



Raising the bar

# **TYPESCRIPT FUNDAMENTALS**

# Mục tiêu

- Phân biệt được TypeScript với JavaScript
- Cài đặt môi trường và khởi chạy thành công project mẫu
- Nắm được cách để transpile từ TypeScript sang JavaScript
- Sử dụng được các cấu trúc, cú pháp của TypeScript

# Điều kiện tiên quyết

- Nắm được lập trình ứng dụng với JavaScript
- Có kiến thức cơ bản về HTML, CSS

# TypeScript là gì?

# TypeScript

- TypeScript (TS) là một superset của JavaScript(JS).
- Được phát triển bởi Microsoft.
- Có thể transpile thành code JS để chạy trên môi trường của Browser hoặc Nodejs.
- Tuân thủ chặt chẽ specs mà ECMAScript (ES) đề ra, do đó tất cả những đoạn code hợp lệ trong JS thì sẽ hợp lệ ở TS.
- Support rất nhiều tính năng nâng cao trong các bản ES mới nhất.
- TypeScript đang được hỗ trợ rất mạnh, từ cộng đồng, IDE/Editor, đến các library/framework.

# Lợi ích của TypeScript

- Support rất nhiều tính năng nâng cao trong các bản ES mới nhất.
- TS có một hệ thống type rất tốt, giúp người sử dụng có thể dễ dàng biết trước được các bug có thể phát sinh.
- Dễ dàng để gợi ý code trong các IDE/Editor.
- Một dạng self-documenting cho code.
- Tạo sự nhất quán trong coding style cho các team đồng người.
- Support Generic.

# Cài đặt TypeScript

# Node.js



The screenshot shows the Node.js website homepage. At the top is a dark navigation bar with the Node.js logo and links for HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, NEWS, and FOUNDATION. The main content area has a white background. It starts with a description of Node.js as a JavaScript runtime built on Chrome's V8 JavaScript engine. Below this is a banner for 'JS Interactive' with details about a conference in October 2018. A green banner highlights 'August 2018 security releases available, upgrade now'. The 'Download for macOS (x64)' section features two green buttons: '8.11.4 LTS Recommended For Most Users' and '10.9.0 Current Latest Features'. Below these are links for 'Other Downloads', 'Changelog', and 'API Docs'. A link to the 'Long Term Support (LTS) schedule' is also present. At the bottom, there is a sign-up link for 'Node.js Everywhere', the official monthly newsletter.

node

HOME | ABOUT | DOWNLOADS | DOCS | GET INVOLVED | SECURITY | NEWS | FOUNDATION

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

**JS Interactive** Join us at #jsinteractive  
October 10-12, 2018  
Vancouver, Canada

August 2018 security releases available, upgrade now

Download for macOS (x64)

**8.11.4 LTS**  
Recommended For Most Users

**10.9.0 Current**  
Latest Features

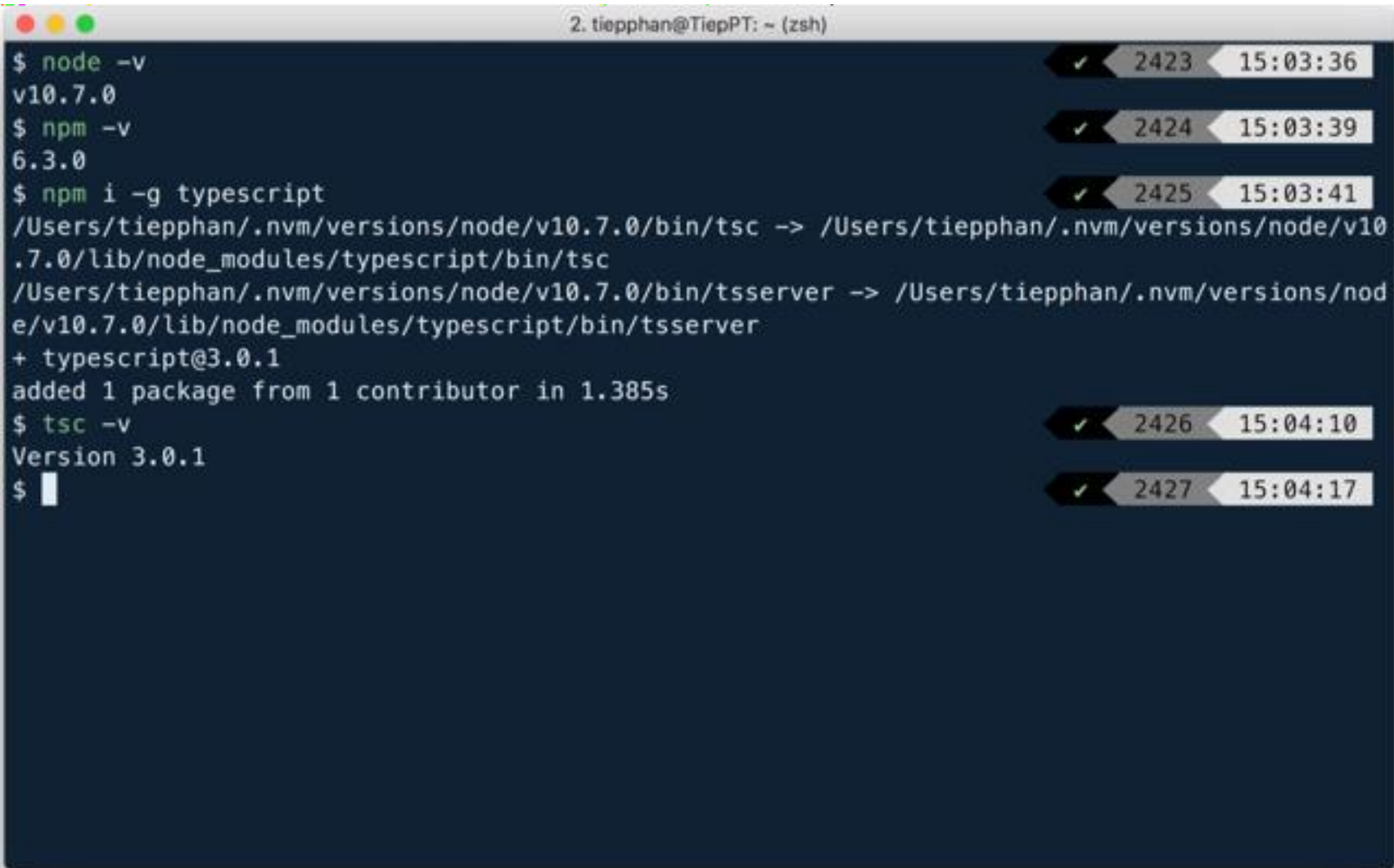
Other Downloads | Changelog | API Docs    Other Downloads | Changelog | API Docs

Or have a look at the Long Term Support (LTS) schedule.

Sign up for [Node.js Everywhere](#), the official Node.js Monthly Newsletter.



# Node.js & TypeScript



A terminal window titled "2. tiepphan@TiepPT: ~ (zsh)" displays the following commands and their outputs. Four orange arrows point to the first four lines of the terminal output:

```
$ node -v
v10.7.0
$ npm -v
6.3.0
$ npm i -g typescript
/Users/tiepphan/.nvm/versions/node/v10.7.0/bin/tsc -> /Users/tiepphan/.nvm/versions/node/v10.7.0/lib/node_modules/typescript/bin/tsc
/Users/tiepphan/.nvm/versions/node/v10.7.0/bin/tsserver -> /Users/tiepphan/.nvm/versions/node/v10.7.0/lib/node_modules/typescript/bin/tsserver
+ typescript@3.0.1
added 1 package from 1 contributor in 1.385s
$ tsc -v
Version 3.0.1
$
```

On the right side of the terminal, there are four status bars, each with a checkmark, an ID number, and a timestamp:

- ✓ 2423 15:03:36
- ✓ 2424 15:03:39
- ✓ 2425 15:03:41
- ✓ 2426 15:04:10
- ✓ 2427 15:04:17

# Transpile từ TS sang JS

# Thực hiện

- Bước 1: Tạo file main.ts, sau đó điền code mẫu ở page sau.
- Bước 2: Di chuyển terminal/cmd vào thư mục chứa file main.ts
- Bước 3: Chạy lệnh `tsc main.ts`
- Bước 4: Kiểm tra file JS được transpile ra.
- Bước 5: Chạy lệnh `node main.js`(js, không phải ts) để nhìn thấy kết quả.

# Biến và Kiểu dữ liệu

# Biến

## **var**

- Function scope.
- Có thể khai báo lại biến trong cùng 1 scope với từ khóa var.
- var thì hoisted lên đầu scope.

## **let/const**

- Block scope.
- Không thể khai báo lại biến trong cùng 1 scope với let hoặc const.
- Khi khai báo biến với let hoặc const thì biến đó không được hoisted lên đầu scope, dẫn đến muốn dùng biến thì bắt buộc phải được khai báo trước.

# Scope

```
function main() {  
  console.log( 'START' );  
  if(true) {  
    var lang = 'vi';  
    let target = 'en-us';  
    console.log('inside block');  
    console.log(target);  
  }  
  console.log(lang);           // OK  
  console.log(target);        // ERROR: [ts] Cannot find name 'target'.  
}
```

# Re-declare variable

```
function main() {  
  var x = 5;  
  console.log(x);  
  var x = 10;           // OK  
  console.log(x);  
  
  let y = 55;  
  console.log(y);  
  let y = 100;  
  // ERROR: [ts] Cannot redeclare block-scoped variable 'y'.  
  console.log(y);  
}
```

# Hoisting

```
function main() {  
  console.log(x); // OK: x is undefined  
  var x = 5;  
  
  console.log(y);  
  // ERROR: [ts] Block-scoped variable 'y' used  
  // before its declaration.  
  let y = 10;  
}
```

Biến y không tồn tại ở khu vực này



# const

- const không bị hoisted.
- Phải được khởi tạo giá trị ngay lúc khai báo.
- Không được phép gán sang một “vùng nhớ” khác.
- Nhưng vẫn có thể thay đổi nội tại: ví dụ const cho 1 object, và object đó được thực hiện gán thêm 1 property mới mà không làm thay đổi vùng nhớ thì hoàn toàn hợp lệ.

# Tổng kết về khai báo biến

	var	let	const
Được phép gán	Yes	Yes	No
Scope	function	block	block
Hoisted	Yes	No	No

# Kiểu dữ liệu

## JavaScript

- null
- undefined
- boolean
- number
- string
- symbol
- Object

## TypeScript

- Tất cả các kiểu dữ liệu của JS
- enum
- interface
- any (có thể lưu trữ bất kỳ kiểu dữ liệu nào. Nên hạn chế dùng.)

# Kiểu dữ liệu

```
let message: string;
```

```
let total: number = 100;
```

Typeinference

```
let isProduction = true;
```

```
let prices: Array<number> = [120, 88, 60];
```

```
let languages: string[] = ['vi', 'en-us'];
```

```
let now = new Date();
```

```
let unknown: any;
```

# Kiểu dữ liệu

```
enum Direction {  
    UP,  
    DOWN  
  
    ,  
    LEFT,  
    RIGHT  
};
```

```
function log (msg: string): void {  
    console.log(msg)  
}
```

# Kiểu dữ liệu

```
interface IPost {  
    id: string;  
    title: string;  
    body?: string;  
}
```

Optional



# Kiểu dữ liệu

```
isProduction = false;
```

```
unknown = Direction.UP;  
unknown = 'changed';
```

Can hold any type

```
const post: IPost = {  
};
```

Error

```
message = 50;
```

Error

# Kiểu dữ liệu

```
function getPost(postId: string): IPost {  
    // do something to retrieve post  
    return {  
        id: postId,  
        title: 'Post Title',  
        body: 'Post Body',  
        extra: 'data'  
    } as IPost;  
}
```

Param with type

Return type

Ép kiểu tường minh



# Cấu trúc điều khiển

# if-else

```
let count = 50;  
if (count > 0) {  
    count--;  
} else {  
    count = 0;  
}  
console.log(count);  
// Output?
```

# for

```
const keys = "abcdef";  
for (let idx = 0; idx < keys.length; ++idx) {  
    console.log(keys[idx]);  
}
```

// Output?

# while

```
let idx = 0;  
while (idx < keys.length) {  
    console.log(keys[idx]);  
    ++idx;  
}
```

// Output?

# do-while

```
let idx = 0;  
do {  
    console.log(keys[idx]);  
    ++idx;  
} while (idx < keys.length);
```

// Output?

# for-of

```
for (const item of keys) {  
    console.log(item);  
}
```

// Output?

# for-in

```
const user = {  
  name: 'Bob',  
  age: 55  
};  
for (const key in user) {  
  console.log(` ${key}: ${user[key]} `);  
}
```

// Output?

# Mảng



# Khởi tạo mảng

```
const list: number[] = [1, 2, 3];
```

```
const categories: Array<string> =  
  ['Sport', 'IT', 'Car'];
```



Generic programming

# Thao tác với mảng

```
console.log('list');  
list.forEach((num) =>  
    console.log(num.toFixed(2))  
);
```

num có type number

```
console.log('categories');  
categories.forEach((str) =>  
    console.log(str.includes('a'))  
);
```

str có type string

# Thao tác với mảng

```
// convert mảng từ dạng này sang dạng khác.  
const listSquare = list.map(num => num * num);  
console.log(listSquare)  
// Output: [1, 4, 9]
```

```
// lọc các phần tử thỏa mãn  
const result = categories.filter(str => str.length > 2)  
console.log(result);  
// Output: ['Sport', 'Car']
```

# Hàm

# Tổng quan về hàm

- Hàm là một tập các câu lệnh để xử lý một task vụ hoặc tính toán giá trị nào đó.
- Để sử dụng hàm chúng ta phải định nghĩa nó trước khi có thể gọi đến nó.

# Khai báo hàm

- Để khai báo một hàm cần phải có 3 yếu tố sau:
  - Tên của hàm
  - Danh sách các tham số của hàm được khai báo trong ngoặc tròn.
  - Phần thân hàm được khai báo trong {}.
- Hàm khai báo sẽ được đưa lên đầu tiên.

# Khai báo và gọi hàm

```
function square(num: number): number {  
    return num * num;  
}
```

// Hoặc

```
const square = function (num: number): number {  
    return num * num;  
}
```

```
console.log(square(5));
```

// Output: 25

# Higher-Order Function

- Một hàm nếu nhận tham số là một hàm khác(callback).
- Một hàm return về một hàm khác.



# Higher-Order Function

```
function add(a: number): Function {  
    return function(b: number): number {  
        return a + b;  
    }  
}
```

Đây là một function được  
return về sau khi lời gọi hàm  
add kết thúc

```
const addWith5 = add(5);  
console.log(addWith5(3));  
console.log(addWith5(15));
```

addWith5 là một hàm, nên  
chúng ta có thể gọi nó

# Object

# Khởi tạo và sử dụng Object

```
interface IUser {  
    name: string;  
    age: number  
}
```

```
let customer: IUser = {  
    name: 'Bob',  
    age: 50  
};
```

customer có dạng là tương  
ứng interface IUser

```
console.log(customer);
```

# Khởi tạo và sử dụng Object

```
customer = {  
    name: 'Anna'  
};  
/*
```

Error: Type '{ name: string; }' is not assignable  
to type 'IUser'.

Property 'age' is missing in type '{ name: string; }'.  
\*/

# Class

```
class Shape {  
    public x: number;  
    public y: number;  
    constructor(x: number, y: number) {  
        this.x = x;  
        this.y = y;  
    }  
    toString(): string {  
        return `(x: ${this.x}, y: ${this.y})`;  
    }  
}
```

# Class

```
interface IArea {  
    area(): number;  
}  
class Rect extends Shape implements IArea {  
    constructor(x: number, y: number,  
        public width: number,  
        public height: number  
    ) {  
        super(x, y);  
    }  
    area(): number {  
        return this.width * this.height;  
    }  
}
```

Định nghĩa interface, interface này yêu cầu các class/object phải có kèm hàm area có return type là number

Đây là cách viết ngắn gọn của TypeScript hỗ trợ, khi bạn có property của class có trùng tên với tham số truyền vào của hàm tạo, và bạn muốn thực hiện lệnh gán như ở class Shape trước đó

Phần cài đặt của interface IArea

# Class

```
const rect = new Rect(5, 5, 10, 20);  
console.log(rect.toString());  
console.log(rect.area());  
/*
```

Output:

(x: 5, y: 5)

200

```
*/
```

# Promise



# Promise

Promises là một giải pháp thay thế cho các callback để cung cấp kết quả của việc thực hiện các tính toán bất đồng bộ

```
const p = new Promise( /* executor */  
function(resolve, reject) {  
  // statements  
} );
```

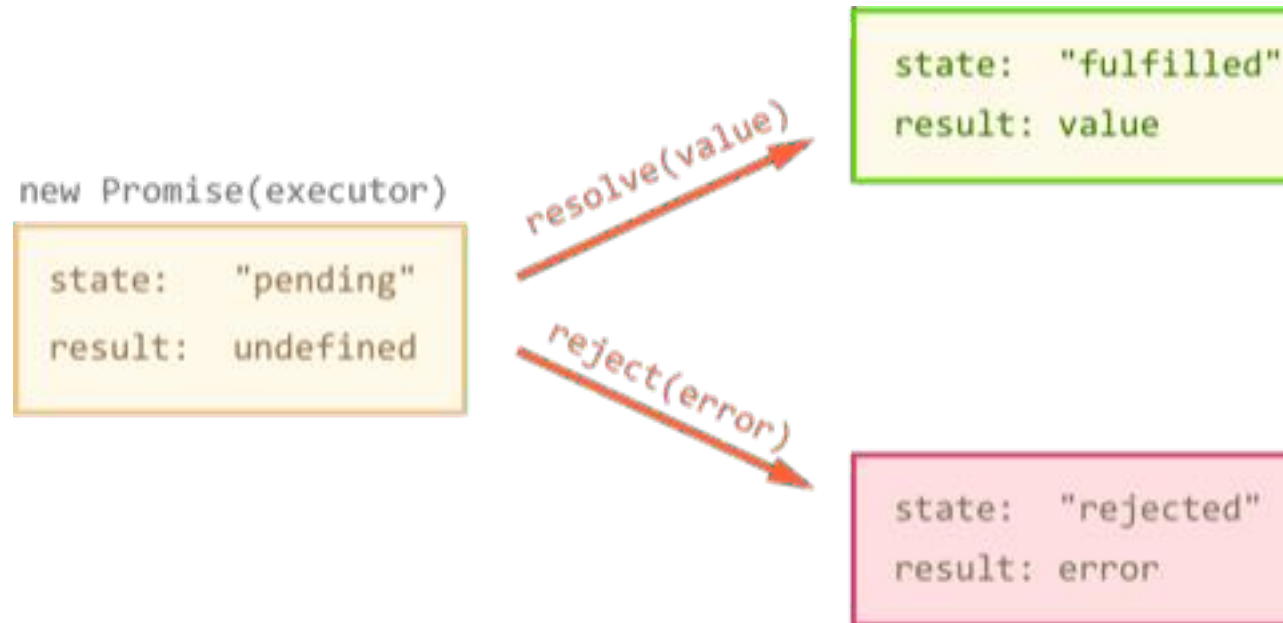
# Promise

- Promise luôn ở 1 trong 3 trạng thái dưới đây
  - Pending: trước khi có kết quả
  - Nếu có kết quả trả về promise sẽ ở trạng thái fulfilled
  - Nếu xảy ra lỗi promise sẽ ở trạng thái rejected
- Promise được thực hiện nếu “thing are done” dù nó ở trạng thái fulfilled hay rejected.
- Promise được thực hiện 1 lần duy nhất

# Promise

- Thenable là một phương thức ghi nhận kết quả của trạng thái (thành công hoặc thất bại) mà ta khai báo ở `Reject` và `Resolve`. Nó có hai tham số truyền vào là 2 callback function. Tham số thứ nhất xử lý cho `Resolve` và tham số thứ 2 xử lý cho `Reject`.

# Promise



# Promise

- Tạo một promise

```
const p = new Promise(  
  function (resolve, reject) { // (A)  
    ...  
    if ( ... ) {  
      resolve(value); // success  
    } else {  
      reject(reason); // failure  
    }  
  });
```

# Promise

- Thực hiện một promise

```
promise
  .then(
    value => { /* fulfillment */ },
    error => { /* rejection */ }
  )
  .catch(error => { /* rejection */ });
```

```
function httpGet(url: string) :Promise<any> {  
    return new Promise(  
        function (resolve, reject) {  
            const request = new XMLHttpRequest();  
            request.onload = function () {  
                if (this.status === 200) {  
                    // Success  
                    resolve(this.response);  
                } else {  
                    // Something went wrong (404 etc.)  
                    reject(new Error(this.statusText));  
                }  
            };  
            request.onerror = function () {  
                reject(new Error('XMLHttpRequest Error: ' + this.statusText));  
            };  
            request.open('GET', url);  
            request.send();  
        });  
}
```

```
httpGet(  
  'https://api.github.com/search/repositories?q=angular'  
)  
  .then(  
    function (value) {  
      console.log('Contents: ' + value);  
    },  
    function (reason) {  
      console.error('Something went wrong', reason);  
    }  
  );
```



# Promise

- **Tạo promises bằng những cách khác**
  - `Promise.resolve()`
  - `Promise.reject()`

# Promise

## • ChainingPromises

```
function parseResponse(value: string) {  
  try {  
    return JSON.parse(value);  
  } catch (_) {  
    return value;  
  }  
}  
  
httpGet('https://api.github.com/search/repositories?q=angular')  
  .then(parseResponse)  
  .then(data => console.log(data))  
  .catch(function(reason) {  
    console.error('Something went wrong', reason);  
  });
```

# Promise

- Chaining Promises

The diagram illustrates promise chaining with the code `P.then(function () { return R }) .then(onFulfilled, onRejected)`. A bracket under the first `.then()` call is labeled `Q`, indicating that `Q` is the promise returned by this call. An arrow labeled "insert here" points to the space between the two `.then()` calls, indicating where a new promise can be inserted into the chain.

```
P.then(function () { return R }) .then(onFulfilled, onRejected)
```

# Promise

- Promise anti-pattern:nested

```
asyncFunc1()  
  .then(function (value1) {  
    asyncFunc2()  
      .then(function (value2) {  
        ...  
      });  
  });  
})
```

# Promise

- Promise anti-pattern: nested(fixed)

```
asyncFunc1()  
  .then(function (value1) {  
    return asyncFunc2();  
  })  
  .then(function (value2) {  
    ...  
  });
```

# Promise

- Promise anti-pattern: nested

```
function asyncFunc1()    {  
    return new Promise(function(resolve, reject){  
        asyncFunc2().then(function(data) {  
            // extra work with data  
            resolve(data);  
        }).catch(reject);  
    });  
}
```

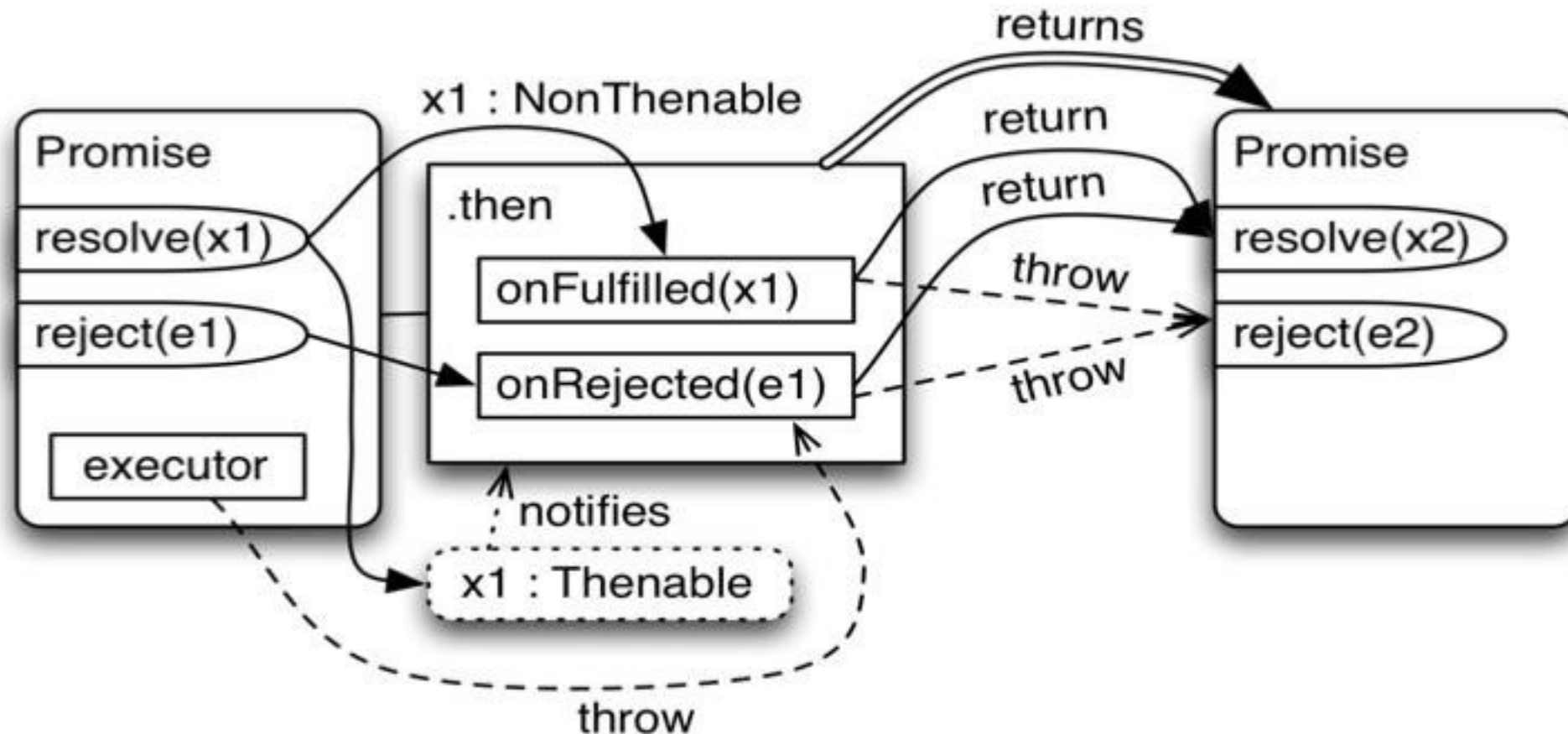
# Promise

- Promise anti-pattern: nested(fixed)

```
function asyncFunc1() {  
    return asyncFunc2().then(function(data) {  
        // extra work with data  
        return data;  
    });  
}
```

# Promise

- then always return a promise







R a i s i n g   t h e   b a r