

**Creating your own Prototypical Inheritance**:

So here we have a *Circle* constructor with radius property and two methods *draw* and *duplicate*, that are defined in its prototype.

function Circle(radius) {

  this.radius = radius;

}

Circle.prototype.draw = function () {

  console.log("draw");

};

Circle.prototype.duplicate = function(){

  console.log("duplicate")

}

Imagine if tomorrow we are going to add a *square* object here which should have this same *duplicate* method as well.

We do not want to want to define a Square.prototype.duplicate as well. Instead we need to use inheritance.

So we will define a *Shape* constructor and put the *duplicate* method on its prototype and let *Circle* and *Square* inherit it.

function Shape() {}

Shape.prototype.duplicate = function () {

  console.log("duplicate");

};

function Circle(radius) {

  this.radius = radius;

}

Circle.prototype.draw = function () {

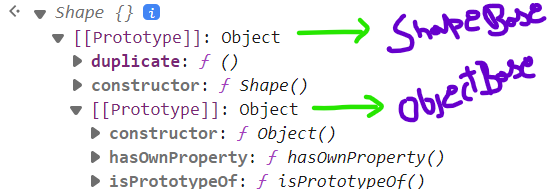
  console.log("draw");

};

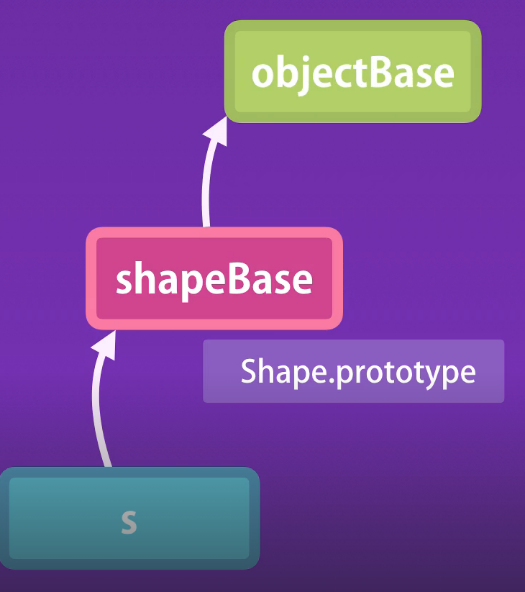
const s = new Shape();

const c = new Circle(1);

In the console,

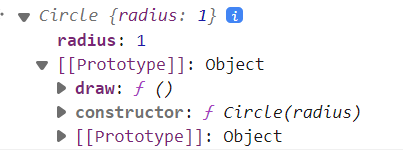


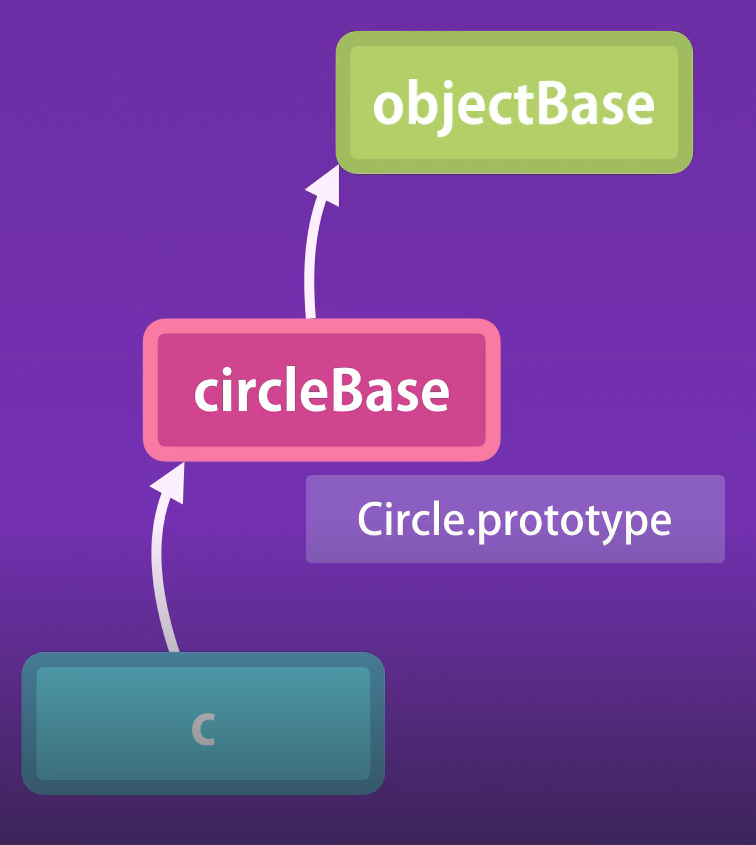
We have our shape object, which inherits from the object we can call *shapeBase* and in shapeBase we have duplicate method.



Furthermore shapeBase inherits from our objectBase which is the root of all objects in JavaScript.

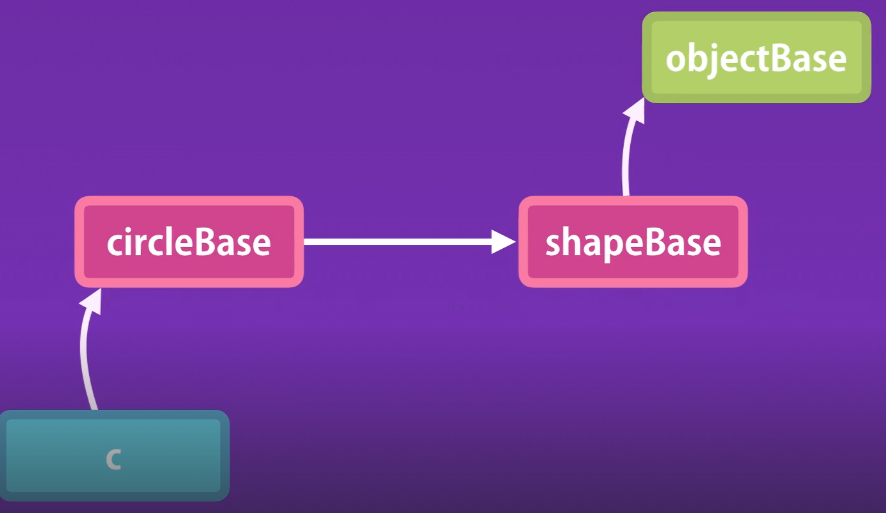
We have the exact same structure in our circle,



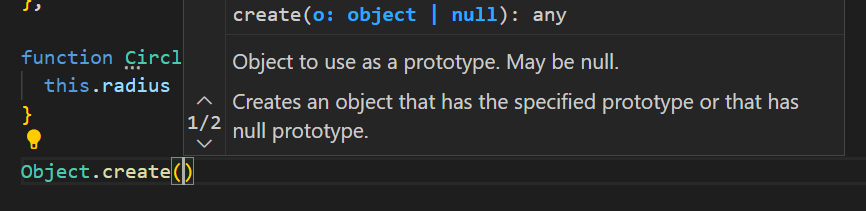
 🡨Here, circleBase is basically circle.prototype.

*To setup inheritance here*…

We want to have circleBase inherit from shapeBase.



In JavaScript we have a method for creating an object with a given prototype.



Look at the argument here, “Object to use as a prototype”. So we need a new circleBase object that inherits from shapeBase instead of objectBase.

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

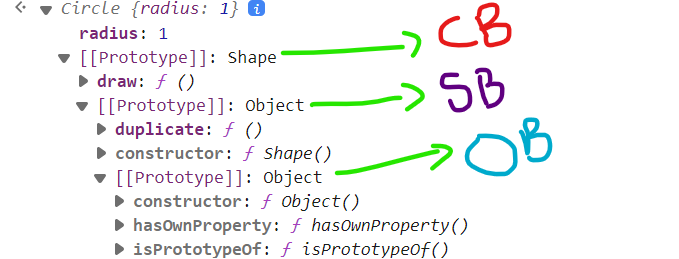
// This is the new relation between circleBase and shapeBase

Before this line our circleBase was like this,

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

// This was the implicit relationship between objectBase and circleBase

Now if you look at the console,



We have shapeBase as the parent of the circleBase, which has the duplicate method. Now our circle object has duplicate method as well.



Complete code:

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 s = new Shape();

const c = new Circle(1);

This is prototypical inheritance in action.

**Resetting the constructor**:

There is a tiny problem in this implementation

First let us remove shapeBase as the parent of the circleBase.

function Shape() {}

Shape.prototype.duplicate = function () {

  console.log("duplicate");

};

function Circle(radius) {

  this.radius = radius;

}

//by Removing this below line, shapeBase is no longer prototype of circleBase

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

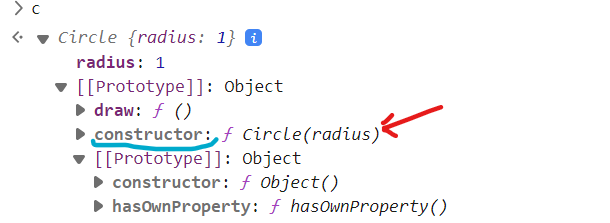
Circle.prototype.draw = function () {

  console.log("draw");

};

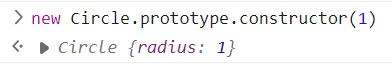
const c = new Circle(1);

And look at the properties of the circle object.



We know that every JavaScript object has a constructor property that returns the function that was used to construct or create that object.

So in the circle object we can see the constructor property which references our *Circle* constructor function. With this we can create a *new* circle object like this,

 ==

Both expressions produce the same result.

Generally we use the second form to create an object because it is shorter and cleaner, but *in some rare circumstances we may get a constructor function somewhere in your application and we may want to dynamically create an object based on that constructor function*.

*In such cases we can access the prototype property to get the constructor and then use the new operator to create the object*.

Now we will uncomment the *Object.create* method line and see what happens,

function Shape() {}

Shape.prototype.duplicate = function () {

  console.log("duplicate");

};

function Circle(radius) {

  this.radius = radius;

}

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

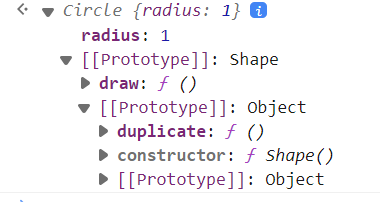
Circle.prototype.draw = function () {

  console.log("draw");

};

const c = new Circle(1);

In the console,



We no longer have the constructor property in the circleBase but we can see constructor property in shapeBase (*which is returning Shape function*).

In simple words with what we have now, we can no longer create circle object based on its constructor in a dynamic fashion.

🡨 Instead of Circle { }

The reason we are having this problem is because we reset the prototype of the circle using *Object.create* method.

So *the best practice is that whenever we reset the prototype of an object, we should also reset the constructor*.

function Shape() {}

Shape.prototype.duplicate = function () {

  console.log("duplicate");

};

function Circle(radius) {

  this.radius = radius;

}

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

Circle.prototype.construtor = Circle; //set the constructor to Circle

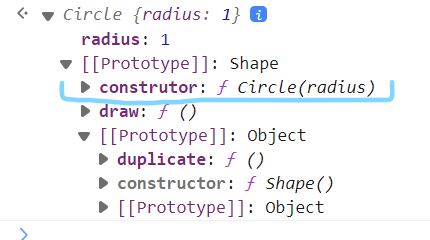
Circle.prototype.draw = function () {

  console.log("draw");

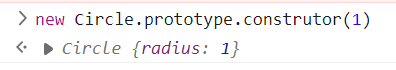
};

const s = new Shape();

const c = new Circle(1);

Now we can see constructor is referencing the Circle function.

And we can dynamically create circle object using constructor.



**Calling the Super constructor**:

Now let us take this example to the next level. We will modify the Shape constructor and introduce the *color* parameter. The idea is to have every shape (*whether square or circle*) a color.

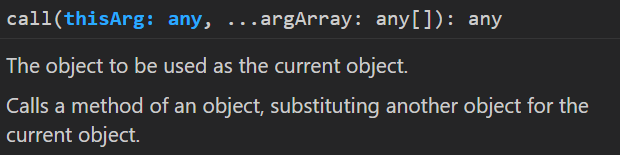
function Shape(color) {

  this.color = color

}

Now how do we make color property from Shape pass to Circle using inheritance?

Inside Circle constructor we call the Shape constructor with *call* method:



function Shape(color) {

  this.color = color;

}

Shape.prototype.duplicate = function () {

  console.log("duplicate");

};

function Circle(radius, color) { //1. pass color as parameter

  Shape.call(this, color); //2. use call method on Shape object

  this.radius = radius;

}

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

Circle.prototype.construtor = Circle;

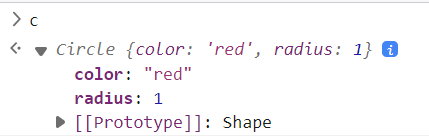
Circle.prototype.draw = function () {

  console.log("draw");

};

const c = new Circle(1, "red"); //3. pass “red” color as argument

In the console,



**Intermediate Function inheritance**:

So we have setup our inheritance chain properly. Now let us create another object like a Square that inherits from the shape.

function Square(size){

  this.size = size;

}

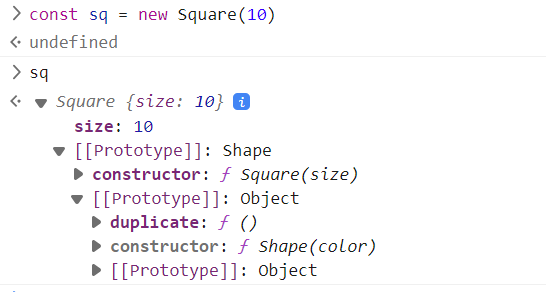
Now we want this *Square* to inherit from *Shape* constructor. So we will reset its prototype like we did with *Circle*.

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

Square.prototype.constructor = Square;

//don’t forget to set the constructor property to squareBase

In the console,



We can even call duplicate method which square object obtained from its parent Shape.



Everything is working as it is supposed to.

However these lines,

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

Circle.prototype.construtor = Circle;

And…

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

Square.prototype.constructor = Square;

For setting up the prototype chain are a little bit noisy and as we define multiple objects we might end up making some kind of mess unintentionally.

So let us refactor this code and extract these two lines into a function that we can reuse.

function extend(Child, Parent) {

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

  Child.prototype.construtor = Child;

}

Here we defined a function called *extend* and note that First letter of the parameter of this function is in uppercase (*Parent and Child*) because we expect constructor functions to be its arguments.

With this we have encapsulated the inheritance creation logic inside extend and we can use it anywhere in our code.

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.construtor = Child;

}

function Circle(radius, color) {

  Shape.call(this, color);

  this.radius = radius;

}

extend(Circle, Shape); //call extend with Parent and Child constructor

Circle.prototype.draw = function () {

  console.log("draw");

};

function Square(size) {

  this.size = size;

}

extend(Square, Shape); // extend method called

Our inheritance still works in a flawless manner.

So this *extend* function is called intermediate function inheritance.

**Method Overriding**:

We have simplified the code from last lecture so that we can focus on one concept.

function extend(Child, Parent) {

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

  Child.prototype.construtor = Child;

}

function Shape() {}

Shape.prototype.duplicate = function () {

  console.log("duplicate");

};

function Circle() {}

extend(Circle, Shape);

const c = new Circle(1);

Now sometimes as we work with inheritance, we may face a situation where below implementation we have defined in the parent object,

Shape.prototype.duplicate = function () {

  console.log("duplicate");

};

…may not work or may not be ideal in a child object.

So let us imagine this algorithm for duplicating a shape works for most of the shape objects but may be it should behave differently for circle objects. This is where we use ***method overriding***. So we *override a method that is defined in the base object*.

It is very simple, all we have to do is redefine the duplicate method in Circle object like this.

function extend(Child, Parent) {

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

  Child.prototype.construtor = Child;

}

function Shape() {}

Shape.prototype.duplicate = function () {

  console.log("duplicate");

};

function Circle() {}

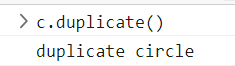
extend(Circle, Shape); 🡪 First extend the Circle

Circle.prototype.duplicate = function () { 🡪Then define the method in Circle

  console.log("duplicate circle");

};

const c = new Circle(1);



Note: It is very important to *put the new implementation after extending the circle*, because we are resetting the prototype. So if we define it before resetting the prototype, then the new implementation is going to disappear.

Circle.prototype.duplicate = function () {

  console.log("duplicate circle"); 🡪 First implement new duplicate method...

};

extend(Circle, Shape); 🡪 Then extend the Circle

🡨 We get old duplicate method.

The reason method overriding works like this in JavaScript goes back to how prototypical inheritance works in JavaScript.

*When we access a property or a method on an object, JavaScript engine walks up the prototype chain and picks the first implementation*.

So even though in this hierarchy, we have implemented the duplicate method both on the parent and child objects. The implementation on the child object will be used.

Now sometimes you may want to call implementation on the parent object as well. This is how we will do this,

function extend(Child, Parent) {

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

  Child.prototype.construtor = Child;

}

function Shape() {}

Shape.prototype.duplicate = function () {

  console.log("duplicate");

};

function Circle() {}

extend(Circle, Shape);

Circle.prototype.duplicate = function () {

  Shape.prototype.duplicate.call(this); 🡪 we use call method and pass *this*

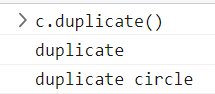
  console.log("duplicate circle");

};

const c = new Circle(1);

Here in Circle.prototype.duplicate, we *call* the duplicate method of its parent or *Shape* object. We use the call method to set the context for *this*.

In the console,



First we get the duplicate message which comes from the implementation of duplicate method in the Shape object and then after that we have ‘duplicate circle’ the new implementation.

**Polymorphism**:

In the last lecture we learned about method overriding which basically means re-implementing a method in a child object.

Now this brings us to a very important and powerful concept in Object oriented programming called Polymorphism.

function extend(Child, Parent) {

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

  Child.prototype.construtor = Child;

}

function Shape() {}

Shape.prototype.duplicate = function () {

  console.log("duplicate");

};

function Circle() {}

extend(Circle, Shape);

Circle.prototype.duplicate = function () {

  console.log("duplicate circle");

};

function Square() {}

extend(Square, Shape);

Square.prototype.duplicate = function () {

  console.log("duplicate Square");

};

In this example, we have a simple hierarchy, on the top we have Shape object and then its two derivatives or child objects, Circle and Square.

Each object provides a different implementation of the duplicate method. So we can say *duplicate method has many forms*.

*How is this so powerful*?

Let us imagine we have an array of shape objects.

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

Now, we can iterate over this array using for – of loop.

for (let shape of shapes) {

  shape.duplicate();

}

*Depending on the type of shape object, a different implementation or a different form of the duplicate method will be called*.



Note: Before OOP, if you wanted to implement this logic, we will have to write code like this…

for (let shape of shapes){ 🡪 a non – OOP way of writing code

  if (shape.type == 'circle'){

    duplicateCircle();

  }

  else if (shape.type == 'square'){ 🡪 multiple if else statements...

    duplicateSquare();

  }

}

These functions duplicateCircle and duplicateSquare are not part of any object, these are just stand alone functions. This is non – object oriented way of writing code.

If we have 10 different types of shape, we would end up with 11 conditional statements in this block. *In contrast when we encapsulate variables and functions into objects and use inheritance, we can execute many forms of a method using a single line of code*.

for (let shape of shapes) {

  shape.duplicate();

}



*In Object oriented programming we encapsulate our variables and functions into objects*.

**When to use Inheritance**:

So we have seen inheritance and polymorphism in action. While inheritance is a great tool for solving the problem of code reuse.

We have to be very careful on how to use it, because it can make your source code complex and fragile.

*Just keep it simple*…

*Start with simple objects and then if you see a number of these objects share similar features then perhaps you can encapsulate those features inside of a generic object and use inheritance*.

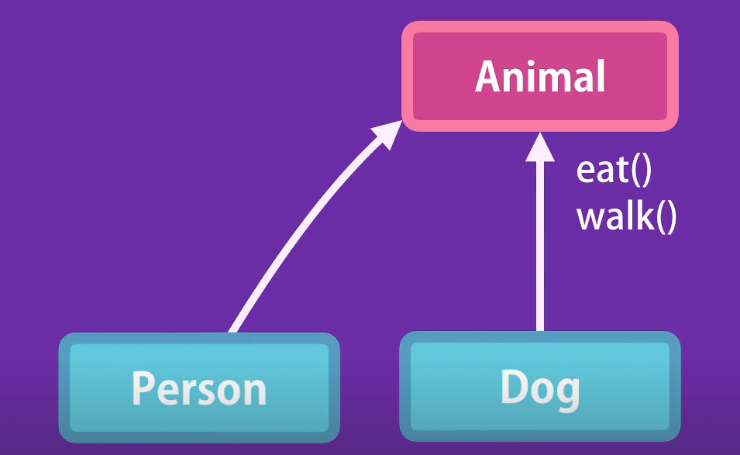
But remember inheritance is not the only solution to enable code reuse.

There is another technique called,

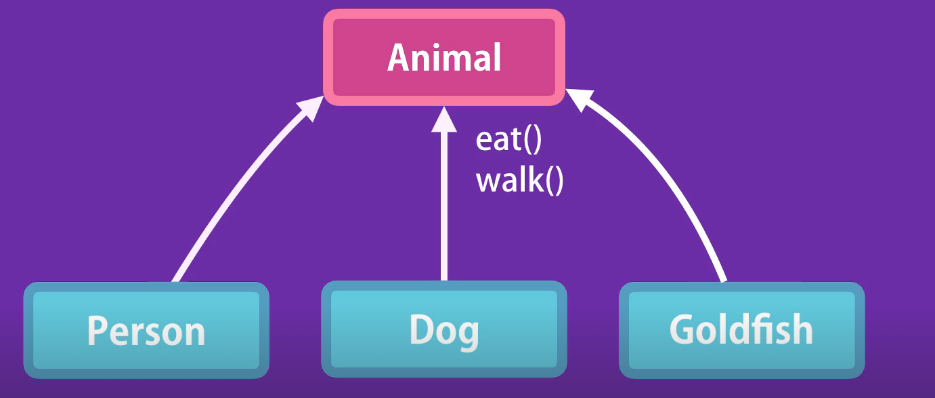


Which we will learn soon.

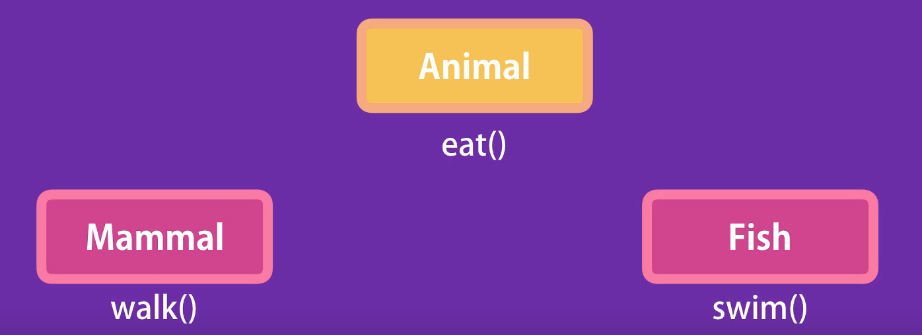
But first let us see the problem with inheritance.



Here we have this Animal object with two methods *eat* and *walk* and two objects that derive from Animal (*Person and Dog*).

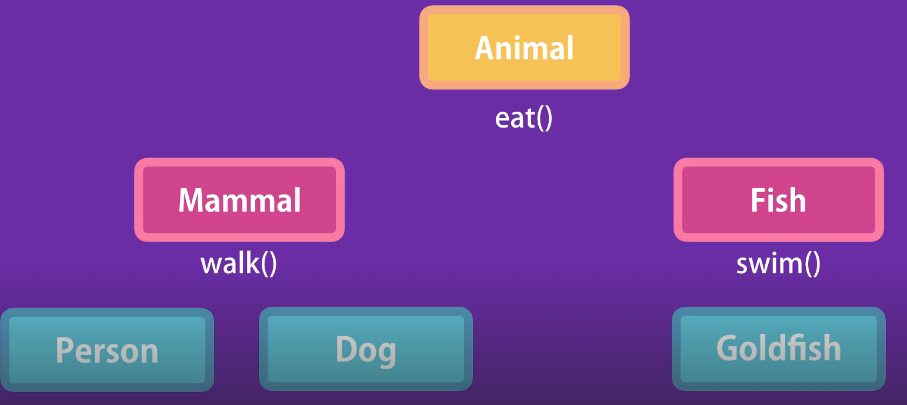


Tomorrow, we are going to introduce a new object called Goldfish that derives from animal. Now our hierarchy is broken because Goldfish cannot walk, they can only swim. This is something that happens quite often amongst inexperienced developers.



So to solve this problem, we need to change our hierarchy. On the top we should have Animal object with the eat method. Under that we will have two objects, Mammal and Fish.

Then we can have Person and Dog derive from Mammal and Goldfish derive from Fish.



See what happened, as a result of introducing a new kind of animal that is Goldfish, we had to change our hierarchy which results in more complications.

Now what would happen, if we had 10 different types of animals, this hierarchy would get more and more complex and we have to constantly go back and forth, to determine the right place to implement a method.

So as a lesson…



If you want to use hierarchies, keep it to one level. Do not go more than one level of inheritance. There is a famous saying that says…

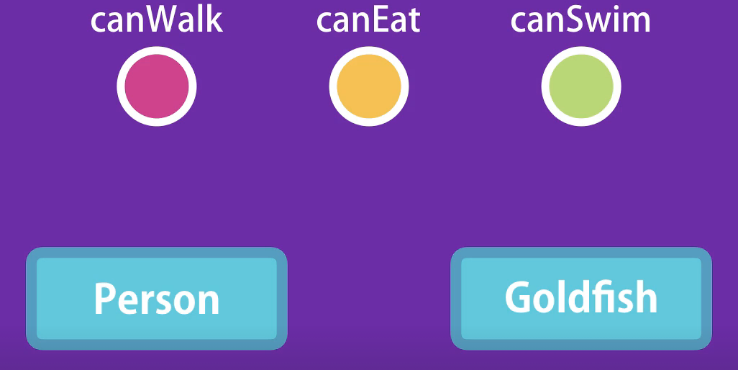


With composition instead of having complex hierarchies like this, we can compose a few objects together to create a new object and this technique gives us great flexibility.



So back to our previous example, instead of using inheritance, we can define various features for our Animals as independent objects.

So we can have three objects called canWalk, canEat and canSwim. Each of these objects are plain JavaScript objects with certain properties and methods.



Now if we want to have a person object, we can simply compose canWalk and canEat to create a person object. Tomorrow if we want to introduce a Goldfish, we can compose canEat and canSwim together to produce a goldfish.

So this way we do not have hierarchy and we can come up with any combination of these objects to create new objects.

In JavaScript we can use *mixins* to *achieve composition*.

**Mixins**:

Let us start by defining a new object *canEat* with the object literal syntax.

const canEat = {

  eat: function () {

    this.hunger--;

    console.log("eating");

  },

};

In this object we add one method called *eat* which can reduce the hunger☺. Here we are defining one feature as an object.

Similarly we can define another feature *canWalk*.

const canWalk = {

  walk: function () {

    console.log("walking");

  },

};

Now, we can compose these objects together to create a person that can eat and walk. So in ES6 we have a new method that is Object.assign. We can *use this method to copy the properties and methods from one object to another*.

So *we pass an empty object as a target and then pass one or more source objects*.

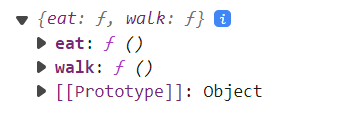
Object.assign({}, canEat) //source can be a single object

Object.assign({}, canEat, canWalk) //or multiple objects as well

If we pass canEat here, this *Object.assign* will copy all the properties and methods that we have defined in canEat into this blank { } object.

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

console.log(person);



Here is our person object with two methods eat and walk.

Now if we are using a constructor function, we can still use this technique.

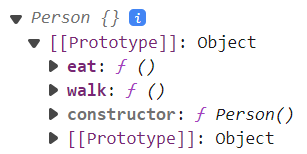
function Person() {} //our Person constructor

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

We passed Person constructor as the first argument of this *Object.assign* method as the target object.

const person = new Person();

console.log(person);



So *we have basically modified the prototype of Person and added the capability to eat and walk*.

Let us say tomorrow we are going to add two new objects in this application, *Goldfish* and *Duck*. They both should have the capability to swim.

So we can define a new feature called *canSwim*.

const canSwim = {

  swim: function () {

    console.log("swimming");

  },

};

Then we can define a new constructor called *Goldfish* and use *Object.assign* to modify the prototype of Goldfish.

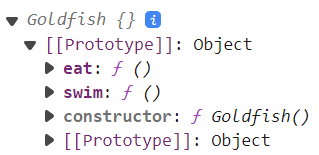
function Goldfish() {}

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

Create a goldfish object,

const goldfish = new Goldfish();

console.log(goldfish);



To make this code a bit more readable, we can extract the Object.assign logic into a function called *mixin*.

function mixin(target, ...sources) { ... is *Rest* operator in parameter

  Object.assign(target.prototype, ...sources); ... is *Spread* operator in body

}

Complete code,

function mixin(target, ...sources) { //custom mixin function

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

}

const canEat = {

  eat: function () {

    this.hunger--;

    console.log("eating");

  },

};

const canWalk = {

  walk: function () {

    console.log("walking");

  },

};

const canSwim = {

  swim: function () {

    console.log("swimming");

  },

};

function Person() {}

function Goldfish() {}

const goldfish = new Goldfish();

const person = new Person();

mixin(Person, canEat, canWalk); //use mixin as helper function to shorten code

console.log(person);

mixin(Goldfish, canEat, canSwim);

console.log(goldfish);

