the Master course

{CODENATION}

- Make 3 variables; one storing a string, one a number and the other a bool

- 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.

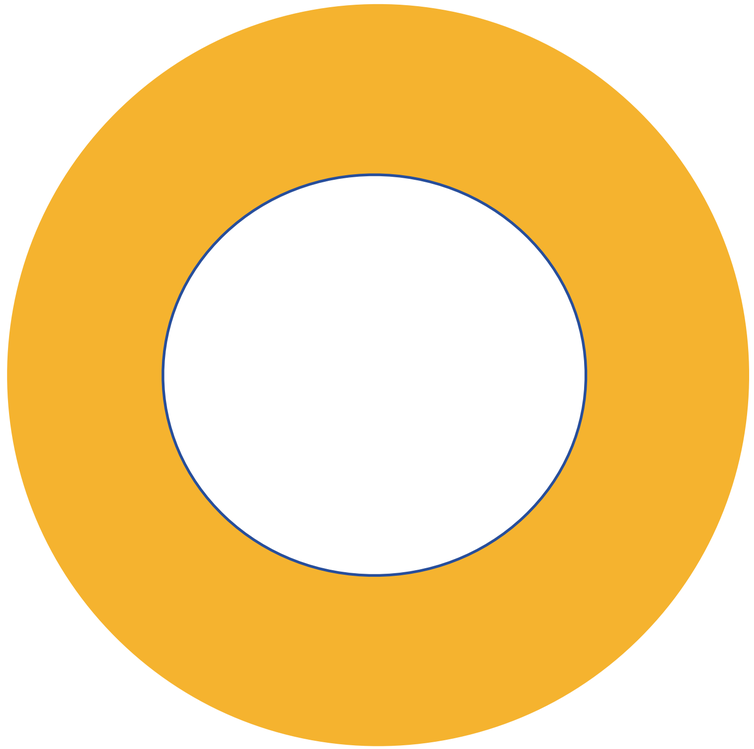
Object-oriented programming

See how that was orange and black? That means it's serious. Not playful. At all.

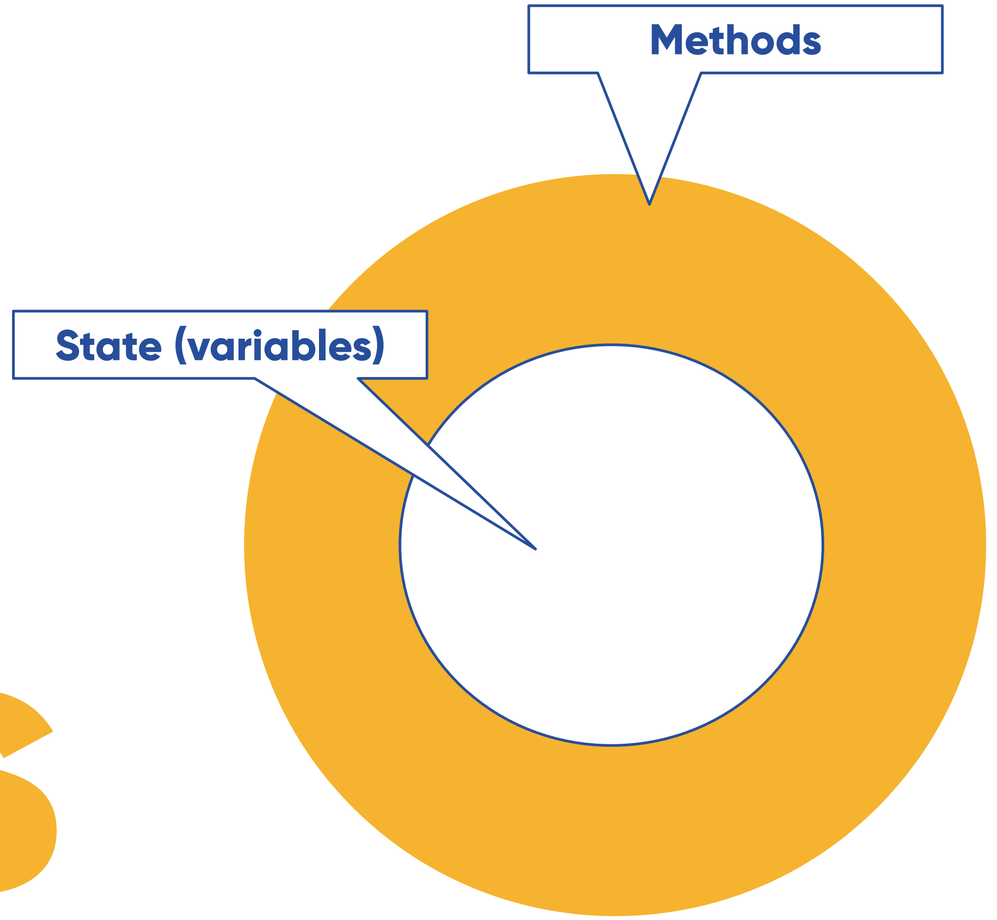
OOP is a fundamental principle of modern development

Its fundamental concepts focus on code reusability using classes and objects

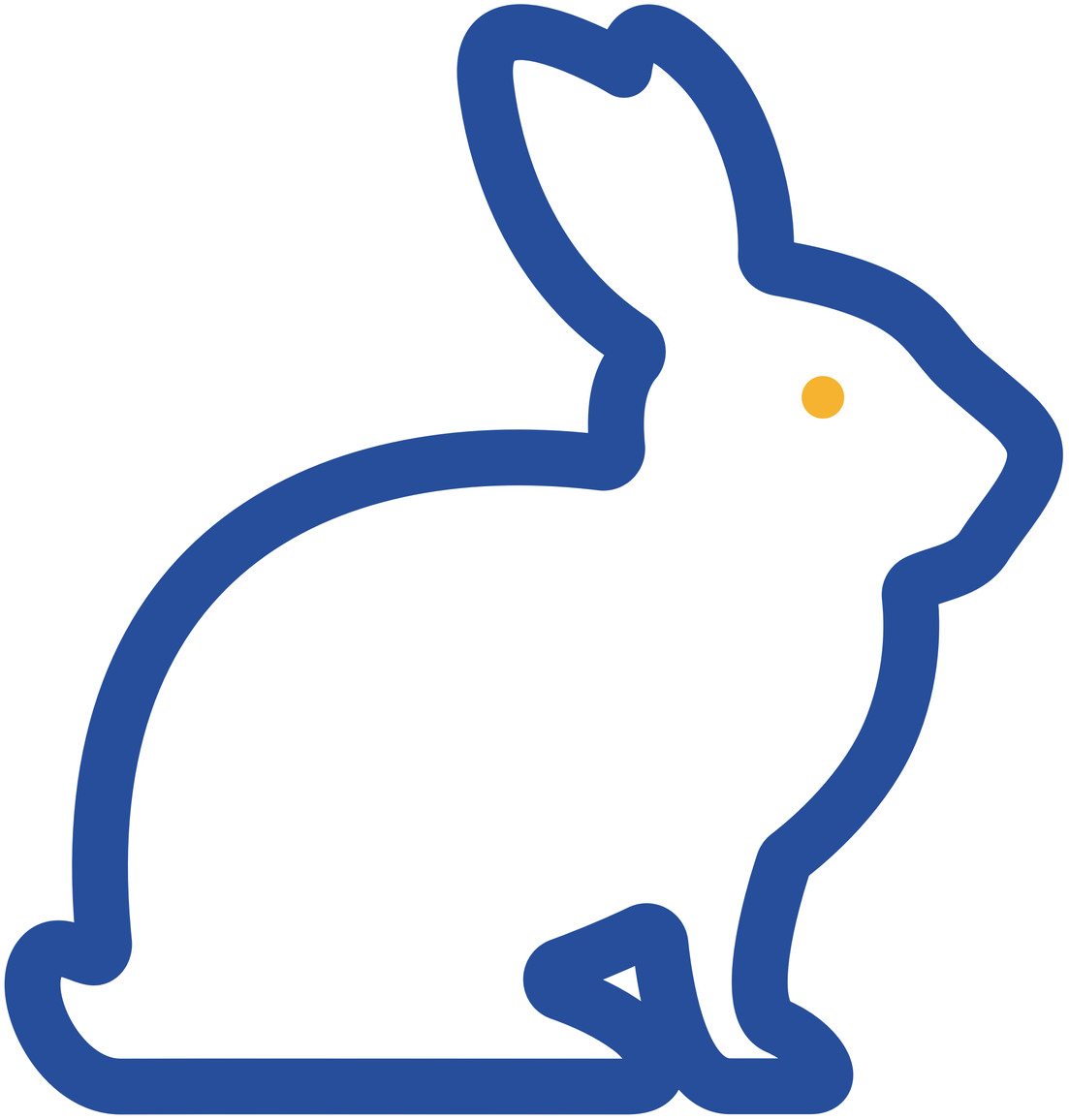
Objects



Objects



Objects



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 Navi. This bunny's [name] (a key) is Navi (a value) and has an age(another key) of 5 (another value). Let's create Navi

```
let Navi = {
  _name: 'Navi',
  _hops: 0,

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

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

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

This is cool but what if we've got loads of bunnies? (The dream.)

```
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 loads of bunny objects

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

We call the constructor() 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;
  }
}

const Navi = new bunny('Navi');
const Sherlock = new bunny('Sherlock')

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

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

We create a new variable named Navi that will store an instance of our bunny class.

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

We pass the 'Navi' string to the bunny constructor, which sets the name property to 'Navi'.

Create a Sherlock object with 0 hops!

```
class bunny {
  constructor(name) {
    this.name = name;
    this.hops = 0;
  }
}

const Navi = new bunny('Navi');

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

Create a Sherlock 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 summat 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++;
  }
}
```

{CN}™

These are known as getters and setters

```
class bunny {
  constructor(name) {
    this._name = name;
    this._hops = 0;
  }

  get name() {
    return this._name;
  }

  get hops() {
    return this._hops;
  }

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

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

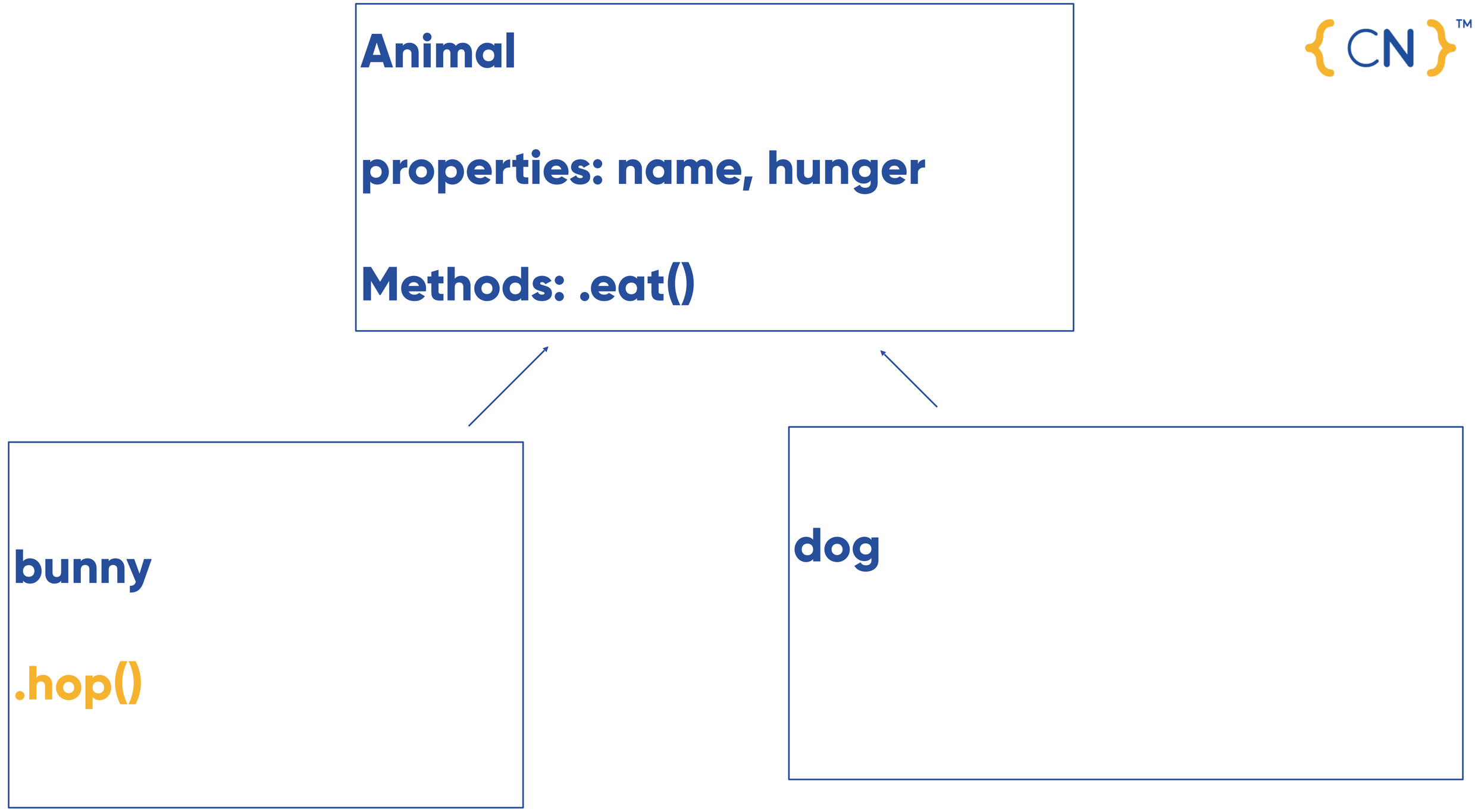
```
class bunny {
  constructor(name) {
    this._name = name;
    this._hops = 0;
  }

  get name() {
    return this._name;
  }

  get hops() {
    return this._hops;
  }

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

const venkman = new bunny('Venkman');
venkman.increaseHops()
```

{CN}™

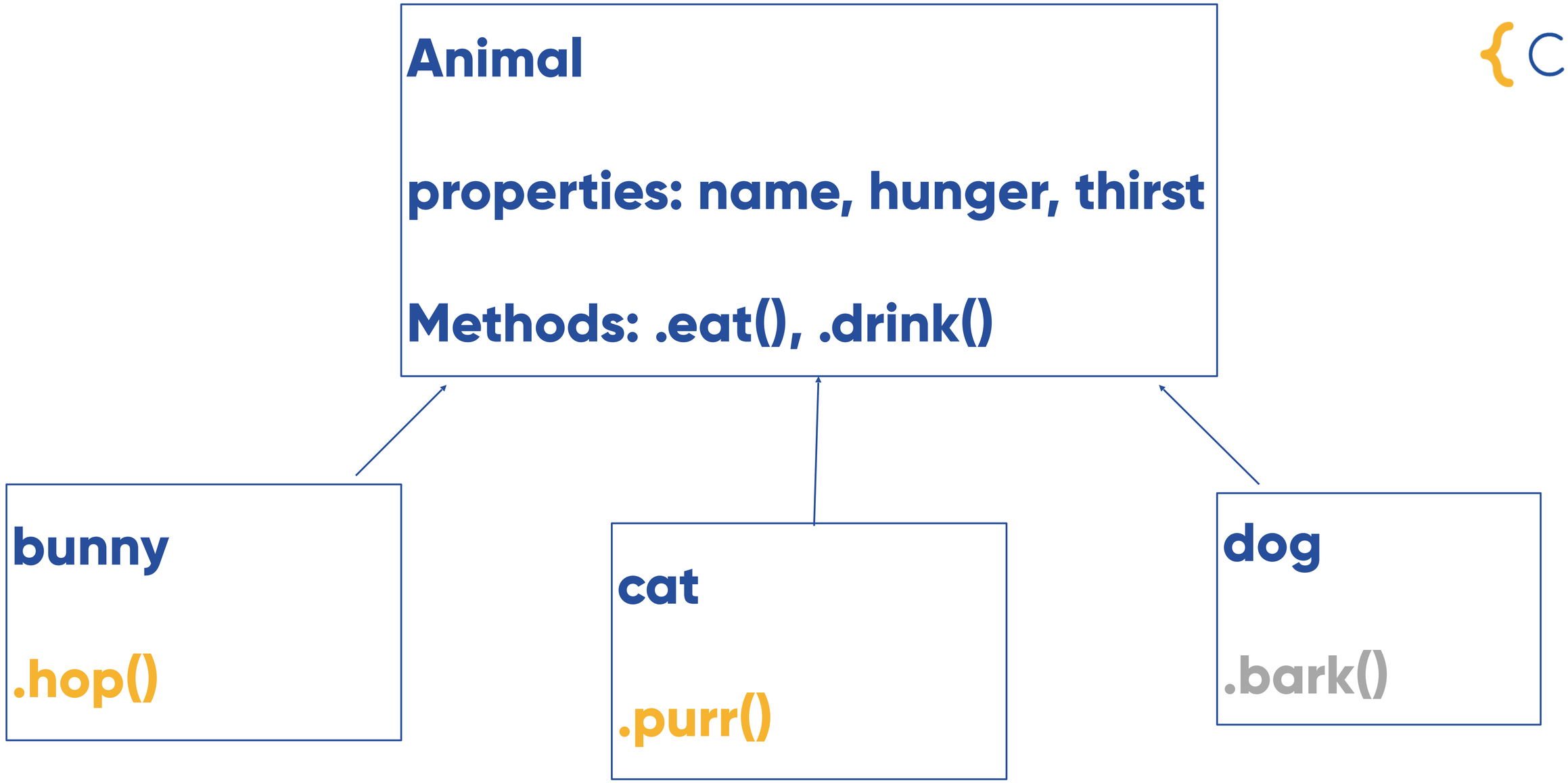
OK. Let's create a class for cars with manufacturer, model and colour constructors and create 3 instances of it. The car should have functions for accelerate, brake, turning and beeping!

OK. Let's create a class for cars with manufacturer, model and colour constructors and create 3 instances of it. The car should have functions for accelerate, brake, turning and beeping!

Inheritance


Animal
properties: name, hunger
Methods: .eat()
bunny
.hop()
dog

Imagine we lost control and added a cat


Animal
properties: name, hunger, thirst
Methods: .eat(), .drink()
bunny
.hop()
cat
.purr()
dog
.bark()

```
class animal {
  constructor(name) {
    this._name = name;
    this._hunger = 60;
  }

  get name() {
    return this._name;
  }

  get hunger() {
    return this._hunger;
  }

  eat() {
    this._hunger--;
  }
}
```

{CN}™

Ooh la la. See how animal now contains name and hunger? Our subclasses, those which inherit from animal, now won't need that coding up again.

```
class animal {
  constructor(name) {
    this._name = name;
    this._hunger = 60;
  }

  get name() {
    return this._name;
  }

  get hunger() {
    return this._hunger;
  }

  eat() {
    this._hunger--;
  }
}
```

```
class bunny extends animal {

constructor(name, lovesCarrots) {

super(name);
this._lovesCarrots = lovesCarrots;}

}

const Stanz = new bunny('Stanz', true)
```

Ooh la la. See how animal now contains name and hunger? 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 = 60;
  }

  get name() {
    return this._name;
  }

  get hunger() {
    return this._hunger;
  }

  eat() {
    this._hunger--;
  }
}
```

```
class bunny extends animal {

constructor(name, lovesCarrots, favouriteHerbs) {

super(name);

this._lovesCarrots= lovesCarrots;
this._favouriteHerbs = favouriteHerbs;
}

get favouriteHerbs () {

return this._favouriteHerbs;

}
}

const Stanz = new bunny('Stanz', true)
const Spengler = new bunny('Spengler',false, ['Basil', 'coriander']);
```

Ooh la la. See how animal now contains name and hunger? 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

Project!

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.