**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Prototypal inheritance\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

- Trong js object có 1 hidden property là [[Prototype]] có đặc điểm là trỏ đến null hoặc 1 object khác. Object khác đó được gọi là prototype. Trong 1 object, khi gọi 1 property nếu nó không có trong object thì object sẽ tự động kiểm tra và lấy property đó từ prototype.

- [[Prototype]] là property nội và bị ẩn đi, nhưng ta có nhiều cách để truy cập nó.

- Đầu tiên là dùng tên property sau : \_\_proto\_\_ : như sau

let animal = {

eats: true

walk() {

alert("Animal walk");

}

};

let rabbit = {

jumps: true

};

rabbit.\_\_proto\_\_ = animal;

alert( rabbit.eats ); // true (\*\*)

alert( rabbit.jumps ); // true

rabbit.walk(); // Animal walk

- khi rabbit gọi eats, ko thấy trong rabbit thì sẽ đi theo tham chiếu prototype là [[Prototype]] và tìm thấy trong animal

- NOTE : \_\_proto\_\_ khác với [[Prototype]], \_\_proto\_\_ là getter/setter của nó.

- Lúc này ta có thể nói animal là prototype của rabbit

- NOTE : chú ý , sẽ có 2 giới hạn như sau :

+ tham chiếu ko được phép là vòng tròn.

+ giá trị của \_\_proto\_\_ có thể là null hoặc object, giá trị khác sẽ bị loại bỏ.

- Prototype sẽ chỉ được dùng để đọc property, việc viết và thay đổi hoạt động diễn ra ngay trên object. Ngoại trừ accessor properties setter/getter sẽ sử dụng trong prototype nhưng các giá trị vẫn chỉ được tạo trong object

let animal = {

eats: true,

walk() {

/\* this method won't be used by rabbit \*/

alert("Rabbit!");

}

};

let rabbit = {

\_\_proto\_\_: animal

};

rabbit.walk = function() {

alert("Rabbit! Bounce-bounce!");

};

rabbit.walk(); // Rabbit! Bounce-bounce!

animal.walk(); // Rabbit!

- This không ảnh hưởng đến prototype : không qtrong nơi method được đặt, khi method được gọi thì this sẽ là object phía trước the dot.

let user = {

name: "John",

surname: "Smith",

set fullName(value) {

[this.name, this.surname] = value.split(" ");

},

get fullName() {

return `${this.name} ${this.surname}`;

}

};

let admin = {

\_\_proto\_\_: user,

isAdmin: true

};

alert(admin.fullName); // John Smith (\*)

// setter triggers!

admin.fullName = "Alice Cooper"; // (\*\*)

user.fullName = "John Smith"; // (\*\*)

- For...in cũng sẽ lọc ra các property thừa kế từ prototype

- Để chọn ra các properties của riêng object ta có thể dùng meethod sau : obj.hasOwnProperty(key) =>> Trả về true nếu key là của riêng object không phải kế thừa.

- NOTE : hasOwnProperty() là 1 method kế thừa từ Object.prototype nhưng lại không xuất hiện trong for.in là do nó ko được enumerable.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*F.prototype\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

- Ta có thể tạo Object bằng cách dùng new F().

- Nếu F.prototype là 1 object thì khi dùng new, new sẽ set [[Prototype]] của object mới đến object gắn với F.prototype.

- F.prototype đơn giản chỉ là 1 property.

let animal = {

eats: true

};

function Rabbit(name) {

this.name = name;

}

Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit"); // rabbit.\_\_proto\_\_ == animal

alert( rabbit.eats ); // true

==>>> khi **new Rabbit** được tạo, gán [[Prototype]] của nó tới animal.

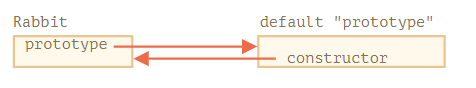
Và F.prototype là 1 property chỉ được dùng khi **new F** được gọi.

- Mỗi function đều có "prototype" property kể cả khi ta không cung cấp nó.

- Mặc định "prototype" là 1 object chứa duy nhất 1 property là constructor, constructor này sẽ trỏ ngược lại đến chính function đó.

- Tức là nếu ta gọi **new F, object mới sẽ có prototype trỏ đến object mặc định này và object này lại trỏ đến chính hàm gốc F.**

- F.prototype = { constructor: F }



function Rabbit() {}

/\*Rabbit.prototype = { constructor: Rabbit };\*/

let rabbit = new Rabbit();

alert(rabbit.constructor == Rabbit); // true

- Từ đây, ta có thể dùng constructor property để tạo 1 object mới từ constructor có sẵn.

function Rabbit(name) {

this.name = name;

alert(name);

}

let rabbit = new Rabbit("White Rabbit");

let rabbit2 = new rabbit.constructor("Black Rabbit");

- Khi ta thay đổi giá trị mặc định "prototye" của function, Khi đó sẽ ko còn "constructor"

- Để giữ constructor thì ta có thể thêm hoặc xóa property trong "prototype" thay vì thay đổi hoàn toàn nó.

function Rabbit() {}

Rabbit.prototype.jumps = true;

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Native prototypes\*\*\*\*\*\*\*\*\*\*\*\*\***

- "prototype" property là đặc tính core của cả JS. All built-in constructor functions use it.

- Object.prototype : ta đã biết let obj = {} <===> obj = new Object().

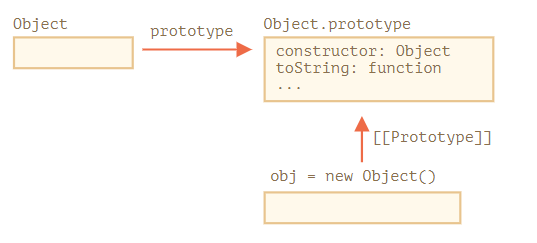
Object là 1 built-in object constructor funcion, có prototype chỉ đến 1 object lớn với nhiều method.

- Object.prototype sẽ tham chiếu đến 1 object cực lớn có chứa các method riêng.

let obj = {};

alert(obj.\_\_proto\_\_ === Object.prototype); // true

alert(Object.prototype.\_\_proto\_\_); // null



- Object là built-in object, Object,prototype là property của nó lúc này là 1 object lớn, Khi gọi **new Object()**  sẽ tạo object mới và gán [[prototype]] đến Object.prototype.

- Các built-in objects khác như Array, Date, Function.. cũng có method bên trong prototypes.

- Ví dụ khi ta tạo mảng bằng [] thì cũng có nghĩa là new Array() đang chạy ngầm.

- Do đó Array.prototype trở thành prototype của nó và cung cấp các method.

- Theo đặc tả, tất cả các the built-in prototypes have Object.prototype on the top.

- Do đó tại sao nói mọi thứ đểu kế thừa từ Object.

let arr = [1, 2, 3];

alert( arr.\_\_proto\_\_ === Array.prototype ); // true

alert( arr.\_\_proto\_\_.\_\_proto\_\_ === Object.prototype ); // true

alert( arr.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ ); // null

- Các built-in đểu có các method riêng nên chúng có thể sử dụng method của riêng mình không cần phải gọi từ prototype.

- Đối với các Primitives vì ko phải là object, nên khi cố gắng access các properties thì 1 object wrapper tạm thời sẽ đc khởi tạo bởi các built-in constructor String, Number, Boolean. Chúng cung cấp method từ String.prototype… và biến mất.

- NOTE : null và undefined thì không có object wrappers và prototype tương ứng cũng không có.

- Thay đổi native prototypes :

- Native prototypes can be modified.

String.prototype.show = function() {

alert(this);

};

"BOOM!".show(); // BOOM! // "BOOM!" sẽ được wrapper bởi String().

- Trong lập trình, a chỉ thay đổi native prototype đó là khi polyfilling.

- Polyfilling là thuật ngữ được áp dụng cho method tồn tại trong đặc tả JS nhưng chưa được supported bới JS engine.

**\*\*\*\*\*Prototype methods, objects without \_\_proto\_\_\*\*\*\*\*\*\***

- Thay vì dùng \_\_Proto\_\_ ta sẽ dùng các method sau :

+ **Object.create(proto[, descriptors]) – creates an empty object with given proto as [[Prototype]] and optional property descriptors.**

**+ Object.getPrototypeOf(obj) – returns the [[Prototype]] of obj.**

**+ Object.setPrototypeOf(obj, proto) – sets the [[Prototype]] of obj to proto.**

- Ví dụ :

let animal = {

eats: true

};

// create a new object with animal as a prototype

let rabbit = Object.create(animal);

alert(rabbit.eats); // true

alert(Object.getPrototypeOf(rabbit) === animal); // true

Object.setPrototypeOf(rabbit, {}); // change the prototype of rabbit to {}

- Hoặc dùng optional như sau :

let animal = {

eats: true

};

let rabbit = Object.create(animal, {

jumps: {

value: true

}

});

alert(rabbit.jumps); // true

- Xem lại Property flags and descriptors.

- Clone 1 object 1 cách tuyệt đối :

let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));

- NOTE : \_\_proto\_\_ là 1 getter/setter cho prototype. Để đặt name property là \_\_proto\_\_ ta phải tạo 1 object rỗng hoàn toàn, ko có prototype.\_\_proto\_\_ sẽ chỉ nhận null hoặc object.

let obj = Object.create(null); => Tạo object ko có prototype.

alert(obj); // Error (no toString)

let key = prompt("What's the key?", "\_\_proto\_\_");

obj[key] = "some value";

alert(obj[key]); // "some value"