JavaScript

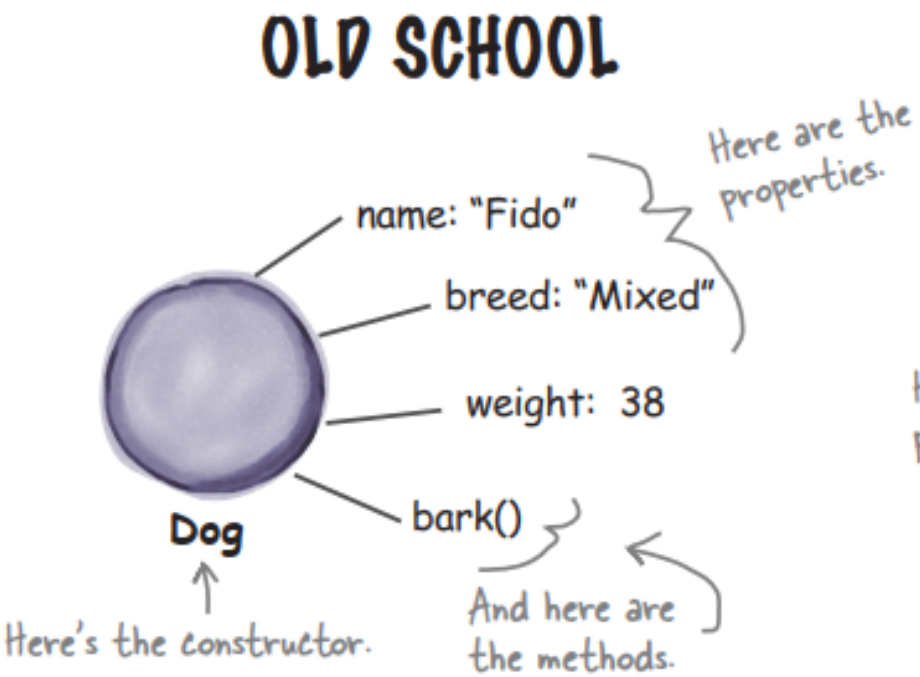
CHAPTER II. OBJECT

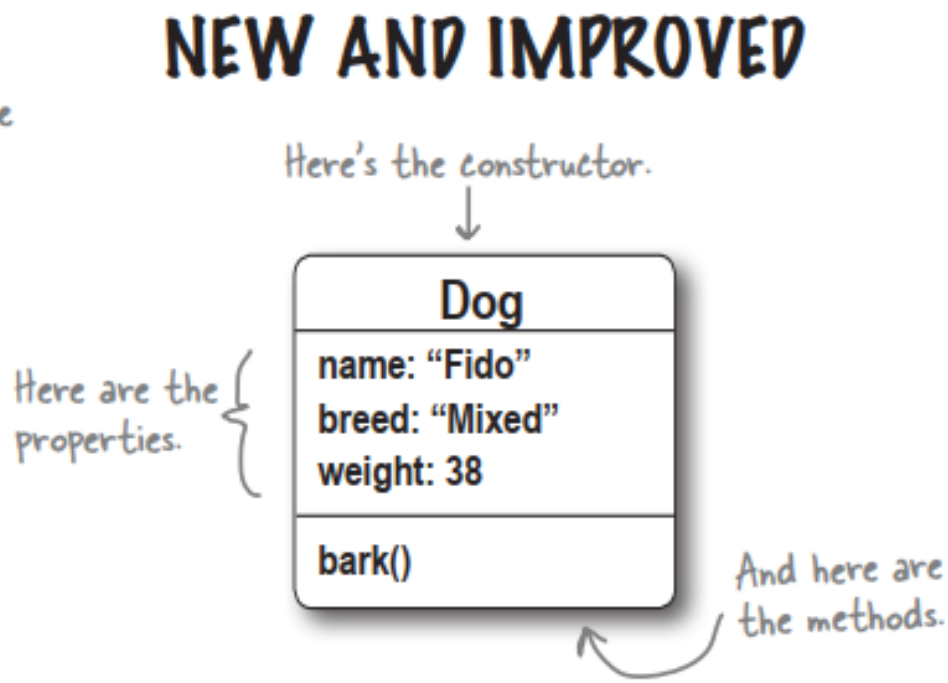
Ths. Hồ Đức Lĩnh
Mail: duclinh47th@gmail.com
Phone: 0905 28 68 87

CONTENT

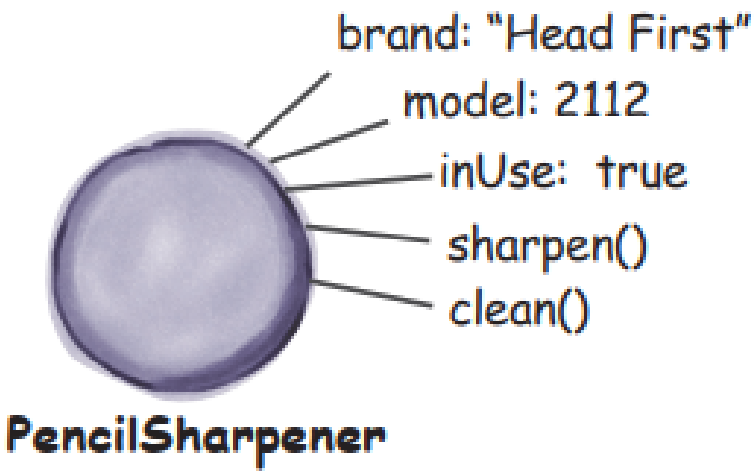
- In this chapter, we'll cover the following topics :
  - Object literals
  - Adding properties to objects
  - Object methods
  - JSON
  - The Math object
  - The Date object
  - The RegExp object
  - Project — we'll create quiz and question objects and ask random questions

Object: Old and New

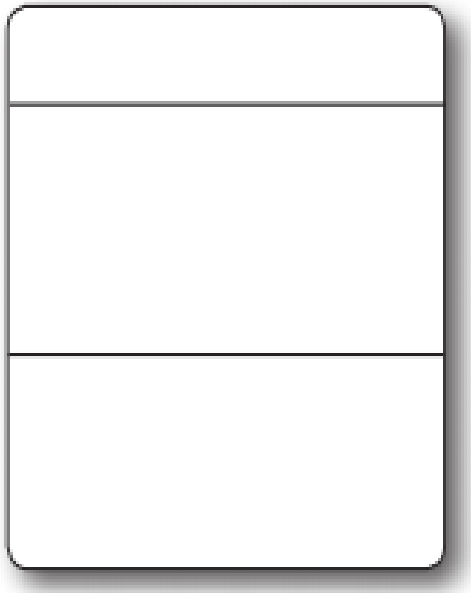




Object: Old and New



Fill in this object diagram.



CREATE AN OBJECT

- In JavaScript almost everything is an `Object`
- Multiple ways to create an Object
  - Object Constructor `var obj = new Object()`
  - Object Literal `var obj = {}`
  - Inbuilt Method `var obj = Object.create()`
  - Constructor function `var obj = new Person()`

Object Literals

- An object in JavaScript is a self-contained set of related values and functions.
- They act as a **collection** of **named properties** that map to any JavaScript **value such as strings, numbers, booleans, arrays** and functions.
- If a property's value is a function, it is known as a **method**

Object Literals - Example

Object	Properties	Methods
	car.name = Fiat	car.start()
	car.model = 500	car.drive()
	car.weight = 850kg	car.brake()
	car.color = white	car.stop()

- In real life, a car is an **object**.
- A car has **properties** like weight and color, and **methods** like start and stop

Creating new objects
- Using object initializers

- The syntax for an object using an object initializer is:

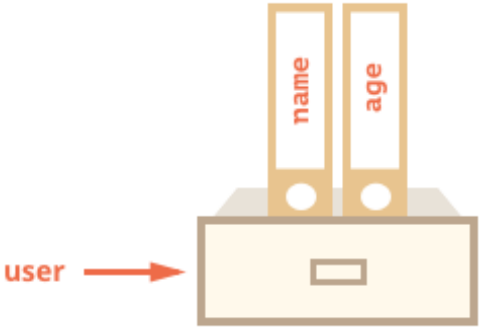
```
var obj = { property_1:   value_1,   // property_# may be an identifier...
            2:            value_2,   // or a number...
            // ...,
            'property n': value_n }; // or a string
```

- Accessing Object Methods

```
objectName.methodName()
```

Creating new objects - Example

```
var user = {          // an object
    name: "John",     // by key "name" store value "John"
    age: 30           // by key "age" store value 30
};
```



- In the user object, there are **two properties**:
  - The first property has the name "**name**" and the value "John".
  - The second one has the name "**age**" and the value 30.
- The resulting user object can be imagined as a cabinet with two signed files labeled “name” and “age”.

Example

```
<p id="demo"></p>

<script>
var Car = {
        //Property
        name: 'Honda',
        model: 'ABC',
        weight: '100',
        color: 'Green',
        //Method
        info: function(){
            return this.name + ' - ' + this.model;
    }
};
document.getElementById("demo").innerHTML = Car.info();
</script>
```

Using a constructor function

- Alternatively, you can create an object with these two steps:
  - Define the object type by writing a constructor function. There is a strong convention, with good reason, to use a capital initial letter.
  - Create an instance of the object with new.

Example

```
<p id="demo"></p>

<script>
function Car(_name, _model, _weight, _color){
    this.name = _name;
    this.model = _model;
    this.weight = _weight;
    this.color =_color;
    this.info = function(){
        return this.name + ' - ' + this.model;
    }
}

var xe1 = new Car("Vinfast","Sudan",1000,"Black");
document.getElementById("demo").innerHTML = xe1.info();
</script>
```

Example: the Dog constructor

0

Dog

name: "Fido"
breed: "Mixed"
weight: 38

bark()

2

```
var fido = new Dog("Fido", "Mixed", 38);
```

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.bark = function() {
        if (this.weight > 25) {
            alert(this.name + " says Woof!");
        } else {
            alert(this.name + " says Yip!");
        }
    };
}
```

1

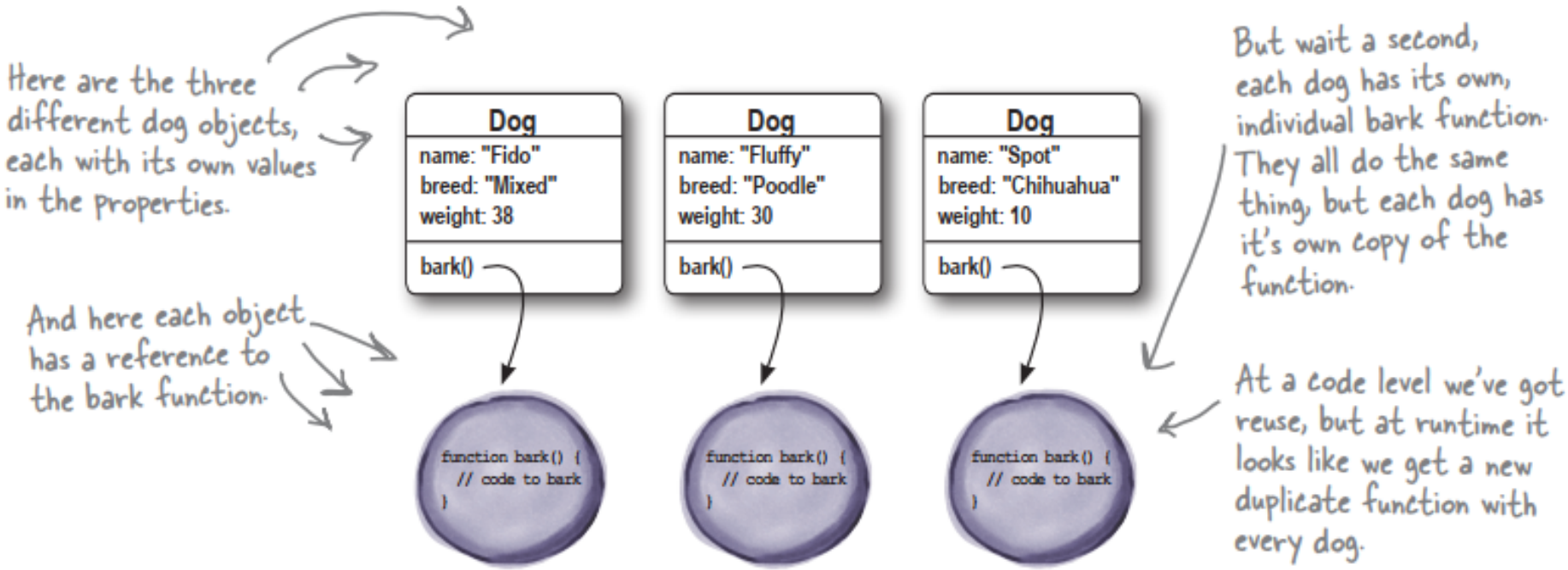
Every dog can have its own custom values and a consistent set of properties.

And every dog comes complete with a bark method.

Even better, we're totally reusing code across all the dogs.

Example: the Dog constructor

```
var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
```



Using the **Object.create** method

```
<p id="demo"></p>
<p id="demo2"></p>
<script>
var Car = {
        //Property
        name: 'Honda',
        model: 'ABC',
        weight: '100',
        color: 'Green',
        //Method
        info: function(){
            return this.name + ' - ' + this.model;
    }
};
//Create new Car
var xe1 = Object.create(Car);
document.getElementById("demo").innerHTML = xe1.info();
//Create new Car
var vinfast = Object.create(Car);
vinfast.name = 'VinFast';
vinfast.model = "Sundan";
vinfast.weight = 1000;
vinfast.color = "Black";
document.getElementById("demo2").innerHTML = vinfast.info();
// document.write(Car.color);
</script>
```

Bài tập tổng hợp

1. Tạo đối tượng gồm các thuộc tính và phương thức như hình:

SINH VIÊN
Họ đệm Tên Giới tính Ngày Sinh
Hiển thị Họ và tên Hiển thị thông tin Sinh viên

NHÂN VIÊN
Họ đệm Tên Giới tính Ngày Sinh Chức vụ Thâm niên công tác Số ngày làm việc Trình độ học vấn Lương cơ sở
Hiển thị Họ và tên Tính lương cơ bản Tính thu nhập của nhân viên Hiển thị thông tin Nhân viên

Bài tập tổng hợp (tt …)

- Lương cơ bản được xác định dựa trên trình độ học vấn như sau
  - Trình độ học vấn là Trung cấp – Hệ số lương là 1.86
  - Trình độ học vấn là Cao đẳng – Hệ số lương là 2.10
  - Trình độ học vấn là Đại học – Hệ số lương là 2.34

  →Lương cơ bản = Mức lương cơ sở * hệ số lương
- **Thu nhập theo ngày làm việc**: Số ngày làm việc cơ sở là 24 ngày/tháng. Vắng làm việc mỗi ngày trừ 200,000;
- **Thu nhập theo chức vụ = Lương cơ sở * Hệ số:** Nhân viên – Hệ số là 1.0; Phó phòng – Hệ số là 2.0; Trưởng phòng – Hệ số là 3.0;
- **Thu nhập theo thâm niên công tác:** < 3 năm hệ số là 1.0, Sau mỗi 3 năm thì hệ số thâm niên tăng 3% lương cơ bản
- **Tổng thu nhập = Lương cơ bản + Thu nhập theo ngày + Thu nhập thâm niên + Thu nhập theo chức vụ**

Prototypes

- Prototypes are the mechanism by which JavaScript objects inherit features from one another;
- All JavaScript **objects inherit properties and methods** from **a prototype**.
- The whole point of this scheme is **to inherit and reuse existing properties (including methods), while extending those properties** in your brand new object

a prototype for a dog object.

I'm the dog prototype. I've got properties that every dog needs and you can use me as a prototype for any dog you want to create.

Here's a prototype for dogs. This is an object that contains properties and methods that all dogs might need.

The prototype doesn't include name, breed or weight because those will be unique to each dog, and supplied by the real dogs that inherit from the prototype.

Dog Prototype

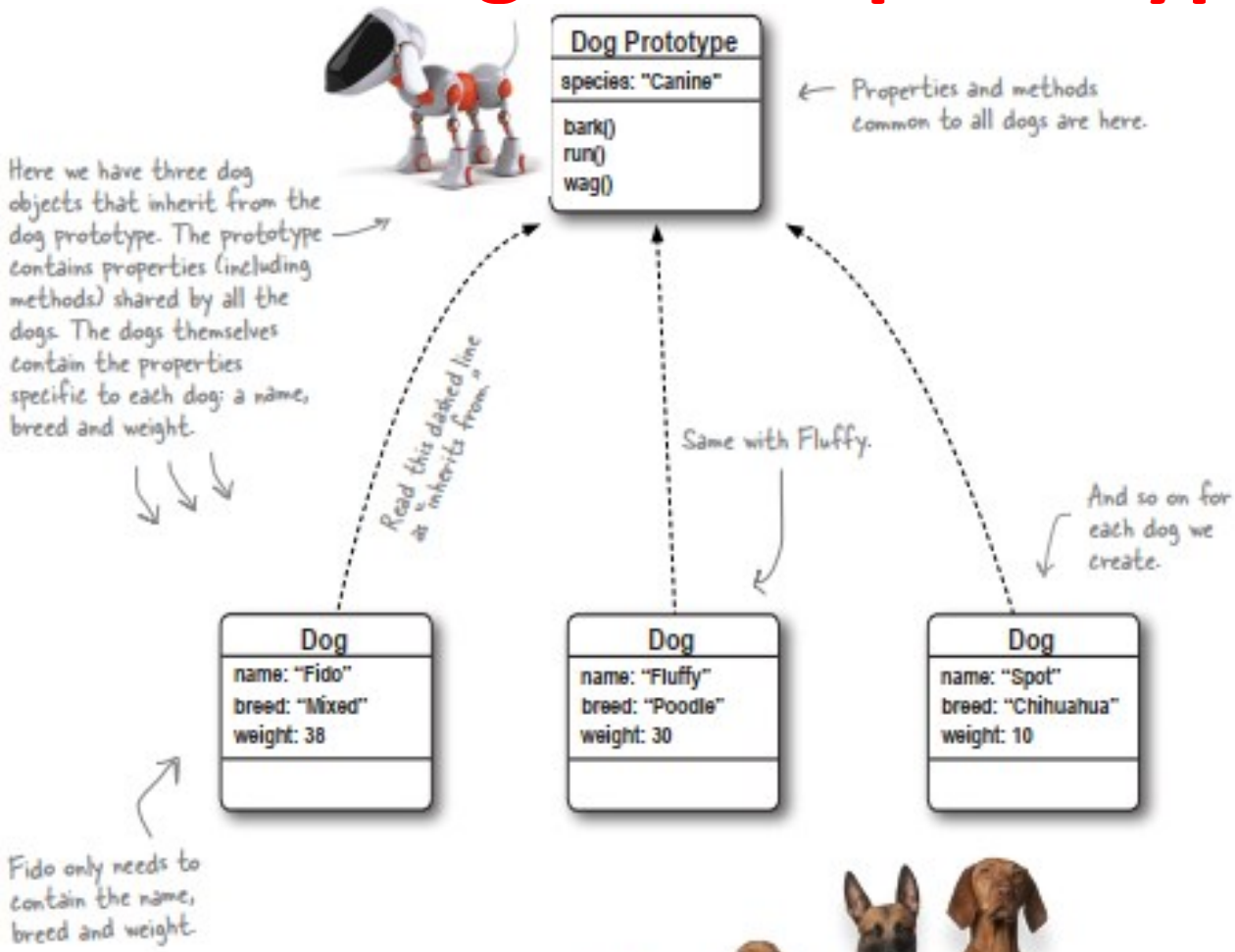
species: "Canine"

bark()
run()
wag()

Contains properties useful to every dog.

Contains behavior we'd like to use in all dogs that we create.

Inheriting from a prototype



How inheritance works?

Start here and follow the numbers: 1, 2, 3, 4, 5

Dog Prototype

species: "Canine"

bark()
run()
wag()

Dog

name: "Fido"
breed: "Mixed"
weight: 38

1 First we need some code. How about we call the bark method of a dog object, like this:

```
fido.bark();
```

Just an everyday call to the fido object's bark method.

2 To evaluate this code we look in the fido instance for a bark method. But there isn't one.

No bark method here! So look UP at the prototype...

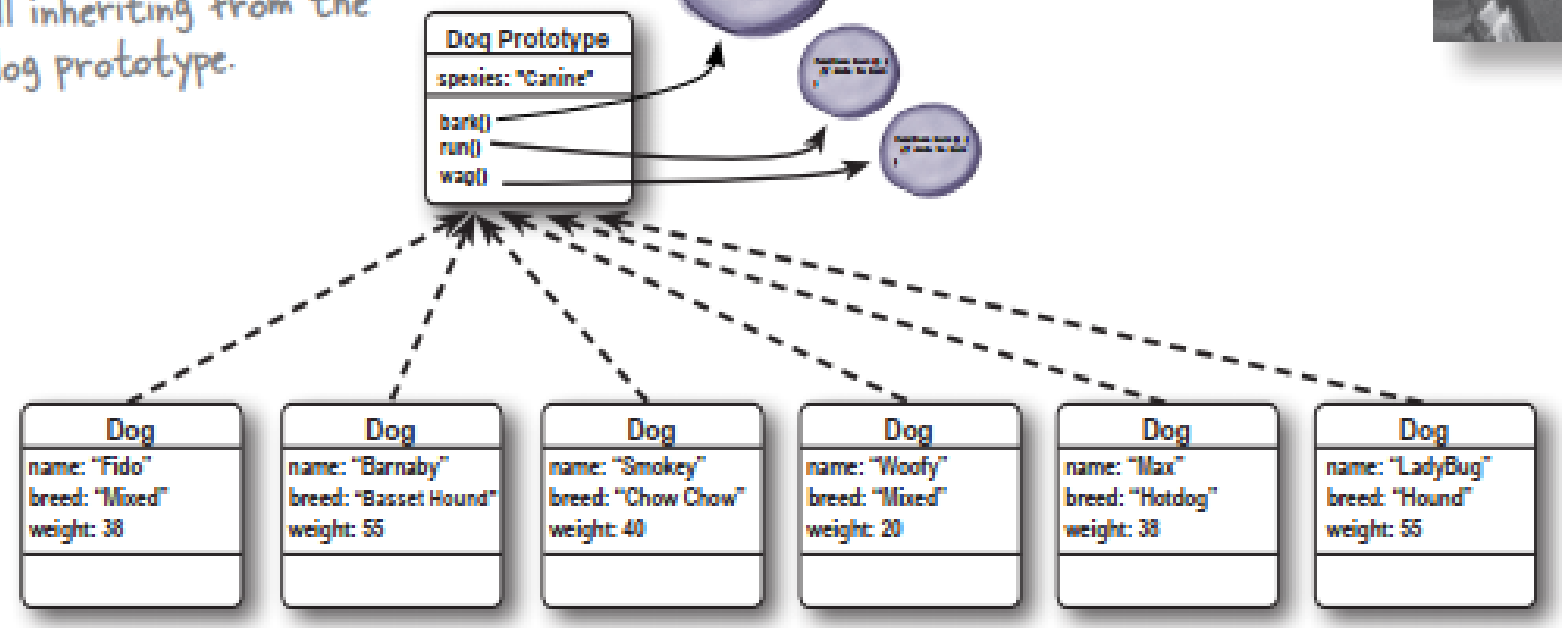
3 If we can't find bark in the fido instance, then we take a look next at its prototype.

4 Checking the Dog prototype we see it does have a bark method.

5 Finally, once we find the bark method, we invoke it, which results in Fido barking.

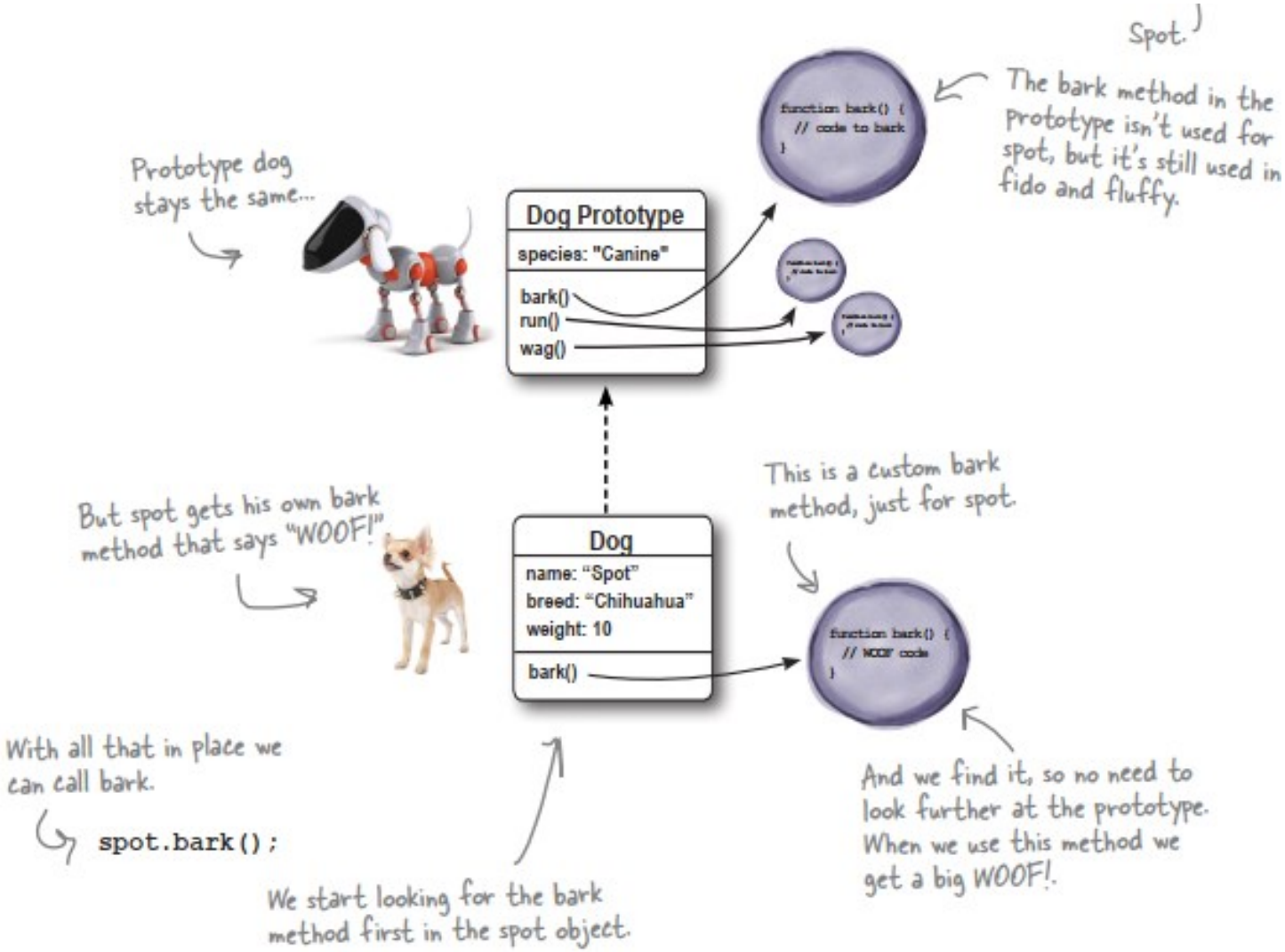
He's started producing lots of dogs already. Here's a bunch of instances he's created, all inheriting from the dog prototype.

Now we only have a single bark function. Much better.



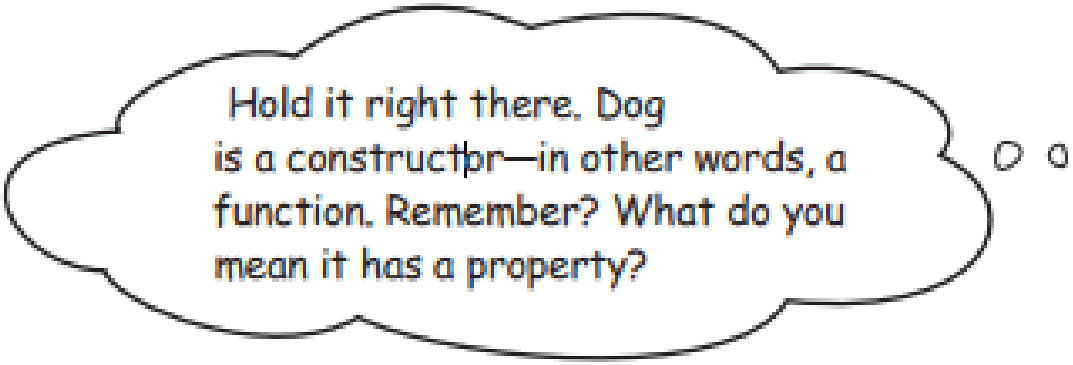
Every dog has been customized with its name, breed and weight, but relies on the prototype for the species property and the bark method.

Overriding the prototype



Overriding the prototype (Cont ..)

`Dog.prototype` ← If you look at your Dog constructor, it has a prototype property that holds a reference to the actual prototype.



How to set up the prototype

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}
```

This is the constructor to create an instance of a dog. Each instance has its own name, breed and weight., so let's incorporate those into the constructor.

Dog
name: "Spot"
breed: "Chihuahua"
weight: 10

But we're going to get our methods from the prototype, so we don't need them in the constructor.

We want it to have the species property and the bark, run and wag methods

our dog prototype

```
Dog.prototype.species = "Canine";

Dog.prototype.bark = function() {
    if (this.weight > 25) {
        console.log(this.name + " says Woof!");
    } else {
        console.log(this.name + " says Yip!");
    }
};

Dog.prototype.run = function() {
    console.log("Run!");
};

Dog.prototype.wag = function() {
    console.log("Wag!");
};
```

We assign the string "Canine" to the prototype's species property.

And for each method, we assign the appropriate function to the prototype's bark, run and wag properties respectively.

Serious Coding

Don't forget about chaining:

```
Dog.prototype.species
```

Start with Dog and grab its prototype property, which is a reference to an object that has a species property.

Test drive the prototype with some dogs

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}

Dog.prototype.species = "Canine";

Dog.prototype.bark = function() {
    if (this.weight > 25) {
        console.log(this.name + " says Woof!");
    } else {
        console.log(this.name + " says Yip!");
    }
};

Dog.prototype.run = function() {
    console.log("Run!");
};

Dog.prototype.wag = function() {
    console.log("Wag!");
};

var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);

fido.bark();
fido.run();
fido.wag();

fluffy.bark();
fluffy.run();
fluffy.wag();

spot.bark();
spot.run();
spot.wag();
```

Here's the Dog constructor.

And here's where we add properties and methods to the dog prototype.

We're adding one property and three methods to the prototype.

Now, we create the dogs like normal...

... and then we call the methods for each dog, just like normal. Each dog inherits the methods from the prototype.

But wait a second, didn't Spot want his bark to be WOOF!?

JavaScript console

```
Fido says Woof!
Run!
Wag!
Fluffy says Woof!
Run!
Wag!
Spot says Yip!
Run!
Wag!
```

Each dog is barking, running and wagging. Good.

Give Spot his WOOF! in code

Hey, don't forget about me. I requested a bigger WOOF!

Spot requested a **bigger WOOF!** so we need to **override** the prototype to give him his own custom **bark method**

The rest of the code goes here. We're just saving trees, or bits, or our carbon footprint, or something...

```
...
var spot = new Dog("Spot", "Chihuahua", 10);

spot.bark = function() {
    console.log(this.name + " says WOOF!");
};

// calls to fido and fluffy are the same

spot.bark();
spot.run();
spot.wag();
```

The only change we make to the code is to give Spot his own custom bark method.

We don't need to change how we call Spot's bark method at all.

JavaScript console

```
Fido says Woof!
Run!
Wag!
Fluffy says Woof!
Run!
Wag!
Spot says WOOF!
Run!
Wag!
```

Using the **prototype** Property

```
<p id="demo"></p>
<script>
    function Car(_name, _model, _weight, _color){
        this.name = _name;
        this.model = _model;
        this.weight = _weight;
        this.color =_color;
        this.info = function(){
            return this.name + ' - ' + this.model;
        }
     }
    //Su dung prototype - Bo sung thuoc tinh
    Car.prototype.year = 2018;
    //tao moi funtction cho Object
    Car.prototype.showInfo = function(){
        return 'Name:'+this.name + ' Model:' + this.model + ' Year:'+this.year;
    }
    //Tao doi duong cua Car
     var xe1 = new Car("Vinfast","Sudan",1000,"Black");
     //Xem doi tuong vua tao
     document.getElementById("demo").innerHTML = xe1.showInfo();
</script>
```

Code Magnets

- We had an object diagram on the fridge, and then someone came and messed it up. Can you help put it back together? To reassemble it, we need two instances of the robot prototype. One is Robby, created in 1956, owned by Dr. Morbius, has an on/off switch and runs to Starbucks for coffee. We've also got Rosie, created in 1962, who cleans house and is owned by George Jetson. Good luck (oh, and there might be some extra magnets below)!

Here's the prototype your robots can inherit from.

Robot Prototype

maker: "ObjectsRUs"

speak()
makeCoffee()
blinkLights()

Build the object diagram here.

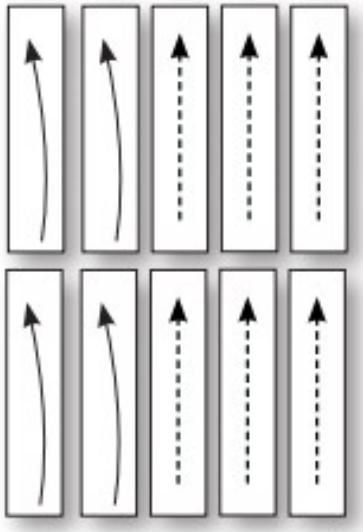
Here's the prototype your robots can inherit from.

Robot Prototype

maker: "ObjectsRUs"

speak()
makeCoffee()
blinkLights()

Build the object diagram here.



Robot

name: "Robby"

owner: Dr. Morbius

owner: George Jetson

Robot

name: "Rosie"

onOffSwitch: true

year: 1956

year: 1962

makeCoffee()

makeCoffee()

cleanHouse()

```
function
blinkLights() {
  // code 4 lights
}
```

```
function
cleanHouse() {
  // code 4 clean
}
```

```
function
makeCoffee() {
  // code 4 coffee
}
```

```
function speak() {
  // code to speak
}
```

```
function
makeCoffee() {
  // starbucks
}
```

Exercise

Here's the basic Robot constructor. You still need to set up its prototype.

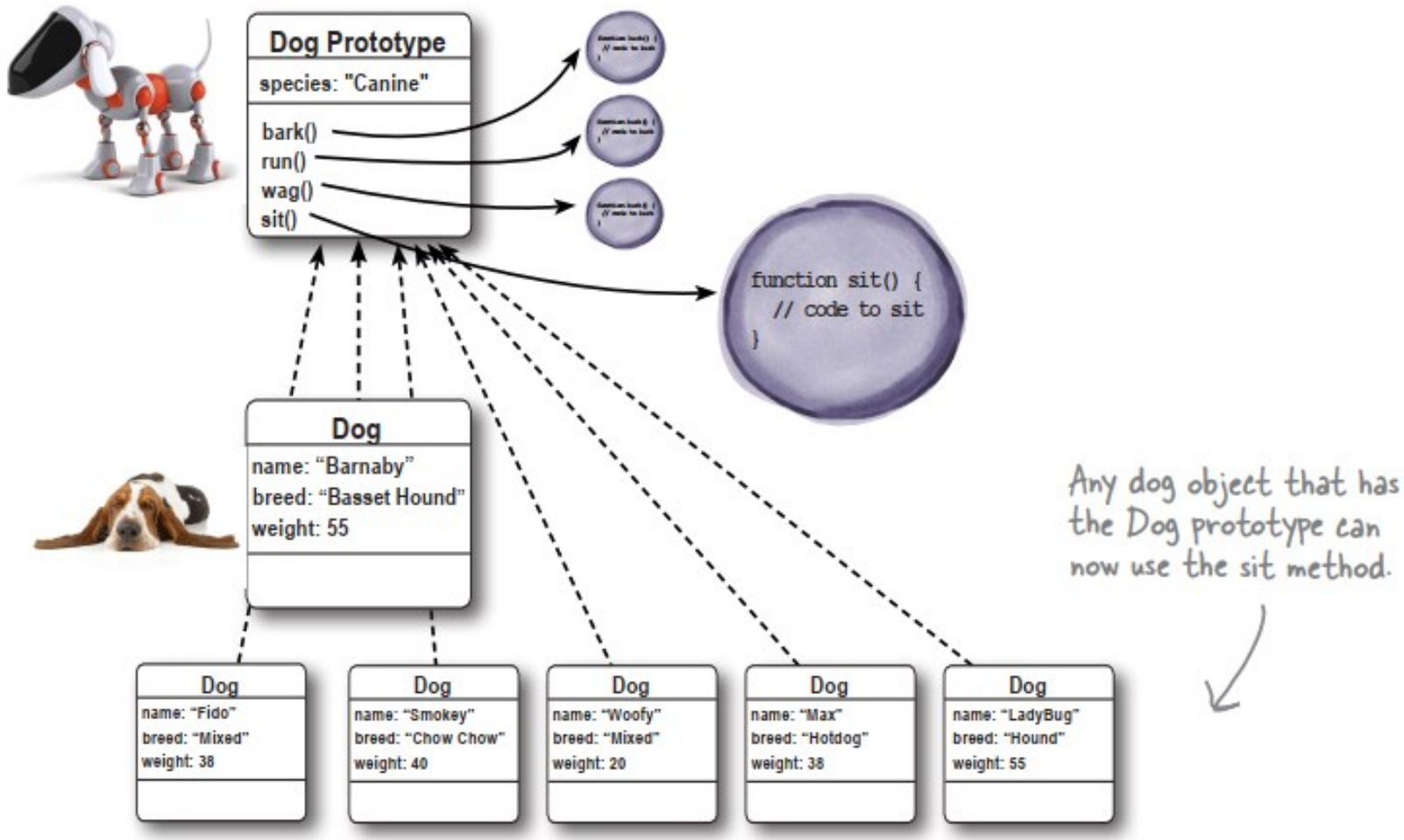
```
function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}
Robot.prototype.maker =

Robot.prototype.speak =

Robot.prototype.makeCoffee =

Robot.prototype.blinkLights =

var robby =
var rosie =

robby.onOffSwitch =
robby.makeCoffee =


rosie.cleanHouse =


console.log(robby.name + " was made by " + robby.maker +
            " in " + robby.year + " and is owned by " + robby.owner);
robby.makeCoffee();
robby.blinkLights();

console.log(rosie.name + " was made by " + rosie.maker +
            " in " + rosie.year + " and is owned by " + rosie.owner);
rosie.cleanHouse();
```

You'll want to set up the robot prototype here.

Write your code to create the Robby and Rosie robots here. Make sure you add any custom properties they have to the instances.

Use this code to test your instances to make sure they are working properly and inheriting from the prototype.

Prototypes are dynamic



Best Dog in Show

Wonderful work on the Dog constructor! We'd love to get you engaged on our dog show simulator. Show dogs are a little different, so they need additional methods (see below).

Thanks! –Webville Kennel Club

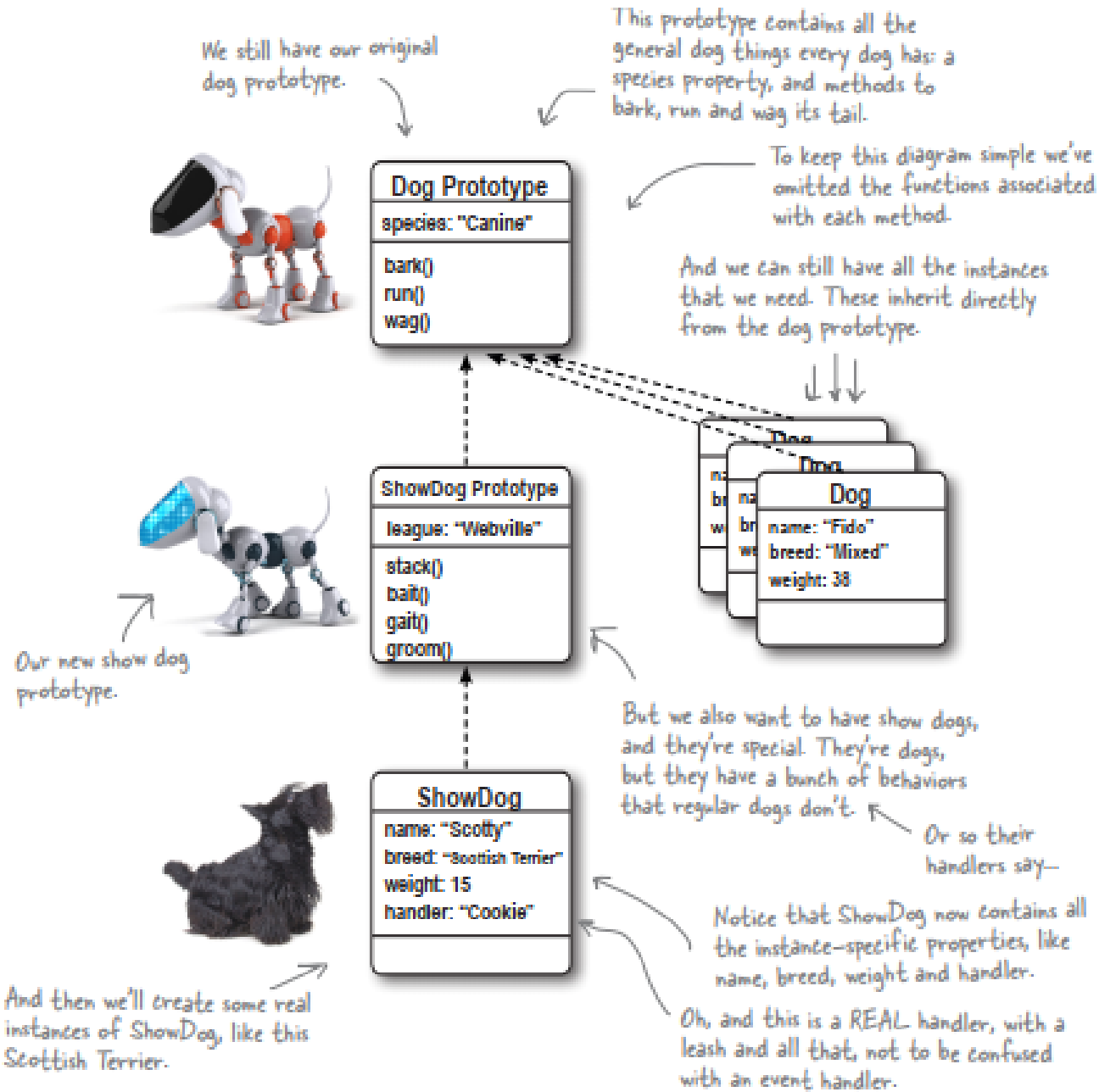
stack() – otherwise known as stand at attention.

gait() – this is like running. The method takes a string argument of "walk", "trot", "pace", or "gallop".

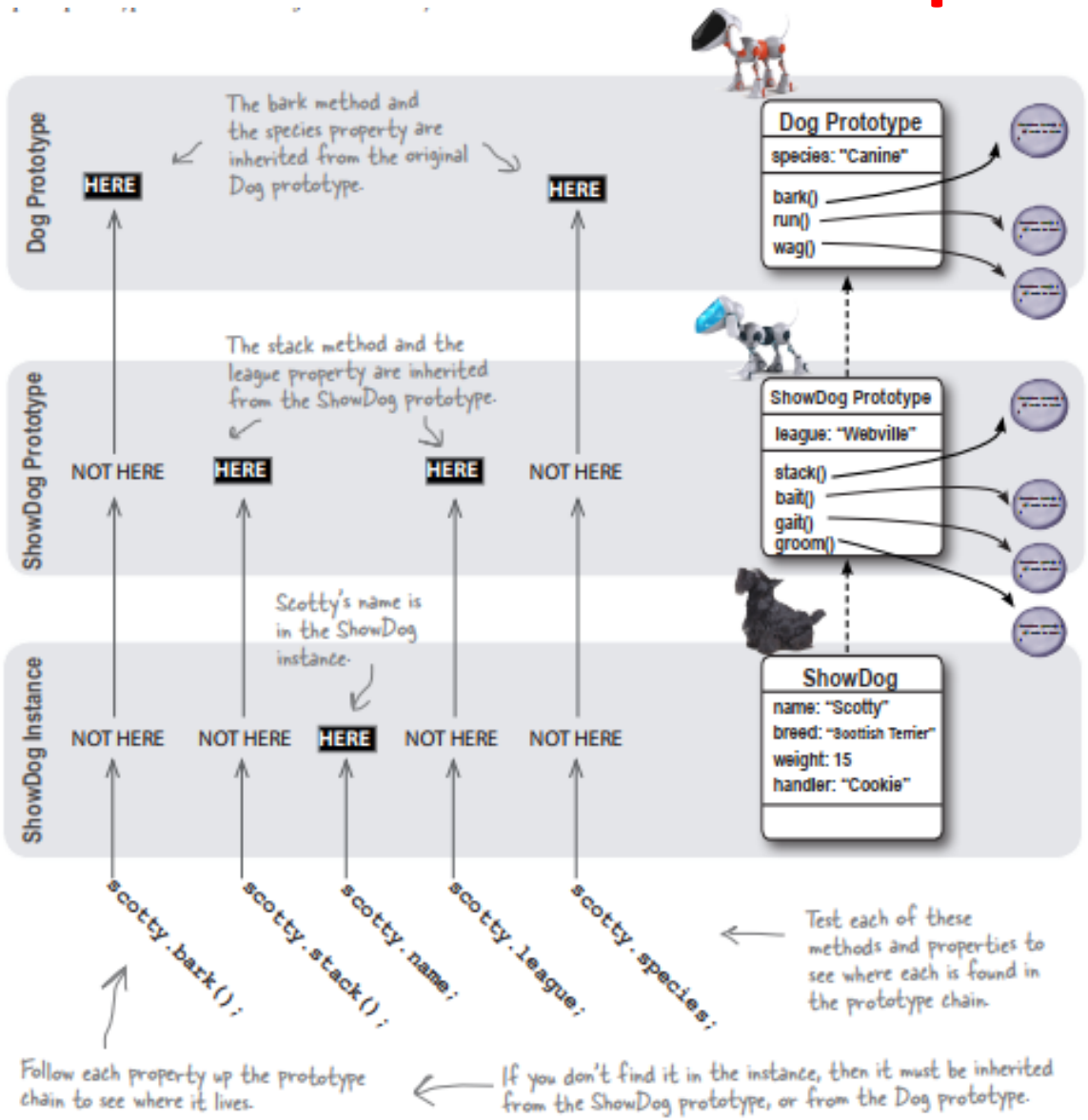
bait() – give the dog a treat.

groom() – doggie shampoo time.

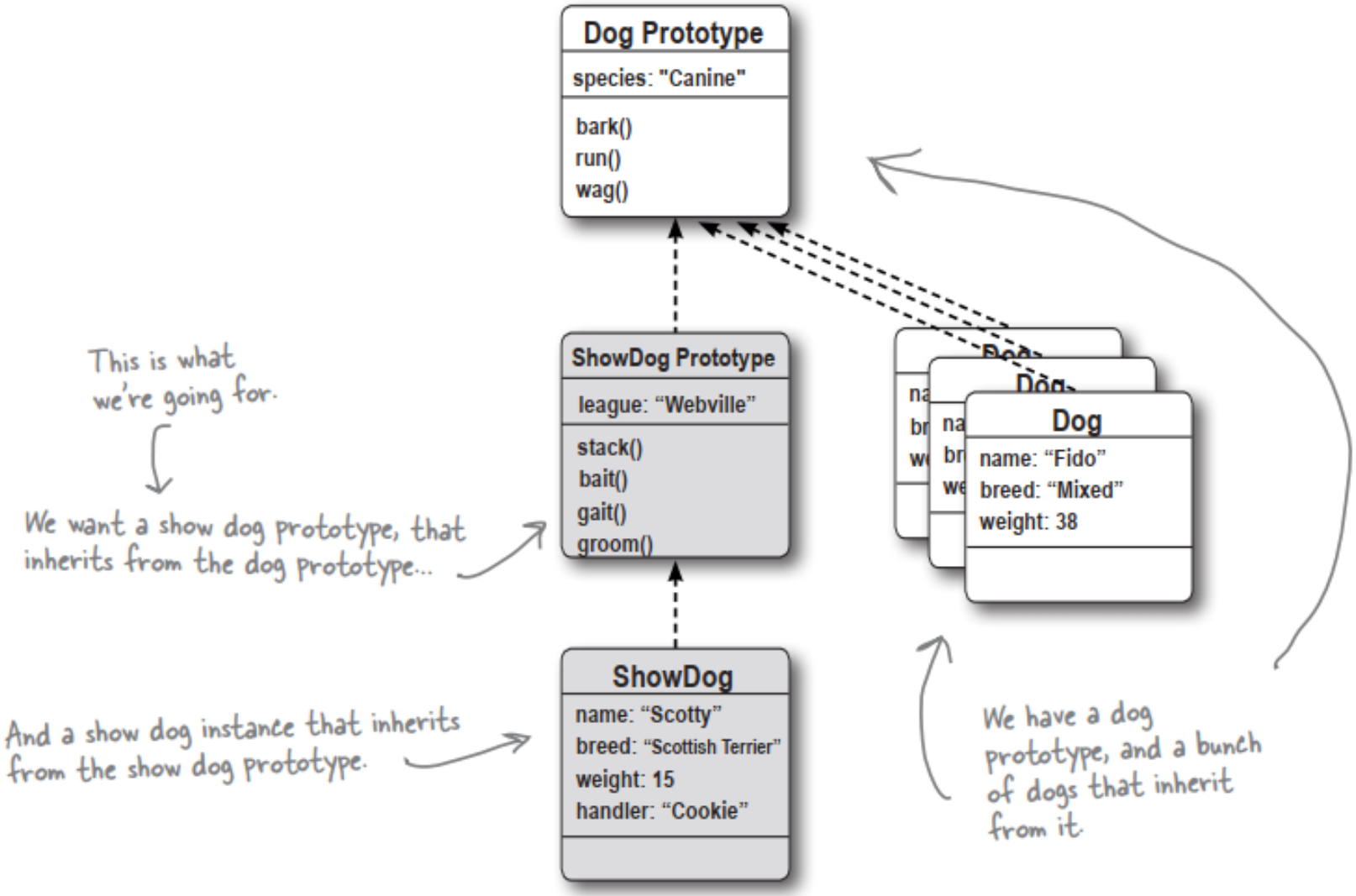
Setting up a chain of prototypes



How inheritance works in a prototype chain?



Creating the show dog prototype



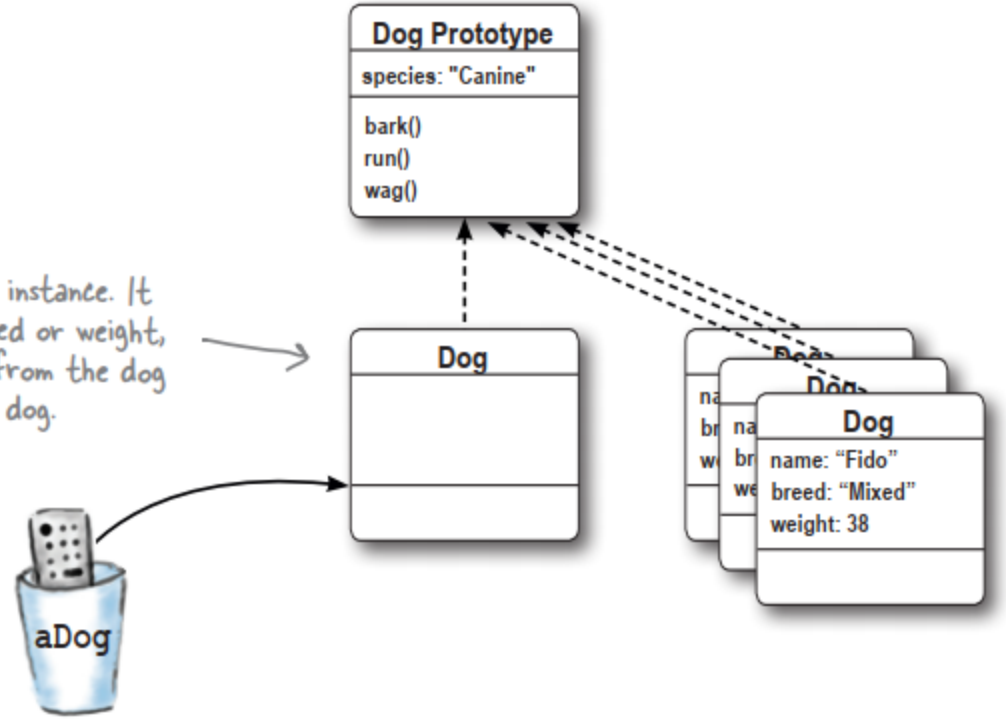
First, we need an object that inherits from the dog prototype

To create an object that inherits from the dog prototype, we just use new with the Dog constructor.

```
var aDog = new Dog();
```

We'll talk about what happened to the constructor arguments in a minute...

We've created a new dog instance. It doesn't have a name, breed or weight, but we know it inherits from the dog prototype, because it's a dog.



Next, turning our dog instance into a show dog prototype

We have a dog instance, but how do we make that our show dog prototype object?

→ We do this by **assigning** the **dog instance** to the prototype property of our **ShowDog constructor**:

```
function ShowDog(name, breed, weight, handler) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.handler = handler;
}
```

This constructor takes everything we need to be a dog (name, breed, weight), and to be a show dog (a handler).

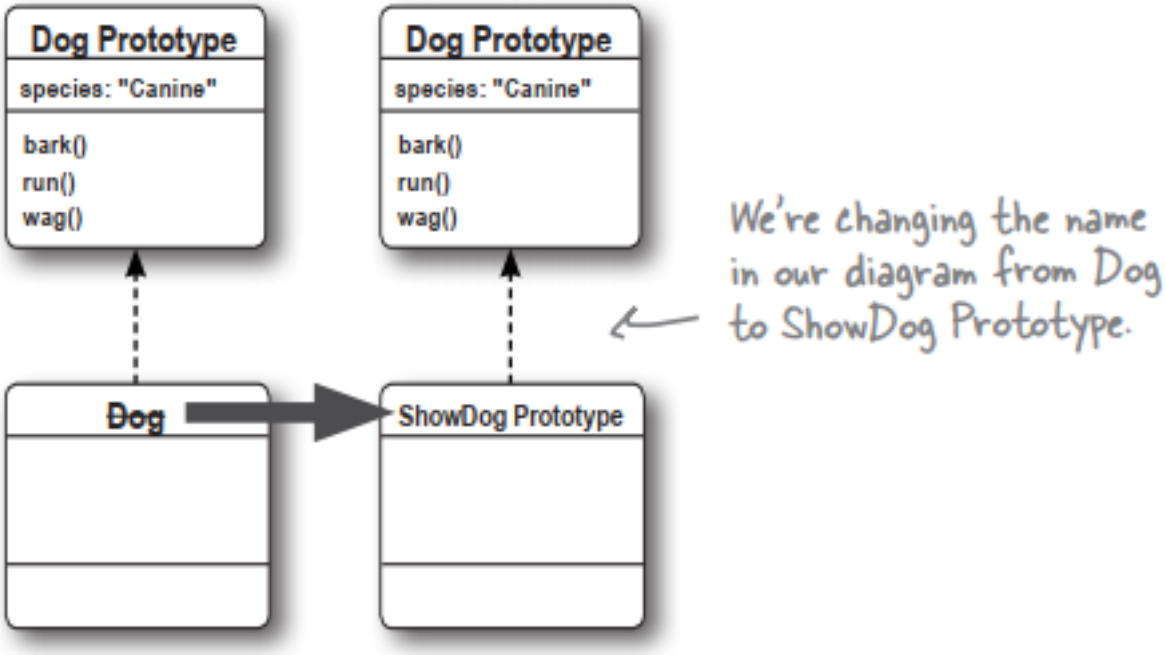
Now that we have a constructor, we can set its prototype property to a new dog instance:

```
ShowDog.prototype = new Dog();
```

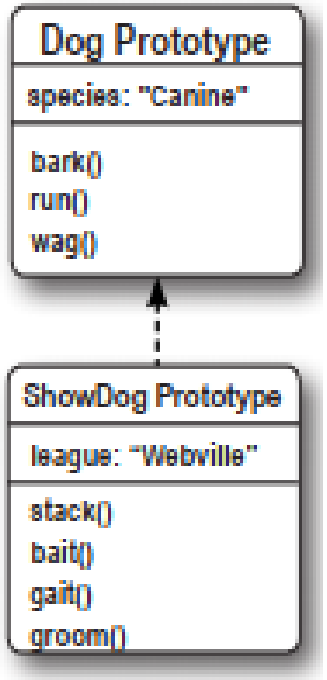
We could have used our dog instance created on the previous page, but we can skip the variable assignment and just assign the new dog straight to the prototype property instead.

Our object **diagram** accurately reflects the roles these objects **are playing** by **changing** the label **"Dog" to "ShowDog Prototype".**

But keep in mind, the **show dog prototype *is still a dog instance***



Now it's time to **fill** in **the prototype**




```
function ShowDog(name, breed, weight, handler) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.handler = handler;
}

ShowDog.prototype = new Dog();

Showdog.prototype.league = "Webville";

ShowDog.prototype.stack = function() {
        console.log("Stack");
};

ShowDog.prototype.bait = function() {
        console.log("Bait");
};

ShowDog.prototype.gait = function(kind) {
        console.log(kind + "ing");
};

ShowDog.prototype.groom = function() {
        console.log("Groom");
};
```

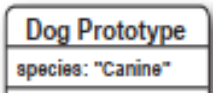
Remember, the ShowDog constructor looks a lot like the Dog constructor. A show dog needs a name, breed, weight, plus one extra property, a handler (the person who handles the show dog). These will end up being defined in the show dog instance.

All our show dogs are in the Webville league, so we'll add this property to the prototype.

Here are all the methods we need for show dogs. We'll just keep them simple for now.

We're adding all these properties to the show dog prototype so all show dogs inherit them.

This is where we're taking the dog instance that is acting as the show dog prototype, and we're adding new properties and methods.

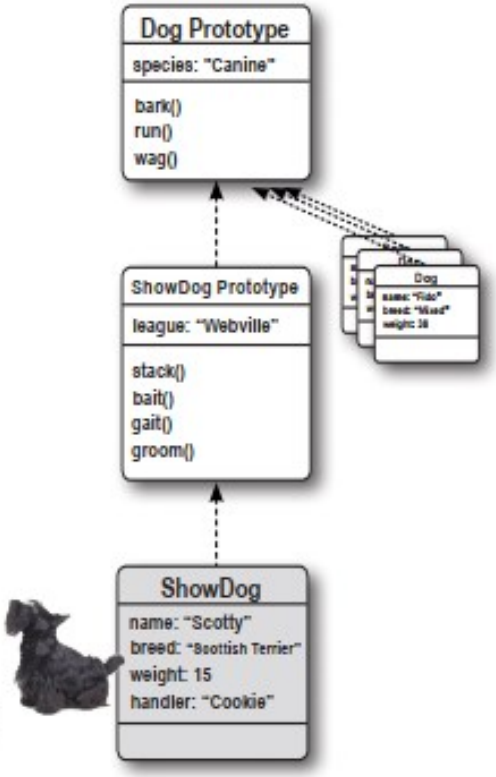


Creating a show dog instance

```
function ShowDog(name, breed, weight, handler) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.handler = handler;
}

ShowDog.prototype = new Dog();

Showdog.prototype.league = "Webville";

ShowDog.prototype.stack = function() {
      console.log("Stack");
};

ShowDog.prototype.bait = function() {
      console.log("Bait");
};

ShowDog.prototype.gait = function(kind) {
      console.log(kind + "ing");
};

ShowDog.prototype.groom = function() {
      console.log("Groom");
};
```

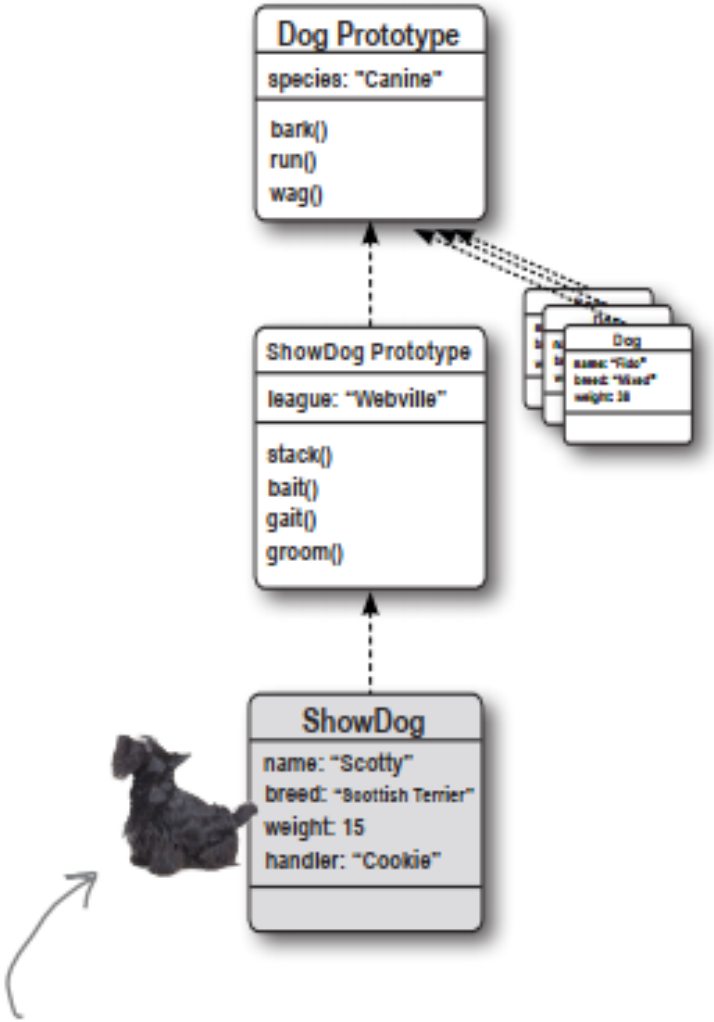
Here's our new show dog, scotty.

```
var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
```



And here's our show dog instance. It inherits from the show dog prototype, which inherits from the dog prototype. Just what we wanted. If you go back and look at page 592, you'll see we've completed the prototype chain.

Test drive the show dog




```
scotty.stack();
scotty.bark();
console.log(scotty.league);
console.log(scotty.species);
```

Here's what we got

JavaScript console

```
Stack
Scotty says Yip!
Webville
Canine
```

A final cleanup of show dogs

```
var fido = new Dog("Fido", "Mixed", 38);
if (fido instanceof Dog) {
    console.log("Fido is a Dog");
}
if (fido instanceof ShowDog) {
    console.log("Fido is a ShowDog");
}

var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
if (scotty instanceof Dog) {
    console.log("Scotty is a Dog");
}
if (scotty instanceof ShowDog) {
    console.log("Scotty is a ShowDog");
}
console.log("Fido constructor is " + fido.constructor);
console.log("Scotty constructor is " + scotty.constructor);
```

Run this code and provide your output below.

```
function ShowDog(name, breed, weight, handler) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.handler = handler;
}
ShowDog.prototype = new Dog();
ShowDog.prototype.constructor = ShowDog;
```

Here we're taking the show dog prototype and explicitly setting its constructor property to the ShowDog constructor.

Remember this is a best practice, without it your code still works as expected.

Note that we didn't have to do this for the dog prototype because it came with the constructor property set up correctly by default.

That's all you need to do. When we check Scotty again he should have the correct constructor property, as should all other show dogs.

A little more cleanup

Let's look again at the constructor:

```
function ShowDog(name, breed, weight, handler) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.handler = handler;
}
```

If you didn't notice, this code is replicated from the Dog constructor.

```
function ShowDog(name, breed, weight, handler) {
    Dog.call(this, name, breed, weight);
    this.handler = handler;
}
```

This bit of code is going to reuse the Dog constructor code to process the name, breed, and weight.

But we still need to handle the handler in this code because the Dog constructor doesn't know anything about it.

Dog is the function we're going to call.

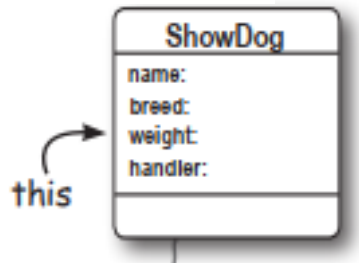
Whatever is in this is used for this in the body of the Dog function.

```
Dog.call(this, name, breed, weight);
```

The rest of the arguments are just passed to Dog like normal.

With this code we're calling the Dog constructor function but telling it to use our ShowDog instance as this, and so the Dog function will set the name, breed and weight properties in our ShowDog object.

call is the method of Dog we're calling. The call method will cause the Dog function to be called. We use the call method instead of just calling Dog directly so we can control what the value of this is.




```
function ShowDog(name, breed, weight, handler) {
    Dog.call(this, name, breed, weight);

    function Dog(name, breed, weight) {
        this.name = name;
        this.breed = breed;
        this.weight = weight;
    }

    this.handler = handler;
}
```

We execute the body of Dog as normal, except that this is a ShowDog, not a Dog object.

The this object created by new for ShowDog gets used as this in the body of Dog.

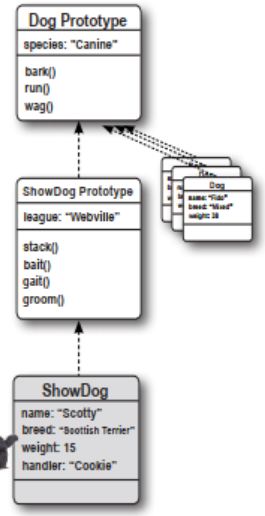
The final test drive

```
function ShowDog(name, breed, weight, handler) {
    Dog.call(this, name, breed, weight);
    this.handler = handler;
}
ShowDog.prototype = new Dog();
ShowDog.prototype.constructor = ShowDog;
ShowDog.prototype.league = "Webville";
ShowDog.prototype.stack = function() {
    console.log("Stack");
};

ShowDog.prototype.bait = function() {
      console.log("Bait");
};
ShowDog.prototype.gait = function(kind) {
    console.log(kind + "ing");
};
ShowDog.prototype.groom = function() {
    console.log("Groom");
};
var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
var beatrice = new ShowDog("Beatrice", "Pomeranian", 5, "Hamilton");
```

We've brought all the ShowDog code together here. Add this to the file with your Dog code to test it.

We've added some test code below.

Create some dogs and some show dogs.




```
fido.bark();
fluffy.bark();
spot.bark();
scotty.bark();
beatrice.bark();
scotty.gait("Walk");
beatrice.groom();
```

JavaScript console

```
Fido says Woof!
Fluffy says Woof!
Spot says Yip!
Scotty says Yip!
Beatrice says Yip!
Walking
Groom
```

Object.**keys()**

- Object.keys() creates an array containing the keys of an object.

```
//Initialize an Object
var employees = {
    name: 'Linh',
    genner: 'Nam',
    birthday: '09/01/2019',
    level: 'Thac sy'
}
//Get the keys of the Object
var emp_keys = Object.keys(employees);
console.log(emp_keys);
```

```
▼(4) ["name", "genner", "birthday", "level"]
    0: "name"
    1: "genner"
    2: "birthday"
    3: "level"
    length: 4
```

Object.**keys()**

- **Object.keys()** can be used to iterate through the keys and values of an object.

```
//Initialize an Object
var employees = {
    name: 'Linh',
    genner: 'Nam',
    birthday: '09/01/2019',
    level: 'Thac sy'
}
// Iterate through the keys
Object.keys(employees).forEach(key => {
    var value = employees[key];
    console.log('Key:'+key+' Value:'+value);
});
```

```
Key:name Value:Linh
Key:genner Value:Nam
Key:birthday Value:09/01/2019
Key:level Value:Thac sy
```

Object.**keys()**

- **Object.keys** is also useful for checking the length of an object → **Object.keys(employees).length;**

```
//Initialize an Object
var employees = {
    name: 'Linh',
    genner: 'Nam',
    birthday: '09/01/2019',
    level: 'Thac sy'
}
//Get lenght of Object
var len = Object.keys(employees).length;
console.log(len);
```

Object.**values()**

- **Object.values()** creates an array containing the values of an object.

```
//Initialize an Object
var employees = {
    name: 'Linh',
    genner: 'Nam',
    birthday: '09/01/2019',
    level: 'Thac sy'
}
// Get all values of the object
var emp_value = Object.values(employees);
console.log(emp_value);
```

```
(4) ["Linh", "Nam", "09/01/2019", "Thac sy"]
  0: "Linh"
  1: "Nam"
  2: "09/01/2019"
  3: "Thac sy"
  length: 4
```

Object.entries()

- Object.entries() creates a nested array of the key/value pairs of an object.

```
// Initialize an object
var operatingSystem = {
    name: 'Ubuntu',
    version: 18.04,
    license: 'Open Source'
};
// Get the object key/value pairs
var entries = Object.entries(operatingSystem);
console.log(entries);
```

```
▼(3) [Array(2), Array(2), Array(2)]
  ▶0: (2) ["name", "Ubuntu"]
  ▶1: (2) ["version", 18.04]
  ▶2: (2) ["license", "Open Source"]
   length: 3
  ▶__proto__: Array(0)
```

Object.entries()

- Once we have the **key/value pair arrays**, we can use the **forEach()** method to loop through and work with the results.

```
// Loop through the results
entries.forEach(entry => {
    var key = entry[0];
    var value = entry[1];
    console.log(`${key}: ${value}`);
});
```

```
Output
name: Ubuntu
version: 18.04
license: Open Source
```

Object.assign()

- Object.assign() is used to **copy values from one object to another**.

```
// Initialize an object
var source_obj = {
    firstName: 'Philip',
    lastName: 'Fry'
};
// Initialize another object
var target_obj = {
    job: 'Delivery Boy',
    employer: 'Planet Express'
};
// Merge the objects
var merge_object = Object.assign(source_obj,target_obj);
console.log(merge_object);
```

```
▼{firstName: "Philip", lastName: "Fry", job: "Delivery Boy", employer: "Planet Express"}
    employer: "Planet Express"
    firstName: "Philip"
    job: "Delivery Boy"
    lastName: "Fry"
  ▶ __proto__: Object
```

```
Object.assign(dest[, src1, src2, src3...])
```

The **String** object

- The String object allows you to associate methods to strings.
- There are differences between a variable that is an instance of String, and a variable to which a string is assigned directly.
- The constructor is also a mean to convert values to string objects.

The **String** object

- The constructor accepts an argument that is a unique literal string or any object that you want to convert to a string.

```
var x = new String("string");

var x = new String("demo");

var s = String(10) + 5
document.write(s)
```

105

The **String** object

- ***length* attribute** → Number of characters in the string.

```
var x = new String(" ");
document.write(x.length);
```

6

- **Chars are accessed by an index** → get the character at a given position.

```
var x = new String("demo");
document.write(x[2])    // s
```

m

```
x = x.substr(0,1) + "Z" + x.substr(2);
document.write(x);
```

dZmo

The **String** object (Cont …)

Method	Description
charAt()	Returns the character at the specified index (position)
charCodeAt()	Returns the Unicode of the character at the specified index
concat()	Joins two or more strings, and returns a new joined strings
endsWith()	Checks whether a string ends with specified string/characters
fromCharCode()	Converts Unicode values to characters
includes()	Checks whether a string contains the specified string/characters
indexOf()	Returns the position of the first found occurrence of a specified value in a string
lastIndexOf()	Returns the position of the last found occurrence of a specified value in a string
localeCompare()	Compares two strings in the current locale
match()	Searches a string for a match against a regular expression, and returns the matches

The **String** object (Cont ...)

repeat()	Returns a new string with a specified number of copies of an existing string
replace()	Searches a string for a specified value, or a regular expression, and returns a new string where the specified values are replaced
search()	Searches a string for a specified value, or regular expression, and returns the position of the match
slice()	Extracts a part of a string and returns a new string
split()	Splits a string into an array of substrings
startsWith()	Checks whether a string begins with specified characters
substr()	Extracts the characters from a string, beginning at a specified start position, and through the specified number of character
substring()	Extracts the characters from a string, between two specified indices
toLocaleLowerCase()	Converts a string to lowercase letters, according to the host's locale
toLocaleUpperCase()	Converts a string to uppercase letters, according to the host's locale
toLowerCase()	Converts a string to lowercase letters

The **String** object (Cont …)

toString()	Returns the value of a String object
toUpperCase()	Converts a string to uppercase letters
trim()	Removes whitespace from both ends of a string
valueOf()	Returns the primitive value of a String object

Date Object

- Creating Date Objects
  - Date objects are created with the new Date() constructor.
  - There are **4 ways** to create a new date object

```
new Date()
new Date(year, month, day, hours, minutes, seconds, milliseconds)
new Date(milliseconds)
new Date(date string)
```

Date Object Methods

Method	Description
getDate()	Returns the day of the month (from 1-31)
getDay()	Returns the day of the week (from 0-6)
getFullYear()	Returns the year
getHours()	Returns the hour (from 0-23)
getMilliseconds()	Returns the milliseconds (from 0-999)
getMinutes()	Returns the minutes (from 0-59)
getMonth()	Returns the month (from 0-11)
getSeconds()	Returns the seconds (from 0-59)
getTime()	Returns the number of milliseconds since midnight Jan 1 1970, and a specified date
getTimezoneOffset()	Returns the time difference between UTC time and local time, in minutes

Date Object Methods

getYear()	Deprecated. Use the getFullYear() method instead
now()	Returns the number of milliseconds since midnight Jan 1, 1970
parse()	Parses a date string and returns the number of milliseconds since January 1, 1970
setDate()	Sets the day of the month of a date object
setFullYear()	Sets the year of a date object
setHours()	Sets the hour of a date object
setMilliseconds()	Sets the milliseconds of a date object
setMinutes()	Set the minutes of a date object
setMonth()	Sets the month of a date object
setSeconds()	Sets the seconds of a date object
setTime()	Sets a date to a specified number of milliseconds after/before January 1, 1970

Date Object Methods

toJSON()	Returns the date as a string, formatted as a JSON date
toLocaleDateString()	Returns the date portion of a Date object as a string, using locale conventions
toLocaleTimeString()	Returns the time portion of a Date object as a string, using locale conventions
toLocaleString()	Converts a Date object to a string, using locale conventions
toString()	Converts a Date object to a string
toTimeString()	Converts the time portion of a Date object to a string
toUTCString()	Converts a Date object to a string, according to universal time
UTC()	Returns the number of milliseconds in a date since midnight of January 1, 1970, according to UTC time
valueOf()	Returns the primitive value of a Date object

JavaScript **Date Formats**

- There are generally 3 types of JavaScript date input formats:

Type	Example
ISO Date	"2015-03-25" (The International Standard)
Short Date	"03/25/2015"
Long Date	"Mar 25 2015" or "25 Mar 2015"

```
var mydate = "2019/01/09";
var date_value = new Date(mydate);

var date_format = new Date(date_value).toLocaleString();
console.log(date_format);
```

```
00:00:00, 9/1/2019
```

The parsing tokens are similar to the formatting tokens used in `moment#format`.

Input	Output
M, MM	Month Number (1 - 12)
MMM, MMMM	Month Name (In currently language set by `moment.lang()`)
D, DD	Day of month
DDD, DDDD	Day of year
d, dd, ddd, dddd	Day of week (NOTE: these formats only make sense when combined with "ww")
e	Day of week (locale) (NOTE: these formats only make sense when combined with "ww")
E	Day of week (ISO) (NOTE: this format only make sense when combined with "WW")
w, ww	Week of the year (NOTE: combine this format with "gg" or "gggg" instead of "YY" or "YYYY")
W, WW	Week of the year (NOTE: combine this format with "GG" or "GGGG" instead of "YY" or "YYYY")
YY	2 digit year (if greater than 68 will return 1900's, otherwise 2000's)
YYYY	4 digit year
gg	2 digit week year (if greater than 68 will return 1900's, otherwise 2000's)
gggg	4 digit week year
GG	2 digit week year (ISO) (if greater than 68 will return 1900's, otherwise 2000's)
GGGG	4 digit week year (ISO)
a, A	AM/PM
H, HH	24 hour time
h, hh	12 hour time (use in conjunction with a or A)
m, mm	Minutes
s, ss	Seconds
S	Deciseconds (1/10th of a second)
SS	Centiseconds (1/100th of a second)
SSS	Milliseconds (1/1000th of a second)
Z, ZZ	Timezone offset as `+07:00` or `+0700`
X	Unix timestamp
LT, L, LL, LLL, LLLL	Locale dependent date and time representation