Objects

Ba Nguyễn

What is object?

Object (đối tượng) là kiểu dữ liệu đặc biệt, object ***mô phỏng một đối tượng thực tế trong ngôn ngữ lập trình***, mỗi object bao gồm 2 phần: ***properties*** *(thuộc tính)* và ***methods*** *(phương thức)*

Thông tin về đối tượng được lưu trữ dưới dạng các cặp **key: value**, key có kiểu string, value có thể là bất kỳ kiểu dữ liệu nào.

Mỗi **property** là một thông tin mô tả về đối tượng, một đối tượng có thể có nhiều properties.

Mỗi **method** là một hành động (chức năng) mà đối tượng có thể thực hiện, method là một key có giá trị là một **function**

Objects

Mỗi **person** là một object, có các properties như **name**, **age**, **job**, ... và các methods như **speak**, **laugh**, **eat**, ...

Mỗi **computer** là một object, có các properties như **brand**, **series**, **size**, **cpu**, **memory**, ..., và các methods như **start**, **shutdown**, ...,

Mỗi **cat** là một object, có các properties như **name**, **breed**, **weight**, ..., và các methods như **meow**, **run**, **bite**, ...,

Mỗi **character** là một object, có các properties như **name**, **level**, **weapon**, **damage**, ... và các methods như **attack**, **run**, ...

Objects

```
// Khai báo một object rỗng (chưa có thông tin)
let obj = {}; // literal syntax

// Hoặc khai báo kèm thông tin
// Mỗi cặp key: value được phân tách bằng dấu ,
let obj = {
    key1: value1,
    // ...
    method1: function() { }
};
```

Objects

```
let myComputer = {
    brand: "Apple",
    type: "Laptop",
    "operating system": "MacOS BigSur",
    start: function () {
        console.log("Starting");
    },
    restart: function () {
        console.log("Restarting");
    },
};
```

Objects

Truy cập thông tin trong đối tượng sử dụng cú pháp: **obj.key** hoặc **obj[“key”]**

```
// Lấy giá trị
myComputer.brand; // Apple
myComputer["operating system"]; // MacOS BigSur
myComputer.start(); // Starting
// Cập nhật giá trị
myComputer.type = "Macbook";
// Thêm thuộc tính
myComputer.release = "2018";
// Xóa thuộc tính
delete myComputer.release;
```

Objects

Một số thao tác khác với object

```
// Kiểm tra một key có tồn tại trong object không
"brand" in myComputer; // true
"monitor" in myComputer; // false

// Lặp qua các (*) thuộc tính trong object
for (let key in myComputer) {
    console.log(key + ": " + myComputer[key]);
}
// brand: Apple
// ...
```

This

Trong hàm tồn tại một **biến** đặc biệt - **this** - tham chiếu tới đối tượng gọi nó (trước dấu **.**), thông qua **this**, methods có thể truy cập được tới các thuộc tính trong object

```
let myComputer = {
    // ...

    getInfo: function () {
        return this.brand + " - " + this.type;
    }
}

// this = myComputer
myComputer.getInfo(); // Apple - Macbook
```

Object to primitive

JavaScript hỗ trợ chuyển đổi kiểu dữ liệu tự động, đối với object cũng tương tự.

Object được tự động chuyển đổi về kiểu **primitive** khi sử dụng các built-in function hoặc toán tử yêu cầu kiểu dữ liệu **primitive** bằng cách gọi các phương thức đặc biệt **toString()** và **valueOf()**

- Đối với các built-in function và toán tử cần kiểu dữ liệu **string**: object **ưu tiên** gọi phương thức **toString()**, nếu không có **toString()** thì gọi phương thức **valueOf()**
- Đối với các built-in function và toán tử cần kiểu dữ liệu **number**: object **ưu tiên** gọi phương thức **valueOf()**, nếu không có **valueOf()** thì gọi phương thức **toString()**

Object to primitive

```
let myComputer = {
    toString: function () {
        return this.getInfo();
    },

    valueOf: function () { return 1; },
};

alert(myComputer); // Apple - Macbook
1 + myComputer; // 1 + 1 = 2
```

Note

- **Key** được lưu với kiểu dữ liệu **string**
- **Key** không bị giới hạn về đặt tên giống như biến, nó có thể chứa ký tự đặc biệt, trùng với keyword hay chứa dấu cách
- **Key** nên sử dụng cú pháp **camelCase** giống như biến và hàm
- Cú pháp truy cập thông tin trong object:
    - Dot Notation: **object.key** (key phải là một thuộc tính có trong object)
    - Bracket Notation: **object[“key”]** dùng cho các key với tên đặc biệt hoặc sử dụng với biến
- Khi truy cập một property không tồn tại trong object, giá trị trả về sẽ là **undefined**
- **this** trong phương thức là từ khóa thay thế, tham chiếu đến chính đối tượng

Primitive vs Reference

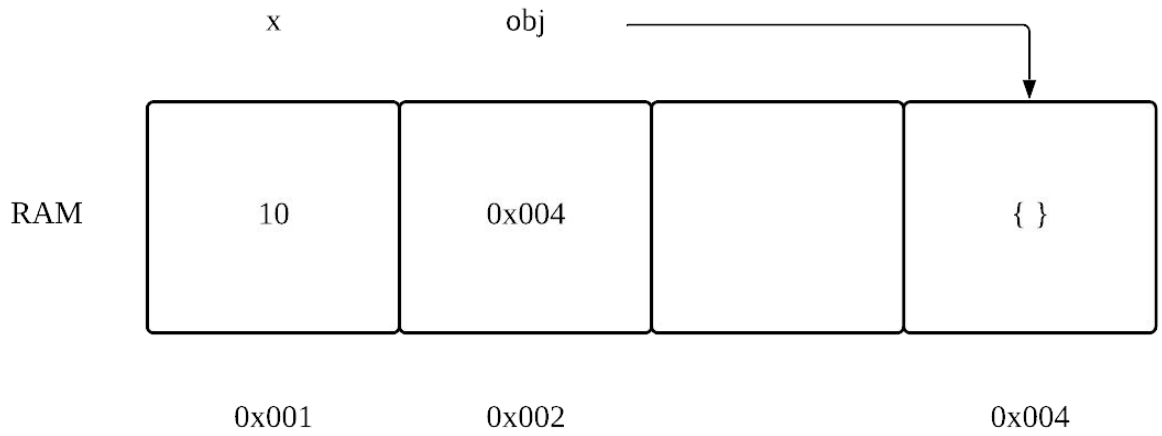
Với các giá trị nguyên thủy - primitive:

- Giá trị là không thể thay đổi - immutable
- Biến lưu trữ trực tiếp giá trị trong vùng nhớ của nó
- Thay đổi giá trị của biến đồng thời làm thay đổi vùng nhớ của nó
- So sánh trực tiếp trên giá trị
- Các biến khác sao chép trực tiếp giá trị

Với object:

- Biến chỉ lưu trữ địa chỉ ô nhớ (tham chiếu - reference) đến đối tượng thực
- Thay đổi giá trị (thuộc tính) đối tượng không làm thay đổi vùng nhớ của biến
- So sánh dựa trên giá trị tham chiếu - reference
- Các biến khác sao chép địa chỉ ô nhớ

Primitive vs Reference



Reference

Khi 2 biến cùng tham chiếu tới một đối tượng, một biến thay đổi giá trị của đối tượng, biến còn lại cũng nhận được sự thay đổi đó

```
let copy = myComputer; // copy reference
copy == myComputer; // true

copy["operating system"] = "MacOS Monterey";
myComputer["operating system"]; // MacOS Monterey
```

Reference

```
// VD primitive vs reference
let x = 1;
let obj = { y: 10 };

function f(primitive, reference) {
    primitive = 10; // thay đổi giá trị
    reference.y = 100; // thay đổi giá trị
}

f(x, obj); // primitive = x, reference = obj
x; // 1
obj.y; // 100
```

Copying Object

```
// Chỉ sao chép giá trị trong object sử dụng for in
let copy = {};

for (let key in myComputer) {
    copy[key] = myComputer[key];
}

copy.brand = "Dell"; // thay đổi giá trị
myComputer.brand; // Apple
```

Property Flags

Mỗi key của object cũng là một đối tượng đặc biệt, bao gồm 4 thuộc tính - **flag**:

- **value**: là giá trị của thuộc tính đó
- **writeable**: nếu **true**, giá trị có thể thay đổi, nếu **false** thì thuộc tính chỉ có thể đọc
- **enumerable**: nếu **true**, thuộc tính xuất hiện trong vòng lặp **for in** hoặc **Object.assign()**, nếu **false** thì không
- **configurable**: nếu **true**, thuộc tính có thể xóa hoặc thay đổi, nếu **false** thì không

💡Mặc định khi khai báo object và thuộc tính với **literal syntax**, các **flags** được đặt thành **true**

💡Để xem chi tiết các flag cho một thuộc tính, sử dụng:

```
Object.getOwnPropertyDescriptor(object, key);
```

Property Flags

Để thay đổi giá trị cho các **flag**, sử dụng:

```
Object.defineProperty(object, key, descriptor);
// Hoặc
Object.defineProperties(object, { key: descriptor });

// Ví dụ
Object.defineProperty(myComputer, "x", {
    value: 1,
    writable: true,
    // các flag không khai báo mặc định là false
});
```

Property Flags

```
// Khi sao chép object với for in
// Nó chỉ sao chép giá trị của thuộc tính
// Để sao chép cả các flag tương ứng
// Sử dụng kết hợp defineProperties
// Và getOwnPropertyDescriptors
let copy = Object.defineProperties(
    {},
    Object.getOwnPropertyDescriptors(myComputer)
);
```

Getter vs Setter

Ngoài các properties thông thường *(data properties)*, object có thể có các properties khác *(accessor properties)* là các hàm được biệt được thực thi khi muốn lấy giá trị thuộc tính ***(getter)*** hoặc khi muốn cập nhật giá trị cho thuộc tính ***(setter)***

- ***Getter*** và ***Setter*** là 2 function đặc biệt, đối với mã bên ngoài object, nó giống như thuộc tính thông thường
- ***Getter*** cho phép tùy chỉnh giá trị trả về khi mã bên ngoài muốn truy cập giá trị của một thuộc tính
- ***Setter*** cho phép thêm các logic xử lý khi mã bên ngoài muốn cập nhật giá trị của một thuộc tính (validate dữ liệu phải hợp lệ, format lại dữ liệu, ...)

Getter vs Setter

```
let myComputer = {
    __ram: 8,

    get ram() { return this.__ram; },
    set ram(value) {
        if (isValid(value)) this.__ram == value
    }
};

myComputer.ram; // 8 - getter
myComputer.ram = 16; // setter
```

Constructor Function

Khi cần tạo ra nhiều đối tượng giống nhau, việc xử dụng cú pháp thông thường khiến code bị lặp lại, khó quản lý và chỉnh sửa.

Cú pháp hàm khởi tạo - **constructor function** cho phép tạo ra nhiều đối tượng dựa trên một bản mẫu với danh sách properties và methods được xác định trước, giúp giảm thiểu code so với cú pháp tạo đối tượng thông thường, dễ dàng quản lý hơn

Với hàm khởi tạo:

- Quy ước tên hàm viết hoa những chữ cái đầu tiên
- Khởi tạo đối tượng mới sử dụng từ khóa **new**

Constructor Function

```
function Computer(brand, type) {
    this.brand = brand;
    this.type = type;

    this.getInfo = function () {
        return this.brand + " - " + this.type;
    }
}

let myComputer = new Computer("Apple", "Macbook");
let otherComputer = new Computer("Dell", "XPS");
```

Exercises

Tạo đối tượng **counter** bao gồm:

- Thuộc tính **value**, giá trị ban đầu bằng **0**
- Phương thức **up()** tăng **value** lên **1**
- Phương thức **down()** giảm **value 1**
- Phương thức **get()** trả về giá trị hiện tại của **value**

💡 Làm thế nào để gọi nối các phương thức???

VD: **counter.up().down().up().down().get().up().down().get()....**

Exercises

1. Viết hàm khởi tạo **Weapon** tạo các đối tượng bao gồm một số thông tin như **type**, **damage**, **speed**, ...
2. Viết hàm khởi tạo **Character** tạo các đối tượng bao gồm một số thông tin như **name**, **level**, ..., **weapon**
3. Thêm các phương thức cho **Character** như **attack()**, **changeWeapon()**, ...