**JavaScript Factory Functions with ES6+**

A factory function is any function which is not a class or constructor that returns a (presumably new) object. In JavaScript, any function can return an object. When it does so without the new keyword, it’s a factory function.

Factory functions have always been attractive in JavaScript because they offer the ability to easily produce object instances without diving into the complexities of classes and the new keyword.

For Example:

let person = {firstName:"John", lastName:"Doe", age:50};

console.log(person.firstName) // John

We can do the same like this

const firstName = "John";

const lastName = "Doe";

const age = "50";

var person = { firstName, lastName, age};

console.log(person) // {firstName:"John", lastName:"Doe", age:50}

**Prototypes**

Normally we achieve the inheritance using classes like Java or other object-oriented programming languages. In JavaScript, we achieve the inheritance model using prototypes. We know, ES6 standard introduces a new piece is called ‘class’, actually, it uses the prototype based inheritance behind the scene. The class syntax does not introduce a new object-oriented inheritance model to JavaScript. You should keep in mind, It is just a higher order function in the prototype chain or layer. You can check it by using **instanceof** operator. Prototypes are just like object delegation.

**Regular (Non-Constructible) Objects:**

The **Object.create()** method creates a new object, from a regular object and the a regular object in JavaScript is called prototype of the newly created object. We can achieve classical single inheritance by using **Object.create()** method.

**For Example 01:**

const food = {

init: function(type){

this.type = type;

},

eat: function(){

console.log('You ate '+ this.type);

}

}

food.init('Waffle');

food.eat();

In the above created object is universal for all kind of foods not specific food object.

If we want to create specific type of more objects we need to use **Object.create()** method for creating new objects from the existing one.

//food.init('Waffle');

//food.eat();

const waffle = Object.create(food);

waffle.init('Waffle');

waffle.eat();

const carrot = Object.create(food);

carrot.init('Carrot');

carrot.eat();

**Summary**:

In the above example, I have seen that we have created two new objects instead of calling directly the eat () method of the food object.

One thing you should remember that is when we create a new object using new keyword actually it does not create a copy of the object. It just creates an empty object and inherits their properties and methods as prototype. When we call a method the javascript interpreter first looks into its own method if not found then it looks into its prototype. In our case “food” object is called a prototype of waffle and carrot.

When we call the carrot.eat() method the javascript interpreter looks into the carrot object the eat is own method or not if not then looks into its prototype (food) if found then it shows the result otherwise throws error.

const food = {

init: function(type){

this.type = type;

},

eat: function(){

console.log('You ate '+ this.type);

}

}

const carrot = Object.create(food);

const waffle = Object.create(food);

food.eat = function(){

console.log("YOU TOTALY ATE THE " + this.type.toUpperCase())};

waffle.init('Waffle');

waffle.eat();

carrot.init('Carrot');

carrot.eat();

Output:

YOU TOTALY ATE THE WAFFLE

YOU TOTALY ATE THE CARROT

**Summary**:

From the above code, we can see that food.eat() method signature is effecting on carrot and waffle objects although we have created those objects first before assigning the signature of the food.eat() method.

Prototype Checking:

console.log(food.isPrototypeOf(waffle)); //true

**Prototype Inheritance**

Understanding Prototypal inheritance in JavaScript.

\_ What is constructor in javaScript.

- How to add properties to prototype.

- How to create classes

\_ ECMA6, ECMA2015

- The root object in javascript

In Javascript, by default every function has a property called prototype and this property object is by default empty so we can add properties and methods in the prototype property of the function. When we create an object from the function the object inherits the prototype.

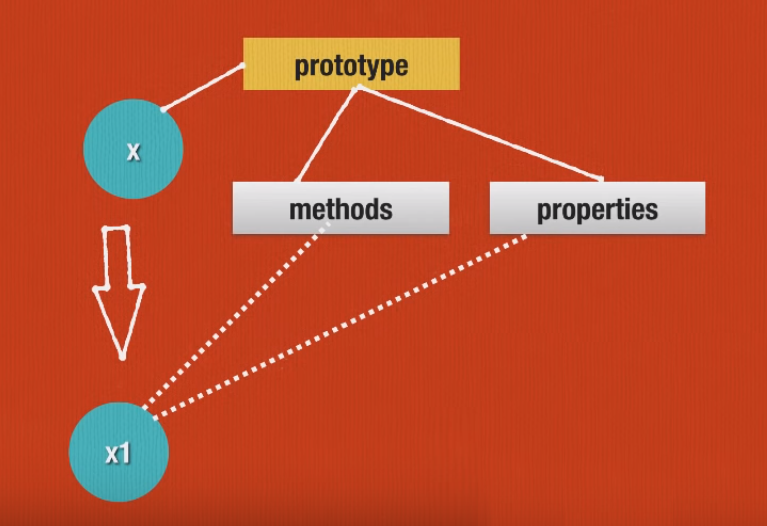
For example:

let Person = function (){

//

}

Person.prototype.propertyName;



**Constructor**:

Normally OOP Class and Javascript Class are not same actually the class in javascript is a constructor and it has prototype based inheritance. Every function expression is a constructor in javascript but we should use capital letter for separating constructor from the normal function.

For example:

let Person = function(age)

{

this.firstName="John";

this. lastName="Doe";

this. age=age;

this.fullName = function(){

return this.firstName +” ”+ this.lastName;

}

}

let p1 = new Person(50);

let p2 = new Person(40);

console.log(p1.fullName);

In the above constructor we can set prototype property like this later when an object will be created that will be inherited.

For example 02:

let Person = function(age)

{

this.firstName="John";

this. lastName="Doe";

this. age=age;

}

Person.prototype.fullName = function(){

return this.firstName +” ”+ this.lastName;

}

let p1 = new Person(50);

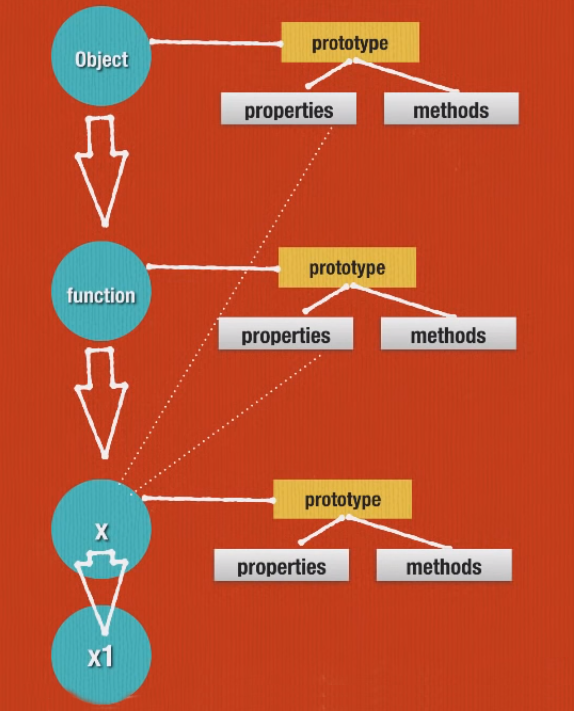
let p2 = new Person(40);

console.log(p1.fullName);

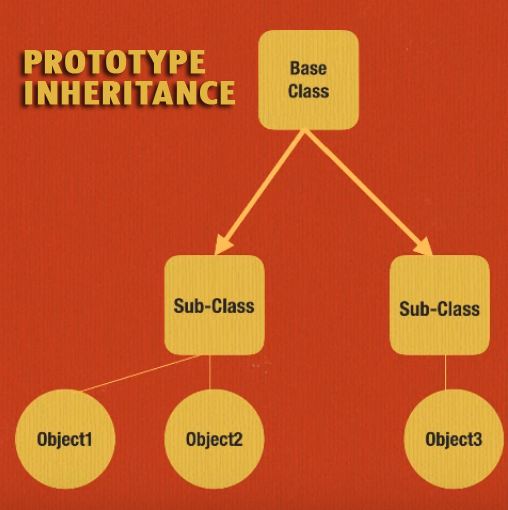
**Prototype Chain:**

console.dir(Person);

Then it will show details about prototype chain.



If we create an object from x then all of the properties and methods from the prototype chain will be inherited to that object like this

New topics to learn:

1. Creating Sub Class (Sub Constructor)
2. Overriding in Prototype Chain
3. Adding prototype to Master Object

**Base Class:**

//Base class constructor

let Job = function ()

{

this.pays = true;

}

// Job Class’s Prototype Method

Job.prototype.print = function()

{

console.log(this.pays ? 'Please hire me' : 'no thank you');

}

// subclass constructor

let TechJob = function(title,pays)

{

Job.call(this);

this.titile = title;

this.pays = pays;

}

// inheritance

TechJob.prototype = Object.create(Job.prototype);

TechJob.prototype.constructor = TechJob;

// method overriding

Job.prototype.print = function()

{

console.log(this.pays ? 'Great Job. Please hire me' : 'No thank you');

}

// Adding Method to Master Object’s Prototype

Object.prototype.print = function()

{

console.log("Hi, From Master");

}

var softwarePosition = new TechJob('Javascript Programmer', true);

var softwarePosition2 = new TechJob('PHP Programmer', false);

console.log(softwarePosition.print());

console.log(softwarePosition2.print());

**N.B**, "Object" is called the master or root object. All objects in Javascript are created from "Object". The Object's prototype methods and properties are available to all objects.

**Summary**:

When we are calling the print method using softwarePosition object it first checkes the method of softwarePosition is its own or not if not it will look into its prototype if not found it will look for its parent class or constructor if not found it will search into master object if not found the method then javascript interpreter will throw an error message.

Classes

Classes are higher order functions and class expressions can be named or unnamed & classes use prototype-based inheritance. The class declaration creates a new class with a given name using prototype-based inheritance.

Class Declaration Syntax:

class name [extends] {

// class body

}

**Example of Class Declaration:**

class Polygon {

constructor(height, width) {

this.area = height \* width;

}

}

console.log(new Polygon(4,3).area);

// expected output: 12

**Class Expression Syntax:**

var MyClass = class [className] [extends] {

// class body

};

**Example of Class Expression:**

var Rectangle = class {

constructor(height, width) {

this.height = height;

this.width = width;

}

area() {

return this.height \* this.width;

}

}

console.log(new Rectangle(5,8).area());

// expected output: 40

**Example 01:**

// unnamed

let Rectangle = class {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

console.log(Rectangle.name);

// output: "Rectangle"

// named

let Rectangle = class Rectangle2 {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

console.log(Rectangle.name);

// output: "Rectangle2"

**Example 02:**

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

// Getter

get area() {

return this.calcArea();

}

// Method

calcArea() {

return this.height \* this.width;

}

}

const square = new Rectangle(10, 10);

console.log(square.area); // 100

**Instance properties:**

Instance properties must be defined inside of class methods:

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

}

Static properties and prototype data properties must be defined outside of the ClassBody declaration:

Rectangle.staticWidth = 20;

Rectangle.prototype.prototypeWidth = 25;

**Public field declarations**

With the JavaScript field declaration syntax, the above example can be written as:

class Rectangle {

height = 0;

width;

constructor(height, width) {

this.height = height;

this.width = width;

}

}

**Private field declarations**

Using private fields, the definition can be refined as below.

class Rectangle {

#height = 0;

#width;

constructor(height, width) {

this.#height = height;

this.#width = width;

}

}

N.B, The Private fields can only be read or written within the class body. By defining things which are not visible outside of the class. Private fields cannot be created later through assigning to them, the way that normal properties can.

**Sub classing with extends**

The extends keyword is used in class declarations or class expressions to create a class as a child of another class.

class Animal {

constructor(name) {

this.name = name;

}

speak() {

console.log(this.name + ' makes a noise.');

}

}

class Dog extends Animal {

constructor(name) {

super(name); // call the super class constructor and pass in the name parameter

}

speak() {

console.log(this.name + ' barks.');

}

}

let d = new Dog('Mitzie');

d.speak(); // Mitzie barks.

**N.B**, If there is a constructor present in the subclass, it needs to first call super() before using "this".

**The same by extending with traditional function-based "classes":**

function Animal (name) {

this.name = name;

}

Animal.prototype.speak = function () {

console.log(this.name + ' makes a noise.');

}

class Dog extends Animal {

speak() {

console.log(this.name + ' barks.');

}

}

let d = new Dog('Mitzie');

d.speak(); // Mitzie barks.

**Regular Objects Extending to the SubClass:**

classes cannot extend regular objects. If you want to inherit from a regular object, you can instead use Object.setPrototypeOf().

const Animal = {

speak() {

console.log(this.name + ' makes a noise.');

}

};

class Dog {

constructor(name) {

this.name = name;

}

}

// If you do not do this you will get a TypeError when you invoke speak

Object.setPrototypeOf(Dog.prototype, Animal);

let d = new Dog('Mitzie');

d.speak(); // Mitzie makes a noise.

**Another use of super:**

class Cat {

constructor(name) {

this.name = name;

}

speak() {

console.log(`${this.name} makes a noise.`);

}

}

class Lion extends Cat {

speak() {

super.speak();

console.log(`${this.name} roars.`);

}

}

let l = new Lion('Fuzzy');

l.speak();

// Fuzzy makes a noise.

// Fuzzy roars.

**Composition over Inheritance**

The Object.assign() method is used to copy (not inherit) the properties & methods from one or more source objects to a target object. Finally, It will return the target object. So, this way is called composition.

**Syntax:**

Object.assign(target, ...sources)

**target:**

The target object.

**sources:**

The source object(s).

**Example 01:**

const object1 = { a: 1, b: 2,c: 3};

const object2 = {c: 4, d: 5};

const object3 = Object.assign({}, object1, object2);

console.log(object3.b, object3.c, object3.d);

// expected output: 2 4 5

**Example 02:**

const object1 = {a: 1, b: 2, c: 3};

const object2 = Object.assign({c: 4, d: 5}, object1);

console.log(object2.c, object2.d);

// expected output: 3 5

**Example 03:**

var o1 = { a: 1 };

var o2 = { b: 2 };

var o3 = { c: 3 };

var obj = Object.assign(o1, o2, o3);

console.log(obj); // { a: 1, b: 2, c: 3 }

console.log(o1); // { a: 1, b: 2, c: 3 }, target object itself is changed.

**Example 04:**

obj1 = { a: 0 , b: { c: 0}};

let obj3 = JSON.parse(JSON.stringify(obj1));

obj1.a = 4;

obj1.b.c = 4;

console.log(JSON.stringify(obj3)); // { a: 0, b: { c: 0}}