

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.CREATED**

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.CREATED**

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.CREATED**

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

```
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

```
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



```
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



```
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 );
```

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

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



```
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

Some Shoes

size

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

Some Shoes

size
color
gender

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

Some Shoes

size
color
gender
construction

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

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

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

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()

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.

Some Shoes

laceColor
laceUp()
jewels
bowPosition
dimensionalTravel()

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

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();
```



The page at <https://www.codeschool.com>
says:

Shoe's on!

OK

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