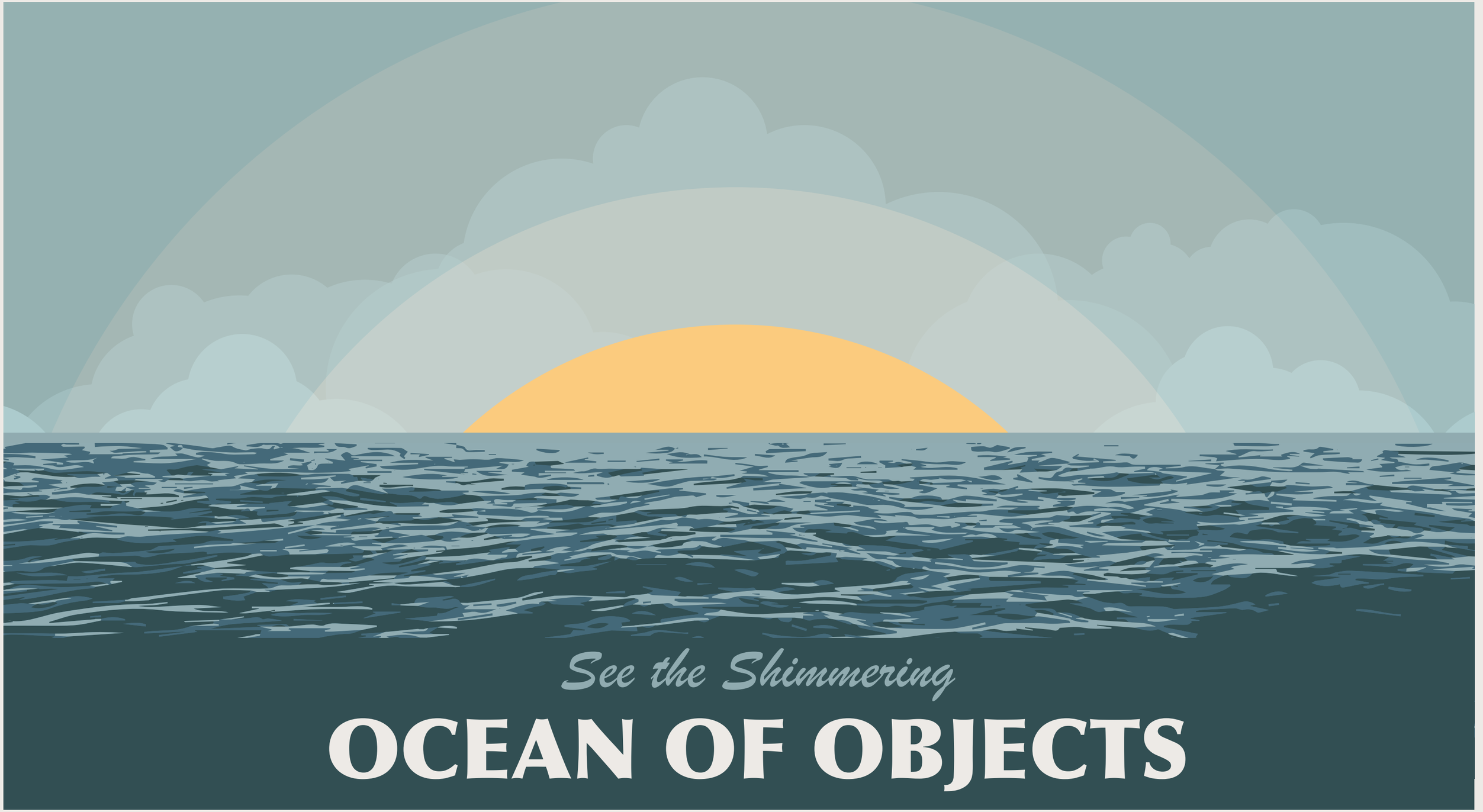
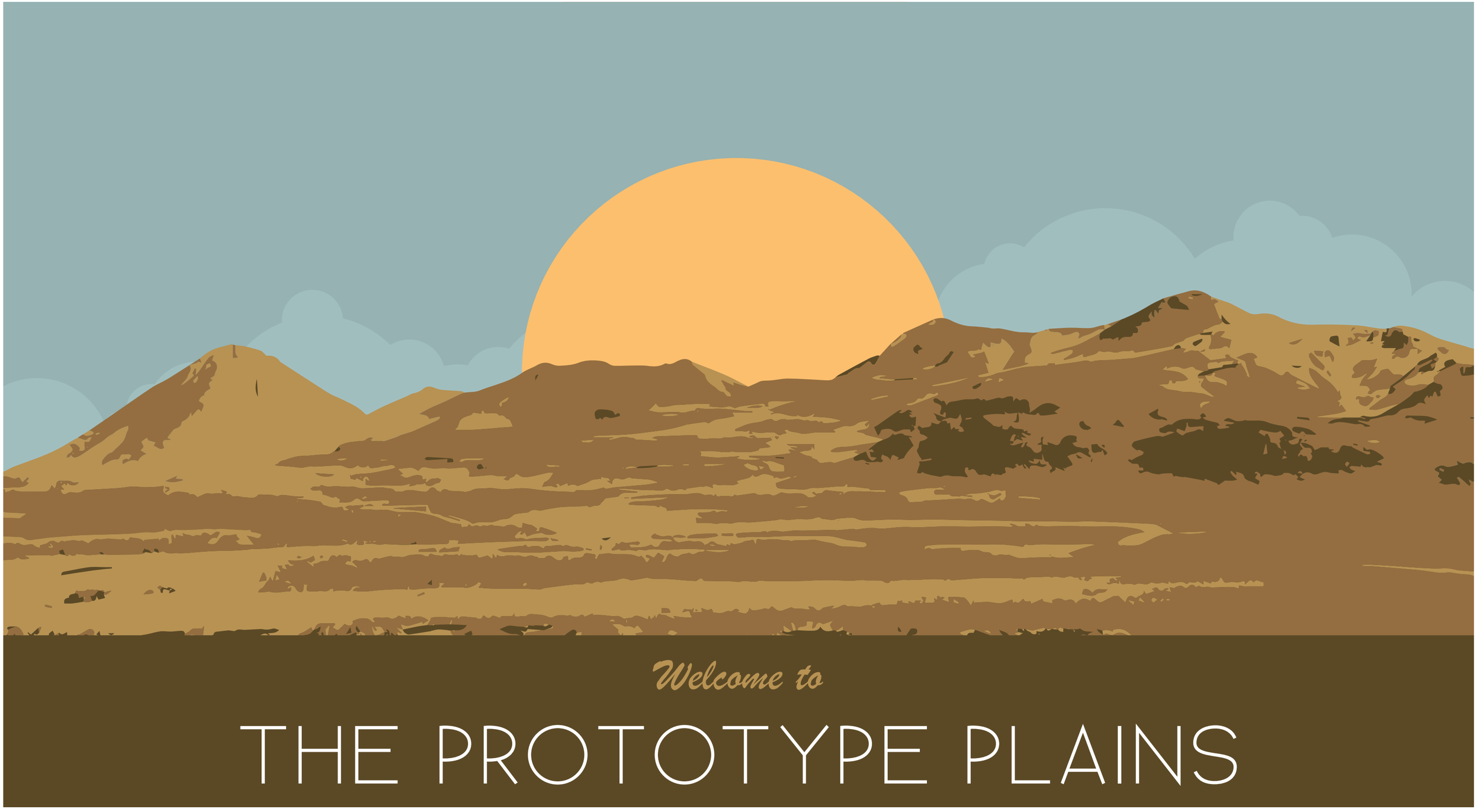

See the Shimmering
OCEAN OF OBJECTS


Welcome to
THE PROTOTYPE PLAINS

LEVEL 5

THE PROTOTYPE PLAINS

SURPRISE!

The Objects we've built so far have secret properties that we never saw!

`valueOf`

`constructor`

`toLocaleString`

`toString`

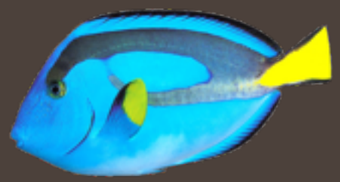
`isPrototypeOf`

`propertyIsEnumerable`

`hasOwnProperty`

WHERE DID ALL OF THESE PROPERTIES COME FROM?

All of these Objects have a mysterious “parent” object that gives them properties



`valueOf`

`constructor`

`toLocaleString`

`toString`

`isPrototypeOf`

`propertyIsEnumerable`

`hasOwnProperty`

WHERE DID ALL OF THESE PROPERTIES COME FROM?

All of these Objects have a mysterious “parent” object that gives them properties

`valueOf`

`constructor`

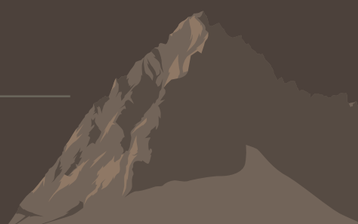
`toLocaleString`

`toString`

`isPrototypeOf`

`propertyIsEnumerable`

`hasOwnProperty`



THE OBJECT'S PARENT IS CALLED ITS "PROTOTYPE"

All of those mysterious properties belong to and come from the Object's prototype

`constructor` `valueOf` `toLocaleString`

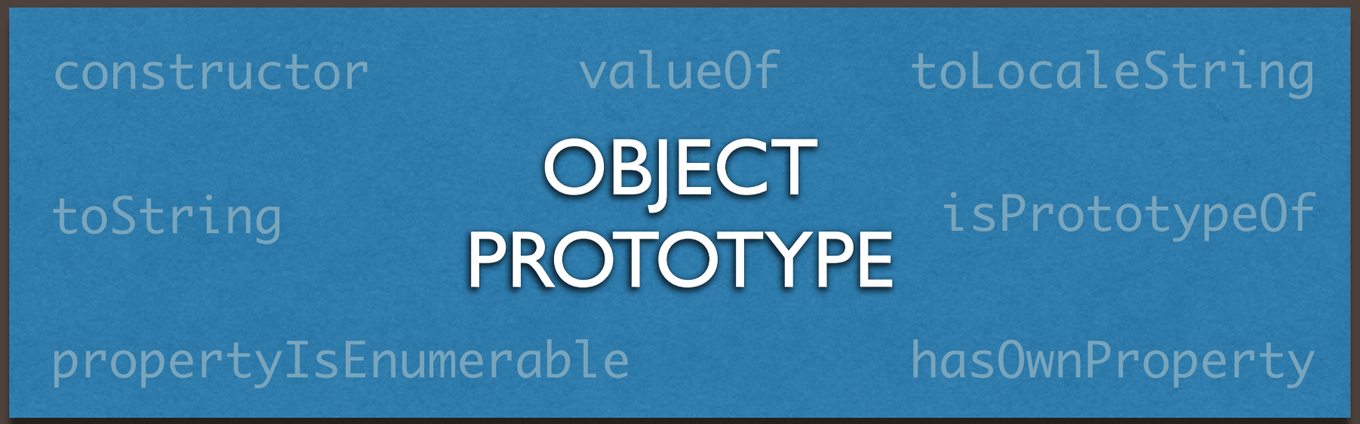
`toString` `isPrototypeOf`

`propertyIsEnumerable` `hasOwnProperty`

OBJECT
PROTOTYPE

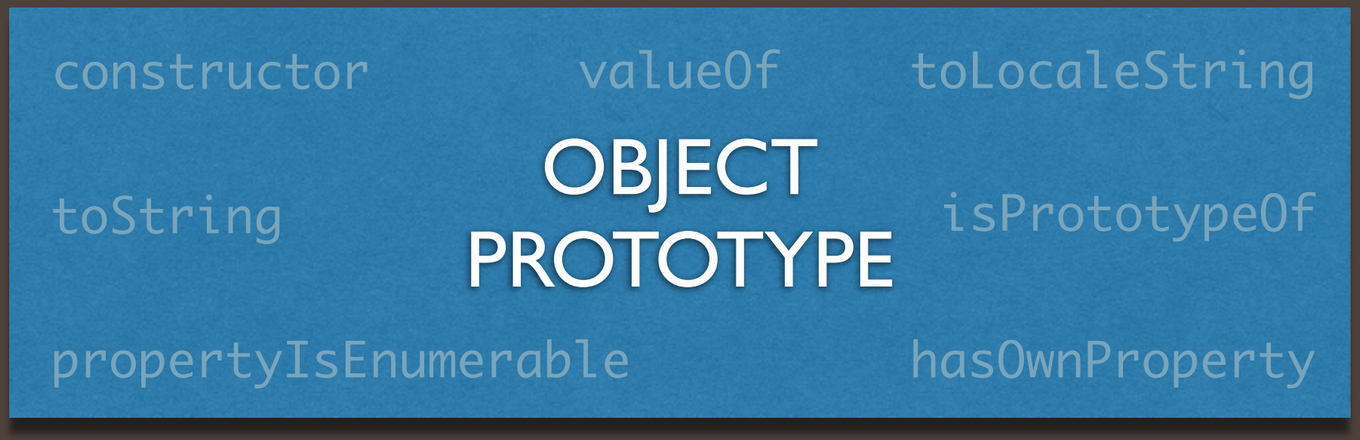
THE OBJECT'S PARENT IS CALLED ITS "PROTOTYPE"

All of those mysterious properties belong to and come from the Object's prototype



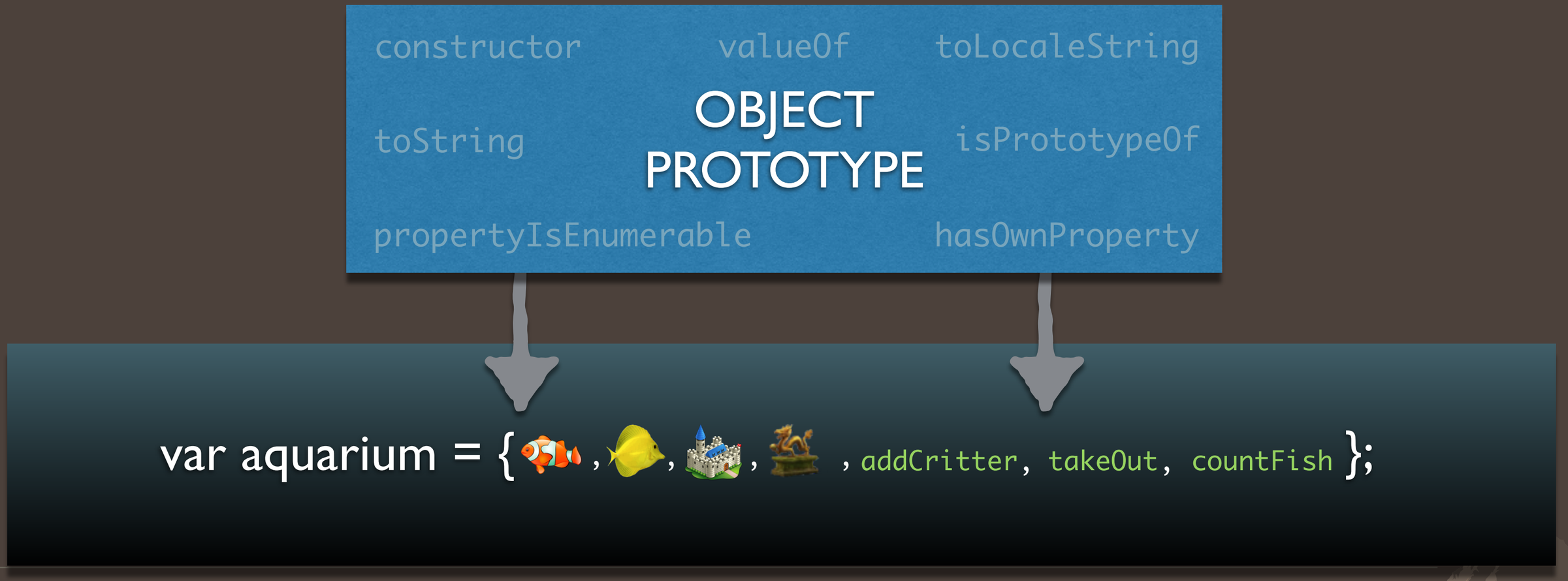
THE OBJECT'S PARENT IS CALLED ITS "PROTOTYPE"

When a generic Object is created, its prototype passes it many important properties



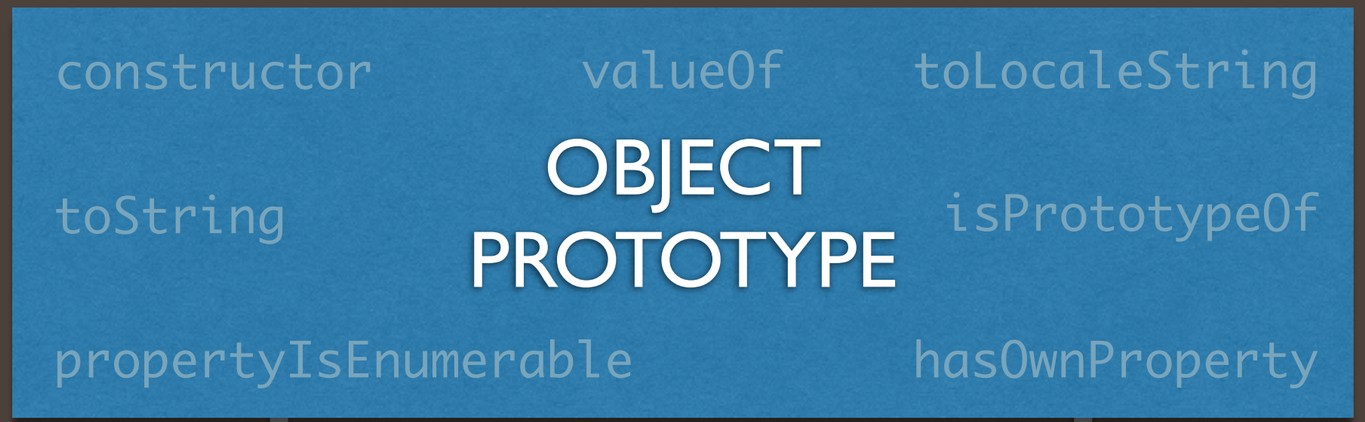
THE OBJECT'S PARENT IS CALLED ITS "PROTOTYPE"

When a generic Object is created, its prototype passes it many important properties



THE OBJECT'S PARENT IS CALLED ITS "PROTOTYPE"

When a generic Object is created, its prototype passes it many important properties




```
var aquarium = { 🐠, 🐟, 🏰, 🐉 , addCritter, takeOut, countFish,

};
```

THE OBJECT'S PARENT IS CALLED ITS "PROTOTYPE"

When a generic Object is created, its prototype passes it many important properties

A Prototype is like a blueprint Object for the Object we are trying to create.

OBJECT
PROTOTYPE

var aquarium = {  ,  ,  ,  , addCritter, takeOut, countFish,
constructor, valueOf, toLocaleString, isPrototypeOf,
toString, propertyIsEnumerable, hasOwnProperty, ... };

PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

Inheritance helps avoid over-coding multiple properties and methods into similar objects.

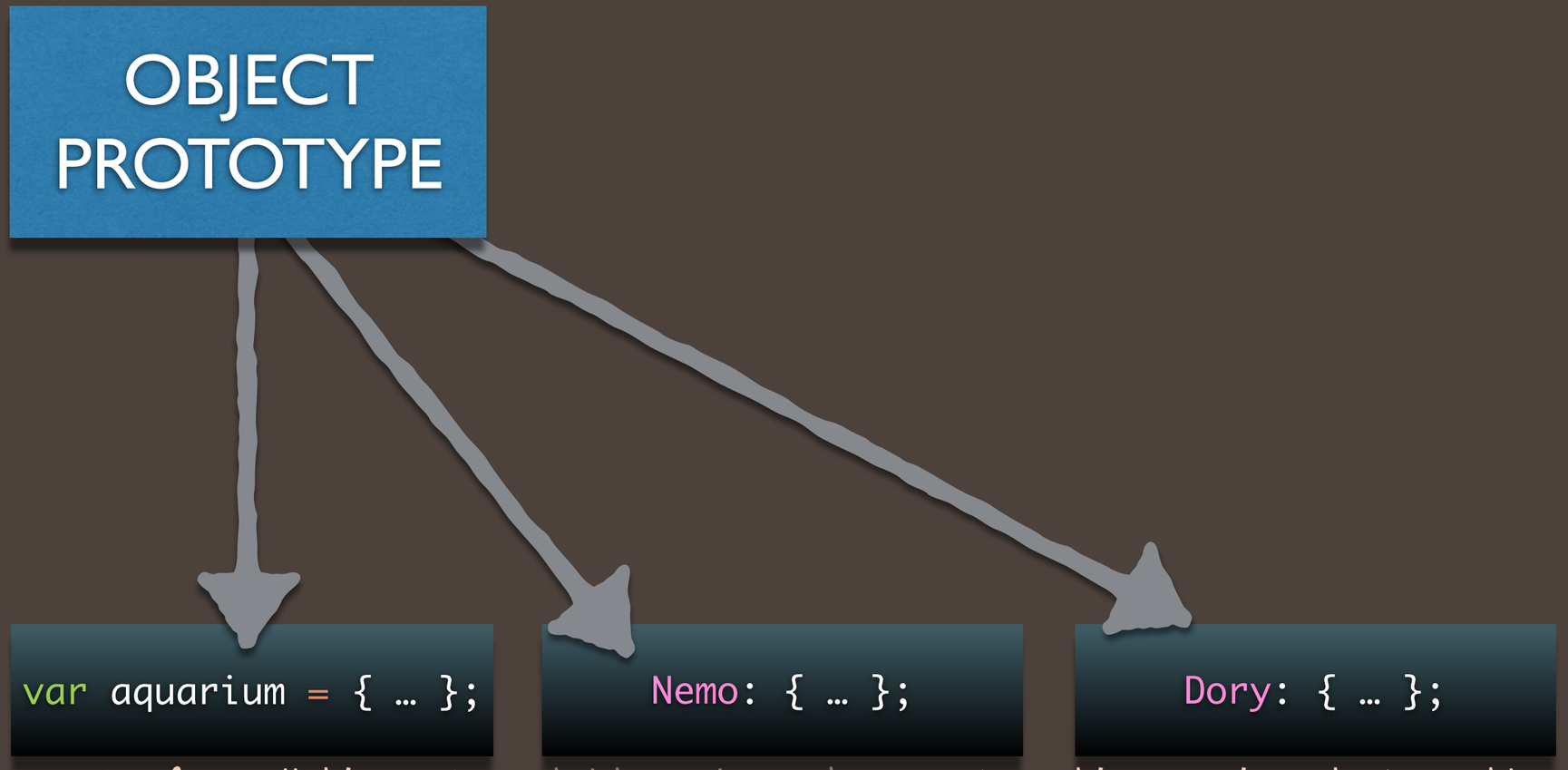
A Prototype is like a blueprint Object for the Object we are trying to create.

OBJECT
PROTOTYPE

var aquarium = {  ,  ,  ,  , addCritter, takeOut, countFish,
constructor, valueOf, toLocaleString, isPrototypeOf,
toString, propertyIsEnumerable, hasOwnProperty, ... };

PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

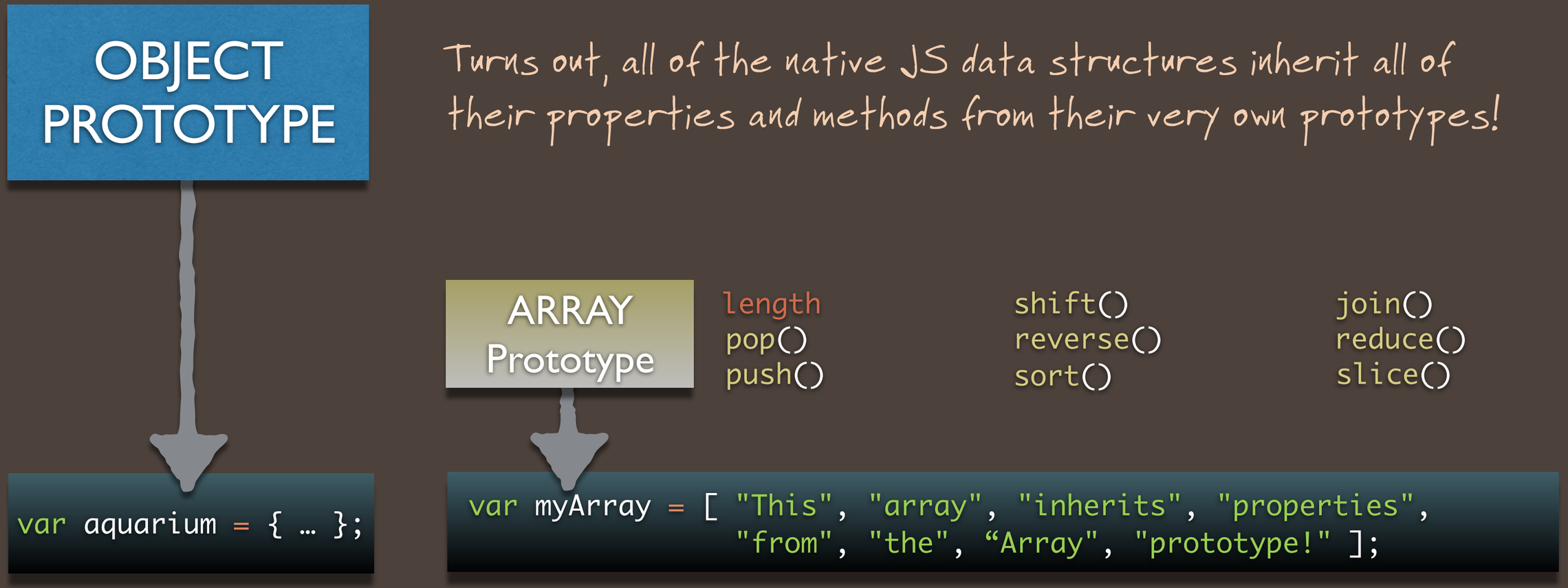
Inheritance helps avoid over-coding multiple properties and methods into similar objects.



So far, all the Object literals we've made with {} inherit directly from the highest level in the JS hierarchy, the Object prototype....

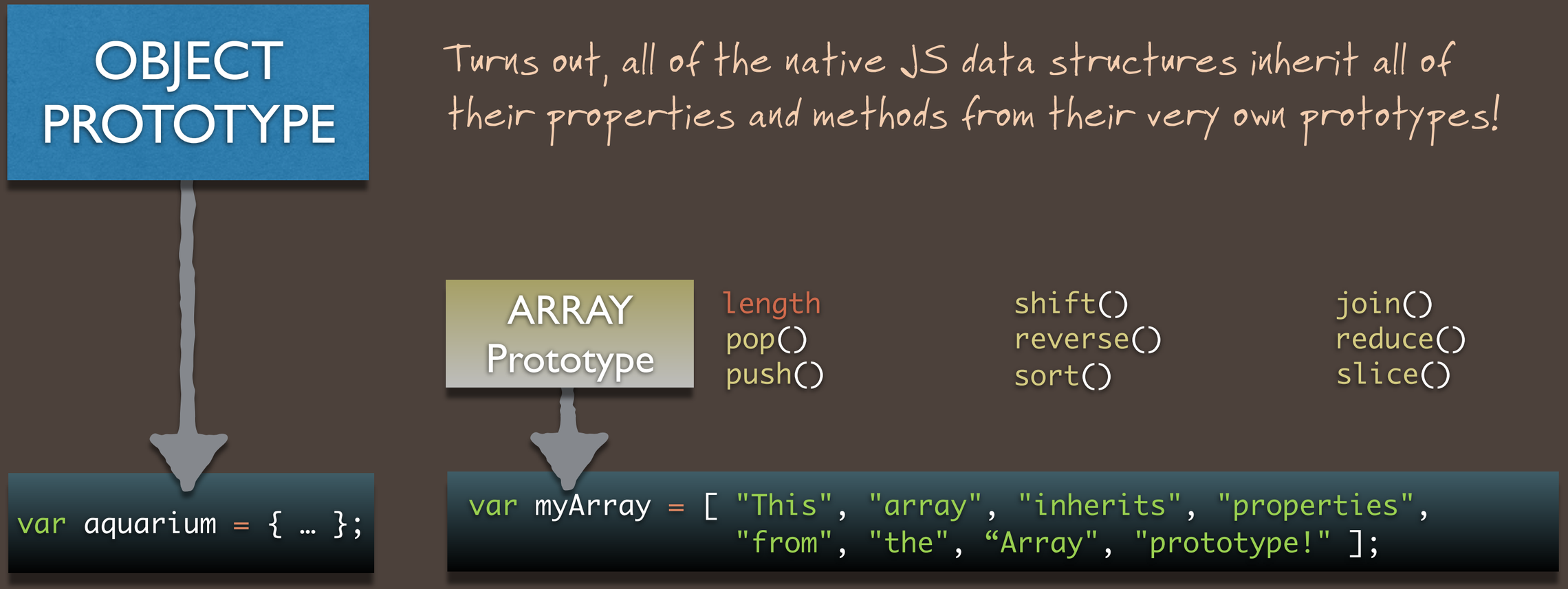
PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

Inheritance helps avoid over-coding multiple properties and methods into similar objects.



PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

Inheritance helps avoid over-coding multiple properties and methods into similar objects.



PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

Inheritance helps avoid over-coding multiple properties and methods into similar objects.

OBJECT PROTOTYPE

Turns out, all of the native JS data structures inherit all of their properties and methods from their very own prototypes!

```
var aquarium = { … };
```

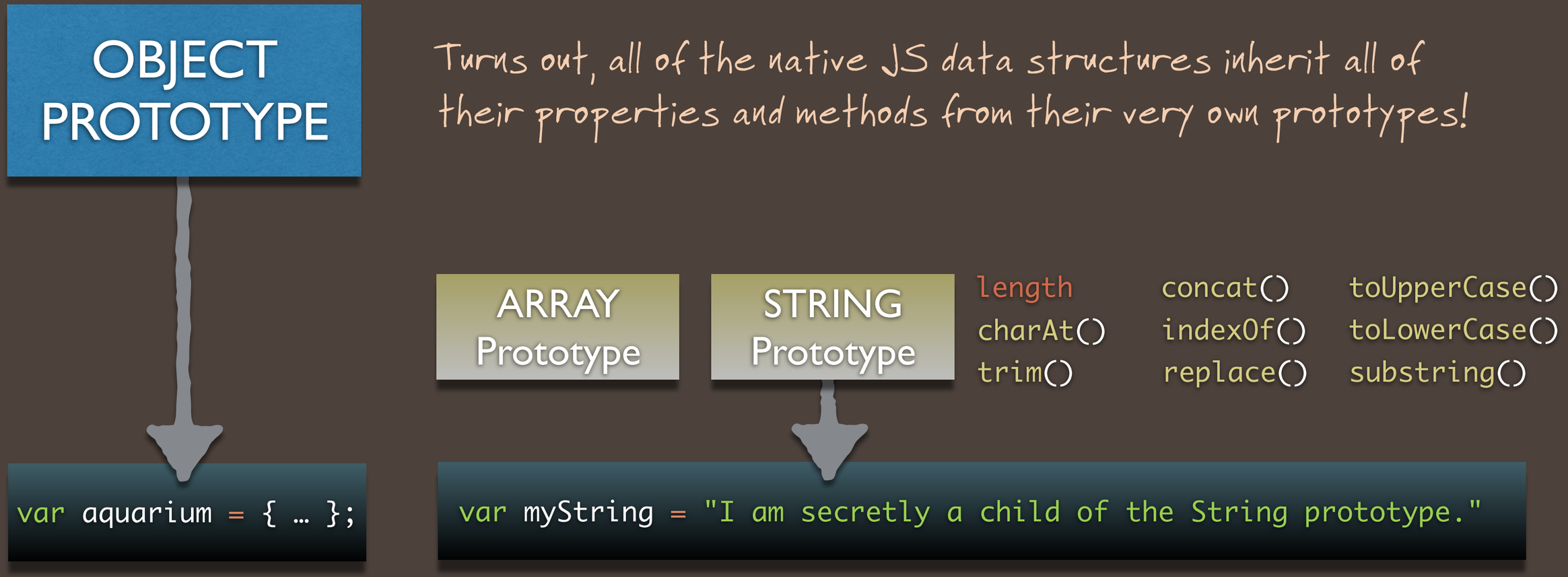
ARRAY Prototype

```
var myArray = [ "This", "array", "inherits", "properties",
                "from", "the", "Array", "prototype!" ];
```

```
myArray.length
myArray.pop()
myArray.push()
myArray.shift()
myArray.reverse()
myArray.sort()
myArray.reduce()
myArray.join()
myArray.slice()
```

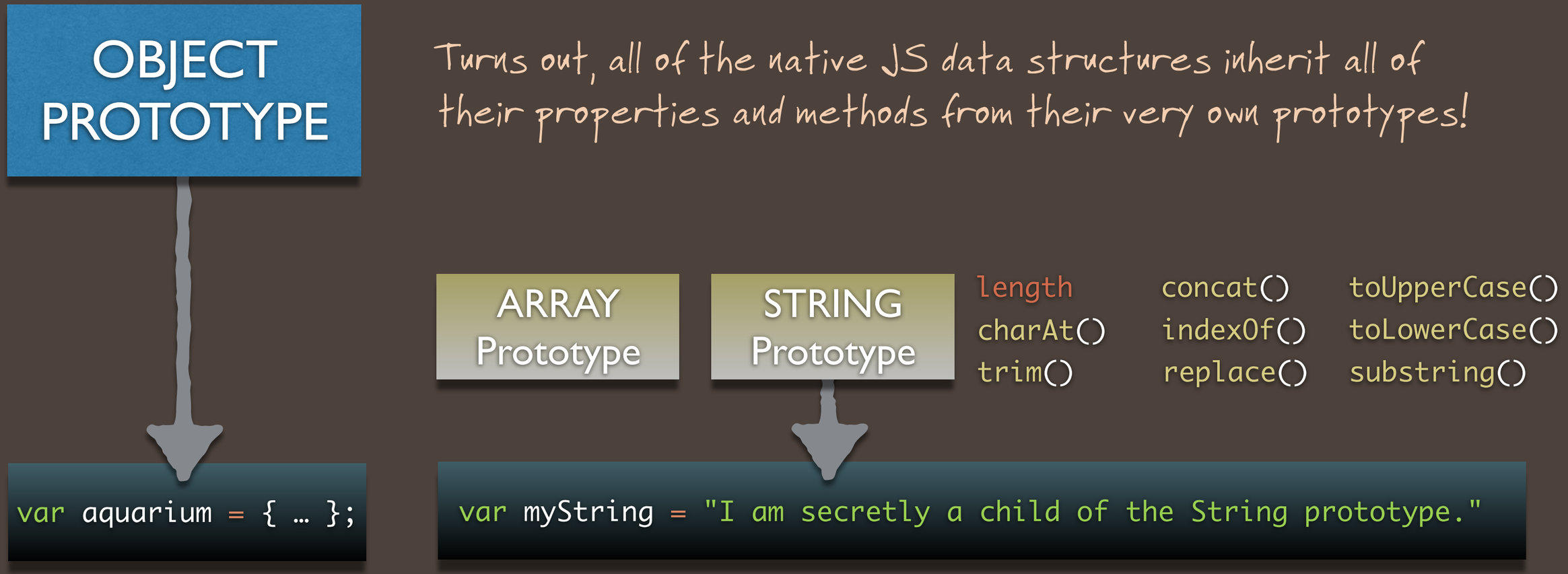
PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

Inheritance helps avoid over-coding multiple properties and methods into similar objects.



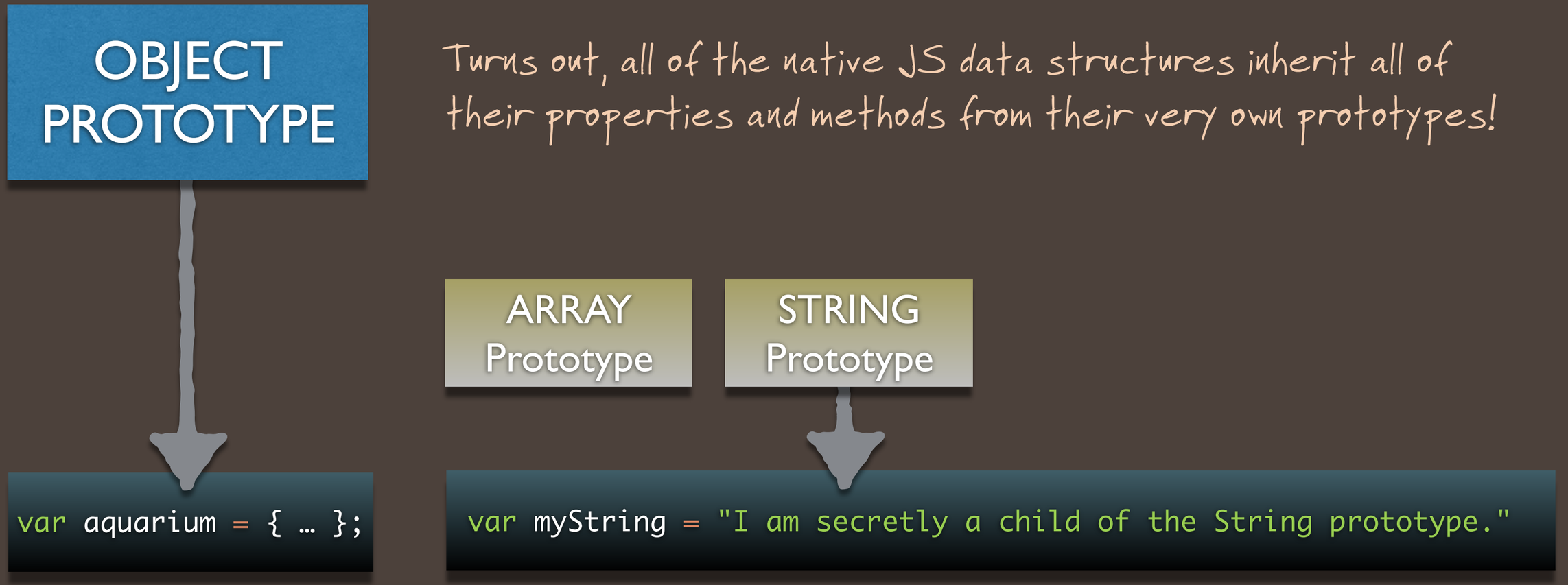
PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

Inheritance helps avoid over-coding multiple properties and methods into similar objects.



PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

Inheritance helps avoid over-coding multiple properties and methods into similar objects.



PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

Inheritance helps avoid over-coding multiple properties and methods into similar objects.

OBJECT PROTOTYPE

Turns out, all of the native JS data structures inherit all of their properties and methods from their very own prototypes!

ARRAY Prototype

STRING Prototype

NUMBER Prototype

```
toFixed()
toExponential()
toPrecision()
```

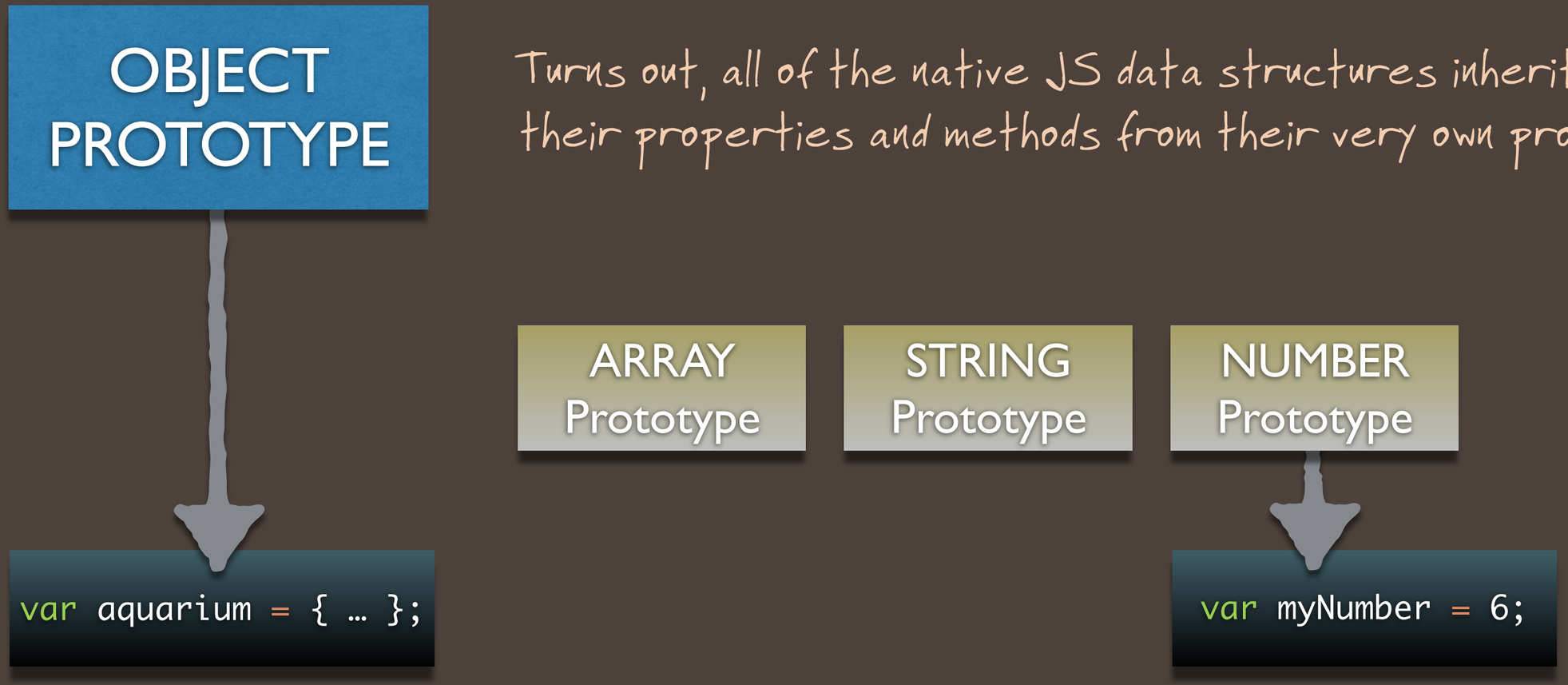
```
var aquarium = { … };
```

```
var myNumber = 6;
```

```
myNumber.
myNumber.
myNumber.
```

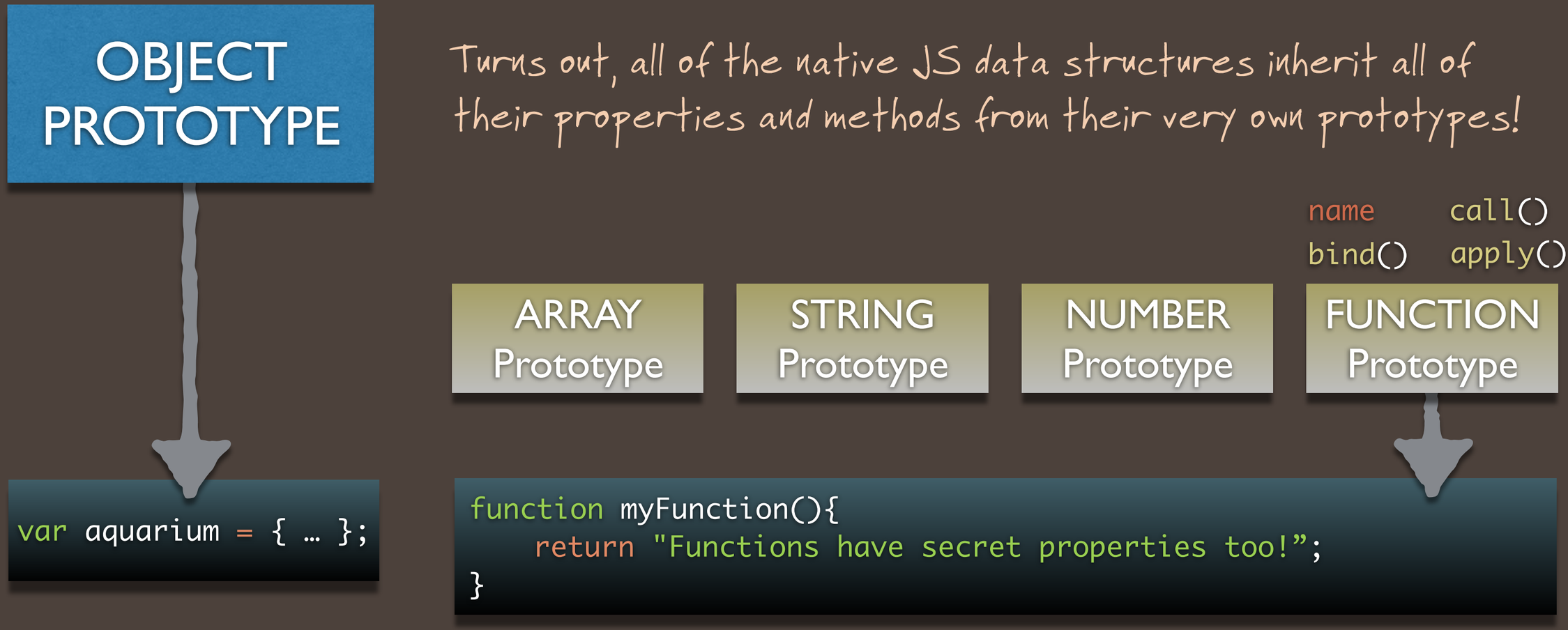
PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

Inheritance helps avoid over-coding multiple properties and methods into similar objects.



PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

Inheritance helps avoid over-coding multiple properties and methods into similar objects.



PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

Inheritance helps avoid over-coding multiple properties and methods into similar objects.

OBJECT PROTOTYPE

Turns out, all of the native JS data structures inherit all of their properties and methods from their very own prototypes!

ARRAY Prototype

STRING Prototype

NUMBER Prototype

FUNCTION Prototype

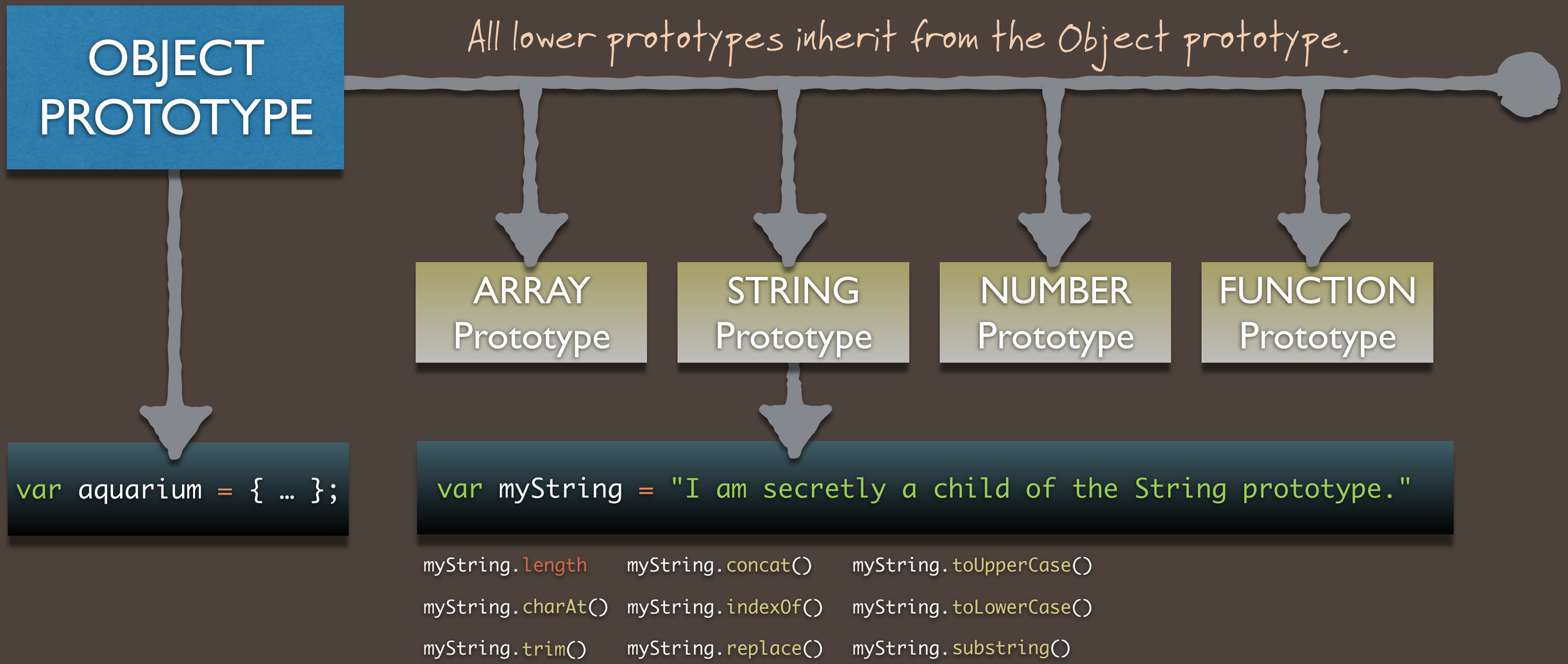
```
var aquarium = { … };
```

```
function myFunction(){
    return "Functions have secret properties too!";
}
```

```
myFunction.name
myFunction.bind()
myFunction.call()
myFunction.apply()
```

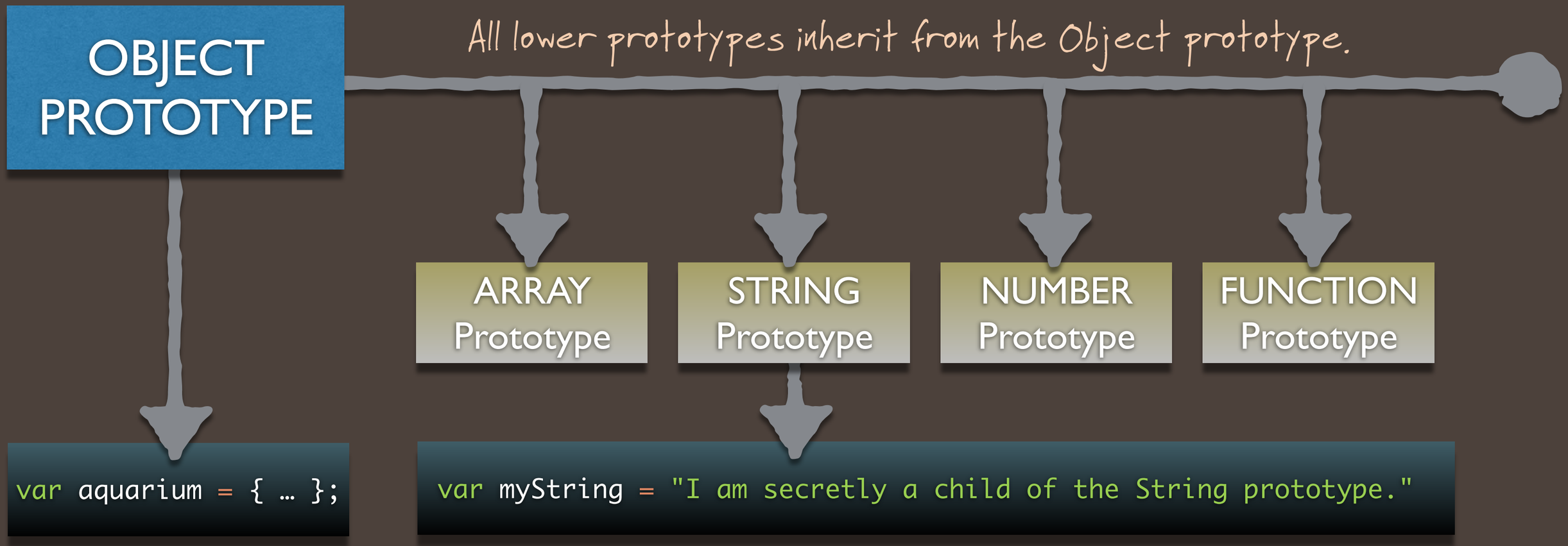
PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

Inheritance helps avoid over-coding multiple properties and methods into similar objects.



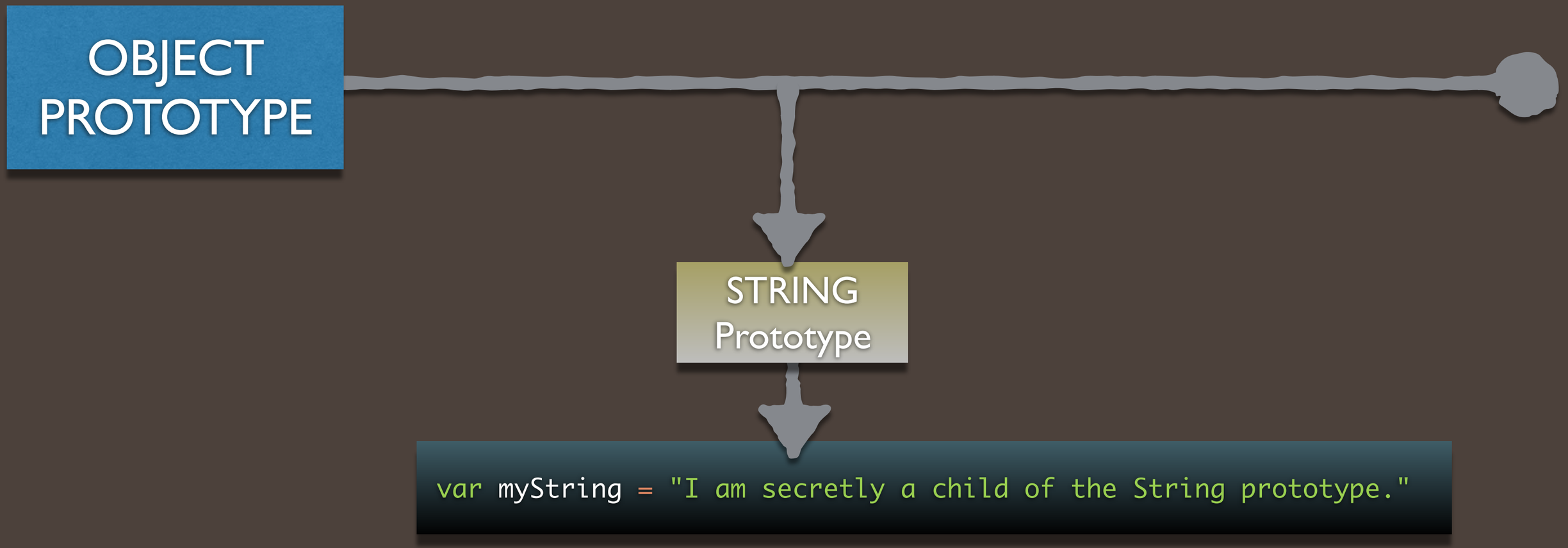
PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

Inheritance helps avoid over-coding multiple properties and methods into similar objects.



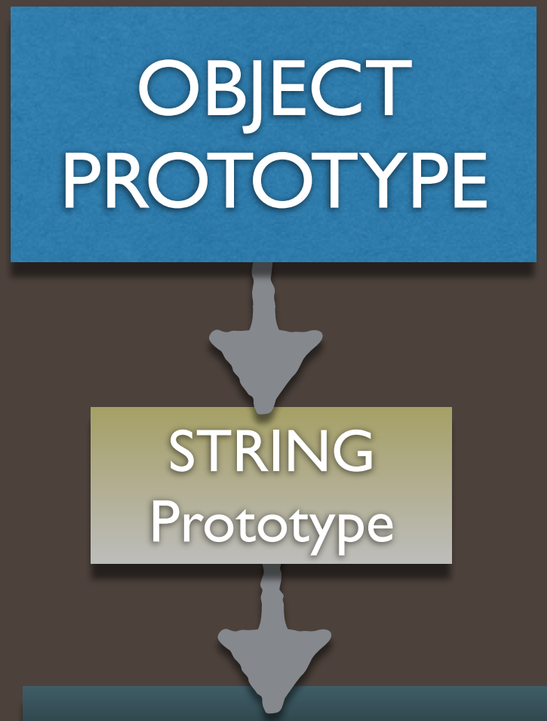
PASSING DOWN PROPERTIES IS CALLED "INHERITANCE"

Inheritance helps avoid over-coding multiple properties and methods into similar objects.



INHERITANCE AVOIDS DUPLICATE MEMORY STORAGE

Though properties are inherited, they are still “owned” by prototypes, not the inheriting Object



`hasOwnProperty()`

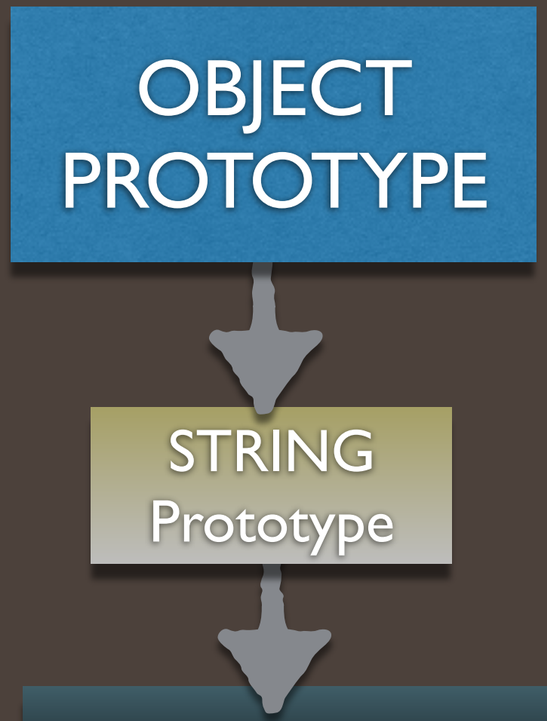
Properties belonging to the Object prototype are actually stored there.

```
var myString = "I am secretly a child of the String prototype."
```

```
myString.
```

INHERITANCE AVOIDS DUPLICATE MEMORY STORAGE

Though properties are inherited, they are still “owned” by prototypes, not the inheriting Object



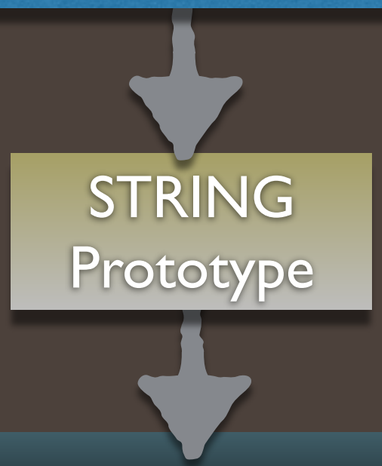

```
hasOwnProperty()
```

```
var myString = "I am secretly a child of the String prototype."
```

```
myString.hasOwnProperty()
```

INHERITANCE AVOIDS DUPLICATE MEMORY STORAGE

Though properties are inherited, they are still “owned” by prototypes, not the inheriting Object



```
hasOwnProperty()
```

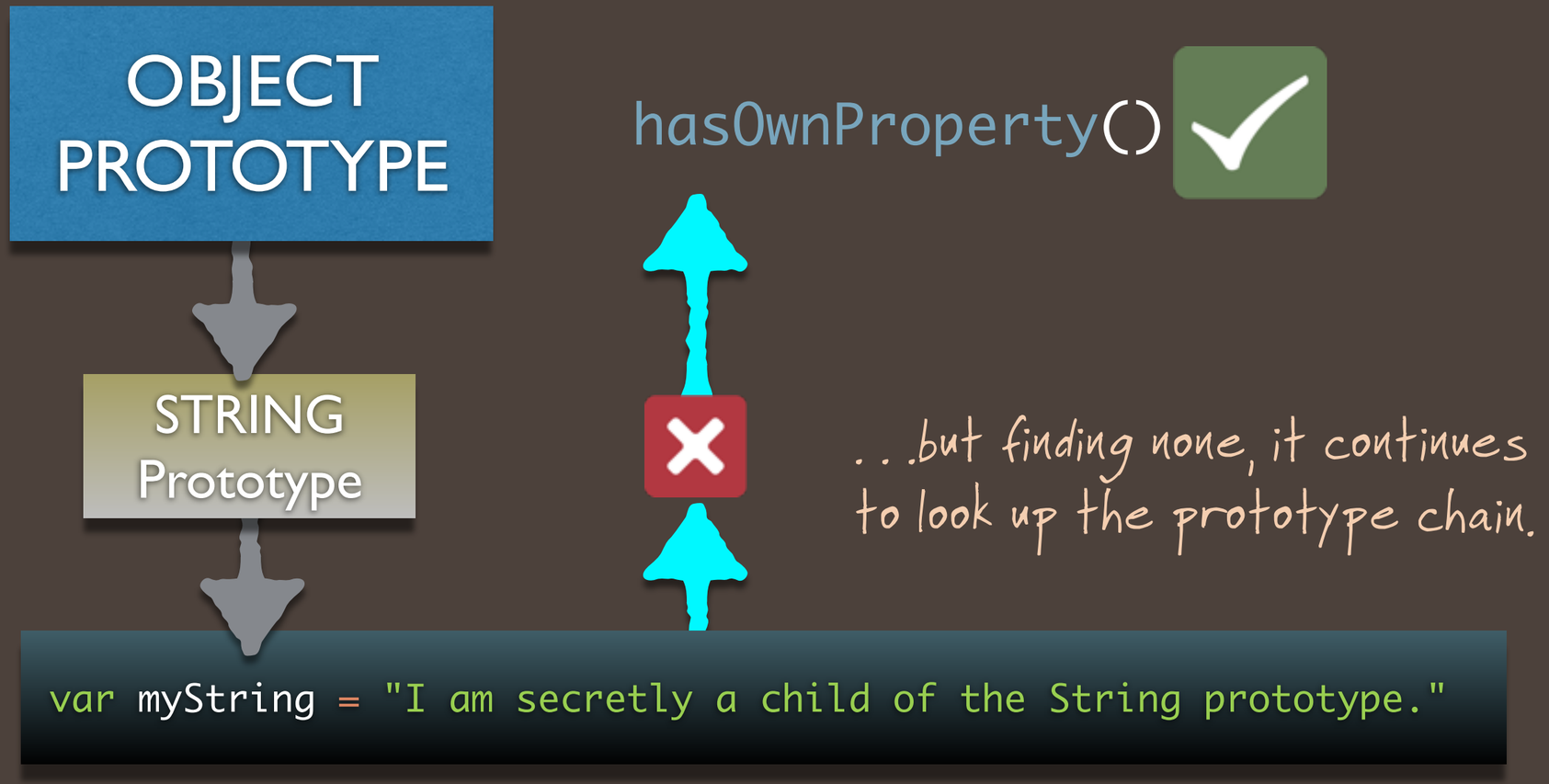
```
var myString = "I am secretly a child of the String prototype."
```

```
myString.hasOwnProperty()
```

When we call this function on a string, the string first looks up to the String prototype to find it...

INHERITANCE AVOIDS DUPLICATE MEMORY STORAGE

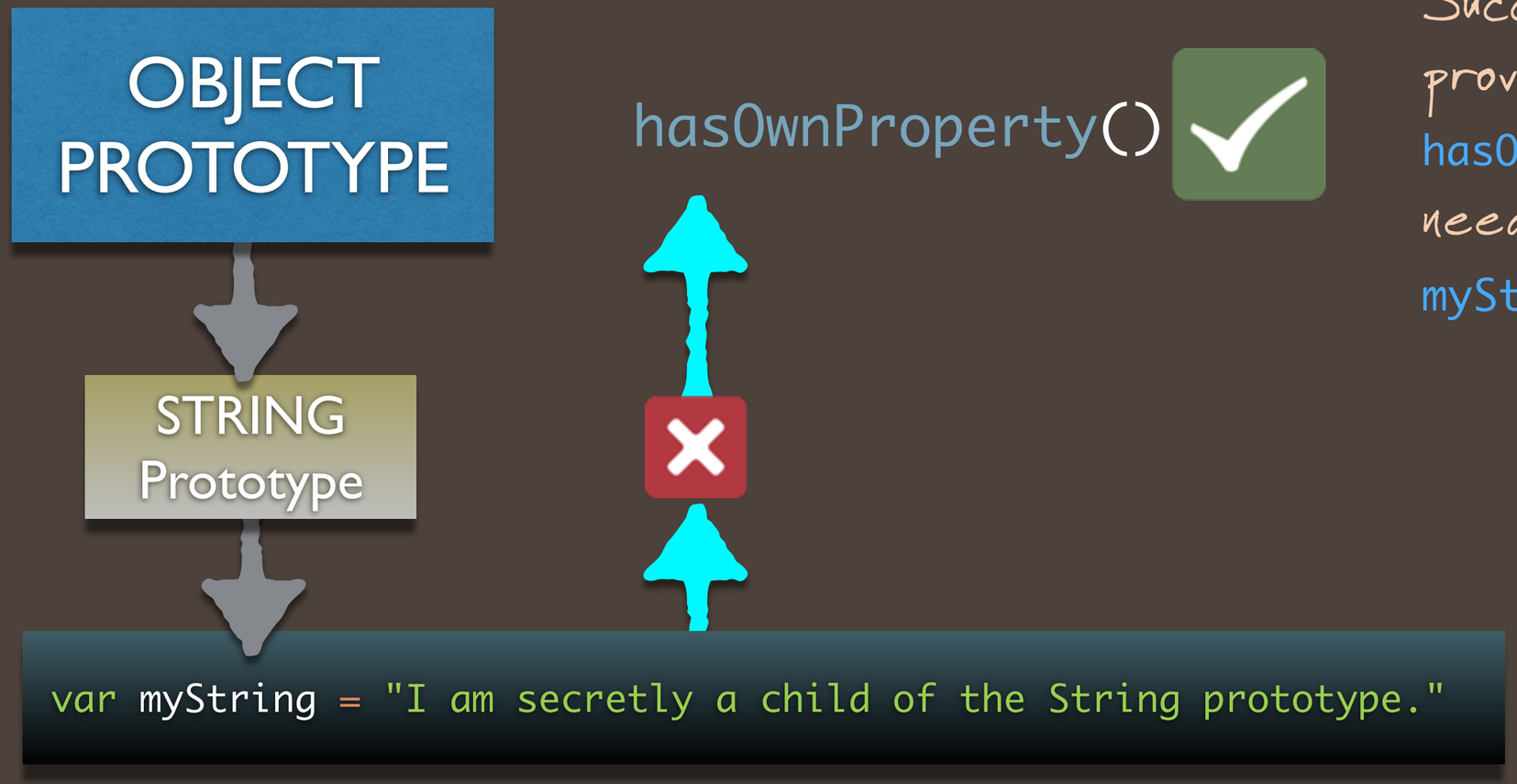
Though properties are inherited, they are still “owned” by prototypes, not the inheriting Object




```
myString.hasOwnProperty()
```

INHERITANCE AVOIDS DUPLICATE MEMORY STORAGE

Though properties are inherited, they are still “owned” by prototypes, not the inheriting Object

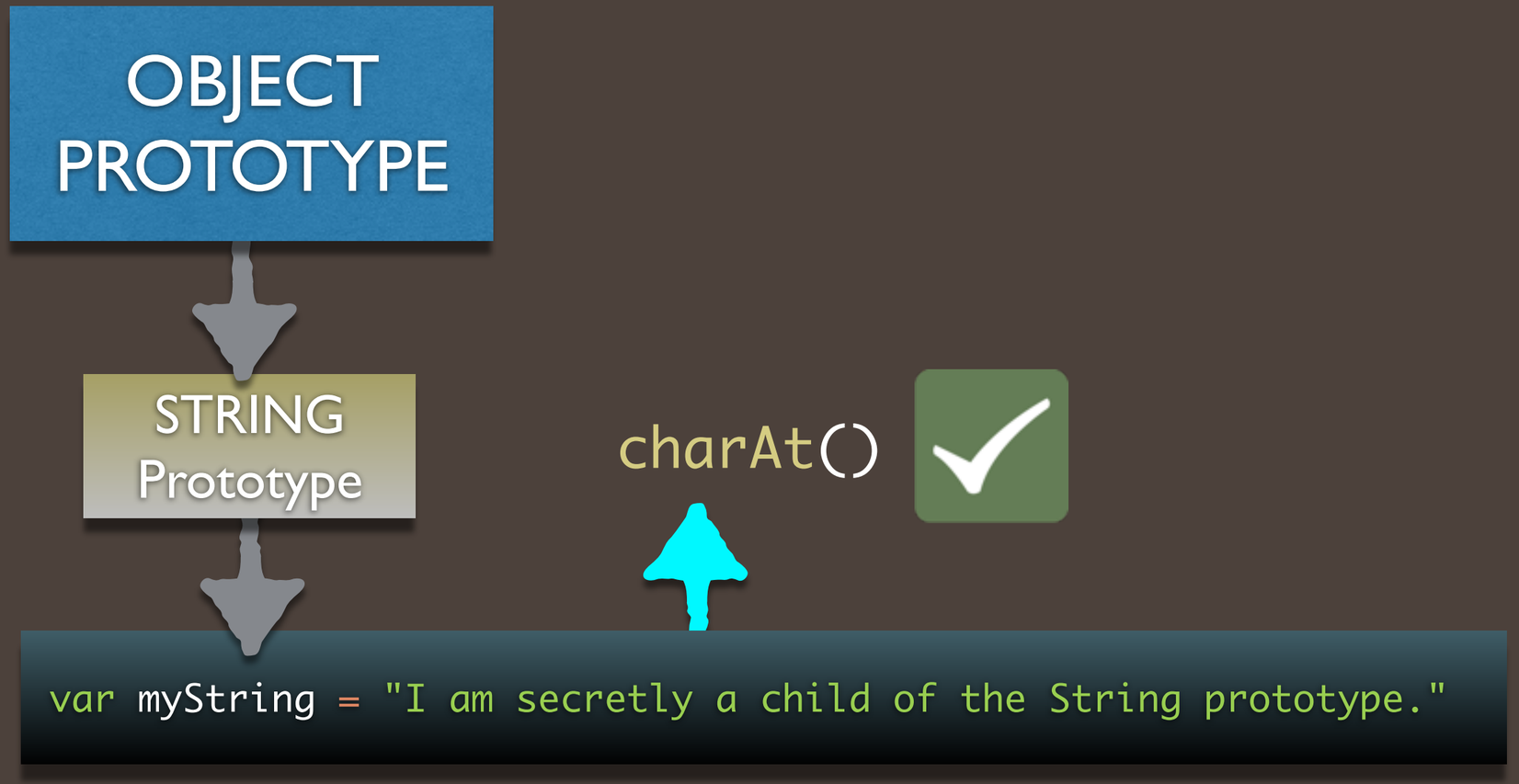


Success! The prototype provides access to the `hasOwnProperty` method without needing it be stored in `myString`.

```
myString.hasOwnProperty()
```

INHERITANCE AVOIDS DUPLICATE MEMORY STORAGE

Though properties are inherited, they are still “owned” by prototypes, not the inheriting Object

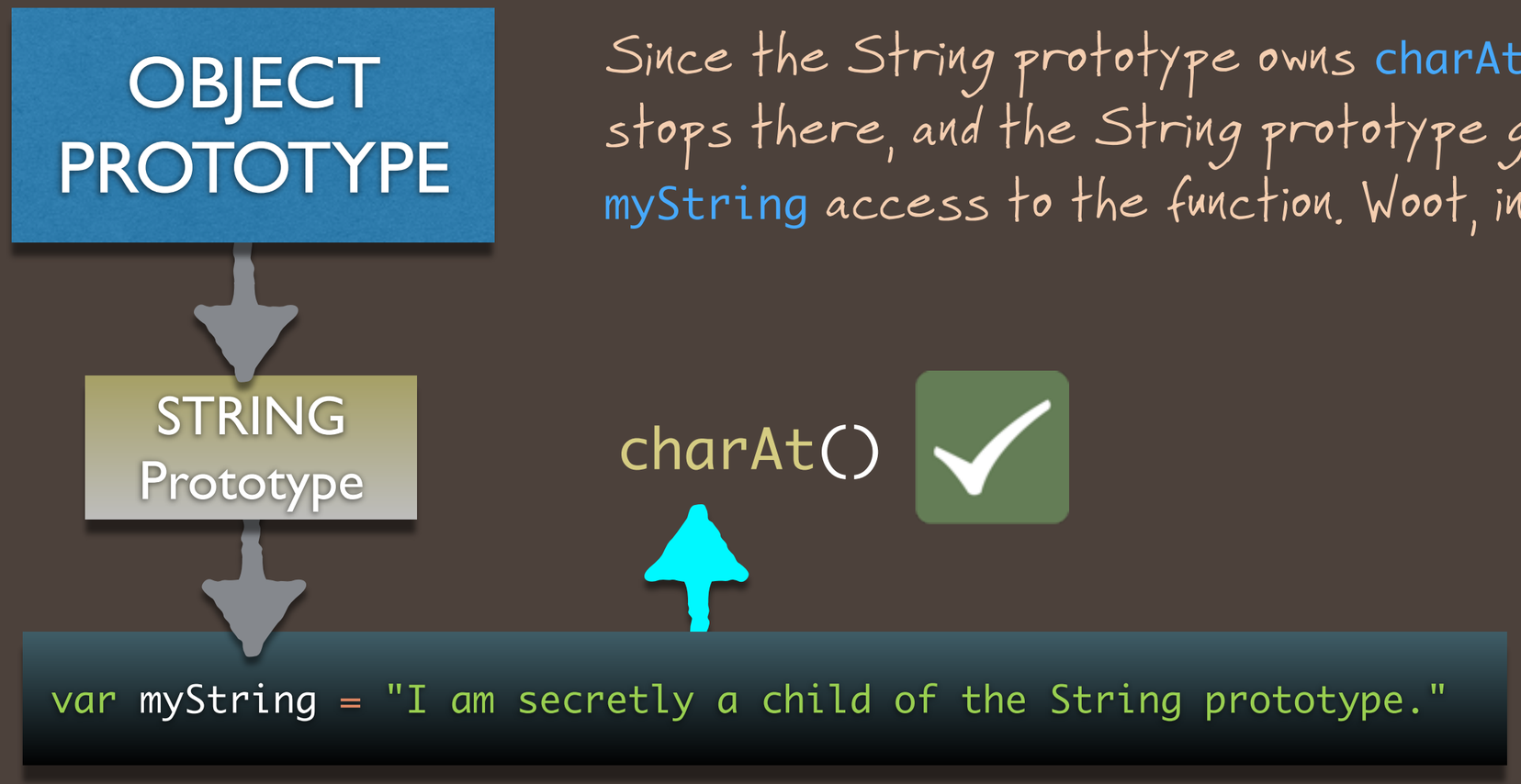



```
myString.charAt()
```

Same goes with a common String property function like charAt...

INHERITANCE AVOIDS DUPLICATE MEMORY STORAGE

Though properties are inherited, they are still “owned” by prototypes, not the inheriting Object




```
myString.charAt()
```

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

# of **A's**	
1	`var witch = "I'll get you, my pretty...and your little dog, too!";`
4	`var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";`
1	`var glinda = "Be gone! Before someone drops a house on you!";`
1	`var dorothy = "There's no place like home.";`
2	`var lion = "Come on, get up and fight, you shivering junkyard!";`
4	`var wizard = "Do not arouse the wrath of the great and powerful Oz!";`
5	`var tinman = "Now I know I have a heart, because it's breaking.";`

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

# of **E's**	
3	`var witch = "I'll get you, my pretty...and your little dog, too!";`
5	`var scarecrow = "Well, some people without brains do an awful lot of talking don't they?"`
7	`var glinda = "Be gone! Before someone drops a house on you!";`
4	`var dorothy = "There's no place like home.";`
3	`var lion = "Come on, get up and fight, you shivering junkyard!";`
5	`var wizard = "Do not arouse the wrath of the great and powerful Oz!";`
5	`var tinman = "Now I know I have a heart, because it's breaking.";`

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

```
function countAll ( string, letter ) { ... }
```

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

countAll

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

STRING Prototype

`countAll`

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

STRING Prototype

countAll

```
dorothy.countAll("h");
```

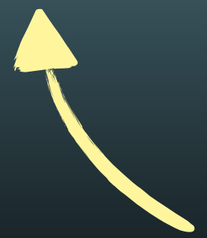
When countAll is part of the prototype, we'll be able to call it from any string! Let's add it in.

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

```
String.prototype
```



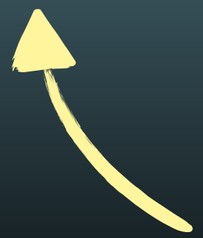
This dot notation finds the prototype for all String values everywhere.

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

```
String.prototype.countAll =
```



To add our function to the String prototype object, we use another dot and name the property with our function's name. This will make it inheritable by all Strings as `countAll`.

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

```
String.prototype.countAll = function ( letter ){

        Since we are giving the function to the
        overarching String prototype, we won't
        need to pass the function a string...
};
```

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

```
String.prototype.countAll = function ( letter ){



};
```

We need to make sure our function can accept a requested letter as a parameter, so that it will return a count for any letter we want.

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

```
String.prototype.countAll = function ( letter ){
    var letterCount = 0;



};
```

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

```
String.prototype.countAll = function ( letter ){
    var letterCount = 0;
    for (var i = 0; i<this.length; i++) {



    }
};
```

Since the string we're interested in will be calling `countAll` on itself, the function should look for its caller's length using `this`.

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

```
String.prototype.countAll = function ( letter ){
    var letterCount = 0;
    for (var i = 0; i<this.length; i++) {
        if ( this.charAt(i).toUpperCase() == letter.toUpperCase() ) {

        }
    }
};
```

We look at the current character in this string...

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

```
String.prototype.countAll =  function ( letter ){
    var letterCount = 0;
    for (var i = 0; i<this.length; i++) {
        if ( this.charAt(i).toUpperCase() == letter.toUpperCase() ) {

        }
    }
};
```

...and convert it to Upper Case to simplify our search. If it's already Upper case, it will stay upper case.

```
"bam!".toUpperCase()
```

→ BAM!

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

```
String.prototype.countAll = function ( letter ){
    var letterCount = 0;
    for (var i = 0; i<this.length; i++) {
        if ( this.charAt(i).toUpperCase() == letter.toUpperCase() ) {

        }
    }
};
```

We compare the converted current character to the converted `letter` to see if we have a match!

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

```
String.prototype.countAll = function ( letter ){
    var letterCount = 0;
    for (var i = 0; i<this.length; i++) {
        if ( this.charAt(i).toUpperCase() == letter.toUpperCase() ) {
            letterCount++;
        }
    }
};
```

If we found a letter, we increment the counter.

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

```
String.prototype.countAll = function ( letter ){
    var letterCount = 0;
    for (var i = 0; i<this.length; i++) {
        if ( this.charAt(i).toUpperCase() == letter.toUpperCase() ) {
            letterCount++;
        }
    }
    return letterCount;
};
```

Lastly, the function returns the final amount.

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

```
String.prototype.countAll = function ( letter ){
    var letterCount = 0;
    for (var i = 0; i<this.length; i++) {
        if ( this.charAt(i).toUpperCase() == letter.toUpperCase() ) {
            letterCount++;
        }
    }
    return letterCount;
};
```

```
witch.countAll("I");
```

→ 2

```
scarecrow.countAll("o");
```

→ 7

ADDING INHERITABLE PROPERTIES TO PROTOTYPES

What if we wanted to add some base values or functionality to ALL objects of a similar type?

```
var witch = "I'll get you, my pretty...and your little dog, too!";
var scarecrow = "Well, some people without brains do an awful lot of talking don't they?";
var glinda = "Be gone! Before someone drops a house on you!";
var dorothy = "There's no place like home.";
var lion = "Come on, get up and fight, you shivering junkyard!";
var wizard = "Do not arouse the wrath of the great and powerful Oz!";
var tinman = "Now I know I have a heart, because it's breaking.";
```

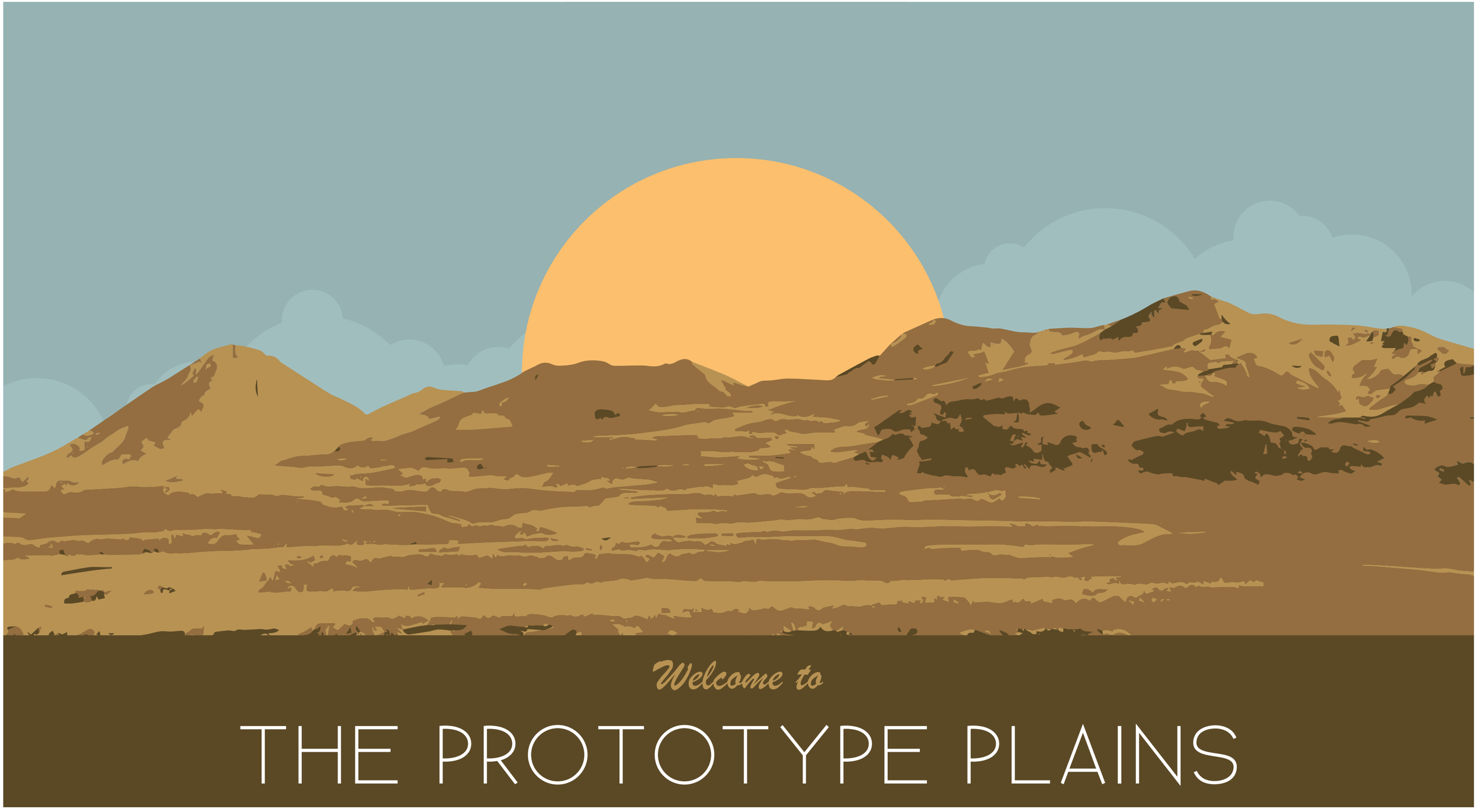
```
String.prototype.countAll = function ( letter ){
    var letterCount = 0;
    for (var i = 0; i<this.length; i++) {
        if ( this.charAt(i).toUpperCase() == letter.toUpperCase() ) {
            letterCount++;
        }
    }
    return letterCount;
};
```

```
lion.countAll("k");
```

→ 1

```
tinman.countAll("N");
```

→ 3


Welcome to
THE PROTOTYPE PLAINS

A SECOND WAY TO BUILD OBJECTS USING OBJECT.CREATE()

Using inheritance, we can create new Objects with our existing Objects as prototypes

```
var shoe = { size: 6, gender: "women", construction: "slipper"};
```

```
var magicShoe = Object.create( shoe );
```

The first argument of the Object.create method will be used as the prototype of the newly created Object.

```
console.log( magicShoe );
```

→ `Object {size: 6, gender: "women", construction: "slipper"}`

The new Object magicShoe inherited all of its properties from shoe, just like we'd expect from a prototype.

A SECOND WAY TO BUILD OBJECTS USING OBJECT.CREATE()

Using inheritance, we can create new Objects with our existing Objects as prototypes

```
var shoe = { size: 6, gender: "women", construction: "slipper"};
```

```
var magicShoe = Object.create( shoe );
```

```
magicShoe.jewels = "ruby";
magicShoe.travelAction = "click heels";
magicShoe.actionsRequired = 3;
```

```
console.log( magicShoe );
```

→ Object {jewels: "ruby", travelAction: "click heels", actionsRequired: 3, size: 6, gender: "women", construction: "slipper"}

A SECOND WAY TO BUILD OBJECTS USING OBJECT.CREATE()

Using inheritance, we can create new Objects with our existing Objects as prototypes

```
var shoe = { size: 6, gender: "women", construction: "slipper"};
```

```
var magicShoe = Object.create( shoe );
```

```
magicShoe.jewels = "ruby";
magicShoe.travelAction = "click heels";
magicShoe.actionsRequired = 3;
```

```
console.log(    shoe    );
```

→ `Object {size: 6, gender: "women", construction: "slipper"}`

EXAMINING THE INHERITANCE WITHIN OUR SHOES

We can use an inherited method to demonstrate our newly created prototype chain


```
Object.prototype.isPrototypeOf( shoe );
```

→ `true`

Remember this property that all JS Objects inherit from the Object prototype? We can use it to find out if any specific Object is a prototype of another.

EXAMINING THE INHERITANCE WITHIN OUR SHOES

We can use an inherited method to demonstrate our newly created prototype chain


```
Object.prototype.isPrototypeOf( shoe );
```

→ `true`

```
shoe
```

EXAMINING THE INHERITANCE WITHIN OUR SHOES

We can use an inherited method to demonstrate our newly created prototype chain


```
Object.prototype.isPrototypeOf( shoe );
```

→ `true`

```
shoe.isPrototypeOf( magicShoe );
```

→ `true`

Since we used `shoe` as the prototype for `magicShoe`, the `isPrototypeOf` property returns true for this line of code as well.

EXAMINING THE INHERITANCE WITHIN OUR SHOES

We can use an inherited method to demonstrate our newly created prototype chain


```
Object.prototype.isPrototypeOf( shoe );
```

→ `true`

```
shoe.isPrototypeOf( magicShoe );
```

→ `true`

```
magicShoe.isPrototypeOf( shoe );
```

→ `false`

EXAMINING THE INHERITANCE WITHIN OUR SHOES

We can use an inherited method to demonstrate our newly created prototype chain

OBJECT PROTOTYPE

```
Object.prototype.isPrototypeOf( magicShoe );
```

→ `true`

The `isPrototypeOf` method will look upward through the entire hierarchy (the prototype "chain") to see whether the `Object.prototype` Object is a prototypical "ancestor" of `magicShoe`.

WHAT IF THERE WERE OTHER KINDS OF SHOES?

Could we use the same prototype to create boots, sneakers, sandals, and...uh...?

OBJECT PROTOTYPE

```
var shoe = { size: 6, gender: "women", construction: "slipper"};
```

```
var mensBoot = Object.create( shoe );
```

```
console.log(mensBoot);
```

✗ → `Object {size: 6, gender: "women", construction: "slipper"}`

?

WHAT IF THERE WERE OTHER KINDS OF SHOES?

Could we use the same prototype to create boots, sneakers, sandals, and...uh...?

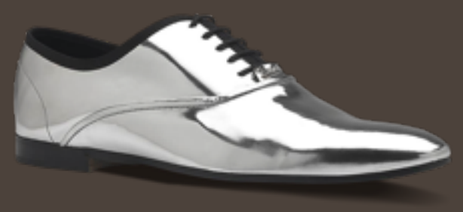
OBJECT PROTOTYPE

```
var shoe = { size: 6, gender: "women", construction: "slipper"};
```

```
var mensBoot = Object.create( shoe );
```

```
console.log(mensBoot);
```

→ `Object {size: 6, gender: "women", construction: "slipper"}`



WHAT IF THERE WERE OTHER KINDS OF SHOES?

Could we use the same prototype to create boots, sneakers, sandals, and...uh...?

WE MIGHT BUILD A PROTOTYPE WITH EMPTY PROPERTIES...

With a generic “shoe”, we could build all of our shoes, and assign property values later.

```
var shoe = { size: undefined, gender: undefined, construction: undefined };
```

All this object has is a bunch of property names with no values. Now what?

```
var mensBoot = Object.create( shoe );
```

```
mensBoot.size = 12;
mensBoot.gender = "men";
mensBoot.construction = "boot";
```

```
var flipFlop = Object.create( shoe );
```

```
flipFlop.size = 5;
flipFlop.gender = "women";
flipFlop.construction = "flipflop";
```

FIRST, WE DETERMINE COMMON PROPERTIES OF A SHOE CLASS

A class is a set of Objects that all share and inherit from the same basic prototype.





All Shoes

Some Shoes

```
size
```

FIRST, WE DETERMINE COMMON PROPERTIES OF A SHOE CLASS

A class is a set of Objects that all share and inherit from the same basic prototype.





All Shoes

`size`

Some Shoes

`color`

FIRST, WE DETERMINE COMMON PROPERTIES OF A SHOE CLASS

A class is a set of Objects that all share and inherit from the same basic prototype.





All Shoes

```
size
color
```

Some Shoes

```
gender
```

FIRST, WE DETERMINE COMMON PROPERTIES OF A SHOE CLASS

A class is a set of Objects that all share and inherit from the same basic prototype.





All Shoes

```
size
color
gender
```

Some Shoes

```
construction
```

FIRST, WE DETERMINE COMMON PROPERTIES OF A SHOE CLASS

A class is a set of Objects that all share and inherit from the same basic prototype.





All Shoes

```
size
color
gender
construction
```

Some Shoes

```
laceColor
```

FIRST, WE DETERMINE COMMON PROPERTIES OF A SHOE CLASS

A class is a set of Objects that all share and inherit from the same basic prototype.





All Shoes

```
size
color
gender
construction
```

```
laceUp()
```

Some Shoes

```
laceColor
```

FIRST, WE DETERMINE COMMON PROPERTIES OF A SHOE CLASS

A class is a set of Objects that all share and inherit from the same basic prototype.





All Shoes

```
size
color
gender
construction
```

```
jewels
```

Some Shoes

```
laceColor
laceUp()
```

FIRST, WE DETERMINE COMMON PROPERTIES OF A SHOE CLASS

A class is a set of Objects that all share and inherit from the same basic prototype.





All Shoes

```
size
color
gender
construction
```

```
bowPosition
```

Some Shoes

```
laceColor
laceUp()
jewels
```

FIRST, WE DETERMINE COMMON PROPERTIES OF A SHOE CLASS

A class is a set of Objects that all share and inherit from the same basic prototype.





All Shoes

```
size
color
gender
construction
```

```
putOn()
```

Some Shoes

```
laceColor
laceUp()
jewels
bowPosition
```

FIRST, WE DETERMINE COMMON PROPERTIES OF A SHOE CLASS

A class is a set of Objects that all share and inherit from the same basic prototype.





All Shoes

```
size
color
gender
construction
putOn()
```

```
dimensionalTravel()
```

Some Shoes

```
laceColor
laceUp()
jewels
bowPosition
```

FIRST, WE DETERMINE COMMON PROPERTIES OF A SHOE CLASS

A class is a set of Objects that all share and inherit from the same basic prototype.





All Shoes

```
size
color
gender
construction
putOn()
```

```
takeOff()
```

Some Shoes

```
laceColor
laceUp()
jewels
bowPosition
dimensionalTravel()
```

FIRST, WE DETERMINE COMMON PROPERTIES OF A SHOE CLASS

A class is a set of Objects that all share and inherit from the same basic prototype.





All Shoes

`size`
`color`
`gender`
`construction`
`putOn()`
`takeOff()`

With a good set of common properties we can expect ALL shoes to have, we're ready to build a Constructor for our class.

Since not all shoes have these properties, they shouldn't go in the prototype

Some Shoes

`laceColor`
`laceUp()`
`jewels`
`bowPosition`
`dimensionalTravel()`

BUILDING A CONSTRUCTOR FUNCTION FOR A SHOE OBJECT

A constructor allows us to set up inheritance while also assigning specific property values.





All Shoes

size
color
gender
construction
putOn()
takeOff()

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {

}
```

Capitalizing this function's name distinguishes it as a maker of an entire "Class" of Objects... a constructor.

BUILDING A CONSTRUCTOR FUNCTION FOR A SHOE OBJECT

A constructor allows us to set up inheritance while also assigning specific property values.





All Shoes

size
color
gender
construction
putOn()
takeOff()

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {



}
```

Each of these parameters will be specific values for a specific kind of Shoe. The constructor function will "construct" a new "instance" of a Shoe and assign these values to it.

BUILDING A CONSTRUCTOR FUNCTION FOR A SHOE OBJECT

A constructor allows us to set up inheritance while also assigning specific property values.





All Shoes

size
color
gender
construction
putOn()
takeOff()

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {

  this.size = shoeSize;
  this.color = shoeColor;
  this.gender = forGender;
  this.construction = constructStyle;


}
```

The this keyword inside a constructor will automatically refer to the new instance of the class that is being made.

BUILDING A CONSTRUCTOR FUNCTION FOR A SHOE OBJECT

A constructor allows us to set up inheritance while also assigning specific property values.





All Shoes

size
color
gender
construction
putOn()
takeOff()

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {

  this.size = shoeSize;
  this.color = shoeColor;
  this.gender = forGender;
  this.construction = constructStyle;

  this.putOn = function () { alert("Shoe's on!"); };
  this.takeOff = function () { alert("Uh, what's that smell?"); };
}
```

These functions will now be common to all shoes.

BUILDING A CONSTRUCTOR FUNCTION FOR A SHOE OBJECT

A constructor allows us to set up inheritance while also assigning specific property values.





```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {

  this.size = shoeSize;
  this.color = shoeColor;
  this.gender = forGender;
  this.construction = constructStyle;

  this.putOn = function () { alert("Shoe's on!"); };
  this.takeOff = function () { alert("Uh, what's that smell?"); };
}
```

BUILDING A CONSTRUCTOR FUNCTION FOR A SHOE OBJECT

A constructor allows us to set up inheritance while also assigning specific property values.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {

  this.size = shoeSize;
  this.color = shoeColor;
  this.gender = forGender;
  this.construction = constructStyle;

  this.putOn = function () { alert("Shoe's on!"); };
  this.takeOff = function () { alert("Uh, what's that smell?"); };
}
```

LET'S USE OUR SHOE CONSTRUCTOR!

JavaScript's 'new' keyword produces a new Object of the class, or "instantiates" the class.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
   this.putOn = function () { alert("Shoe's on!"); };
   this.takeOff = function () { alert("Uh, what's that smell?"); };
}
```

```
var beachShoe = new Shoe( 10, "blue", "women", "flipflop" );
```

The new keyword asks to build a new instance of something. What something? A Shoe, in this case.

LET'S USE OUR SHOE CONSTRUCTOR!

JavaScript's 'new' keyword produces a new Object of the class, or "instantiates" the class.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
   this.putOn = function () { alert("Shoe's on!"); };
   this.takeOff = function () { alert("Uh, what's that smell?"); };
}
```

```
var beachShoe = new Shoe( 10, "blue", "women", "flipflop" );
```

LET'S USE OUR SHOE CONSTRUCTOR!

JavaScript's 'new' keyword produces a new Object of the class, or "instantiates" the class.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
   this.putOn = function () { alert("Shoe's on!"); };
   this.takeOff = function () { alert("Uh, what's that smell?"); };
}
```

```
var beachShoe = new Shoe( 10, "blue", "women", "flipflop" );
console.log( beachShoe );
```

→

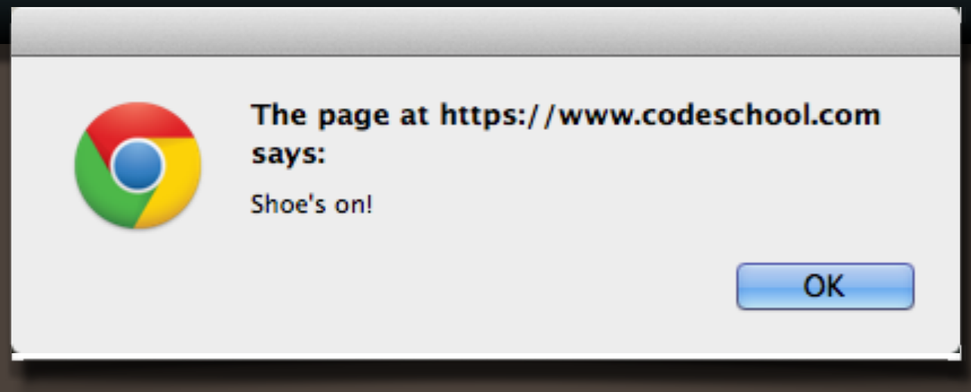
```
Shoe {size: 10,
      color: "blue",
      gender: "women",
      construction: "flipflop",
      putOn: function () {…},
      takeOff: function () {…}}
```

LET'S USE OUR SHOE CONSTRUCTOR!

JavaScript's 'new' keyword produces a new Object of the class, or "instantiates" the class.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
   this.putOn = function () { alert("Shoe's on!"); };
   this.takeOff = function () { alert("Uh, what's that smell?"); };
}
```

```
var beachShoe = new Shoe( 10, "blue", "women", "flipflop" );
beachShoe.putOn();
```



LET'S USE OUR SHOE CONSTRUCTOR!

JavaScript's 'new' keyword produces a new Object of the class, or "instantiates" the class.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
   this.putOn = function () { alert("Shoe's on!"); };
   this.takeOff = function () { alert("Uh, what's that smell?"); };
}
```

```
var beachShoe = new Shoe( 10, "blue", "women", "flipflop" );
beachShoe.straps = 2;
```

Later, we could add properties that are more shoe-specific.

LET'S USE OUR SHOE CONSTRUCTOR!

JavaScript's 'new' keyword produces a new Object of the class, or "instantiates" the class.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
   this.putOn = function () { alert("Shoe's on!"); };
   this.takeOff = function () { alert("Uh, what's that smell?"); };
}
```

```
var beachShoe = new Shoe( 10, "blue", "women", "flipflop" );
beachShoe.straps = 2;
```

Hold on, where's my efficient inheritance?

LET'S USE OUR SHOE CONSTRUCTOR!

JavaScript's 'new' keyword produces a new Object of the class, or "instantiates" the class.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
   this.putOn = function () { alert("Shoe's on!"); };
   this.takeOff = function () { alert("Uh, what's that smell?"); };
}
```

Since these functions don't change between any Shoe, we should put them in a Shoe prototype so that they are stored efficiently in only one location that all Shoes can access.

ASSIGNING A PROTOTYPE TO A CONSTRUCTOR

By setting a constructor's prototype property, every new instance will refer to it for extra properties!

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
   this.putOn = function () { alert("Shoe's on!"); };
   this.takeOff = function () { alert("Uh, what's that smell?"); };
}
```

```
Shoe.prototype = {

};
```

We build a new, secret Object within the constructor function's prototype property! This will tell every created Shoe to inherit from that Object.

Array.prototype

String.prototype

Object.prototype

ASSIGNING A PROTOTYPE TO A CONSTRUCTOR

By setting a constructor's prototype property, every new instance will refer to it for extra properties!

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
   this.putOn = function () { alert("Shoe's on!"); };
   this.takeOff = function () { alert("Uh, what's that smell?"); };
}
```

```
Shoe.prototype = {
    putOn: function () { alert ("Shoe's on, dood!"); },
    takeOff: function () { alert ("Uh, what's that smell?"); }
};
```

Just like building an Object literal, we set each function as a property in the Prototype Object.

ASSIGNING A PROTOTYPE TO A CONSTRUCTOR

By setting a constructor's prototype property, every new instance will refer to it for extra properties!

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;



}
```

```
Shoe.prototype = {

    putOn: function () { alert ("Shoe's on!"); },
    takeOff: function () { alert ("Uh, what's that smell?"); }
};
```

ASSIGNING A PROTOTYPE TO A CONSTRUCTOR

By setting a constructor's prototype property, every new instance will refer to it for extra properties!

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
}
```

```
Shoe.prototype = {
    putOn: function () { alert ("Shoe's on!"); },
    takeOff: function () { alert ("Uh, what's that smell?"); }
};
```

ASSIGNING A PROTOTYPE TO A CONSTRUCTOR

By setting a constructor's prototype property, every new instance will refer to it for extra properties!

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
}
```

```
Shoe.prototype = {
    putOn: function () { alert ("Shoe's on!"); },
    takeOff: function () { alert ("Uh, what's that smell?"); }
};
```

```
var beachShoe = new Shoe( 10, "blue", "women", "flipflop" );
console.log(beachShoe.gender);
```

First the compiler looks in the beachShoe Object itself, and finds it! We get the specific value given to the constructor when beachShoe was created.

→ women

ASSIGNING A PROTOTYPE TO A CONSTRUCTOR

By setting a constructor's prototype property, every new instance will refer to it for extra properties!

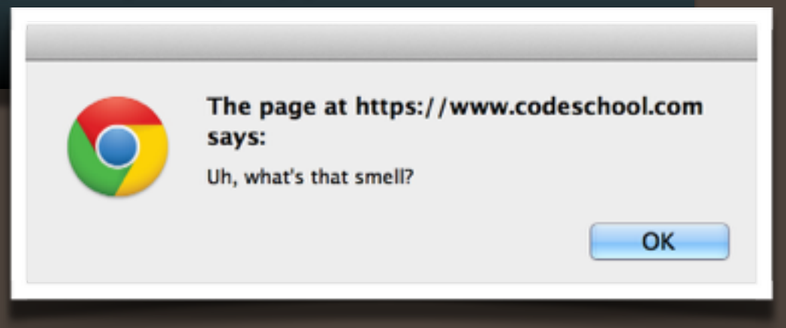
```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
}
```

```
Shoe.prototype = {
    putOn: function () { alert ("Shoe's on!"); },
    takeOff: function () { alert ("Uh, what's that smell?"); }
};
```

```
var beachShoe = new Shoe( 10, "blue", "women", "flipflop" );
beachShoe.takeOff();
```

First the compiler looks in the beachShoe Object itself... but finds nothing.

Then it checks the prototype... and there's the takeOff property!



ASSIGNING A PROTOTYPE TO A CONSTRUCTOR

By setting a constructor's prototype property, every new instance will refer to it for extra properties!

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
}
```

```
Shoe.prototype = {
    putOn: function () { alert ("Shoe's on!"); },
    takeOff: function () { alert ("Uh, what's that smell?"); }
};
```

```
var beachShoe = new Shoe( 10, "blue", "women", "flipflop" );
beachShoe.hasOwnProperty("construction");
```

OBJECT PROTOTYPE

hasOwnProperty()

Neither the beachShoe Object nor the Shoe prototype have a property called hasOwnProperty, so the compiler proceeds up the prototype chain to the Object prototype, where it finds the called function.

ASSIGNING A PROTOTYPE TO A CONSTRUCTOR

By setting a constructor's prototype property, every new instance will refer to it for extra properties!

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
}
```

```
Shoe.prototype = {
    putOn: function () { alert ("Shoe's on!"); },
    takeOff: function () { alert ("Uh, what's that smell?"); }
};
```

```
var beachShoe = new Shoe( 10, "blue", "women", "flipflop" );
beachShoe.hasOwnProperty("construction");
```

The hasOwnProperty function now looks to see if beachShoe has its OWN property (not inherited) called "construction", which it does.

→ true

OBJECT PROTOTYPE

hasOwnProperty()

PROTOTYPES CAN ALSO REFER BACK TO THE INSTANCE!

We can modify the message functions in our prototype to use the data values in the calling instance.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
}
```

```
Shoe.prototype = {
    putOn: function () { alert ("Shoe's on!"); },
    takeOff: function () { alert ("Uh, what's that smell?"); }
};
```

PROTOTYPES CAN ALSO REFER BACK TO THE INSTANCE!

We can modify the message functions in our prototype to use the data values in the calling instance.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
}
```

```
Shoe.prototype = {
    putOn:
    takeOff:
};
```

PROTOTYPES CAN ALSO REFER BACK TO THE INSTANCE!

We can modify the message functions in our prototype to use the data values in the calling instance.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
}
```

```
Shoe.prototype = {
    putOn:  function () { alert ("Your " + this.construction + "'s" + "on!"); },
    takeOff:  function () { alert ("Phew! Somebody's size " + this.size + "'s" +
                                   " are fragrant! "); }
};
```

The this keyword looks "back down" to the particular Shoe that called the inherited function, and pulls property data from it.

PROTOTYPES CAN ALSO REFER BACK TO THE INSTANCE!

We can modify the message functions in our prototype to use the data values in the calling instance.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
}
```

```
Shoe.prototype = {
    putOn:  function () { alert ("Your " + this.construction + "'s" + "on!"); },
    takeOff:  function () { alert ("Phew! Somebody's size " + this.size + "'s" +
                                   " are fragrant! "); }
};
```

PROTOTYPES CAN ALSO REFER BACK TO THE INSTANCE!

We can modify the message functions in our prototype to use the data values in the calling instance.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
}
```

```
size: 10
color: "blue"
gender: "women"
construction: "flipflop"
```

```
Shoe.prototype = {
    putOn:  function () { alert ("Your " + this.construction + "'s" + "on!"); },
    takeOff:  function () { alert ("Phew! Somebody's size " + this.size + "'s" +
                                   " are fragrant! "); }
};
```

PROTOTYPES CAN ALSO REFER BACK TO THE INSTANCE!

We can modify the message functions in our prototype to use the data values in the calling instance.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
}
```

```
size: 10
color: "blue"
gender: "women"
construction: "flipflop"
```

```
Shoe.prototype = {
    putOn:  function () { alert ("Your " + this.construction + "'s" + "on!"); },
    takeOff:  function () { alert ("Phew! Somebody's size " + this.size + "'s" +
                                   " are fragrant! "); }
};
```

```
beachShoe.putOn();
```

PROTOTYPES CAN ALSO REFER BACK TO THE INSTANCE!

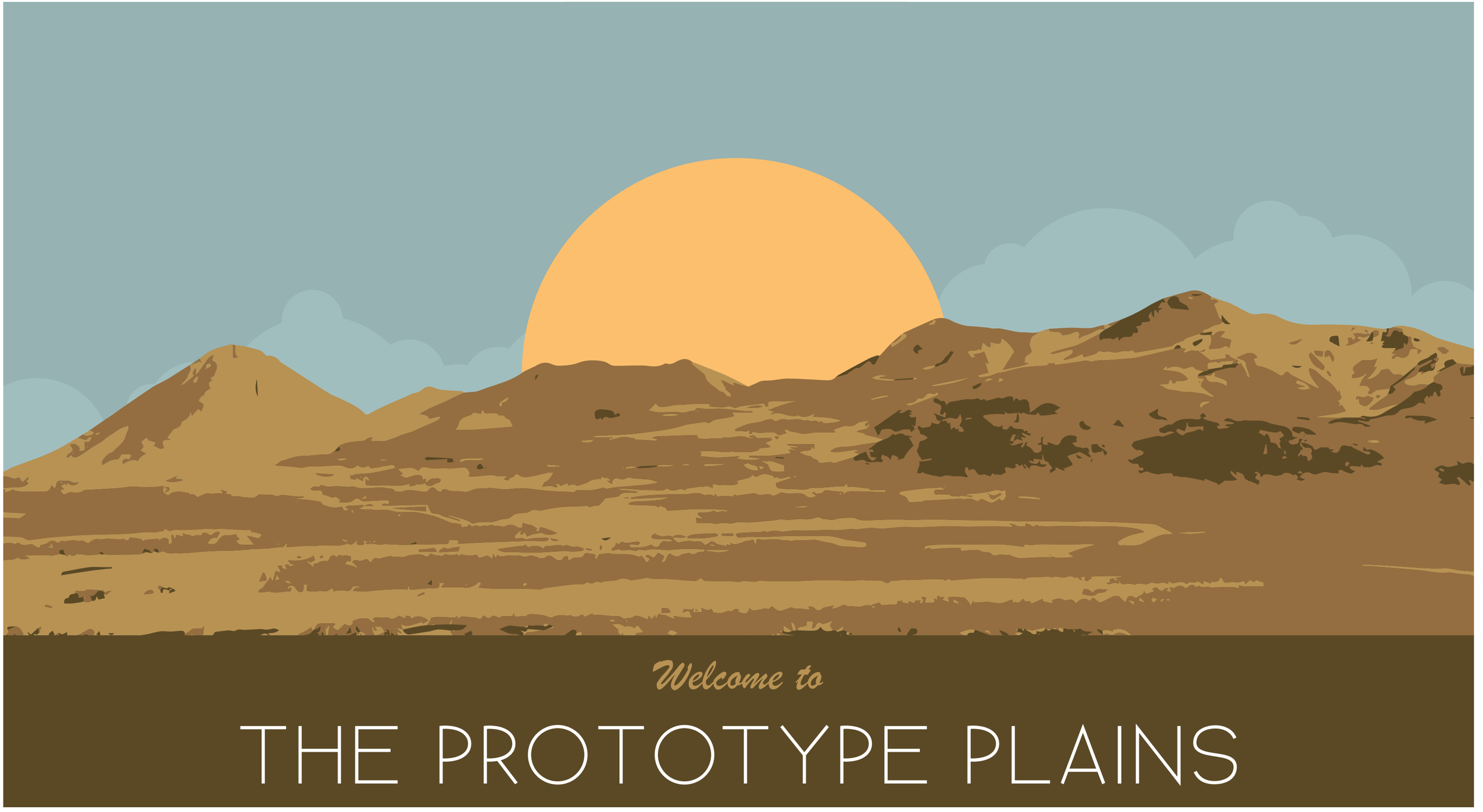
We can modify the message functions in our prototype to use the data values in the calling instance.

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
   this.size = shoeSize;
   this.color = shoeColor;
   this.gender = forGender;
   this.construction = constructStyle;
}
```

```
size: 10
color: "blue"
gender: "women"
construction: "flipflop"
```

```
Shoe.prototype = {
    putOn:  function () { alert ("Your " + this.construction + "'s" + "on!"); },
    takeOff:  function () { alert ("Phew! Somebody's size " + this.size + "'s" +
                                   " are fragrant! "); }
};
```

```
beachShoe.takeOff();
```


Welcome to
THE PROTOTYPE PLAINS

USEFUL PROPERTIES IN THE OBJECT PROTOTYPE

We've seen a few properties inherited from the Object.prototype...let's test a few more.

```
var x = 4;
var y = "4";
```

```
x.valueOf();
```

→ 4

```
y.valueOf();
```

→ "4"

The "value" in `valueOf()` isn't looking for numbers necessarily, but instead returns whatever primitive type is associated with the object.

```
x.valueOf() == y.valueOf();
```

→ true

Be careful! The `==` tries to help us out by using "type coercion," which turns a number contained within a string into an actual number. Here, the `"4"` we got back from `y.valueOf()` became `4` when the `==` examined it.

USEFUL PROPERTIES IN THE OBJECT PROTOTYPE

We've seen a few properties inherited from the Object.prototype...let's test a few more.

```
var x = 4;
var y = "4";
```

```
x.valueOf();
```

→ 4

```
y.valueOf();
```

→ "4"

The "value" in `valueOf()` isn't looking for numbers necessarily, but instead returns whatever primitive type is associated with the object.

```
x.valueOf() == y.valueOf();
```

→ true

```
x.valueOf() === y.valueOf();
```

→ false

The `===` operator does NOT ignore the type of the value, and gives us a more detailed interpretation of equality. JavaScript experts often prefer this comparator exclusively over `==` for this reason.

USEFUL PROPERTIES IN THE OBJECT PROTOTYPE

We've seen a few properties inherited from the Object.prototype...let's test a few more.

```
var x = 4;
var y = "4";
```

```
var a = [ 3, "blind", "mice" ];
```

```
var b = new Number(6);
```

```
x.valueOf();
```

→ 4

```
y.valueOf();
```

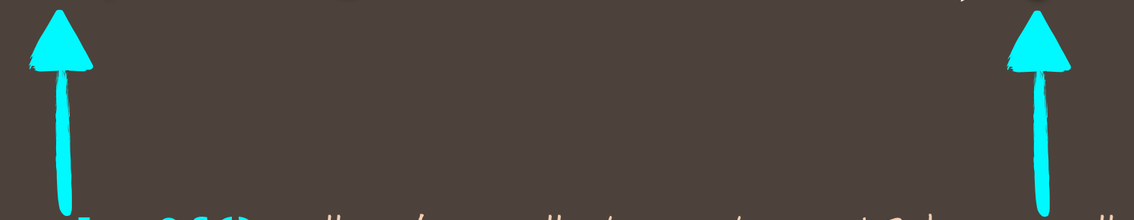
→ "4"

```
a.valueOf();
```

→ [3, "blind", "mice"]

```
b.valueOf();
```

→ 6



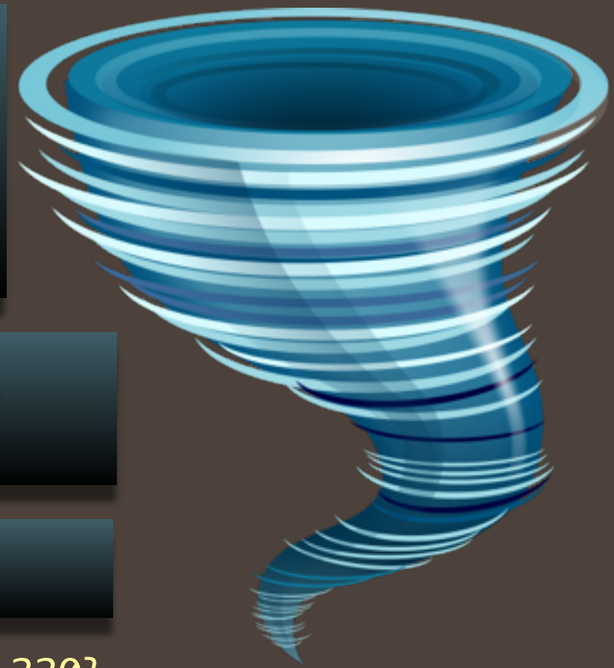
Most `valueOf()` calls, when called on ordinary JS types, will not produce anything different than you might expect when just logging out the Object. Don't worry, it gets more interesting...

VALUEOF( ) ON CUSTOM OBJECTS

What happens when we call valueOf( ) on an object we make ourselves?

```
var Tornado = function (category, affectedAreas, windGust) {
    this.category = category;
    this.affectedAreas = affectedAreas;
    this.windGust = windGust;
};
```

Constructors can be function expressions, too!



```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
```

```
twister.valueOf();
```

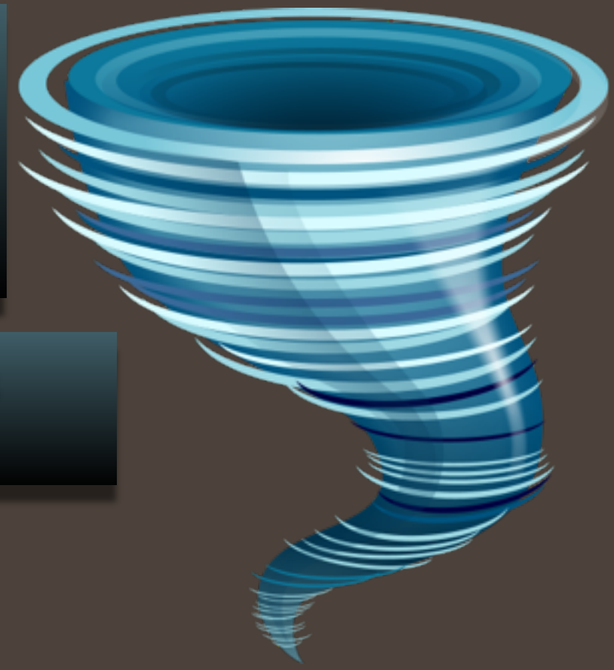
→ `Tornado {category: "F5", affectedAreas: Array[3], windGust: 220}`

The `valueOf()` function for custom Objects just defaults to a list of their properties, just like logging them out.

OVERRIDING PROTOTYPAL PROPERTIES

Many situations require special functionality that's different from the first available property

```
var Tornado = function (category, affectedAreas, windGust) {
    this.category = category;
    this.affectedAreas = affectedAreas;
    this.windGust = windGust;
};
```



```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
```

```
Tornado.prototype.valueOf = function( ) {
   var sum = 0;
   for (var i = 0; i < this.affectedAreas.length; i++) {
       sum += this.affectedAreas[i][1];
   }
   return sum;
};
```

When overriding the property, we want to modify the Tornado prototype rather than the Object prototype! We only want to change `valueOf()` for Tornado's, not all Objects!

OVERRIDING PROTOTYPAL PROPERTIES

Many situations require special functionality that's different from the first available property

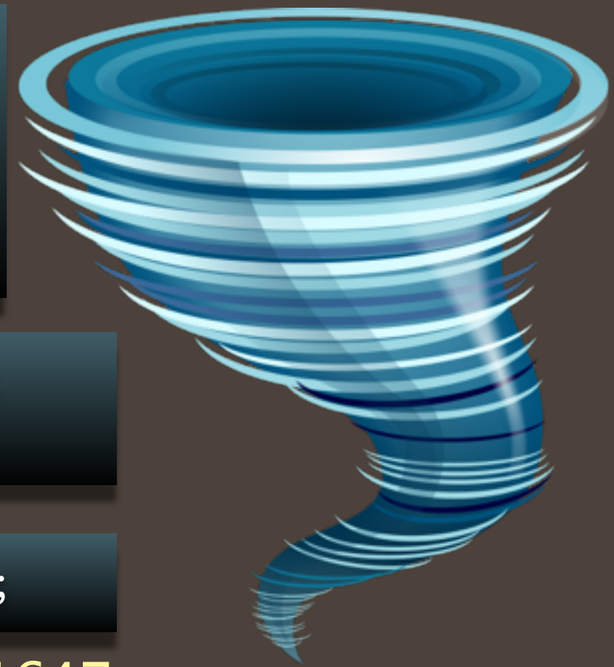
```
var Tornado = function (category, affectedAreas, windGust) {
    this.category = category;
    this.affectedAreas = affectedAreas;
    this.windGust = windGust;
};
```

```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
```

```
Tornado.prototype.valueOf = function( ) {
   var sum = 0;
   for (var i = 0; i < this.affectedAreas.length; i++) {
       sum += this.affectedAreas[i][1];
   }
   return sum;
};
```

```
twister.valueOf();
```

→ 641647



This `valueOf` is found in the Tornado prototype, which comes before the Object prototype in the chain. Thus, the Object prototype's `valueOf` has been effectively overridden, since it will never be found in the search.

OUR VALUE WILL EVEN UPDATE AS CITIES UPDATES

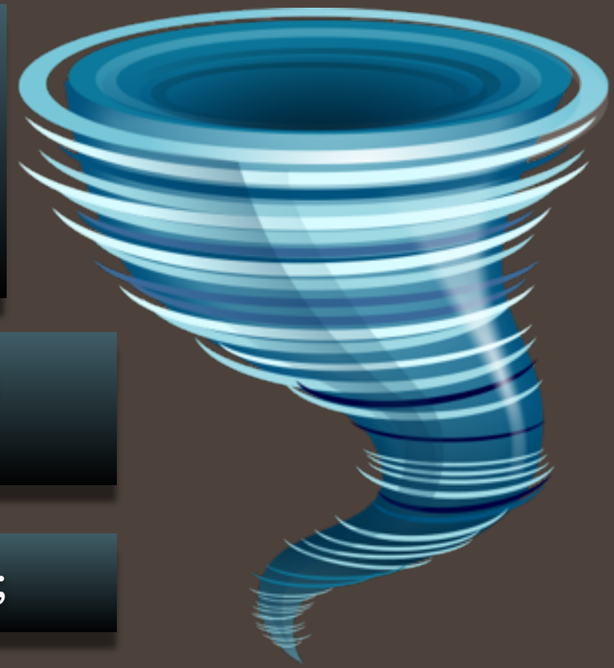
Each Tornado's 'affectedAreas' property can be updated outside the object with no loss of accuracy.

```
var Tornado = function (category, affectedAreas, windGust) {
    this.category = category;
    this.affectedAreas = affectedAreas;
    this.windGust = windGust;
};
```

```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
```

```
Tornado.prototype.valueOf = function( ) {
   var sum = 0;
   for (var i = 0; i < this.affectedAreas.length; i++) {
       sum += this.affectedAreas[i][1];
   }
   return sum;
};
```

```
twister.valueOf();
```



OUR VALUE WILL EVEN UPDATE AS CITIES UPDATES

Each Tornado's 'affectedAreas' property can be updated outside the object with no loss of accuracy.

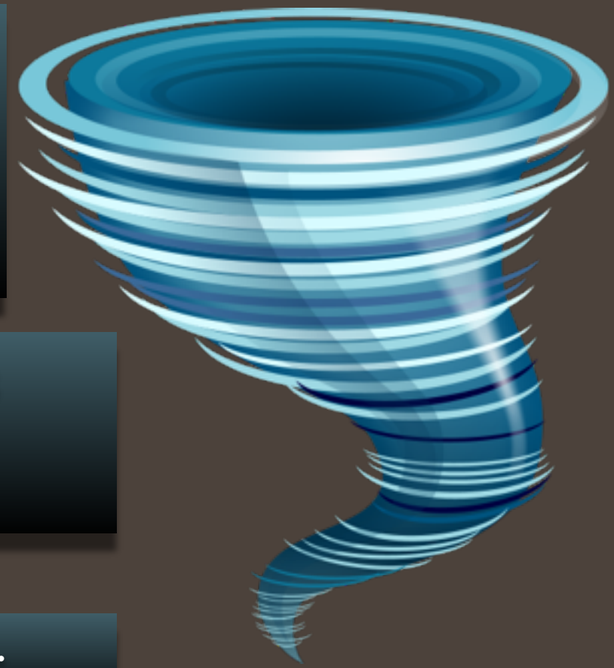
```
var Tornado = function (category, affectedAreas, windGust) {
    this.category = category;
    this.affectedAreas = affectedAreas;
    this.windGust = windGust;
};
```

```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```

```
Tornado.prototype.valueOf = function( ) {
   var sum = 0;
   for (var i = 0; i < this.affectedAreas.length; i++) {
       sum += this.affectedAreas[i][1];
   }
   return sum;
};
```

```
twister.valueOf();
```

→ 771692



Since the cities array was passed by reference, we'll get an updated value each time an affected area is added to the list.

ANOTHER USEFUL PROTOTYPAL PROPERTY IS TOSTRING( )

Default responses for Object's toString method are often uninteresting...but overriding it is cool!

```
var x = 4;
var y = "4";
```

```
var a = [ 3, "blind", "mice" ];
```

```
x.toString();
```

→ "4"

```
y.toString();
```

→ "4"

```
a.toString();
```

→ "3,blind,mice"

A call to toString on an Array will just string-ify and concatenate all the contents, separating each entry by a comma without any whitespace. Overriding toString in the Array prototype is often desirable.

ANOTHER USEFUL PROTOTYPAL PROPERTY IS TOSTRING( )

Default responses for Object's toString method are often uninteresting...but overriding it is cool!

```
var x = 4;
var y = "4";
```

```
var a = [ 3, "blind", "mice" ];
```

```
var double = function ( param ){
    return param *2;
};
```

```
x.toString();
```

→ "4"

```
a.toString();
```

→ "3,blind,mice"

```
y.toString();
```

→ "4"

```
double.toString();
```

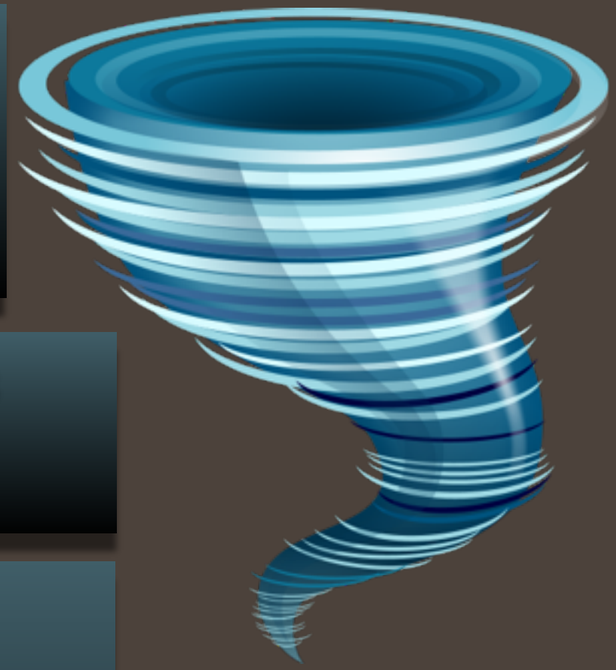
→ "function (param){
    return param *2;
}"

toString on a function can be pretty cool, if you ever need to concatenate a function into a formatted printout.

LET'S OVERRIDE TOSTRING( ) IN OUR TORNADO PROTOTYPE

We want a good representation of the data to come back when we call toString( ) on a Tornado Object

```
var Tornado = function (category, affectedAreas, windGust) {
    this.category = category;
    this.affectedAreas = affectedAreas;
    this.windGust = windGust;
}
```

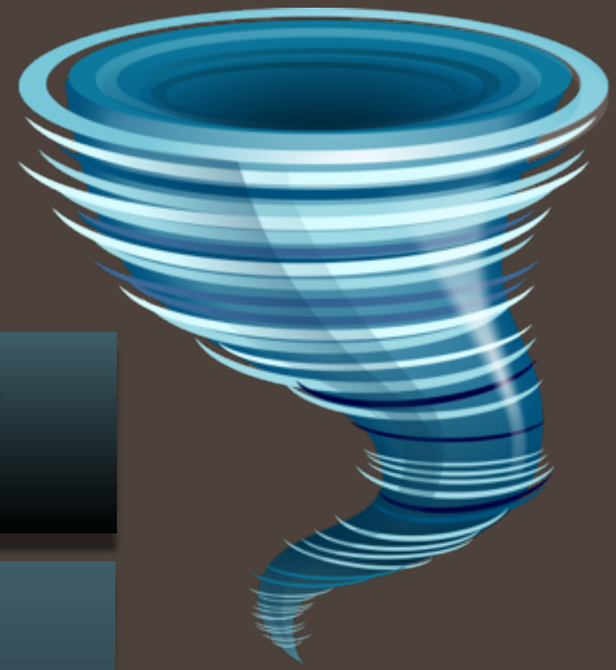


```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```

```
Tornado.prototype.toString = function( ) {
   var list = "";
   for (var i = 0; i < this.affectedAreas.length; i++) {
       if (i < this.affectedAreas.length - 1) {
           list = list + this.affectedAreas[i][0] + ", ";
       } else {
           list = list + "and " + this.affectedAreas[i][0]; }
   }
   return "This tornado has been classified as an " + this.category +
          ", with wind gusts up to " + this.windGust + "mph. Affected areas are: " +
          list + ", potentially affecting a population of " + this.valueOf() + ".";
}
```

LET'S OVERRIDE TOSTRING( ) IN OUR TORNADO PROTOTYPE

We want a good representation of the data to come back when we call toString( ) on a Tornado Object



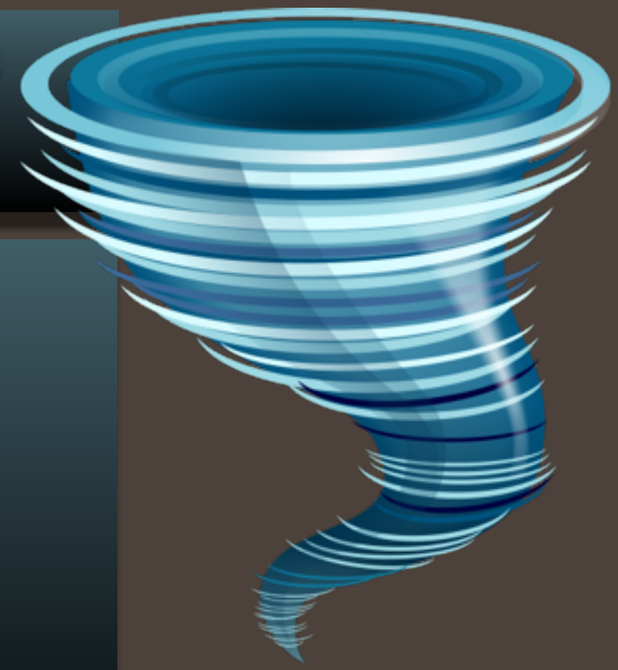
```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```

```
Tornado.prototype.toString = function( ) {
   var list = "";
   for (var i = 0; i < this.affectedAreas.length; i++) {
       if (i < this.affectedAreas.length - 1) {
           list = list + this.affectedAreas[i][0] + ", ";
       } else {
           list = list + "and " + this.affectedAreas[i][0]; }
   }
   return "This tornado has been classified as an " + this.category +
          ", with wind gusts up to " + this.windGust + "mph. Affected areas are: " +
          list + ", potentially affecting a population of " + this.valueOf() + ".";
}
```

LET'S OVERRIDE TOSTRING( ) IN OUR TORNADO PROTOTYPE

We want a good representation of the data to come back when we call toString( ) on a Tornado Object

```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```



```
Tornado.prototype.toString = function( ) {
   var list = "";
   for (var i = 0; i < this.affectedAreas.length; i++) {
       if (i < this.affectedAreas.length - 1) {
           list = list + this.affectedAreas[i][0] + ", ";
       } else {
           list = list + "and " + this.affectedAreas[i][0]; }
   }
   return "This tornado has been classified as an " + this.category +
          ", with wind gusts up to " + this.windGust + "mph. Affected areas are: " +
          list + ", potentially affecting a population of " + this.valueOf() + ".";
}
```

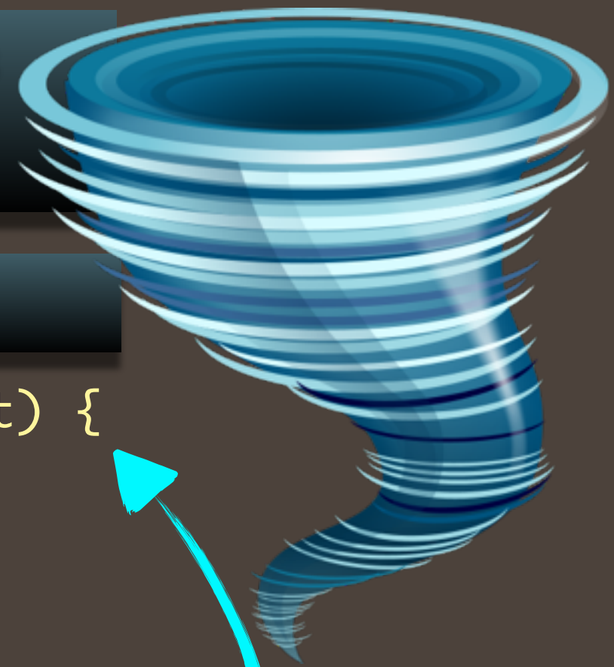
```
twister.toString();
```

→ "This tornado has been classified as an F5, with wind gusts up to 220mph. Affected areas are: Kansas City, Topeka, Lenexa, and Olathe, potentially affecting a population of 771692."

FINDING AN OBJECT'S CONSTRUCTOR AND PROTOTYPE

Some inherited properties provide ways to find an Object's nearest prototype ancestor

```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```



```
twister.constructor;
```

```
→ function (category, affectedAreas, windGust) {
      this.category = category;
      this.affectedAreas = affectedAreas;
      this.windGust = windGust;
  }
```

```
twister.constructor.prototype;
```

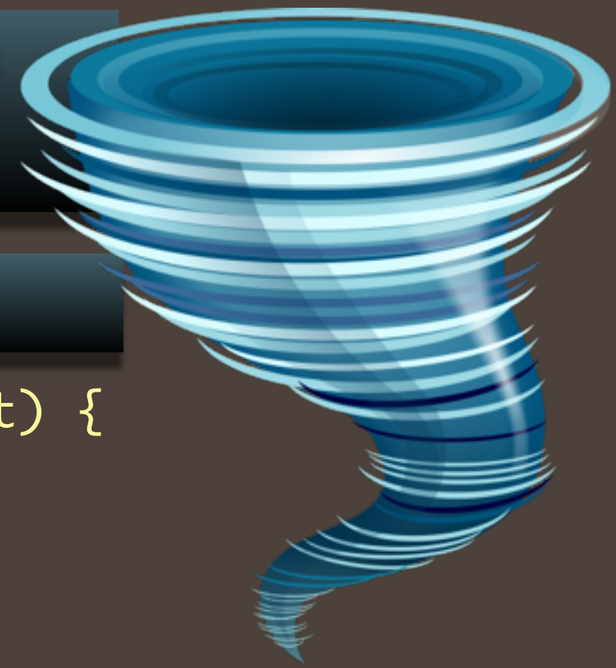
```
→ Object {valueOf: function, toString: function}
```

Remember that if a prototype Object is defined for a specific class, it will always be a property of the class's constructor, which is just another function Object.

FINDING AN OBJECT'S CONSTRUCTOR AND PROTOTYPE

Some inherited properties provide ways to find an Object's nearest prototype ancestor

```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```



```
twister.constructor;
```

```
→ function (category, affectedAreas, windGust) {
      this.category = category;
      this.affectedAreas = affectedAreas;
      this.windGust = windGust;
  }
```

```
twister.constructor.prototype;
```

```
→ Object {valueOf: function, toString: function}
```

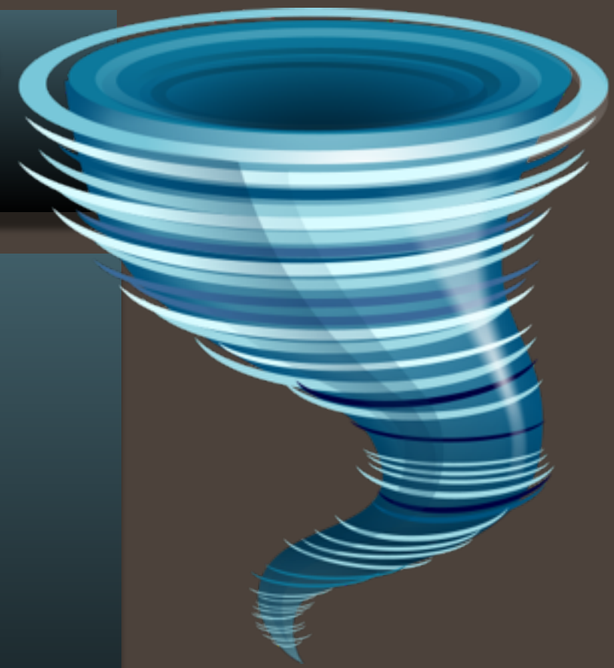
```
twister.__proto__;
```

```
→ Object {valueOf: function, toString: function}
```

HASOWNPROPERTY( ) HELPS IDENTIFY PROPERTY LOCATION

Searching prototype chains for potential overridden properties becomes easy with this function

```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```



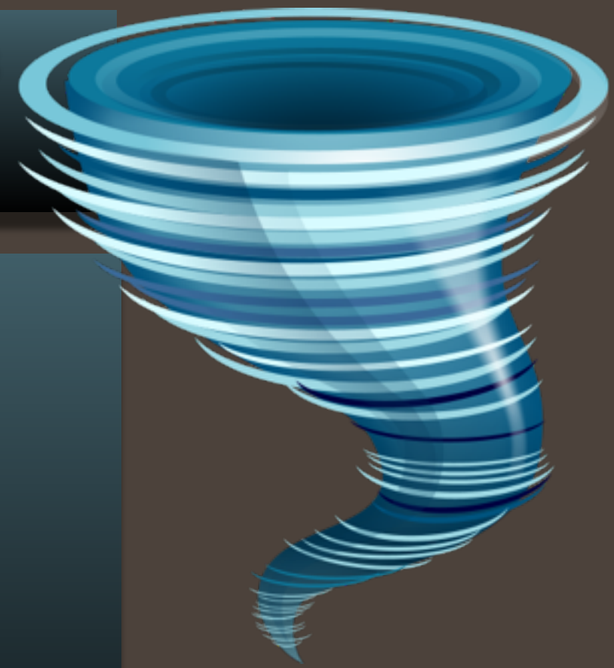
```
Object.prototype.findOwnerOfProperty  = function ( propName ) {



};
```

We'll build the function directly on the Object prototype so that every object we ever make can use the function!

HASOWNPROPERTY( ) HELPS IDENTIFY PROPERTY LOCATION

Searching prototype chains for potential overridden properties becomes easy with this function

```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```



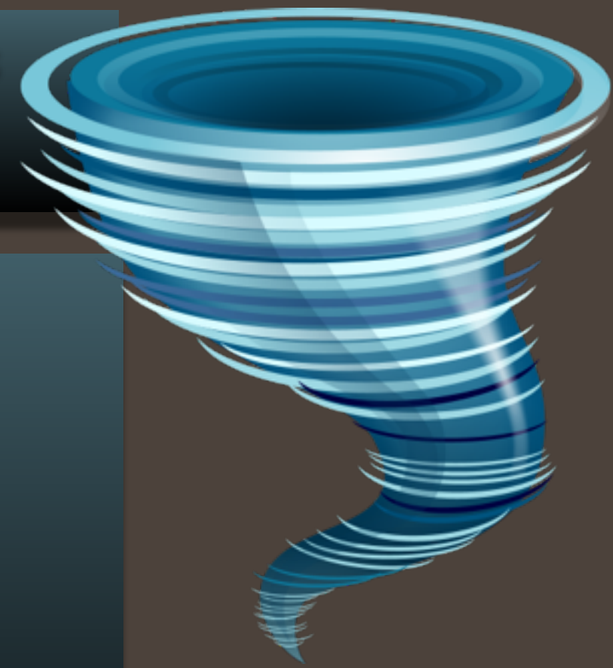
```
Object.prototype.findOwnerOfProperty  = function ( propName ) {
   var currentObject = this;



};
```

We'll start off looking for the property within the caller Object itself.

HASOWNPROPERTY( ) HELPS IDENTIFY PROPERTY LOCATION

Searching prototype chains for potential overridden properties becomes easy with this function

```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```



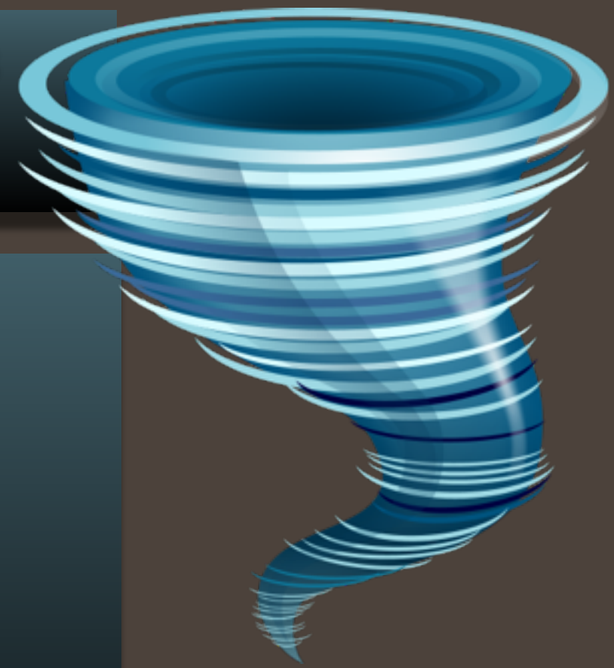
```
Object.prototype.findOwnerOfProperty  = function ( propName ) {
   var currentObject = this;
   while (currentObject !== null){



   }

};
```

We'll keep searching the prototype chain until we've tried to go beyond the Object prototype...which has no prototype. Trying to access it would produce null.

HASOWNPROPERTY( ) HELPS IDENTIFY PROPERTY LOCATION

Searching prototype chains for potential overridden properties becomes easy with this function

```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```



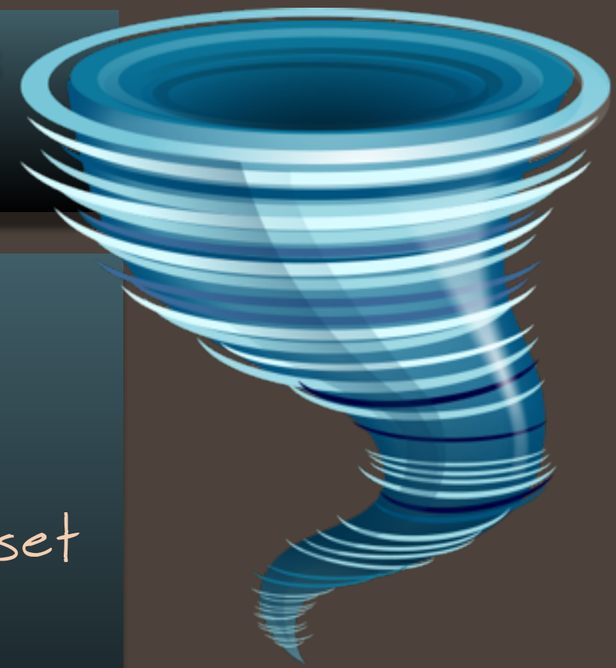
```
Object.prototype.findOwnerOfProperty  = function ( propName ) {
   var currentObject = this;
   while (currentObject !== null){
      if (currentObject.hasOwnProperty(propName)) {
         return currentObject;
      }

   }

};
```

If the currently examined Object has the property, success! Return that Object.

HASOWNPROPERTY( ) HELPS IDENTIFY PROPERTY LOCATION

Searching prototype chains for potential overridden properties becomes easy with this function

```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```



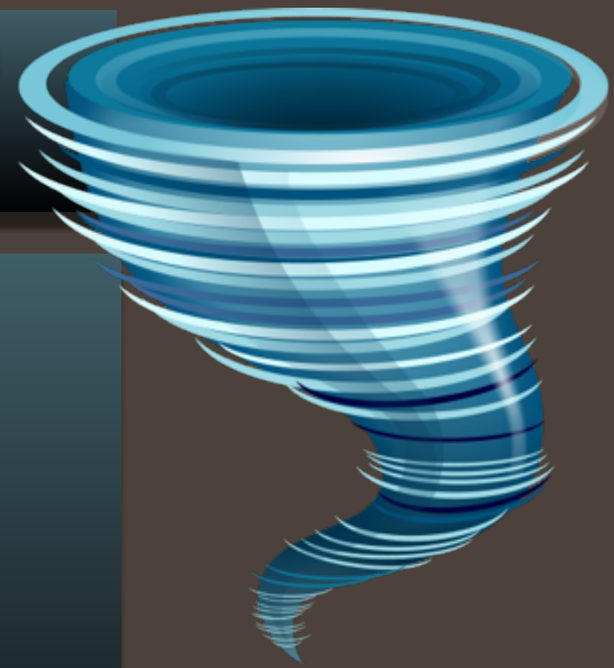
```
Object.prototype.findOwnerOfProperty  = function ( propName ) {
   var currentObject = this;
   while (currentObject !== null){
      if (currentObject.hasOwnProperty(propName)) {
         return currentObject;
      }
      else {
         currentObject = currentObject.__proto__;
      }
   }

};
```

Otherwise, we set the currently examined Object to be the previously examined Object's prototype.

HASOWNPROPERTY( ) HELPS IDENTIFY PROPERTY LOCATION

Searching prototype chains for potential overridden properties becomes easy with this function

```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```



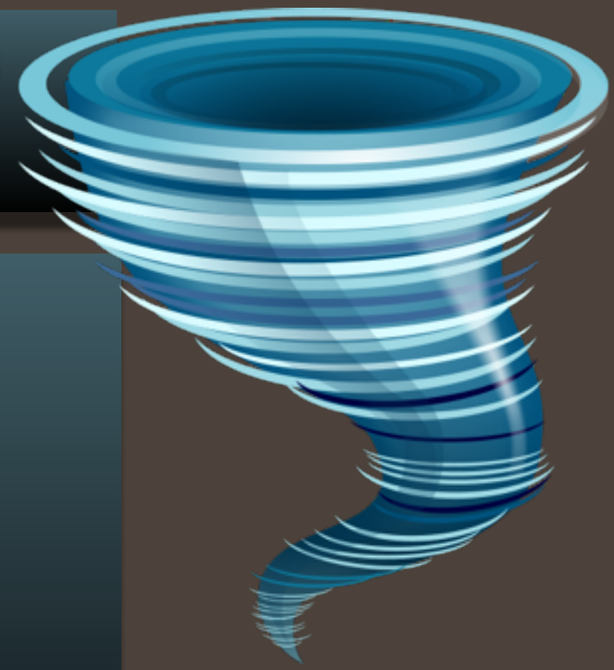
```
Object.prototype.findOwnerOfProperty  = function ( propName ) {
   var currentObject = this;
   while (currentObject !== null){
      if (currentObject.hasOwnProperty(propName)) {
         return currentObject;
      }
      else {
         currentObject = currentObject.__proto__;
      }
   }
   return "No property found!";
};
```

If the while loop exits, we know we didn't find the property, and should probably let ourselves know, right?

HASOWNPROPERTY( ) HELPS IDENTIFY PROPERTY LOCATION

Searching prototype chains for potential overridden properties becomes easy with this function

```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```

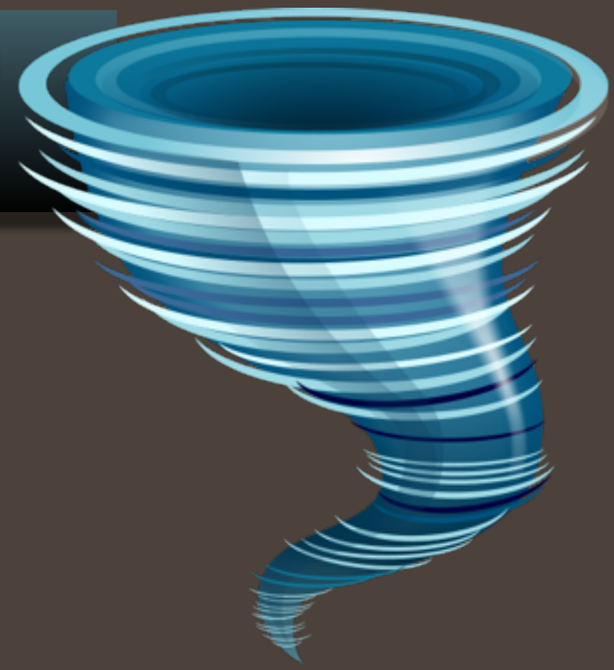


```
Object.prototype.findOwnerOfProperty  = function ( propName ) {
   var currentObject = this;
   while (currentObject !== null){
      if (currentObject.hasOwnProperty(propName)) {
         return currentObject;
      }
      else {
         currentObject = currentObject.__proto__;
      }
   }
   return "No property found!";
};
```

HASOWNPROPERTY( ) HELPS IDENTIFY PROPERTY LOCATION

Searching prototype chains for potential overridden properties becomes easy with this function

```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```



```
Object.prototype.findOwnerOfProperty  = function ( propName ) {
    var currentObject = this;
    while (currentObject !== null){
        if (currentObject.hasOwnProperty(propName)) {
            return currentObject;
        }
        else {
            currentObject = currentObject.__proto__;
        }
    }
    return "No property found!";
};
```

Searching for the valueOf() to which twister has access reveals the Tornado prototype as the owner. Thanks, hasOwnProperty()!

TORNADO Prototype

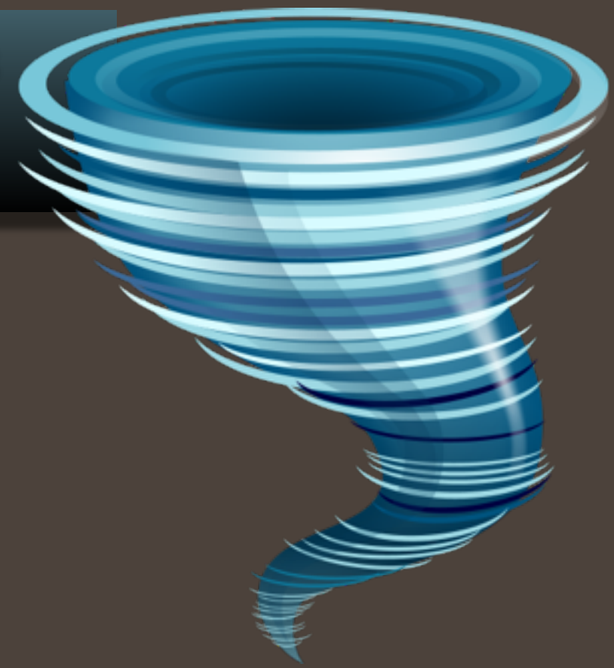
```
twister.findOwnerOfProperty("valueOf");
```

→ Object {valueOf: function, toString: function}

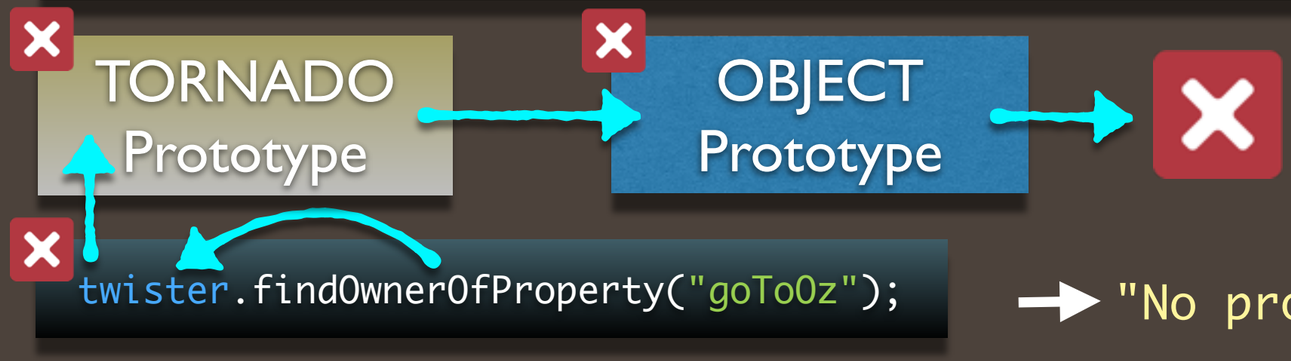
HASOWNPROPERTY( ) HELPS IDENTIFY PROPERTY LOCATION

Searching prototype chains for potential overridden properties becomes easy with this function

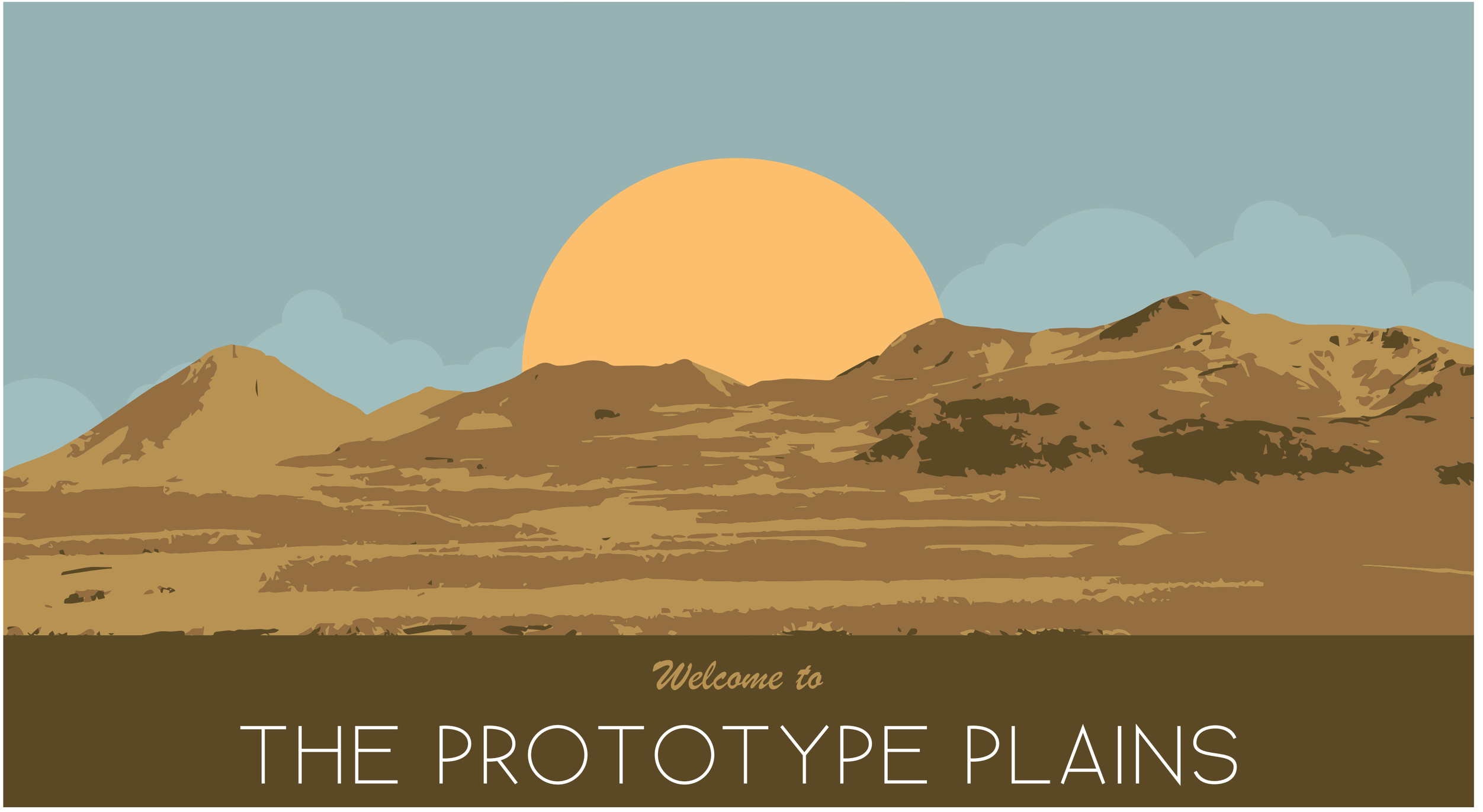
```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];
var twister = new Tornado( "F5", cities, 220 );
cities.push( ["Olathe", 130045] );
```



```
Object.prototype.findOwnerOfProperty  = function ( propName ) {
    var currentObject = this;
    while (currentObject !== null){
        if (currentObject.hasOwnProperty(propName)) {
            return currentObject;
        }
        else {
            currentObject = currentObject.__proto__;
        }
    }
    return "No property found!";
};
```



Trying to find the goToOz property reveals that none exists for this Tornado. Which sort of sucks.


Welcome to
THE PROTOTYPE PLAINS