Code Nation

JavaScript Fundamentals

OOP

{codenation}®

First thing's first

- Make 3 variables: storing a string, a number and a boolean
- Make an array that stores 4 items, add something to the end of the array using a method
- Create a loop to cycle through the array to print out all the values
- Create a function that when called asks you to withdraw an amount. Balance should reduce as appropriate.

Learning Objectives

- To understand the concept of OOP
- To understand and use the idea os inheritance
- To write programs to create both classes and subclasses

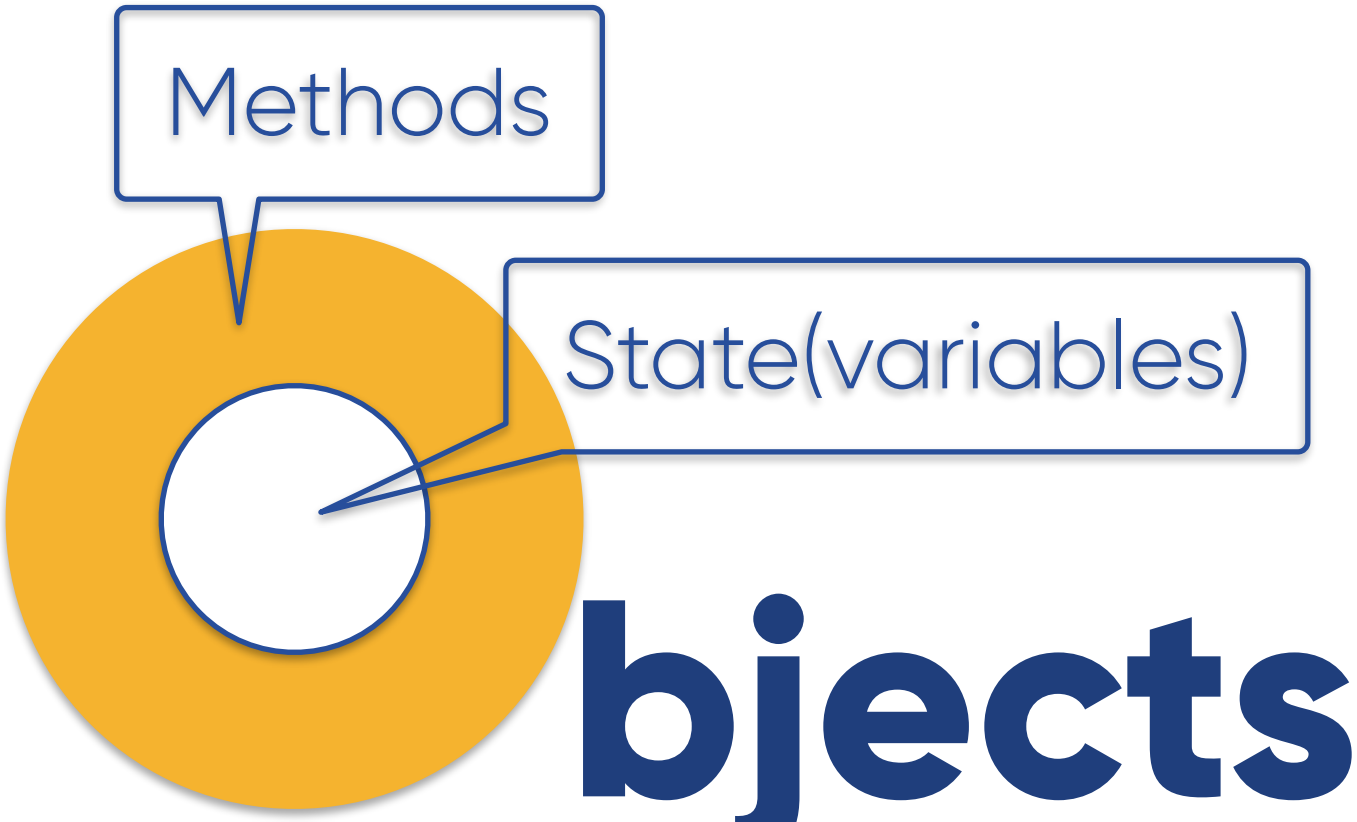
{codenation}®

Object-oriented programming

OOP is a fundamental principle of modern development

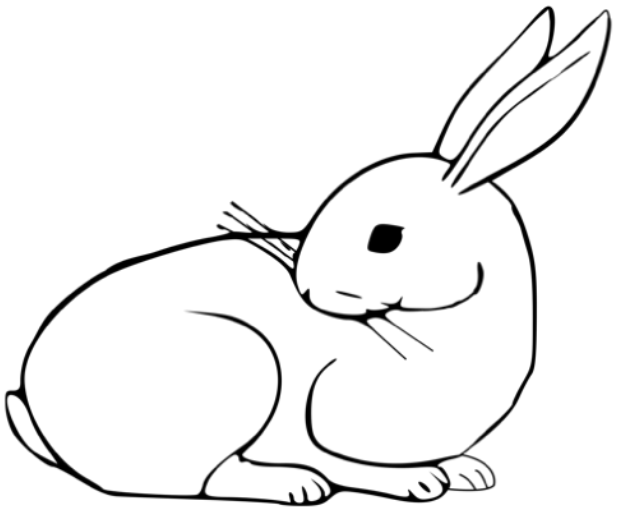
Its fundamental concepts focus on code reusability using classes and objects

Revisiting Objects



bjects

Behaviours	States
hop()	hopping
drink()	thirstLevel
wiggleNose()	wiggling



An object is made up of data and functions to operate on that data

Imagine an object representing a rabbit named Rosie.

This bunny's [name] (key) is Rosie (value).

Let's create Rosie.

```
let Rosie = {
    _name: "Rosie",
    _hops: 0,

    get name() {
      return this._name;
    },

    get hops() {
      return this._hops;
    },

    increaseHops() {
      this._hops++;
    }
}
```

{cn}®

This is cool but what if we've got lots of bunnies?

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}
```

This creates a template for lots of bunny objects

Introducing Classes

Classes are templates for objects. It's where we do our stuff.

Constructors

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}
```

Constructors differentiate object and class syntax

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}
```

Bunny is the name of our class.

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}
```

Bunny is the name of our class.

We call the constructors() method every time we create a new instance of our bunny class.

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}
```

Bunny is the name of our class.

We call the constructors() method every time we create a new instance of our bunny class.

This constructor() method accepts one argument, name.

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}
```

Bunny is the name of our class.

We call the constructors() method every time we create a new instance of our bunny class.

This constructor() method accepts one argument, name.

Under this._name, we create a property called hops, which will keep track of the number of times a bunny hops.

Objects are instances of classes

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}

const rosie = new Bunny("Rosie");

console.log(rosie.name);
```

{cn}®

Let's take this in...

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}

const rosie = new Bunny("Rosie");

console.log(rosie.name);
```

We use the new keyword to create an instance of our bunny class

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}

const rosie = new Bunny("Rosie");

console.log(rosie.name);
```

We create a new variable named rosie that will store an instance of our Bunny class.

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}

const rosie = new Bunny("Rosie");

console.log(rosie.name);
```

The new keyword calls the constructor(), runs the code inside of it, and then returns the new instance.

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}

const rosie = new Bunny("Rosie");

console.log(rosie.name);
```

We pass the "Rosie" string to the Bunny constructor(), which sets the name property to "Rosie".

Create a "Lizzie" object with 0 hops!

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}

const rosie = new Bunny("Rosie");
const lizzie = new Bunny("Lizzie");

console.log(rosie.name);
console.log(lizzie.hops);
```

{cn}®

Create a "Lizzie" object with 0 hops!

We put methods inside objects too, not just data

We have a Bunny class that we store two items of data in.

Let's do something with it.

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}
const rosie = new Bunny("Rosie");
const lizzie = new Bunny("Lizzie");

console.log(rosie.name);
console.log(lizzie.hops);
```

{cn}®

These are known as getters

*There are also setters, check them out!

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}
const rosie = new Bunny("Rosie");
const lizzie = new Bunny("Lizzie");

console.log(rosie.name);
console.log(lizzie.hops);
```

We've added _ to the names, which means don't access these things directly, use the getter methods.

```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}
rosie.increaseHops();
console.log(rosie.hops);
```

We've also created another method increaseHops(), which we can update the number of hops indirectly.

Activity:

Let's create a class called **Cars** for a car parking company.

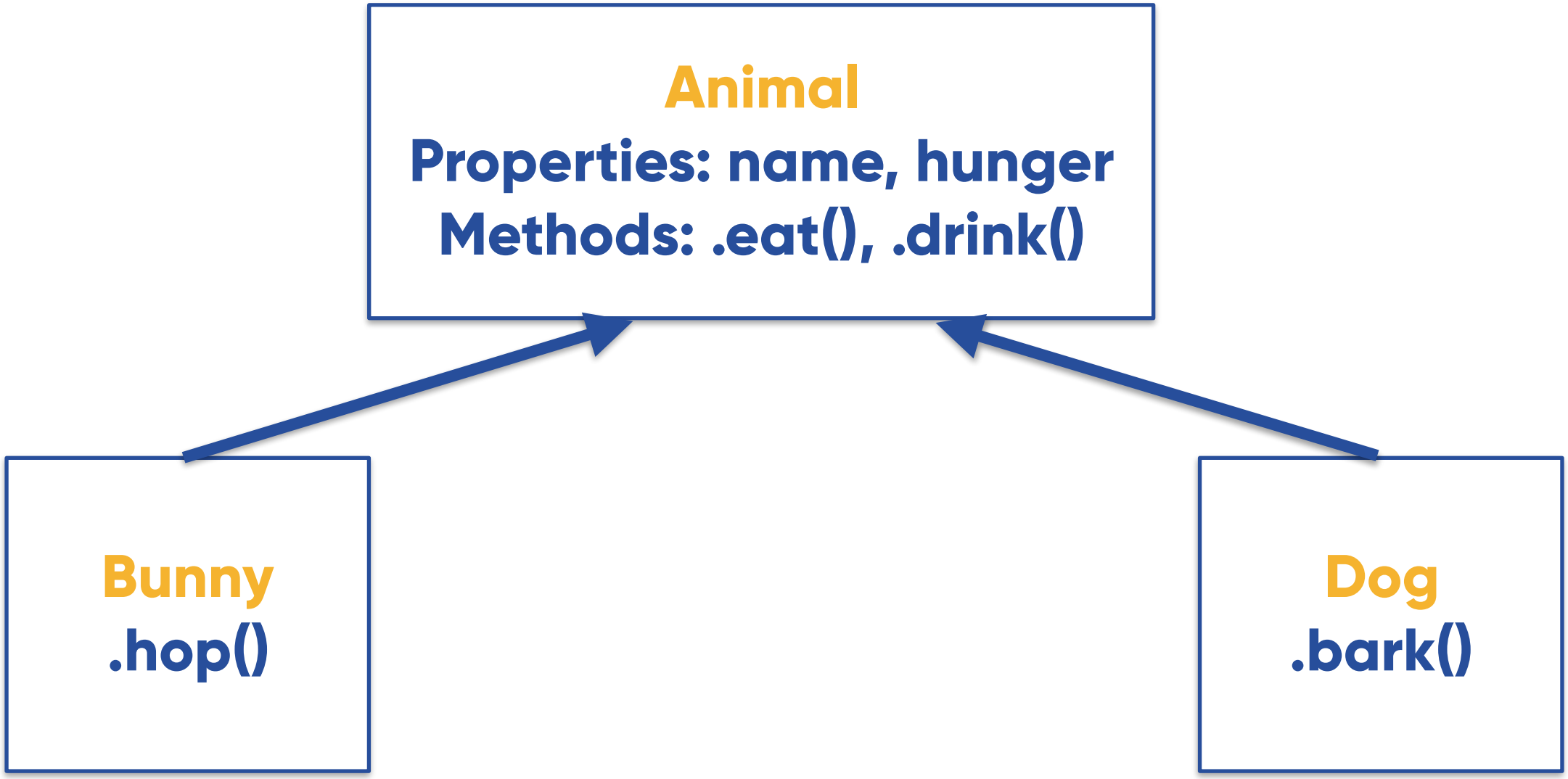
This will allow you to store information of: car registration number, number of hours parked and total amount charged. (Say £1.50 per hour?)

The first car entered the car park, parked for 5 hours and ready to pay. Display this information so the driver can pay for his/her parking fee.

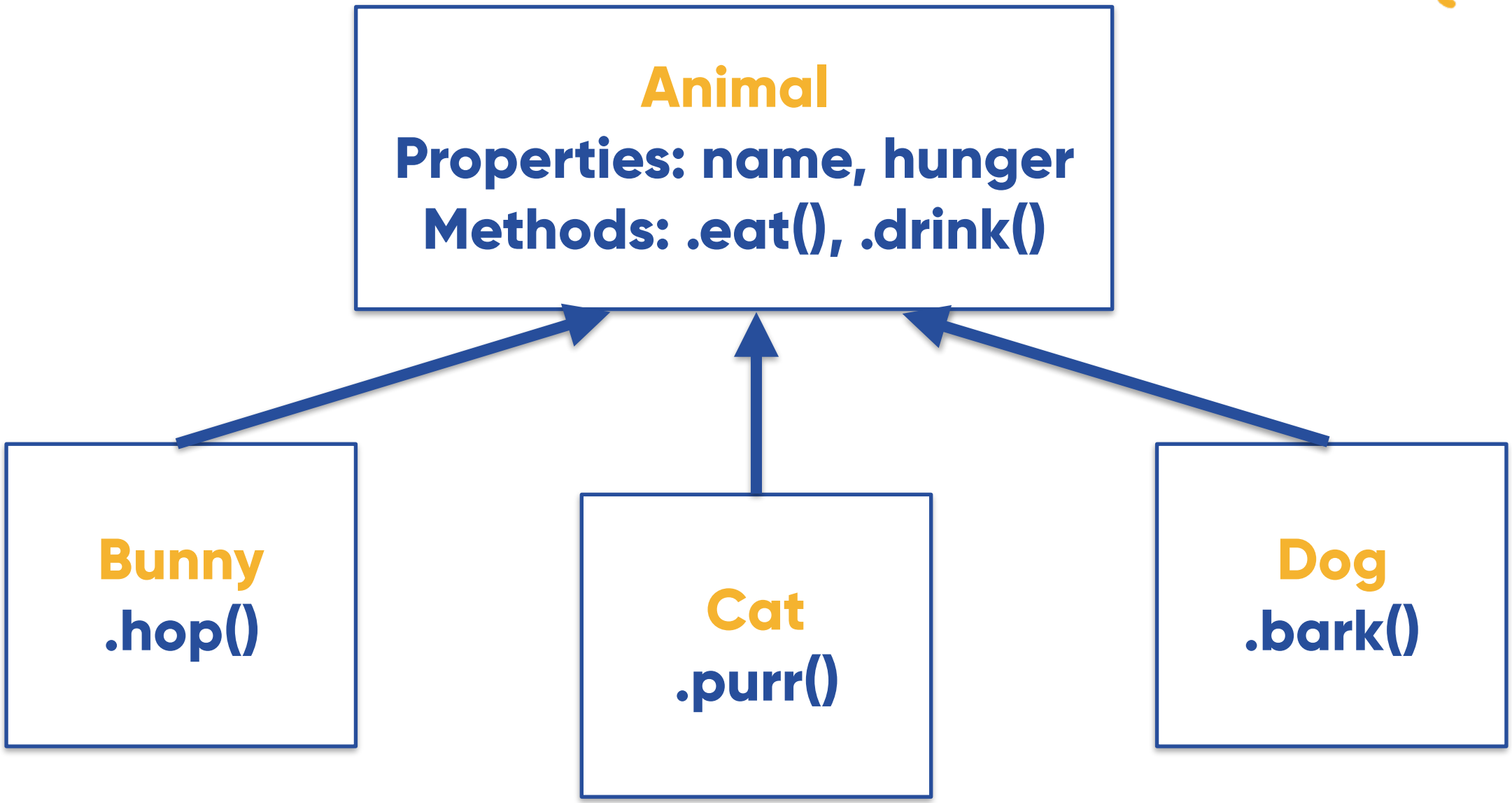
```
class Car{
    constructor(regnum){
        this._regnum = regnum;
        this._hours = 0;
        this._charge = 0.00;
    }
    get regnum(){
        return this._regnum;
    }
    get hours(){
        return this._hours;
    }
    get charge(){
        return this._charge;
    }
    increaseHours(){
        this._hours++;
        this._charge += 1.50;
    }
}
const pay = (reg, hr) => {
    const car = new Car(reg);
    for (i = 0; I < hr; i++){
        car.increaseHours();
    }
    console.log(`You need to pay £${car.charge} for ${car.hours} hours.`);
}
pay("M7 CAR", 5); //Output: You need to pay £7.5 for 5 hours.
```

There are many other ways, this is just one of the ways!

Inheritance


Animal
Properties: name, hunger
Methods: .eat(), .drink()
Bunny
.hop()
Dog
.bark()

Imagine we now added a new member of our animal kingdom: a cat


Animal
Properties: name, hunger
Methods: .eat(), .drink()
Bunny
.hop()
Cat
.purr()
Dog
.bark()

```
class Animal{
    constructor(name){
        this._name = name;
        this._hunger = 100;
        this._thirst = 100;
    }
    get name(){
        return this._name;
    }
    get hunger(){
        return this._hunger;
    }
    get thirst(){
        return this._thirst;
    }
    eat(){
        this._hunger--;
    }
    drink(){
        this.thirst--;
    }
}
```

See how Animal now contains name, hunger and thirst?

```
class Animal{
    constructor(name){
        this._name = name;
        this._hunger = 100;
        this._thirst = 100;
    }
    get name(){
        return this._name;
    }
    get hunger(){
        return this._hunger;
    }
    get thirst(){
        return this._thirst;
    }
    eat(){
        this._hunger--;
    }
    drink(){
        this.thirst--;
    }
}
class Bunny extends Animal {
    constructor(name, lovesCarrot){
        super(name);
        this._lovesCarrot = lovesCarrot;
    }
    get lovesCarrots(){
        return this._lovesCarrot;
    }
}

const rosie = new Bunny("Rosie", true);
```

{cn}®

Our subclasses, those which inherit from Animal, now won't need that coding up again.

We can also pass in an array to a constructor

```
class Animal{
    constructor(name){
        this._name = name;
        this._hunger = 100;
        this._thirst = 100;
    }
    get name(){
        return this._name;
    }
    get hunger(){
        return this._hunger;
    }
    get thirst(){
        return this._thirst;
    }
    eat(){
        this._hunger--;
    }
    drink(){
        this.thirst--;
    }
}
class Bunny extends Animal {
    constructor(name, lovesCarrot, favFood){
        super(name);
        this._lovesCarrot = lovesCarrot;
        this._favFood = favFood;
    }
    get lovesCarrots(){
        return this._lovesCarrot;
    }
    get favFood(){
        return this._favFood;
    }
}
const rosie = new Bunny(
    "Rosie",
    true,
    ["basil", "rockets", "broccoli"]
);
```

Our subclasses, those which inherit from Animal, now won't need that coding up again.

Summary

Think about all the lines of code we've been able to save

Inheritance means we can reuse code and it's nice to read and work with

Learning Objectives

- To understand the concept of OOP
- To understand and use the idea os inheritance
- To write programs to create both classes and subclasses

{codenation}®

Activity(1): Car Park

Let's continue with our car park project.

Add a **subclass** for staff, so that staff can provide their staff number, and credits they have left to pay for the car park fees.

Given a staff member parked for 5 hours as before, show how much it will be charged and remaining balance.

Activity(2): Cyber Pet

Cyber pet time!

User selects the kind of animal they'd like (dog, cat, rabbit) and you have to play with it, feed it, give it drinks etc.

There should be consequences across the board – if you don't play, it gets bored, if you do, it's happy, but gets thirsty, that kind of thing.