**Classes and Objects**

**Introduction**

Up until now, we’ve been building one off objects. However, to build more structured applications, we need to learn how to build classes (categories of objects).

In this lesson we will create new objects with constructor functions. We will also leverage the *this* keyword to access objects. Finally we will learn how to base objects on other objects with prototypal inheritance and subclassing.

**Properties and Methods**

We can use objects to represent real-life things (i.e.: someone or something) in code.

Object-Oriented JavaScript is a way to write code that will automatically classify objects.

**Object-Oriented Programming**

Objects in JavaScript can represent real-life things. Objects have properties that represent attributes (adjectives), and methods, that represent actions (verbs).

**Constructor Functions**

We don’t need to create an object using object literal notation. And, we can also write a function that returns an object.

However, one of the main ways that objects are created is with a **constructor function**.

To instantiate (create) a new object we use the *new* operator to invoke the function:

**new** SoftwareDeveloper();

1. The new keyword is used.
2. The name of the constructor function starts with a capitalized letter, distinguishing it from a regular function.

**Structure and Syntax**

This is what the internal of a constructor function looks like:

**function** **SoftwareDeveloper**() {

**this**.favoriteLanguage = 'JavaScript';  
}

1. Rather than declaring local variables, constructor functions persist data with the *this* keyword.
2. This function doesn’t seem to return anything. Constructor functions in JavaScript should not have an explicit return value (no return or console.log statement).

**Creating a New Object**

**let** developer = **new** SoftwareDeveloper();

The *new* operator creates a new object with the same property from SoftwareDeveloper constructor function.

We can use the same constructor function to create multiple objects.

**let** engineer = **new** SoftwareDeveloper();  
**let** programmer = **new** SoftwareDeveloper();

**Constructor Functions Can Have Parameters**

Just like regular functions, constructor functions also accept arguments:

**function** **SoftwareDeveloper**(name) {

**this**.favoriteLanguage = 'JavaScript';

**this**.name = name;  
}

**let** instructor = **new** SoftwareDeveloper('Andrew');

console.log(instructor);  
*// SoftwareDeveloper { favoriteLanguage: 'JavaScript', name: 'Andrew' }*

**Omitting the *new* Operator**

Without using the *new* operator, no object is created. The function is invoked just like any other regular function.

The variable *coder* ends up being assigned to undefined:

***function******SoftwareDeveloper****(name) {*

***this****.favoriteLanguage = 'JavaScript';*

***this****.name = name;*

*}*

***let*** *coder = SoftwareDeveloper('David');*

*console.log(coder);  
// undefined*

**Object’s Constructor (instanceof)**

We can use the *instanceof* to identify if an object was created with a constructor function:

function Developer(name) {

this.name = name;

}

let dev = new Developer('Veronika');

typeof dev

// "object"

dev instanceof Developer;

// true

**The *‘this’* Keyword**

***this* is Constructor Functions**

A common misconception is that *this* refers to the object where it is defined. But, it is not the case.

The value of *this* is actually not assigned to anything until an object calls the method where *this* is used. In other words, the value assigned to *this* is based on the object that invokes the method where *this* is defined:

**const** dog = {

bark: **function** () {

console.log('Woof!');

},

barkTwice: **function** () {

**this**.bark();

**this**.bark();

}  
};

dog.bark();

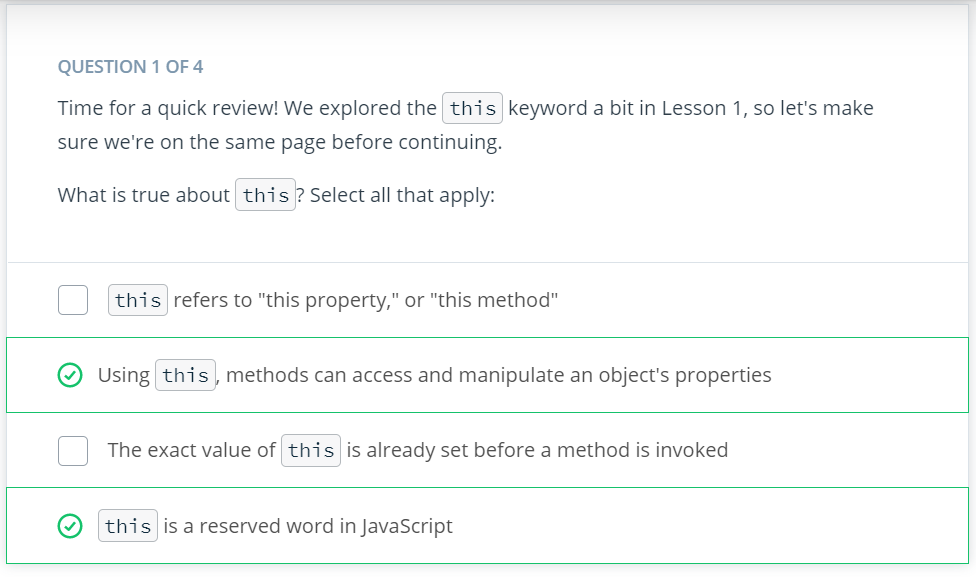
*// Woof!*

dog.barkTwice();

*// Woof!*  
*// Woof!*

To tie things all together:

*this.bark()* tells *barkTwice()* to look at the dog, which is the object that the method was called on to find bark(). Otherwise it can go further up the scope chain.



**What Does *this* Get Set To?**

At this point, we have seen *this* in many different contexts: within a method or referenced by a constructor function.

There are four ways to call functions, and each way sets *this* differently:

1. Constructor function: the *new* keyword sets *this* to a newly-created object.
2. Calling a function that belongs to an object: sets *this* to the object itself.
3. Calling a function on its own (simply invoking a regular function): sets *this* to *window*, which is the global object if the host environment is the browser:

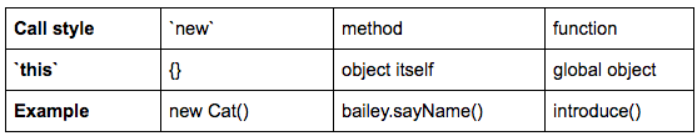
**function** **funFunction**() {

**return** **this**;

}

funFunction();  
*// (returns the global object, `window`)*

1. The fourth way to call functions allows us to set *this* ourselves. Will take a deep dive soon.



**More Ways to Invoke Functions**

We’ve seen various ways to invoke functions, each with their own implications regarding the value of *this.* There are yet two more ways to invoke a function: either using *call()* or the *apply()* method.

**call()**

Using *call()* to invoke a method allows us to “borrow” a method from one object -- then use it for another object:

**const** mockingbird = {

title: 'To Kill a Mockingbird',

describe: **function** () {

console.log(`${this.title} is a classic novel`);

}  
};

mockingbird.describe();  
*// 'To Kill a Mockingbird is a classic novel'*

**const** pride = {

title: 'Pride and Prejudice'

};

mockingbird.describe.call(pride);

*// 'Pride and Prejudice is a classic novel'*

*call()* is very effective when invoking a function in the scope of the first argument passed into it. Likewise, we can leverage the *apply()* method to do the same, albeit with differences in how arguments are passed into.

**apply()**

Just like *call()*, the *apply()* method is called on a function to not only invoke that function, but also to associate with it a specific value of *this.* However, rather than passing arguments one-by-one, separated by commas, *apply()* takes the function’s arguments in an array:

**function** **multiply**(n1, n2) {

**return** n1 \* n2;

}  
multiply.call(window, 3, 4);  
*// 12*

multiply.apply(window, [3, 4]);  
*// 12*

Using *apply()* we collect all of the *multiply()* function’s arguments in an array. Then, we pass that entire array into *apply()*.

**const** mockingbird = {

title: 'To Kill a Mockingbird',

describe: **function** () {

console.log(`${this.title} is a classic novel`);

}  
};

mockingbird.describe();  
*// 'To Kill a Mockingbird is a classic novel'*

**const** pride = {

title: 'Pride and Prejudice'

};

mockingbird.describe.call(pride);

*// 'Pride and Prejudice is a classic novel'*

mockingbird.describe.apply(pride);

*// 'Pride and Prejudice is a classic novel'*

**Choosing One Method Over Other**

*call()* may be limited if we don’t know ahead of time the number of arguments that the function needs. In this case, apply() would be a better option, since it simply takes an array of arguments, then unpacks them to pass along to the function.



**Callback and *this***

The value of *this* has some potential scope issues when callback functions are involved, and things can get a bit tricky.

Since it is such a common pattern, JavaScript provides an alternate and less verbose approach: the bind() method.

Similar to *call()* and *apply()*, the *bind()* method allows us to define a value for *this*. *bind()* is a method that is also called \_on\_ a function, but unlike *call()* or *apply()*, which both invoke the function right away -- *bind()* returns a new function that, when called, has *this* set to the value we give it.

function invokeTwice(cb) {

cb();

cb();

}

const dog = {

age: 5,

growOneYear: function () {

this.age += 1;

}

};

const myGrow = dog.growOneYear.bind(dog);

invokeTwice(myGrow)

dog.age;

// 7

**Prototypal Inheritance**

In JavaScript, inheritance is when an object is based on another object. This is called inheriting or extending.

**Adding Methods to the Prototype**

Earlier, we simply added methods to the constructor function itself:

function Cat(name) {

this.lives = 9;

this.name = name;

this.sayName = function () {

console.log(`Meow! My name is ${this.name}`);

};

}

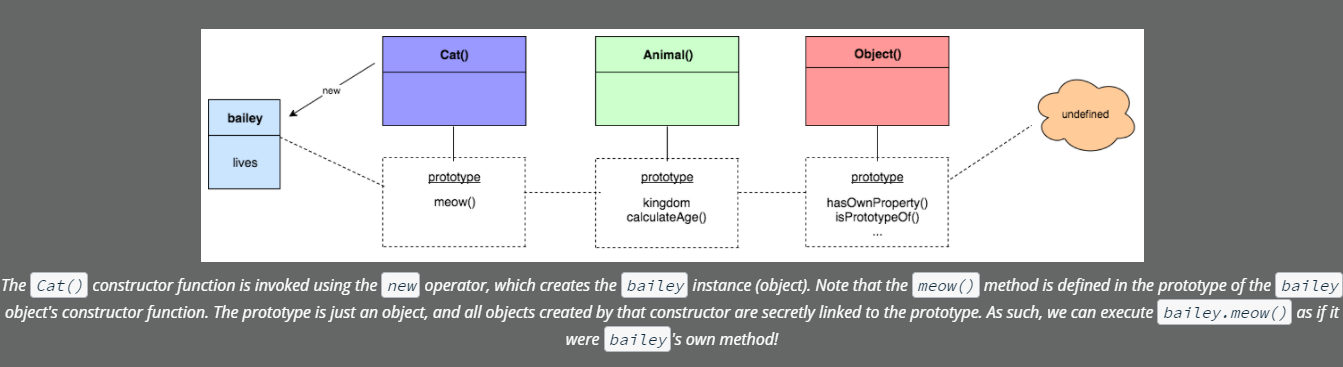
This way, a *sayName* method gets added to all *Cat* objects by saving a function to the *sayName* attribute of newly-created Cat objects.

* This works fine, but what if we want to instantiate more and more *Cat* objects with this constructor?
  + Will we create a new function every single time for that Cat object’s *sayName*?
* What if we want to make changes to the method?
  + Will we update all objects individually?

In this situation, it makes sense to have all objects created by the same *Cat* constructor function just share a single *sayName* method.

**Prototype Property:** to save memory and keep things DRY, we can add methods to the constructor function’s prototype property. The prototype is just an object, and all objects created by a constructor function keep a reference to the prototype. Those objects can even use the prototype’s properties as their own!

JavaScript leverages this secret link -- between an object and its prototype -- to implement inheritance. Consider the following prototype chain.



Recall that each function has a prototype property, which is really just an object. When this function is invoked as a constructor using the new operator, it creates and returns a new object. This object is secretly linked to its constructor's prototype, and this secret link allows the object to access the prototype's properties and methods as if it were its own!

**Finding Properties and Methods on the Prototype Chain**

1. First, the JavaScript engine will look at the object's own properties. This means that any properties and methods defined directly in the object itself will take precedence over any properties and methods elsewhere if their names are the same (similar to variable shadowing in the scope chain).
2. If it doesn't find the property in question, it will then search the object's constructor's prototype for a match.
3. If the property doesn't exist in the prototype, the JavaScript engine will continue looking up the chain.
4. At the very end of the chain is the Object() object, or the top-level parent. If the property still cannot be found, the property is undefined.

function Dog(age, weight, name) {

this.age = age;

this.weight = weight;

this.name = name;

}

Dog.prototype.bark = function () {

console.log(`${this.name} says woof!`);

};

dog1 = new Dog(2, 60, 'Java');

dog2 = new Dog(4, 55, 'Jodi');

dog1.bark();

dog2.bark();

**Prototype Quiz:**

**// (A)**

**function Dalmatian (name) {**

**this.name = name;**

**this.bark = function() {**

**console.log(`${this.name} barks!`);**

**};**

**}**

**// (B)**

**function Dalmatian (name) {**

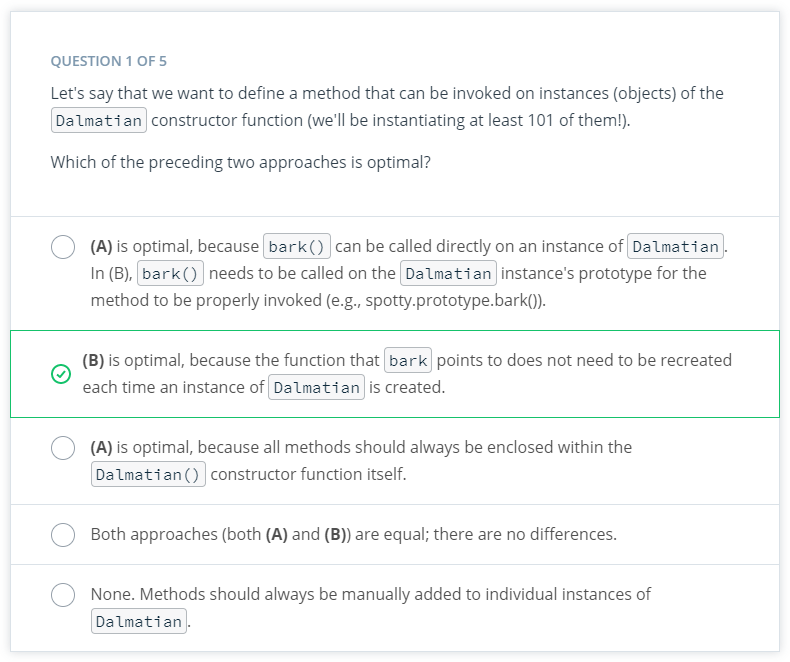
**this.name = name;**

**}**

**Dalmatian.prototype.bark = function() {**

**console.log(`${this.name} barks!`);**

**};**

****

**Replacing the prototype Object**

//\* replacing the prototype object \*//

function Hamster() {

this.hasFur = true;

}

let waffle = new Hamster();

let pancake = new Hamster();

Hamster.prototype.eat = function () {

console.log('Chomp chomp chomp!');

};

waffle.eat();

// 'Chomp chomp chomp!'

pancake.eat();

// 'Chomp chomp chomp!'

Hamster.prototype = {

isHungry: false,

color: 'brown'

};

console.log(waffle.color);

// undefined

waffle.eat();

// 'Chomp chomp chomp!'

console.log(pancake.isHungry);

// undefined

const muffin = new Hamster();

muffin.eat();

// TypeError: muffin.eat is not a function

console.log(muffin.isHungry);

// false

console.log(muffin.color);

// 'brown

1. Even after we make the new objects, *waffle* and *pancake*, *w*e can still add properties to Hamster’s prototype and it will still be able to access those new properties.
2. When replacing Hamster’s prototype object with something else entirely, the previous objects don’t have access to the updated prototype’s properties, they just retain their secret link to the old prototype.
3. Any new Hamster objects created moving forward will use the updated prototype.

**Checking an Object’s Properties**

As we’ve seen, if an object doesn’t have a particular property of its own, it can access one somewhere along the prototype chain (assuming it exists, of course). With so many options, it can sometimes get tricky to tell just where a particular property is coming from!

Below are a few useful methods to help us along the way:

**hasOwnProperty():** allows us to find the origin of a particular property. Upon passing in a string of the property name we’re looking for, the method will return a boolean indicating whether or not the property belongs to the object itself (i.e., that property was not inherited). Consider the *Phone* constructor with a single property defined directly in the function, and another property on its *prototype* object:

function Phone() {

this.operatingSystem = 'Android';

}

Phone.prototype.screenSize = 6;

const myPhone = new Phone();

const own = myPhone.hasOwnProperty('operatingSystem');

console.log(own);

// true

const inherited = myPhone.hasOwnProperty('screenSize');

console.log(inherited);

// false

**isPrototypeOf():** checks whether or not an object exists in another object’s prototype chain. Using this method, we can confirm if a particular object serves as the prototype of another object.

const rodent = {

favoriteFood: 'cheese',

hasTail: true

};

function Mouse() {

this.favoriteFood = 'cheese';

}

Mouse.prototype = rodent;

const ralph = new Mouse();

const result = rodent.isPrototypeOf(ralph);

console.log(result);

// true

isPrototypeOf() is a great way to confirm if an object exists in another object’s prototype chain.

**Object.getPrototypeOf():** retrieves the prototype of a given object:

//\* getPrototypeOf \*//

const myPrototype = Object.getPrototypeOf(ralph);

console.log(myPrototype);

// { favoriteFood: 'cheese', hasTail: true

**The *constructor* Property**

Each time an object is created, a special property is assigned to it under the hood: constructor.

function Longboard() {

this.material = 'bamboo';

}

const board = new Longboard();

console.log(board.constructor);

// function Longboard() {

// this.material = 'bamboo';

// }

Object.constructor shows the original constructor function itself.

**Prototypal Inheritance: Subclasses**

**Subclasses**

By establishing inheritance, we can subclass, that is, have a “child” object take on most or all of a “parent” object’s properties while retaining unique properties of its own.

Parent Object: Animal

Properties: age, weight

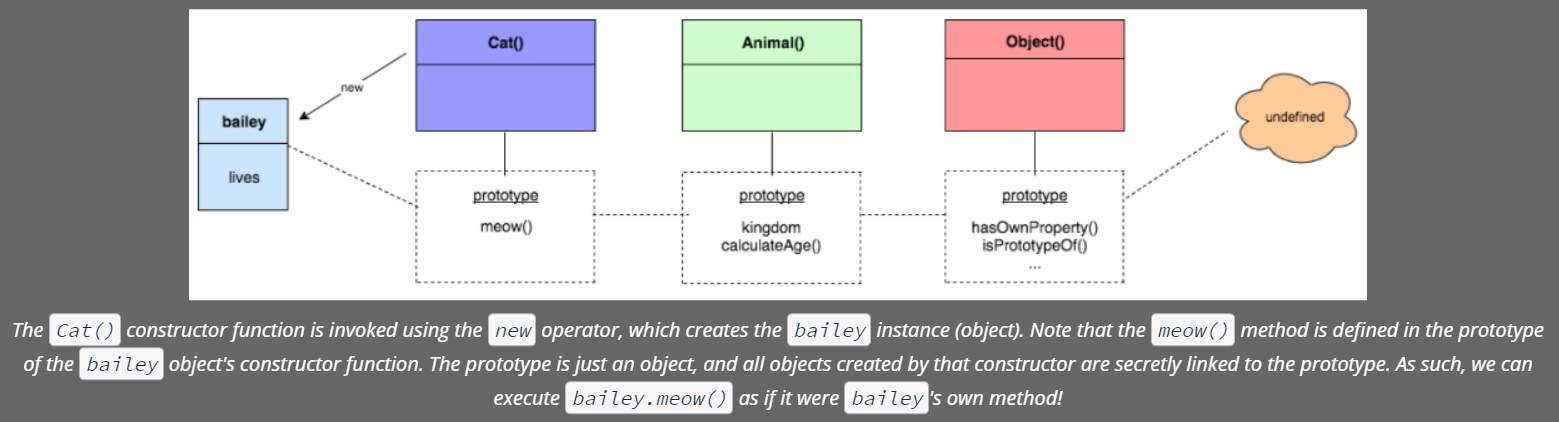
Methods: eat, sleep

Child Object: Cat

Properties: Animal.properties + lives

Methods: Animal.methods + meow()

**Inheritance via Prototypes**



When calling any property on any object, the JavaScript engine will first look for the property in the object itself (non-inherited properties). If the property is not found, JavaScript will then look at the object’s prototype. If the property still isn't found in the object’s prototype, JavaScript will continue the search up the prototype chain.

**The Secret Link**

const bear = {

claws: true,

diet: 'carnivore'

};

function PolarBear() {

// ...

}

PolarBear.prototype = bear;

const snowball = new PolarBear();

console.log(Object.getPrototypeOf(snowball));

//{ claws: true, diet: 'carnivore' }

snowball.color = 'white';

snowball.favoriteDrink = 'cola';

console.log(snowball);

//{ color: 'white', favoriteDrink: 'cola' }

*snowball* has just two properties of its own: *color* and *favoriteDrink*. However, *snowball* has access to properties that don’t exist inside it: *claws* and *diet*.

Since *claws* and *diet* both exist as properties in the prototype object, they are looked up because objects are secretly linked to their constructor’s prototype property.

**\_\_proto\_\_ The Secret Link**

\_\_proto\_\_ is a property of all objects (instances) made by a constructor function, and points directly to that constructor’s prototype object.

console.log(snowball.\_\_proto\_\_);

// { claws: true, diet: 'carnivore' }

Since the \_\_proto\_\_ property refers to the same object as *PolarBear’s* prototype, *bear*, comparing them returns true:

console.log(snowball.\_\_proto\_\_ === bear);

// true

console.log(snowball.\_\_proto\_\_ === PolarBear.prototype);

// true

It’s great to know the secret link for learning how functions and objects are interconnected, but we should not use \_\_proto\_\_ to manage inheritance. If we just need to review an object’s prototype, we can use the ***Object.getPrototypeOf()***

**Object.create()**

Using *Object.create()*, let us manage inheritance without altering the prototype!

*Object.create()* takes a single object as an argument, and returns a new object with its *\_\_proto\_\_* property set to what argument is passed into it. From that point, we simply set the returned object to be the prototype of the child object’s constructor function:

const mammal = {

vertebrate: true,

earBones: 3

};

const rabbit = Object.create(mammal);

console.log(Object.getPrototypeOf(rabbit));

//{ vertebrate: true, earBones: 3 }

console.log(rabbit.\_\_proto\_\_ === mammal);

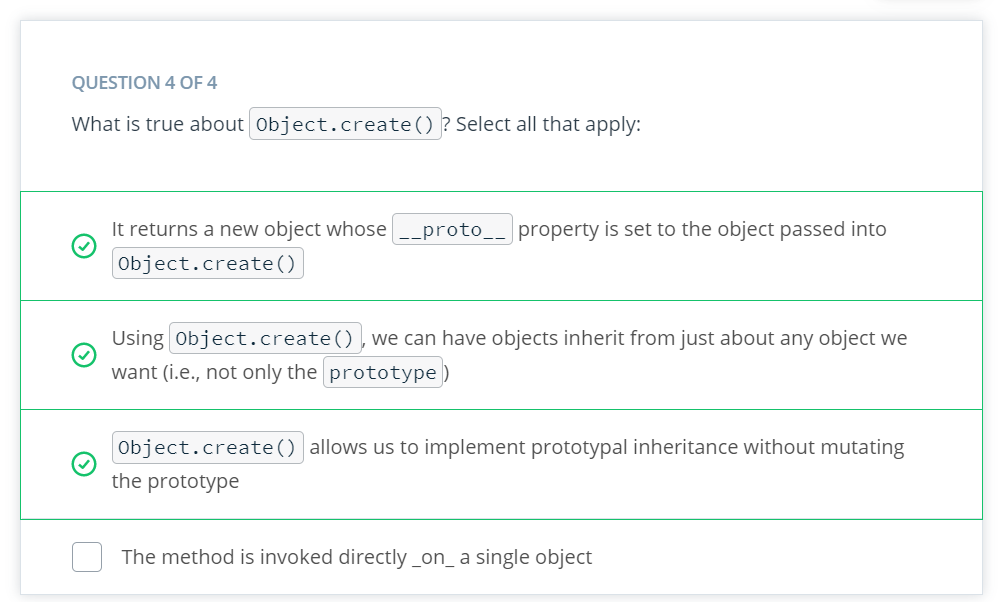
// true

Object.create() takes in a single object as an argument, and returns a new object. That new object’s \_\_proto\_\_ property is set to whatever was originally passed into Object.create().

*rabbit* has no properties of its own. However, the *rabbit* is secretly linked to the *mammal*. That is, its \_\_proto\_\_ points to mammal.

*rabbit* extends (inherits) from *mammal*, and can access *mammal*’s properties as if it were its own.

***Object.create()*** gives us a clean method of establishing prototypal inheritance in JavaScript. We can easily extend the prototype chain this way, and we can have objects inherit from just about any object we want!



**Summary**

Inheritance in JavaScript is all about setting up the prototype chain. This allows us to **subclass**, that is, create a "child" object that inherits most or all of a "parent" object's properties and methods. We can then implement any of the child object's unique properties and methods separately, while still retaining data and functionality from its parent.

An object (instance) is secretly linked to its constructor function's prototype object through that instance's \_\_proto\_\_ property. **We should never use the \_\_proto\_\_ property in any code you write.** Using \_\_proto\_\_ in any code, or even inheriting just the prototype directly, leads to some unwanted side effects.

To efficiently manage inheritance in JavaScript, an effective approach is to avoid mutating the prototype completely. Object.create() allows us to do just that, taking in a parent object and returning a *new* object with its \_\_proto\_\_ property set to that parent object.