Core Javascript

Some words about javascript

- Call so-called object-oriented programming languages like java , c# etc., as “ class based “ language And javascript as “ object-oriented language.
- In fact , there are only two languages (javascript , other is called something like levo) among all other languages that truly are object-oriented.
- Or you can say javascript is an object based language as many of the things we access or we work with are objects like “document”.
- Example document.getElementById.
- There is a misconception about javascript that “ everything in javascript is object “ that is not true because , althought , the things mostly you work with are objects but javascript also have primitive types.
- JavaScript is a loosely typed or a dynamic language. That means you don't have to declare the type of a variable ahead of time.
- There is no right way of doing things in javascript. Thought each and every approach has pros and cons and based on that , there are some suggestions like “ javascript design patterns”

DataTypes in Javascript

Six data types that are primitives:

- Boolean
- Null
- Undefined
- Number
- String
- Symbol (new in ECMAScript 6)
- and Object

Its just the overview of datatypes javascript has , we are not at all interested in discussing about each an every data type javascript has except for one data type or built-in data structure called “Object”.

Primitive values

All types except objects define immutable values (values, which are incapable of being changed). For example and unlike to C, Strings are immutable. We refer to values of these types as "primitive values".

what are function and array in javascript then

- By contrast, there are a few special object sub-types, which we can refer to as complex primitives.
- function is a sub-type of object (technically, a "callable object"). Functions in JS are said to be "first class" in that they are basically just normal objects (with callable behavior semantics bolted on), and so they can be handled like any other plain object.
- Arrays are also a form of objects, with extra behavior. The organization of contents in arrays is slightly more structured than for general objects.

Built-in Objects

There are several other object sub-types, usually referred to as built-in objects

- String
- Number
- Boolean
- Object
- Function
- Array
- Date
- RegExp
- Error.

Objects - part 1

- An object is a collection of related data and/or functionality
- In JavaScript, most things are objects, from core JavaScript features like strings and arrays to the browser APIs built on top of JavaScript. You can even create your own objects to encapsulate related functions and variables into efficient packages, and act as handy data containers.

```
var person = {}; // empty object


var person = {
  name: ['Bob', 'Smith'],
  age: 32,
  gender: 'male',
  interests: ['music', 'skiing'],
  bio: function() {
    alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age + ' years old. He likes ' + this.interests[0] + ' and ' + this.interests[1] + '.');
  },
  greeting: function() {
    alert('Hi! I\'m ' + this.name[0] + '.');
  }
};
Person.greeting();
```

```
var anObject = new Object();


Var onObject1 = Object.create({name: ['Bob', 'Smith'],
  age: 32,
  gender: 'male',
  interests: ['music', 'skiing'],
  bio: function() {
    alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age + ' years old.
He likes ' + this.interests[0] + ' and ' + this.interests[1] + '.');
  },
  greeting: function() {
    alert('Hi! I\'m ' + this.name[0] + '.');
  });
onObject1.greeting();
```

The this keyword

why this keyword ?

Let's try to illustrate the motivation and utility of this:

```
function identify() {
  return this.name.toUpperCase();
}
function speak() {
  var greeting = "Hello, I'm " + identify.call( this );
  console.log( greeting );
}
var me = {
  name: "Kyle"
};
var you = {
  name: "Reader"
};
identify.call( me ); // KYLE
identify.call( you ); // READER
speak.call( me ); // Hello, I'm KYLE
speak.call( you ); // Hello, I'm READER
```

- This code snippet allows the identify() and speak() functions to be re-used against multiple context (me and you) objects, rather than needing a separate version of the function for each object.
- Instead of relying on this, you could have explicitly passed in a context object to both identify() and speak().

```
function identify(context) {
  return context.name.toUpperCase();
}

function speak(context) {
  var greeting = "Hello, I'm " + identify( context );
  console.log( greeting );
}

identify( you ); // READER
speak( me ); // Hello, I'm KYLE
```

- However, the this mechanism provides a more elegant way of implicitly "passing along" an object reference, leading to cleaner API design and easier re-use.

- The more complex your usage pattern is, the more clearly you'll see that passing context around as an explicit parameter is often messier than passing around a this context.

misconceptions about this keyword

1) it refers to the function itself.

The first common temptation is to assume this refers to the function itself.

Consider the following code, where we attempt to track how many times a function (foo) was called:

```
function foo(num) {
  console.log( "foo: " + num );
  // keep track of how many times `foo` is called
  this.count++;
}
foo.count = 0;
var i;
for (i=0; i<10; i++) {
  if (i > 5) {
    foo( i );
  }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9
// how many times was `foo` called?
console.log( foo.count ); // 0 -- WTF?
```

foo.count is still 0, even though the four console.log statements clearly indicate foo(..) was in fact called four times.

When the code executes foo.count = 0, indeed it's adding a property count to the function object foo. But for the this.count reference inside of the function, this is not in fact pointing at all to that function object, and so even though the property names are the same, the root objects are different, and confusion ensues.

------ avoid anonymous functions ---------

To reference a function object from inside itself, this by itself will typically be insufficient. You generally need a reference to the function object via a lexical identifier (variable) that points at it.

Consider these two functions:

```
function foo() {
  foo.count = 4; // `foo` refers to itself
}
setTimeout( function(){
  // anonymous function (no name), cannot
  // refer to itself
}, 10 );
```

In the first function, called a "named function", foo is a reference that can be used to refer to the function from inside itself.

But in the second example, the function callback passed to setTimeout(..) has no name identifier (so called an "anonymous function"), so there's no proper way to refer to the function object itself.

Note: The old-school but now deprecated and frowned-upon arguments.callee reference inside a function also points to the function object of the currently executing function. This reference is typically the only way to access an anonymous function's object from inside itself. The best approach, however, is to avoid the use of anonymous functions altogether, at least for those which require a self-reference, and instead use a named function (expression). arguments.callee is deprecated and should not be used.

```
function foo(num) {
  console.log( "foo: " + num );
  // keep track of how many times `foo` is called
  foo.count++;
}
foo.count = 0;
var i;
for (i=0; i<10; i++) {
  if (i > 5) {
    foo( i );
  }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( foo.count ); // 4
```

2) its refers to the scope

quiz - will the function bar get called ???

```
function foo() {
  var a = 2;
  this.bar();
}
function bar() {
  console.log( this.a );
}
foo(); //undefined
```

what is this keyword and how it works

- this is not an author-time binding but a runtime binding. It is contextual based on the conditions of the function's invocation. this binding has nothing to do with where a function is declared, but has instead everything to do with the manner in which the function is called.
- When a function is invoked, an activation record, otherwise known as an execution context, is created. This record contains information about where the function was called from (the call-stack), how the function was invoked, what parameters were passed, etc. One of the properties of this record is the this reference which will be used for the duration of that function's execution.
- As this binding depends on the call-site of the function we need to understand the call-site.

Call-site

- To understand this binding, we have to understand the call-site: the location in code where a function is called (not where it's declared). We must inspect the call-site to answer the question: what's this this a reference to?

Let's demonstrate call-stack and call-site:

```
function baz() {
    // call-stack is: `baz`
    // so, our call-site is in the global scope
    console.log( "baz" );
    bar(); // <-- call-site for `bar`
}
function bar() {
    // call-stack is: `baz` -> `bar`
    // so, our call-site is in `baz`
    console.log( "bar" );
    foo(); // <-- call-site for `foo`
}
function foo() {
    // call-stack is: `baz` -> `bar` -> `foo`
    // so, our call-site is in `bar`
    console.log( "foo" );
}
baz(); // <-- call-site for `baz`
```

- Take care when analyzing code to find the actual call-site (from the call-stack), because it's the only thing that matters for this binding.

Rules for this binding

- We turn our attention now to how the call-site determines where this will point during the execution of a function.
- You must inspect the call-site and determine which of 4 rules applies.
- 4 this binding rules :-
  1) Default binding
  2) Implicit binding
  3) Explicit binding
  4) New keyword binding

Default Binding

- The first rule we will examine comes from the most common case of function calls: standalone function invocation. Think of this this rule as the default catch-all rule when none of the other rules apply.

Consider this code:

```
function foo() {
  console.log( this.a );
}
var a = 2;
foo(); // 2
```

To be continued…

- we see that when foo() is called, this.a resolves to our global variable a. Why? Because in this case, the default binding for this applies to the function call, and so points this at the global object.
- How do we know that the default binding rule applies here? We examine the call-site to see how foo() is called. In our snippet, foo() is called with a plain, un-decorated function reference. None of the other rules we will demonstrate will apply here, so the default binding applies instead.
- If strict mode is in effect, the global object is not eligible for the default binding, so the this is instead set to undefined.

```
function foo() {
  "use strict";
  console.log( this.a );
}
var a = 2;
foo(); // TypeError: `this` is `undefined`
```

Implicit Binding

- Another rule to consider is: does the call-site have a context object, also referred to as an owning or containing object,

Consider:

```
function foo() {
  console.log( this.a );
}
var obj = {
  a: 2,
  foo: foo
};
obj.foo(); // 2
```

- Firstly, notice the manner in which foo() is declared and then later added as a reference property onto obj. Regardless of whether foo() is initially declared on obj, or is added as a reference later (as this snippet shows), in neither case is the function really "owned" or "contained" by the obj object.
- However, the call-site uses the obj context to reference the function, so you could say that the obj object "owns" or "contains" the function reference at the time the function is called.
- Whatever you choose to call this pattern, at the point that foo() is called, it's preceded by an object reference to obj. When there is a context object for a function reference, the implicit binding rule says that it's that object which should be used for the function call's this binding.
- Because obj is the this for the foo() call, this.a is synonymous with obj.a.
- Only the top/last level of an object property reference chain matters to the call-site. For instance:

```
function foo() {
  console.log( this.a );
}
var obj2 = {
  a: 42,
  foo: foo
};
var obj1 = {
  a: 2,
  obj2: obj2
};
obj1.obj2.foo(); // 42
```

Implicitly Lost

- One of the most common frustrations that this binding creates is when an implicitly bound function loses that binding, which usually means it falls back to the default binding, of either the global object or undefined, depending on strict mode.

Consider:

```
function foo() {
  console.log( this.a );
}
var obj = {
  a: 2,
  foo: foo
};
var bar = obj.foo; // function reference/alias!
var a = "oops, global"; // `a` also property on global object
bar(); // "oops, global“
```

- Even though bar appears to be a reference to obj.foo, in fact, it's really just another reference to foo itself. Moreover, the call-site is what matters, and the call-site is bar(), which is a plain, un-decorated call and thus the default binding applies.

- The more subtle, more common, and more unexpected way this occurs is when we consider passing a callback function:

```
function foo() {
  console.log( this.a );
}
function doFoo(fn) {
  // `fn` is just another reference to `foo`
  fn(); // <-- call-site!
}
var obj = {
  a: 2,
  foo: foo
};
var a = "oops, global"; // `a` also property on global object
doFoo( obj.foo ); // "oops, global“
```

Explicit Binding

- what if you want to force a function call to use a particular object for the this binding, without putting a property function reference on the object?
- "All" functions in the language have some utilities available to them which can be useful for this task. Specifically, functions have call(..) and apply(..) methods. The vast majority of functions provided, and certainly all functions you will create, do have access to call(..) and apply(..).
- How do these utilities work? They both take, as their first parameter, an object to use for the this, and then invoke the function with that this specified. Since you are directly stating what you want the this to be, we call it explicit binding.

Consider:

```
function foo() {
  console.log( this.a );
}
var obj = {
  a: 2
};
foo.call( obj ); // 2
```

- Invoking foo with explicit binding by foo.call(..) allows us to force its this to be obj.
- If you pass a simple primitive value (of type string, boolean, or number) as the this binding, the primitive value is wrapped in its object-form (new String(..), new Boolean(..), or new Number(..), respectively). This is often referred to as "boxing".

Hard Binding

Consider:

```
function foo() {
  console.log( this.a );
}
var obj = {
  a: 2
};
var bar = function() {
  foo.call( obj );
};
bar(); // 2
setTimeout( bar, 100 ); // 2
// `bar` hard binds `foo`'s `this` to `obj`
// so that it cannot be overriden
bar.call( window ); // 2
```

- Let's examine how this variation works. We create a function bar() which, internally, manually calls foo.call(obj), thereby forcibly invoking foo with obj binding for this. No matter how you later invoke the function bar, it will always manually invoke foo with obj. This binding is both explicit and strong, so we call it hard binding.
- The most typical way to wrap a function with a hard binding creates a pass-thru of any arguments passed and any return value received:

```
function foo(something) {
  console.log( this.a, something );
  return this.a + something;
}
var obj = {
  a: 2
};
var bar = function() {
  return foo.apply( obj, arguments );
};
var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

- Bind helper

```
function foo(something) {
  console.log( this.a, something );
  return this.a + something;
}
// simple `bind` helper
function bind(fn, obj) {
  return function() {
    return fn.apply( obj, arguments );
  };
}
var obj = {
  a: 2
};
var bar = bind( foo, obj );
var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

- Since hard binding is such a common pattern, it's provided with a built-in utility as of ES5: Function.prototype.bind, and it's used like this:

```
function foo(something) {
  console.log( this.a, something );
  return this.a + something;
}
var obj = {
  a: 2
};
var bar = foo.bind( obj );
var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

bind(..) returns a new function that is hard-coded to call the original function with the this context set as you specified.

new binding (new keyword)

what are constructors in javascript (new keyword)

- In JS, constructors are just functions that happen to be called with the new operator in front of them. They are not attached to classes, nor are they instantiating a class. They are not even special types of functions. They're just regular functions that are, in essence, hijacked by the use of new in their invocation.
- This is an important but subtle distinction: there's really no such thing as "constructor functions", but rather construction calls of functions.
- When a function is invoked with new in front of it, otherwise known as a constructor call, the following things are done automatically:

1)a brand new object is created (aka, constructed) out of thin air
2)the newly constructed object is [[Prototype]]-linked
3)the newly constructed object is set as the this binding for that function call
4)unless the function returns its own alternate object, the new-invoked function call will automatically return the newly constructed object.

- Consider this code:

```
function foo(a) {
  this.a = a;
}
var bar = new foo( 2 );
console.log( bar.a ); // 2
```

- By calling foo(..) with new in front of it, we've constructed a new object and set that new object as the this for the call of foo(..). So new is the final way that a function call's this can be bound. We'll call this new binding.

rule precedence

- So, now we've uncovered the 4 rules for binding this in function calls. All you need to do is find the call-site and inspect it to see which rule applies. But, what if the call-site has multiple eligible rules? There must be an order of precedence to these rules
- It should be clear that the default binding is the lowest priority rule of the 4. So we'll just set that one aside.

- Which is more precedent, implicit binding or explicit binding? Let's test it:

```
function foo() {
  console.log( this.a );
}
var obj1 = {
  a: 2,
  foo: foo
};
var obj2 = {
  a: 3,
  foo: foo
};
obj1.foo(); // 2
obj2.foo(); // 3
obj1.foo.call( obj2 ); // 3
obj2.foo.call( obj1 ); // 2
```

- So, explicit binding takes precedence over implicit binding, which means you should ask first if explicit binding applies before checking for implicit binding.
- Before we explore that in a code listing, think back to how hard binding physically works, which is that Function.prototype.bind(..) creates a new wrapper function that is hard-coded to ignore its own this binding (whatever it may be), and use a manual one we provide.
- By that reasoning, it would seem obvious to assume that hard binding (which is a form of explicit binding) is more precedent than new binding, and thus cannot be overridden with new.

- Let's check:

```
function foo(something) {
  this.a = something;
}
var obj1 = {};
var bar = foo.bind( obj1 );
bar( 2 );
console.log( obj1.a ); // 2
var baz = new bar( 3 );
console.log( obj1.a ); // 2
console.log( baz.a ); // 3
```

- Whoa! bar is hard-bound against obj1, but new bar(3) did not change obj1.a to be 3 as we would have expected. Instead, the hard bound (to obj1) call to bar(..) is able to be overridden with new.
- Since new was applied, we got the newly created object back, which we named baz, and we see in fact that baz.a has the value 3.

Determining this

- Now, we can summarize the rules for determining this from a function call's call-site, in their order of precedence. Ask these questions in this order, and stop when the first rule applies.
- Is the function called with new (new binding)? If so, this is the newly constructed object.
  - var bar = new foo()
- Is the function called with call or apply (explicit binding), even hidden inside a bind hard binding? If so, this is the explicitly specified object.
  - var bar = foo.call(obj2)
- Is the function called with a context (implicit binding), otherwise known as an owning or containing object? If so, this is that context object.
  - var bar = obj1.foo()
- Otherwise, default the this (default binding). If in strict mode, pick undefined, otherwise pick the global object.
  - var bar = foo()
- That's it. That's all it takes to understand the rules of this binding for normal function calls. Well... almost.

Ignored this

- If you pass null or undefined as a this binding parameter to call, apply, or bind, those values are effectively ignored, and instead the default binding rule applies to the invocation.

```
function foo() {
  console.log( this.a );
}
var a = 2;
foo.call( null ); // 2
```

- Why would you intentionally pass something like null for a this binding?
- It's quite common to use apply(..) for spreading out arrays of values as parameters to a function call. Similarly, bind(..) can curry parameters (pre-set values), which can be very helpful.

```
function foo(a,b) {
  console.log( "a:" + a + ", b:" + b );
}
// spreading out array as parameters
foo.apply( null, [2, 3] ); // a:2, b:3
// currying with `bind(..)`
var bar = foo.bind( null, 2 );
bar( 3 ); // a:2, b:3
```

objects part 2

[[Prototype]]

- Objects in JavaScript have an internal property, denoted in the specification as [[Prototype]], which is simply a reference to another object. Almost all objects are given a non-null value for this property, at the time of their creation.

Consider:

```
var myObject = {
  a: 2
};
myObject.a; // 2
```

- the [[Get]] operation that is invoked when you reference a property on an object, such as myObject.a. For that default [[Get]] operation, the first step is to check if the object itself has a property a on it, and if so, it's used.
- But it's what happens if a isn't present on myObject that brings our attention now to the [[Prototype]] link of the object.
- The default [[Get]] operation proceeds to follow the [[Prototype]] link of the object if it cannot find the requested property on the object directly.

```
var anotherObject = {
 a: 2
};
// create an object linked to `anotherObject`
var myObject = Object.create( anotherObject );
myObject.a; // 2
```

- So, we have myObject that is now [[Prototype]] linked to anotherObject. Clearly myObject.a doesn't actually exist, but nevertheless, the property access succeeds (being found on anotherObject instead) and indeed finds the value 2.
- But, if a weren't found on anotherObject either, its [[Prototype]] chain, if non-empty, is again consulted and followed.
- This process continues until either a matching property name is found, or the [[Prototype]] chain ends. If no matching property is ever found by the end of the chain, the return result from the [[Get]] operation is undefined
- The look-up stops once the property is found or the chain ends.

Object.prototype

- The top-end of every normal [[Prototype]] chain is the built-in Object.prototype. This object includes a variety of common utilities used all over JS, because all normal (built-in, not host-specific extension) objects in JavaScript "descend from" (aka, have at the top of their [[Prototype]] chain) the Object.prototype object.
- Some utilities found here you may be familiar with include .toString() and .valueOf().
    - myObject.foo = "bar";
- If foo is not already present directly on myObject, the [[Prototype]] chain is traversed, just like for the [[Get]] operation. If foo is not found anywhere in the chain, the property foo is added directly to myObject with the specified value, as expected.
- If the property name foo ends up both on myObject itself and at a higher level of the [[Prototype]] chain that starts at myObject, this is called shadowing. The foo property directly on myObject shadows any foo property which appears higher in the chain, because the myObject.foo look-up would always find the foo property that's lowest in the chain.

"Class"

- in JavaScript, there are no abstract patterns/blueprints for objects called "classes" as there are in class-oriented languages. JavaScript just has objects.
- In JavaScript, classes can't (being that they don't exist!) describe what an object can do. The object defines its own behavior directly. There's just the object.

"Class" Functions

- The peculiar "sort-of class" behavior hinges on a strange characteristic of functions: all functions by default get a public property on them called prototype, which points at an otherwise arbitrary object.

```
function Foo() {
  // ...
}
Foo.prototype;
```

- This object is often called "Foo's prototype", because we access it via an unfortunately-named Foo.prototype property reference.
- The most direct way to explain it is that each object created from calling new Foo() (see Chapter 2) will end up (somewhat arbitrarily) [[Prototype]]-linked to this "Foo dot prototype" object.
- Let's illustrate:

```
function Foo() {
  // ...
}
var a = new Foo();
Object.getPrototypeOf( a ) === Foo.prototype; // true
```

- When a is created by calling new Foo(), one of the things (see Chapter 2 for all four steps) that happens is that a gets an internal [[Prototype]] link to the object that Foo.prototype is pointing at

- But in JavaScript, there are no such copy-actions performed. You don't create multiple instances of a class. You can create multiple objects that [[Prototype]] link to a common object. But by default, no copying occurs, and thus these objects don't end up totally separate and disconnected from each other, but rather, quite linked.
- new Foo() results in a new object (we called it a), and that new object a is internally [[Prototype]] linked to the Foo.prototype object.

"Constructors“

Let's go back to some earlier code:

```
function Foo() {
  // ...
}
var a = new Foo();
```

- The Foo. Prototype object by default (at declaration time on line 1 of the snippet!) gets a public, non-enumerable (see Chapter 3) property called .constructor, and this property is a reference back to the function (Foo in this case) that the object is associated with. Moreover, we see that object a created by the "constructor" call new Foo() seems to also have a property on it called .constructor which similarly points to "the function which created it"

Constructor Or Call?

- In the above snippet, it's tempting to think that Foo is a "constructor", because we call it with new and we observe that it "constructs" an object.
- In reality, Foo is no more a "constructor" than any other function in your program. Functions themselves are not constructors. However, when you put the new keyword in front of a normal function call, that makes that function call a "constructor call". In fact, new sort of hijacks any normal function and calls it in a fashion that constructs an object, in addition to whatever else it was going to do.

For example:

```
function NothingSpecial() {
  console.log( "Don't mind me!" );
}
var a = new NothingSpecial();
// "Don't mind me!"
a; // {}
```

- NothingSpecial is just a plain old normal function, but when called with new, it constructs an object, almost as a side-effect, which we happen to assign to a. The call was a constructor call, but NothingSpecial is not, in and of itself, a constructor.
- In other words, in JavaScript, it's most appropriate to say that a "constructor" is any function called with the new keyword in front of it.
- Functions aren't constructors, but function calls are "constructor calls" if and only if new is used.

```
function Foo(name) {
  this.name = name;
}
Foo.prototype.myName = function() {
  return this.name;
};
var a = new Foo( "a" );
var b = new Foo( "b" );
a.myName(); // "a"
b.myName(); // "b“
```

- So, by virtue of how they are created, a and b each end up with an internal [[Prototype]] linkage to Foo.prototype. When myName is not found on a or b, respectively, it's instead found (through delegation, see Chapter 6) on Foo.prototype.

"Constructor" Redux

- Recall the discussion from earlier about the .constructor property, and how it seems like a.constructor === Foo being true means that a has an actual .constructor property on it, pointing at Foo? Not correct.
- This is just unfortunate confusion. In actuality, the .constructor reference is also delegated up to Foo.prototype, which happens to, by default, have a .constructor that points at Foo.
- the .constructor property on Foo.prototype is only there by default on the object created when Foo the function is declared. If you create a new object, and replace a function's default .prototype object reference, the new object will not by default magically get a .constructor on it.

Consider:

```
function Foo() { /* .. */ }
Foo.prototype = { /* .. */ }; // create a new prototype object
var a1 = new Foo();
a1.constructor === Foo; // false!
a1.constructor === Object; // true!
```

- a1 has no .constructor property, so it delegates up the [[Prototype]] chain to Foo.prototype. But that object doesn't have a .constructor either (like the default Foo.prototype object would have had!), so it keeps delegating, this time up to Object.prototype, the top of the delegation chain. That object indeed has a .constructor on it, which points to the built-in Object(..) function.

"(Prototypal) Inheritance"

```
function Foo(name) {
  this.name = name;
}
Foo.prototype.myName = function() {
  return this.name;
};
function Bar(name,label) {
  Foo.call( this, name );
  this.label = label;
}
// here, we make a new `Bar.prototype`
// linked to `Foo.prototype`
Bar.prototype = Object.create( Foo.prototype );
// Beware! Now `Bar.prototype.constructor` is gone,
// and might need to be manually "fixed" if you're
// in the habit of relying on such properties!
Bar.prototype.myLabel = function() {
  return this.label;
};
var a = new Bar( "a", "obj a" );
a.myName(); // "a"
a.myLabel(); // "obj a"
```

- The important part is Bar.prototype = Object.create(Foo.prototype). Object.create(..) creates a "new" object out of thin air, and links that new object's internal [[Prototype]] to the object you specify (Foo.prototype in this case).
- In other words, that line says: "make a new 'Bar dot prototype' object that's linked to 'Foo dot prototype'.“
- Note: A common mis-conception/confusion here is that either of the following approaches would also work, but they do not work as you'd expect:

```
// doesn't work like you want!
// works kinda like you want, but with
// side-effects you probably don't want :(
Bar.prototype = new Foo();
Bar.prototype = Foo.prototype;
```

- Bar.prototype = Foo.prototype doesn't create a new object for Bar.prototype to be linked to. It just makes Bar.prototype be another reference to Foo.prototype, which effectively links Bar directly to the same object as Foo links to: Foo.prototype. This means when you start assigning, like Bar.prototype.myLabel = ..., you're modifying not a separate object but the shared Foo.prototype object itself, which would affect any objects linked to Foo.prototype. This is almost certainly not what you want. If it is what you want, then you likely don't need Bar at all, and should just use only Foo and make your code simpler
- Bar.prototype = new Foo() does in fact create a new object which is duly linked to Foo.prototype as we'd want. But, it uses the Foo(..) "constructor call" to do it. If that function has any side-effects (such as logging, changing state, registering against other objects, adding data properties to this, etc.), those side-effects happen at the time of this linking (and likely against the wrong object!), rather than only when the eventual Bar() "descendants" are created, as would likely be expected.
- So, we're left with using Object.create(..) to make a new object that's properly linked, but without having the side-effects of calling Foo(..). The slight downside is that we have to create a new object, throwing the old one away, instead of modifying the existing default object we're provided.

example of prototypal inheritance and an explaination of it

```
function Person(first, last, age, gender, interests) {

    this.name = {

      first,

      last

    };

    this.age = age;

    this.gender = gender;

    this.interests = interests;

  };

  Person.prototype.bio = function() {

    // First define a string, and make it equal to the part of

    // the bio that we know will always be the same.

    var string = this.name.first + ' ' + this.name.last + ' is ' + this.age + ' years old. ';

    // define a variable that will contain the pronoun part of

    // the sencond sentence

    var pronoun;

    // check what the value of gender is, and set pronoun

    // to an appropriate value in each case

    if(this.gender === 'male' || this.gender === 'Male' || this.gender === 'm' || this.gender === 'M') {

      pronoun = 'He likes ';

    } else if(this.gender === 'female' || this.gender === 'Female' || this.gender === 'f' || this.gender === 'F') {

      pronoun = 'She likes ';

    } else {

      pronoun = 'They like ';

    }

    // add the pronoun string on to the end of the main string

    string += pronoun;
```