Visit the Wondrous

FOREST OF FUNCTION EXPRESSIONS

LEVEL 1

FOREST OF FUNCTION EXPRESSIONS

FUNCTIONS ON THE FLY

Building functions within code execution rather than at program load time

```
function diffOfSquares (a, b) {
    return a*a - b*b;
}
```

Builds in memory immediately when the program loads

FUNCTIONS ON THE FLY

Building functions within code execution rather than at program load time

```
function diffOfSquares (a, b) {
    return a*a - b*b;
}
```

Builds in memory immediately when the program loads

```
var diff =
```

FUNCTIONS ON THE FLY

Building functions within code execution rather than at program load time

```
function diffOfSquares (a, b) {
    return a*a - b*b;
}
```

Builds in memory immediately when the program loads

```
var diff =  function diffOfSquares (a, b) {
    return a*a - b*b;
};
```

The function keyword will now assign the following function to the variable.

FUNCTIONS ON THE FLY

Building functions within code execution rather than at program load time

```
function diffOfSquares (a, b) {
    return a*a - b*b;
}
```

Builds in memory immediately when the program loads

```
var diff =  function diffOfSquares (a, b) {
    return a*a - b*b;
};
```

Needs a semicolon to complete the assignment statement in a file.

FUNCTIONS ON THE FLY

Building functions within code execution rather than at program load time

```
function diffOfSquares (a, b) {
    return a*a - b*b;
}
```

Builds in memory immediately when the program loads

```
var diff =  function diffOfSquares (a, b) {
    return a*a - b*b;
};
```

Now the function builds ONLY when this line of code is reached.

```
diff( 9, 5 );
```

→ 56

FUNCTIONS ON THE FLY

Building functions within code execution rather than at program load time

```
function diffOfSquares (a, b) {
    return a*a - b*b;
}
```

This name is optional in JavaScript since we now use the variable name.

```
var diff =  function diffOfSquares (a, b) {
    return a*a - b*b;
};
```

```
diff( 9, 5 );
```

→ 56

Notice the variable name needs parentheses, parameters and a semicolon to execute the function it contains.

ANONYMOUS FUNCTIONS

No need for naming the function a second time

```
var diff =  function diffOfSquares (a, b) {
    return a*a - b*b;
};
```

ANONYMOUS FUNCTIONS

No need for naming the function a second time

```
var diff =  function                    (a, b) {
    return a*a - b*b;
};
```

ANONYMOUS FUNCTIONS

No need for naming the function a second time

```
var diff =  function (a, b) {
    return a*a - b*b;
};
```

Look Ma, no name!

```
diff( 4, 2 );
```

→ 12

Parameters are still passed to the variable name, which JavaScript believes is a function.

ANONYMOUS FUNCTIONS

No need for naming the function a second time

```
var diff =  function  (a, b) {
    return a*a - b*b;
};
```

```
console.log(diff);
```

```
→ function (a, b) {
      return a*a - b*b;
  }
```

Logging out using the variable name will display the entire function it contains.

STORED FUNCTIONS IN A NATIONAL PARK TERMINAL

A variable that holds a function can be passed into other functions

terminal.js

```
var greeting = function () {
    alert("Thanks for visiting the Badlands!\n" +
          "We hope your stay is...better than most.");
};


...


closeTerminal( greeting );


function closeTerminal( message ){
    ...
    message();
    ...
}
```

The greeting variable is passed in as a parameter to an existing declared function.

STORED FUNCTIONS IN A NATIONAL PARK TERMINAL

A variable that holds a function can be passed into other functions

terminal.js

```
var greeting = function () {
    alert("Thanks for visiting the Badlands!\n" +
          "We hope your stay is...better than most.");
};


...


closeTerminal( greeting );


function closeTerminal( message ){
    ...
    message();
    ...
}
```

```
function closeTerminal( message ){
    ...
    message();
    ...
}
```

STORED FUNCTIONS IN A NATIONAL PARK TERMINAL

A variable that holds a function can be passed into other functions

terminal.js

```
var greeting = function () {
    alert("Thanks for visiting the Badlands!\n" +
          "We hope your stay is...better than most.");
};


...


closeTerminal( greeting );


function closeTerminal( message ){
    ...
    message();
    ...
}
```

```
function closeTerminal(          ){
    ...
    message();
    ...
}
```

STORED FUNCTIONS IN A NATIONAL PARK TERMINAL

A variable that holds a function can be passed into other functions

terminal.js

```
var greeting = function () {
    alert("Thanks for visiting the Badlands!\n" +
          "We hope your stay is...better than most.");
};


...


closeTerminal( greeting );


function closeTerminal( message ){
    ...
    message();
    ...
}
```

```
function closeTerminal( greeting ){
    ...
    message();
    ...
}
```

STORED FUNCTIONS IN A NATIONAL PARK TERMINAL

A variable that holds a function can be passed into other functions

terminal.js

```
var greeting = function () {
    alert("Thanks for visiting the Badlands!\n" +
          "We hope your stay is...better than most.");
};

...

closeTerminal( greeting );


function closeTerminal( message ){
    ...
    message();
    ...
}
```

```
function closeTerminal( greeting ){
    ...
    greeting();
    ...
}
```

NOW, WHAT IF WE HAD MULTIPLE GREETINGS?

Function expressions can give flexibility in choosing which functionality to build

terminal.js

```
var greeting = function () {
    alert("Thanks for visiting the Badlands!\n" +
          "We hope your stay is...better than most.");
};

...

closeTerminal( greeting );

function closeTerminal( message ){
    ...
    message();
    ...
}
```

```
function closeTerminal( greeting ){
    ...
    greeting();
    ...
}
```

NOW, WHAT IF WE HAD MULTIPLE GREETINGS?

Function expressions can give flexibility in choosing which functionality to build

terminal.js

```
var greeting;
...some code sets newCustomer to true or false...
if( newCustomer ){
    greeting = function () {
        alert("Thanks for visiting the Badlands!\n" +
              "We hope your stay is...better than most.");
    };
} else {
    greeting = function () {
        alert("Welcome back to the Badlands!\n" +
              "Guess they aren't so bad huh?");
    };
}
closeTerminal( greeting );
function closeTerminal( message ){
    ...
    message();
    ...
}
```

```
var newCustomer = false;
```

NOW, WHAT IF WE HAD MULTIPLE GREETINGS?

Function expressions can give flexibility in choosing which functionality to build

terminal.js

```
var greeting;
...some code sets newCustomer to true or false...
if( newCustomer ){
    greeting = function () {
        alert("Thanks for visiting the Badlands!\n" +
              "We hope your stay is...better than most.");
    };
} else {
    greeting = function () {
        alert("Welcome back to the Badlands!\n" +
              "Guess they aren't so bad huh?");
    };
}
closeTerminal( greeting );
function closeTerminal( message ){
    ...
    message();
    ...
}
```

```
var newCustomer = true;
```

Visit the Wondrous

FOREST OF FUNCTION EXPRESSIONS

USING FE'S WITH ARRAYS AND MAP()

A function expression is just that...an expression. We can pass them without variables!

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( *some coolFunction goes here* );
```

The map( ) method will always take in a function as a parameter, and return a new array with the results.

USING FE'S WITH ARRAYS AND MAP()

A function expression is just that...an expression. We can pass them without variables!

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( *some coolFunction goes here* );
```

USING FE'S WITH ARRAYS AND MAP()

A function expression is just that...an expression. We can pass them without variables!

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( *some coolFunction goes here* );
```

USING FE'S WITH ARRAYS AND MAP()

A function expression is just that...an expression. We can pass them without variables!

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( *some coolFunction goes here* );
```

USING FE'S WITH ARRAYS AND MAP()

A function expression is just that...an expression. We can pass them without variables!

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( *some coolFunction goes here* );
```

USING FE'S WITH ARRAYS AND MAP()

A function expression is just that...an expression. We can pass them without variables!

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( *some coolFunction goes here* );
```

USING FE'S WITH ARRAYS AND MAP()

A function expression is just that...an expression. We can pass them without variables!

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( *some coolFunction goes here* );
```

USING FE'S WITH ARRAYS AND MAP()

A function expression is just that...an expression. We can pass them without variables!

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( *some coolFunction goes here* );
```

USING FE'S WITH ARRAYS AND MAP()

A function expression is just that...an expression. We can pass them without variables!

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( *some coolFunction goes here* );
```

12	4	3	9	8	6	10	1

USING FE'S WITH ARRAYS AND MAP()

Map works like a loop that applies a function to each array index

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( *some coolFunction goes here* );
```

```
var results = [ ];
for(var i = 0; i < numbers.length; i++){

    results[i] =

}
```

USING FE'S WITH ARRAYS AND MAP()

Map works like a loop that applies a function to each array index

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( *some coolFunction goes here* );
```

```
var results = [ ];
for(var i = 0; i < numbers.length; i++){

    results[i] = coolFunction(numbers[i]);

}
```

The array's map conveniently takes this entire loop format and consolidates it to one nice line of code.

USING FE'S WITH ARRAYS AND MAP()

Let's pass in function that will double each cell's value in our numbers array.

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( function (arrayCell) {
                                return arrayCell * 2;
                            }
);
```

We build an anonymous function for map's parameter, which takes in the contents of each cell of numbers and returns a doubled value to results.

12	4	3	9	8	6	10	1

USING FE'S WITH ARRAYS AND MAP()

Let's pass in function that will double each cell's value in our numbers array.

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( function (arrayCell) {
                                return arrayCell * 2;
                            }
);
```

Don't forget to close both your anonymous function with a } and the map method with a), while also adding a semicolon in order to execute the map.

12	4	3	9	8	6	10	1

USING FE'S WITH ARRAYS AND MAP()

Let's pass in function that will double each cell's value in our numbers array.

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( function (arrayCell) {
                                return arrayCell * 2;
                            }
);
```

12	4	3	9	8	6	10	1

doubled!

24							

USING FE'S WITH ARRAYS AND MAP()

Let's pass in function that will double each cell's value in our numbers array.

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( function (arrayCell) {
                                return arrayCell * 2;
                            }
);
```

12	4	3	9	8	6	10	1

doubled!

24	8						

USING FE'S WITH ARRAYS AND MAP()

Let's pass in function that will double each cell's value in our numbers array.

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( function (arrayCell) {
                                return arrayCell * 2;
                            }
);
```

12	4	3	9	8	6	10	1

doubled!

24	8	6					

USING FE'S WITH ARRAYS AND MAP()

Let's pass in function that will double each cell's value in our numbers array.

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( function (arrayCell) {
                                return arrayCell * 2;
                            }
);
```

12	4	3	9	8	6	10	1

doubled!

24	8	6	18				

USING FE'S WITH ARRAYS AND MAP()

Let's pass in function that will double each cell's value in our numbers array.

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( function (arrayCell) {
                                return arrayCell * 2;
                            }
);
```

12	4	3	9	8	6	10	1

doubled!

24	8	6	18	16			

USING FE'S WITH ARRAYS AND MAP()

Let's pass in function that will double each cell's value in our numbers array.

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( function (arrayCell) {
                                return arrayCell * 2;
                            }
);
```

12	4	3	9	8	6	10	1

doubled!

24	8	6	18	16	12		

USING FE'S WITH ARRAYS AND MAP()

Let's pass in function that will double each cell's value in our numbers array.

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( function (arrayCell) {
                                return arrayCell * 2;
                            }
);
```

12	4	3	9	8	6	10	1

doubled!

24	8	6	18	16	12	20	

USING FE'S WITH ARRAYS AND MAP()

Let's pass in function that will double each cell's value in our numbers array.

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( function (arrayCell) {
                                return arrayCell * 2;
                            }
);
```

12	4	3	9	8	6	10	1

doubled!

24	8	6	18	16	12	20	2

USING FE'S WITH ARRAYS AND MAP()

Let's pass in function that will double each cell's value in our numbers array.

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( function (arrayCell) {
                                return arrayCell * 2;
                            }
);
```

```
console.log(results);
```

→ [24, 8, 6, 18, 16, 12, 20, 2]

USING FE'S WITH ARRAYS AND MAP()

Let's pass in function that will double each cell's value in our numbers array.

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map( function (arrayCell) { return arrayCell * 2; } );
```

Short functions are often built in one line for clarity and simplicity.

```
console.log(results);
```

→ [24, 8, 6, 18, 16, 12, 20, 2]

Visit the Wondrous

FOREST OF FUNCTION EXPRESSIONS

```
var sweetAnnouncement = function () { alert("ATTENTION!\n CUPCAKES.\n\n...THAT IS ALL.") };
sweetAnnouncement();
```

```
var fruits = [ "Apple", "Orange", "Pineapple", "Cranberry", "Pomegranate" ];
var fruitJuice = fruits.map( function (fruit) { return "\n" + fruit + " juice"; } );
alert(fruitJuice);
```

```
function mystery () {

    *some mystery code...ooh, spooky.*

    return *a function expression*

}
```

A TICKET SYSTEM FOR THE FOREST THEME PARK

Using an array as a “queue” for Fast Pass delivery

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

fastPassQueue

"Cedar Coaster"

"Pines Plunge"

New items in the Queue are added at "the end of the line," just like in real life.

A TICKET SYSTEM FOR THE FOREST THEME PARK

Using an array as a “queue” for Fast Pass delivery

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

fastPassQueue

"Cedar Coaster"	"Pines Plunge"

"Birch Bumpers"

New items in the Queue are added at "the end of the line," just like in real life.

A TICKET SYSTEM FOR THE FOREST THEME PARK

Using an array as a “queue” for Fast Pass delivery

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

fastPassQueue

"Cedar Coaster"	"Pines Plunge"	"Birch Bumpers"

We know a method that adds cells to the back of arrays, right??

```
fastPassQueue.push("Pines Plunge");
```

"Pines Plunge"

A TICKET SYSTEM FOR THE FOREST THEME PARK

Using an array as a “queue” for Fast Pass delivery

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

fastPassQueue

"Cedar Coaster"	"Pines Plunge"	"Birch Bumpers"	"Pines Plunge"

We know a method that adds cells to the back of arrays, right??

```
fastPassQueue.push("Pines Plunge");
```

A TICKET SYSTEM FOR THE FOREST THEME PARK

Using an array as a “queue” for Fast Pass delivery

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

fastPassQueue

"Cedar Coaster"	"Pines Plunge"	"Birch Bumpers"	"Pines Plunge"

To empty the queue, however, we need a new method that removes cells from the front of the array!

A TICKET SYSTEM FOR THE FOREST THEME PARK

Using an array as a “queue” for Fast Pass delivery

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

fastPassQueue

~~"Cedar Coaster"~~	"Pines Plunge"	"Birch Bumpers"	"Pines Plunge"

```
fastPassQueue.shift();
```

A TICKET SYSTEM FOR THE FOREST THEME PARK

Using an array as a "queue" for Fast Pass delivery

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

fastPassQueue

"Pines Plunge"	"Birch Bumpers"	"Pines Plunge"

```
fastPassQueue.shift();
```

→ "Cedar Coaster"

The shift method will return the cell that it removes from the front of the array.

A TICKET SYSTEM FOR THE FOREST THEME PARK

Using an array as a “queue” for Fast Pass delivery

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

fastPassQueue

"Pines Plunge"	"Birch Bumpers"	"Pines Plunge"

```
fastPassQueue.shift();
```

→ "Cedar Coaster"

The shift method will return the cell that it removes from the front of the array.

A TICKET SYSTEM FOR THE FOREST THEME PARK

Using an array as a "queue" for Fast Pass delivery

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

fastPassQueue

"Pines Plunge"	"Birch Bumpers"	"Pines Plunge"

```
fastPassQueue.shift();
```

→ "Cedar Coaster"

```
fastPassQueue.length;
```

→ 3

Like pop and push, shift will automatically modify the array's length.

A TICKET SYSTEM FOR THE FOREST THEME PARK

Using an array as a “queue” for Fast Pass delivery

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

fastPassQueue

~~"Pines Plunge"~~	"Birch Bumpers"	"Pines Plunge"

```
fastPassQueue.shift();
```

→ "Cedar Coaster"

```
var firstFastPass = fastPassQueue.shift();
```

A TICKET SYSTEM FOR THE FOREST THEME PARK

Using an array as a “queue” for Fast Pass delivery

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

fastPassQueue

"Birch Bumpers"	"Pines Plunge"

```
fastPassQueue.shift();
```

→ "Cedar Coaster"

```
var firstFastPass = fastPassQueue.shift();
```

A TICKET SYSTEM FOR THE FOREST THEME PARK

Using an array as a "queue" for Fast Pass delivery

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

fastPassQueue

"Birch Bumpers"	"Pines Plunge"

```
fastPassQueue.shift();
```

→ "Cedar Coaster"

```
var firstFastPass = fastPassQueue.shift();
console.log(firstFastPass);
```

→ "Pines Plunge"

shift always returns the first cell, whether you use it in an expression, store it in a variable, or don't even use it at all.

A TICKET SYSTEM FOR THE FOREST THEME PARK

Using an array as a "queue" for Fast Pass delivery

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

fastPassQueue

"Birch Bumpers"	"Pines Plunge"

```
fastPassQueue.shift();
```

→ "Cedar Coaster"

```
var firstFastPass = fastPassQueue.shift();
console.log(firstFastPass);
```

→ "Pines Plunge"

Now, we'll build a simple ticket system using our queue!

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
function buildTicket ( allRides, passRides, pick ) {



}
```

This parameter will be the array of the rides and their wait times.

...and this will be the array of the next available Fast Pass rides.

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
function buildTicket ( allRides, passRides, pick ) {



}
```

This will be the actual ride for which our customer would like a ticket!

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
function buildTicket ( allRides, passRides, pick ) {
  if(passRides[0] == pick){


  }


}
```

If the next available Fast Pass is for the ride that the customer seeks, we'll give them that pass.

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
function buildTicket ( allRides, passRides, pick ) {
  if(passRides[0] == pick){
     var pass = passRides.shift();


  }


}
```

Here's our shift! It takes the front cell off the array and stores it in the pass variable.

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
function buildTicket ( allRides, passRides, pick ) {
  if(passRides[0] == pick){
     var pass = passRides.shift();
     return function ( ) { alert("Quick! You've got a Fast Pass to " + pass + "!");
            };
  }



}
```

Notice we are treating the function as an expression and returning it directly. No extra storage variable needed!

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
function buildTicket ( allRides, passRides, pick ) {
  if(passRides[0] == pick){
     var pass = passRides.shift();
     return function ( ) { alert("Quick! You've got a Fast Pass to " + pass + "!");
            };
  } else {
     for(var i = 0; i<allRides.length; i++){



     }
  }
}
```

To search for the ride the customer wants, we'll loop over the entire array of rides.

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
function buildTicket ( allRides, passRides, pick ) {
  if(passRides[0] == pick){
     var pass = passRides.shift();
     return function ( ) { alert("Quick! You've got a Fast Pass to " + pass + "!");
            };
  } else {
     for(var i = 0; i<allRides.length; i++){
        if(allRides[i][0] == pick){



        }
     }
  }
}
```

The ride names are contained within the first index of each subarray, or [0].

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
function buildTicket ( allRides, passRides, pick ) {
  if(passRides[0] == pick){
     var pass = passRides.shift();
     return function ( ) { alert("Quick! You've got a Fast Pass to " + pass + "!");
            };
  } else {
     for(var i = 0; i<allRides.length; i++){
        if(allRides[i][0] == pick){
           return function () { alert("A ticket is printing for " + pick + "!\n" +
                                "Your wait time is about " + allRides[i][1] + " minutes.");
           };
        }
     }
  }
}
```

The wait times are in the second index of the subarrays, or [1].

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
function buildTicket ( allRides, passRides, pick ) {
  if(passRides[0] == pick){
     var pass = passRides.shift();
     return function ( ) { alert("Quick! You've got a Fast Pass to " + pass + "!");
            };
  } else {
     for(var i = 0; i<allRides.length; i++){
        if(allRides[i][0] == pick){
           return function () { alert("A ticket is printing for " + pick + "!\n" +
                                "Your wait time is about " + allRides[i][1] + " minutes.");
           };

        }
     }
  }
}
```

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Birch Bumpers";
```

```
var ticket = buildTicket( parkRides, fastPassQueue, wantsRide );
```

When buildTicket returns the correct ticket function, we'll store it in a ticket variable for later use!

```
function buildTicket ( allRides, passRides, pick ) {
   ...
}
```

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Birch Bumpers";
```

```
var ticket = buildTicket( parkRides, fastPassQueue, wantsRide );
```

```
ticket();
```

To call the function contained in the `ticket` variable, we need a set of parentheses and a semicolon.

```
function buildTicket ( allRides, passRides, pick ) {
    ...
}
```

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Birch Bumpers";
```

```
var ticket = buildTicket( parkRides, fastPassQueue, wantsRide );
```

```
ticket();
```

```
function buildTicket ( allRides, passRides, pick ) {
    ...
}
```

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Cedar Coaster", "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Cedar Coaster";
```

Now, the desired ride matches the first Fast Pass!

```
var ticket = buildTicket( parkRides, fastPassQueue, wantsRide );
```

```
ticket();
```

```
function buildTicket ( allRides, passRides, pick ) {
    ...
}
```

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [                "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Cedar Coaster";
```

```
var ticket = buildTicket( parkRides, fastPassQueue, wantsRide );
```

```
ticket();
```

```
function buildTicket ( allRides, passRides, pick ) {
    ...
}
```

LET'S MAKE SOME TICKETS!

Since functions can be treated as expressions, they can also be returned like values!

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Cedar Coaster";
```

```
var ticket = buildTicket( parkRides, fastPassQueue, wantsRide );
```

```
ticket();
```

```
function buildTicket ( allRides, passRides, pick ) {
    ...
}
```

USING AN IMMEDIATELY-INVOKED FUNCTION

Calling returned functions instantly instead of variable storage

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Pines Plunge";
```

```
var ticket = buildTicket( parkRides, fastPassQueue, wantsRide );
```

```
function buildTicket ( allRides, passRides, pick ) {
    ...
}
```

USING AN IMMEDIATELY-INVOKED FUNCTION

Calling returned functions instantly instead of variable storage

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Pines Plunge";
```

```
buildTicket( parkRides, fastPassQueue, wantsRide );
```

```
function buildTicket ( allRides, passRides, pick ) {
    ...
}
```

USING AN IMMEDIATELY-INVOKED FUNCTION

Calling returned functions instantly instead of variable storage

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Pines Plunge";
```

```
buildTicket( parkRides, fastPassQueue, wantsRide );
```

```
function buildTicket ( allRides, passRides, pick ) {
    ...
}
```

USING AN IMMEDIATELY-INVOKED FUNCTION

Calling returned functions instantly instead of variable storage

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Pines Plunge";
```

```
buildTicket( parkRides, fastPassQueue, wantsRide );
```

```
( function ( ) {
    alert("Quick! You've got a Fast Pass to " + pass + "!");
} )
```

So far, all we get back is a function expression. Need more in order to call it!

The contents of pass are saved in a process called "closure," which we'll explore in the next level. For now, we know it's filled with the first available Fast Pass.

```
function buildTicket ( allRides, passRides, pick ) {
   ...
}
```

USING AN IMMEDIATELY-INVOKED FUNCTION

Calling returned functions instantly instead of variable storage

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Pines Plunge";
```

```
buildTicket( parkRides, fastPassQueue, wantsRide );
```

```
( function ( ) {
    alert("Quick! You've got a Fast Pass to " + pass + "!");
} )
```

```
function buildTicket ( allRides, passRides, pick ) {
   ...
}
```

USING AN IMMEDIATELY-INVOKED FUNCTION

Calling returned functions instantly instead of variable storage

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Pines Plunge";
```

```
buildTicket( parkRides, fastPassQueue, wantsRide )()              ;
```

```
( function ( ) {
    alert("Quick! You've got a Fast Pass to " + pass + "!");
} )()
```

Okay, now we've given our expression some parameter parentheses. We're on the right track!

```
function buildTicket ( allRides, passRides, pick ) {
   ...
}
```

USING AN IMMEDIATELY-INVOKED FUNCTION

Calling returned functions instantly instead of variable storage

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Pines Plunge";
```

```
buildTicket( parkRides, fastPassQueue, wantsRide )()              ;
```

```
( function ( ) {
    alert("Quick! You've got a Fast Pass to " + pass + "!");
} )()
```

```
function buildTicket ( allRides, passRides, pick ) {
   ...
}
```

USING AN IMMEDIATELY-INVOKED FUNCTION

Calling returned functions instantly instead of variable storage

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Pines Plunge";
```

```
buildTicket( parkRides, fastPassQueue, wantsRide )();
```

```
( function ( ) {
    alert("Quick! You've got a Fast Pass to " + pass + "!");
} )();
```

Yep, a semicolon gives the instruction to execute the function!

```
function buildTicket ( allRides, passRides, pick ) {
   ...
}
```

USING AN IMMEDIATELY-INVOKED FUNCTION

Calling returned functions instantly instead of variable storage

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Pines Plunge", "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Pines Plunge";
```

```
buildTicket( parkRides, fastPassQueue, wantsRide )();
```

```
( function ( ) {
    alert("Quick! You've got a Fast Pass to " + pass + "!");
} )();
```

```
function buildTicket ( allRides, passRides, pick ) {
   ...
}
```

USING AN IMMEDIATELY-INVOKED FUNCTION

Calling returned functions instantly instead of variable storage

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [                    "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Pines Plunge";
```

```
buildTicket( parkRides, fastPassQueue, wantsRide )();
```

```
( function ( ) {
    alert("Quick! You've got a Fast Pass to " + pass + "!");
} )();
```

```
function buildTicket ( allRides, passRides, pick ) {
   ...
}
```

USING AN IMMEDIATELY-INVOKED FUNCTION

Calling returned functions instantly instead of variable storage

```
var parkRides = [ ["Birch Bumpers", 40] , ["Pines Plunge", 55]
                  ["Cedar Coaster", 20] , ["Ferris Wheel of Firs", 90]  ];
```

```
var fastPassQueue = [ "Birch Bumpers", "Pines Plunge" ];
```

```
var wantsRide = "Pines Plunge";
```

```
buildTicket( parkRides, fastPassQueue, wantsRide )();
```

```
( function ( ) {
    alert("Quick! You've got a Fast Pass to " + pass + "!");
} )();
```

```
function buildTicket ( allRides, passRides, pick ) {
   ...
}
```

Visit the Wondrous

FOREST OF FUNCTION EXPRESSIONS

Explore

COLD CLOSURES COVE

LEVEL 2

COLD CLOSURES COVE

A PACKAGE DEAL

Guess what? Congratulations! You've already made a basic closure!

```
function buildTicket ( allRides, passRides, pick ) {
  if(passRides[0] == pick){
     var pass = fastAvail.shift();
     return function ( ) { alert("Quick! You've got a Fast Pass to " + pass + "!");
            };
  } else {
     for(var i = 0; i<allRides.length; i++){
        if(allRides[i][0] == pick){
           return function () { alert("A ticket is printing for " + pick + "!\n" +
                                "Your wait time is about " + allRides[i][1] + " minutes.");
           };
        }
     }
  }
}
```

The entire contents of one of these inner functions will still be available OUTSIDE the outermost function.

Returning a function from a function, complete with variables from an external scope, is called a closure.

A PACKAGE DEAL

A closure wraps up an entire environment, binding necessary variables from other scopes.

```
function  testClosure ( ) {

        var x = 4;
        return x;

}
```

Local Variable only!

```
testClosure();
```

→ 4

```
x;
```

→ undefined

A function's local variables aren't available once the function's scope is closed!

A PACKAGE DEAL

A closure wraps up an entire environment, binding necessary variables from other scopes.

The inner function can access the outer function's variables, because they "feel" like global variables.

```
function  testClosure ( ) {
        var x = 4;
        function closeX ( ) {
            return x;
        }
        return closeX;
}
```

Notice x does not need to be "stored" anywhere in closeX, not even as a parameter!

A PACKAGE DEAL

A closure wraps up an entire environment, binding necessary variables from other scopes.

```
function  testClosure ( ) {
        var x = 4;
        function closeX ( ) {
                return x;
        }
        return closeX;
}
```

```
var checkLocalX = testClosure();
```

```
checkLocalX();
```

→ 4

Even though `testClosure` has finished operating, its local variable is now bound within `checkLocalX`.

CLOSURES HELP IN FUNCTION "CONSTRUCTION ZONES"

A closure can make the creation of very similar functions ultra-efficient.

```
function buildCoveTicketMaker( transport ) {
        return function ( name ) {
           alert("Here is your transportation ticket via the " + transport + ".\n" +
                 "Welcome to the Cold Closures Cove, " + name + "!");
        }
}
```

CLOSURES HELP IN FUNCTION "CONSTRUCTION ZONES"

A closure can make the creation of very similar functions ultra-efficient.

```
function buildCoveTicketMaker( transport ) {
        return function ( name ) {
           alert("Here is your transportation ticket via the " + transport + ".\n" +
                 "Welcome to the Cold Closures Cove, " + name + "!");
        }
}
```

```
var getSubmarineTicket = buildCoveTicketMaker("Submarine");
```

```
var getBattleshipTicket = buildCoveTicketMaker("Battleship");
```

```
var getGiantSeagullTicket = buildCoveTicketMaker("Giant Seagull");
```

We give `buildCoveTicketMaker` the mode of transportation, which is closed into the returned anonymous function.

CLOSURES HELP IN FUNCTION "CONSTRUCTION ZONES"

A closure can make the creation of very similar functions ultra-efficient.

```
var getSubmarineTicket = buildCoveTicketMaker("Submarine");
```

```
var getBattleshipTicket = buildCoveTicketMaker("Battleship");
```

```
var getGiantSeagullTicket = buildCoveTicketMaker("Giant Seagull");
```

CLOSURES HELP IN FUNCTION “CONSTRUCTION ZONES”

A closure can make the creation of very similar functions ultra-efficient.

```
getSubmarineTicket
```

```
getBattleshipTicket
```

```
getGiantSeagullTicket
```

BEWARE: BOUND VARIABLES WON'T BE EVIDENT IN THE STORED FUNCTION

Examining the contents of our new variables doesn't reveal closures.

```
getSubmarineTicket;
```

```
function ( name ) {
   alert("Here is your transportation ticket via the " + transport + ".\n" +
         "Welcome to the Cold Closures Cove, " + name + "!");
}
```

Holds "Submarine"

```
getBattleshipTicket;
```

```
function ( name ) {
   alert("Here is your transportation ticket via the " + transport + ".\n" +
         "Welcome to the Cold Closures Cove, " + name + "!");
}
```

Holds "Battleship"

```
getGiantSeagullTicket;
```

```
function ( name ) {
   alert("Here is your transportation ticket via the " + transport + ".\n" +
         "Welcome to the Cold Closures Cove, " + name + "!");
}
```

Holds "Giant Seagull"

BEWARE: BOUND VARIABLES WON'T BE EVIDENT IN THE STORED FUNCTION

Examining the contents of our new variables doesn't reveal closures.

```
getSubmarineTicket;
```

```
function ( name ) {
   alert("Here is your transportation ticket via the " + transport + ".\n" +
         "Welcome to the Cold Closures Cove, " + name + "!");
}
```

```
getBattleshipTicket;
```

```
function ( name ) {
   alert("Here is your transportation ticket via the " + transport + ".\n" +
         "Welcome to the Cold Closures Cove, " + name + "!");
}
```

```
getGiantSeagullTicket;
```

```
function ( name ) {
   alert("Here is your transportation ticket via the " + transport + ".\n" +
         "Welcome to the Cold Closures Cove, " + name + "!");
}
```

Until we call any of these functions with a parameter, the name variable is still undefined.

LET'S MAKE SOME TICKETS!

Passing a name to any of our ticket makers will complete our ticket-making process.

```
getSubmarineTicket;
```

```
function ( name ) {
   alert("Here is your transportation ticket via the " + transport + ".\n" +
         "Welcome to the Cold Closures Cove, " + name + "!");
}
```

```
getBattleshipTicket;
```

```
function ( name ) {
   alert("Here is your transportation ticket via the " + transport + ".\n" +
         "Welcome to the Cold Closures Cove, " + name + "!");
}
```

```
getGiantSeagullTicket;
```

```
function ( name ) {
   alert("Here is your transportation ticket via the " + transport + ".\n" +
         "Welcome to the Cold Closures Cove, " + name + "!");
}
```

LET'S MAKE SOME TICKETS!

Passing a name to any of our ticket makers will complete our ticket-making process.

```
getSubmarineTicket;
```

```
getBattleshipTicket;
```

```
getGiantSeagullTicket;
```

LET'S MAKE SOME TICKETS!

Passing a name to any of our ticket makers will complete our ticket-making process.

```
getSubmarineTicket("Mario");
```

```
getBattleshipTicket("Luigi");
```

```
getGiantSeagullTicket("Bowser");
```

LET'S MAKE SOME TICKETS!

Passing a name to any of our ticket makers will complete our ticket-making process.

```
getSubmarineTicket("Mario");
```

```
getBattleshipTicket("Luigi");
```

```
getGiantSeagullTicket("Bowser");
```

Explore

COLD CLOSURES COVE

ADDING A PASSENGER TRACKER

Closure functions can even modify bound variables in the background

```
function buildCoveTicketMaker( transport ) {
        return function ( name ) {
           alert("Here is your transportation ticket via the " + transport + ".\n" +
                 "Welcome to the Cold Closures Cove, " + name + "!");
        }
}
```

ADDING A PASSENGER TRACKER

Closure functions can even modify bound variables in the background

```
function buildCoveTicketMaker( transport ) {
        var passengerNumber = 0;
        return function ( name ) {

           alert("Here is your transportation ticket via the " + transport + ".\n" +
                 "Welcome to the Cold Closures Cove, " + name + "!" +
                                                                  );
        }
}
```

We'll start every ticket maker's tracker at 0 passengers.

ADDING A PASSENGER TRACKER

Closure functions can even modify bound variables in the background

```
function buildCoveTicketMaker( transport ) {
        var passengerNumber = 0;
        return function ( name ) {
            passengerNumber++;
            alert("Here is your transportation ticket via the " + transport + ".\n" +
                  "Welcome to the Cold Closures Cove, " + name + "!" +
                                                                  );
        }
}
```

When a particular ticket maker is called, we know a new passenger should be added, so we'll increase the tracker.

ADDING A PASSENGER TRACKER

Closure functions can even modify bound variables in the background

```
function buildCoveTicketMaker( transport ) {
        var passengerNumber = 0;
        return function ( name ) {
            passengerNumber++;
            alert("Here is your transportation ticket via the " + transport + ".\n" +
                  "Welcome to the Cold Closures Cove, " + name + "!" +
                  "You are passenger #" + passengerNumber + "." );
        }
}
```

Each time a ticket is "printed," this passengerNumber will contain the precise amount of times this kind of ticket has been given.

ADDING A PASSENGER TRACKER

Closure functions can even modify bound variables in the background

```
function buildCoveTicketMaker( transport ) {
        var passengerNumber = 0;
        return function ( name ) {
            passengerNumber++;
            alert("Here is your transportation ticket via the " + transport + ".\n" +
                  "Welcome to the Cold Closures Cove, " + name + "!" +
                  "You are passenger #" + passengerNumber + "." );
        }
}
```

```
var getSubmarineTicket = buildCoveTicketMaker("Submarine");
getSubmarineTicket;
```

```
function (name) {
    passengerNumber++;
    alert("Here is your transportation ticket via the " + transport + ".\n" +
          "Welcome to the Cold Closures Cove, " + name + "!\n" +
          "You are passenger #" + passengerNumber + ".");
}
```

Notice that no initial value for passengerNumber is evident in our new function. It's value starts at 0 and is adjusted with each call to getSubmarineTicket.

ADDING A PASSENGER TRACKER

Closure functions can even modify bound variables in the background

```
function buildCoveTicketMaker( transport ) {
        var passengerNumber = 0;
        return function ( name ) {
            passengerNumber++;
            alert("Here is your transportation ticket via the " + transport + ".\n" +
                  "Welcome to the Cold Closures Cove, " + name + "!" +
                  "You are passenger #" + passengerNumber + "." );
        }
}
```

```
var getSubmarineTicket = buildCoveTicketMaker("Submarine");
getSubmarineTicket("Mario");
```

On our first call to the new getSubmarineTicket, passengerNumber is incremented to 1.

ADDING A PASSENGER TRACKER

Closure functions can even modify bound variables in the background

```
function buildCoveTicketMaker( transport ) {
        var passengerNumber = 0;
        return function ( name ) {
            passengerNumber++;
            alert("Here is your transportation ticket via the " + transport + ".\n" +
                  "Welcome to the Cold Closures Cove, " + name + "!" +
                  "You are passenger #" + passengerNumber + ".");
        }
}
```

```
var getSubmarineTicket = buildCoveTicketMaker("Submarine");
getSubmarineTicket("Toad");
```

Another call to getSubmarineTicket has passengerNumber incremented to 2! Wow, even though the function's local scope disappeared after Mario's ticket, it KEPT the progress of passengerNumber!

Explore

COLD CLOSURES COVE

LOOPS WITH CLOSURES: A CAUTIONARY TALE

Let's try to make a torpedo assigner for the Cove's Submarine

```
function assignTorpedo ( name, passengerArray ){



}
```

We'll pass in the name of a passenger, as well as a list of passengers.

LOOPS WITH CLOSURES: A CAUTIONARY TALE

Let's try to make a torpedo assigner for the Cove's Submarine

```
function assignTorpedo ( name, passengerArray ){
        var torpedoAssignment;

            This variable will hold a function that alerts name's torpedo assignment.




}
```

LOOPS WITH CLOSURES: A CAUTIONARY TALE

Let's try to make a torpedo assigner for the Cove's Submarine

```
function assignTorpedo ( name, passengerArray ){
        var torpedoAssignment;
        for (var i = 0; i<passengerArray.length; i++) {



        }

}
```

We'll loop over the list of passengers to find name.

LOOPS WITH CLOSURES: A CAUTIONARY TALE

Let's try to make a torpedo assigner for the Cove's Submarine

```
function assignTorpedo ( name, passengerArray ){
        var torpedoAssignment;
        for (var i = 0; i<passengerArray.length; i++) {
            if (passengerArray[i] == name) {
              torpedoAssignment = function ( ) {


              };
            }
        }

}
```

When we find the right name, we'll make a function that will hold our torpedo assignment closure.

LOOPS WITH CLOSURES: A CAUTIONARY TALE

Let's try to make a torpedo assigner for the Cove's Submarine

```
function assignTorpedo ( name, passengerArray ){
        var torpedoAssignment;
        for (var i = 0; i<passengerArray.length; i++) {
            if (passengerArray[i] == name) {
              torpedoAssignment = function ( ) {
                 alert("Ahoy, " + name + "!\n" +
                       "Man your post at Torpedo #" + (i+1) + "!");
              };
            }
        }

}
```

We'll close up the name variable and the loop counter i, and assign a person to the torpedo associated with their index value in the list (adjusted for zero).

LOOPS WITH CLOSURES: A CAUTIONARY TALE

Let's try to make a torpedo assigner for the Cove's Submarine

```
function assignTorpedo ( name, passengerArray ){
        var torpedoAssignment;
        for (var i = 0; i<passengerArray.length; i++) {
            if (passengerArray[i] == name) {
              torpedoAssignment = function ( ) {
                 alert("Ahoy, " + name + "!\n" +
                       "Man your post at Torpedo #" + (i+1) + "!");
              };
            }
        }
        return torpedoAssignment;
}
```

Finally, we'll hand the correct assignment back over to the global scope.

LOOPS WITH CLOSURES: A CAUTIONARY TALE

Let's try to make a torpedo assigner for the Cove's Submarine

```
function assignTorpedo ( name, passengerArray ){
        var torpedoAssignment;
        for (var i = 0; i<passengerArray.length; i++) {
            if (passengerArray[i] == name) {
              torpedoAssignment = function ( ) {
                 alert("Ahoy, " + name + "!\n" +
                        "Man your post at Torpedo #"
              };
            }
        }
        return torpedoAssignment;
}
```

Should be Torpedo #4!
What happened?

```
var subPassengers = ["Luke", "Leia", "Han", "Chewie", "Yoda", "R2-D2", "C-3PO", "Boba"];
```

```
var giveAssignment = assignTorpedo("Chewie", subPassengers);
```

```
giveAssignment();
```

CLOSURES BIND VALUES AT THE VERY LAST MOMENT

We have to pay close attention to return times and final variable states

```
function assignTorpedo ( name, passengerArray ){
        var torpedoAssignment;
        for (var i = 0; i<passengerArray.length; i++) {
            if (passengerArray[i] == name) {
              torpedoAssignment = function ( ) {
                 alert("Ahoy, " + name + "!\n" +
                       "Man your post at Torpedo #" + (i+1) + "!");
              };
            }
        }
        return torpedoAssignment;
}
```

Way before torpedoAssignment is returned, the i loop counter has progressed in value to 8 and stopped the loop.

```
var subPassengers = ["Luke", "Leia", "Han", "Chewie", "Yoda", "R2-D2", "C-3PO", "Boba"];
```

```
var giveAssignment = assignTorpedo("Chewie", subPassengers);
```

```
giveAssignment();
```

CLOSURES BIND VALUES AT THE VERY LAST MOMENT

We have to pay close attention to return times and final variable states

```
function assignTorpedo ( name, passengerArray ){
        var torpedoAssignment;
        for (var i = 0; i<passengerArray.length; i++) {
            if (passengerArray[i] == name) {
              torpedoAssignment = function ( ) {
                 alert("Ahoy, " + name + "!\n" +
                        "Man your post at Torpedo #" + (i+1) + "!");
              };
            }
        }
        return torpedoAssignment;
}
```

8+1=9

The function's actual return is the true "moment of closure," when the environment and all necessary variables are packaged up.

```
var subPassengers = ["Luke", "Leia", "Han", "Chewie", "Yoda", "R2-D2", "C-3PO", "Boba"];
```

```
var giveAssignment = assignTorpedo("Chewie", subPassengers);
```

```
giveAssignment();
```

WHAT CAN WE DO TO ENSURE THE CORRECT VALUE?

Several options exist for timing closures correctly

```
function assignTorpedo ( name, passengerArray ){
        var torpedoAssignment;
        for (var i = 0; i<passengerArray.length; i++) {
            if (passengerArray[i] == name) {
              torpedoAssignment = function ( ) {
                 alert("Ahoy, " + name + "!\n" +
                       "Man your post at Torpedo #" + (i+1) + "!");
              };
            }
        }
        return torpedoAssignment;
}
```

```
var subPassengers = ["Luke", "Leia", "Han", "Chewie", "Yoda", "R2-D2", "C-3PO", "Boba"];
```

WHAT CAN WE DO TO ENSURE THE CORRECT VALUE?

Several options exist for timing closures correctly

```
function assignTorpedo ( name, passengerArray ){

        for (var i = 0; i<passengerArray.length; i++) {
            if (passengerArray[i] == name) {
                                    function ( ) {
                alert("Ahoy, " + name + "!\n" +
                      "Man your post at Torpedo #" + (i+1) + "!");
              };
            }
        }

}
```

```
var subPassengers = ["Luke", "Leia", "Han", "Chewie", "Yoda", "R2-D2", "C-3PO", "Boba"];
```

WHAT CAN WE DO TO ENSURE THE CORRECT VALUE?

Several options exist for timing closures correctly

```
function assignTorpedo ( name, passengerArray ){

        for (var i = 0; i<passengerArray.length; i++) {
            if (passengerArray[i] == name) {
                return function ( ) {
                alert("Ahoy, " + name + "!\n" +
                      "Man your post at Torpedo #" + (i+1) + "!");
                };
            }
        }
}
```

Now the function will be immediately returned when the right name is found, locking i in place.

```
var subPassengers = ["Luke", "Leia", "Han", "Chewie", "Yoda", "R2-D2", "C-3PO", "Boba"];
```

WHAT CAN WE DO TO ENSURE THE CORRECT VALUE?

Several options exist for timing closures correctly

```
function assignTorpedo ( name, passengerArray ){

        for (var i = 0; i<passengerArray.length; i++) {
            if (passengerArray[i] == name) {
                return function ( ) {
                alert("Ahoy, " + name + "!\n" +
                      "Man your post at Torpedo #"
                };
            }
        }

}
```

An immediate return has the expected effect, because i is not allowed to progress!

```
var subPassengers = ["Luke", "Leia", "Han", "Chewie", "Yoda", "R2-D2", "C-3PO", "Boba"];
```

```
var giveAssignment = assignTorpedo("Chewie", subPassengers);
```

```
giveAssignment();
```

WHAT CAN WE DO TO ENSURE THE CORRECT VALUE?

We could also design the torpedo assigners a bit more like our ticket makers

```
function makeTorpedoAssigner ( passengerArray ) {

        return function ( name ) {



        };

}
```

This time our external function will only take in the passengerArray, and we'll let the returned function deal with a specific name.

```
var subPassengers = ["Luke", "Leia", "Han", "Chewie", "Yoda", "R2-D2", "C-3PO", "Boba"];
```

WHAT CAN WE DO TO ENSURE THE CORRECT VALUE?

We could also design the torpedo assigners a bit more like our ticket makers

```
function makeTorpedoAssigner ( passengerArray ) {

        return function ( name ) {
           for (var i = 0; i<passengerArray.length; i++) {


           }
        };

}
```

At this point, whatever passengerArray got passed in to makeTorpedoAssigner will be bound into the closure. Parameters are part of the environment, too!

```
var subPassengers = ["Luke", "Leia", "Han", "Chewie", "Yoda", "R2-D2", "C-3PO", "Boba"];
```

WHAT CAN WE DO TO ENSURE THE CORRECT VALUE?

We could also design the torpedo assigners a bit more like our ticket makers

```
function makeTorpedoAssigner ( passengerArray ) {

        return function ( name ) {
            for (var i = 0; i<passengerArray.length; i++) {
                if (passengerArray[i] == name) {
                    alert("Ahoy, " + name + "!\n" +
                          "Man your post at Torpedo #" + (i+1) + "!");
                }
            }
        };

}
```

Since we've put the loop inside the returned function, i will come directly from that local scope.

The only closed variable from the external scope is passengerArray, which never changes.

```
var subPassengers = ["Luke", "Leia", "Han", "Chewie", "Yoda", "R2-D2", "C-3PO", "Boba"];
```

NOW WE CAN PASS OUT TORPEDOES LIKE CANDY

TIE Fighter, dead ahead!...Er, underwater...

```
function makeTorpedoAssigner ( passengerArray ) {

        return function ( name ) {
           for (var i = 0; i<passengerArray.length; i++) {
              if (passengerArray[i] == name) {
                 alert("Ahoy, " + name + "!\n" +
                       "Man your post at Torpedo #"
              }
           }
        };

}
```

```
var subPassengers = ["Luke", "Leia", "Han", "Chewie", "Yoda", "R2-D2", "C-3PO", "Boba"];
```

```
var getTorpedoFor = makeTorpedoAssigner(subPassengers);
```

```
getTorpedoFor("Chewie");
```

NOW WE CAN PASS OUT TORPEDOES LIKE CANDY

TIE Fighter, dead ahead!...Er, underwater...

```
function makeTorpedoAssigner ( passengerArray ) {

        return function ( name ) {
           for (var i = 0; i<passengerArray.length; i++) {
              if (passengerArray[i] == name) {
                 alert("Ahoy, " + name + "!\n" +
                       "Man your post at Torpedo #"
              }
           }
        };

}
```

```
var subPassengers = ["Luke", "Leia", "Han", "Chewie", "Yoda", "R2-D2", “C-3PO”, “Boba”];
```

```
var getTorpedoFor = makeTorpedoAssigner(subPassengers);
```

```
getTorpedoFor( "R2-D2" );
```

Explore

COLD CLOSURES COVE

Climb
THE HOISTING HILLS

LEVEL 3

THE HOISTING HILLS

THE IMPORTANCE OF LOAD ORDER

Ensuring that every line of code can execute when it's needed

THE IMPORTANCE OF LOAD ORDER

Ensuring that every line of code can execute when it's needed

"HOISTING" WITHIN A JAVASCRIPT SCOPE

First, memory is set aside for all necessary variables and declared functions.

We build it like this...

```
function sumOfSquares (a, b){

    var x = add(a*a, b*b);
    return x;

    function add (c, d){
        var a = c + d;
        return a;
    }
}
```

...but JavaScript loads it like this.

```
function sumOfSquares (a, b){

    var x = undefined;
    function add (c, d){
        var a = c + d;
        return a;
    }

    x = add(a*a, b*b);
    return x;
}
```

Declared stuff that needs space in memory is first "hoisted" to the top of scope before any operational code is run.

CODING CAREFULLY FOR SMOOTH EXECUTION

Some examples of the impact of hoisting

```
function getMysteryNumber () {

  function chooseMystery() {
          return 12;
  }

  return chooseMystery();

  function chooseMystery() {
          return 7;
  }

}
```

Loads like this

```
function getMysteryNumber () {

  function chooseMystery() {
          return 12;
  }

  function chooseMystery() {
          return 7;
  }

  return chooseMystery();

}
```

```
getMysteryNumber( );
```

→ ?

The chooseMystery function is redefined by the time all hoisting is finished!

```
getMysteryNumber( );
```

→ 7

CODING CAREFULLY FOR SMOOTH EXECUTION

Function Expressions are never hoisted! They are treated as assignments.

```
function getMysteryNumber () {

  var chooseMystery = function() {
        return 12;
  }

  return chooseMystery();

  var chooseMystery = function() {
        return 7;
  }

}
```

```
getMysteryNumber( );
```

→ ?

Loads like this

```
function getMysteryNumber () {

  var chooseMystery = undefined;
  var chooseMystery = undefined;

  chooseMystery = function () {
          return 12;
  };

  return chooseMystery();

  chooseMystery = function () {
          return 7;
  }
}
```

Unreachable!

```
getMysteryNumber( );
```

→ 12

CODING CAREFULLY FOR SMOOTH EXECUTION

Function Expressions are never hoisted! They are treated as assignments.

```
function getMysteryNumber () {

  return chooseMystery();


  var chooseMystery = function() {
        return 12;
  }

  var chooseMystery = function() {
        return 7;
  }

}
```

```
getMysteryNumber( );
```

→ ?

Loads like this

```
function getMysteryNumber () {

  var chooseMystery = undefined;
  var chooseMystery = undefined;
  return chooseMystery();

  chooseMystery = function () {
          return 12;
  };

  chooseMystery = function () {
          return 7;
  }
}
```

Unreachable!

```
getMysteryNumber( );
```

→ ERROR

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {

    *if the train is full*
        execute a function that alerts a message
        that no seats remain and then returns false.


    *if the train is NOT full*
        execute a function that alerts a message with
        how many seats remain, and then returns true.



}
```

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {

    if (numPassengers == capacity)
        execute a function that alerts a message
        that no seats remain and then returns false.


    *if the train is NOT full*
        execute a function that alerts a message with
        how many seats remain, and then returns true.



}
```

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {

    if (numPassengers == capacity) {
        noSeats();
    }



    *if the train is NOT full*
        execute a function that alerts a message with
        how many seats remain, and then returns true.



}
```

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {

    if (numPassengers == capacity) {
         noSeats();
    } else {

    }



}
```

execute a function that alerts a message with
how many seats remain, and then returns true.

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {

    if (numPassengers == capacity) {
         noSeats();
    } else {
         seatsAvail();
    }

}
```

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {

    if (numPassengers == capacity) {
        noSeats();
    } else {
        seatsAvail();
    }

    var noSeats = function (){
        alert("No seats left!");
        return false;
    }



}
```

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {

    if (numPassengers == capacity) {
         noSeats();
    } else {
         seatsAvail();
    }

    var noSeats = function (){
        alert("No seats left!");
        return false;
    }
    var seatsAvail = function(){
        alert("There are " + (capacity - numPassengers) + " seats left!");
        return true;
    }
}
```

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {

    if (numPassengers == capacity) {
         noSeats();
    } else {
         seatsAvail();
    }

    var noSeats = function (){
        alert("No seats left!");
        return false;
    }
    var seatsAvail = function(){
        alert("There are " + (capacity - numPassengers) + " seats left!");
        return true;
    }
}
```

```
capacityStatus(60, 60);
```

?

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {

    if (numPassengers == capacity) {
         noSeats();
    } else {
         seatsAvail();
    }

    var noSeats = function (){
        alert("No seats left!");
        return false;
    }
    var seatsAvail = function(){
        alert("There are " + (capacity - numPassengers) + " seats left!");
        return true;
    }
}
```

```
capacityStatus(60, 60);
```

?

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {
    var noSeats = undefined;
    var seatsAvail = undefined;
    if (numPassengers == capacity) {
        noSeats();
    } else {
        seatsAvail();
    }
    noSeats = function (){
        alert("No seats left!");
        return false;
    }
    seatsAvail = function(){
        alert("There are " + (capacity - numPassengers) + " seats left!");
        return true;
    }
}
```

Doesn't exist yet!

```
capacityStatus(60, 60);
```

?

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {
    var noSeats = undefined;
    var seatsAvail = undefined;
    if (numPassengers == capacity) {
        noSeats();
    } else {
        seatsAvail();
    }
    noSeats = function (){
        alert("No seats left!");
        return false;
    }
    seatsAvail = function(){
        alert("There are " + (capacity - numPassengers) + " seats left!");
        return true;
    }
}
```

Doesn't exist yet!

```
capacityStatus(60, 60);
```

→ ERROR

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {

    if (numPassengers == capacity) {
         noSeats();
    } else {
         seatsAvail();
    }

    var noSeats = function (){
        alert("No seats left!");
        return false;
    }
    var seatsAvail = function(){
        alert("There are " + (capacity - numPassengers) + " seats left!");
        return true;
    }
}
```

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {

    var noSeats = function (){
        alert("No seats left!");
        return false;
    }
    var seatsAvail = function(){
        alert("There are " + (capacity - numPassengers) + " seats left!");
        return true;
    }

    if (numPassengers == capacity) {
         noSeats();
    } else {
         seatsAvail();
    }
}
```

```
capacityStatus(60, 60);
```

?

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {
    var noSeats = undefined;
    var seatsAvail = undefined;
    noSeats = function (){
        alert("No seats left!");
        return false;
    }
    seatsAvail = function(){
        alert("There are " + (capacity - numPassengers) + " seats left!");
        return true;
    }
    if (numPassengers == capacity) {
        noSeats();
    } else {
        seatsAvail();
    }
}
```

```
capacityStatus(60, 60);
```

?

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {
    var noSeats = undefined;
    var seatsAvail = undefined;
    noSeats = function (){
        alert("No seats left!");
        return false;
    }
    seatsAvail = function(){
        alert("There are " + (capacity - numPassengers) + " seats left!");
        return true;
    }
    if (numPassengers == capacity) {
        noSeats();
    } else {
        seatsAvail();
    }
}
```

```
capacityStatus(60, 60);
```

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {

    if (numPassengers == capacity) {
         noSeats();
    } else {
         seatsAvail();
    }


    var noSeats = function (){
        alert("No seats left!");
        return false;
    }
    var seatsAvail = function (){
        alert("There are " + (capacity - numPassengers) + " seats left!");
        return true;
    }
}
```

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {

    if (numPassengers == capacity) {
         noSeats();
    } else {
         seatsAvail();
    }


    function noSeats (){
        alert("No seats left!");
        return false;
    }
    function seatsAvail (){
        alert("There are " + (capacity - numPassengers) + " seats left!");
        return true;
    }
}
```

```
capacityStatus(20, 60);
```

?

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {
    function noSeats (){
        alert("No seats left!");
        return false;
    }
    function seatsAvail (){
        alert("There are " + (capacity - numPassengers) + " seats left!");
        return true;
    }

    if (numPassengers == capacity) {
        noSeats();
    } else {
        seatsAvail();
    }

}
```

```
capacityStatus(20, 60);
```

?

HOW MIGHT THIS AFFECT OUR EARLIER TRAIN SYSTEM?

Hoisting in a function that returns a capacity status for the JavaScript Express

```
function capacityStatus (numPassengers, capacity) {
    function noSeats (){
        alert("No seats left!");
        return false;
    }
    function seatsAvail (){
        alert("There are " + (capacity - numPassengers) + " seats left!");
        return true;
    }

    if (numPassengers == capacity) {
        noSeats();
    } else {
        seatsAvail();
    }

}
```

```
capacityStatus(20, 60);
```

Climb
THE HOISTING HILLS

See the Shimmering
OCEAN OF OBJECTS

LEVEL 4

OCEAN OF OBJECTS

OBJECTS ARE "CONTAINERS" OF RELATED INFORMATION

Multiple pieces of data, called properties, are grouped within an Object

OBJECT

property 1
property 2
property 3
property 4
property 5
property 6

All of these properties "belong" to this Object.

WE CAN REPRESENT EVERYDAY STUFF WITH JS OBJECTS

Since common things have “bits” of related info, they make good Object examples

OBJECT

property 1
property 2
property 3
property 4
property 5
property 6

WE CAN REPRESENT EVERYDAY STUFF WITH JS OBJECTS

Since common things have “bits” of related info, they make good Object examples

BOOK

property 1
property 2
property 3
property 4
property 5
property 6

WE CAN REPRESENT EVERYDAY STUFF WITH JS OBJECTS

Since common things have “bits” of related info, they make good Object examples

BOOK

title	numChapters
author	numPages
publisher	illustrator

Each property is an important bit of data associated with a book.

WE CAN REPRESENT EVERYDAY STUFF WITH JS OBJECTS

Since common things have “bits” of related info, they make good Object examples

BOX

height	volume
width	material
length	contents

Because an Object contains multiple bits of info, it's often called a "composite value."

AN OBJECT'S PROPERTIES CAN BE ASSIGNED VALUES

Like with everyday objects, properties can point to specific amounts or qualities

BOX

height	: 6	volume	: 480
width	: 8	material	: "cardboard"
length	: 10	contents	: booksArray

Properties can refer to numbers, strings, arrays, functions, and even other Objects!

CREATING AN OBJECT WITH JAVASCRIPT

There are multiple ways to build Objects...let's look first at the "Object literal."

BOX

height	: 6	volume	: 480
width	: 8	material	: "cardboard"
length	: 10	contents	: booksArray

```
var myBox = { };
```

A set of curly brackets says to make a new object...in this case, however, it's an empty one with no properties.

CREATING AN OBJECT WITH JAVASCRIPT

There are multiple ways to build Objects...let's look first at the "Object literal."

BOX

height	: 6	volume	: 480
width	: 8	material	: "cardboard"
length	: 10	contents	: booksArray

```
var myBox = { height: 6 };
```

Adding a property involves creating a name for the property using a string, and then assigning a value to it using a :.

CREATING AN OBJECT WITH JAVASCRIPT

There are multiple ways to build Objects...let's look first at the "Object literal."

BOX

height	: 6	volume	: 480
width	: 8	material	: "cardboard"
length	: 10	contents	: booksArray

```
var myBox = { height: 6, width: 8, length: 10, volume: 480 };
```

Multiple properties are separated by commas.

CREATING AN OBJECT WITH JAVASCRIPT

There are multiple ways to build Objects...let's look first at the "Object literal."

BOX

height	: 6	volume	: 480
width	: 8	material	: "cardboard"
length	: 10	contents	: booksArray

```
var myBox = { height: 6, width: 8, length: 10, volume: 480,
              material: "cardboard",
              contents: ["Great Expectations", "The Remains of the Day", "Peter Pan"]
            };
```

Sweet, a box Object complete with properties!

OBJECT PROPERTIES WILL ALSO ACCEPT VARIABLES

We can initialize the ‘contents’ property with a booksArray variable

BOX

height	: 6	volume	: 480
width	: 8	material	: “cardboard”
length	: 10	contents	: booksArray

```
var myBox = { height: 6, width: 8, length: 10, volume: 480,
              material: "cardboard",
              contents: ["Great Expectations", "The Remains of the Day", "Peter Pan"]
            };
```

OBJECT PROPERTIES WILL ALSO ACCEPT VARIABLES

We can initialize the ‘contents’ property with a booksArray variable

BOX

height	: 6	volume	: 480
width	: 8	material	: “cardboard”
length	: 10	contents	: booksArray

```
var booksArray = ["Great Expectations", "The Remains of the Day", "Peter Pan"];
var myBox = { height: 6, width: 8, length: 10, volume: 480,
              material: "cardboard",
              contents: ["Great Expectations", "The Remains of the Day", "Peter Pan"]
            };
```

OBJECT PROPERTIES WILL ALSO ACCEPT VARIABLES

We can initialize the ‘contents’ property with a booksArray variable

BOX

height	: 6	volume	: 480
width	: 8	material	: “cardboard”
length	: 10	contents	: booksArray

```
var booksArray = ["Great Expectations", "The Remains of the Day", "Peter Pan"];
var myBox = { height: 6, width: 8, length: 10, volume: 480,
              material: "cardboard",
              contents: booksArray
            };
```

REFERENCING AN OBJECT'S PROPERTIES

We can take a peek at any particular property of an object using the dot operator

```
var booksArray = ["Great Expectations", "The Remains of the Day", "Peter Pan"];
var myBox = { height: 6, width: 8, length: 10, volume: 480,
              material: "cardboard",
              contents: booksArray
            };
```

```
myBox.width;
```

→ 8

```
myBox.materials;
```

→ "cardboard"

```
myBox.contents;
```

→ ["Great Expectations", "The Remains of the Day", "Peter Pan"]

CHANGING PROPERTY VALUES

The dot operator also allows modification of properties, even using methods

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan"]
```

myBox

```
height:   6
width:    8
length:   10
volume:   480
material:   "cardboard"
contents:   booksArray
```

```
myBox.width = 12;
```

CHANGING PROPERTY VALUES

The dot operator also allows modification of properties, even using methods

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan"]
```

myBox

```
height:   6
width:    12
length:   10
volume:   480
material:   "cardboard"
contents:   booksArray
```

```
myBox.width = 12;
```

CHANGING PROPERTY VALUES

The dot operator also allows modification of properties, even using methods

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan"]
```

myBox

```
height:   6
width:    12
length:   10
volume:   480
material:   "cardboard"
contents:   booksArray
```

```
myBox.width = 12;
console.log( myBox.width );
```

→ 12

Oops, that makes our volume incorrect!

CHANGING PROPERTY VALUES

The dot operator also allows modification of properties, even using methods

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan"]
```

myBox

```
height:   6
width:    12
length:   10
volume:   480
material:   "cardboard"
contents:   booksArray
```

```
myBox.width = 12;
console.log( myBox.width );
```

→ 12

```
myBox.volume = myBox.length * myBox.width * myBox.height;
```

CHANGING PROPERTY VALUES

The dot operator also allows modification of properties, even using methods

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan"]
```

myBox

```
height:   6
width:    12
length:   10
volume:   720
material:   "cardboard"
contents:   booksArray
```

```
myBox.width = 12;
console.log( myBox.width );
```

→ 12

```
myBox.volume = myBox.length * myBox.width * myBox.height;
```

CHANGING PROPERTY VALUES

The dot operator also allows modification of properties, even using methods

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan"]
```

myBox

```
height:   6
width:    12
length:   10
volume:   720
material:   "cardboard"
contents:   booksArray
```

```
myBox.width = 12;
console.log( myBox.width );
```

→ 12

```
myBox.volume = myBox.length * myBox.width * myBox.height;
console.log( myBox.volume );
```

→ 720

CHANGING PROPERTY VALUES

The dot operator also allows modification of properties, even using methods

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan"]
```

myBox

```
height:   6
width:    12
length:   10
volume:   720
material:   "cardboard"
contents:   booksArray
```

```
myBox.contents.push("On The Road");
```

myBox.contents returns an entire Array, to which we can easily apply Array methods.

CHANGING PROPERTY VALUES

The dot operator also allows modification of properties, even using methods

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan"]
```

myBox

```
height:   6
width:    12
length:   10
volume:   720
material:   "cardboard"
contents:   booksArray
```

```
myBox.contents.push("On The Road");
```

CHANGING PROPERTY VALUES

The dot operator also allows modification of properties, even using methods

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:   6
width:    12
length:   10
volume:   720
material:   "cardboard"
contents:   booksArray
```

```
myBox.contents.push("On The Road");
```

Whoa, we modified the external array outside of myBox ? How'd we do that?

CHANGING PROPERTY VALUES

The dot operator also allows modification of properties, even using methods

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:   6
width:    12
length:   10
volume:   720
material:   "cardboard"
contents:   booksArray
```

```
myBox.contents.push("On The Road");
```

Passing in booksArray only makes a REFERENCE to the external Array contained in the variable, and doesn't create a brand new copied Array.

CHANGING PROPERTY VALUES

The dot operator also allows modification of properties, even using methods

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:   6
width:    12
length:   10
volume:   720
material:   "cardboard"
contents:   booksArray
```

```
myBox.contents.push("On The Road");
```

Since we're only referring to booksArray, pushing to myBox.contents (or using any Array method) will just modify booksArray!

```
console.log( myBox.contents );
```

→ `["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]`

CHANGING PROPERTY VALUES

The dot operator also allows modification of properties, even using methods

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:   6
width:    12
length:   10
volume:   720
material:   "cardboard"
contents:   booksArray
```

```
myBox.contents.push("On The Road");
```

```
console.log( myBox.contents );
```

→ `["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]`

CHANGING PROPERTY VALUES

The dot operator also allows modification of properties, even using methods

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:   6
width:    12
length:   10
volume:   720
material:   "cardboard"
contents:   booksArray
```

```
myBox.contents.push("On The Road");
```

```
console.log( myBox.contents );
```

→ `["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]`

```
console.log( booksArray );
```

→ `["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]`

ADDING PROPERTY VALUES POST-CREATION

Even after an object has been created, properties can continue to be added

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:   6
width:    12
length:   10
volume:   720
material:   "cardboard"
contents:   booksArray
```

```
myBox.weight = 24;
```

ADDING PROPERTY VALUES POST-CREATION

Even after an object has been created, properties can continue to be added

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:   6
width:    12
length:   10
volume:   720
material:   "cardboard"
contents:   booksArray
weight:   24
```

```
myBox.weight = 24;
```

The myBox Object looks around for a weight property. Finding none, it creates one!

ADDING PROPERTY VALUES POST-CREATION

Even after an object has been created, properties can continue to be added

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:  6
width:   12
length:  10
volume:  720
material:   "cardboard"
contents:   booksArray
weight:  24
destination1:  "Orlando"
destination2:  "Miami"
```

```
myBox.weight = 24;
```

```
myBox.destination1 = "Orlando";
```

```
myBox.destination2 = "Miami";
```

A SECOND WAY OF ACCESSING OR CREATING PROPERTIES

We can use brackets on Objects in similar fashion to accessing array indices

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:  6
width:   12
length:  10
volume:  720
material:   "cardboard"
contents:   booksArray
weight:  24
destination1:  "Orlando"
destination2:  "Miami"
```

```
myBox["volume"];
```

→ 720

```
myBox["material"];
```

→ "cardboard"

An object is like an Array whose indices can be accessed with strings (with quotes) instead of numbers.

A SECOND WAY OF ACCESSING OR CREATING PROPERTIES

We can use brackets on Objects in similar fashion to accessing array indices

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:  6
width:   12
length:  10
volume:  720
material:   "cardboard"
contents:   booksArray
weight:  24
destination1: "Orlando"
destination2: "Miami"
```

```
myBox["# of stops"] = 2;
```

Since the brackets use or "check for" an exactly matching string, we can also create properties with spaces and characters in their names.

A SECOND WAY OF ACCESSING OR CREATING PROPERTIES

We can use brackets on Objects in similar fashion to accessing array indices

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:  6   width:   12
length:  10  volume:  720
material:   "cardboard"
contents:   booksArray
weight:  24
destination1: "Orlando"
destination2: "Miami"
"# of stops": 2
```

```
myBox["# of stops"] = 2;
```

Since the brackets use or "check for" an exactly matching string, we can also create properties with spaces and characters in their names.

A SECOND WAY OF ACCESSING OR CREATING PROPERTIES

We can use brackets on Objects in similar fashion to accessing array indices

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:  6   width:   12
length:  10  volume:  720
material:   "cardboard"
contents:   booksArray
weight:  24
destination1:  "Orlando"
destination2:  "Miami"
"# of stops": 2
```

```
myBox["# of stops"] = 2;
```

```
console.log( myBox."# of stops" );
```

→ ERROR

No such syntax. Can't put a string after a dot. Beware!

A SECOND WAY OF ACCESSING OR CREATING PROPERTIES

We can use brackets on Objects in similar fashion to accessing array indices

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:  6   width:   12
length:  10  volume:  720
material:   "cardboard"
contents:   booksArray
weight:  24
destination1:  "Orlando"
destination2:  "Miami"
"# of stops": 2
```

```
myBox["# of stops"] = 2;
```

```
console.log( myBox."# of stops" );
```

→ ERROR

```
console.log( myBox["# of stops"] );
```

→ 2

Thus, key names with spaces can only be accessed with brackets!

BRACKETS ENABLE DYNAMIC PROPERTY ACCESS

Since brackets take expressions, we can avoid hard-coding every property access

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:  6    width:   12
length:  10   volume:  720
material:   "cardboard"
contents:   booksArray
weight:  24
destination1:  "Orlando"
destination2:  "Miami"
"# of stops": 2
```

```
for(var i = 1, i <= myBox["# of stops"]; i++){
    console.log( myBox["destination" + i] );
}
```

→ Orlando

→ Miami

We can place string-based expressions in the brackets to construct specific property names.

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Each book in our ‘contents’ property could be a Book object

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:  6   width:   12
length:  10  volume:  720
material:   "cardboard"
contents:   booksArray
weight:  24
destination1: "Orlando"
destination2: "Miami"
"# of stops": 2
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

First, we'll delete our contents property with the delete keyword.

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:  6    width:   12
length:  10   volume:  720
material:   "cardboard"
contents:   booksArray
weight:  24
destination1:  "Orlando"
destination2:  "Miami"
"# of stops":  2
```

```
delete myBox.contents;
```

The delete keyword will completely delete the entire contents property...not just the value associated with that property.

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

First, we'll delete our contents property with the delete keyword.

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:  6   width:   12
length:  10  volume:  720
material:   "cardboard"

weight:  24
destination1: "Orlando"
destination2: "Miami"
"# of stops": 2
```

```
delete myBox.contents;
```

→ true

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

First, we'll delete our contents property with the delete keyword.

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:  6   width:   12
length:  10  volume:  720
material:   "cardboard"
weight:  24
destination1: "Orlando"
destination2: "Miami"
"# of stops": 2
```

```
delete myBox.contents;
```

→ `true`

```
console.log( booksArray );
```

→ `["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]`

Additionally, we've only deleted the property name and the reference, but not the original `booksArray` *outside the Box.*

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

First, we'll delete our contents property with the delete keyword.

booksArray

```
["Great Expectations", "The Remains of the Day", "Peter Pan", "On The Road"]
```

myBox

```
height:  6   width:   12
length:  10  volume:  720
material:   "cardboard"
weight:  24
destination1:  "Orlando"
destination2:  "Miami"
"# of stops": 2
```

```
delete myBox.contents;
```

→ true

```
delete myBox.nonexistentProperty;
```

→ true

Watch out, though...delete will return true each time, regardless of whether the property existed or not! Think of it as asking: is this property gone?

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Now, we'll build a function that creates Book objects and adds them to our Box

myBox

```
height:  6   width:   12
length:  10  volume:  720
material:   "cardboard"
weight:  24
destination1: "Orlando"
destination2: "Miami"
"# of stops": 2
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Now, we'll build a function that creates Book objects and adds them to our Box

myBox

```
height:  6   width:   12
length:  10  volume:  720
material:   "cardboard"
weight:  24
destination1: "Orlando"
destination2: "Miami"
"# of stops": 2
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Now, we'll build a function that creates Book objects and adds them to our Box

myBox

```
height: 6   length: 10   width: 12   volume:  720   weight:  24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Now, we'll build a function that creates Book objects and adds them to our Box

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2          "# of Books": 0
```

We'll create a property that tracks the number of books, set initially to zero. Our function will use this value to dynamically assign property names.

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Now, we'll build a function that creates Book objects and adds them to our Box

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2          "# of Books": 0
```

Each time we create and add a Book object, we'll increase the number of books in our Box.

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

We'll concatenate the current book # with "book" to get our property name.

Our Book's properties will come from the function parameters we've passed in.

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Now, we'll build a function that creates Book objects and adds them to our Box

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2         "# of Books": 0
```

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Let's add some books!

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2          "# of Books": 0
```

```
addBook(myBox, "Great Expectations", "Charles Dickens");
```

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Let's add some books!

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2          "# of Books": 1
```

New book, new number of books.

```
addBook(myBox, "Great Expectations", "Charles Dickens");
```

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Let's add some books!

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2          "# of Books": 1
 book1: { title: "Great Expectations", author: "Charles Dickens"}
```

The correct book number in the property name has been dynamically assigned.

```
addBook(myBox, "Great Expectations", "Charles Dickens");
```

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Let's add some books!

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2         "# of Books": 1
 book1: { title: "Great Expectations", author: "Charles Dickens"}
```

```
addBook(myBox, "Great Expectations", "Charles Dickens");
```

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Let's add some books!

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2         "# of Books": 1
 book1: { title: "Great Expectations", author: "Charles Dickens"}
```

```
addBook(myBox, "The Remains of the Day", "Kazuo Ishiguro");
```

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Let's add some books!

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2         "# of Books": 2
 book1: { title: "Great Expectations", author: "Charles Dickens"}
```

```
addBook(myBox, "The Remains of the Day", "Kazuo Ishiguro");
```

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Let's add some books!

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2          "# of Books": 2
 book1: { title: "Great Expectations", author: "Charles Dickens"}
 book2: { title: "The Remains of the Day", author: "Kazuo Ishiguro"}
```

```
addBook(myBox, "The Remains of the Day", "Kazuo Ishiguro");
```

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Let's add some books!

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2          "# of Books": 2
 book1: { title: "Great Expectations", author: "Charles Dickens"}
 book2: { title: "The Remains of the Day", author: "Kazuo Ishiguro"}
```

```
addBook(myBox,"Peter Pan", "J. M. Barrie");
```

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Let's add some books!

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2          "# of Books": 3
 book1: { title: "Great Expectations", author: "Charles Dickens"}
 book2: { title: "The Remains of the Day", author: "Kazuo Ishiguro"}
```

```
addBook(myBox,"Peter Pan", "J. M. Barrie");
```

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Let's add some books!

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2          "# of Books": 3
 book1: { title: "Great Expectations", author: "Charles Dickens"}
 book2: { title: "The Remains of the Day", author: "Kazuo Ishiguro"}
 book3: { title: "Peter Pan", author: "J. M. Barrie"}
```

```
addBook(myBox,"Peter Pan", "J. M. Barrie");
```

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Let's add some books!

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2          "# of Books": 3
 book1: { title: "Great Expectations", author: "Charles Dickens"}
 book2: { title: "The Remains of the Day", author: "Kazuo Ishiguro"}
 book3: { title: "Peter Pan", author: "J. M. Barrie"}
```

```
addBook(myBox,"On the Road", "Jack Kerouac");
```

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Let's add some books!

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2          "# of Books": 4
  book1: { title: "Great Expectations", author: "Charles Dickens"}
  book2: { title: "The Remains of the Day", author: "Kazuo Ishiguro"}
  book3: { title: "Peter Pan", author: "J. M. Barrie"}
```

```
addBook(myBox,"On the Road", "Jack Kerouac");
```

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

CHANGING OUR CONTENTS TO INDIVIDUAL OBJECTS

Let's add some books!

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2          "# of Books": 4
  book1: { title: "Great Expectations", author: "Charles Dickens"}
  book2: { title: "The Remains of the Day", author: "Kazuo Ishiguro"}
  book3: { title: "Peter Pan", author: "J. M. Barrie"}
  book4: { title: "On the Road", author: "Jack Kerouac"}
```

```
addBook(myBox,"On the Road", "Jack Kerouac");
```

```
function addBook (box, name, writer){
    box["# of Books"]++;
    box["book" + box["# of Books"]] = {title: name, author: writer};
}
```

REFERENCING OBJECTS WITHIN OBJECTS

Use the dot extension or subsequent bracket notation to get to deeper properties

myBox

```
height: 6   length: 10   width: 12   volume: 720   weight: 24
material: "cardboard"   destination1: "Orlando"   destination2: "Miami"
"# of stops": 2          "# of Books": 4
  book1: { title: "Great Expectations", author: "Charles Dickens"}
  book2: { title: "The Remains of the Day", author: "Kazuo Ishiguro"}
  book3: { title: "Peter Pan", author: "J. M. Barrie"}
  book4: { title: "On the Road", author: "Jack Kerouac"}
```

```
console.log( myBox.book1.title );
```

→ Great Expectations

```
console.log( myBox["book4"]["author"] );
```

→ Jack Kerouac

See the Shimmering
OCEAN OF OBJECTS

AN AQUARIUM OBJECT FILLED WITH OTHER OBJECTS

Let's first build some add/remove functionality for creatures and environment toys

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Marlin: { type: "fish", species: "clownfish", length: 4.1 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false }
};
```

```
function addCritter( container, name, type, species, length ){
    container[name] = {type: type, species: species, length: length};
}
```

```
function addToy( container, name, type, material, moves ){
    container[name] = {type: type, material: material, moves: moves};
}
```

Wouldn't it be nice if these functions belonged only to the aquarium instead of an entire program? Let's try adding one.

AN AQUARIUM OBJECT FILLED WITH OTHER OBJECTS

Let's first build some add/remove functionality for creatures and environment toys

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Marlin: { type: "fish", species: "clownfish", length: 4.1 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false }
};
```

```
function addCritter( container, name, type, species, length ){
    container[name] = {type: type, species: species, length: length};
}
```

PROPERTIES CAN ALSO BE FUNCTIONS

An Object's function properties are often called its "methods"

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Marlin: { type: "fish", species: "clownfish", length: 4.1 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){

    }

};
```

We add a new property to our aquarium that takes the name of our addCritter function. Then we build an anonymous function.

```
function addCritter( container, name, type, species, length ){
    container[name] = {type: type, species: species, length: length};
}
```

PROPERTIES CAN ALSO BE FUNCTIONS

An Object's function properties are often called its "methods"

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Marlin: { type: "fish", species: "clownfish", length: 4.1 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){

    }

};
```

Our container parameter now disappears, since we are making the function BELONG TO that very container.

```
function addCritter( container, name, type, species, length ){
    container[name] = {type: type, species: species, length: length};
}
```

THE VERY USEFUL "THIS" KEYWORD

"This" always refers to the owner Object of the function in which the "this" is used.

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Marlin: { type: "fish", species: "clownfish", length: 4.1 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    }
};
```

When called with this, addCritter says: Hey, aquarium! Make a new property called name and assign to it a new Object with these properties!

```
function addCritter( container, name, type, species, length ){
    container[name] = {type: type, species: species, length: length};
}
```

WOOHOO, A PROPERTY THAT HOLDS A FUNCTION!

Our addCritter function is now available as a property on the aquarium Object

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Marlin: { type: "fish", species: "clownfish", length: 4.1 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    }

};
```

Let's add a creature!

WOOHOO, A PROPERTY THAT HOLDS A FUNCTION!

Our addCritter function is now available as a property on the aquarium Object

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Marlin: { type: "fish", species: "clownfish", length: 4.1 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    }
};
```

```
aquarium.addCritter("Bubbles", "fish", "yellow tang", 5.6);
```

We call the function just like referencing any other property in aquarium, but we also pass it a set of appropriate parameters.

WOOHOO, A PROPERTY THAT HOLDS A FUNCTION!

Our addCritter function is now available as a property on the aquarium Object

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Marlin: { type: "fish", species: "clownfish", length: 4.1 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.addCritter("Bubbles", "fish", "yellow tang", 5.6);
```

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Marlin: { type: "fish", species: "clownfish", length: 4.1 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.addCritter("Bubbles", "fish", "yellow tang", 5.6);
```

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
      Nemo: { type: "fish", species: "clownfish", length: 3.7 },
      Marlin: { type: "fish", species: "clownfish", length: 4.1 },
      Dory: { type: "fish", species: "blue tang", length: 6.2 },
      Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
      "Coral Castle": { type: "environment", material: "coquina", moves: false },
      "Dragon Statue": { type: "environment", material: "plastic", moves: false },
      addCritter: function ( name, type, species, length ){
         this[name] = {type: type, species: species, length: length};
      },
      Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Marlin: { type: "fish", species: "clownfish", length: 4.1 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {


};
```

All we will need to delete any property, whether creature or toy, is its name.

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Marlin: { type: "fish", species: "clownfish", length: 4.1 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {
    var temp = this[name];

};
```

A temp variable will help us hold on to the Object that we remove. This way we'll still have access to it outside the aquarium.

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Marlin: { type: "fish", species: "clownfish", length: 4.1 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {
    var temp = this[name];
    delete this[name];

};
```

Next we remove the property from the Owner object, in this case, the aquarium.

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Marlin: { type: "fish", species: "clownfish", length: 4.1 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {
    var temp = this[name];
    delete this[name];
    return temp;
};
```

Finally we return the temp variable, so that we can still have a reference to the removed Object.

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Marlin: { type: "fish", species: "clownfish", length: 4.1 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {
    var temp = this[name];
    delete this[name];
    return temp;
};
```

```
var fishOutOfWater = aquarium.takeOut("Marlin");
```

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
      Nemo: { type: "fish", species: "clownfish", length: 3.7 },

      Dory: { type: "fish", species: "blue tang", length: 6.2 },
      Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
      "Coral Castle": { type: "environment", material: "coquina", moves: false },
      "Dragon Statue": { type: "environment", material: "plastic", moves: false },
      addCritter: function ( name, type, species, length ){
         this[name] = {type: type, species: species, length: length};
      },
      Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {
    var temp = this[name];
    delete this[name];
    return temp;
};
```

```
var fishOutOfWater = aquarium.takeOut("Marlin");
```

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
      Nemo: { type: "fish", species: "clownfish", length: 3.7 },


      Dory: { type: "fish", species: "blue tang", length: 6.2 },
      Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
      "Coral Castle": { type: "environment", material: "coquina", moves: false },
      "Dragon Statue": { type: "environment", material: "plastic", moves: false },
      addCritter: function ( name, type, species, length ){
         this[name] = {type: type, species: species, length: length};
      },
      Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {
    var temp = this[name];
    delete this[name];
    return temp;
};
```

```
var fishOutOfWater = aquarium.takeOut("Marlin");
```

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

Uh oh! Notice that we lost Marlin's name! Let's fix that problem with some property trickery.

```
aquarium.takeOut = function ( name ) {
    var temp = this[name];
    delete this[name];
    return temp;
};
```

```
var fishOutOfWater = aquarium.takeOut("Marlin");
console.log( fishOutOfWater );
```

→ `Object {type: "fish", species: "clownfish", length: 4.1}`

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {
    var temp = this[name];
    delete this[name];
    return temp;
};
```

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {
    this[name].name = name;
    var temp = this[name];
    delete this[name];
    return temp;
};
```

The first name in this line of code finds the desired Object in the aquarium using the parameter as a property name.

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {
    this[name].name = name;
    var temp = this[name];
    delete this[name];
    return temp;
};
```

Coming after a dot, the second name creates a new property IN the Object we want to remove! Notice that this is NOT the same as the function's parameter!

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {
    this[name].name = name;
    var temp = this[name];
    delete this[name];
    return temp;
};
```

The third name assigns the old property name to the newly created name property in the removed Object. Sneaky!

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

Woohoo, no identity crisis for Marlin!

```
aquarium.takeOut = function ( name ) {
    this[name].name = name;
    var temp = this[name];
    delete this[name];
    return temp;
};
```

```
var fishOutOfWater = aquarium.takeOut("Marlin");
console.log( fishOutOfWater );
```

→ `Object {type: "fish", species: "clownfish", length: 4.1, name: "Marlin"}`

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    "Dragon Statue": { type: "environment", material: "plastic", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {
    this[name].name = name;
    var temp = this[name];
    delete this[name];
    return temp;
};
```

```
var toy = aquarium.takeOut("Dragon Statue");
```

Our removal method works for toy Objects too!

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
      Nemo: { type: "fish", species: "clownfish", length: 3.7 },
      Dory: { type: "fish", species: "blue tang", length: 6.2 },
      Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
      "Coral Castle": { type: "environment", material: "coquina", moves: false },


          addCritter: function ( name, type, species, length ){
              this[name] = {type: type, species: species, length: length};
          },
      Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {
    this[name].name = name;
    var temp = this[name];
    delete this[name];
    return temp;
};
```

```
var toy = aquarium.takeOut("Dragon Statue");
```

Our removal method works for toy Objects too!

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
      Nemo: { type: "fish", species: "clownfish", length: 3.7 },
      Dory: { type: "fish", species: "blue tang", length: 6.2 },
      Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
      "Coral Castle": { type: "environment", material: "coquina", moves: false },


          addCritter: function ( name, type, species, length ){
              this[name] = {type: type, species: species, length: length};
          },
      Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {
    this[name].name = name;
    var temp = this[name];
    delete this[name];
    return temp;
};
```

```
var toy = aquarium.takeOut("Dragon Statue");
```

Our removal method works for toy Objects too!

HMM...OUR TANK'S A LITTLE FULL

Let's build another method that removes any object from our aquarium

```
var aquarium = {
    Nemo: { type: "fish", species: "clownfish", length: 3.7 },
    Dory: { type: "fish", species: "blue tang", length: 6.2 },
    Peach: { type: "echinoderm", species: "starfish", length: 5.3 },
    "Coral Castle": { type: "environment", material: "coquina", moves: false },
    addCritter: function ( name, type, species, length ){
        this[name] = {type: type, species: species, length: length};
    },
    Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
};
```

```
aquarium.takeOut = function ( name ) {
    this[name].name = name;
    var temp = this[name];
    delete this[name];
    return temp;
};
```

```
var toy = aquarium.takeOut("Dragon Statue");
console.log( toy );
```

→ `Object {type: "environment", material: "coquina", moves: false, name: "Dragon Statue"}`

See the Shimmering
OCEAN OF OBJECTS

COUNTING FISH IN OUR TANK

What if we wanted to know how many fish our tank has at any given time?

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

```
Nemo: { type: "fish", species: "clownfish", length: 3.7 }
```

COUNTING FISH IN OUR TANK

What if we wanted to know how many fish our tank has at any given time?

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

```
Nemo: { type: "fish", species: "clownfish", length: 3.7 }
Dory: { type: "fish", species: "blue tang", length: 6.2 }
```

COUNTING FISH IN OUR TANK

What if we wanted to know how many fish our tank has at any given time?

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

```
Nemo: { type: "fish", species: "clownfish", length: 3.7 }
Dory: { type: "fish", species: "blue tang", length: 6.2 }
Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
```

COUNTING FISH IN OUR TANK

What if we wanted to know how many fish our tank has at any given time?

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

✅ ✅ ✅ ❌

```
Nemo: { type: "fish", species: "clownfish", length: 3.7 }
Dory: { type: "fish", species: "blue tang", length: 6.2 }
Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
Peach: { type: "echinoderm", species: "starfish", length: 5.3 }
```

COUNTING FISH IN OUR TANK

What if we wanted to know how many fish our tank has at any given time?

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

✅ ✅ ✅ ❌

❌

```
Nemo: { type: "fish", species: "clownfish", length: 3.7 }
Dory: { type: "fish", species: "blue tang", length: 6.2 }
Bubbles: { type: "fish", species: "yellow tang", length: 5.6 }
Peach: { type: "echinoderm", species: "starfish", length: 5.3 }
"Coral Castle": { type: "environment", material: "coquina", moves: false }
```

COUNTING FISH IN OUR TANK

What if we wanted to know how many fish our tank has at any given time?

```
var aquarium = {🐠, 🐟, 🐡, ⭐, 🏰, addCritter, takeOut};
```

✅ ✅ ✅ ❌ ❌

❌

```
addCritter: function ( name, type, species, length ){
   this[name] = {type: type, species: species, length: length};
}
```

COUNTING FISH IN OUR TANK

What if we wanted to know how many fish our tank has at any given time?

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

✅ ✅ ✅ ❌ ❌ ❌

❌

```
addCritter: function ( name, type, species, length ){
    this[name] = {type: type, species: species, length: length};
}
takeOut: function ( name ){
    this[name].name = name;
    var temp = this[name];
    delete this[name];
    return temp;
}
```

COUNTING FISH IN OUR TANK

What if we wanted to know how many fish our tank has at any given time?

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

✅ ✅ ✅ ❌ ❌ ❌ ❌ = 3 fish total

COUNTING FISH IN OUR TANK

What if we wanted to know how many fish our tank has at any given time?

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

✅ ✅ ✅ ❌ ❌ ❌ ❌ = 3 fish total

```
aquarium.length;
```

→ `undefined`

Hmm, uh oh. Generic Objects don't have a native `length` like Arrays and Strings do, so we can't use that in a loop format in order to get to each property.

ENUMERATION WITH THE FOR-IN LOOP

The for-in loop allows us to access each enumerable property in turn.

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

```
for ( var key in aquarium ) {

}
```

The in keyword looks "in" the Object to its right and finds each enumerable property in turn. Think of it like accessing each index of an Array.

ENUMERATION WITH THE FOR-IN LOOP

The for-in loop allows us to access each enumerable property in turn.

```
var aquarium = {🐠, 🐟, 🐡, 🐟, 🏰, addCritter, takeOut};
```

```
for ( var key in aquarium ) {
    console.log(key);
}
```

→ Nemo
→ Dory
→ Bubbles
→ Peach
→ Coral Castle
→ addCritter
→ takeOut

Logging out each property produces only their names as strings.

key in aquarium

ENUMERATION WITH THE FOR-IN LOOP

The for-in loop allows us to access each enumerable property in turn.

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

```
for ( var key in aquarium ) {
    console.log(key);
}
```

→ Nemo
→ Dory
→ Bubbles
→ Peach
→ Coral Castle
→ addCritter
→ takeOut

Logging out each property produces only their names as strings.

item key in aquarium

ENUMERATION WITH THE FOR-IN LOOP

Now we need a way to determine which properties in ‘aquarium’ are fish!

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

```
for ( var key in aquarium ) {

}
```

ENUMERATION WITH THE FOR-IN LOOP

Now we need a way to determine which properties in ‘aquarium’ are fish!

```
var aquarium = {🐠, 🐟, 🐡, 🐟, 🏖, addCritter, takeOut};
```

```
var numFish = 0;
for ( var key in aquarium ) {
      if ( aquarium[key].type == "fish" ) {

      }
}
```

Since key contains the string name of a property, we can use it in a set of brackets as an expression.

ENUMERATION WITH THE FOR-IN LOOP

Now we need a way to determine which properties in ‘aquarium’ are fish!

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

```
var numFish = 0;
for ( var key in aquarium ) {
      if ( aquarium[key].type == "fish" ) {

      }
}
```

Once we've accessed the Object that key refers to, we can seek its own type property, and check to see if the current Object is a fish.

ENUMERATION WITH THE FOR-IN LOOP

Now we need a way to determine which properties in ‘aquarium’ are fish!

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

?

```
var numFish = 0;
for ( var key in aquarium ) {
      if ( aquarium[key].type == "fish" ) {
            numFish++;
      }
}
```

```
aquarium["addCritter"].type;
```

→ `undefined`

```
undefined == "fish";
```

→ `false`

ENUMERATION WITH THE FOR-IN LOOP

Now we need a way to determine which properties in ‘aquarium’ are fish!

```
var aquarium = { , , , , , addCritter, takeOut};
```

```
var numFish = 0;
for ( var key in aquarium ) {
      if ( aquarium[key].type == "fish" ) {
            numFish++;
      }
}
```

```
console.log(numFish);
```

→ 3

Current Property	aquarium[property]	has .type?	aquarium[property].type	type == fish?	numFish
Nemo	aquarium[“Nemo”]	YES	“fish”	TRUE	1
Dory	aquarium[“Dory”]	YES	“fish”	TRUE	2
Bubbles	aquarium[“Bubbles”]	YES	“fish”	TRUE	3
Peach	aquarium[“Peach”]	YES	“echinoderm”	FALSE	3
Coral	aquarium[“Coral Castle”]	YES	“environment”	FALSE	3
addCritter	aquarium[“addCritter”]	no	undefined	FALSE	3
takeOut	aquarium[“takeOut”]	no	undefined	FALSE	3

ADDING OUR FISH COUNTER TO THE AQUARIUM

We'll need to build a function property using our loop

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

```
var numFish = 0;
for ( var key in aquarium ) {
      if ( aquarium[key].type == "fish" ) {
            numFish++;
      }
}
```

ADDING OUR FISH COUNTER TO THE AQUARIUM

We'll need to build a function property using our loop

```
var aquarium = {🐠, 🐟, 🐡, 🐟, 🐟, addCritter, takeOut};
```

```
aquarium.countFish = function ( ) {
        var numFish = 0;
        for ( var key in aquarium ) {
              if ( aquarium[key].type == "fish" ) {
                    numFish++;
              }
        }

}
```

ADDING OUR FISH COUNTER TO THE AQUARIUM

We'll need to build a function property using our loop

```
var aquarium = { , , , , , addCritter, takeOut};
```

```
aquarium.countFish = function ( ) {
        var numFish = 0;
        for ( var key in          ) {
              if (          [key].type == "fish" ) {
                    numFish++;
              }
        }

}
```

ADDING OUR FISH COUNTER TO THE AQUARIUM

We'll need to build a function property using our loop

```
var aquarium = {🐠, 🐟, 🐡, 🌟, 🏰, addCritter, takeOut};
```

```
aquarium.countFish = function ( ) {
        var numFish = 0;
        for ( var key in this ) {
              if ( this[key].type == "fish" ) {
                    numFish++;
              }
        }
}
```

Remember, since countFish will be "owned" by aquarium, it will use the this keyword to refer to it as an owner Object.

ADDING OUR FISH COUNTER TO THE AQUARIUM

We'll need to build a function property using our loop

```
var aquarium = {🐠, 🐟, 🐡, 🐟, 🐟, addCritter, takeOut};
```

```
aquarium.countFish = function ( ) {
        var numFish = 0;
        for ( var key in this) {
              if ( this[key].type == "fish" ) {
                    numFish++;
              }
        }
        return numFish;
}
```

ADDING OUR FISH COUNTER TO THE AQUARIUM

We'll need to build a function property using our loop

```
var aquarium = {🐠, 🐟, 🐡, 🐟, 🐟, addCritter, takeOut, countFish};
```

```
aquarium.countFish = function ( ) {
        var numFish = 0;
        for ( var key in this ) {
                if ( this[key].type == "fish" ) {
                        numFish++;
                }
        }
        return numFish;
}
```

```
aquarium.countFish();
```

→ 3

ADDING OUR FISH COUNTER TO THE AQUARIUM

We'll need to build a function property using our loop

```
var aquarium = {🐠, 🐟, 🐡, 🐟, 🐟, addCritter, takeOut, countFish};
```

```
aquarium.countFish = function ( ) {
        var numFish = 0;
        for ( var key in this ) {
                if ( this[key].type == "fish" ) {
                        numFish++;
                }
        }
        return numFish;
}
```

```
var poorDory = aquarium.takeOut("Dory");
```

ADDING OUR FISH COUNTER TO THE AQUARIUM

We'll need to build a function property using our loop

```
var aquarium = {🐠,      🐡, 🌟, 🏰, addCritter, takeOut, countFish};
```

```
aquarium.countFish = function ( ) {
        var numFish = 0;
        for ( var key in this ) {
              if ( this[key].type == "fish" ) {
                    numFish++;
              }
        }
        return numFish;
}
```

```
var poorDory = aquarium.takeOut("Dory");
```

ADDING OUR FISH COUNTER TO THE AQUARIUM

We'll need to build a function property using our loop

```
var aquarium = {🐠, 🐤, ⭐, 🏰, addCritter, takeOut, countFish};
```

```
aquarium.countFish = function ( ) {
        var numFish = 0;
        for ( var key in this ) {
                if ( this[key].type == "fish" ) {
                        numFish++;
                }
        }
        return numFish;
}
```

```
var poorDory = aquarium.takeOut("Dory");
```

→ true

```
aquarium.countFish();
```

→ 2

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

OBJECT
PROTOTYPE

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

OBJECT PROTOTYPE

constructor	valueOf	toLocaleString
toString		isPrototypeOf
propertyIsEnumerable		hasOwnProperty

THE OBJECT'S PARENT IS CALLED ITS "PROTOTYPE"

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

OBJECT PROTOTYPE

```
constructor          valueOf          toLocaleString
toString                              isPrototypeOf
propertyIsEnumerable                  hasOwnProperty
```

THE OBJECT'S PARENT IS CALLED ITS "PROTOTYPE"

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

OBJECT PROTOTYPE

- constructor
- valueOf
- toLocaleString
- toString
- isPrototypeOf
- propertyIsEnumerable
- hasOwnProperty

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

OBJECT PROTOTYPE

- constructor
- valueOf
- toLocaleString
- toString
- isPrototypeOf
- propertyIsEnumerable
- hasOwnProperty

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.

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

length	shift()	join()
pop()	reverse()	reduce()
push()	sort()	slice()

```
var aquarium = { … };
```

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

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

length	shift()	join()
pop()	reverse()	reduce()
push()	sort()	slice()

```
var aquarium = { … };
```

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

myArray.	myArray.	myArray.
myArray.	myArray.	myArray.
myArray.	myArray.	myArray.

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

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

length	concat()	toUpperCase()
charAt()	indexOf()	toLowerCase()
trim()	replace()	substring()

```
var aquarium = { … };
```

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

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

length	concat()	toUpperCase()
charAt()	indexOf()	toLowerCase()
trim()	replace()	substring()

```
var aquarium = { … };
```

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

```
myString.
myString.
myString.
```

```
myString.
myString.
myString.
```

```
myString.
myString.
myString.
```

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

```
var aquarium = { … };
```

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

```
myString.length
myString.charAt()
myString.trim()
myString.concat()
myString.indexOf()
myString.replace()
myString.toUpperCase()
myString.toLowerCase()
myString.substring()
```

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.

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

```
var aquarium = { … };
```

```
var myNumber = 6;
```

```
myNumber.toFixed()
myNumber.toExponential()
myNumber.toPrecision()
```

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!

name call()
bind() apply()

ARRAY Prototype

STRING Prototype

NUMBER Prototype

FUNCTION Prototype

```
var aquarium = { … };
```

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

myFunction. myFunction.

myFunction. myFunction.

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.

```
myString.length
myString.charAt()
myString.trim()
myString.toString()
myString.concat()
myString.indexOf()
myString.replace()
myString.valueOf()
myString.toUpperCase()
myString.toLowerCase()
myString.substring()
myString.hasOwnProperty()
myString.toLocaleString()
myString.isPrototypeOf()
myString.constructor()
myString.propertyIsEnumerable()
```

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()
```

```
myString.hasOwnProperty()
```

INHERITANCE AVOIDS DUPLICATE MEMORY STORAGE

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

```
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

Since the String prototype owns charAt, the search stops there, and the String prototype grants myString access to the function. Woot, inheritance!

```
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;

};
```

We get a counter variable ready...

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.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

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

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()

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()

Some Shoes

laceColor
laceUp()
jewels
bowPosition
dimensionalTravel()

takeOff()

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?"); }
};
```

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

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

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

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!";
};
```

TORNADO
Prototype

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

→ `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!";
};
```

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

→ "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