 → Value 20

When you update:

```
b = 25
```

The computer:

1. Looks up where b is stored (Address 8)
2. Goes to Address 8
3. Changes 20 → 25

No confusion! Because each variable has a unique address.

Part 3: Bit Requirements for Addresses

Address Size Calculation

For 16-byte RAM:

- Total bytes = 16
- Need 16 unique addresses (one per byte)
- To represent 16 different values, you need: **4 bits**
    - Because 2^4 = 16

Address Representation:

Address 0: 0000
Address 1: 0001
Address 2: 0010
Address 3: 0011
...
Address 15: 1111

For 32-byte RAM:

- Need 32 addresses
- 2^5 = 32, so need **5 bits**

Real-World Systems

32-bit Operating System:

- Address size = 32 bits = 4 bytes
- Can create 2^32 different addresses
- Can handle up to **4 GB RAM**

64-bit Operating System:

- Address size = 64 bits = 8 bytes
- Can create 2^64 addresses
- Can handle RAM > 4 GB (8 GB, 16 GB, 32 GB, etc.)

Part 4: Stack vs Heap - The Genesis

Now comes the crucial part: **Why do we need TWO types of memory (Stack and Heap)?**

Initial Approach: Sequential Storage

Simple Strategy:

RAM: [_____]

Store data sequentially:
a = 10 → [10][_][_][_]...
b = 20 → [10][20][_][_]...
c = 30 → [10][20][30][_]...

Rule: Next data goes right after the previous one.

This is essentially **STACK** - data stored one after another (Last In, First Out).

The Problem Arises

Now, what if:

b = "Sun" // Change b from 20 to a string

Issue:

- The number `20` takes 1 byte
- The string `"Sun"` needs 3 bytes (S + U + N, each character = 1 byte)
- The original space (1 byte) is NOT enough!

What are the options?

Option 1: Shift everything

Before: [10][20][30][_][_]...
After: [10][S][U][N][30]...

Problems with Option 1:

1. **Slow**: Need to move all subsequent data
2. **Address Changes**: The address of `30` changed from position 3 to position 5
3. If `c` is used 50 times in code, all 50 references need address updates!

This is **extremely inefficient**.

Alternative Approach: Random Allocation (Heap)

Instead of sequential storage, store wherever space is available:

RAM: [_____]

a = 10 → Store at position 5
b = 20 → Store at position 12
c = 30 → Store at position 3

Benefits:

- Flexible space allocation
- Can allocate any amount of space needed

Same Problem Persists: When `b` changes from `20` to `"Sun"`:

1. Need to find NEW space for 3 bytes
2. Address of `b` changes
3. All code references need updating

Part 5: The Brilliant Solution - Pointer System

The Insight

The lecturer had a genius realization:

Problem: When data changes location, addresses change, causing widespread code updates.

Solution: Use an **extra layer of indirection**!

How It Works

Step 1: Store actual data in HEAP

"Sun" → Stored at Heap address 0x1010

Step 2: Store the HEAP address in STACK

Variable b → Stack address 0x0004
Value at 0x0004 = 0x1010 (points to heap)

The Magic

When `b` changes:

```
b = "Mohit"  // 5 characters now
```

Process:

1. Find new space in Heap for "Mohit" → say address 0x2020
2. **Only update the Stack**: Change 0x0004 from `0x1010` to `0x2020`

Result:

- Stack address of `b` (0x0004) **never changes**
- Code doesn't need updates
- Only the heap address pointer is modified

Two-Step Access

To access `b`:

1. Go to Stack (b's address) → Get heap address
2. Go to that Heap address → Get actual data

Summary of Stack vs Heap

STACK:

- Stores addresses/pointers
- Fixed, small size (KBs to few MBs)
- Fast access
- Sequential allocation

HEAP:

- Stores actual data
- Large size (MBs to GBs)
- Slower access (need to find free space)
- Random allocation

Part 6: Fixed-Size vs Dynamic-Size Data

Fixed-Size Data → Can Use Stack Directly?

For numbers like `10`, `20`, `30`:

- Size is fixed (always takes same space)
- If `a = 10` changes to `a = 50`, both take same space
- Could theoretically store directly in stack

BUT WAIT! There's a catch with JavaScript...

Part 7: JavaScript's Immutability Rule

Primitive Data Types Are IMMUTABLE

In JavaScript, **primitive values cannot be changed**.

let a = 10

When stored:

Memory location 0xABC: [10]

Now:

a = 50

JavaScript's Rule: Cannot modify the original `10`. Instead:

1. Create NEW memory location
2. Store `50` there (say address 0xDEF)
3. Update `a` to point to new address

The Problem This Creates

Even though numbers are fixed-size, **their memory location changes** due to immutability!

Consequence:

let a = 10 // Address: 0xABC

// Later:
a = 50 // New address: 0xDEF

If `a` is used 50 times in code, all 50 references need address update from `0xABC` to `0xDEF`.

This is slow!

Part 8: The Final Solution - Everything in Heap

JavaScript's Approach

ALL DATA goes to Heap (primitives AND non-primitives), but addresses stored in Stack.

let a = 10
let b = 20

Memory Layout:

HEAP:
Address 0x1000: [10]
Address 0x1008: [20]

STACK:
Variable a → 0x1000 (points to heap)
Variable b → 0x1008 (points to heap)

When Value Changes

a = 50

Process:

1. Create new location in Heap → 0x2000
2. Store 50 there
3. Update Stack: `a` now points to `0x2000` (instead of `0x1000`)

Key Point: Stack's address for variable `a` **never changes**. Only the heap address it points to changes.

Part 9: Optimization 1 - Pre-allocated Values

The Special Cases

For `true`, `false`, `null`, `undefined`:

- Only a few possible values
- Used very frequently

Optimization: When program starts, **pre-allocate** these in Heap:

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

Benefit: All boolean variables can point to **same** `true` or `false`:

```
let x = true  // Points to 0x0001
let y = true  // Also points to 0x0001
let z = true  // Also points to 0x0001
```

No need to create multiple `true` values!

Part 10: The Loop Performance Problem

The Inefficiency

```
for(let i = 0; i < 100; i++) {
   console.log(i)
}
```

What happens:

- i = 0 → Create in Heap, allocate memory
- i = 1 → Create NEW location in Heap, allocate memory
- i = 2 → Create NEW location in Heap, allocate memory
- ...continues 100 times

Problem:

- Finding free space in Heap is **SLOW**
- Allocating memory 100 times makes loop **very slow**

Attempted Solution: Pre-allocated Array

Create an array with numbers 0 to 100,000 when program starts:

const numbers = [0, 1, 2, 3, ..., 100000]

Memory:

Base Address: 1000
numbers[0] at 1000
numbers[1] at 1008 (1000 + 8 bytes)
numbers[2] at 1016 (1000 + 8*2)
...

Formula to access:

Address = Base Address + (Index × Size of Data)

Benefit: No need to search for free space!

Problem: Uses 100,000 × 8 bytes = 800 KB just for this array!

Part 11: The Genius Solution - SMI (Small Integers)

The Breakthrough

Address size in 32-bit system = 32 bits (4 bytes)

Key Insight: Use the address itself to store small numbers!

How It Works

An address looks like:

[31 bits][1 bit]

Strategy:

- **Last bit = 0**: The address contains a number directly
- **Last bit = 1**: The address points to heap data

Example

let i = 5

Instead of:

Stack: i → 0x1000
Heap: 0x1000 → [5]

Do:

Stack: i → [0000...0101][0]
↑ ↑
number 5 flag=0 (is number)

The first 31 bits **store the number directly**!

Reading the Value

To get the value:

// If address = 0x00000028 (binary: ...0101000)
// Last bit = 0 → It's a number
// First 31 bits = 5

actual_value = address >> 1 // Shift right by 1 bit

Range of SMI

With 31 bits:

- Total combinations = 2^31
- Half for positive: 0 to 2^30 - 1
- Half for negative: -2^30 to -1

First bit determines sign:

- First bit = 0 → Positive
- First bit = 1 → Negative

Benefits

1. **No heap allocation needed** for small numbers
2. **No memory search** required
3. **Instant value access** - just read the address
4. **Loop performance**: Dramatically faster!

for(let i = 0; i < 100; i++)

Now just increments address value directly - **no heap operations**!

Part 12: Complete Memory Model

Final Architecture

STACK:

- Stores addresses only
- Address size fixed (32-bit or 64-bit)

HEAP:

- Stores actual data
- All dynamic/large data
- Numbers outside SMI range
- Strings, objects, arrays

Decision Tree

Is it a number?
  ├─ Yes: Is it in range [-2^30, 2^30-1]?
  │   ├─ Yes: Store as SMI (in address itself)
  │   └─ No: Store in Heap
  │
  └─ No: Store in Heap

Why Everything Appears in Heap

Technically:

- Small integers (SMI): Stored as addresses (not actually in heap)
- Large numbers: Stored in Heap
- Strings: Stored in Heap

- Objects: Stored in Heap
- Arrays: Stored in Heap

For simplicity, we say: "Everything goes to heap" because most data does, and SMI is an optimization detail.

Part 13: Garbage Collection

The Problem

let a = 10 // Heap: 0x1000 → [10]
a = 50   // Heap: 0x2000 → [50]

Now `[10]` at 0x1000 is **orphaned** - no variable points to it.

Solution: Garbage Collector

Process:

1. **Mark Phase:** Start from Stack, mark all reachable heap data
2. **Sweep Phase:** Delete all unmarked heap data

Example:

Stack:
  a → 0x2000
  b → 0x3000

Heap:
  0x1000: [10] ← Orphaned
  0x2000: [50] ← Marked (a points here)
  0x3000: [20] ← Marked (b points here)

Garbage collector removes data at 0x1000.

Why Important:

- Frees up heap space
- Prevents memory leaks
- Prevents program crashes from running out of memory

Part 14: 64-bit Systems

In 64-bit systems:

- Address size = 64 bits = 8 bytes
- More bits available for SMI
- Can represent even larger range of small integers

Summary of Key Concepts

1. **Memory Hierarchy:** Hard Disk → RAM → CPU
2. **Addressing:** Each byte gets unique address
3. **Stack vs Heap:** Stack for addresses, Heap for data
4. **Pointer System:** Extra indirection layer prevents widespread address updates
5. **Immutability:** JavaScript primitives can't be modified in place
6. **SMI Optimization:** Store small integers in address itself
7. **Garbage Collection:** Automatic cleanup of unreferenced data

---

This system demonstrates brilliant engineering - solving performance problems through clever bit manipulation and understanding of computer architecture!

---

1. The Big Picture: Where Does Your Code and Data Live? (Permanent vs. Temporary Storage)

First Principle: Computers are machines that execute instructions on data. Instructions (code) and data must be *stored somewhere* and *accessed quickly*. Storage speed vs. capacity is the core trade-off.

- **Permanent Storage (Disk/SSD/HDD)**: Your apps (PUBG, Instagram) are files on disk. Why? Disks are cheap, high-capacity (TB+), but *slow* (ms latency). Code here is "dormant"—like a book on a shelf.
    - Example: Instagram.apk (or .exe) is ~100MB on your SSD.
    - Persistence: Survives power-off. Switch off your phone? Apps don't vanish.
- **Temporary Storage (RAM - Random Access Memory)**: To run code, load it into RAM. Why RAM? *Ultra-fast* access (ns latency), but volatile (loses data on power-off) and expensive (~$50/GB).
    - Process: OS (Android/iOS/Windows) copies code from disk → RAM. CPU fetches instructions from RAM, executes them.

- Why not direct disk-to-CPU? Disk is 10^6x slower than RAM. Bottleneck: CPU idles waiting for data.
- Analogy: Disk = library (vast, slow walk). RAM = desk (small, instant grab). CPU = you reading at the desk.

Aspect	Permanent Storage (Disk/SSD)	Temporary Storage (RAM)
Speed	Slow (10-100ms seek time)	Fast (10-100ns access)
Capacity	High (GB-TB, cheap)	Low (GB, expensive)
Persistence	Yes (survives reboot)	No (cleared on shutdown)
Use Case	Store code/data at rest	Execute code/data in flight
Size Example	1TB SSD for 1000s of apps	8GB RAM for 5-10 active apps

Derivation: RAM size limits multitasking. 16GB RAM? Run 2-3 heavy apps (PUBG + Instagram) before swapping to disk (thrashing → lag). Close PUBG? OS deallocates its RAM chunk.

Pitfall: Forgetting this leads to "memory leaks" later—unused data clogs RAM, forcing swaps.

---

2. Inside RAM: Byte-Addressable Memory and Unique Identification

First Principle: RAM is a sea of bits. To store/retrieve data, we need *addresses*—like house numbers. Without them, "where is my 20?" becomes a search nightmare.

- **Byte-Addressable Memory**: RAM divided into 8-bit chunks (bytes). Each byte gets a unique address (0 to N-1).
    - Why bytes? 1 byte = 256 values (2^8). Fits chars (ASCII), small nums. Larger data (e.g., int) spans multiple bytes.

Example: 16-byte RAM (toy model). Addresses: 0x00 to 0x0F (hex for brevity).
text
```
RAM Layout (16 bytes):
Addr: 00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F
```

    - Byte: [ ] [ ] [10][ ] [20][ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
        - Store a = 10: Assume 10 fits 1 byte (binary: 00001010). Put at addr 02.
        - Store b = 20: Binary 00010100 at addr 04.
- **The Problem: Uniqueness**: Update b = 25. Scan RAM for "20"? Multiple 20s? Data corruption!

    - Solution: **Mapping Table** (implicit in code). Variables → Addresses.
        - a maps to 0x02 (holds 10).
        - b maps to 0x04 (holds 20).

Behind scenes: Compiler/OS replaces a with its address (0x02). Code becomes: "Load from 0x02, store 10."
text
```
Source Code:    let a = 10; b = 20; b = 25;
```

    - Compiled View: [addr:0x02] = 10; [addr:0x04] = 20; [addr:0x04] = 25;
- **Address Size Math**: For M bytes, need log2(M) bits to address them.
    - 16 bytes: 2^4 = 16 → 4 bits (0000 to 1111).
    - 32 bytes: 2^5 = 32 → 5 bits.
    - Real RAM (4GB = 2^32 bytes): 32 bits (4 bytes/address).
    - 64-bit system: 64 bits (8 bytes/address) → Addresses 16 exabytes (huge).

RAM Size	Addresses Needed	Bits per Address	Binary Example
16 bytes	16	4	0000 to 1111
1KB (1024B)	1024	10	0000000000+
4GB (2^32 B)	2^32	32	0x00000000+

Derivation: Addresses stored in binary. For 32-bit: 4 bytes/address. Total addressable: 2^32 bytes = 4GB. Exceed? OS uses virtual memory (pages to disk).

Pitfall: Fixed address size limits RAM. 32-bit OS? Max ~4GB user RAM (even if hardware has more).

---

3. Stack vs. Heap: Sequential vs. Flexible Allocation

First Principle: Allocation strategies trade speed for flexibility. Fixed-size data? Predictable. Dynamic? Free-form.

- **The Split**: RAM divided: Stack (small, fast, sequential) + Heap (large, flexible, scattered).
    - Stack: LIFO (Last In, First Out). Grows/shrinks predictably (e.g., function calls).
        - Size: KB-MB (e.g., 1MB default).

Allocation: Sequential. Next item = last + fixed size.
text
```
Stack Growth (downward, convention):
```

```
High Addr
| ... |
| c=30 (addr 0x10, 1 byte) |
| b=20 (addr 0x09, 1 byte) |
| a=10 (addr 0x02, 1 byte) | ← Top (growing down)
```

        - Low Addr
        - Pros: Fast (no search). Fixed-size data (ints: 8 bytes) fits perfectly.
        - Cons: Variable-size (strings) breaks it—overwrite neighbors or shift everything (O(n) time, address updates everywhere).
    - Heap: Dynamic. Allocate anywhere free.
        - Size: MB-GB (rest of RAM).

Allocation: Search free blocks (e.g., buddy system). References (addresses) stored in stack.
text

```
Heap (scattered):
Addr: 0x1000 [Rohit (5 bytes: R-O-H-I-T)]  ← Dynamic string
      0x2000 [free]
      0x3000 [Array: [1,2,3] (24 bytes)]
```

        - Stack Ref: d → 0x1000 (8-byte address)

Feature	Stack	Heap
Allocation	Sequential (next slot)	Scattered (find free block)
Size	Fixed per item (predictable)	Variable (dynamic)
Speed	Ultra-fast (no search)	Slower (search + fragmentation)
Data Types	Primitives (ints, bools)	Objects, arrays, strings
Access	Direct value	Via reference (2-step: stack → heap)
Deallocation	Auto (scope end)	Manual/GC (later)

- **Why Split?** From scratch: Sequential for small/fixed (no waste). Flexible for big/variable (no shifts).
    - Fixed-size update: a=50 → Overwrite slot (O(1)).
    - Variable: b="Rohit" (5 bytes) → Can't fit 1-byte slot. Shift? Update 50+ code refs (slow). Solution: Heap + ref in stack.
    - Math: String "Rohit" = 5 chars × 1 byte/char (UTF-8 basic) = 5 bytes + null terminator.

Derivation: Fragmentation in heap: Free blocks scatter → "holes." Allocators merge/coalesce.

Pitfall: Stack overflow (deep recursion). Heap fragmentation → OOM (Out of Memory).

---

4. JavaScript Specifics: Everything in Heap, Immutability, and References

First Principle: JS is high-level—hides low-level details. But V8 (JS engine) must map to machine code efficiently.

- **Immutability of Primitives**: Primitives (number, string, boolean, null, undefined, symbol) can't mutate. Why? Thread-safety, predictability.
    - let a = 10; a = 20; → Not mutate 10. Create new 20, re-point a.
    - Generic: Even fixed-size primitives go to heap (avoids address churn).

All in Heap: Variables (in stack-like "activation records") hold *references* (addresses, 8 bytes on 64-bit).
text

```
JS Code:       let a = 10; let b = "Hi";
V8 Memory:
Stack:         a → 0x7FFF... (ref to heap)
               b → 0x7FFE... (ref to heap)
Heap:          0x7FFF... [10 (8 bytes, Number object)]
```

- 0x7FFE... ["Hi" (3 bytes + overhead)]
- **Access**: 2-step indirection. console.log(a) → Stack ref → Heap value.
    - Update: a=20 → New heap slot for 20, update stack ref. Old 10 orphaned → GC later.
- **Pre-Allocation for Constants**: true/false/null/undefined created at startup, single instances.
    - Benefit: Reuse refs (e.g., all true point to one heap slot). Saves space (no 100x true allocations).

JS Type	Mutable?	Heap Allocation	Stack Holds	Example Update Cost
Number (primitive)	No	Yes (8 bytes)	Ref (8 bytes)	New alloc + ref swap
String	No	Yes (var len)	Ref	New alloc (copy chars)
Object	Yes	Yes (props dynamic)	Ref	Mutate in-place if space; else realloc
Array	Yes	Yes (elements)	Ref	Resize/realloc if overflow

Derivation: Ref size = pointer size (64-bit: 8 bytes). Total for a=10: 8 (ref) + 8 (value) = 16 bytes.

Pitfall: Immutability illusion—strings concat creates new strings (inefficient loops).

---

5. Garbage Collection: Cleaning the Heap

First Principle: Heap grows forever without cleanup → OOM crash.

- **Mark-and-Sweep GC**: V8's algorithm.
    - **Mark**: From stack roots (variables), follow refs. Mark reachable (e.g., a → 10 marked "live").
    - **Sweep**: Unmarked → Free (e.g., old 10 after a=20).
    - Frequency: Generational (young/old gen). Young: Frequent, small. Old: Rare, large.

text
```
Before GC:
Stack: a → Heap[20] (marked)
Heap: [10 (unmarked)] [20 (marked)] ["OldStr" (unmarked)]

After: Free [10] and "OldStr". Compact to reduce fragmentation.
```

- **Why Needed?** Recursion without base case: Stack fills (crash), but heap leaks if cycles.

Derivation: Time: O(heap size). Space: Compaction halves pauses.

Pitfall: GC pauses → Jank in UI. V8 mitigates with incremental GC.

---

6. Ultra-Optimization: Tagged Pointers for Small Integers (SMI)

First Principle: Most loop vars? Small ints (-2^30 to 2^30). Heap alloc per tick? Wasteful (search + alloc).

- **Problem in Loops**: for(let i=0; i<100; i++) → 100 heap allocs (slow).
    - Each: Find free slot (O(log N)), alloc 8 bytes.
- **Solution 1: Pre-Alloc Array** (Naive).
    - Startup: Create array[0...1e6] in heap (base addr 0x1000).
    - Access: i ref = base + i * 8.
        - Read: Heap[base + i*8].
        - But: Wastes 8MB upfront. Still indirection.
- **Solution 2: Virtual Array (No Real Alloc)**.
    - Treat refs as math: Value = (ref - base) / 8.

    - No heap visit for read—pure calc.
- **Solution 3: Tagged Pointers (V8 Magic)**.
    - 64-bit pointer: 64 bits. Use LSB (bit 0) as tag.
        - LSB=0: SMI. Value = pointer >> 1 (shift right, divide by 2).
        - LSB=1: Heap ref. Dereference for real data.
    - Why LSB? Alignments ensure pointers even (LSB=0 free).
    - Math:
        - Store 42: Binary ...0101010 | 0 (tag). Value = (...0101010 | 0) >> 1 = 42.
        - Read: If LSB=0, >>1 → int. Else, load from addr.
    - Range: 31 bits signed (MSB sign bit): -2^30 to 2^30-1 (~1B).
        - Positive: 0xxx... → Value as-is.
        - Negative: 1xxx... → Two's complement.

Pointer Bits (64)	Tag (LSB)	Interpretation	Example (Hex)	Decoded Value
...0000 101010	0	SMI (small int)	0x...2A	42
...0001 000000	1	Heap ref (string/obj)	0x...1	Load[addr]
...1000 000000	0	SMI (negative)	0x...80	-64

- **Loop Win**: i++ → Increment pointer (add 2, keep LSB=0). No alloc/search.
    - i=42 → Pointer = 42 << 1 = 84 (binary ...1010100).
    - Read: LSB=0 → 84 >> 1 = 42.

Derivation: 2^63 total pointers. Half tagged (even). Signed: 2^31 values.

Pitfall: Large ints (>2^30) → Heap (slow). V8 detects, converts.

Conclusion: The Engineering Mindset

This isn't "stack=primitives, heap=objects" trivia—it's *why*: Efficiency from immutability + tagging. JS hides it, but understanding prevents leaks/slowness. Homework: Read V8 source on tagged pointers (or 64-bit variants—uses more bits). Replay: Build a toy allocator in Python.

Minds blown? That's the point. Next: Apply to leaks in your code. Jai Hind!