**04 Prototypical Inheritance**

**1) Creating Your Own Prototypical Inheritance**:

Suppose we have a Circle object and in this circle object we have an instance property “radius” and two prototype property “draw()” and “duplicate()”

function Circle(radius) {

*this*.radius = radius;

}

Circle.prototype.draw = function() {

console.log("draw");

};

Circle.prototype.duplicate = function() {

console.log("duplicate");

};

Now we want to add a Square object here that Square object also have a duplicate method with the exact same implementation. We don’t want to repeat the implementation. We want to inheritance.

So, we define a “Shape” object and put this duplicate method here, and then have Square and Circle inherited from the Shape object.

**Example**:

function Shape() {}

Shape.prototype.duplicate = function() {

console.log("duplicate");

};

function Circle(radius) {

*this*.radius = radius;

}

Circle.prototype.draw = function() {

console.log("draw");

};

const s = new Shape();

const c = new Circle(20);

console.log(s);

*/\**

*Shape {}*

*\_\_proto\_\_:*

*duplicate: ƒ ()*

*constructor: ƒ Shape()*

*\_\_proto\_\_: Object //objectBase*

*\*/*

console.log(c);

*/\**

*Circle {radius: 20}*

*radius: 20*

*\_\_proto\_\_:*

*draw: ƒ ()*

*constructor: ƒ Circle(radius)*

*\_\_proto\_\_: Object //objectBase*

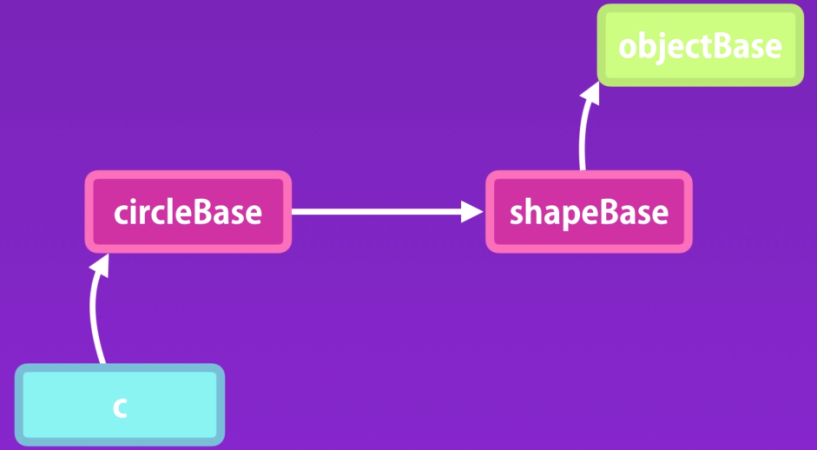
*\*/*

Here in memory

c (Object)-----(inherit from)----->circelBase(Circle.prototype)-----(inherit from)----->objectBase

s (Object)-----(inherit from)----->shapeBase(Shape.prototype)-----(inherit from)----->objectBase

Now to set up inheritance here, we want to have circleBase inherit from shapeBase.



**Example**:

function Shape() {}

Shape.prototype.duplicate = function() {

console.log("duplicate");

};

function Circle(radius) {

*this*.radius = radius;

}

*//circleBase is like this*

*//Circle.prototype = Object.create(Object.prototype); //objectBase*

*//Create ShapeBase prototype and set to to CircleBase*

Circle.prototype = Object.create(Shape.prototype);

Circle.prototype.draw = function() {

console.log("draw");

};

const s = new Shape();

const c = new Circle(20);

console.log(s);

*/\**

*Shape {}*

*\_\_proto\_\_:*

*duplicate: ƒ ()*

*constructor: ƒ Shape()*

*\_\_proto\_\_: Object*

*\*/*

console.log(c);

*/\**

*Circle {radius: 20}*

*radius: 20*

*\_\_proto\_\_: Shape*

*draw: ƒ ()*

*\_\_proto\_\_:*

*duplicate: ƒ ()*

*constructor: ƒ Shape()*

*\_\_proto\_\_: Object*

*\*/*

This is prototypical inheritance.

**2) Resetting the Constructor**:

Every object in JavaScript have a constructor property, that returns the function that was use to construct or create that object.

We can create object by using prototype constructor

**Example**:

function Shape() {}

Shape.prototype.duplicate = function() {

console.log("duplicate");

};

function Circle(radius) {

*this*.radius = radius;

}

Circle.prototype.draw = function() {

console.log("draw");

};

const s = new Shape();

const c = new Circle(20);

console.log(new Circle.prototype.constructor(20));

*/\**

*Circle {radius: 20}*

*radius: 20*

*\_\_proto\_\_:*

*draw: ƒ ()*

*constructor: ƒ Circle(radius)*

*\_\_proto\_\_: Object*

*\*/*

Here, "new Circle.prototype.constructor(20)" is equivalent to “new Circle(20)”

**Example**:

console.log(new Circle(20));

*/\**

*Circle {radius: 20}*

*radius: 20*

*\_\_proto\_\_:*

*draw: ƒ ()*

*constructor: ƒ Circle(radius)*

*\_\_proto\_\_: Object*

*\*/*

But now we replace the circleBase with shapeBase.

Circle.prototype = Object.create(Shape.prototype);

Now the constructor property of Circle is not present in Circle object. It replaces with shapeBase. The constructor property is now the Shape() function.

**Example**:

function Shape() {}

Shape.prototype.duplicate = function() {

console.log("duplicate");

};

function Circle(radius) {

*this*.radius = radius;

}

Circle.prototype = Object.create(Shape.prototype);

Circle.prototype.draw = function() {

console.log("draw");

};

const c = new Circle(20);

console.log(new Circle(20));

*/\**

*Circle {radius: 20}*

*radius: 20*

*\_\_proto\_\_: Shape*

*draw: ƒ ()*

*\_\_proto\_\_:*

*duplicate: ƒ ()*

*constructor: ƒ Shape()*

*\_\_proto\_\_: Object*

*\*/*

Here constructor property is the Shape() function not the Circle function. Now we cannot create a circle object based on the constructor in dynamic way.

**Example**:

console.log(new Circle.prototype.constructor()); //Shape {}

*/\**

*Shape {}*

*\_\_proto\_\_:*

*duplicate: ƒ ()*

*constructor: ƒ Shape()*

*\_\_proto\_\_: Object*

*\*/*

The reason behind this we reset the prototype of the Circle and replace with Shape object.

For resolve this the best practice is, whenever we reset the prototype of an object, we should also reset the constructor. Now the constructor reference to the Circle function

Circle.prototype = Object.create(Shape.prototype);

Circle.prototype.constructor = Circle;

console.log(new Circle.prototype.constructor());

*/\**

*Circle {radius: undefined}*

*radius: undefined*

*\_\_proto\_\_: Shape*

*constructor: ƒ Circle(radius)*

*draw: ƒ ()*

*\_\_proto\_\_: Object*

*\*/*

**3) Calling the Super Constructor**:

Here we have a Shape object and a Circle object. We reset the prototype of Circle object with Shape object. After that we also reset the constructor of Circle object with Circle constructor.

**Example**:

function Shape() {}

Shape.prototype.duplicate = function() {

console.log("duplicate");

};

function Circle(radius) {

*this*.radius = radius;

}

Circle.prototype = Object.create(Shape.prototype);

Circle.prototype.constructor = Circle;

Circle.prototype.draw = function() {

console.log("draw");

};

Now we modify the Shape() constructor and introduce a “color” parameter. We want every Shape object to have a color.

function Shape(color) {

*this*.color = color;

}

Now when we create a Circle object, we only pass the "radius" property. There is no "color" here

const c = new Circle(20);

console.log(c);

*/\**

*Circle {radius: 20}*

*radius: 20*

*\_\_proto\_\_: Shape*

*\*/*

Now from an inheritance point of view this circle object should have a color property, and that should be initialized at the time of creating a Circle object.

Now for do this we have to call the Shape() constructor inside Circle() constructor. But it don’t work if we implements this in the following way.

**Example**:

function Shape(color) {

*this*.color = color;

}

Shape.prototype.duplicate = function() {

console.log("duplicate");

};

function Circle(radius, color) {

Shape(color);

*this*.radius = radius;

}

Circle.prototype = Object.create(Shape.prototype);

Circle.prototype.constructor = Circle;

Circle.prototype.draw = function() {

console.log("draw");

};

const c = new Circle(20, "red");

console.log(c);

*/\**

*Circle {radius: 20}*

*radius: 20*

*\_\_proto\_\_: Shape*

*constructor: ƒ Circle(radius, color)*

*draw: ƒ ()*

*\_\_proto\_\_: Object*

*\*/*

Here in circle object we have only the color "radius" property we don’t have any "color" property. Here the color property is not working.

Here when we use the "new" operator three thing happened.

1. This "new" operator create an empty JavaScript object like Circle ={};
2. Next it will set "this" (like radius this.radius = radius;) to point the empty object
3. Finally, the new keyword returns the object from the constructor function. it is implicitly we no need to write the return code

If we don’t use the "new" operator then "this" by default will be point to the global object which is "window" in browser and "global" in node.

The above "color" property is not working because when we call the "Shape(color)" in "this.color = color" it will point to the global object.

**Example**:

function Shape(color) {

*this*.color = color; *//this refer to window object*

}

Shape.prototype.duplicate = function() {

console.log("duplicate");

};

function Circle(radius, color) {

Shape(color); *// without new operator it point to window*

*this*.radius = radius;

}

Circle.prototype = Object.create(Shape.prototype);

Circle.prototype.constructor = Circle;

Circle.prototype.draw = function() {

console.log("draw");

};

const c = new Circle(20, "red");

console.log(window.color);*//red*

Now we have to fix this problem. If we use "new" operator before Shape() object it will create another new object and then set the color property on that object

function Circle(radius, color) {

*//create ew object and then set the color property on that object*

new Shape(color);

*this*.radius = radius;

}

Now to fix this problem we have to call the "Shape()" function inside "Circle" function and set "this" (inside this.color = color) inside the "Shape()" function and point to the Circle object.

**Example**:

function Shape(color) {

*this*.color = color;

}

Shape.prototype.duplicate = function() {

console.log("duplicate");

};

function Circle(radius, color) {

Shape.call(*this*, color);

*this*.radius = radius;

}

Circle.prototype = Object.create(Shape.prototype);

Circle.prototype.constructor = Circle;

Circle.prototype.draw = function() {

console.log("draw");

};

const c = new Circle(20, "red");

console.log(c); *//Circle {color: "red", radius: 20}*

*/\**

*Circle {color: "red", radius: 20}*

*color: "red"*

*radius: 20*

*\_\_proto\_\_: Shape*

*\*/*

Now in Circle object we have two properties "color" and "radius". This is call super constructor.

**call() function**:

The call() method calls a function with a given this value and arguments provided individually.

**4) Intermediate Function Inheritance**:

In the above example we have set up our inheritance chain properly. Now creates another object

"square" that inherits from the shape.

**Example**:

function Shape(color) {

*this*.color = color;

}

Shape.prototype.duplicate = function() {

console.log("duplicate");

};

function Circle(radius, color) {

Shape.call(*this*, color);

*this*.radius = radius;

}

Circle.prototype = Object.create(Shape.prototype);

Circle.prototype.constructor = Circle;

Circle.prototype.draw = function() {

console.log("draw");

};

function Square(size) {

*this*.size = size;

}

Square.prototype = Object.create(Shape.prototype);

Square.prototype.constructor = Square;

const s = new Square(20);

console.log(s); *//Circle {color: "red", radius: 20}*

*/\**

*Square {size: 20}*

*size: 20*

*\_\_proto\_\_: Shape*

*constructor: ƒ Square(size)*

*\_\_proto\_\_:*

*duplicate: ƒ ()*

*constructor: ƒ Shape(color)*

*\_\_proto\_\_: Object*

*\*/*

Now we simplify the prototype chain code. Let refactor this code and create a function that can be reuse.

function extend(Child, Parent) {

Child.prototype = Object.create(Parent.prototype);

Child.prototype.constructor = Child;

}

**Example**:

function Shape(color) {

*this*.color = color;

}

Shape.prototype.duplicate = function() {

console.log("duplicate");

};

function extend(Child, Parent) {

Child.prototype = Object.create(Parent.prototype);

Child.prototype.constructor = Child;

}

function Circle(radius, color) {

Shape.call(*this*, color);

*this*.radius = radius;

}

extend(Circle, Shape);

Circle.prototype.draw = function() {

console.log("draw");

};

extend(Square, Shape);

function Square(size) {

*this*.size = size;

}

const sq = new Square(10);

console.log(sq);

*/\**

*Square {size: 10}*

*size: 10*

*\_\_proto\_\_: Shape*

*constructor: ƒ Square(size)*

*\_\_proto\_\_: Object*

*\*/*

This function is call Intermediate Function Inheritance.

function extend(Child, Parent) {

Child.prototype = Object.create(Parent.prototype);

Child.prototype.constructor = Child;

}

**5) Method Overriding**:

Overriding means reimplementing a method. JavaScript supports overriding, so if you define two functions with the same name, the last one defined will override the previously defined version and every time a call will be made to the function, the last defined one will get executed.

Consider the following example

function extend(Child, Parent) {

Child.prototype = Object.create(Parent.prototype);

Child.prototype.constructor = Child;

}

function Shape() {}

function Circle() {}

extend(Circle, Shape);

Shape.prototype.duplicate = function() {

console.log("duplicate");

};

function Circle() {}

*//Circle inherits from Shape*

extend(Circle, Shape);

const c = new Circle();

console.log(c.duplicate()); *//duplicate*

Suppose the duplicate() method is not working properly for the Circle function. We have to modify the duplicate() method for the Circle object. For this we can use method overriding. Override a method that is defined in the base object.

**Example**:

function extend(Child, Parent) {

Child.prototype = Object.create(Parent.prototype);

Child.prototype.constructor = Child;

}

function Shape() {}

function Circle() {}

extend(Circle, Shape);

Shape.prototype.duplicate = function() {

console.log("duplicate");

};

function Circle() {}

*//Circle inherit from Shape*

extend(Circle, Shape);

*//must be places after "extend(Circle, Shape);"*

*//method overriding*

Circle.prototype.duplicate = function() {

console.log("duplicate-circle");

};

const c = new Circle();

console.log(c.duplicate()); *//duplicate-circle*

In JavaScript when we access a property or a method in an object JavaScript engine works up the prototype chain and picks the first implementation. Here we implement the duplicate() method both on Parent and Child objects, the implementation on the child object will be use.

Sometime we have to call the parent implementation also. For do this we have to modify the overridden method as follow.

**Example**:

Circle.prototype.duplicate = function() {

Shape.prototype.duplicate.call(*this*);

console.log("duplicate-circle");

};

const c = new Circle();

console.log(c.duplicate());

*/\**

*duplicate*

*duplicate-circle*

*\*/*

**6)** **Polymorphism**:

Poly means many and morphism means form. So, Polymorphism means many forms. Polymorphism is a very powerful technique in object-oriented programming language.

Consider the following example

function extend(Child, Parent) {

Child.prototype = Object.create(Parent.prototype);

Child.prototype.constructor = Child;

}

function Shape() {}

function Circle() {}

Shape.prototype.duplicate = function() {

console.log("duplicate");

};

extend(Circle, Shape);

Circle.prototype.duplicate = function() {

console.log("duplicate-circle");

};

function Square() {}

extend(Square, Shape);

Square.prototype.duplicate = function() {

console.log("duplicate-square");

};

Here we have a Shape object and two child object Circle and Square. Each object provides a different implementation of duplicate method. So, we have many implementations of duplicate method. This concept is called polymorphism.

Suppose we have an array of Safe object. This shape object can hold Circle and Square object.

**Example**:

function extend(Child, Parent) {

Child.prototype = Object.create(Parent.prototype);

Child.prototype.constructor = Child;

}

function Shape() {}

function Circle() {}

Shape.prototype.duplicate = function() {

console.log("duplicate");

};

extend(Circle, Shape);

Circle.prototype.duplicate = function() {

console.log("duplicate-circle");

};

function Square() {}

extend(Square, Shape);

Square.prototype.duplicate = function() {

console.log("duplicate-square");

};

const shapes = [new Circle(), new Square()];

for (let shape of shapes) {

shape.duplicate();

}

*/\**

*duplicate-circle*

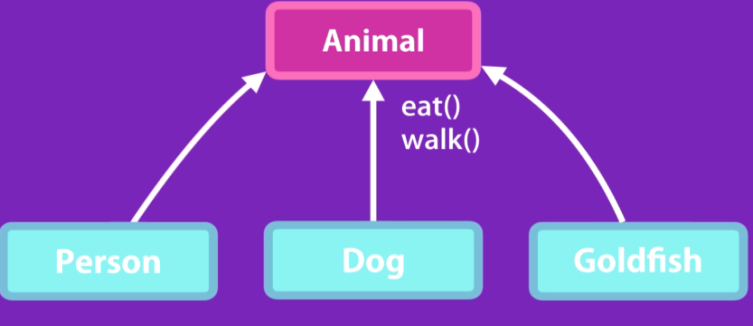
*duplicate-square*

*\*/*

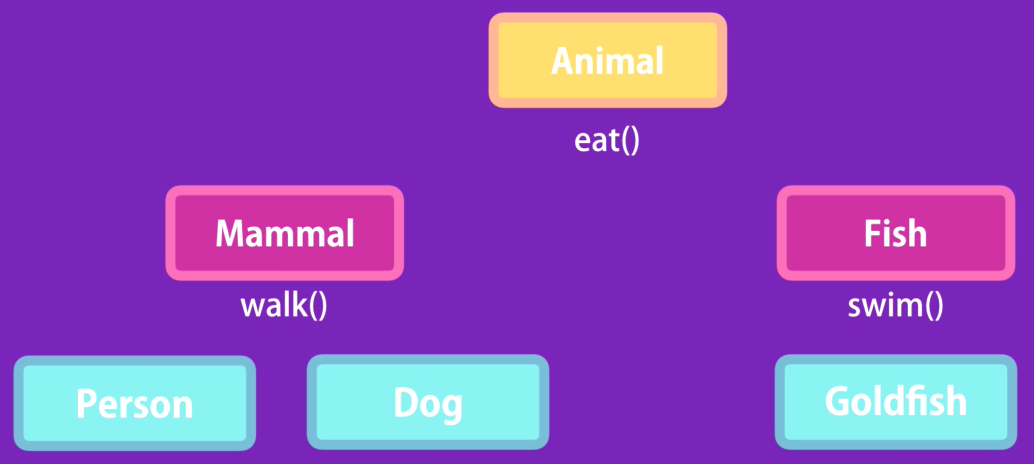
**7) When to Use Inheritance**:

When we are working with JavaScript, we have to tack care about it. Because it may make our code complex. So, while working in a small project it is not recommended to use inheritance. We have to remember that inheritance is not the only feature where we can reuse code.

Suppose we have an Animal object with two method eat() and walk(). And we have two object that derive from Animal Person and Dog. Later we have a new object called Goldfish that derives from Animal. In this case our hierarchy is broken because Goldfish cannot walk.



To solve this problem, we have to solve our hierarchy. We ha to do the following thing.



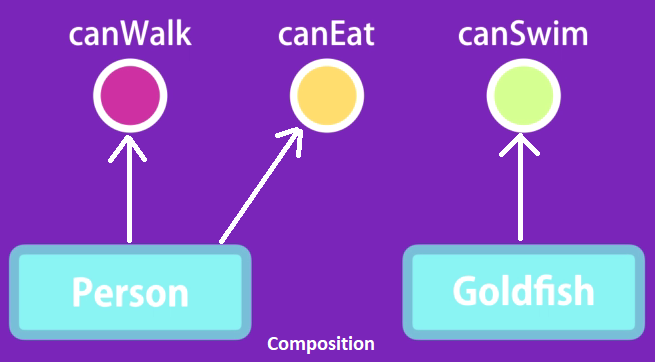
So, always avoid creating inheritance hierarchies. Because they can make our code complex. When use inheritance keep it one level. Never go more than one level.

**Composition**:

"Favor Composition over Inheritance". By compassion we can compose a few objects with together to create a new object. This technique gives us big flexibility.

Suppose we have three object "canWalk", "canEat", and "canSwim". Each of the object is a plane JavaScript object with some property and method. Now we have a Person object which compose canWalk and canEat. Also, we have a Goldfish object which compose canSwim.

In JavaScript we can use Mixins to achieve compassion.



**8) Mixins**:

Mixin is a way property are added to objects without using inheritance — Darren Jones

Mixins are a form of object composition, where component features get mixed into a composite object so that properties of each mixin become properties of the composite object. — Eric Elliot

Link: https://blog.bitsrc.io/understanding-mixins-in-javascript-de5d3e02b466

**Example**:

const canEat = {

eat: function() {

*this*.hunger--;

console.log("Eating");

}

};

const canWalk = {

walk: function() {

console.log("Walking");

}

};

const person = Object.assign({}, canEat, canWalk);

console.log(person);

*/\**

*{eat: ƒ, walk: ƒ}*

*eat: ƒ ()*

*walk: ƒ ()*

*\_\_proto\_\_: Object*

*\*/*

Now we want a constructor function in the above example. For this we have to define a Person object and pass this object in the place of empty object.

**Example**:

const canEat = {

eat: function() {

*this*.hunger--;

console.log("Eating");

}

};

const canWalk = {

walk: function() {

console.log("Walking");

}

};

function Person() {}

*//we are modifying the prototype of person and added canEat and canWalk*

Object.assign(Person.prototype, canEat, canWalk);

*//Create a person object*

const person = new Person();

console.log(person);

*/\**

*Person {}*

*\_\_proto\_\_:*

*eat: ƒ ()*

*walk: ƒ ()*

*constructor: ƒ Person()*

*\_\_proto\_\_: Object*

*\*/*

Now we want to add two new objects in this application, Dog and Goldfish. Both Dog and Goldfish have capability to swing. Now we added the swing method in our application. Now we define a new constructor Goldfish().

const canSwim = {

swim: function(){

console.log("Swim");

}

};

function Goldfish() {}

Object.assign(Goldfish.prototype, canEat, canSwim);

**Example**:

const canEat = {

eat: function() {

*this*.hunger--;

console.log("Eating");

}

};

const canWalk = {

walk: function() {

console.log("Walking");

}

};

const canSwim = {

swim: function() {

console.log("Swim");

}

};

function Person() {}

Object.assign(Person.prototype, canEat, canWalk);

function Goldfish() {}

Object.assign(Goldfish.prototype, canEat, canSwim);

const person = new Person();

const goldfish = new Goldfish();

console.log(person);

console.log(goldfish);

*/\**

*Person {}*

*\_\_proto\_\_:*

*eat: ƒ ()*

*walk: ƒ ()*

*constructor: ƒ Person()*

*\_\_proto\_\_: Object*

*Goldfish {}*

*\_\_proto\_\_:*

*eat: ƒ ()*

*swim: ƒ ()*

*constructor: ƒ Goldfish()*

*\_\_proto\_\_: Object*

*\*/*

Now to make this code "Object.assign(Goldfish.prototype, canEat, canSwim)" more readable we can extract this logic into a function.

**Example**:

function mixin(target, ...sources) {

Object.assign(target, ...sources);

}

As the second parameter of "mixin()" function we are using the rest operator "…" … resources. Because this operator can take any number of parameters and convert them into an array.

Now in "Object.assign ()" method we pass the target and here we have to pass "…sources" explicitly as an element not an array. This time we can use "…" spread operator to spread an array into multiple arguments.

Now we can simplify our code.

**Example**:

function mixin(target, ...sources) {

Object.assign(target, ...sources);

}

const canEat = {

eat: function() {

*this*.hunger--;

console.log("Eating");

}

};

const canWalk = {

walk: function() {

console.log("Walking");

}

};

const canSwim = {

swim: function() {

console.log("Swim");

}

};

function Person() {}

mixin(Person.prototype, canEat, canWalk);

function Goldfish() {}

mixin(Goldfish.prototype, canEat, canSwim);

const person = new Person();

const goldfish = new Goldfish();

console.log(person);

console.log(goldfish);

*/\**

*Person {}*

*\_\_proto\_\_:*

*eat: ƒ ()*

*walk: ƒ ()*

*constructor: ƒ Person()*

*\_\_proto\_\_: Object*

*Goldfish {}*

*\_\_proto\_\_:*

*eat: ƒ ()*

*swim: ƒ ()*

*constructor: ƒ Goldfish()*

*\_\_proto\_\_: Object*

*\*/*

**1) Exercise Prototypical Inheritance**:

Design two objects, one is HtmlElement and another is HtmlSelectElement. We have prototypical inheritance between HtmlSelectElement and its parent HtmlElement.

In HtmlElement we have a instance property “click()” and a prototypical property “focus()” property. In HtmlSelectElement we have three property items, addItem(), and removeItem().

Now write a program that can do the following thing.

*/\**

*const e = new HtmlElement()*

*e*

*HtmlElement {click: ƒ}*

*click: ƒ ()*

*\_\_proto\_\_:*

*focus: ƒ ()*

*constructor: ƒ HtmlElement()*

*\_\_proto\_\_: Object*

*e.click()*

*clicked*

*e.focus()*

*focus*

*\*/*

*/\**

*const e = new HtmlSelectElement();*

*e*

*HtmlSelectElement {items: Array(0), addItem: ƒ, removeItem: ƒ}*

*addItem: ƒ (item)*

*items: []*

*removeItem: ƒ (item)*

*\_\_proto\_\_: HtmlElement*

*e.addItem("1")*

*e.addItem("2")*

*e.removeItem("2")*

*e*

*HtmlSelectElement {items: Array(1), addItem: ƒ, removeItem: ƒ}*

*addItem: ƒ (item)*

*items: ["1"]*

*removeItem: ƒ (item)*

*\_\_proto\_\_: HtmlElement*

*\*/*

**1) Solution Prototypical Inheritance**:

**Solution**:

function HtmlElement() {

*this*.click = function() {

console.log("clicked");

};

}

HtmlElement.prototype.focus = function() {

console.log("focus");

};

*//we initialized the item parameter with an empty array (ES-6)*

*/\**

*Before ES-6*

*function HtmlSelectElement(items) {*

*this.items = items || [];*

*\*/*

function HtmlSelectElement(items = []) {

*this*.items = items;

*this*.addItem = function(item) {

*this*.items.push(item);

};

*this*.removeItem = function(item) {

*this*.items.splice(*this*.items.indexOf(item), 1);

};

}

*/\**

*HtmlSelectElement.prototype = Object.create(HtmlElement.prototype);*

*dont work here. because "Object.create()" method create q new object and set the prototype of the object to the prototype of "HtmlElement"*

*In HtmlElement prototype there is only one member "focus", but "click" in not in the prototype.*

*So, to get both instance and prototypical member we use "HtmlSelectElement.prototype = new HtmlElement();"*

*\*/*

HtmlSelectElement.prototype = new HtmlElement(); *//baseHtmlElement*

*//set constructor*

HtmlSelectElement.prototype.constructor = HtmlSelectElement;

**2) Exercise- Polymorphism**:

**2) Solution- Polymorphism**:

function HtmlElement() {

*this*.click = function() {

console.log('clicked');

}

}

HtmlElement.prototype.focus = function(){

console.log('focued');

}

function HtmlSelectElement(items = []) {

*this*.items = items;

*this*.addItem = function(item) {

*this*.items.push(item);

}

*this*.removeItem = function(item) {

*this*.items.splice(*this*.items.indexOf(item), 1);

}

*this*.render = function() {

return `

<select>${*this*.items.map(item => `

<option>${item}</option>`).join('')}

</select>`;

}

}

HtmlSelectElement.prototype = new HtmlElement();

HtmlSelectElement.prototype.constructor = HtmlSelectElement;

function HtmlImageElement(src) {

*this*.src = src;

*this*.render = function() {

return `<img src="${*this*.src}" />`

}

}

HtmlImageElement.prototype = new HtmlElement();

HtmlImageElement.prototype.constructor = HtmlImageElement;

const elements = [

new HtmlSelectElement([1, 2, 3]),

new HtmlImageElement('http://')

];

for (let element of elements)

console.log(element.render());

04 Prototypical Inheritance