

1 - Global Context

When 'this' is not inside of a declared object

```
console.log(this) // window

function whatIsThis(){
  return this
}

function variablesInThis(){
  // since the value of this is the window
  // all we are doing here is creating a global variable
  this.person = "Elie"
}

console.log(person) // Elie

whatIsThis() // window
```

this: default windows(browser)

Nested Objects

What happens when we have a nested object?

```
var person = {
  firstName: "Colt",
  sayHi: function(){
    return "Hi " + this.firstName;
  },
  determineContext: function(){
    return this === person;
  },
  dog: {
    sayHello: function(){
      return "Hello " + this.firstName;
    },
    determineContext: function(){
      return this === person;
    }
  }
}

person.sayHi() // "Hi Colt"
person.determineContext() // true

// but what is the value of the keyword this right now?
person.dog.sayHello() // "Hello undefined"
person.dog.determineContext() // false
```

this assign to person

this assign to dog

Fixing Up With Call

```
var person = {
  firstName: "Colt",
  sayHi: function(){
    return "Hi " + this.firstName
  },
  determineContext: function(){
    return this === person
  },
  dog: {
    sayHello: function(){
      return "Hello " + this.firstName
    },
    determineContext: function(){
      return this === person
    }
  }
}
```

```
person.sayHi() // "Hi Colt"
person.determineContext() // true

person.dog.sayHello.call(person) // "Hello Colt"
person.dog.determineContext.call(person) // true

// Using call worked! Notice that we do NOT invoke sayHello or determineContext
```

Using Call in the Wild

Let's examine a very common use case

```
var colt = {
  firstName: "Colt",
  sayHi: function(){
    return "Hi " + this.firstName
  }
}

var elie = {
  firstName: "Elie",
  // Look at all this duplication :(
  sayHi: function(){
    return "Hi " + this.firstName
  }
}

colt.sayHi() // Hi Colt
elie.sayHi() // Hi Elie (but we had to copy and paste the function from above...)

// How can we refactor the duplication using call?

// How can we "borrow" the sayHi function from colt
// and set the value of 'this' to be elie?
```

What about Apply?

It's almost identical to call - except the parameters!

```
var colt = {
  firstName: "Colt",
  sayHi: function(){
    return "Hi " + this.firstName
  },
  addNumbers: function(a,b,c,d){
    return this.firstName + " just calculated " + (a+b+c+d);
  }
}

var elie = {
  firstName: "Elie"
}

colt.sayHi() // Hi Colt
colt.sayHi.apply(elie) // Hi Elie

// well this seems the same...but what happens when we start adding arguments?

colt.addNumbers(1,2,3,4) // Colt just calculated 10

colt.addNumbers.call(elie,1,2,3,4) // Elie just calculated 10

colt.addNumbers.apply(elie,[1,2,3,4]) // Elie just calculated 10
```

And what about bind?

The parameters work like call, but bind returns a function with the context of 'this' bound already!

```
var colt = {
  firstName: "Colt",
  sayHi: function(){
    return "Hi " + this.firstName
  },
  addNumbers: function(a,b,c,d){
    return this.firstName + " just calculated " + (a+b+c+d);
  }
}

var elie = {
  firstName: "Elie"
}

var elieCalc = colt.addNumbers.bind(elie,1,2,3,4) // function(){}...
elieCalc() // Elie just calculated 10

// With bind - we do not need to know all the arguments up front!

var elieCalc2 = colt.addNumbers.bind(elie,1,2) // function(){}...
elieCalc2(3,4) // Elie just calculated 10
```

Bind in the wild

Very commonly we lose the context of 'this', but in functions that we do not want to execute right away!

```
var colt = {
  firstName: "Colt",
  sayHi: function(){
    setTimeout(function(){
      console.log("Hi " + this.firstName)
    },1000)
  }
}
```

```
colt.sayHi() // Hi undefined (1000 milliseconds later)
```

Use bind to set the correct context of 'this'

```
var colt = {
  firstName: "Colt",
  sayHi: function(){
    setTimeout(function(){
      console.log("Hi " + this.firstName)
    }.bind(this),1000)
  }
}
```

```
colt.sayHi() // Hi Colt (1000 milliseconds later)
```

Creating an object

So how do we use our constructor to actually construct objects?

```
function House(bedrooms, bathrooms, numSqft){
  this.bedrooms = bedrooms;
  this.bathrooms = bathrooms;
  this.numSqft = numSqft;
}
```

```
var firstHouse = House(2,2,1000) // does this work?
firstHouse // undefined...guess not!
```

Why is this not working??

- We are not returning anything from the function so our House function returns undefined
- We are not explicitly binding the keyword 'this' or placing it inside a declared object. This means the value of the keyword 'this' will be the global object, which is not what we want!

The 'new' keyword

Our solution to the problem!

```
function House(bedrooms, bathrooms, numSqft){  
  this.bedrooms = bedrooms;  
  this.bathrooms = bathrooms;  
  this.numSqft = numSqft;  
}
```

```
var firstHouse = new House(2,2,1000)  
firstHouse.bedrooms // 2  
firstHouse.bathrooms // 2  
firstHouse.numSqft // 1000
```

So what does the new keyword do? A lot more than we might think...

- It first creates an empty object
- It then sets the keyword 'this' to be that empty object
- It adds the line `return this` to the end of the function, which follows it
- It adds a property onto the empty object called "__proto__", which links the prototype property on the constructor function to the empty object (more on this later)

Using call/apply

We can refactor our code quite a bit using call + apply

```
function Car(make, model, year){  
  this.make = make;  
  this.model = model;  
  this.year = year;  
  // we can also set properties on the keyword this  
  // that are preset values  
  this.numWheels = 4;  
}
```

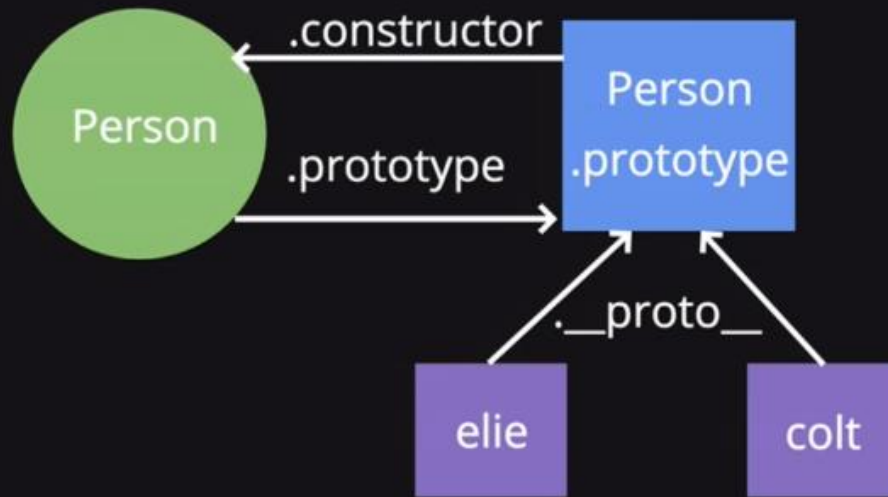
```
function Motorcycle(make, model, year){  
  // using call  
  Car.call(this, make, model, year)  
  this.numWheels = 2;  
}
```

```
function Motorcycle(make, model, year){  
  // using apply  
  Car.apply(this, [make,model,year]);  
  this.numWheels = 2;  
}
```

```
function Motorcycle(make, model, year){  
  // even better using apply with arguments  
  Car.apply(this, arguments);  
  this.numWheels = 2;  
}
```

199 people have written a note here.

A Small Diagram



Code

Let's see that previous example in code - feel free to look back at the diagram

```
// this is the constructor function
function Person(name){
  this.name = name;
}

// this is an object created from the Person constructor
var elie = new Person("Elie");
var colt = new Person("Colt");

// since we used the new keyword, we have established
// a link between the object and the prototype property
// we can access that using __proto__

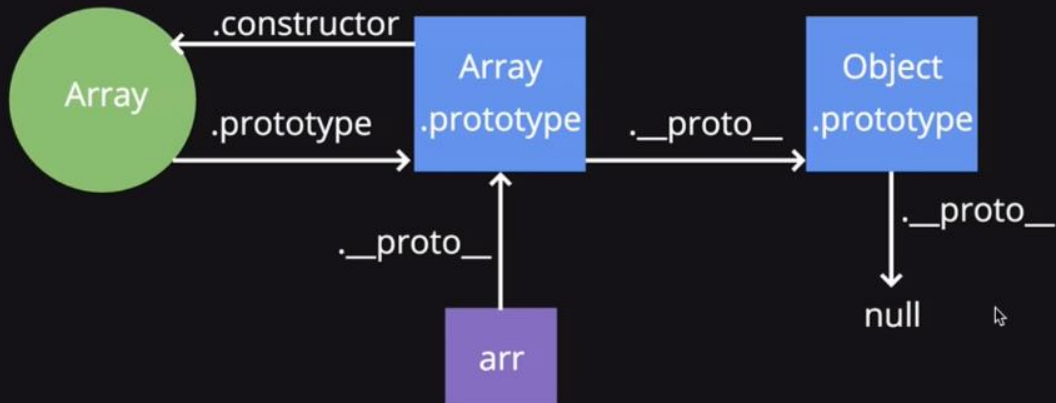
elie.__proto__ === Person.prototype; // true
colt.__proto__ === Person.prototype; // true

// The Person.prototype object also has a property
// called constructor which points back to the function
Person.prototype.constructor === Person; // true
```

```
All Errors Warnings Info Logs Debug
> function Person(name){
  this.name = name;
}
< undefined
> Person.prototype
< Object {}
> var elie = new Person("Elie");
  var colt = new Person("Colt");
< undefined
> elie.__proto__ === Person.prototype
< true
> Person.prototype.constructor
< function Person(name){
  this.name = name;
}
>
```

Prototype Chain

How does JavaScript find methods and properties?



Refactoring

Now that we know that objects created by the same constructor have a shared prototype, let's refactor some code

```
function Person(name){
  this.name = name;
  this.sayHi = function(){
    return "Hi " + this.name;
  }
}

elie = new Person("Elie");
elie.sayHi(); // Hi Elie

// now this code works, but it is inefficient
// every time we make an object using the new keyword we have to redefine this function
// but its the same for everyone! Let's put it on the prototype instead!

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

Person.prototype.sayHi = function(){
  return "Hi " + this.name;
}

elie = new Person("Elie");
elie.sayHi(); // Hi Elie
```

How to link a function to a object(js has not class)

```

function Vehicle(make, model, year){
  this.make = make;
  this.model = model;
  this.year = year;
  this.isRunning = false;
}

Vehicle.prototype.turnOn = function(){
  this.isRunning = true;
}

Vehicle.prototype.turnOff = function(){
  this.isRunning = false;
}

Vehicle.prototype.honk = function(){
  if(this.isRunning){
    return "beep!";
  }
}

```

Our first closure

```

function outer(a){
  return function inner(b){
    // the inner function is making use of the variable "a"
    // which was defined in an outer function called "outer"
    // and by the time this is called, that outer function has returned
    // this function called "inner" is a closure!
    return a + b
  }
}

outer(5)(5) // 10

var storeOuter = outer(5)
storeOuter(10) // 15

```



Private Variables

In other languages, there exists support for variables that can not be modified externally, we call those private variables, but in JavaScript we don't have that built in. No worries - closures can help!

```
function counter(){
  var count = 0
  return function(){
    return ++count
  }
}

counter1 = counter()
counter1() // 1
counter1() // 2

counter2 = counter()
counter2() // 1
counter2() // 2

counter1() // 3 this is not affected by counter2!

count // ReferenceError: count is not defined - because it is private!
```

More Privacy

Let's look at this example:

```
function classRoom(){
  var instructors = ["Colt", "Elie"]
  return {
    getInstructors: function(){
      return instructors;
    },
    addInstructor: function(instructor){
      instructors.push(instructor);
      return instructors;
    }
  }
}

course1 = classRoom()
course1.getInstructors() // ["Elie", "Colt"]
course1.addInstructor("Ian") // ["Elie", "Colt", "Ian"]
course1.getInstructors() // ["Elie", "Colt", "Ian"]

course2 = classRoom()
course2.getInstructors() // ["Elie", "Colt"] - not affected by course1

// we also have NO access to the instructors variable
// which makes it private - no one can modify it...you're stuck with Colt and Elie
```