

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

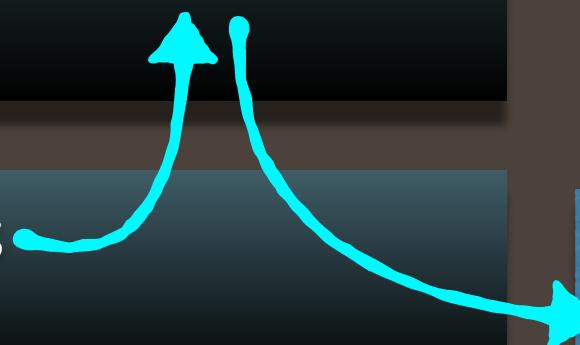
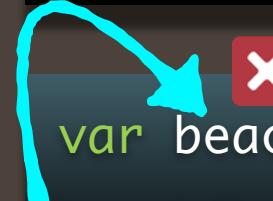
By setting a constructor's prototype property, every new instance will refer to it for extra properties!



```
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");
```



OBJECT
PROTOTYPE

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

`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.

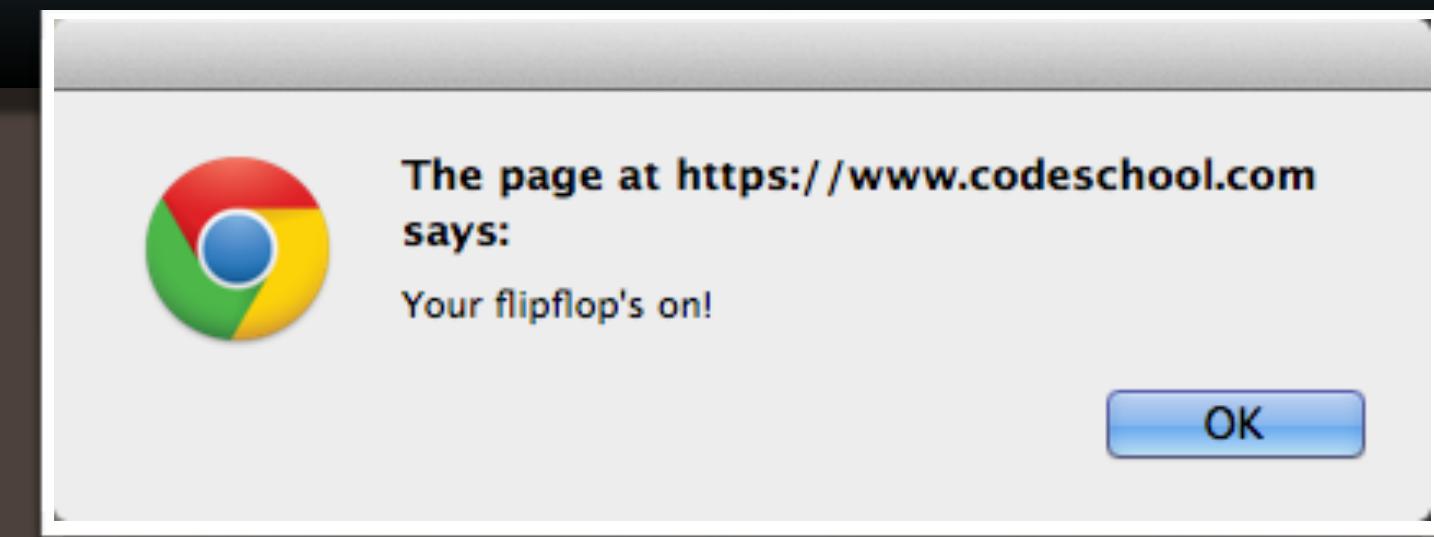


```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {  
    this.size = shoeSize;  
    this.color = shoeColor;  
    this.gender = forGender;  
    this.construction = constructStyle;  
}
```

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

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

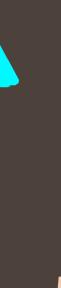
→ [3, "blind", "mice"]

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

→ "4"

```
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;  
};
```

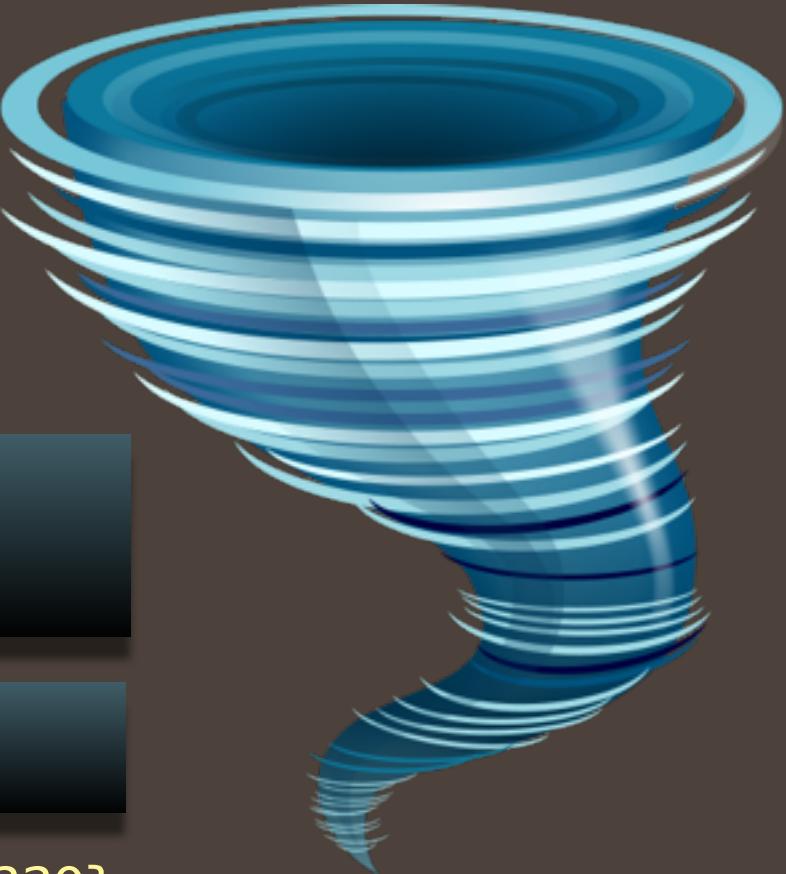
Constructors can be function expressions, too!

```
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.



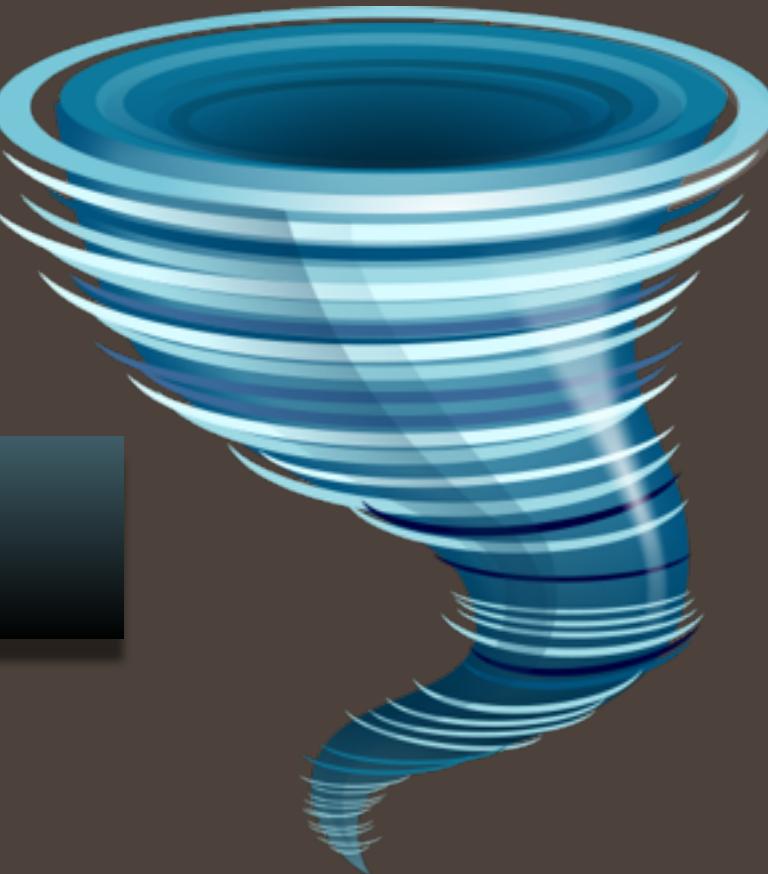
OVERRIDING PROTOTYPAL PROPERTIES

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

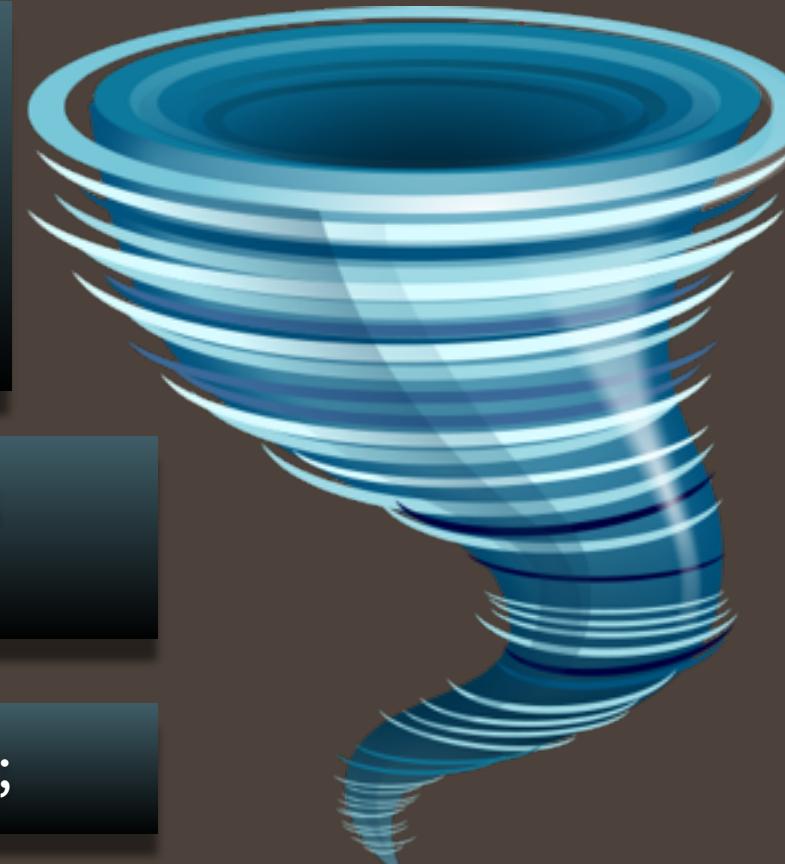
```
var Tornado = function (category, affectedAreas, windGust) {  
    this.category = category;  
    this.affectedAreas = affectedAreas;  
    this.windGust = windGust;  
};
```

```
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

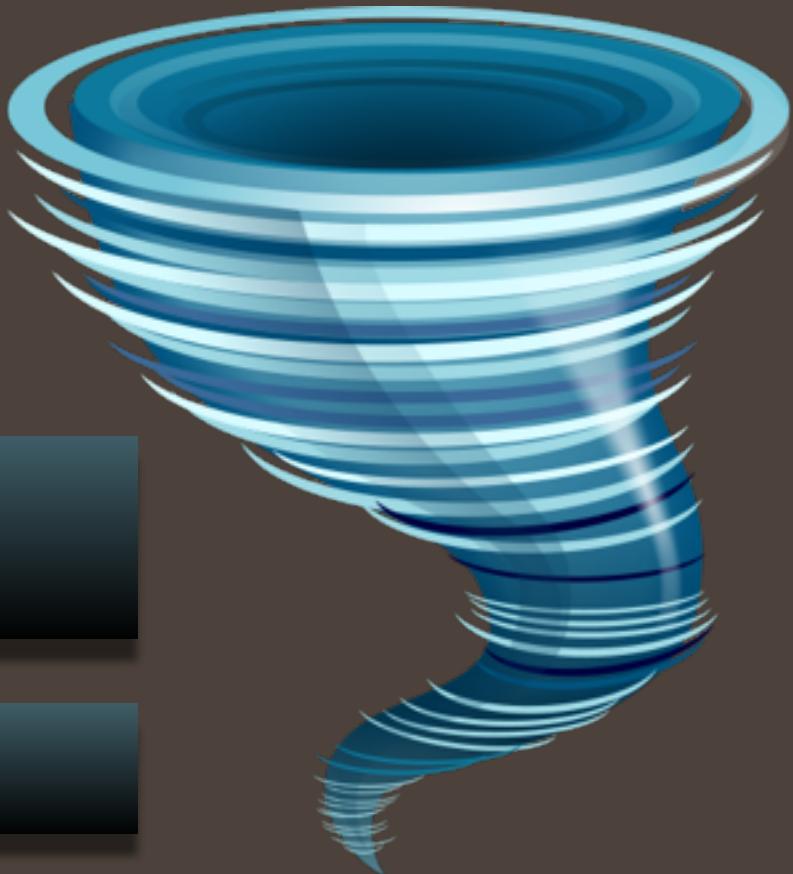
Each Tornado's 'affectedAreas' property can be updated outside the object with no loss of accuracy.

```
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;  
};
```

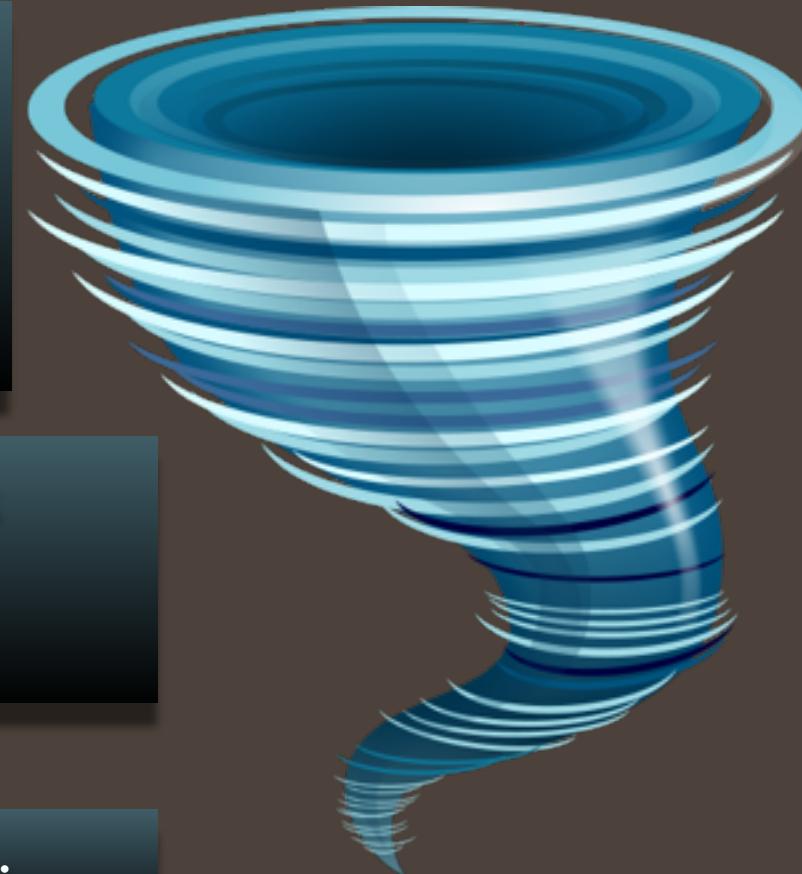
```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];  
var twister = new Tornado( "F5", cities, 220 );  
cities.push( ["Olathe", 130045] );
```

```
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";
```

`x.toString();`

→ "4"

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

`a.toString();`

→ "3,blind,mice"

`y.toString();`

→ "4"

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

`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.

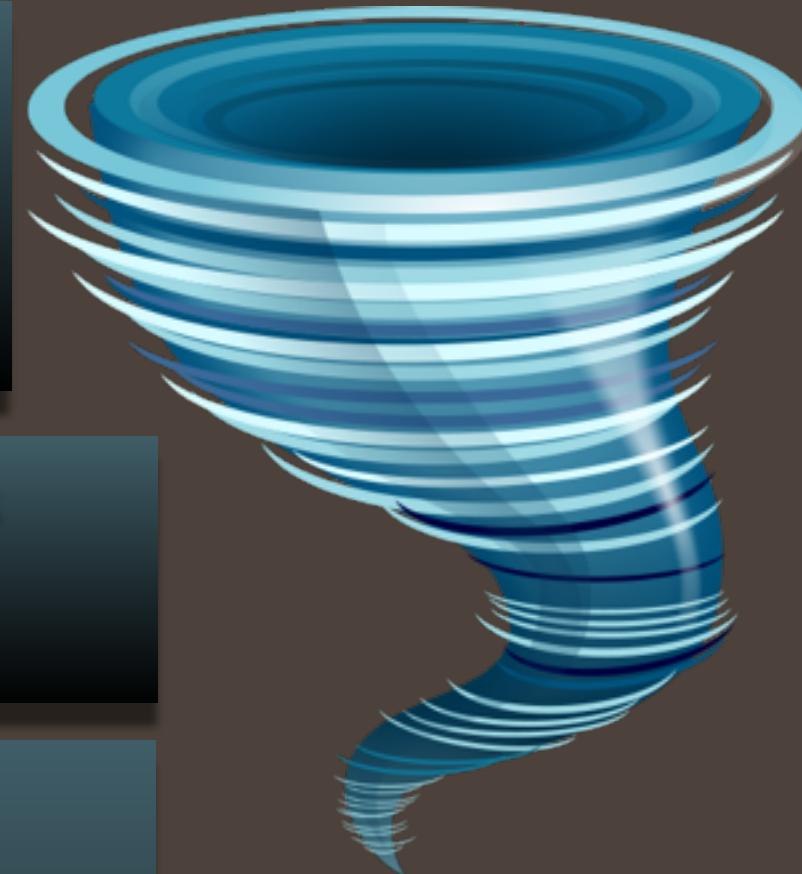
LET'S OVERRIDE `TOSTRING()` IN OUR `TORNADO` PROTOTYPE

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;  
}
```

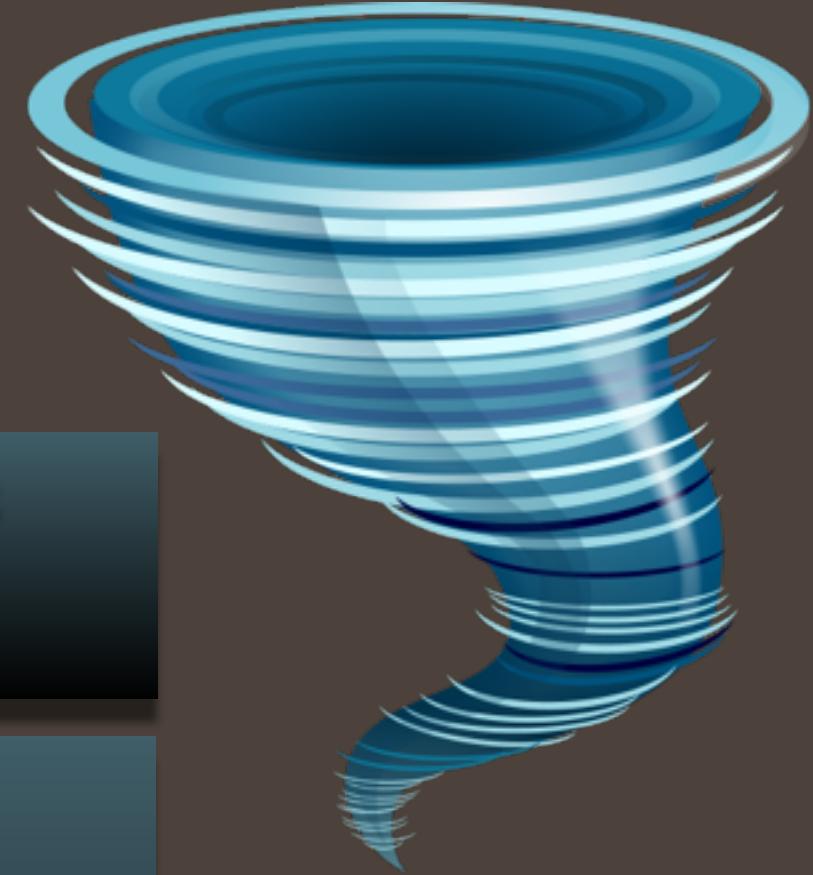
```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];  
var twister = new Tornado( "F5", cities, 220 );  
cities.push( ["Olathe", 130045] );
```

```
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

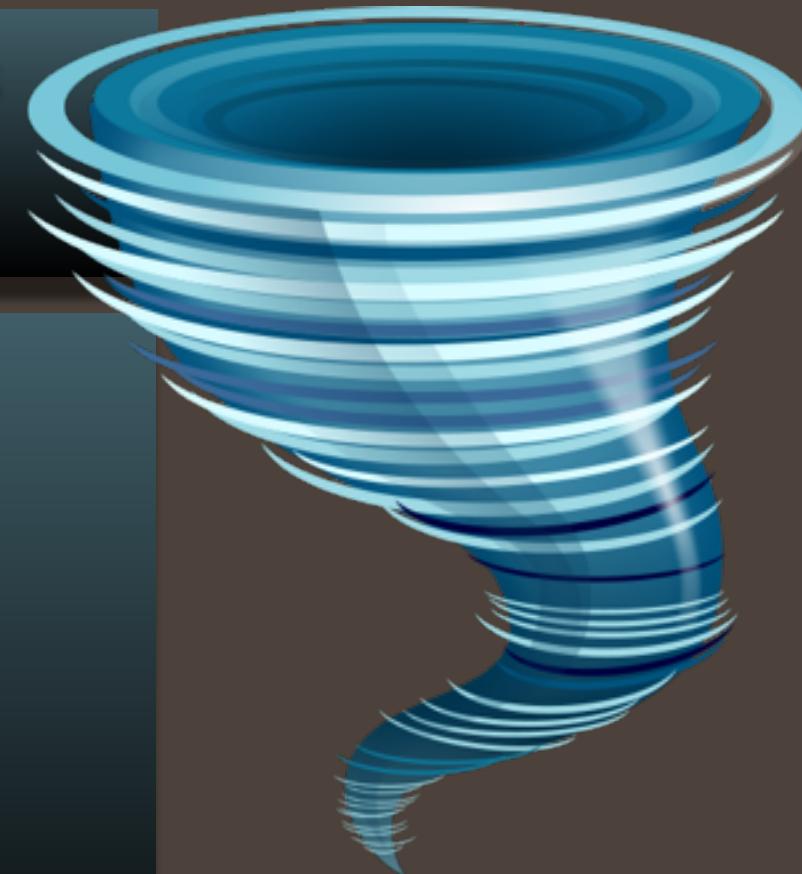
We want a good representation of the data to come back when we call `toString()` on a Tornado Object

```
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

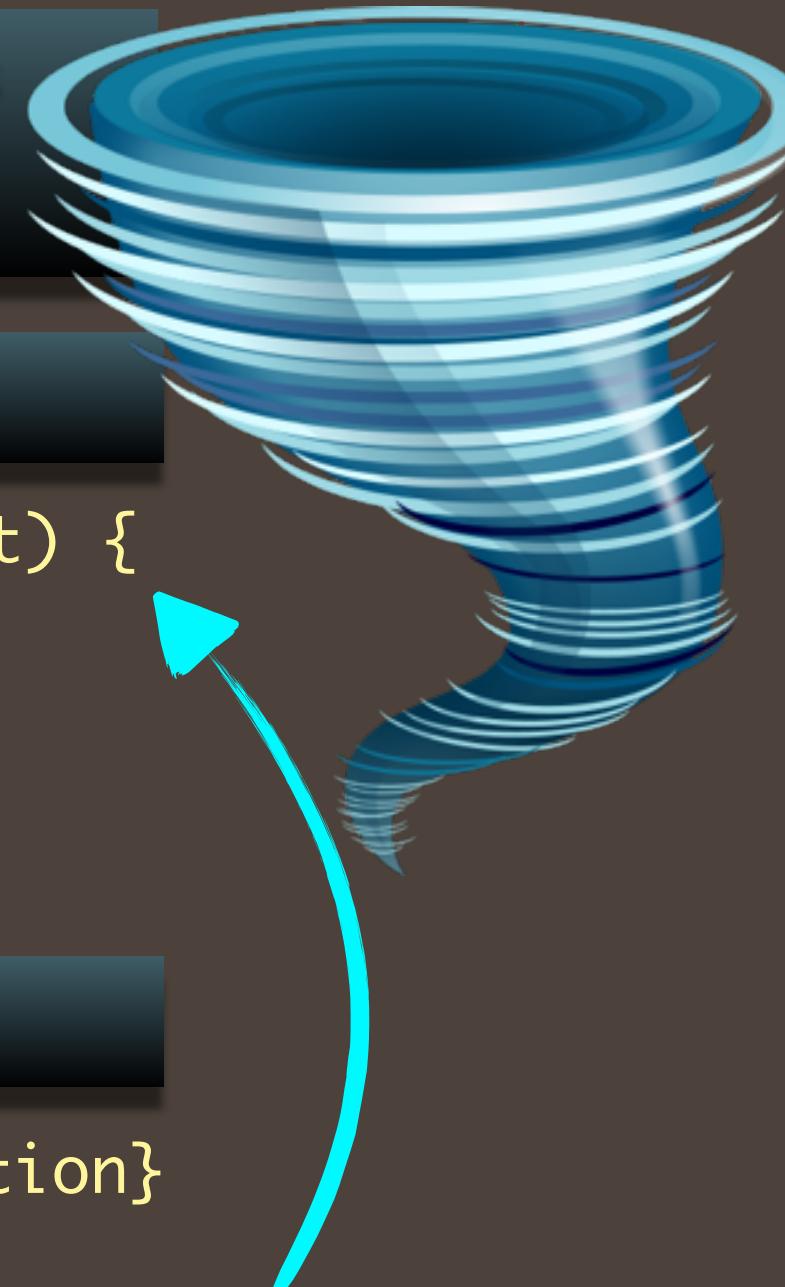
```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];  
var twister = new Tornado( "F5", cities, 220 );  
cities.push( ["Olathe", 130045] );
```

```
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;
```

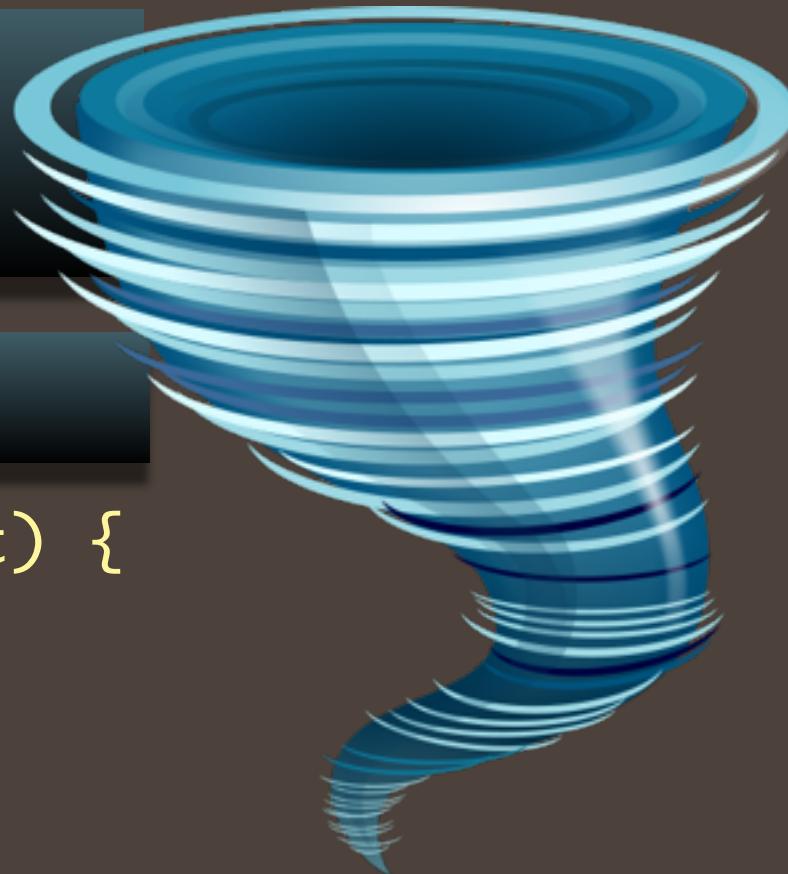
→ function (category, affectedAreas, windGust) {
 this.category = category;
 this.affectedAreas = affectedAreas;
 this.windGust = windGust;
}

```
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

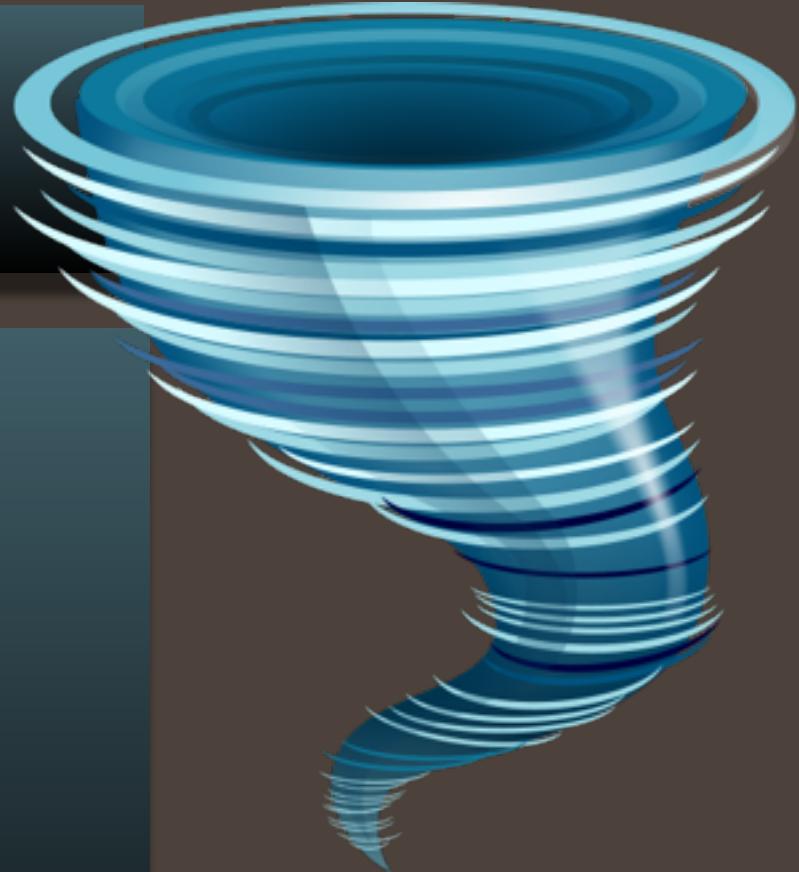
```
var cities = [ ["Kansas City", 464310], ["Topeka", 127939], ["Lenexa", 49398] ];  
var twister = new Tornado( "F5", cities, 220 );  
cities.push( ["Olathe", 130045] );
```

```
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

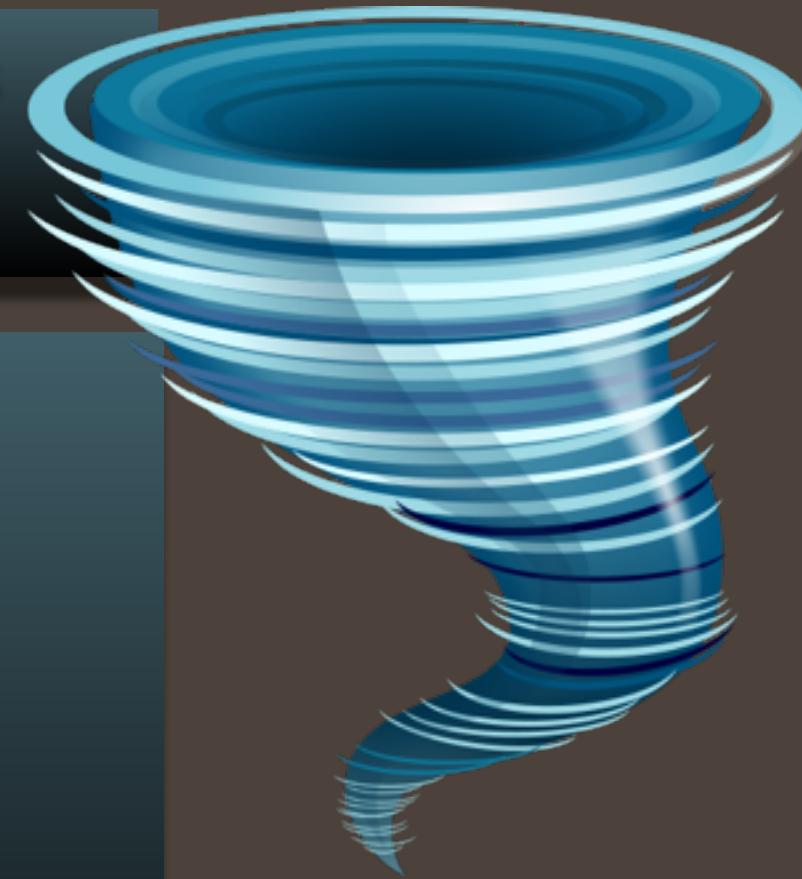
Searching prototype chains for potential overridden properties becomes easy with this function

```
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

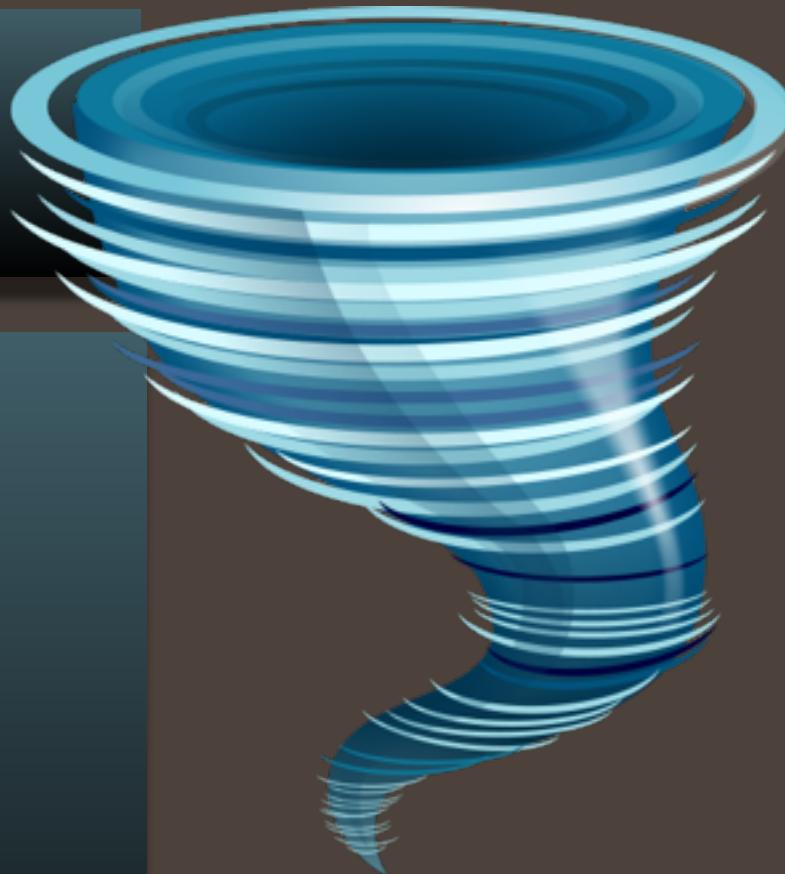
Searching prototype chains for potential overridden properties becomes easy with this function

```
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.



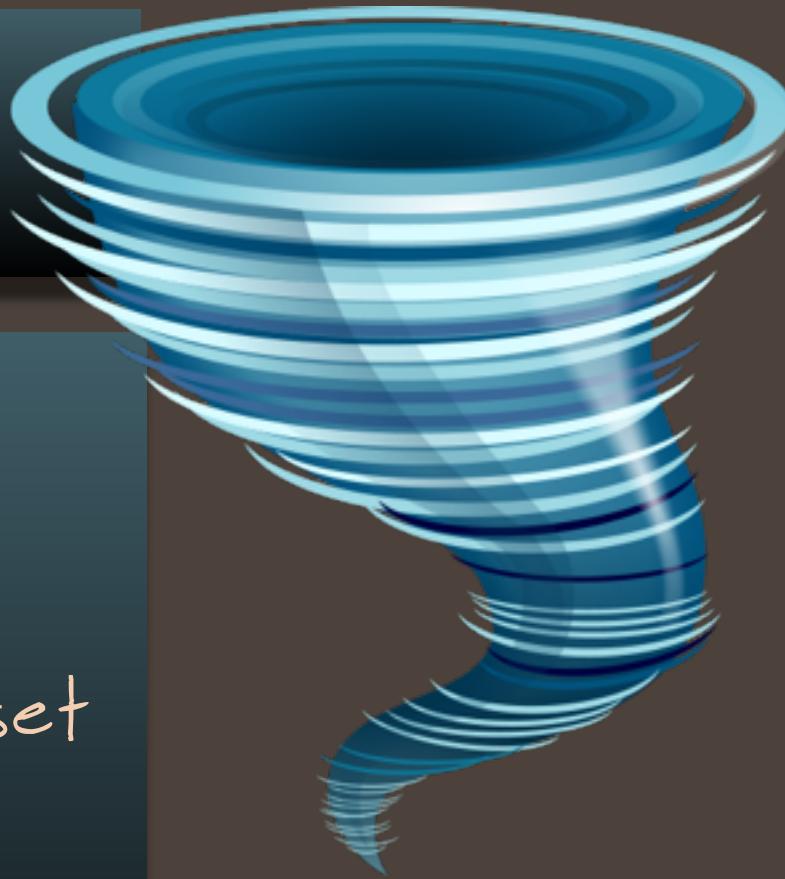
HASOWNPROPERTY() HELPS IDENTIFY PROPERTY LOCATION

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.



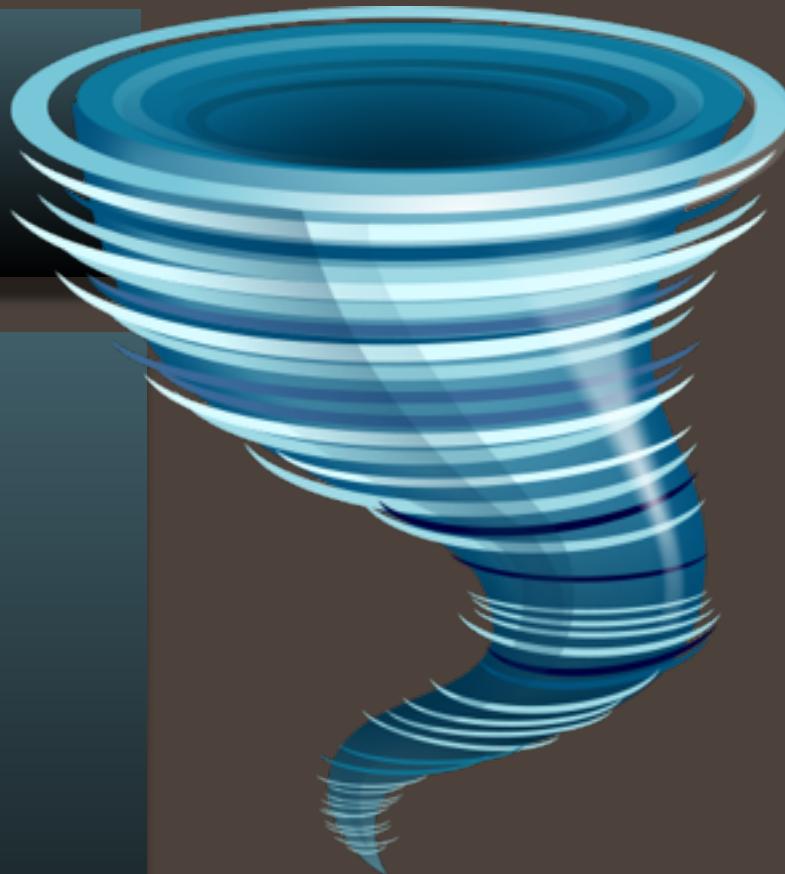
HASOWNPROPERTY() HELPS IDENTIFY PROPERTY LOCATION

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?

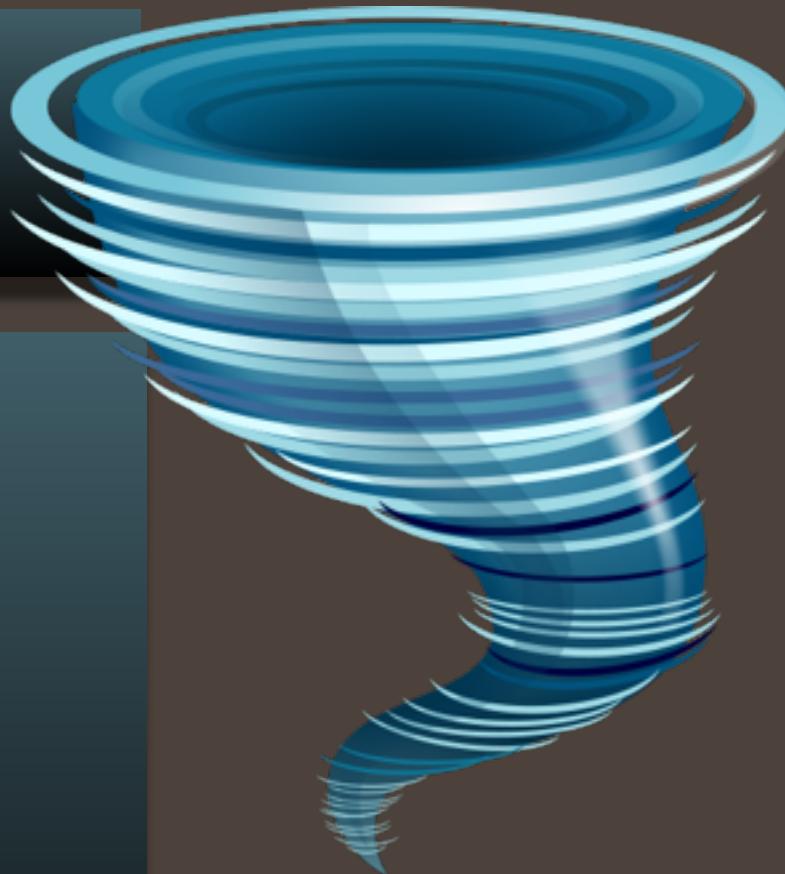


HASOWNPROPERTY() HELPS IDENTIFY PROPERTY LOCATION

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

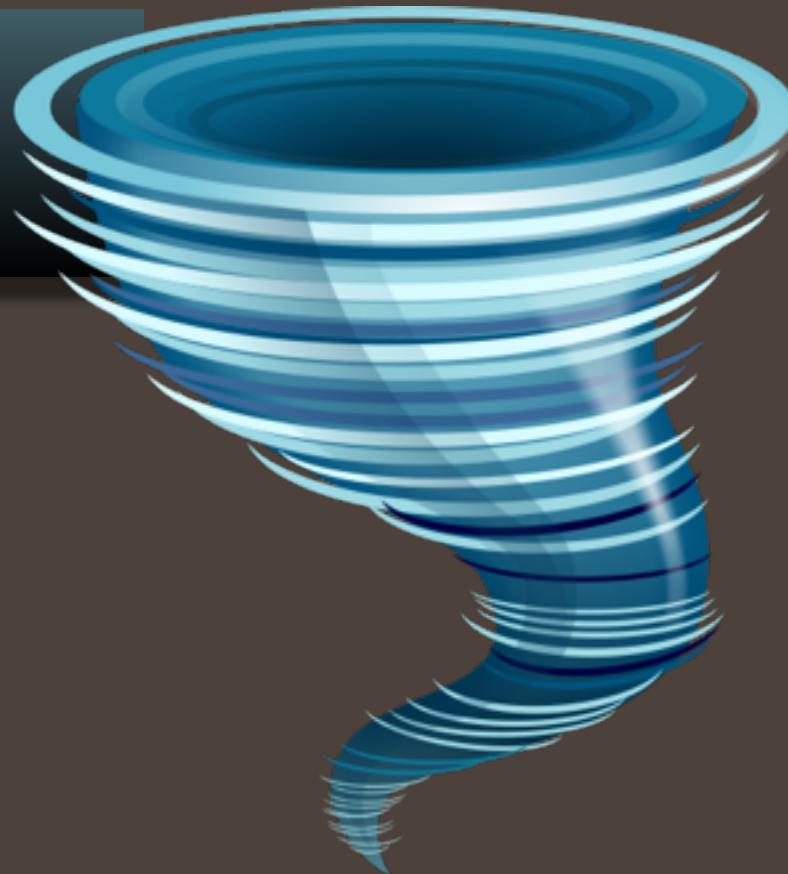
Searching prototype chains for potential overridden properties becomes easy with this function

```
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!";  
};
```



→ Object {valueOf: function, toString: function}



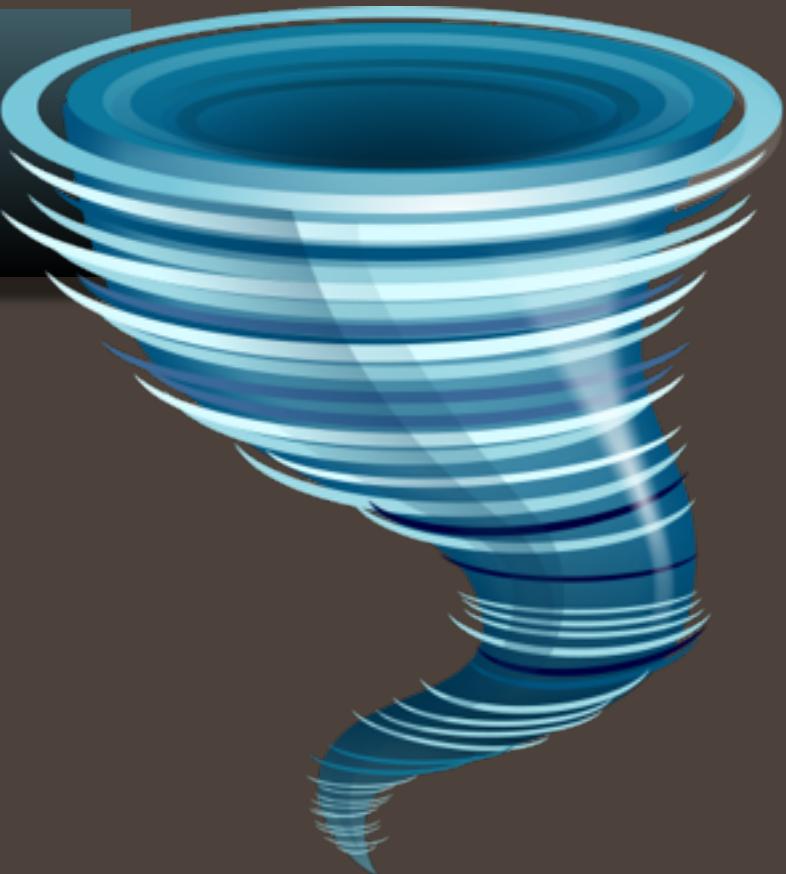
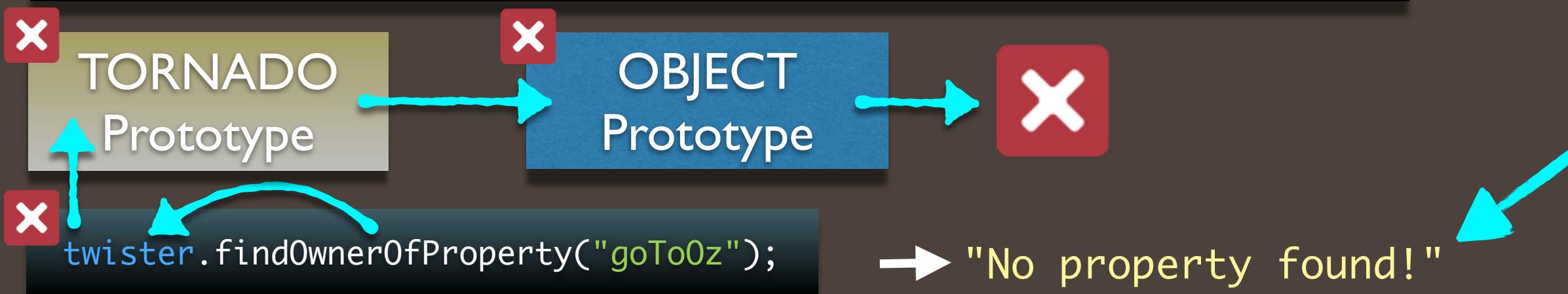
Searching for the `valueOf()` to which `twister` has access reveals the `Tornado` prototype as the owner. Thanks, `hasOwnProperty()`!

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 `goTo0z` property reveals that none exists for this Tornado. Which sort of sucks.



Welcome to
THE PROTOTYPE PLAINS