Before we begin...

- Open up these slides:
  - https://goo.gl/TdXMjM

This, Factories & Constructors

Learning Objectives

- **Understand** and **explain** JavaScript context
- **Understand** the *this* keyword and its patterns
- **Understand** prototypal inheritance and its purpose
- **Distinguish** the difference between prototypal from classical inheritance
- **Understand** how the prototype chain works in JavaScript
- **Create** and **use** Factories and Constructors

Agenda

- *this*
- Prototypes
- Factories
- Constructors

A quick review

- Review
- Templating
- Functional array methods
- Lab

this

What is *this*?

- One of the most confusing mechanisms in JavaScript
- A special identifier that's automatically defined for us (an always available variable)
    - Kind of, sort of, almost, only readable. You can't change its value in the same way as you normally would
- It bedevils even senior JavaScript developers
- It can seem downright magical but it aims to represent the ***current context***

Let's get *this* over with

Ugh...

- That *this* is the wrong *this*
- I don't get this *this*
- "Sometimes when I'm writing Javascript I want to throw up my hands and say "this is bullshit!" but I can never remember what "*this*" refers to" - Ben Halpern
- *"*JavaScript makes me want to flip the table and say 'F*** *this* shit', but I can never be sure what '*this*' refers to" - @oscherler
- WHAT IS *THIS*
- Plus many, many more

So, how does *this* work?

- It all comes back to the call-site
- To understand how the ***this*** keyword works, we need to know exactly where and how the function was called (and by who)
    - There are more ways than we have seen so far!
- Every* function, when it is running, has access to its current execution context

Why does *this* exist?

- So we can:
  - Reuse functions with different contexts
  - Change the focus of our code
  - Make methods more dynamic
  - We don't always know what we are talking about!
    - e.g. Maybe we have a function creating objects for us, or maybe we don't know which element is being interacted with

The call-site

Knowing that *this* represents the **context** of whatever code is running, there are five main ways of it being automatically defined for us:

1. Global Binding (*window*)
2. Event Binding
3. Implicit Binding
4. Explicit Binding
5. *new* Binding

Global Binding

```
console.log( this );

function checkThisOut() {
    console.log( this );
}

checkThisOut();
```

This is the default binding!

It refers to the window object

Event Binding

```
var img = document.querySelector("img");

function onImageClick() {
    console.log( this );
}

img.addEventListener("click", onImageClick);
```

When you run an event listener, the *this* keyword refers to whatever was interacted with

In the above case, it is the image DOM node!

Implicit Binding

```
var person = {
    name: "Groucho",
    speak: function () {
        console.log( this, this.name );
    }
};

person.speak();
```

When you run a method, the this keyword will refer to the containing object

In this case, the person object!

Explicit Binding

```
function sayHello() {
    console.log( "Hello, " + this.name );
}

var person = { name: "Zeppo" };

sayHello.call( person );
```

When you use **.call**, the *this* keyword refers to the parameter you provide

In this case, the person object!

Explicit Binding

```
function sayHello() {
    console.log( "Hello, " + this.name );
}

var person = { name: "Zeppo" };

sayHello.apply( person );
```

When you use **.apply**, the *this* keyword refers to the parameter you provide

In this case, the person object!

Explicit Binding

```
function sayHello() {
    console.log( "Hello, " + this.name );
}

var person = { name: "Zeppo" };

var personsHello = sayHello.bind( person );
personsHello();
```

When you use **.bind**, the *this* keyword refers to the parameter you provide

In this case, the person object!

@threequal

new Binding

```
var Person = function (name) {
    this.name = name;
    console.log( this );
    // => { name: "Roger" }
};

var serge = new Person( "Serge" );
```

When you use **new**, the *this* keyword refers to a new empty object that you can add properties to

It is also implicitly returned (automatically returned)

@threequal

Determining *this*

The order of precedence:

1. Is the function called with the **new** keyword?
2. Is the function called with **.call**, **.apply** or **.bind**?
3. Is the function called on an object (is it a method)?
4. Otherwise, it is the default binding - the global object*

Resources

- Tyler McGinniss: WTF is this?
- Todd Motto: this
- MDN: this
- Kyle Simpson: this and Object Prototypes
- JavaScript is Sexy: this
- Rachel Ralston: this
- Quirks Mode: this

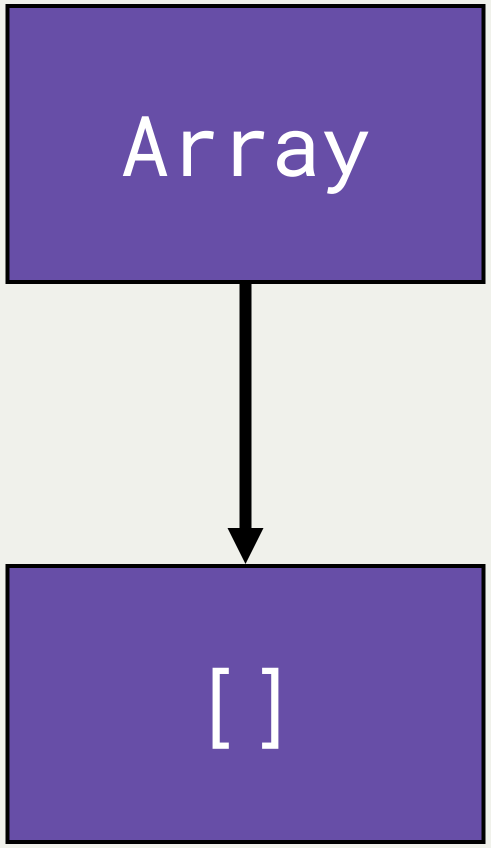
Prototypes & Inheritance

Prototypes & Inheritance

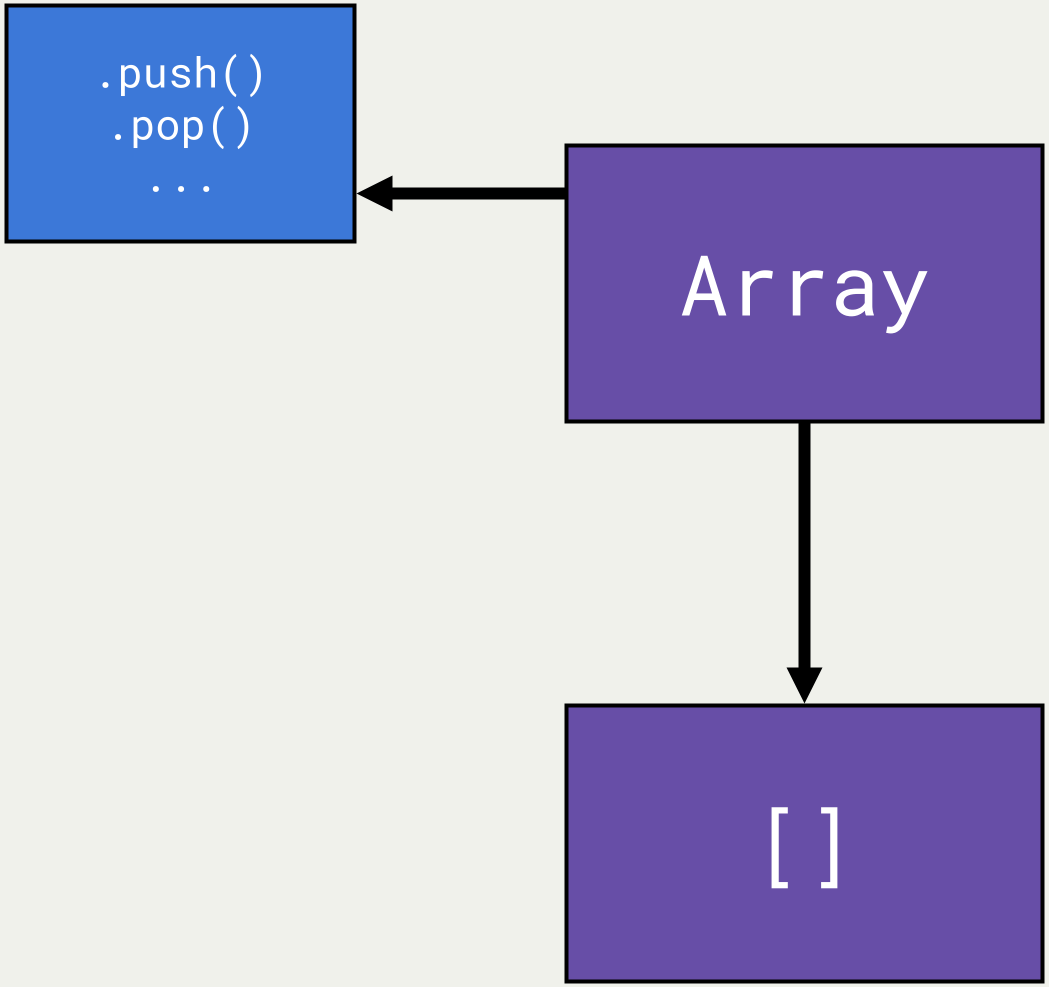
When you create a piece of data, that data will be linked to a parent *thing.* Really, it becomes an instance and is linked to a prototype

- The Prototype Chain is the link
- We can delegate behaviour through this

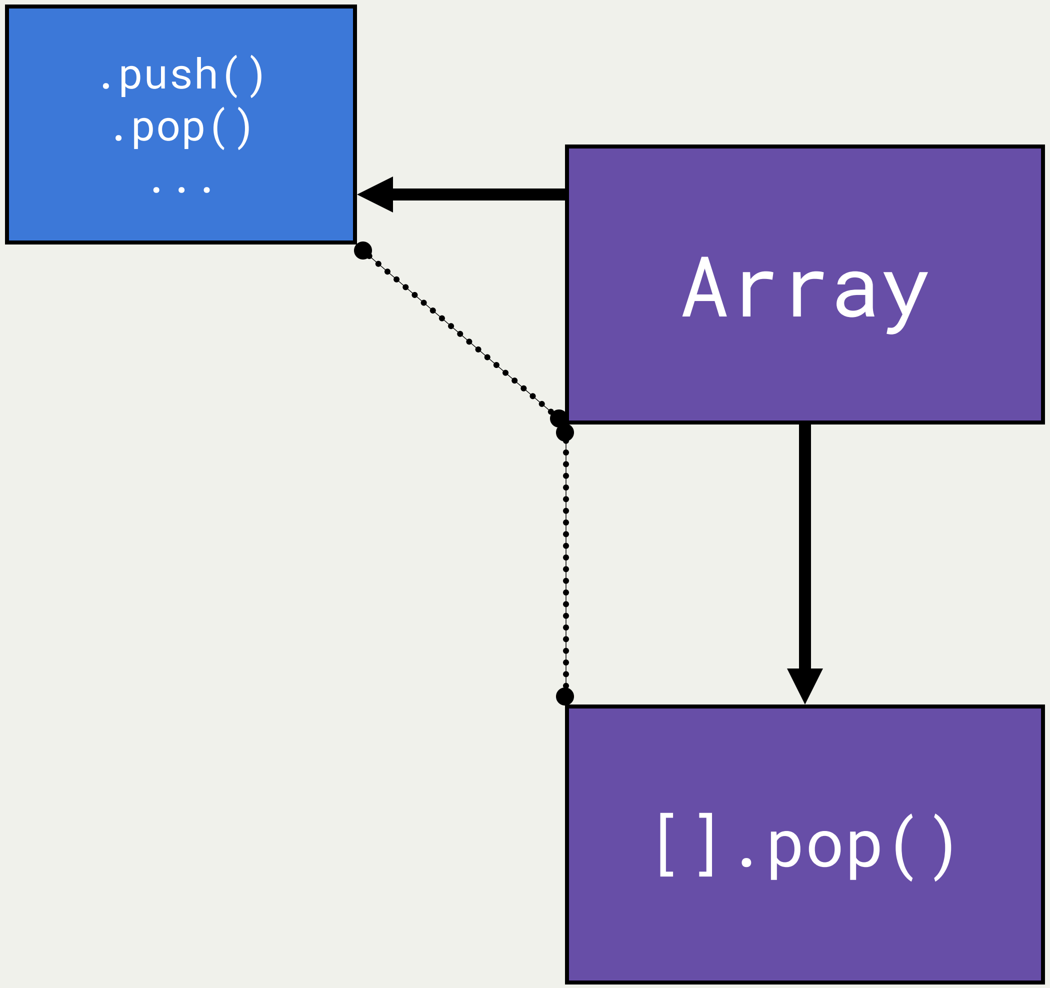
Essentially, the prototype is a blueprint and then we use it to create something



@threequal



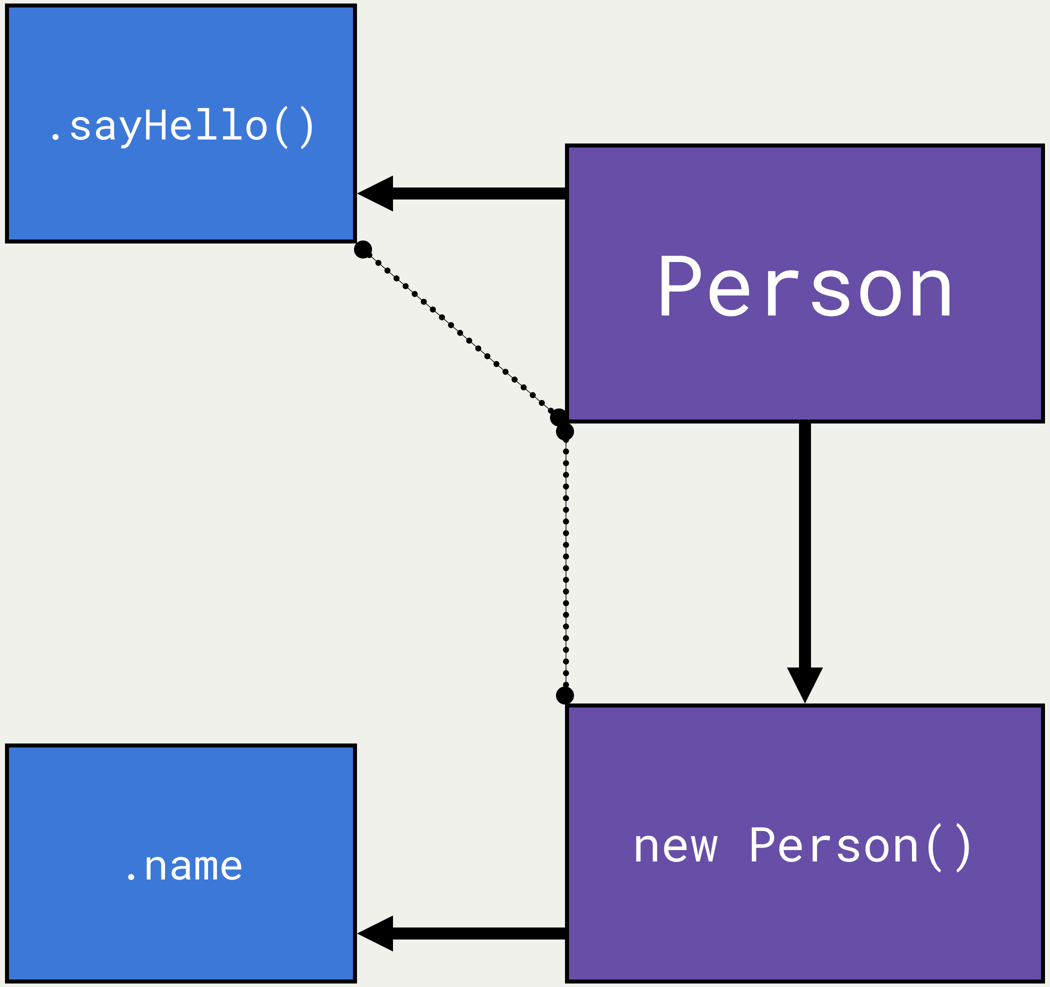
@threequal



@threequal

We can create our own!

- We can create a constructor
  - We can add generic properties and methods to the prototype
- We can create instances
  - We can add specific properties and methods to the instance
- We can also extend this prototype chain to other constructors


.sayHello()
Person
new Person()
.name

Factories & Constructors

What are they? Both:

- Are JavaScript patterns
- Help us easily create objects
- Help us share functionality
    - That makes our code really reusable
- Give us consistency in our data
- Can give us inheritance
- Can help us have private data

Factories

GENERAL ASSEMBLY

Factory Pattern

```
var DogFactory = function (name, breed) {
    var dog = {};
    dog.name = name;
    dog.breed = breed;
    return dog;
};

var tamaskan = DogFactory("Tammy", "Tamaskan");
var buddy = DogFactory("Buddy", "Labrador");
```

Factory + Inheritance

```
var AnimalFactory = function () {
    var animal = {};
    animal.isAlive = true;
    return animal;
};

var DogFactory = function (name, breed) {
    var dog = AnimalFactory();
    dog.name = name;
    dog.breed = breed;
    return dog;
};

var tamaskan = DogFactory("Tammy", "Tamaskan");
```

Exercise

Create a Factory for a Book

Bonus One

Create an AudioBook Factory that inherits from Book

Bonus Two

Add a user interface to this! So you can create Books easily

Resources

- ATEN Design
- Ilya Kantor's Version

Constructors

Constructor Pattern

```
function Person() {}

var serge = new Person();

serge instanceof Person; // => true
```

Constructor Pattern

```
function Person(name) {
  this.name = name;
}

var serge = new Person("Serge");
```

Constructor Pattern

```
function Person(name) {
  this.name = name;
}

Person.prototype.sayHello = function() {
  console.log(`Hello, I am ${this.name}`);
};

var serge = new Person("Serge");
serge.sayHello();
```

Constructor Pattern

```
var Dog = function ( name, breed ) {
    this.name = name;
    this.breed = breed;
    this.bark = function () {
        console.log( "Woof!" );
    }
};

var tamaskan = new Dog( "Tammy", "Tamaskan" );
var buddy = new Dog( "Buddy", "Labrador" );
```

Constructor + Privacy

```
function User(username, email, password) {
  this.username = username;
  this.email = email;
  this.authenticate = function(providedPassword) {
    return password === providedPassword;
  };
}

var serge = new User("serge", "serge@ga.co", "chicken");
```

Constructor + Inheritance

```
function Car(brand) {
    this.brand = brand;
}

function ElectricCar(brand) {
    Car.call(this, brand);
    this.electric = true;
}

ElectricCar.prototype = Object.create(Car.prototype);

var tesla = new ElectricCar("Tesla");
```

Constructor + Inheritance

```
function Animal(breed) {
  this.breed = breed;
};
Animal.prototype.beBorn = function() {
  this.alive = true;
  console.log(`A ${this.breed} is born!`);
};

function Dog(name, breed) {
  Animal.call(this, breed);
  this.name = name;
};
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.bark = function() {
  console.log(`Woof, woof! Says ${this.name}`);
};

var tammy = new Dog("Tammy", "Tamaskan");
```

Exercise

Do the same thing as the Factory exercises,
but with constructors instead (same bonuses)

Pick one:

```
User -> Admin
WaterVehicle -> Boat
Employee -> FullTimeEmployee
Character -> Enemy
Shape -> Circle
```

Resources

- Toby Ho
- Phrogz
- CSS Tricks

Homework

- Continue working on previous homework
  - e.g. Dancing Cats, Interactive Glossary
- Read through this & Object Prototypes by Kyle Simpson (great review for tonight)
- Go through The Modern JavaScript Tutorial
- Read Eloquent JavaScript
- Read Speaking JavaScript

What's next?

- Dealing with Asynchronous code in JS
  - Higher-Order Functions
  - Callbacks
  - JavaScript
- Then, AJAX and APIs!

Questions?

Feedback

https://ga.co/js05syd

Thanks!