First thing's first... I'm the realest

- 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.

Code Nation

JavaScript Fundamentals

OOP

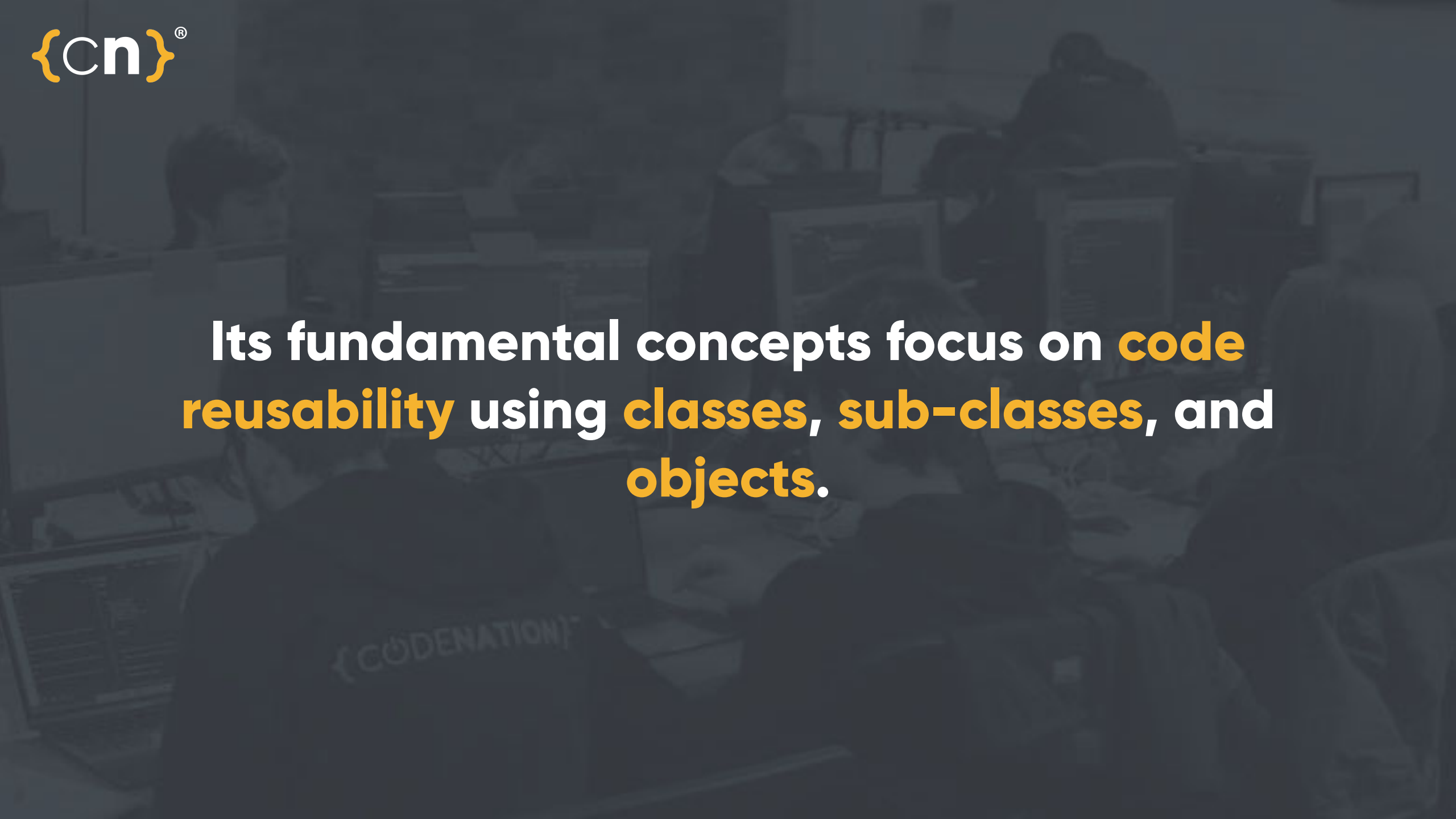
{codenation}®

Learning Objectives

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

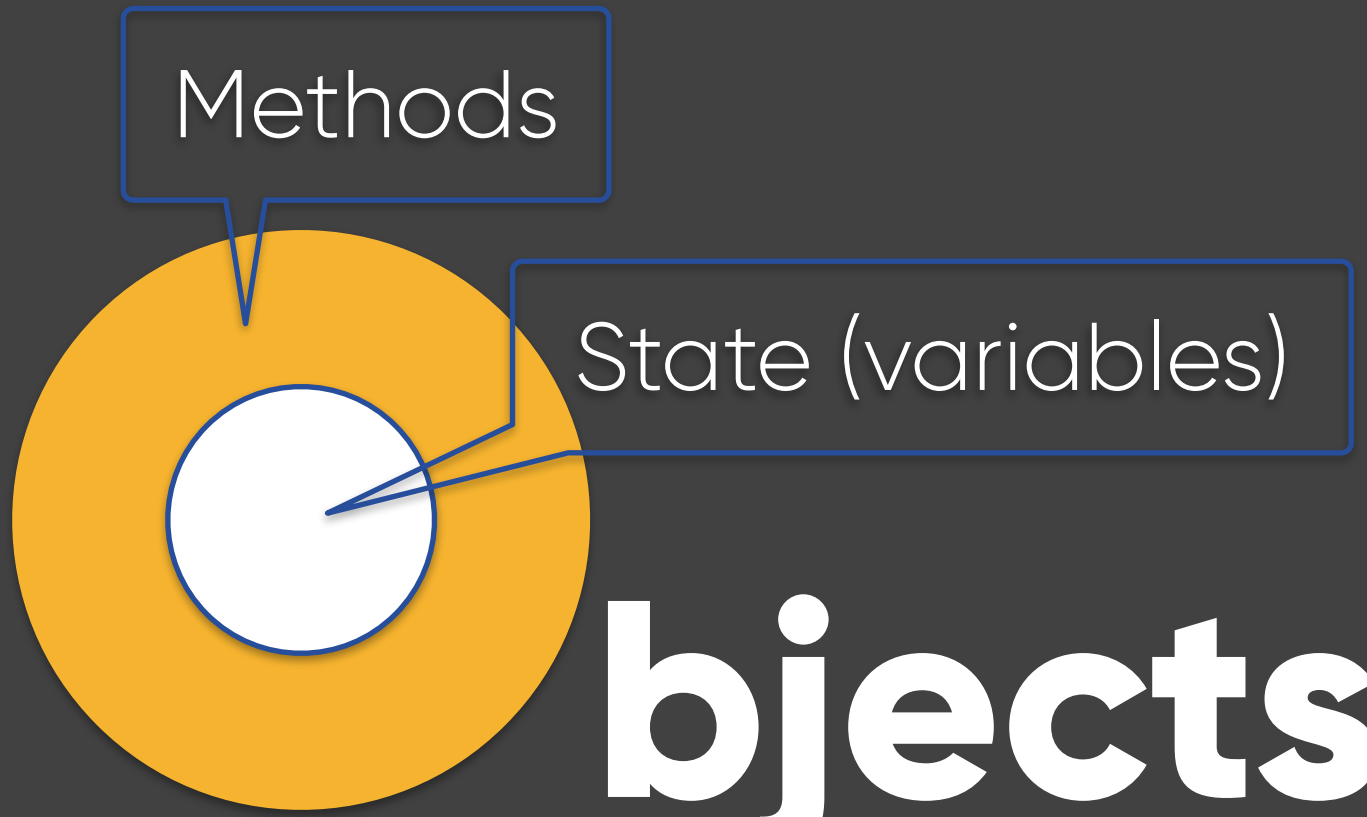

{cn}®
OOP's I did it again... (never enough puns)
Object-oriented Programming


{cn}®
OOP is a fundamental principle of modern development - not something to be afraid of.


{cn}®
Its fundamental concepts focus on code reusability using classes, sub-classes, and objects.

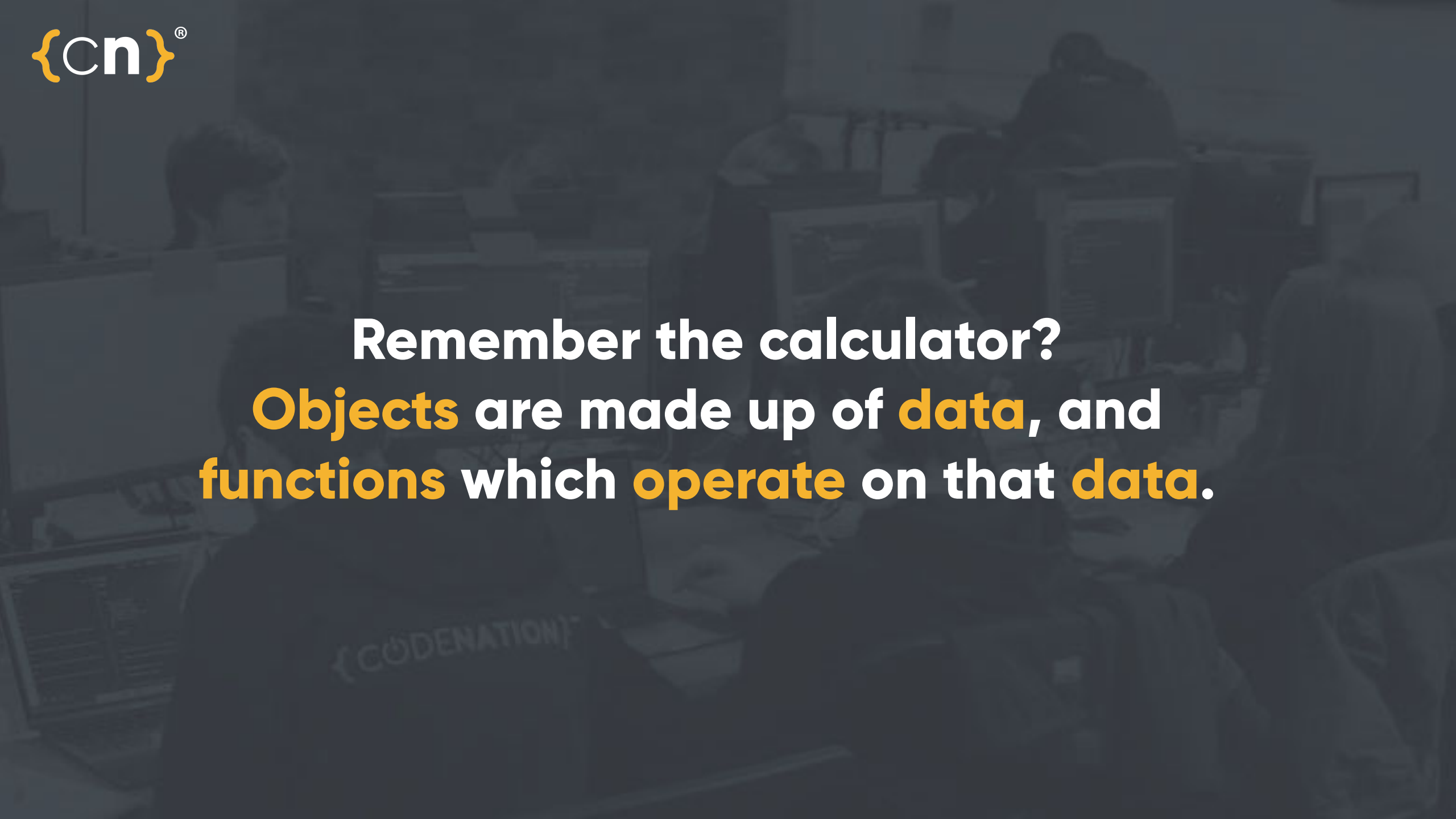

{cn}®
The 4 key pillars of OOP:
1 Encapsulation
2 Abstraction
3 Polymorphism
4 Inheritance


{cn}®
Let's revisit objects.



bjects

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


Remember the calculator?
Objects are made up of data, and
functions which 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++;
    }
}
```

{cn}®

This creates a **template** for lots of bunny objects.


{cn}®
Classes.
So classy.


{cn}®
In short, classes are object templates.
We can create multiple objects from the same template – so convenient!


Let's start with:
Constructors.

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

{cn}®

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.

{cn}®

```
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 constructor() 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.

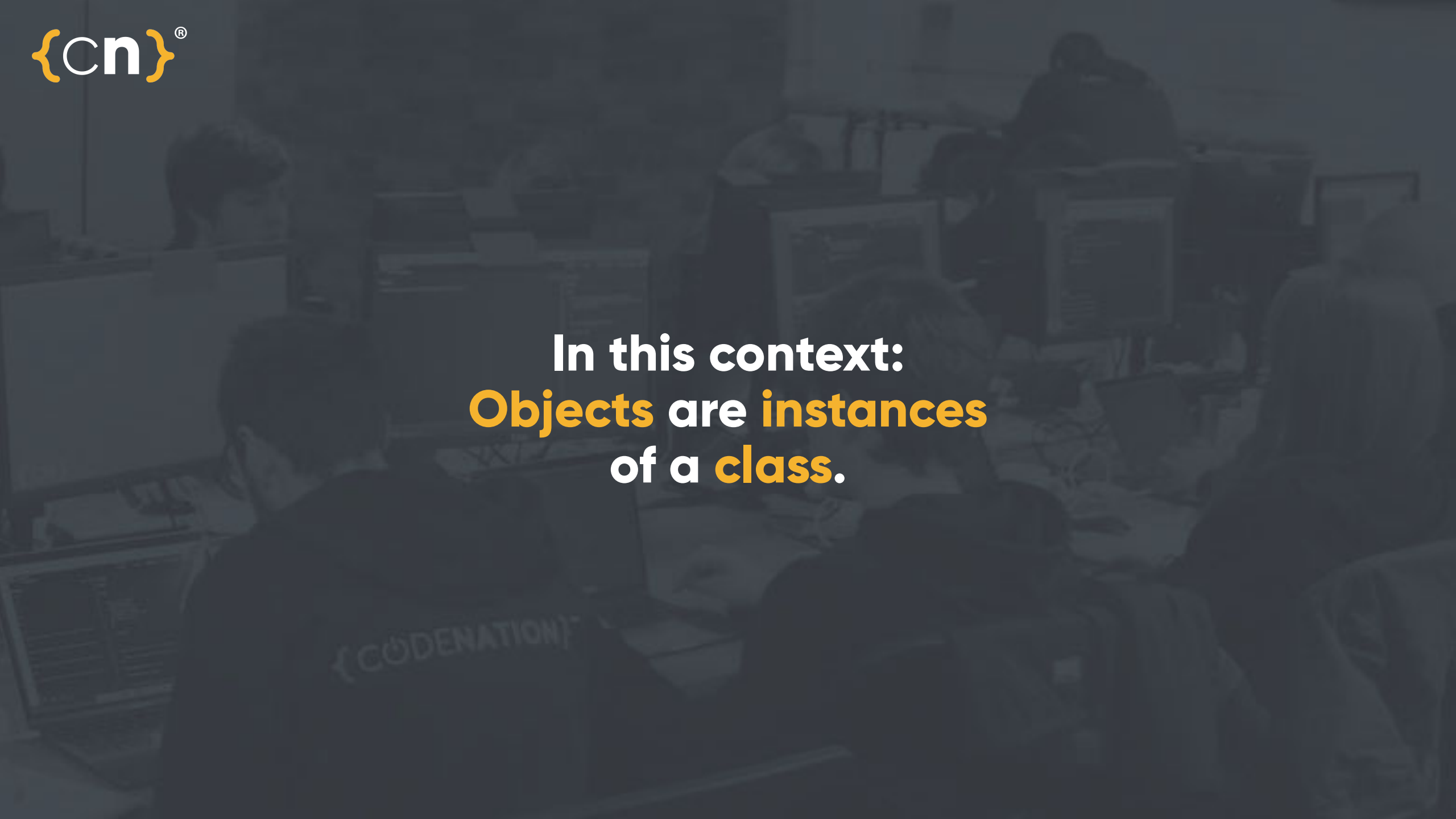
```
class Bunny{
    constructor(name){
        this._name = name;
        this._hops = 0;
    }
    get name(){
        return this._name;
    }
    get hops(){
        return this._hops;
    }
    increaseHops(){
        this._hops++;
    }
}
```

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.


In this context:
Objects are instances
of a 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);
```

{cn}®

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);
```

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".

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.

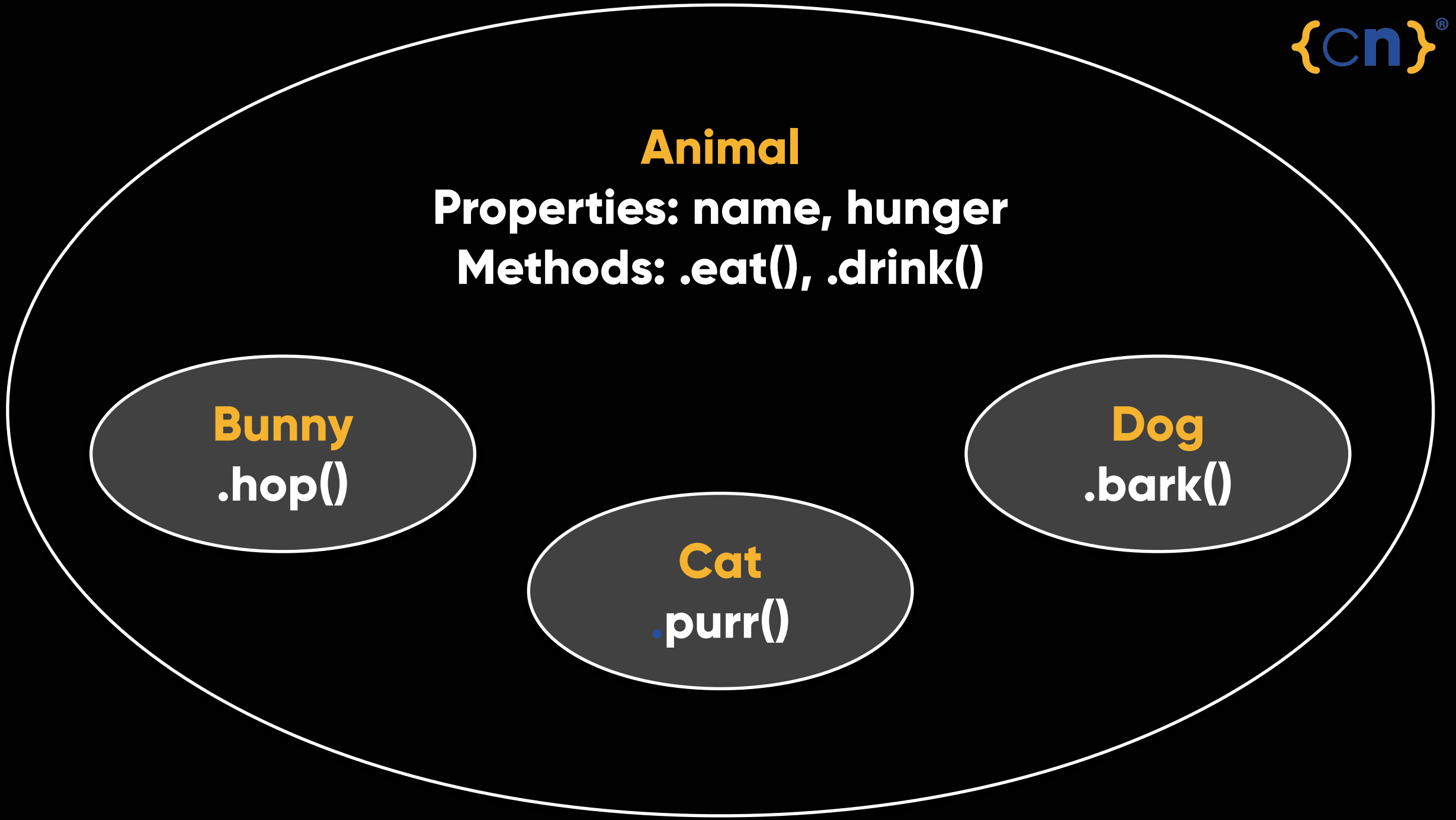
```
class Car{
    constructor(regnum){
        this._regnum = regnum;
        this._hours = 0;
        this._charge = 0.00;
    }
    get regnum(){
        return this._regnum;
    }
    get hours(){
        return this._hours;
    }
    get charge(){
        return this._charge;
    }
    increaseHours(){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

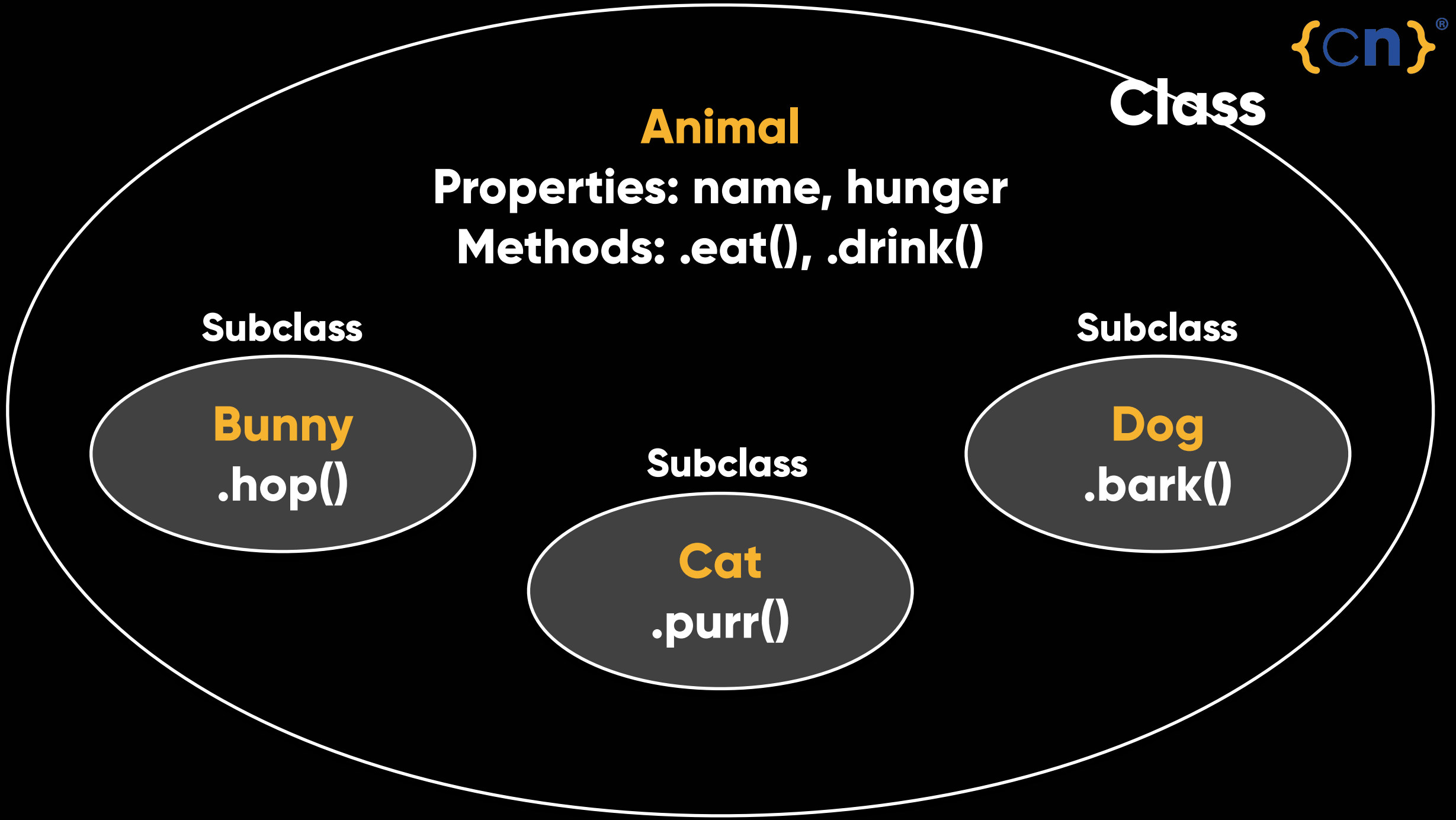

Imagine we now added a new member
of our animal kingdom: a cat


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-;
    }
}
```

{cn}®

In code.


Class
Animal
Properties: name, hunger
Methods: .eat(), .drink()
Subclass
Bunny
.hop()
Subclass
Cat
.purr()
Subclass
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-;
    }
}
class Bunny extends Animal {
    constructor(name, lovesCarrot){
        super(name);
        this._lovesCarrot = lovesCarrot;
    }
    get lovesCarrots(){
        return this._lovesCarrot;
    }
}

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

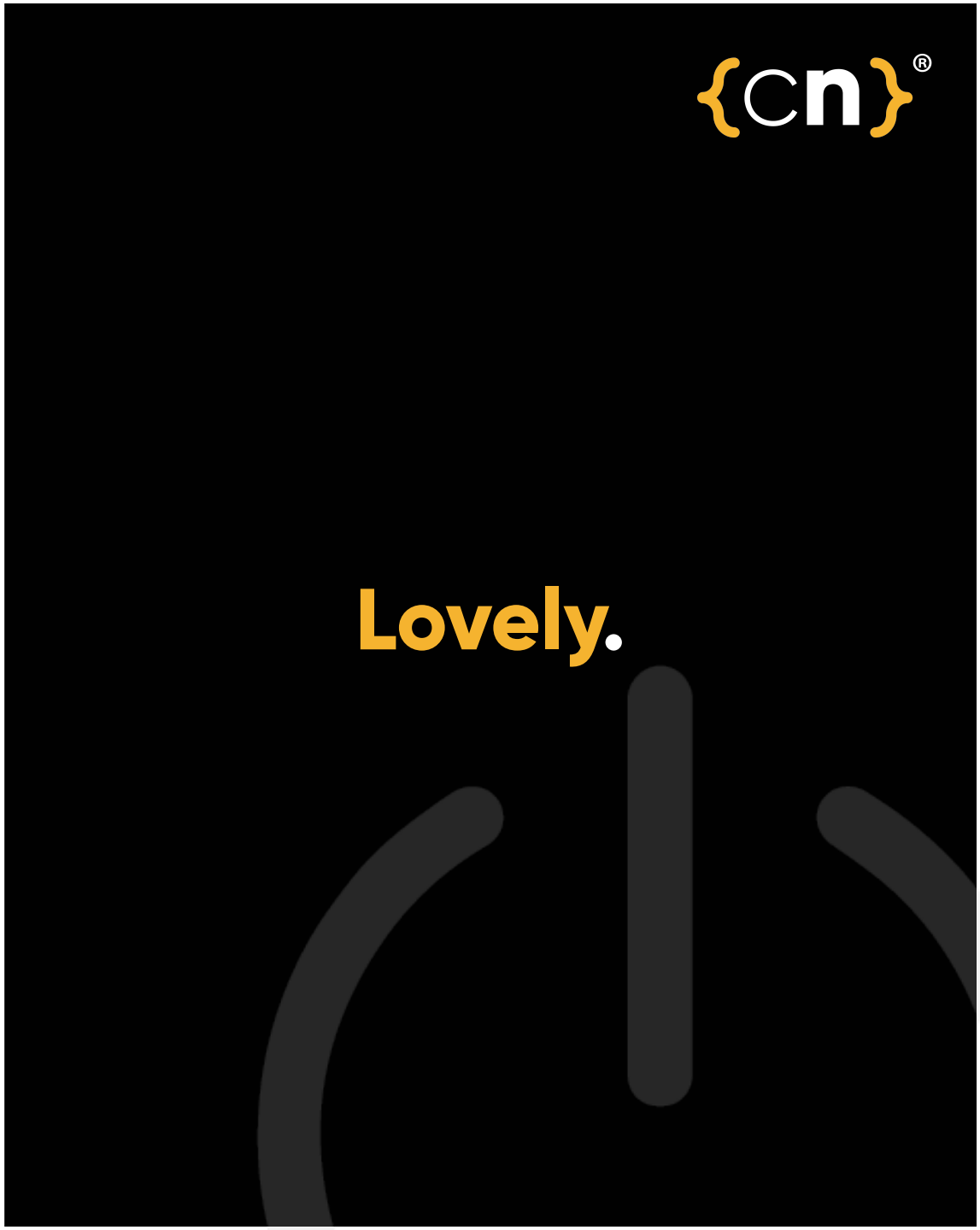
{cn}®

Our **subclasses** are direct copies due to **inheritance**.

OOP fundamental: **reusable** code.


We can also pass arrays
through the 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"]
);
```




Think about how many
lines of code we've
saved.


{cn}®
Inheritance makes our lives easier; code is reusable.
OOP is wonderful.

Learning Objectives

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

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.

Activity (3): DOM!

Convert the Cyber Pet to a DOM project.

Extension: Use part of your DOM dice game code, use the dice to control of your pet!