Trash Talk
Understanding Memory Management

Shelley Vohr
@codebytere

Garbage Collection?

A form of **automatic memory management.** Attempts to **reclaim garbage**, or memory occupied by objects that are no longer in use by the program.

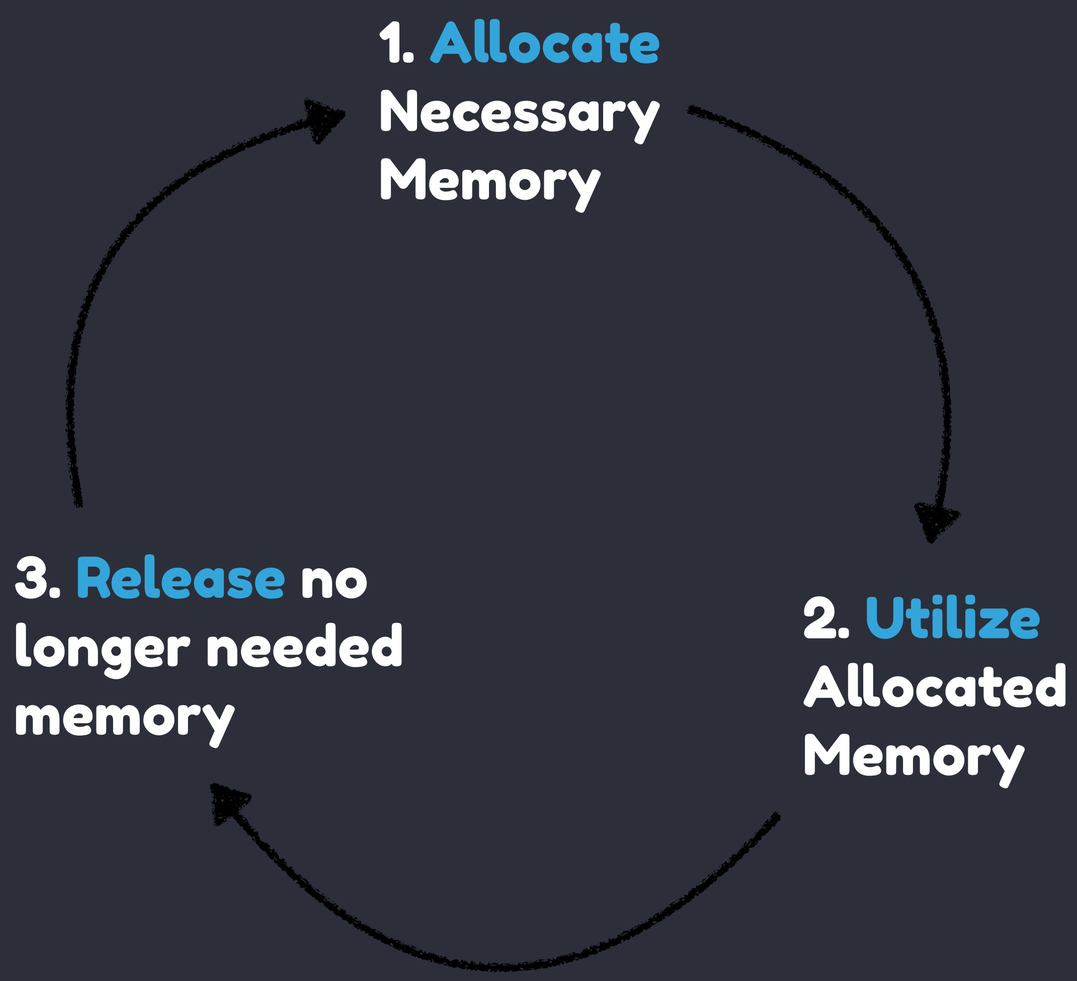
GC in JavaScript?

JavaScript **automatically allocates** and **frees** memory

Out of **Sight**, Out of **Mind**?

_
•.•) You should probably care about
⊂/ memory management in JS

Memory Lifecycle



1. Explicit in lower-level languages, **implicit in JavaScript**

2. **Always explicit**, no matter how close you are to the metal

3. Explicit in lower-level languages, **implicit in JavaScript**

1. Allocate Necessary Memory

Value Initialization

```
// allocate memory for a number
const num = 12345

// allocate memory for a string
const str =  'Covalence Conf'

// allocate memory for object & contained values
const conf = {
  name,
  location: 'Slack HQ'
}
```

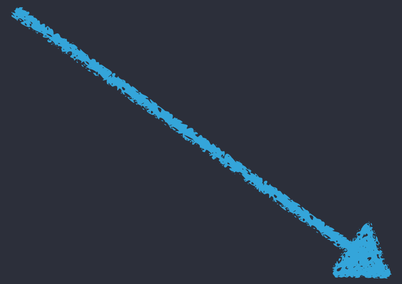
Allocation Via Function Call

```
// allocates a Date object
const date = new Date(2020, 1, 24)

const name = 'Covalence Conf'

// may allocate memory
const part = s.substr(0, 9)
```

2. Use Allocated Memory

Reading and **writing**
in allocated memory

Use variable
by passing it
to a function



```
const confName = 'Covalence Conf'

function printName (name) {
    console.log(`Name is ${name}`)
}

printName(confName)
```

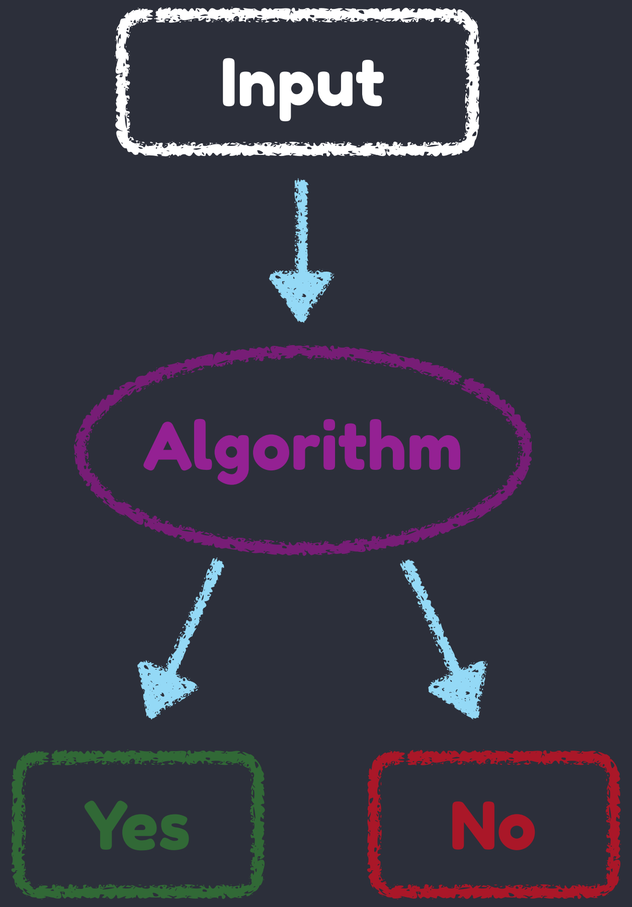
3. Release No Longer Needed Memory

Finding whether or not memory is needed is an **undecidable problem**

How do we know when an allocated piece of memory is **garbage collectible**?

Decidability

A problem is **decidable** when there is a proven effective method for **determining membership** (returning a correct answer after a finite period of time) exists for all cases.



Reference-Counting

Whether or not an object is **still needed** → whether an object still has any **other objects referencing it**

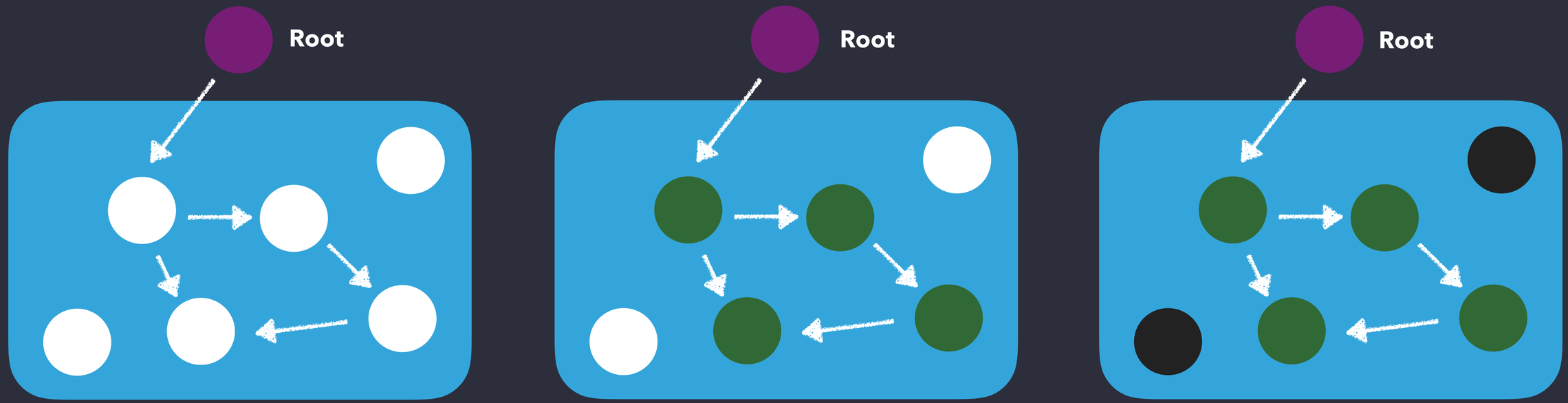
Problem: circular references

```
// 2 objects created; nothing can be gc'd
const first = { a: { b: 'covalence' } }

const second = first

first = 1

const third = second.a

// 'first' can now be gc'd,
// but 'a' is still ref'd and cannot be
second = 'electron'

// 'a' can now be gc'd
third = null
```

Mark-and-Sweep

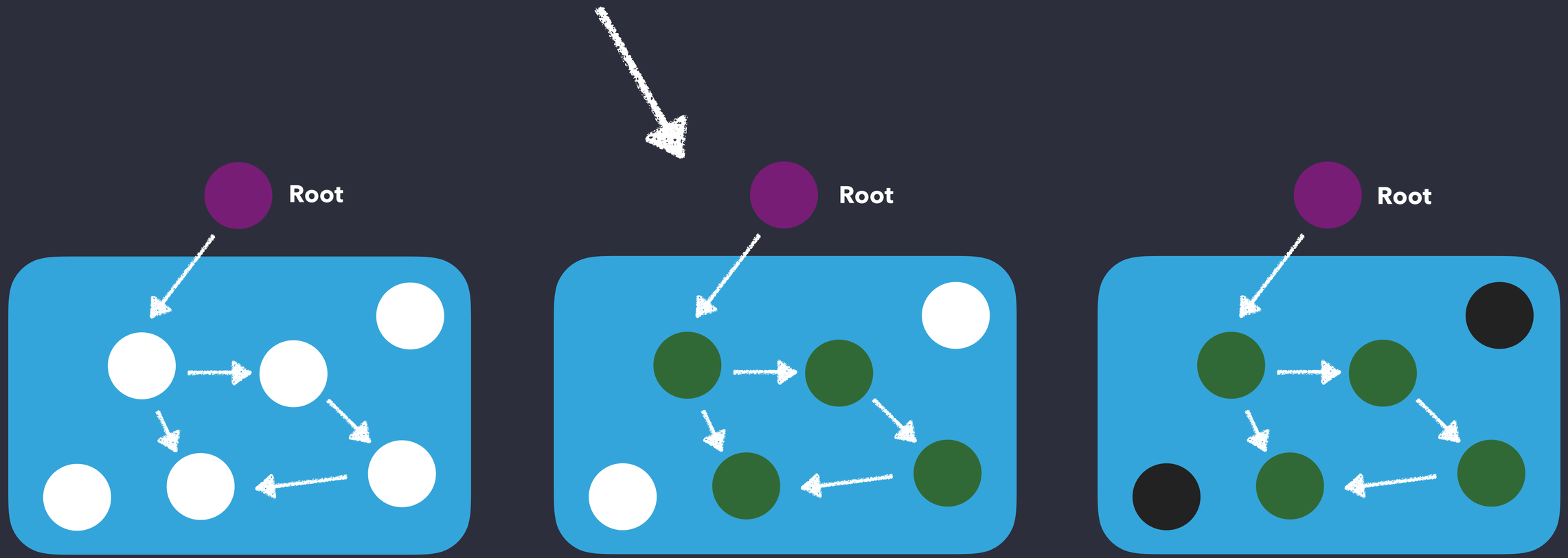
Whether or not an object is **still needed** → whether an object is **unreachable**

Problem: can't manually release memory



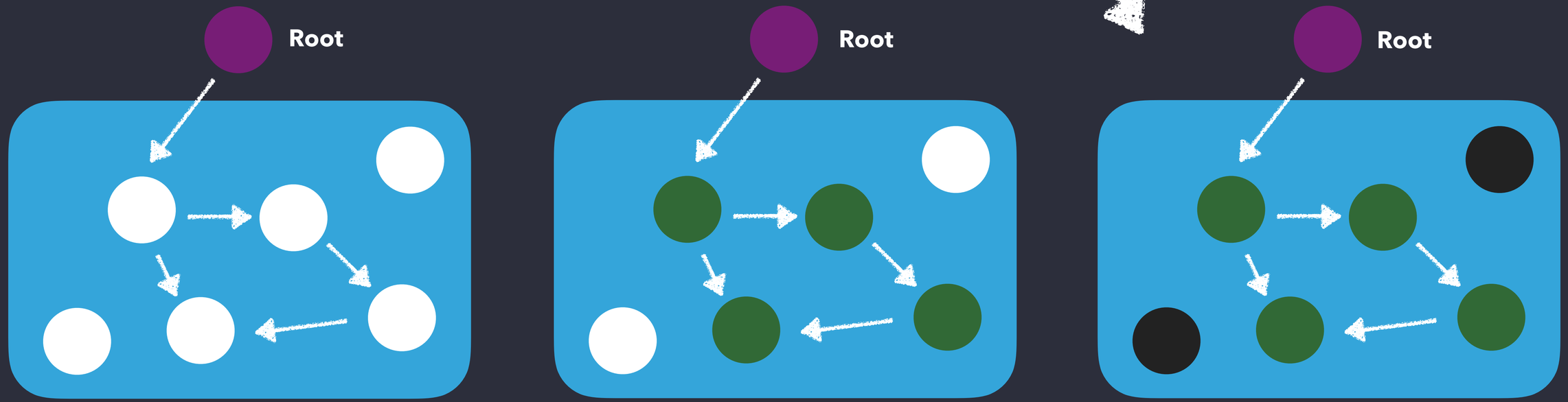
Marking

The process by which **reachable objects** are **found.**



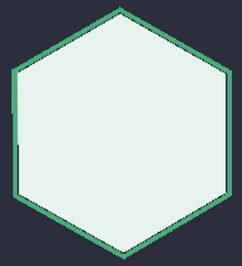
Sweeping

The process by which **unreachable objects** are **cleared** from **heap memory.**



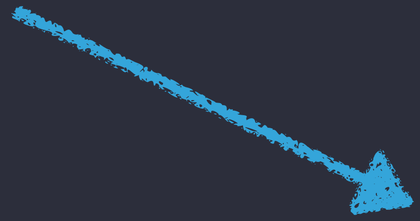
Electron

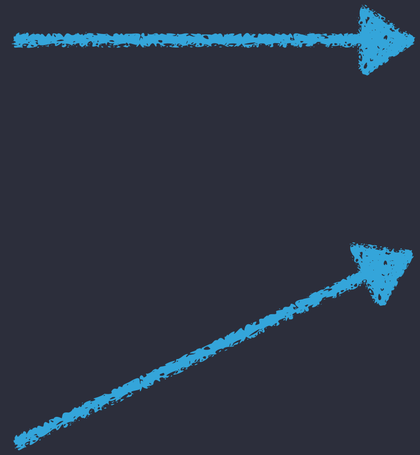
Node.js for filesystems and networks



Chromium for making web pages

Native APIs for three systems





V8

Google's open source **high-performance** JavaScript and WebAssembly **engine**, written in C++.

Implements **ECMAScript** and **WebAssembly.**

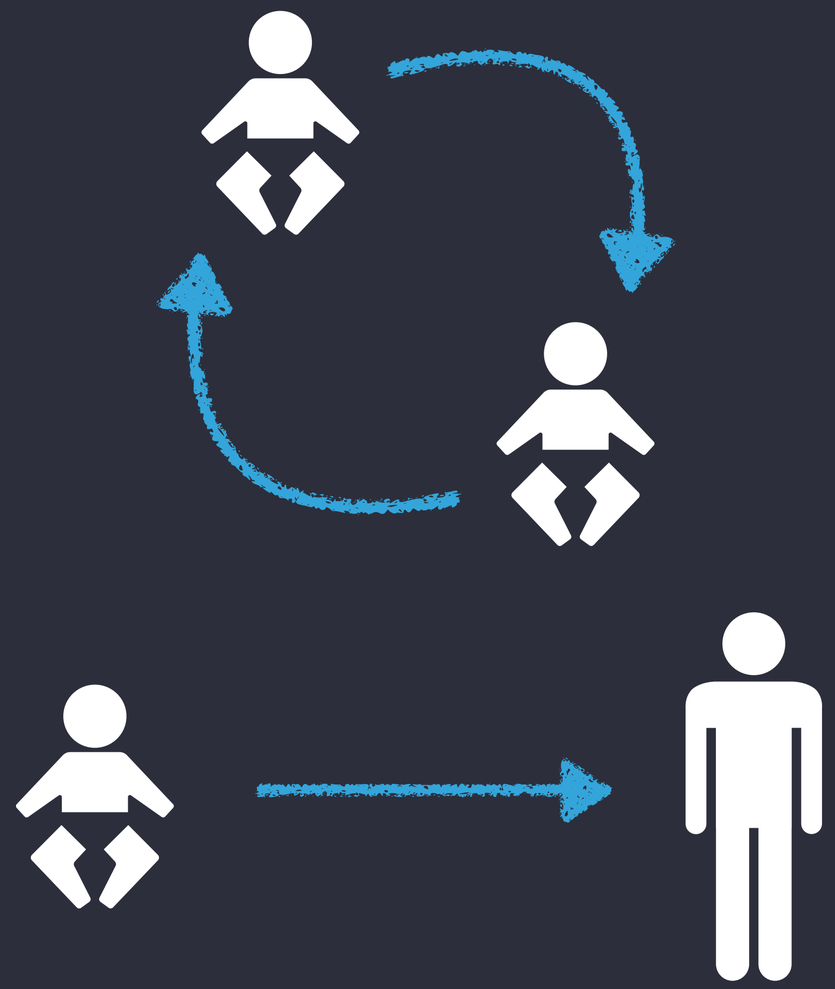
Contains a **mark-and-sweep garbage collector** called Orinico.

Mostly parallel and concurrent collector with incremental fallback.

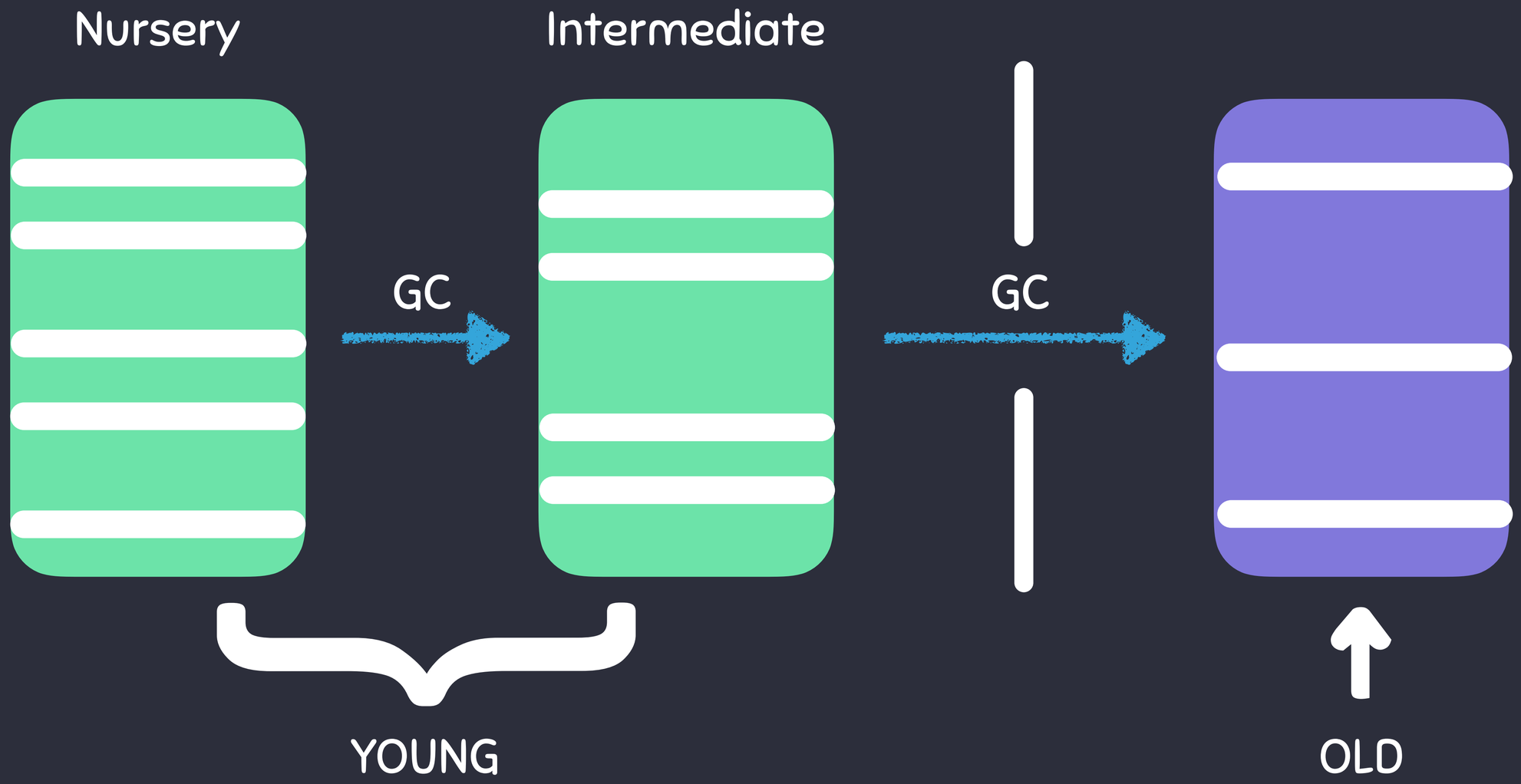
Generational Hypothesis

- Most allocated values become **unused quickly**

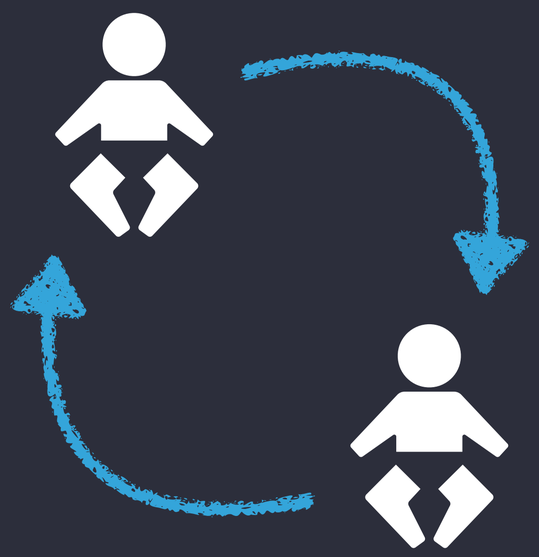
- The ones that do not usually **survive** for a **long time**



Heap Layout



Minor GC



Collects dead objects from the 2-part **young generation**

Increases efficiency by **leveraging** the Generational Hypothesis

Major GC

Collects garbage across the **whole heap**

Mark → **Sweep** → **Compact**

What about Electron?

What are we **learning all this** for?

What **GC considerations** does **Electron specifically** need to make?

UI Disappearing Problems

```
const { app, BrowserWindow } = require('electron')
app.on('ready', () => {
  const win = new BrowserWindow({width: 800, height: 600 })
  win.loadURL('http://electronjs.org')
})
```

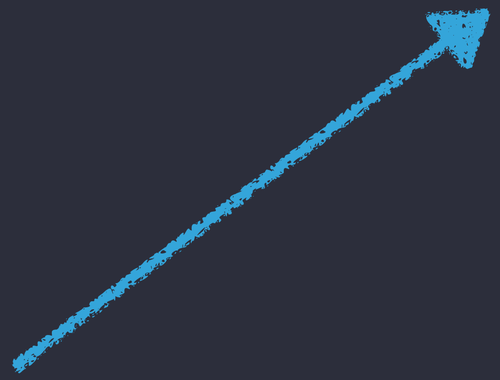
```
const { app, BrowserWindow } = require('electron')

let win
app.on('ready', () => {
  win = new BrowserWindow({width: 800, height: 600 })
  win.loadURL('http://electronjs.org')
})
```

UI considerations can sometimes **run afoul** of garbage collector principles

Weak References

A **reference to an object** that allows you to get the object **without affecting** whether it will be garbage collected. It also **allows you to be notified** when the object is garbage collected.

No way to know when an object has been garbage collected



JavaScript as of ES2020 does **not have** a way to assign weak references

Weak References

There's a Stage 3 proposal!

```
const cache = new Map();

function getImageCached(name) {
  const ref = cache.get(name);
  if (ref !== undefined) {
    const deref = ref.deref();
    if (deref !== undefined) return deref;
  }
  const image = performExpensiveOperation(name);
  const wr = new WeakRef(image);
  cache.set(name, wr);
  return image;
}
```

- Creating weak references to objects with the **WeakRef class**

- Running user-defined finalizers after objects are garbage-collected, with the **FinalizationGroup class**

https://github.com/tc39/proposal-weakrefs

--expose-gc

```
$ electron /path/to/app --js-flags="--expose-gc"
```

Add a function to the global object that will **request a garbage collection** to occur

Use wisely: garbage collection can be CPU-intensive!

The gc function

```
// Same as below default
gc()

// Default
gc({type: 'major', execution: 'sync'})

// Invoke the Scavenge GC synchronously
gc({type: 'minor', execution: 'sync'})

// Invoke whole-heap GC asynchronously
await gc({type: 'major', execution: 'async'})

// Invoke the Scavenge GC asynchronously
await gc({type: 'minor', execution: 'async'})
```

You can specific
more granular
garbage collection
behavior!

The gc function

```
const { app, BrowserWindow } = require('electron')

let win

app.on('ready', () => {
  win = new BrowserWindow()
  const obj = {}
  // use obj somehow
  obj = null
  gc({type: 'minor', execution: 'sync'})
})
```

Starts a garbage collection, resulting in a collection of **`obj`**

--expose-gc

Why use this flag?

- To better understand how much **memory you're holding onto** at certain points in your app

- To see how **application performance** is **affected** by freeing up unused memory

Other Useful V8 Flags

```
$ node --v8-options | grep gc
```

--trace-gc

```
$ electron /path/to/app --js-flags="--trace-gc"
```

Print one trace line following each garbage collection

```
117 ms: Scavenge 2.9 (3.9) -> 2.5 (4.9) MB, 1.5 / 0.0 ms  (average mu = 1.000, current mu = 1.000)
allocation failure

8272 ms: Mark-sweep 2.7 (4.9) -> 2.2 (4.4) MB, 11.6 / 0.0 ms  (+ 0.6 ms in 3 steps since start of
marking, biggest step 0.6 ms, walltime since start of marking 24 ms) (average mu = 1.000, current mu =
1.000) finalize incremental marking via task GC in old space requested

8892 ms: Mark-sweep 2.2 (4.4) -> 2.2 (4.6) MB, 4.7 / 0.0 ms  (+ 0.5 ms in 3 steps since start of
marking, biggest step 0.5 ms, walltime since start of marking 16 ms) (average mu = 0.992, current mu =
0.992) finalize incremental marking via task GC in old space requested

60152 ms: Mark-sweep 2.2 (4.6) -> 2.2 (4.6) MB, 7.5 / 0.0 ms  (average mu = 1.000, current mu = 1.000)
low memory notification GC in old space requested

60158 ms: Mark-sweep 2.2 (4.6) -> 2.2 (4.9) MB, 5.7 / 0.0 ms  (average mu = 1.000, current mu = 0.005)
low memory notification GC in old space requested
```

--gc-interval

Garbage collect after <n> allocations

```
$ electron /path/to/app --js-flags="--gc-interval=10"
```

--predictable-gc-schedule

Predictable garbage collection schedule.

```
$ electron /path/to/app --js-flags="--predictable-gc-schedule"
```

--trace-gc-object-stats

Trace object counts and memory usage

```
$ electron /path/to/app --js-flags="--trace-gc-object-stats"
```


```
{ "isolate": "0x1f8d00000000", "id": 2, "key": "live", "type": "field_data", "tagged_fields": 1277324,
"embedder_fields": 5248, "inobject_smi_fields": 1556, "unboxed_double_fields": 0,
"boxed_double_fields": 448, "string_data": 830420, "other_raw_fields": 1192152 }

{ "isolate": "0x1f8d00000000", "id": 2, "key": "live", "type": "bucket_sizes", "sizes": [ 32, 64, 128,
256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576 ] }

{ "isolate": "0x1f8d00000000", "id": 2, "key": "live", "type": "instance_type_data", "instance_type":
0, "instance_type_name": "INTERNALIZED_STRING_TYPE", "overall": 0, "count": 0, "over_allocated": 0,
"histogram": [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],"over_allocated_histogram": [ 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] }
```

Memory Mismanagement

1. Accidental Globals

2. Poor Scoping

Accidental Globals

```
function doThing() {
  something = 'oh no'
}

doThing()

// 'oh no'
console.log(window.something)
```

When a **undeclared variable** is **referenced**, a new variable gets created in the **global object**.

```
function doThing() {
  this.something = 'oh no'
}

doThing()

// 'oh no'
console.log(window.something)
```

Solution:

```
'use strict'
```

Poor Scoping

```
const { app, Tray, Menu } = require('electron')

app.on('ready', () => {
  const tray = new Tray('/path/to/icon.png')

  const contextMenu = Menu.buildFromTemplate([
    { label: 'first-item', type: 'radio' },
    { label: 'second-item', type: 'checkbox', checked: true },
  ])

  tray.setContextMenu(contextMenu)
})
```

A variable should have an **active reference** for the entire **lifetime** of its **intended usage**

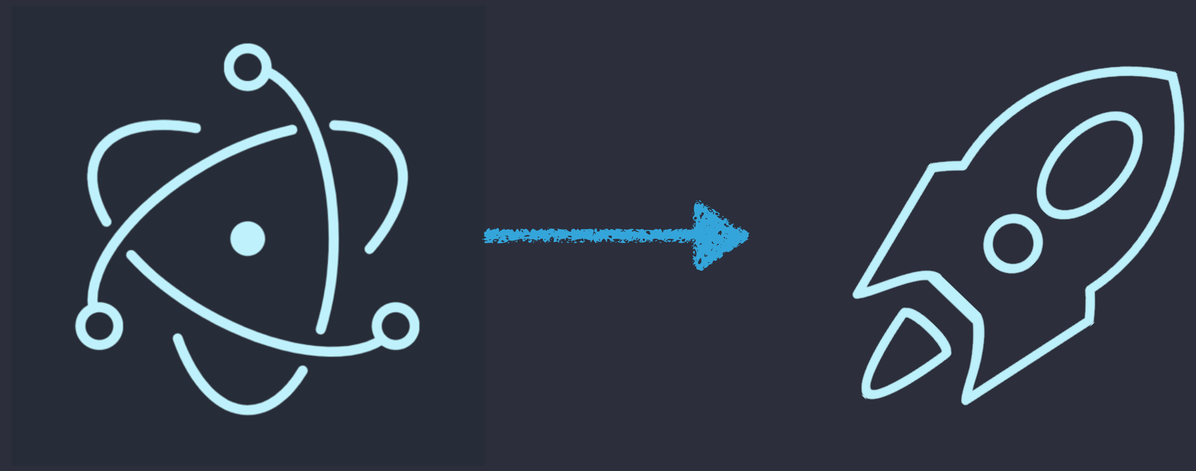
This tray will disappear!

```
const { app, Tray, Menu } = require('electron')

let tray
app.on('ready', () => {
  tray = new Tray('/path/to/icon.png')

  const contextMenu = Menu.buildFromTemplate([
    { label: 'first-item', type: 'radio' },
    { label: 'second-item', type: 'checkbox', checked: true },
  ])

  tray.setContextMenu(contextMenu)
})
```

Solution: tray should be global in this case

Wrapping Up

While you can get by in JavaScript with only cursory knowledge of garbage collection, you'll write more robust code when you consider it actively!



- **Actively consider** UI element **lifetimes**

- **Profile** your **heap usage** regularly

- **Watch** your **globals!**

THANK YOU!

@codebytere