Notes: 1.3 - Module 3 – Prototypes and Inheritance - (6/16/2021)

4 Principles of 'this'

Window Binding

If none of the other rules apply 'this' defaults to the window, global object in node or undefined in strict mode. - This happens when we do not give 'this' keyword any context.

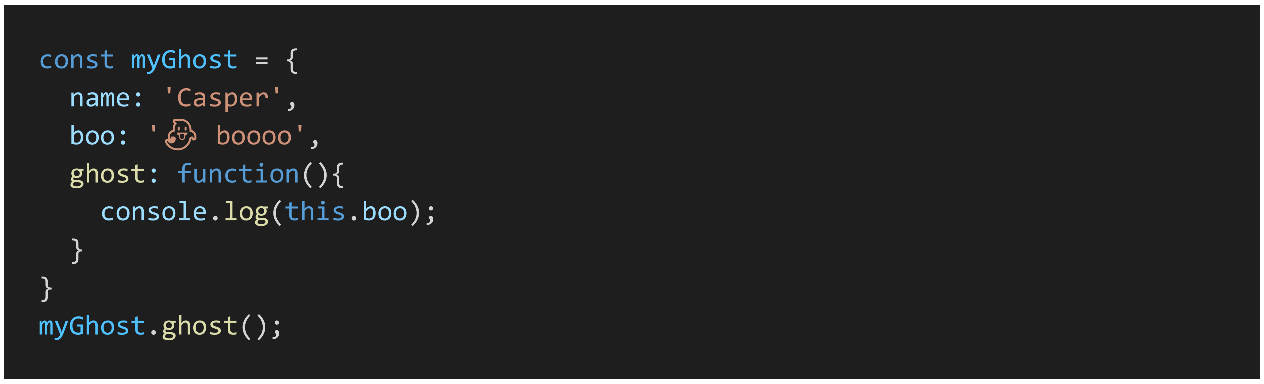
Strict mode forces us to write clean code, with more errors if we do not.

```
'use strict';
  dog = 'Ada';
    // This will return as not defined because
       it's missing the variable declaration
  console.log(dog);
```

```
'use strict';
  function ghost(){
    console.log(this.boo);
  }

  const boo = '👻 boooo';

  ghost(); // returns undefined because 'this' has no context
```

Implicit Binding

- Most common rule - found in 80% of use cases
- It applies to objects with methods (Functions that belong to an object)
- When the function is invoked, look to the left of the dot, that's what 'this' refers to. (Only applies to objects with methods)



```
const myGhost = {
  name: 'Casper',
  boo: '👻 boooo',
  ghost: function(){
    console.log(this.boo);
  }
}
myGhost.ghost();
```

We are invoking the function here with myGhost.ghost(); , if we look to the left of the dot, we see my ghost. So, we are assigning the 'this' keyword to myGhost, therefore we get back myGhosts boo.

Example: Let’s create some pets

```
const petOne = {
  // properties / values
  name: 'Ada',
  species: 'Bali dog',
  pronoun: 'her',
  favFood: 'salmon',
  eat: function(){
    return `${this.name} is a ${this.species} and ${this.pronoun}
               favorite food is ${this.favFood}`;
  }
}

const petTwo = {
  // properties / values
  name: 'Egg',
  species: 'Border Collie',
  pronoun: 'his',
  favFood: 'carrots',
  eat: function(){
    return `${this.name} is a ${this.species} and ${this.pronoun}
               favorite food is ${this.favFood}`;
  }
}

const petThree = {
  // properties / values
  name: 'Frost',
  species: 'cat',
  pronoun: 'his',
  favFood: 'cheese',
  eat: function(){
    return `${this.name} is a ${this.species} and ${this.pronoun}
               favourite food is ${this.favFood}`;
  }
}                                                    // Continued next page
```

```javascript
const petFour = {
  // properties / values
  name: 'Lola',
  species: 'dog',
  pronoun: 'her',
  favFood: 'peanut butter',
  eat: function(){
    return `${this.name} is a ${this.species} and ${this.pronoun}
               favourite food is ${this.favFood}`;
  }
}

const petFive = {
  // properties / values
  name: 'Silver',
  species: 'Blue Sapphire Chicken',
  pronoun: 'her',
  favFood: 'worms',
  eat: function(){
    return `${this.name} is a ${this.species} and ${this.pronoun}
               favourite food is ${this.favFood}`;
  }
}

console.log(petOne.eat());
console.log(petTwo.eat());
console.log(petThree.eat());
console.log(petFour.eat());
console.log(petFive.eat());
```

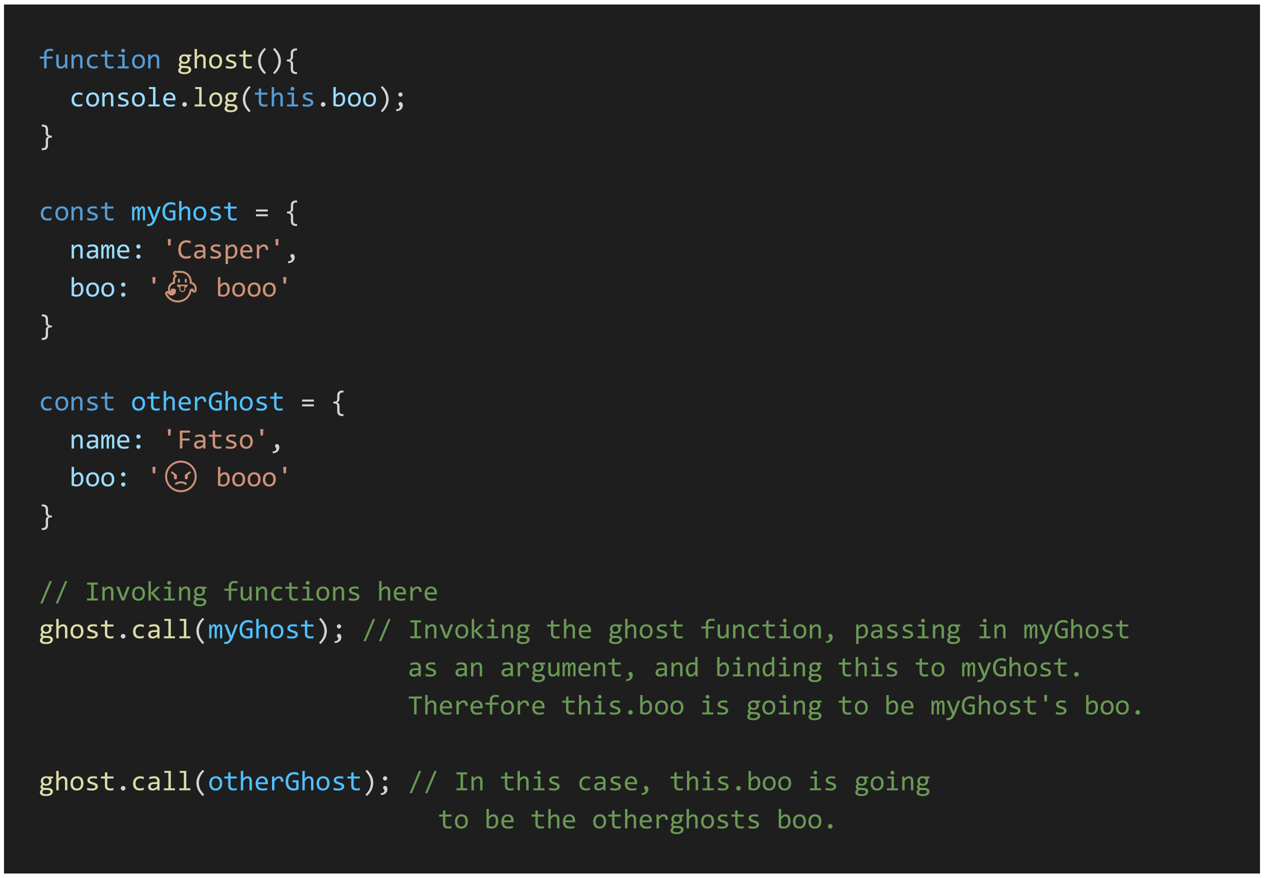
Explicit Binding

- **call** - immediately invokes the function, we pass in arguments one by one.

- **apply** - immediately invokes the function, we pass in arguments as an array.

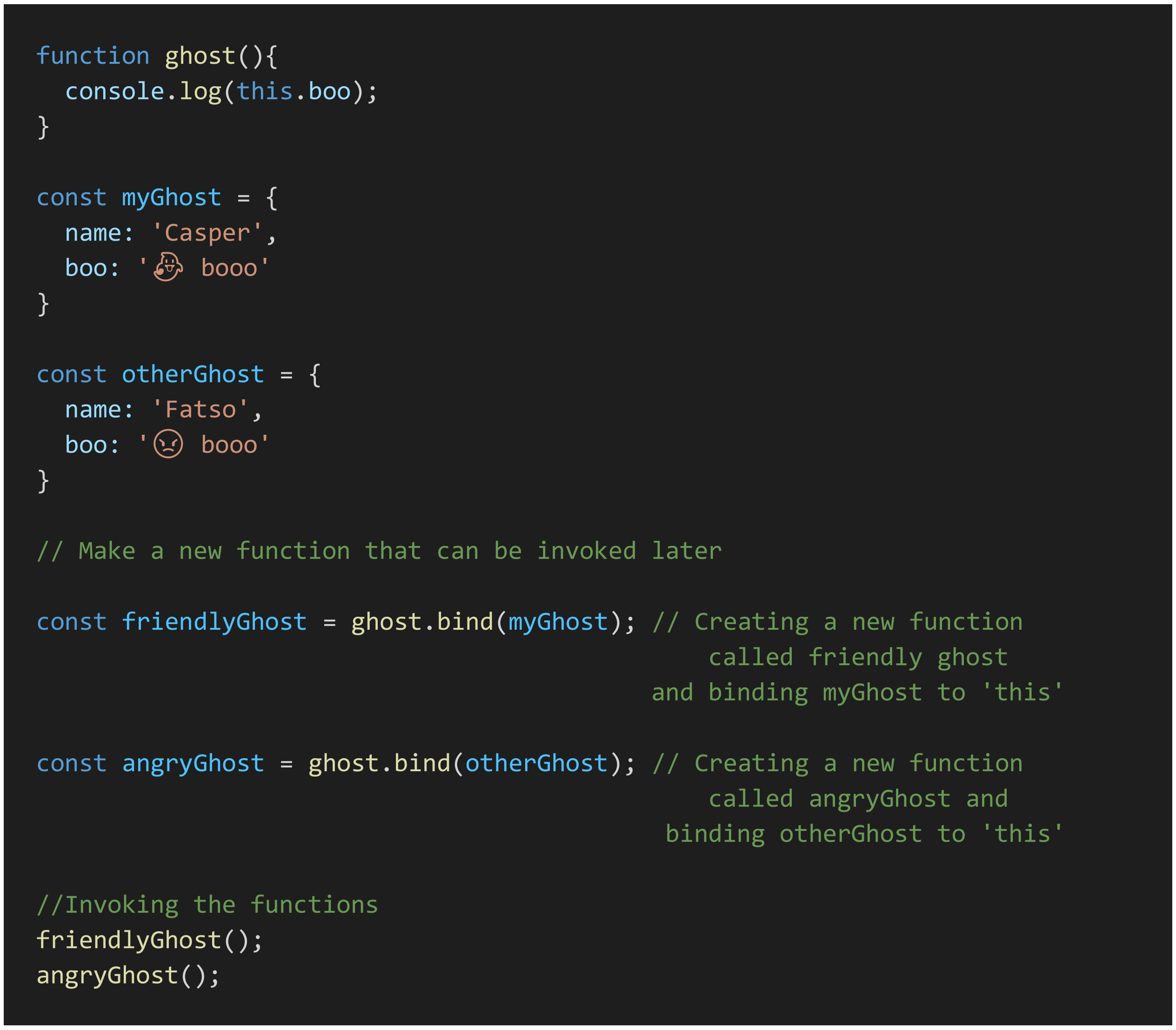
- **bind** - Does not immediately invoke the function, instead it returns a brand-new function that can be invoked later, we pass in arguments one by one.

Call



```
function ghost(){
  console.log(this.boo);
}

const myGhost = {
  name: 'Casper',
  boo: '👻 booo'
}

const otherGhost = {
  name: 'Fatso',
  boo: '😠 booo'
}

// Invoking functions here
ghost.call(myGhost); // Invoking the ghost function, passing in myGhost
                        as an argument, and binding this to myGhost.
                        Therefore this.boo is going to be myGhost's boo.

ghost.call(otherGhost); // In this case, this.boo is going
                          to be the otherghosts boo.
```

Bind



```javascript
function ghost(){
  console.log(this.boo);
}

const myGhost = {
  name: 'Casper',
  boo: '👻 booo'
}

const otherGhost = {
  name: 'Fatso',
  boo: '😠 booo'
}

// Make a new function that can be invoked later

const friendlyGhost = ghost.bind(myGhost); // Creating a new function
                                               called friendly ghost
                                           and binding myGhost to 'this'

const angryGhost = ghost.bind(otherGhost); // Creating a new function
                                               called angryGhost and
                                            binding otherGhost to 'this'

//Invoking the functions
friendlyGhost();
angryGhost();
```

BREAK OUT

My attempt

```javascript
function animal(){
  console.log(this.rawr);
}

const myCat = {
  type: 'Cat',
  rawr: 'mew mew mew'
}

const myTiger = {
  type: 'Tiger',
  rawr: 'ROAR!'
}

const lilCat = animal.bind(myCat);
const bigCat = animal.bind(myTiger);

lilCat();
bigCat();
```

Class example

```javascript
function callOfTheWild(){
  return `${this.name} says ${this.sound}`;
}

const animal = {
  name: 'Larry',
  species: 'Llama',
  sound: 'bahhhhhhhhhh'
}

console.log(callOfTheWild.call(animal));
```

New Binding

- **Using the new keyword constructs a new object and 'this' points to it**
- **When a function is invoked as a constructor function 'this' points to the newly created object**

Initializing with arguments (today's homework)

```javascript
function Ghost(saying){
  this.saying = saying;
}

// Creating an object (initializing with an argument)

const myGhost = new Ghost('Casper the friendly 👻');


console.log(myGhost.saying);
console.log(myGhost);
```

Initialize as an object

(this could be on the sprint challenge maybe? 🤷‍♀️)

```javascript
function Ghost(attr){
  this.saying = attr.saying;
}
      // Create our object (initalizing as an object)
const myGhost = new Ghost({
  saying: 'Casper the friendly 👻'
});

console.log(myGhost.saying);
console.log(myGhost);
```

Constructor functions and prototypes

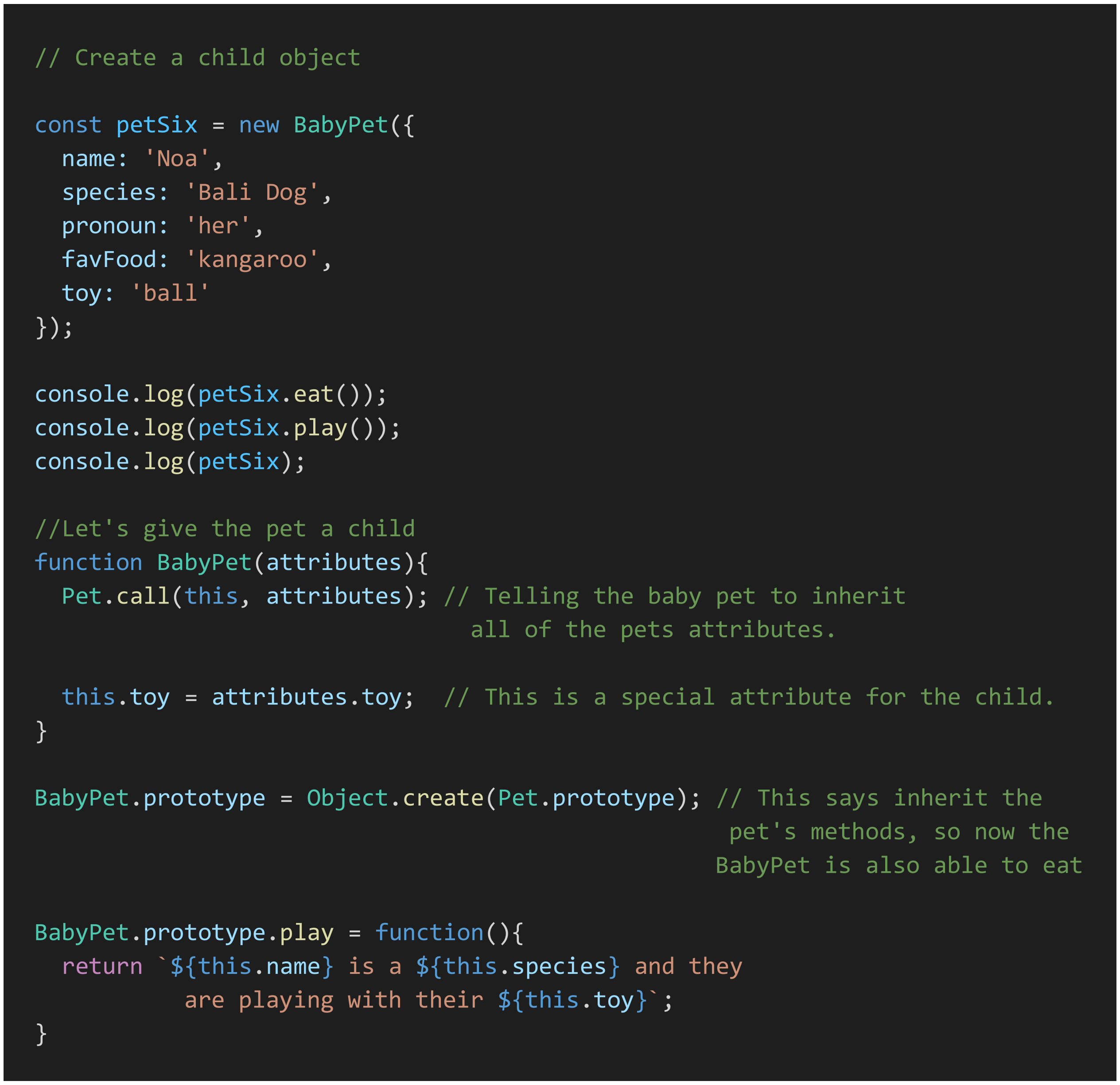
- **Constructor functions construct other objects - that is the whole purpose.**
- **Think of it as a template for an object.**
- **Capitalized function name.**
- **It has an assignment of the 'this' keyword.**
- **It is likely missing a return statement (not a guarantee).**

Initializing with an object

```
function Pet(attributes){
  this.name = attributes.name;
  this.species = attributes.species;
  this.pronoun = attributes.pronoun;
  this.favFood = attributes.favFood
}

Pet.prototype.eat =  function(){
    return `${this.name} is a ${this.species} and ${this.pronoun}
                 favourite food is ${this.favFood}`;
}

const petOne = new Pet({
  // properties / values
  name: 'Ada',
  species: 'Bali dog',
  pronoun: 'her',
  favFood: 'salmon',
});

const petTwo = new Pet({
  // properties / values
  name: 'Egg',
  species: 'Border Collie',
  pronoun: 'his',
  favFood: 'carrots',
});
                                               // Continued next page
```

```
const petThree = new Pet({
  // properties / values
  name: 'Frost',
  species: 'cat',
  pronoun: 'his',
  favFood: 'cheese',
});

const petFour = new Pet({
  // properties / values
  name: 'Lola',
  species: 'dog',
  pronoun: 'her',
  favFood: 'peanut butter'
});

const petFive = new Pet({
  // properties / values
  name: 'Silver',
  species: 'Blue Sapphire Chicken',
  pronoun: 'her',
  favFood: 'worms',
});

console.log(petOne);
console.log(petOne.eat());
console.log(petTwo.eat());
console.log(petThree.eat());
console.log(petFour.eat());
console.log(petFive.eat());
```

// Continued next page



```javascript
// Create a child object

const petSix = new BabyPet({
  name: 'Noa',
  species: 'Bali Dog',
  pronoun: 'her',
  favFood: 'kangaroo',
  toy: 'ball'
});

console.log(petSix.eat());
console.log(petSix.play());
console.log(petSix);

//Let's give the pet a child
function BabyPet(attributes){
  Pet.call(this, attributes); // Telling the baby pet to inherit
                                 all of the pets attributes.

  this.toy = attributes.toy;  // This is a special attribute for the child.
}

BabyPet.prototype = Object.create(Pet.prototype); // This says inherit the
                                                   pet's methods, so now the
                                                  BabyPet is also able to eat

BabyPet.prototype.play = function(){
  return `${this.name} is a ${this.species} and they
           are playing with their ${this.toy}`;
}
```

Initializing with arguments

```
function Pet(name, species, pronoun, favFood){
  this.name = name;
  this.species = species;
  this.pronoun = pronoun;
  this.favFood = favFood;
}

Pet.prototype.eat = function(){
  return `${this.name} is a ${this.species} and ${this.pronoun}
           favourite food is ${this.favFood}`;
}

function BabyPet(name, species, pronoun, favFood, toy){
  Pet.call(this, name, species, pronoun, favFood); // Inheriting from Pet
  this.toy = toy; // Assigning specific attribute for the baby
}

// Inheriting the parent's methods
BabyPet.prototype = Object.create(Pet.prototype);

// Creating a "play" prototype

BabyPet.prototype.play = function(){
  return `${this.name} is a ${this.species} and they
             are playing with their ${this.toy}`;
}

console.log(petOne.eat());
console.log(petOne);

// Creating my new object with arguments
const petOne = new Pet('Ada', 'Bali dog', 'her', 'salmon');

// Create the new baby pet
const petSix = new BabyPet('Noa', 'Bali dog', 'her', 'kangaroo', 'ball');


console.log(petSix.eat());
console.log(petSix.play());
console.log(petSix);
```