Bài 2
Xử lý bất đồng bộ 1

Module: Web backend development with NodeJS

Mục tiêu

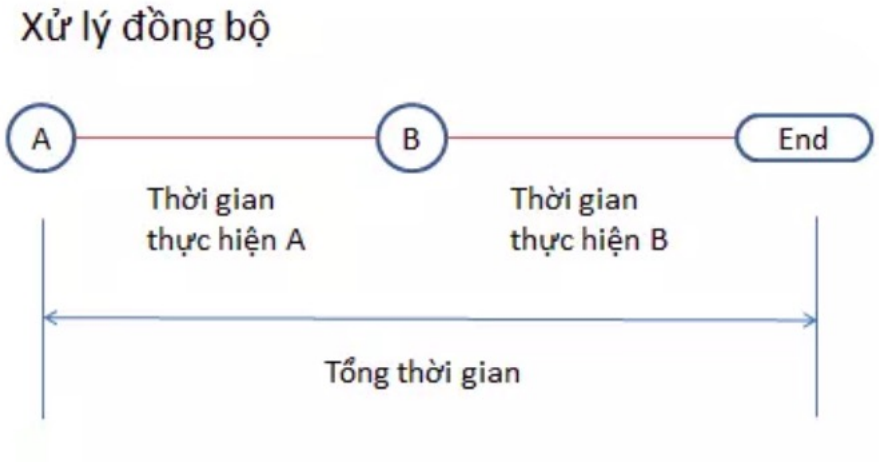
- Trình bày được khái niệm lập trình bất đồng bộ
- Trình bày được mô hình Event Loop trong JS Engine
- Liệt kê được các kĩ thuật xử lý bất đồng bộ trong JavaScript
- Trình bày được khái niệm callback
- Trình bày được vấn đề callback-hell trong JavaScript
- Trình bày được ý nghĩa của Promise trong xử lý bất đồng bộ

Đồng bộ và bất đồng bộ

Thảo luận

Lập trình đồng bộ

- Đồng bộ là cơ chế tuần tự, các dòng code chạy tuần tự từ trên xuống dưới
- Các hàm gọi nhau sẽ đợi kết quả của hàm khác trả về rồi mới chạy tiếp



Bất đồng bộ

- Code thực thi không tuần tự.
- Nhiều lệnh chạy cùng một lúc.

```
function asyncFunction(callback) {
   console.log("Start");
   setTimeout(() => {
      callback();
   }, 1000);
   console.log("Waiting");
}

let printEnd = function() {
   console.log("End");
}

asyncFunction(printEnd)
```

Kết quả:

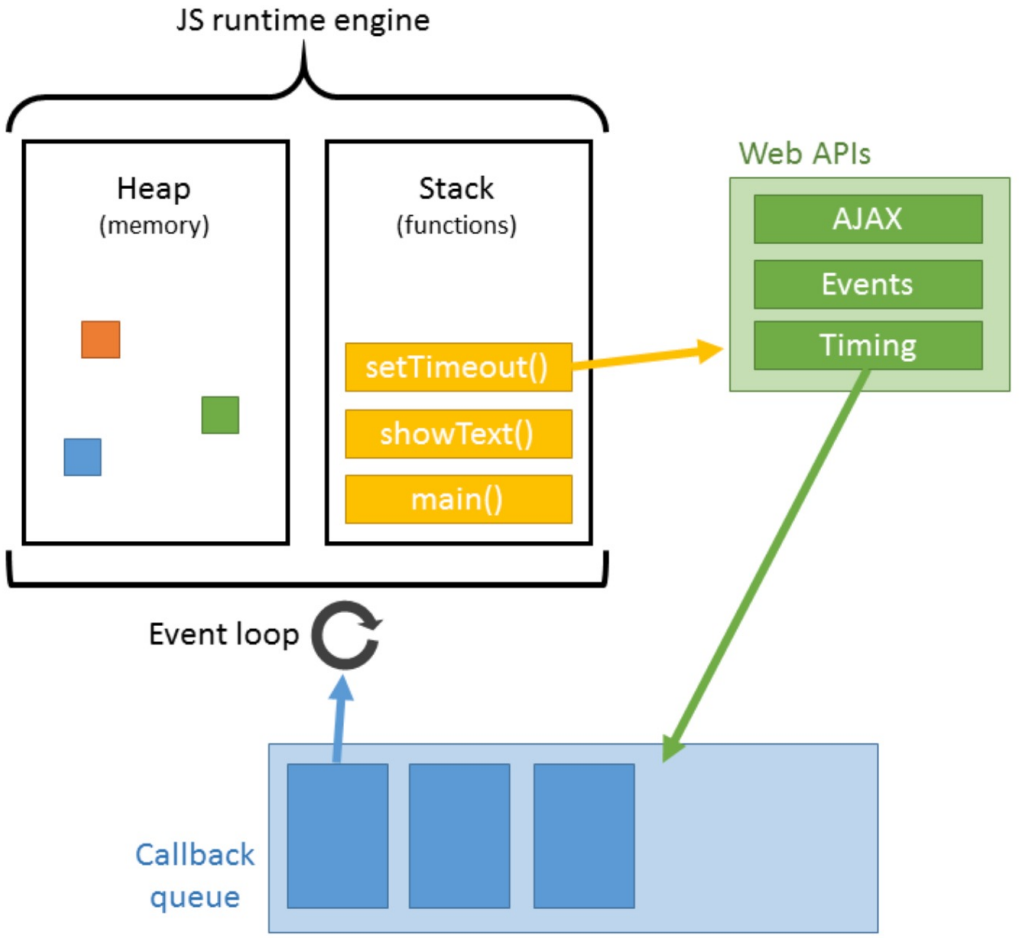
```
Start
Waiting
End
```

Mô hình Event Loop

Stack funtions
Heap memory
Event Loop

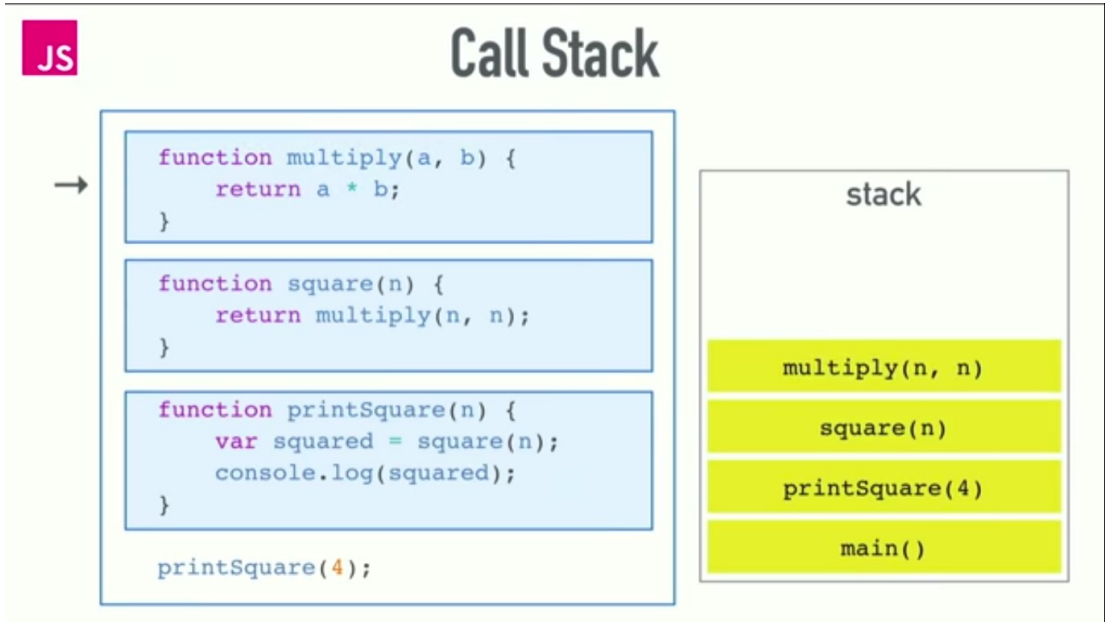
Mô hình Event Loop

- Cơ chế giúp Javascript có thể thực hiện nhiều thao tác cùng một lúc.
- Vòng lặp vô hạn theo dõi các Event



Stack function

- Stack là một vùng nhớ đặc biệt trên con chip máy tính phục vụ cho quá trình thực thi các dòng lệnh.
- Một hàm chỉ được lấy ra khỏi stack khi nó hoàn thành và return.



Bộ nhớ Heap

- Heap là vùng nhớ được dùng để chứa kết quả tạm thời phục vụ cho việc thực thi các hàm trong stack
- Heap chứa các đối tượng hoặc funtions

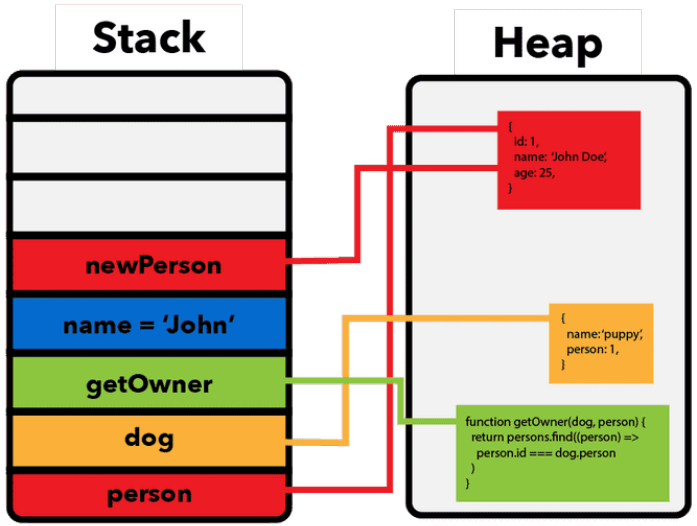
```
const person = {
  id: 1,
  name: 'John',
  age: 25,
}

const dog = {
  name: 'puppy',
  personId: 1,
}

function getOwner(dog, persons) {
  return persons.find((person) =>
    person.id === dog.person
  )
}

const name = 'John';

const newPerson = person;
```



Cách hoạt động Event Loop

- Khi hàm main được chạy thì các đoạn code trong main sẽ được thực thi. Nó sẽ lần lượt đẩy các hàm vào bên trong call stack theo nguyên tắc LIFO.
- Các hàm hay tác vụ liên quan đến Events (click, change, listener, ...), AJAX (Call APIs), Timing (setTimeout, setInterval) sẽ được đẩy từ call stack sang Web APIs.
- Còn lại thì sẽ được thực thi trong call stack đến khi nào xong thì pop ra cho hàm bên dưới được thực thi.

Cách hoạt động Event Loop

- Web APIs sẽ tận dụng các nhân của thiết bị để xử lý riêng biệt các tác vụ này. Sau khi hoàn tất thì Web APIs sẽ trả về một callback và đẩy nó vào trong Callback Queue.
- Event loop sẽ theo dõi Callback Queue và Call stack. Bất kể khi nào mà Call stack trống (tất cả các hàm được pop ra) thì nó sẽ lấy callback đầu tiên ở trong Callback Queue và đưa vào trong Call Stack để tiếp tục thực thi.

Xử lý bất đồng bộ

Callback
Promise

Sử dung Callback

- **Callback** là một hàm sẽ được thực hiện sau khi một hàm khác đã thực hiện xong.
- Là một cách để đảm bảo code nhất định không thực thi cho đến khi code khác thực hiện xong.

```
function first(){
    setTimeout( handler: function(){
        console.log("Đang làm bài tập!");
    }, timeout: 5000 );
}
function second(){
    console.log("Đã làm xong!");
}

first();
second();
```

```
Đã làm xong!
Đang làm bài tập!
```

Callback

```
function first(callback){
    setTimeout( handler: function(){
        console.log("Đang làm bài tập!");
        callback();
    }, timeout: 5000 );
}
function second(){
    console.log("Đã làm xong!");
}

first(second);
```

```
Đang làm bài tập!
Đã làm xong!
```

Callback hell

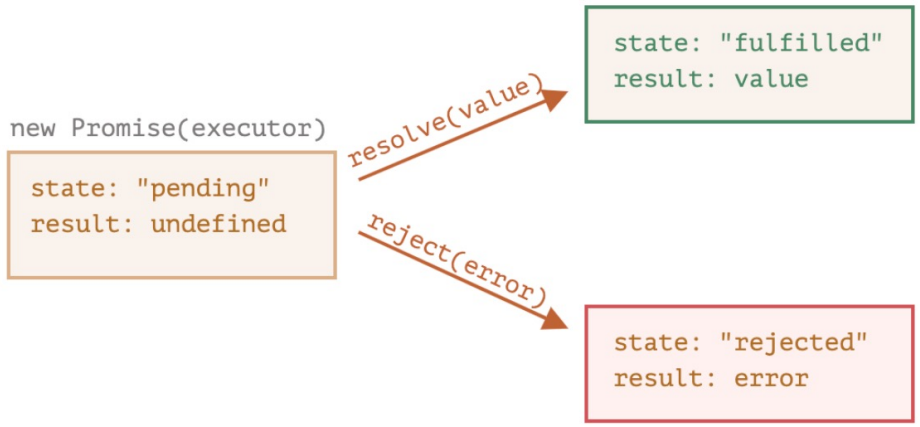
- Khi xử lý bất động bộ bằng callback, có những trường hợp chúng ta quá lạm dụng callback để gọi lại giá trị hàm sẽ khiến mọi thứ trở nên tồi tệ và nhìn code sẽ rất rối.
- Có thể sử dụng Promise để tránh callback hell

Demo callback

- GV demo callback

Promise

- Là một kỹ thuật nâng cao giúp xử lý vấn đề bất đồng bộ hiệu quả hơn
- Đối tượng Promise đại diện cho việc hoàn thành (hoặc thất bại) cuối cùng của một hoạt động không đồng bộ và giá trị kết quả của nó.



Tạo mới Promise

- Cú pháp

```
let promise = new Promise(function(resolve, reject){

});
```

- Trong đó
  - **resolve** là một hàm callback xử lý cho hành động thành công.
  - **reject** là một hàm callback xử lý cho hành động thất bại.

Ví dụ Promise

- Xem ví dụ sau

```
let happyHanding = (message) => {
    return new Promise( executor: (resolve, reject) => {
        setTimeout( handler: () => {
            if (message === 'Yes') {
                resolve( value: 'Em đồng ý')
            } else {
                reject(new Error('Không đồng ý'))
            }
        }, timeout: 5000)
    });
}

happyHanding('Yes').then(result => {
    console.log(result)
})
```

Demo Promise

- GV demo Promise

Chú ý khi sử dung Promise

- Khi sử dung Promise chúng ta cần tránh:
  - ”Kim tự tháp” Promise
  - Không sử dung vòng lặp với Promise
  - Quên không thêm .cath khi sử dụng Promise

Tóm tắt bài học

- Javascript là ngôn ngữ lập trình đơn luồng tại một thời điểm chỉ có một tác vụ được chạy.
- Các tác vụ bất đồng bộ có thể được xử lý bằng Callback, Promise hay Asyn/Await
- Event loop là vòng lặp vô hạn, cơ chế giúp xử lý nhiều thao tác một cùng
- Khi sử lý bất đồng bộ cần chú ý để tránh rơi vào Callback hell

Hướng dẫn

- Hướng dẫn làm bài thực hành và bài tập
- Chuẩn bị bài tiếp: Xử lý bất đồng bộ 2