
# Introduction to "This"

We've seen how functions can be used to give a name to a set of statements so that you can call them independently and how they can be assigned to variables and protected from the global scope. There's another use of functions that I would like to talk to you about. That brings up the keyword `this`.

# Functions to Construct Objects

Functions can also be used to construct objects to take advantage of JavaScript's object oriented capabilities. You can use a function to define the template of an object and then instantiate instances of that object using the `new` keyword.

```
var example = function(str) {
  this.name = str;
};
var exampleJudy = new example("Judy");
var exampleFred = new example("Fred");

console.log(exampleJudy.name);
// "Judy"

console.log(exampleFred.name);
// "Fred"
```

Using this approach, you can create objects that you can instantiate by calling a function with the `new` keyword to generate objects constructed out of properties based on the parameters you pass to the function.

# Adding Methods to Constructors

Objects constructed this way are not limited to properties. You can also create methods that can be executed, since in JavaScript, a method or a function is just a value that can be assigned to a variable. So, you can assign it to a single property of your new object.

```
var example = function(str) {
  this.name = str;
  this.greet = function() {
    return("Hello, " + this.name);
  };
};
var exampleJudy = new example("Judy");
var exampleFred = new example("Fred");

console.log(exampleJudy.greet());
// "Hello, Judy"
console.log(exampleFred.greet());
// "Hello, Fred"
```

In this way, we can create objects in JavaScript that can have their own properties and their own methods. You can instantiate multiple independent instances of each of these objects and use them independently of each other.

# Private and Public Properties

We've talked a little bit about scope before in the context of the global scope and functional scope. But in an object-oriented context, JavaScript also respects the scope of variables that are declared inside of constructor functions. The variables that are declared with `var` in a constructor are considered private to the child objects and their own functions.

This mean that if you want to create a variable that's going to be unique to each child object and not accessible to anything outside of that child object, you can declare it was a `var` inside of the constructor function for that object.

```
var example = function(str) {
  var special = "Judy";
  this.name = str;
  this.greet = function() {
    if (this.name === special) {
      return("Well, isn't that special?");
    } else {
      return("Hello, " + this.name);
    }
  };
};
var exampleJudy = new example("Judy");
var exampleFred = new example("Fred");

console.log(exampleJudy.greet());
// "Well, isn't that special?"
console.log(exampleJudy.special);
// undefined
console.log(exampleFred.greet());
// "Hello, Fred"
console.log(exampleFred.special);
// undefined
```

Using this approach, we can see how we can use JavaScript as an object -orientated language to create template constructor functions and instantiate instances of those objects, and each of those instances can have public properties, public methods, and also private variables that can't be accessed outside of the instance.