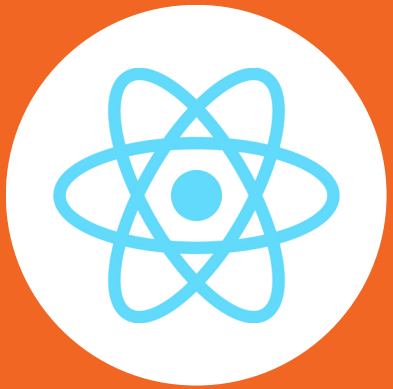
FPT Education

FPT POLYTECHNIC



# LẬP TRÌNH ĐA NỀN TẢNG VỚI REACT NATIVE

## BÀI 2: ÔN TẬP KIẾN THỨC JAVASCRIPT NÂNG CAO

## PHẦN 1: ĐIỀU KIỆN, THAO TÁC VỚI OBJECT VÀ MẢNG, VÒNG LẶP TRONG JS

# Mục tiêu

- Ôn lại điều kiện trong javascript
- Ôn lại các phương thức thao tác với mảng thường gặp
- Ôn lại vòng lặp trong javascript
- Ôn lại các phương thức thao tác với object thường gặp
- Ôn lại kiến thức xử lý bất đồng bộ
- Bắt lỗi và xử lý bằng try...catch

# Điều kiện

- Trong **Javascript**, để xử lý điều kiện chúng ta có thể dùng **if else** hoặc **switch case**

```
function testNum(a) {
  let result;
  if (a > 0) {
    result = 'Số dương';
  } else {
    result = 'Không phải số dương';
  }
  return result;
}

console.log(testNum(-5));
// Kết quả: "Không phải số dương"
```

## if...else

Câu lệnh **If** sẽ thực thi khi điều kiện nó là **true**, câu lệnh **else** sẽ thực thi khi điều kiện là **false**

# Điều kiện

- Bạn có thể thay thế câu điều kiện **if else** bằng một cách ngắn gọn hơn bằng câu điều kiện **ternary**
  - Toán tử có điều kiện (**ternary**) là toán tử **JavaScript** duy nhất có ba toán hạng: một điều kiện theo sau là dấu chấm hỏi (**?**), sau đó là biểu thức để thực thi nếu điều kiện là **true** theo sau là dấu hai chấm (:) và cuối cùng là biểu thức để thực thi nếu điều kiện là **false**. Toán tử này thường được sử dụng thay thế cho **if...else**

# Điều kiện

```
function getFee(isMember) {
  return isMember ? '$2.00' : '$10.00';
}

console.log(“Phi thanh vien la ”, getFee(true));
// Kết quả: "Phi thanh vien la $2.00"

console.log(“Phi thanh vien la ”, getFee(false));
// Kết quả: "Phi thanh vien la $10.00"

console.log(“Phi thanh vien la ”,  getFee(null));
// Kết quả: "Phi thanh vien la $10.00"
```

## Điều kiện (**ternary**)

Chúng ta có hàm **getFree** để thực hiện tính toán phí thành viên**. isMember** là điều kiện cần so sánh. Nếu **isMember** là **true** thì hàm **getFee** sẽ return ra “**$2.00**” và ngược lại

# Các điều kiện khác

- Dưới đây, sẽ là vài tip xử lý câu điều kiện ngắn gọn hơn.

```
const getName = (name) => {
 return name || 'Xin loi, Ban chua nhap ten';
}
console.log("Ten cua ban la ", getName())
//Ket qua: Ten cua ban la Xin loi, Ban chua nhap ten

console.log("Ten cua ban la ", getName("Nam"))
//Ket qua: Ten cua ban la Nam
```

Khi bạn gọi hàm getName mà không truyền tham số vào, thì tham số **name** sẽ là **undefine. Undefine** này sẽ được JS chuyển sang dạng boolean là **false.** Còn đoạn text **"Xin loi, Ban chua nhap ten"** là một chuỗi có nghĩa nên được JS định nghĩa là **true.**

Đó là lý do ở log thứ nhất nó log ra kết quả là **true.** Nên chúng ta có kết quả **"Ten cua ban la Xin loi, Ban chua nhap ten"**

# Các điều kiện khác

```
const getMe = (isLogin) => {
 return isLogin && {name: "Nguyen Van A", age: 23};
}

console.log(Thông tin của tôi ", getMe(false))
// Kết quả: Thông tin của tôi false

getMe(true))
// Kết quả: Thông tin của tôi {name: "Nguyen Van A", age: 23}
```

Bạn cũng có thể áp dụng câu điều kiện **&&** vào bài toán điều kiện của mình.

Khi gọi hàm **getMe** truyền vào tham số **isLogin** là **true** thì nó sẽ thực hiện return ra object **{name: "Nguyen Van A", age: 23}.**

Nếu **isLogin** là **false** return trực tiếp ra **false**

## switch

**Switch** được sử dụng để kiểm tra một biểu thức hoặc giá trị và thực hiện hàm khác nhau dựa trên giá trị đó. Cấu trúc **switch** thường được sử dụng khi bạn muốn kiểm tra một biểu thức so sánh với nhiều giá trị khác nhau . Cấu trúc **switch** bao gồm một biểu thức được kiểm tra và các trường hợp (**case**) thực hiện xử lý cho mỗi giá trị tương ứng.  **Default** để xử lý trường hợp không khớp nào.

# Điều kiện

```
const expr = 'Cam';
switch (expr) {
   case 'Tao':
      console.log('Oranges are $0.59 a pound.');
    break;
   case 'Cam':
   case 'Buoi':
      console.log('Cam va Buoi gia 20,000/kg');
      // Expected output: "'Cam va Buoi gia 20,000/kg."
      break;
   default:
      console.log(`Xin loi, chung toi khong co gia ${expr}.`);
}
```

# Vòng lặp

- Bạn có thể nghĩ về một vòng lặp như một phiên bản trò chơi máy tính, nơi bạn bảo ai đó thực hiện các bước X theo một hướng, sau đó Y bước theo một hướng khác. Ví dụ: ý tưởng 'Đi năm bước về phía đông' có thể được thể hiện theo cách này dưới dạng một vòng lặp:

```
for (let step = 0; step < 5; step++) {
    // Chạy 5 lần, giá trị đi từ 0 -> 4
    console.log("Đi bộ mỗi bước về phía đông, bước thứ ", i + 1);
}
```

# Vòng lặp

- Các lệnh vòng lặp được cung cấp bởi Javscript:
  - Câu lệnh for
  - Câu lệnh do...while
  - Câu lệnh while
  - Câu lệnh for...in
  - Câu lệnh for...off
- Trong quá trình phát triển các ứng dụng đa nền tảng bằng React Native, các bạn sẽ thường xài vòng lặp để render ra các phần tử trong danh sách

# Vòng lặp

- Tiếp theo đây, các bạn sẽ được ôn lại một số vòng lặp phổ biến thường gặp

  - Câu lệnh **do...while**

    Câu lệnh **do..while** lặp lại cho đến khi một điều kiện cụ thể là **false**. **do..while** ... trông như sau:

```
do
    console.log("//Làm gì đó ở đây");
while (condition);
```

## Câu lệnh **for...in**

Trong câu lệnh **for...in** lặp lại một biến được chỉ định trên tất cả các thuộc tính liệt kê của một đối tượng. Đối với mỗi thuộc tính riêng biệt, JavaScript thực thi các câu lệnh được chỉ định.

```
const dumpProps = (obj, objName) => {
    let result = "";
    for (const i in obj) {
        result += `${objName}.${i} = ${obj[i]}\n`
    }
    console.log(result)
}
const myCar = {make: "Ford", model: "Mustang"}
dumpProps(myCar, "car")
```

```
Kết quả:
car.make = Ford
car.model = Mustang
```

Trong câu lệnh **for** ở trên thực hiện lặp các phần có trong **obj,** biến i lặp qua các phần tử, lấy key của **obj.**

# Thao tác với mảng

- Trong quá trình phát triển ứng dụng đa nền tảng (React native). Bạn cần phải nắm chắc được các kiến thức về mảng, cách xử lý và sử dụng các phương thức có sẵn trong **Javascript.** Ngoài ra, các bạn cũng phải biết xử lý các trường hợp gây lỗi khi thao tác với mảng.

```
const arr = ['banana', 'apple', 'peach', 2, 'mango', {name: "Name"}]
```

  - Đây là một mảng, mảng thuộc kiểu dữ liệu tham chiếu (non-primary), thế nên bạn không cần khai báo biến bằng **let,** và có thể sử dụng các phương thức có sẵn trong array, để thay đổi giá trị của nó.

# Thao tác với mảng

- Các phương thức với mảng thường gặp:
  - **Array: length:** Thuộc tính dữ liệu độ dài của một phiên bản Array đại diện cho số lượng phần tử trong mảng đó. Giá trị là một số nguyên 32 bit không dấu, luôn lớn hơn chỉ số cao nhất trong mảng.

```
console.log("Do dai cua mang la: ", arr.length)
```

# Thao tác với mảng

- **Array.prototype.concat():** Phương thức **concat()** của các trường hợp **Array** được sử dụng để hợp nhất hai hoặc nhiều mảng. Phương thức này không thay đổi các mảng hiện có, mà thay vào đó trả về một mảng mới.

```
const array1 = ['a', 'b', 'c'];
const array2 = ['d', 'e', 'f'];
const array3 = array1.concat(array2);

console.log(array3);
// Kết quả: Array ["a", "b", "c", "d", "e", "f"]
```

# Thao tác với mảng

- **Array.prototype.filter():** Phương thức **filter()** tạo ra một mảng mới từ một mảng nhất định, chỉ lấy các phần tử nếu thoả mãn điều kiện bạn cung cấp

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction',
'present'];

const result = words.filter((word) => word.length > 6);

console.log(result);
// Kết quả: Array ["exuberant", "destruction", "present"]
```

# Thao tác với mảng

- **Array.prototype.find():** Phương thức **find()** trả về phần tử đầu tiên trong mảng được cung cấp thỏa mãn điều kiện được cung cấp. Nếu không có giá trị nào thỏa mãn hàm kiểm tra, **undefined** sẽ được trả về.

```
const array1 = [5, 12, 8, 130, 44];

const found = array1.find((element) => element > 10);
const found2 = array1.find((element) => element > 10000);

console.log(found);
// Kết quả: 12

console.log(found);
// Kết quả: undefined
```

# Thao tác với mảng

- **Array.prototype.map():** Phương thức **map()** tạo ra một mảng mới được được sao chép từ mảng bạn cung cấp

```
const array1 = [1, 4, 9, 16];

// Tạo một mảng mới, gấp đôi giá trị từ mảng cũ
const map1 = array1.map((x) => x * 2);

console.log(map1);
// Kết quả: Array [2, 8, 18, 32]
```

# Thao tác với mảng

- **Array.prototype.forEach():** Phương thức **forEach()** dùng để duyệt qua từng phần tử trong mảng

```
const array1 = ['a', 'b', 'c'];

array1.forEach((element) => console.log(element));

// Kết quả: "a"
// Kết quả: "b"
// Kết quả: "c"
```

# Thao tác với mảng

- Ở trên đây là một số phương thức thao tác với mảng phổ biến. Nhưng ở tình huống thực tế, có những mảng dữ liệu bạn sẽ lấy từ **API**, và đôi khi dữ liệu này sẽ là **null** hoặc **undefine,** và khi sử dụng các dữ liệu này gọi các phương thức của mảng, trường hợp chúng ta không mong muốn sẽ làm dừng chương trình, gọi là **crash app.**

```
const array1 = null

array1.forEach((element) => console.log(element));
```

```
▸ TypeError: Cannot read properties of null (reading 'forE
  ach')
      at <anonymous>:3:8
      at mn (<anonymous>:16:5455)
```

# Thao tác với mảng

- Để tránh việc **crash app** này xảy ra bạn sẽ sử dụng **Optional chaining (?.)**, đã được giới thiệu từ bài trước

```
const array1 = null

array1?.forEach((element) => console.log(element));
```

  - Sẽ không có vòng lặp nào diễn ra ở đây, bởi vì **array1** có giá trị là **null**

# Thao tác với object

- **Object** đại diện cho một trong các kiểu dữ liệu của JavaScript. Nó được sử dụng để lưu trữ các key collections khác nhau và các thực thể phức tạp hơn. Các **object** có thể được tạo bằng cách sử dụng hàm tạo **Object()** hoặc trình khởi tạo đối tượng **{}** cú pháp theo nghĩa đen.

```
const object1 = {
  a: 'somestring',
  b: 42,
};
```

# Thao tác với object

- Các phương thức xử lý **object** phổ biến
  - **Object.entries():** Phương thức static **Object.entries()** trả về một mảng của các cặp **key - value** thuộc tính key value có thể liệt kê của chính một đối tượng nhất định.

```
const object1 = { a: 'somestring', b: 42 };

for (const [key, value] of Object.entries(object1)) {
  console.log(`${key}: ${value}`);
}

// Kết quả:
// "a: somestring"
// "b: 42"
```

# Thao tác với object

- **Object.keys():** Phương thức static **Object.keys()** trả về một mảng của tên thuộc tính **key** chuỗi liệt kê của chính một đối tượng nhất định.

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false,
};

console.log(Object.keys(object1));
// Kết quả: Array ["a", "b", "c"]
```

# Thao tác với object

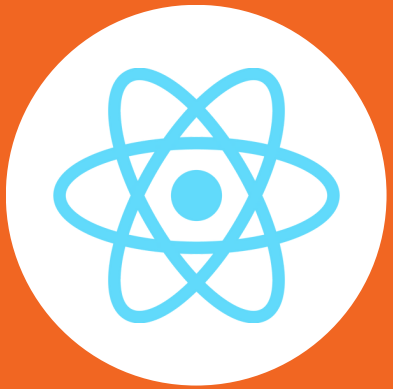
- **Object.values():** Phương thức static **Object.values()** trả về một mảng các giá trị thuộc tính khóa chuỗi có thể liệt kê của chính một đối tượng nhất định.

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false,
};

console.log(Object.values(object1));
// Kết quả: Array ["somestring", 42, false]
```

FPT Education

FPT POLYTECHNIC



# LẬP TRÌNH ĐA NỀN TẢNG VỚI REACT NATIVE

## BÀI 2: ÔN TẬP KIẾN THỨC JAVASCRIPT NÂNG CAO

## PHẦN 2: XỬ LÝ LỖI VÀ XỬ LÝ BẤT ĐỒNG BỘ TRONG JAVASCRIPT

https://caodang.fpt.edu.vn/

## Mục tiêu của bài học này

Bài học này sẽ giúp các bạn sẽ được học về cách xử lý bất động trong ngôn ngữ lập trình Javascript.

Thứ hai, các bạn sẽ được ôn lại cách bắt lỗi code, cách xử lý chúng để ứng dụng của bạn hoạt động tốt trong các trường hợp không mong muốn xảy ra đây là một bài học khá quan trọng, các bạn hãy cố gắng nắm bắt rõ phần này nhé.

## Giới thiệu

Bản chất JavaScript là một ngôn ngữ đơn luồng, và nó chỉ chạy trên một luồng (thread) duy nhất. Thread này được dựa trên một khái niệm gọi là even loop. Thread này sẽ phản hồi tới các event khi chúng xảy ra. Là một ngôn ngữ đơn luồng, JavaScript chỉ có thể xử lý từng câu lệnh một. Trong khi đang xử lý câu lệnh đó thì thread sẽ bị block.

Việc đơn luồng này có nhiều ưu điểm, đơn giản nhất là nó sẽ giúp code dễ hơn. Bạn sẽ không cần phải quan tâm đến các vấn đề liên quan đến concurency. Code của bạn sẽ được thực thi tuần tự từ trên xuống dưới theo cách bạn code. Bạn cũng sẽ không phải lo đến việc có nhiều việc khác nhau chạy trên cùng một thời điểm.

Tuy nhiên nó vẫn có nhược điểm lớn là chỉ có một công việc được làm tại một thời điểm. Mọi việc khác phải chờ đến khi nó hoàn thiện mới có thể được thực thi. Việc này thật sự sẽ là một vấn đề nếu bạn phải lấy data từ một API.

- **Javscript cung cấp 3 loại xử lý bất đồng bộ:**

1. Callback
2. Promise
3. Async/await

## Callback là gì?

Cách đầu tiên và cũng là lâu đời nhất để xử lý bất đồng bộ trong JavaScript là sử dụng **callbacks**. Một **callback** là một hàm bất đồng bộ được truyền vào một hàm khác dưới dạng tham số khi gọi. Khi function bạn chạy xong thì nó sẽ gọi lại hàm **callback** được truyền vào.

Hàm **callback** này sẽ không được gọi, không được thực thi hay làm bất cứ điều gì cho đến khi hàm chính chạy xong. Hàm này sẽ không block main thread do đó main thread sẽ có thể làm các việc khác. Ví dụ, khi phát triển ứng dụng đa nền tảng (React native), bạn sẽ có prop **onChange** trong thẻ **TextInput,** mỗi khi bạn nhập kí tự vào, **onChange** này sẽ chạy hàm callback bên trong đó nếu có, bạn có thể lấy value được nhập ở **TextInput**

## Ví dụ hàm callback

```
function makeAPICall() {
  console.log('Bắt đầu gọi API.');
  // delay 2 giây
  setTimeout(() => {
    console.log('Nhận data từ API.');
    processAPIData();
    console.log('Gọi API thành công.');
  }, 2000)
}

function processAPIData() {
  console.log('Xử lý data');
};
makeAPICall();
```

# Callback

Khi gọi hàm **makeAPICall,** setTimeout sẽ được thực thi và chờ 2 giây để gọi bắt đầu chạy các hàm **processAPIData** bên trong đó, **processAPIData** đây chính là một **callback function**

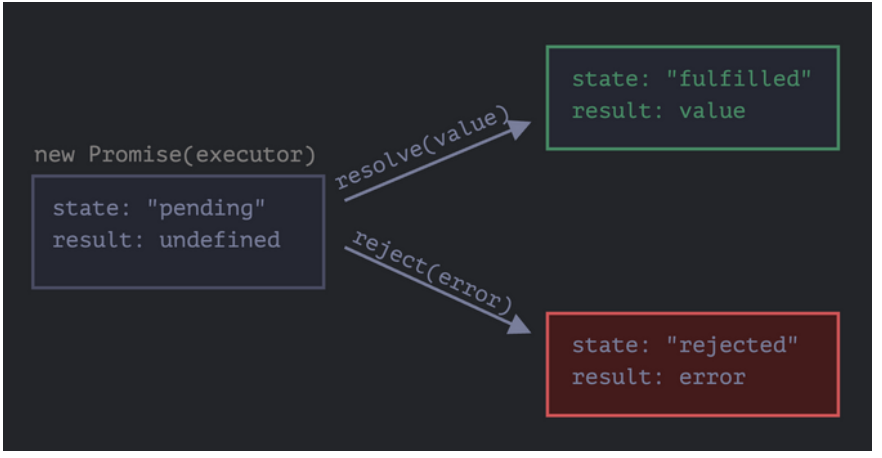
## Promise là gì?

Cách thứ hai đó để xử lý bất đồng bộ là promise. **Promise** là một tính năng mới được giới thiệu ở ES6, nó cung cấp một cách dễ dàng hơn để xử lý bất đồng bộ trong JavaScript.

Một **Promise** là một object chứa một giá trị. Giá trị này sẽ không được show ra khi bạn tạo **Promise**. **Promise** sẽ trả về giá trị này khi nó thành công hoặc thất bại. Có 3 hàm handler bạn có thể sử dụng để lấy giá trị mà **Promise** trả về.

Các hàm này là **then()**, **catch()** và **finally()**. Đẻ sử dụng những hàm handler này thì ta gắn chúng vào một object **Promise**. Dựa vào trạng thái của **Promise** thì handler tương ứng sẽ được invoke.

# Promise




- Khi gọi **resolve** trong promise **then** sẽ được invoke
- Khi gọi **reject** trong promise **catch** sẽ được invoke
- **finally** sẽ được gọi khi promise thực hiện xong, không kể nó là resolve hay reject

# Promise

```
let promise = new Promise(
    function(resolve, reject) {
        setTimeout(() => resolve("done!"), 1000);
    }
);
// resolve sẽ invoke then
promise
    .finally(() => console.log(“Hoàn thành promise”)
        .then(result => console.log(“Thành công ”, result))
            .catch(error => console.log(“Thất bại”))

//  Kết quả:
Hoàn thành promise
Thành công done!
```

## Async/await là gì?

Cách cuối cùng để xử lý bất đồng bộ là sử dụng **async/await**. **Async/await** được giới thiệu ở ES8, chúng được cấu tạo từ 2 phần. Phần đầu tiên là function **async**. Hàm này sẽ được tự động thực thi bất đồng bộ. Giá trị nó trả về là một **Promise**. Vì trả về **Promise** nên bạn sẽ phải sử dụng các handler của **Promise** để xử lý giá trị này.

Phần thứ hai của **async/await** là operator await. Operator này sẽ được dùng cùng với một **Promise**. Nó sẽ khiến cho function **async** tạm dừng cho đến khi **Promise** đó chạy xong. Ngay sau đó nó sẽ lấy giá trị của **Promise** mà cho function **async** tiếp tục chạy.

Các function **async** đều bất đồng bộ, khi bị pause bởi **await** thì phần code còn lại vẫn chạy bình thường vì function đấy không block main thread. Khi **Promise** chạy xong thì hàm **async** sẽ chạy tiếp và trả về giá trị của **Promise**.

Một điều quan trọng hơn là **await** phải được viết trong hàm **async** nếu không thì sẽ gặp lỗi syntax error

# Aync/await

```
// tạo một hàm async
async function makeAPICall() {
    console.log('Bắt đầu gọi API.');
    // Tạo một Promise gọi api
    const dataPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
       console.log('Nhận data từ API.');
       resolve('Gọi API thành công.');
     }, 2000);
    });
 return (dataReceived = await dataPromise);
}
```

```
makeAPICall
  // add handle cho status thành công
    .then(result => {
       console.log("Goi API thanh cong ", result);
    })
    // add handle cho status thất bại
    .catch(error => {
       console.log(`Có lỗi xảy ra: ${error}.`)
 })
```

Với **async/await**, chúng ta hiếm khi cần phải viết **promise.then/catch**, nhưng chúng ta vẫn không nên quên rằng chúng dựa trên **Promise**, bởi vì đôi khi chúng ta phải sử dụng các phương thức này. Ngoài ra **Promise.all** là khá tốt khi chúng ta muốn làm chờ nhiều công việc cùng một lúc.

# Xử lý lỗi try...catch

## Giới thiệu

Cho dù chúng ta lập trình giỏi đến đâu, đôi khi các script của chúng ta cũng có lỗi. Chúng có thể xảy ra do lỗi của chúng ta, đầu vào người dùng không mong muốn, phản hồi máy chủ sai và vì hàng nghìn lý do khác.

Thông thường, một script **"dies"** (ngay lập tức dừng lại) trong trường hợp có lỗi, nó sẽ được ra console.

Nhưng có một cấu trúc cú pháp **try...catch** cho phép chúng ta 'bắt' lỗi, thay vì dừng chương trình, làm điều gì đó hợp lý hơn.

# Xử lý lỗi try...catch

## Cú pháp của **try...catch**

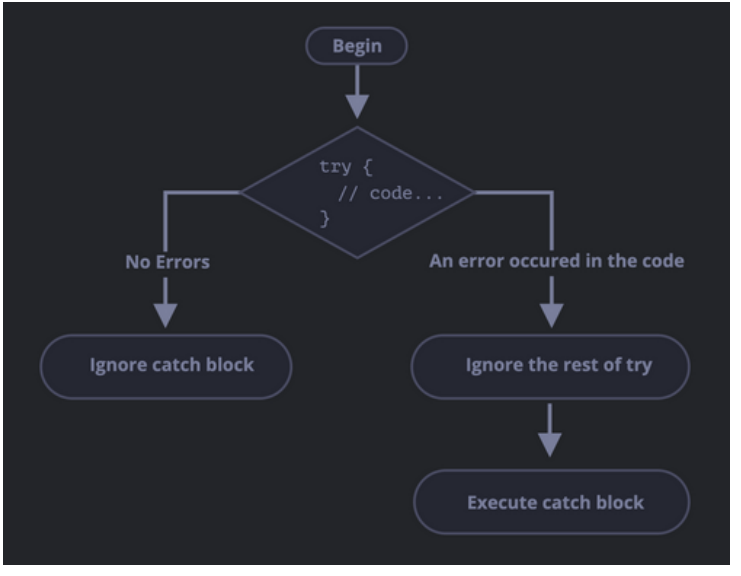
**try...catch** có 2 thành phần chính: **try** sau đó là **catch**

```
try {
   // code...
} catch (err) {
   // error handling
}
```

- **try...catch** hoạt động như sau:

1. Đầu tiên, code trong **try {...}** được chạy.
2. Nếu không có lỗi, thì **catch (err)** sẽ bị bỏ qua: việc thực thi đến cuối **try** và tiếp tục, bỏ qua **catch**.

3. Nếu xảy ra lỗi, thì quá trình thực thi **try** sẽ bị dừng và luồng điều khiển đến đầu **catch (err)**. Biến **err** (chúng ta có thể sử dụng bất kỳ tên nào cho nó) sẽ chứa một đối tượng lỗi với các chi tiết về những gì đã xảy ra.



# Xử lý lỗi try...catch

```
try {
    console.log('Bắt đầu chọn'); // (1) <--
    lalala; // Lỗi, biến không xác định
    console.log('Cuối try “Không được chạy”'); // (2)
} catch (err) {
    console.log(“Lỗi xảy ra“);
}
```

# Xử lý lỗi try...catch

## Tình huống

Điều gì sẽ xảy ra nếu json đúng cú pháp, nhưng không có thuộc tính **name**?

```
let json = '{ "age": 30 }'; // Dữ liệu đầu vào

try {
    let user = JSON.parse(json); // <-- Không có lỗi nào
    console.log( user.name ); // Không có name
} catch (err) {
    console.log( "Không được thực thi" );
}
```

# Xử lý lỗi try...catch

Ở đây **JSON.parse** chạy bình thường, nhưng việc không **name** thực sự là một lỗi đối với chúng ta.

Để thông báo xử lý lỗi, chúng ta sẽ sử dụng toán tử **throw**.

- Toán tử **"Throw"**

Toán tử **throw** tạo ra lỗi, để các bạn có thể **catch** bắt được nó.

Cú pháp là:

```
throw <error object>
```

# Xử lý lỗi try...catch

Về mặt kỹ thuật, chúng ta có thể sử dụng bất cứ thứ gì như một object lỗi. Đó thậm chí có thể là một nguyên thủy, như một số hoặc một chuỗi, nhưng tốt hơn là sử dụng các đối tượng, tốt nhất là với các thuộc tính **name** và **message.**

JavaScript có nhiều hàm tạo tích hợp cho các lỗi tiêu chuẩn: **Error**, **SyntaxError**, **ReferenceError**, **TypeError** và các trình tạo khác. Chúng ta cũng có thể sử dụng chúng để tạo các object lỗi.

```
let error = new Error(message);
// or
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...
```

# Xử lý lỗi try...catch

Và trong trường hợp của chúng ta, việc không có **name** là một lỗi, vì người dùng phải có tên.

Vậy nên hãy xử lý lỗi như bên dưới:

# Xử lý lỗi try...catch

```
let json = '{ "age": 30 }'; // Dữ liệu khởi tạo

try {
    let user = JSON.parse(json); // <-- Không có lỗi
    if (!user.name) {
        throw new SyntaxError("Dữ liệu không tồn tại: không có name"); (*)
    }
     console.log( user.name );
} catch (err) {
    console.log( "JSON Error: " + err.message );
    // JSON Error: Incomplete data: no name
}
```

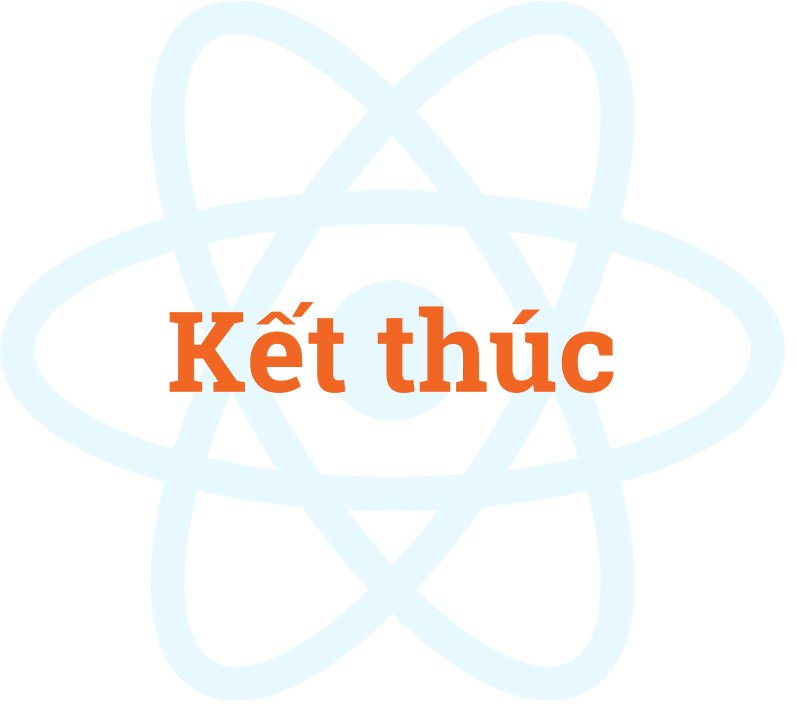
# Xử lý lỗi try...catch

Trong dòng **(*)**, toán tử ném tạo ra một **SyntaxError** với message đã cho, giống như cách **JavaScript** sẽ tự tạo ra nó. Việc thực hiện **try** ngay lập tức dừng lại và nhảy vào **catch**.

Bây giờ **catch** đã trở thành một nơi duy nhất cho tất cả các xử lý lỗi: cho cả **JSON.parse** và các trường hợp khác.

# Tổng kết

- [ ] Ôn lại điều kiện trong javascript
- [ ] Ôn lại các phương thức thao tác với mảng thường gặp
- [ ] Ôn lại vòng lặp trong javascript
- [ ] Ôn lại các phương thức thao tác với object thường gặp
- [ ] Ôn lại kiến thức xử lý bất đồng bộ
- [ ] Bắt lỗi và xử lý bằng try...catch



# Kết thúc