ÔN LẠI VỀ JAVASCRIPT

JAVASCRIPT

JAVASCRIPT LÀ NGÔN NGỮ LẬP TRÌNH CẤP CAO, HƯỚNG ĐỐI TƯỢNG, ĐA MÔ HÌNH.

ÔN LẠI VỀ JAVASCRIPT

JAVASCRIPT

JAVASCRIPT LÀ NGÔN NGỮ LẬP TRÌNH BẬC CAO, HƯỚNG ĐỐI TƯỢNG DỰA TRÊN NGUYÊN MẪU, ĐA MÔ HÌNH, THÔNG DỊCH HOẶC BIÊN DỊCH ĐÚNG LÚC, ĐỘNG, ĐƠN LUỒNG, GIÚP QUẢN LÝ BỘ NHỚ VỚI CÁC HÀM FIRST-CLASS VÀ LÀ MÔ HÌNH ĐỒNG THỜI VÒNG LẶP SỰ KIỆN NON-BLOCKING.

🤔 🤯 🤣

DIỄN GIẢI KHÁI NIỆM

Bậc cao

Quản lý bộ nhớ

Thông dịch hoặc biên dịch đúng lúc

Đa mô hình

Hướng đối tượng dựa trên nguyên mẫu

Hàm first-class

Động

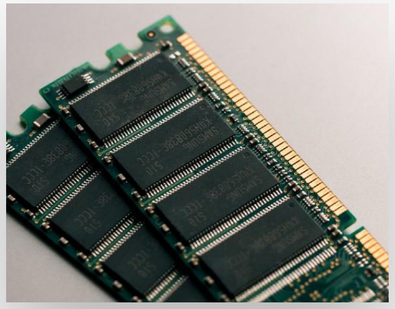
Đơn luồng

Vòng lặp sự kiện Non-blocking

DIỄN GIẢI KHÁI NIỆM

- **Bậc cao**
- Quản lý bộ nhớ
- Thông dịch hoặc biên dịch đúng lúc
- Đa mô hình
- Hướng đối tượng dựa trên nguyên mẫu
- Hàm first-class
- Động
- Đơn luồng
- Vòng lặp sự kiện Non-blocking

👉 **Bất kỳ chương trình máy tính nào cũng cần các tài nguyên:**



+

BẬC THẤP

BẬC CAO

Lập trình viên phải xử lý tài nguyên **theo cách thủ công**

Lập trình viên **KHÔNG** cần lo mọi thứ, vì chúng diễn ra tự động

DIỄN GIẢI KHÁI NIỆM

Bậc cao

Quản lý bộ nhớ

Thông dịch hoặc biên dịch đúng lúc

Đa mô hình

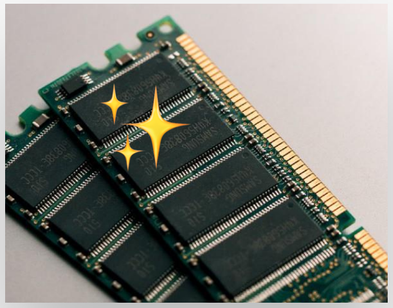
Hướng đối tượng dựa trên nguyên mẫu

Hàm first-class

Động

Đơn luồng

Vòng lặp sự kiện Non-blocking



Dọn dẹp bộ nhớ,
chúng ta không phải
lo điều này

DIỄN GIẢI KHÁI NIỆM

Bậc cao

Quản lý bộ nhớ

Thông dịch hoặc biên dịch đúng lúc

Đa mô hình

Hướng đối tượng dựa trên nguyên mẫu

Hàm first-class

Động

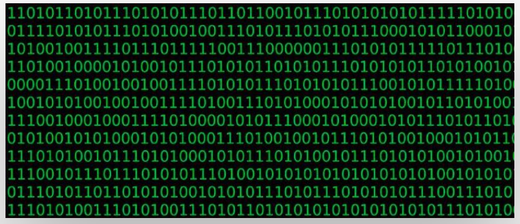
Đơn luồng

Vòng lặp sự kiện Non-blocking

```
document.querySelector(".again").addEventListener("click", () => {
  document.querySelector(".message").textContent = "Start guessing...";
  document.querySelector(".number").textContent = "?";
  document.querySelector(".guess").value = "";
  score = 20;
  document.querySelector(".score").textContent = score;
  number = Math.floor(Math.random() * 20) + 1;
});
```

Tính trừu tượng với 0 và 1

CHUYỂN THÀNH MACHINE CODE = BIÊN DỊCH



Xảy ra bên trong JavaScript engine

Tìm hiểu thêm về điều này **sau trong mục này** 👇

DIỄN GIẢI KHÁI NIỆM

Bậc cao

Quản lý bộ nhớ

Thông dịch hoặc biên dịch đúng lúc

Đa mô hình

Hướng đối tượng dựa trên nguyên mẫu

Hàm first-class

Động

Đơn luồng

Vòng lặp sự kiện Non-blocking

👉 **Mô hình (Paradigm)**: Là cách tiếp cận và tư duy về cấu trúc code, định hướng phong cách và kỹ thuật viết code của bạn.

The one we've been using so far

1. Lập trình thủ tục
2. Lập trình hướng đối tượng (OOP)
3. Lập trình hàm (FP)

☝️ **Imperative với**

👋 **Declarative**

Tìm hiểu thêm về điều này **ở nhiều mục** 👉

DIỄN GIẢI KHÁI NIỆM

Bậc cao

Quản lý bộ nhớ

Thông dịch hoặc biên dịch đúng lúc

Đa mô hình

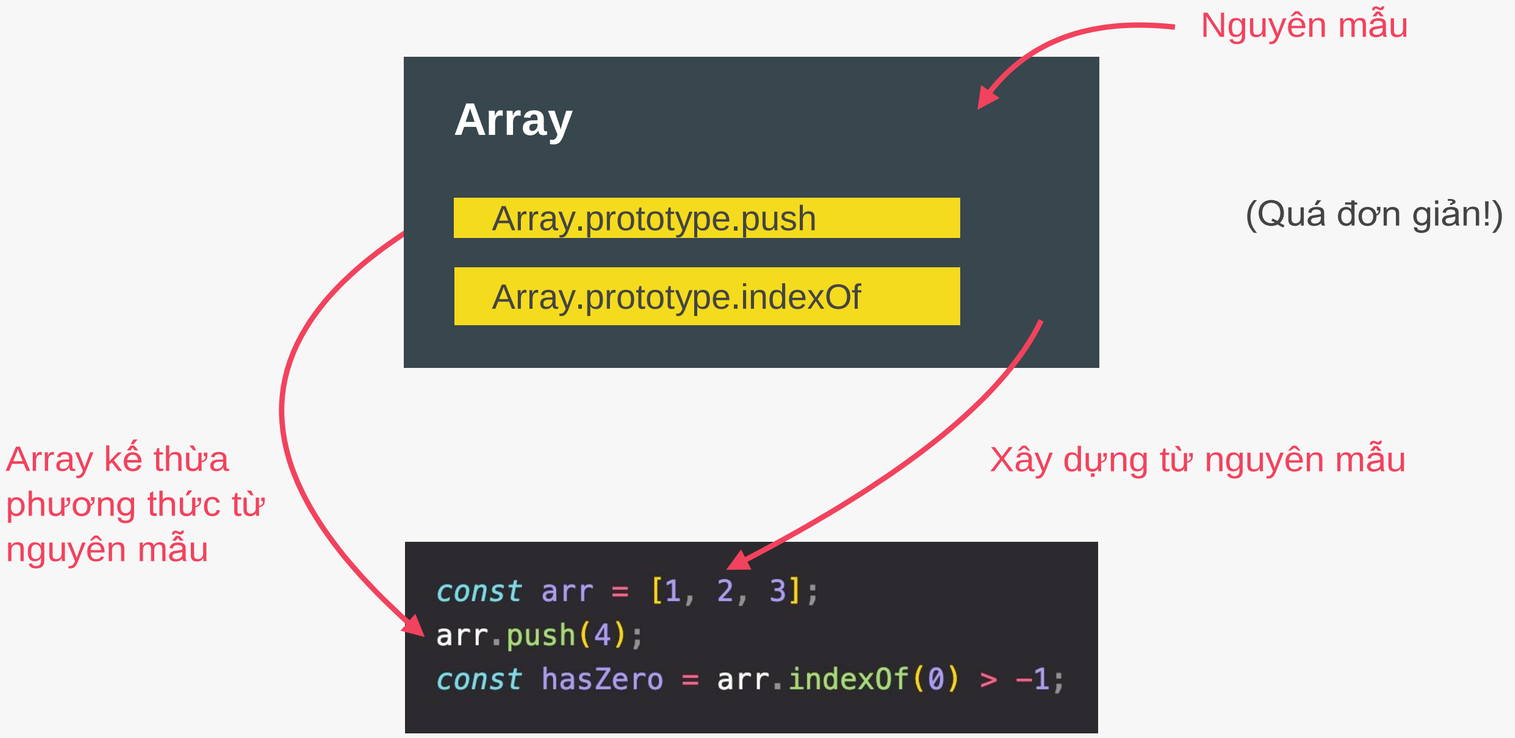
Hướng đối tượng dựa trên nguyên mẫu

Hàm first-class

Động

Đơn luồng

Vòng lặp sự kiện Non-blocking



Tìm hiểu thêm về điều này trong mục **Object Oriented Programming** 👉

DIỄN GIẢI KHÁI NIỆM

- Bậc cao
- Quản lý bộ nhớ
- Thông dịch hoặc biên dịch đúng lúc
- Đa mô hình
- Hướng đối tượng dựa trên nguyên mẫu
- Hàm first-class
- Động
- Đơn luồng
- Vòng lặp sự kiện Non-blocking

👉 Trong một ngôn ngữ với **các hàm first-class**, các hàm **được coi là biến**. Có thể truyền chúng vào các hàm khác và trả về chúng từ các hàm.

```
const closeModal = () => {
  modal.classList.add("hidden");
  overlay.classList.add("hidden");
};

overlay.addEventListener("click", closeModal);
```

Truyền hàm vào một hàm khác: Hàm first-class!

Tìm hiểu thêm về điều này trong mục **A Closer Look at Functions** 👉

DIỄN GIẢI KHÁI NIỆM

- Bậc cao
- Quản lý bộ nhớ
- Thông dịch hoặc biên dịch đúng lúc
- Đa mô hình
- Hướng đối tượng dựa trên nguyên mẫu
- Hàm first-class
- Động
- Đơn luồng
- Vòng lặp sự kiện Non-blocking

👉 **Ngôn ngữ Dynamically-typed:**

Không có định nghĩa kiểu dữ liệu.
Các kiểu sẽ rõ khi chạy

Kiểu dữ liệu của biến được thay đổi tự động

```
let x = 23;
let y = 19;
x = "Jonas";
```

DIỄN GIẢI KHÁI NIỆM

- Bậc cao
- Quản lý bộ nhớ
- Thông dịch hoặc biên dịch đúng lúc
- Đa mô hình
- Hướng đối tượng dựa trên nguyên mẫu
- Hàm first-class
- Động
- Đơn luồng
- Vòng lặp sự kiện Non-blocking

👉 **Mô hình đồng thời (Concurrency model)**: cách JavaScript engine xử lý nhiều tác vụ cùng một lúc.

↓ Tại sao cần điều đó?

👉 JavaScript chạy trong **một luồng duy nhất**, vì vậy nó chỉ có thể thực hiện từng tác vụ.

↓ Vậy còn các tác vụ dài hạn?

👉 Có vẻ như nó sẽ chặn luồng đơn. Tuy nhiên, chúng ta cần không chặn!

(Quá đơn giản!)

↓ Làm cách nào để thực hiện điều đó?

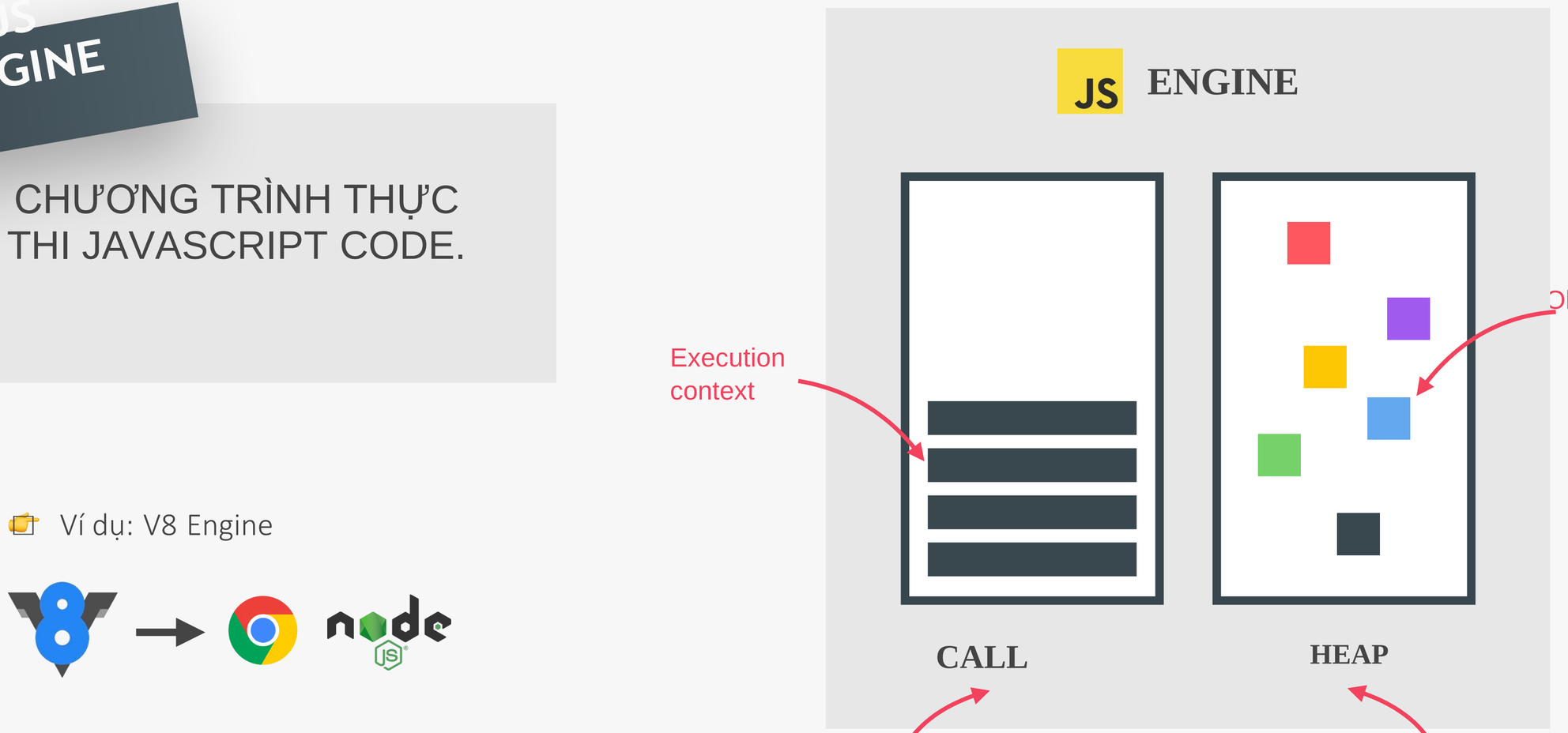
👉 Bằng cách sử dụng **event loop**: nhận các tác vụ đang dài hạn, thực thi chúng trong “background”, và đưa chúng trở lại luồng chính sau khi hoàn thành.

Tìm hiểu thêm về điều này **sau trong mục này** 👇

JAVASCRIPT ENGINE LÀ GÌ?

JS ENGINE

CHƯƠNG TRÌNH THỰC THI JAVASCRIPT CODE.

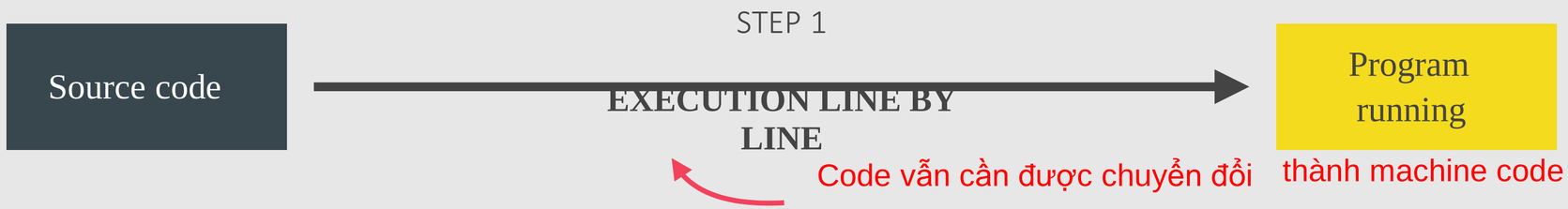


Ví dụ: V8 Engine

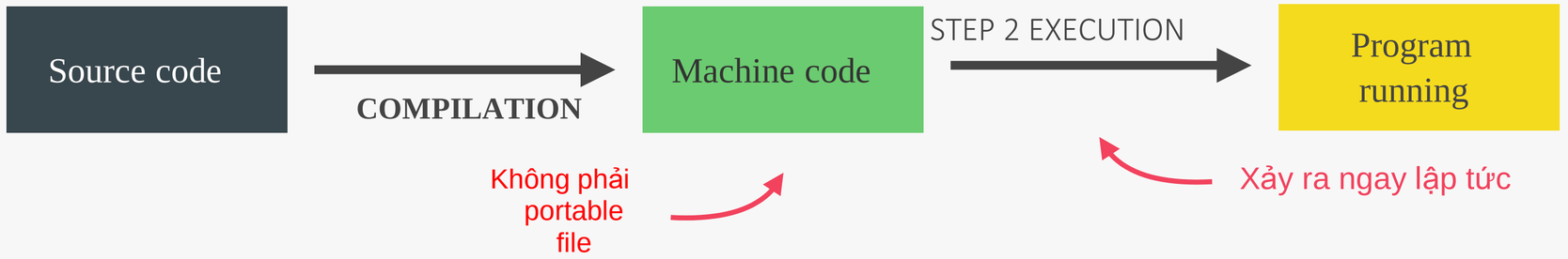
CHÚ THÍCH KHOA HỌC MÁY TÍNH: COMPILATION VÀ INTERPRETATION

- *Compilation: Toàn bộ code được chuyển đổi thành machine code cùng một lúc và được ghi vào một tệp nhị phân có thể được thực thi bởi máy tính.*

- *Interpretation: Interpreter chạy qua Source code và thực thi nó theo dòng.*



- *Just-in-time (JIT) compilation: Toàn bộ code được chuyển đổi thành machine code cùng một lúc, sau đó được thực thi ngay lập tức.*



JUST-IN-TIME COMPILATION CỦA JAVASCRIPT

```
document.querySelector(".again").addEventListener("click", () => {
  document.querySelector(".message").textContent = "Start guessing...";
  document.querySelector(".number").textContent = "?";
  document.querySelector(".guess").value = "";
});
```

~~"JavaScript là một interpreted~~

JS ENGIN

Xảy ra trong luồng đặc biệt mà chúng ta không thể truy cập

Parsing

AS

Compilation

Just-in-time compilation

Optimization

Execution

Xảy ra trongCall

Trong suốt execution!

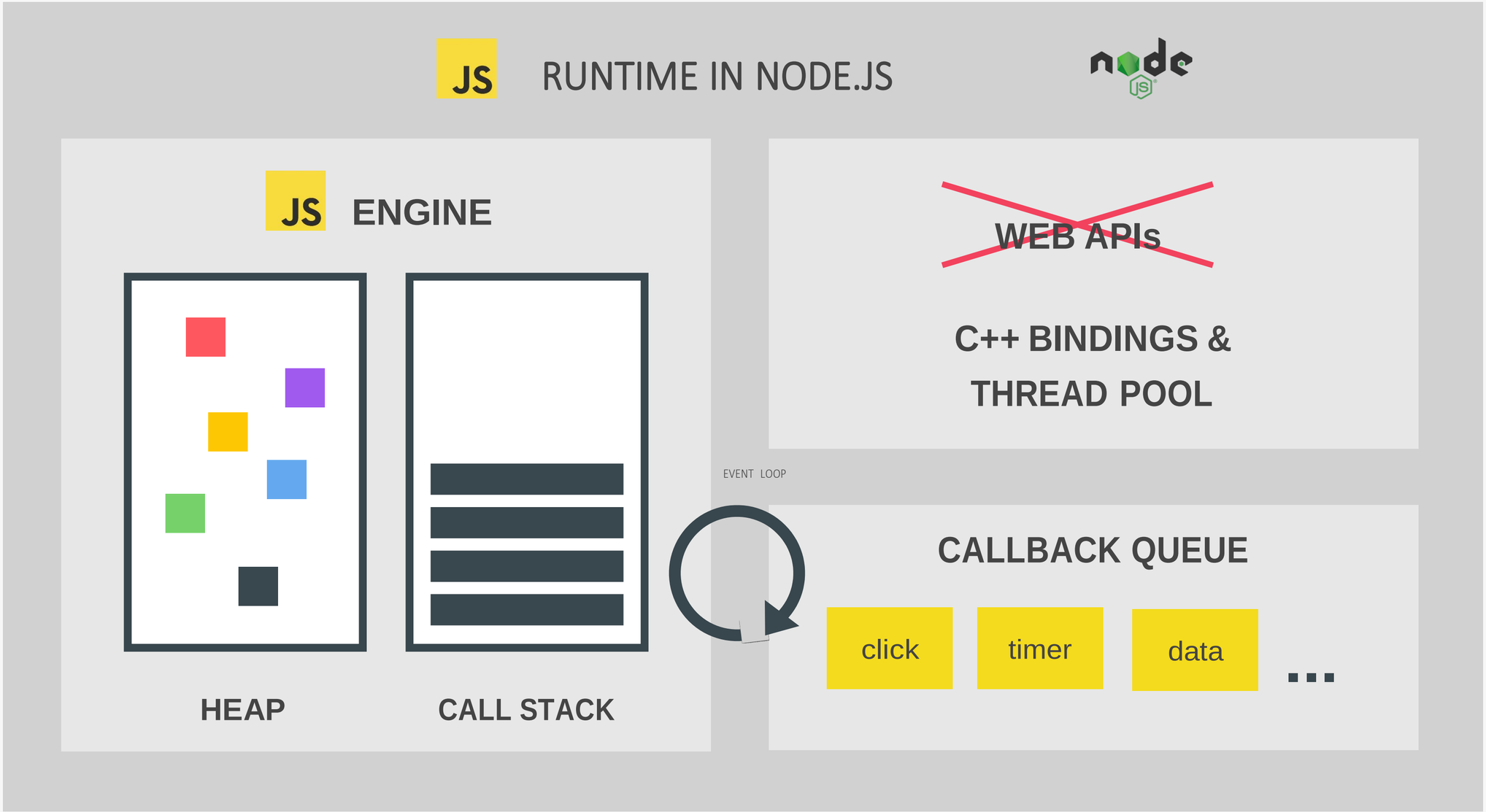
Ví dụ

AST

```
const x = 23;
```

```
- VariableDeclaration {
    start: 0
    end: 13
  - declarations: [
    - VariableDeclarator {
        start: 6
        end: 12
      - id: Identifier {
          start: 6
          end: 7
          name: "x"
        }
      - init: Literal = $node {
          start: 10
          end: 12
          value: 23
          raw: "23"
        }
    }
  ]
  kind: "const"
```

THE BIGGER PICTURE: JAVASCRIPT RUNTIME

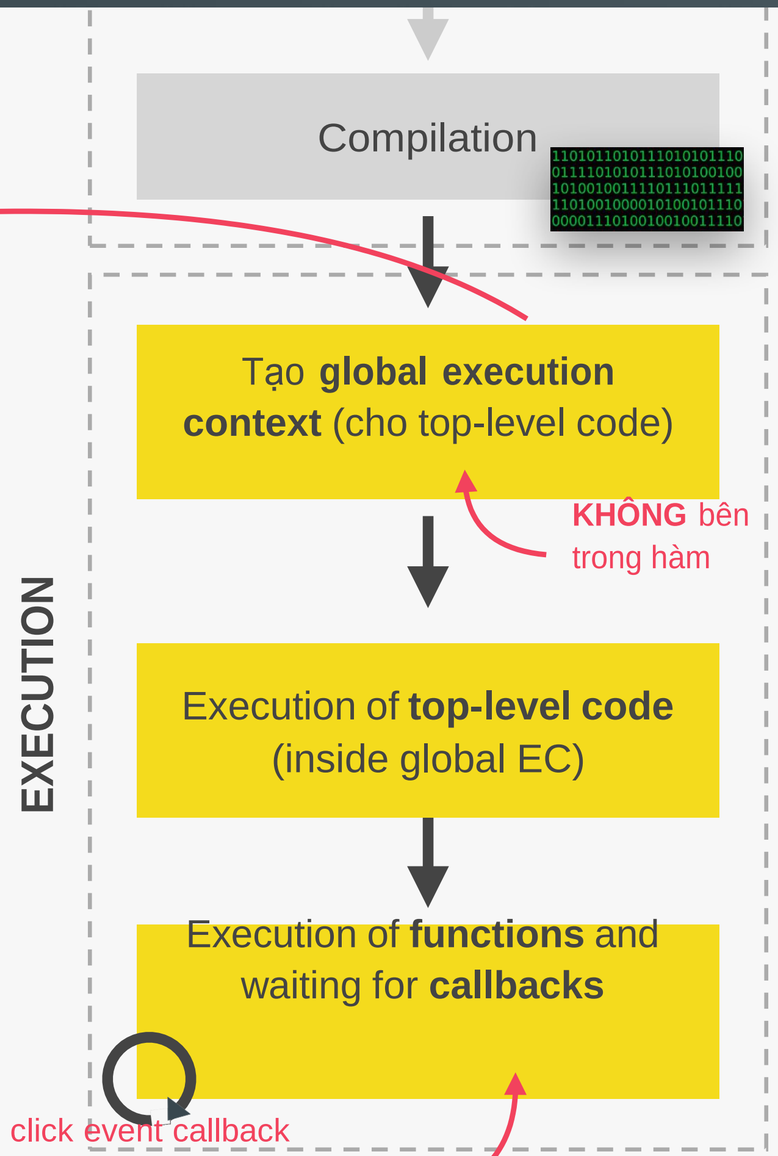


EXECUTION CONTEXT LÀ GÌ?

👉Code mà con người đọc được:

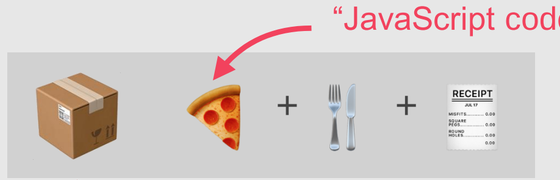
```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second();
  a = a + b;
  return a;
};

function second() {
  var c = 2;
  return c;
}
```

Function body only executed when called!



EXECUTION CONTEXT

Môi trường thực thi một phần của Javascript. Lưu trữ tất cả các thông tin cần thiết để thực thi một số code.



👉 Chính xác thì global execution context (EC): Là context mặc định, được tạo cho những code không ở bên trong bất kỳ hàm nào (top-level).

👉 Một execution context **<u>mỗi hàm</u>**: Mỗi lệnh gọi hàm sẽ có một execution context mới được tạo ra.

Tất cả cùng nhau tạo nên call stack

CHI TIẾT HƠN VỀ EXECUTION CONTEXT

BÊN TRONG EXECUTION CONTEXT CÓ GÌ?

1. Variable Environment
   - 👉 khai báo `let`, `const` và `var`
   - 👉 Hàm
   - 👉 ~~`arguments`~~ object
2. Scope chain
3. ~~**`this`**~~ keyword

KHÔNG có trong hàm mũi tên!

Tạo ra trong giai đoạn tạo, ngay trước khi thực thi

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```

Global

```
name = 'Jonas'
first = <function>
second = <function>
x = <unknown>
```

Code hàm

Cần chạy **`first()`** trước

first()

```
a = 1
b = <unknown>
```

Cần chạy **`second()`** trước

second()

```
c = 2
arguments = [7, 9]
```

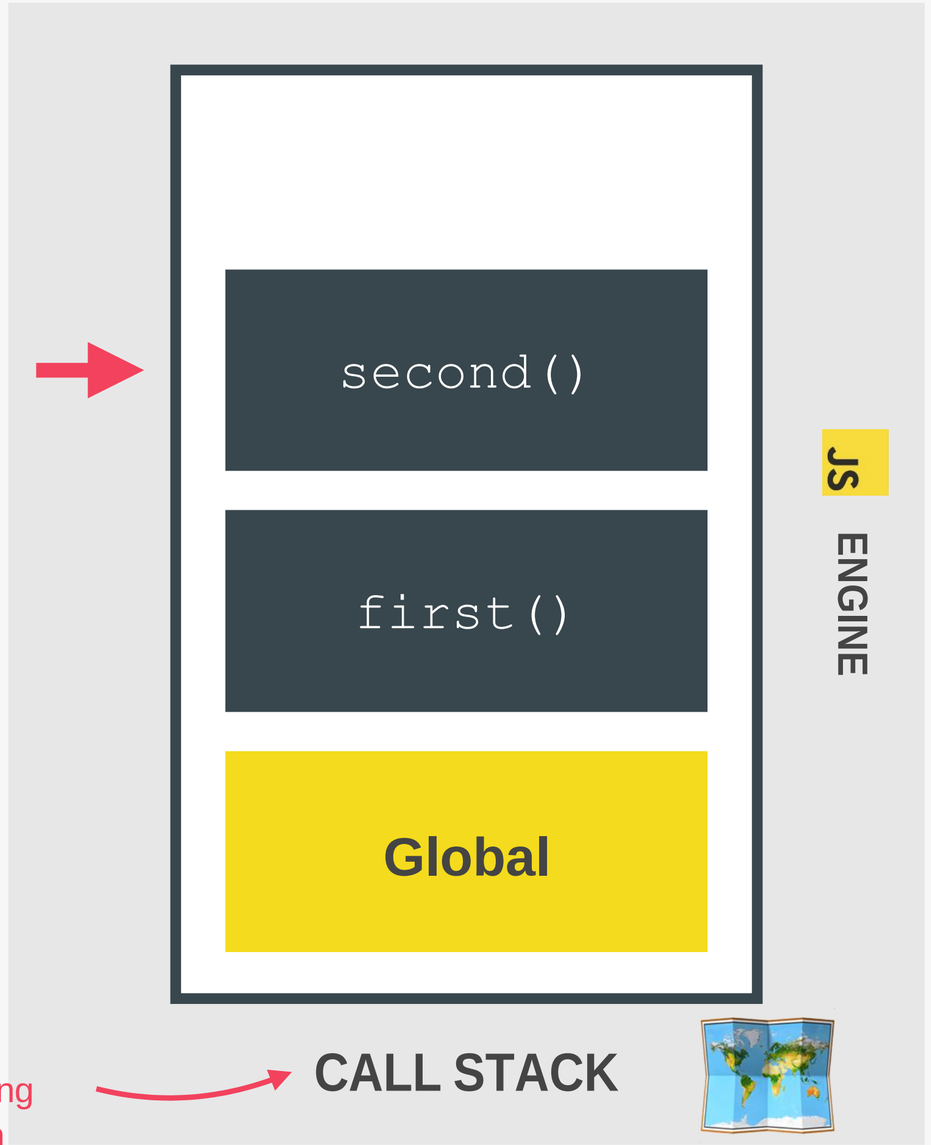
Có sẵn trong tất cả các hàm “thông thường” (không có trong hàm mũi tên)

(Chính xác thì các giá trị sẽ rõ trong quá trình thực thi)

THE CALL STACK

👉 Code đã biên dịch bắt đầu thực thi

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```



"Nơi" xếp chồng các execution context lên trên nhau để biết chúng ta đang ở đâu trong khi thực hiện

KHÁI NIỆM SCOPING VÀ SCOPE TRONG JAVASCRIPT

EXECUTION CONTEXT

- 👉 Variable environment
- 👉 Scope chain
- 👉 `this` keyword

KHÁI NIỆM SCOPE

- 👉 **Scoping:** Cách tổ chức và truy cập các biến. *“Các biến ở đâu?”* hay *“Ở những đâu chúng ta có thể hoặc không thể truy cập một biến nhất định?”;*
- 👉 **Lexical scoping:** Là scoping được kiểm soát bởi việc đặt các hàm và block trong code;
- 👉 **Scope:** Không gian hoặc môi trường khai báo một biến nhất định *(variable trong trường hợp của các hàm)*. Có global scope, function scope và block scope;
- 👉 **Scope của biến:** Vùng code mà một biến nhất định có thể truy cập.

3 LOẠI SCOPE

GLOBAL SCOPE

```
const me = 'Jonas';
const job = 'teacher';
const year = 1989;
```

👉 Bên ngoài bất kỳ hàm hoặc block nào

👉 Các biến được khai báo trong global scope có thể truy cập ở bất cứ đâu

FUNCTION SCOPE

```
function calcAge(birthYear) {
  const now = 2037;
  const age = now - birthYear;
  return age;
}

console.log(now); // ReferenceError
```

👉 Các biến chỉ có thể truy cập bên trong hàm, KHÔNG THỂ truy cập từ bên ngoài

👉 Còn được gọi là local scope

BLOCK SCOPE (ES6)

```
if (year >= 1981 && year <= 1996) {
  const millenial = true;
  const food = 'Avocado toast';
}
```

Example: `if` block, `for` loop block, etc.

```
console.log(millenial); // ReferenceError
```

👉 Các biến chỉ có thể truy cập bên trong block (block scope)

⚠️ TUY NHIÊN, điều này chỉ áp dụng cho các biến **`let`** và **`const`!**

👉 Các hàm cũng bị giới hạn theo block (chỉ trong strict mode)

SCOPE CHAIN

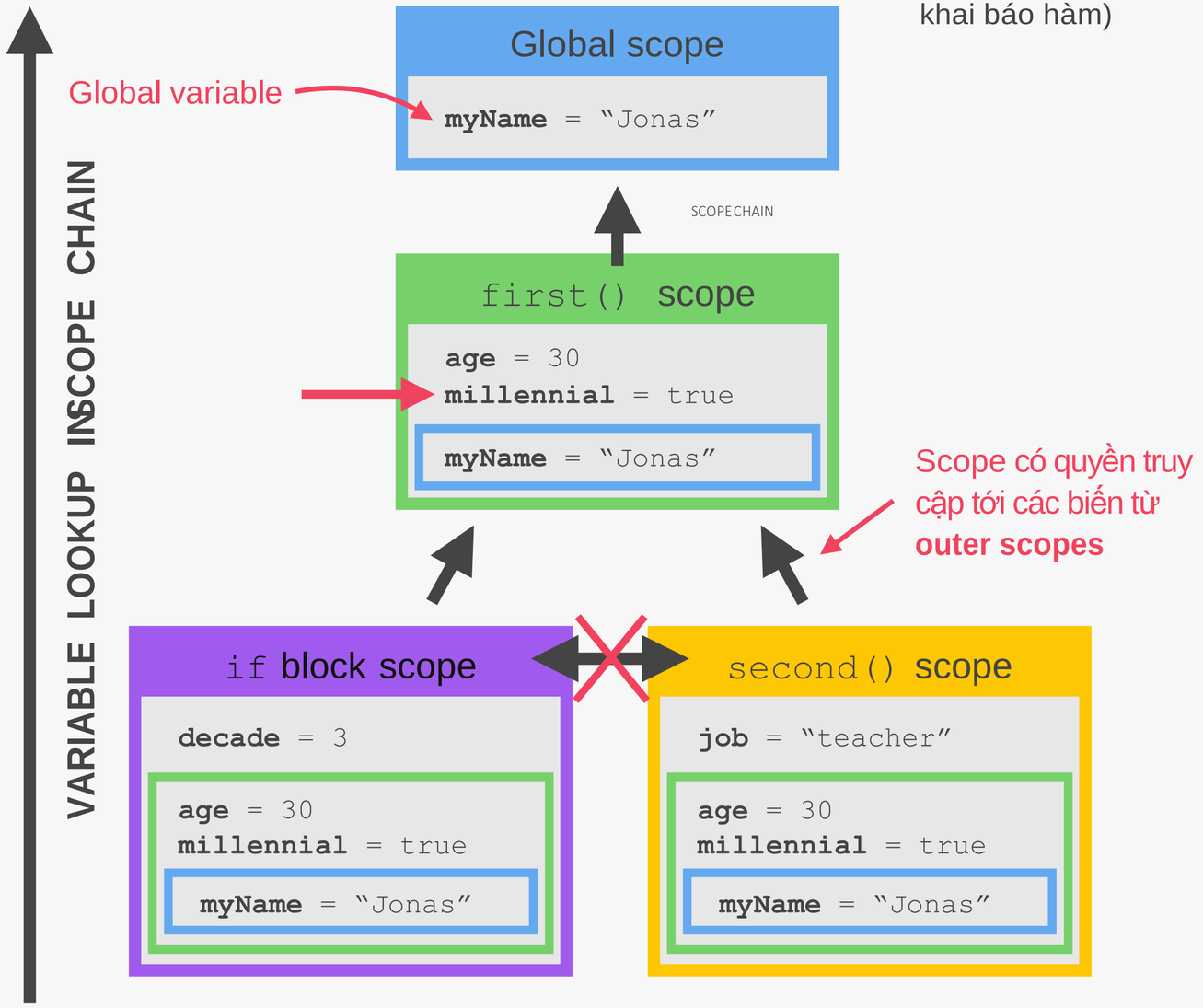
```
const myName = 'Jonas';

function first() {
  const age = 30;

  if (age >= 30) {  // true
    const decade = 3;
    var millenial = true;
  }

  function second() {
    const job = 'teacher';

    console.log(`${myName} is a ${age}-old ${job}`);
    // Jonas is a 30-old teacher
  }

  second();
}

first();
```

let and const are **block-scoped**

var is **function-scoped**

Variables not in current scope

(Chỉ cân nhắc các khai báo hàm)

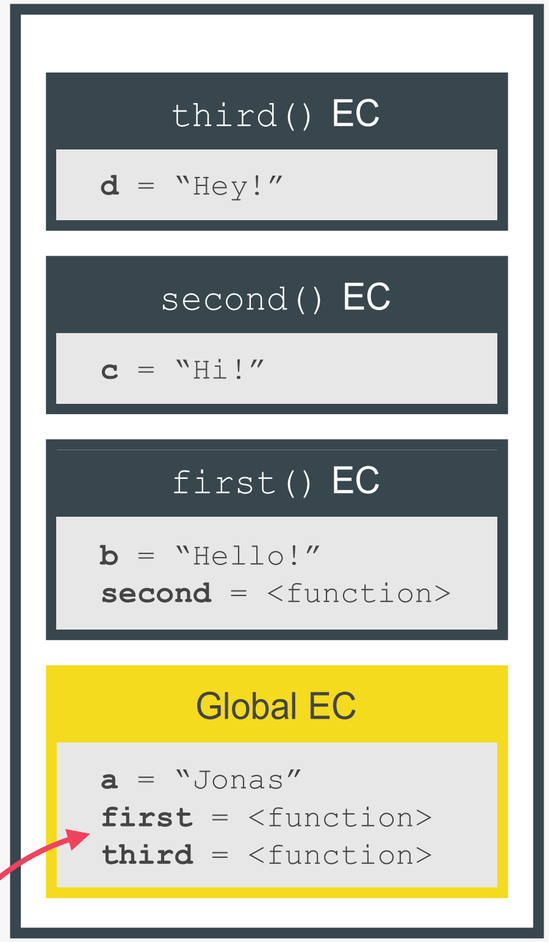


SCOPE CHAIN VỚI CALL STACK

```
const a = 'Jonas';
first();

function first() {
  const b = 'Hello!';
  second();

  function second() {
    const c = 'Hi!';
    third();
  }
}

function third() {
  const d = 'Hey!';
  console.log(d + c + b + a);
  // ReferenceError
}
```

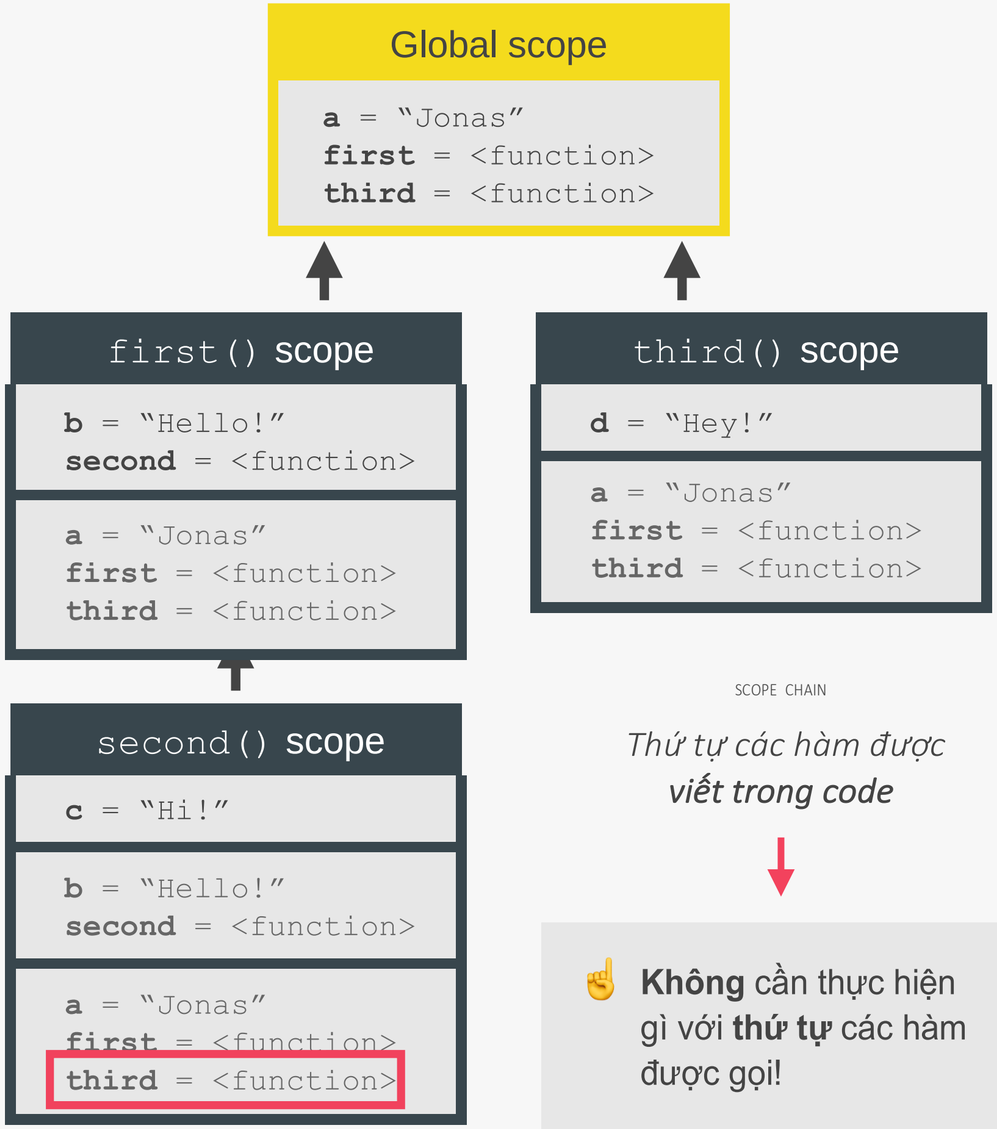
c và b có thể KHÔNG thấy trong third() scope!

Variable environment (VE)



CALL STACK

Thứ tự các hàm được gọi



SCOPE CHAIN

*Thứ tự các hàm được **viết trong code***

☝ **Không** cần thực hiện gì với **thứ tự** các hàm được gọi!

TỔNG KẾT🥳

👉 Scoping đặt câu hỏi *"Các biến ở đâu?"* hoặc *"Ở những đâu chúng ta ta có thể hoặc không thể truy cập một biến nhất định?"*;

👉 Có 3 loại scope trong JavaScript: global scope, function scope và block scope; chỉ biến `let` và `const` thuộc block-scope. Những biến được khai báo với `var` kết thúc với function scope gần nhất;

👉 Trong JavaScript, chúng ta có lexical scoping, là các quy tắc về nơi chúng ta có thể truy cập các biến dựa trên vị trí chính xác trong các hàm và khối code được viết;

👉 Mỗi scope luôn có quyền truy cập vào tất cả các biến từ tất cả outer scope của nó. Đó là scope chain!

👉 Khi một biến không nằm trong scope hiện tại, engine sẽ tra cứu trong scope chain cho đến khi nó tìm thấy biến mà nó đang tìm kiếm. Điều này được gọi là variable lookup;

👉 Scope chain hoạt động theo một chiều: scope sẽ không bao giờ có quyền truy cập vào các biến của inner scope;

👉 Scope chain trong một scope nhất định tương đương với việc cộng tất cả các variable environment của tất cả các parent scope;

👉 Scope chain không liên quan gì đến thứ tự các hàm được gọi. Nó hoàn toàn không ảnh hưởng đến scope chain!