Advanced JavaScript

JS

Agenda

- Functions as a first class citizen in JavaScript
- Understanding the apply/call methods
- The context of this
- Closures
- Scopes
- Object orientation with JavaScript
- Unit testing in JavaScript
- Debugging JavaScript

Closures/Functions are first-class objects in JS

- In JavaScript, functions are first-class objects; that is, they coexist with, and can be treated like, any other JavaScript object.
- In addition, functions can:
  - be created as literals.
  - assigned to variables, array entries, and properties of other objects.
  - passed as arguments to functions.
  - returned as values from functions.
  - contain functions with it.
- (source functions.js)

Closures/Functions are first-class objects in JS

- And more than just being treated with the same respect as other objects types, functions have a special capability in that they can be invoked.
- Invocation is frequently done in an asynchronous manner.

JavaScript Functions

- What really happens when a function is called?
- As it turns out, the manner in which a function is invoked has a huge impact the context of the function.
- The context the function is the 'this' declaration.

Functions (cont)

- There are actually four different ways to invoke a function, each with their own nuances.

  1. As a function, in which the function is invoked in a straightforward manner.

  2. As a method, which ties the invocation to an object, enabling object-oriented programming.

  3. As a constructor, in which a new object is brought into being.

  4. Via their apply() or call() methods.

1. Invocation as a function

- This type of invocation occurs when a function is invoked using the () operator.
- Some simple examples:

```
function ninja(){};
ninja();
var samurai = function(){};
samurai();
```

2.) As a method, which ties the invocation to an object, enabling object-orientation

- When a function is assigned to a property of an object and the invocation occurs by referencing the function using that property, then the function is invoked as a method of that object.

```
var o = {};
o.eat = function(){};
o.eat();
```

- NOTE: a function declared this way is considered anonymous..we are just assigning the annoymous function to a a member of the object o.

3. As a constructor, in which a new object is brought into being.

- Invoking a function as a constructor is a powerful feature of JavaScript because when a constructor is invoked, the following special actions take place:
  - A new empty object is created.
  - This object is passed to the constructor as the this parameter, and thus becomes the constructor’s function context.
  - In the absence of any explicit return value, the new object is returned as the constructor’s value.
- (source constructors.js)

A side note on constructors..

- Functions and methods are generally named starting with a verb that describes what they do start with a lowercase letter.
  - startCount();
- Constructors on the other hand, are usually named as a noun that describes the object that's being constructed, and start with an uppercase character;
  - Ninja()

More on Instantiation

- We’ve already noted that functions can serve a dual purpose: as “normal” functions, and as constructors.
- Because of this, it may not always be clear to users of our code which is which.
- Therefore, follow best practices so your intentions are clear.

4. Via their apply() or call() methods

- So far, we've looked at the context of the 'this' parameter. For global functions, it is always window; for methods, the method’s owning object; and for constructors, a newly created object instance.
- But what if we wanted to make it whatever we wanted?
- What if we wanted to set it explicitly?
- Can we even do this??

Using the apply() and call() methods

- JavaScript provides a means for us to invoke a function, and to explicitly specify any object we want as the function context. The way that we do this is through the use of one of two methods that exist for every function: apply() or call().
- To invoke a function using its apply() method, we pass two parameters to apply():
  - the object to be used as the function context, and an array of values to be used as the invocation arguments.
- The call() method is used in a similar manner, except that the arguments are passed directly in the argument list rather than as an array.
- (source apply-call.js)
- (source apply-call2.js)

Arguments of a function

- If there are a different number of arguments than there are parameters, no error is raised; JavaScript is perfectly fine with this situation, and deals with it as follows:
  - If more arguments are supplied than there are parameters, the “excess” arguments are simply not assigned to parameter names. We’ll see in just a bit that even though the arguments aren’t assigned to parameter names, we still have a way to get at them.
  - If there are more parameters than there are arguments, the parameters that have no corresponding argument are set to undefined.

Arguments of a function

- And, very interestingly, all function invocations are also passed two implicit parameters:
  - arguments and this.
- All functions are implicitly passed this important parameter, which will give our functions the power to handle any number of passed arguments.
- Even if we only define a certain number of parameters, we'll always be able to access all passed arguments through the arguments parameter.
- (source apply-call.js)

This context of this

- Whenever a function is invoked, 2 variables are always passed:
  - arguments that were provided on the function call
  - an implicit parameter named this
- The this variable refers to an object that is implicitly associated with the function invocation and is termed the **function context**.

The context of this

- The function context is a notion that those coming from OO languages such as Java will think that they understand – that this points to an instance of the class within which the method is defined. But beware!
- Think of 'this' as the *invocation* context instead.
- (source function-contexts.js)
- (source function-contexts.html)

Anonymous functions

- Functions that are assigned to a variable and without a name is considered anonymous.
- Functions can have a name when assigned to a variable
- Name can be refereneced within the function (ie. for recursion).
- (souce annoymous_functions.js)

Immediately Invoked Function Expressions

- (function(){})()
- The result of this code is an expression that creates, executes, and discards a function all in the same statement.
- Additionally, since we're dealing with a function that can have a closure as any other, we also have access to all outside variables in the same scope as the statement. As it turns out, this simple construct, called an immediate function, ends up becoming immensely useful as we'll soon see.
- (source iife.js)

IIFEs cont.

- refer to common pitfall in code
- This means that within the scope of each step of the for loop, the i variable is defined anew, giving the click handler closure the value that we expect.

Closures

- A closure is the scope created when a function is declared that allows the function to access and manipulate variables that are external to that function.
- Closures allow a function to access all the variables, as well as other functions, that are declared in the same scope within which the function itself is declared.

Uses of Closures

- Private variables
- A common use of closures is to encapsulate some information as a "private variable" of sorts – in other words, to limit their scope. Object-oriented code written in JavaScript is unable to use traditional private variables: properties of the object that are hidden from outside parties. However, using the concept of a closure, we can achieve an acceptable approximation.
- (source closure.js)

JavaScript scoping

- Scopes in JavaScript act somewhat differently than in most other languages whose syntax is influenced by C; namely, those that use braces ({ and }) as block delimiters. In most of such languages, each block creates its own scope; not so in JavaScript!
- In JavaScript, scopes are declared by functions, and not by blocks.
- (source scope.js)

Object orientation with prototypes

- All Javascript objects have an internal property called constructor.
- The constructor property has a prototype property that points to the prototype object.
- If you try to look up a key on an object and it is not found, JavaScript will look for it in the prototype. It will follow the “prototype chain” until it sees a null value. In that case, it returns undefined.
- (source prototypes.js)

Rules for prototypes

- Properties are bound to the object instance from the prototype.
- This means the context of this defined in a function assigned to the protoype is bound to the object instance.
- In the first example, because we defined the method swingSword() on the object, it will use that definition first.

Inheritance with prototypes

- Since every object has a prototype, you can actually change the pointer of the prototype property to point to something else, hence mimicking inhertiance.
- (source prototypes2.js)

Finally, a note about semicolon insertion

- JavaScript tries to be smart and inserts semicolons for you at the end of statements.
- There are (complex-ish) rules where semi-colons are added for you.
- My personal preference is to always insert semicolons yourself.
- (semicolon-insertion.js)