Web Advanced: Javascript APIs

"We will learn JavaScript properly. Then, we will learn useful design patterns. Then we will pick up useful tools for making cool things better."

FALL 2018

SESSION #8

ERROR HANDLING

OOP APPROACH

jaink@newschool.edu

https://canvas.newschool.edu/courses/1407281

https://classroom.github.com/classrooms/42809645-parsons-web-advanced-javascript-fall-2018

RECAP

ERROR MANAGEMENT

- ➔ Approaches
- ➔ Logging
- ➔ Try...Catch

COMMON ERRORS

- ➔ Variable Definition (var, let)
- ➔ Case sensitivity
- ➔ Strict mode
- ➔ Semicolons and braces/parentheses
- ➔ Using = vs == or ===
- ➔ Non-existent methods or properties
- ➔ ...

DEBUGGING

Using console.log()

Use browser breakpoints and stack traces: debugger

Step through code line by line

Watch expressions

ERROR HANDLING

Using the try...catch statement:

```
try {
    alert("This is code inside the try clause");
    alert("No Errors so catch code will not execute");
} catch (exception) {
    alert("The error is " + exception.message);
}
```

OBJECTS REVISITED

- ➔ Object Types
- ➔ Constructors vs Literals

CREATING OBJECTS

DATE `const the_date = new Date(949278000000);`

ARRAY `const array_one = new Array("one", "two", 3);`

STRING `const string_one = new String("Hello");`

PROPERTIES `string_one.length`

METHODS `string_one.indexOf("l");`

CUSTOM OBJECTS

```
const john_object = new Object();

OR

const john_object = {};
john_object.firstName = "John";
john_object.lastName = "Doe";

john_object.greet = function() {
    alert("My name is " + this.firstName + "
" + this.lastName);
};

john_object.greet();
```

CUSTOM OBJECT LITERALS

```
function createPerson(first_name, last_name) {
    return {
        first_name: first_name,
        last_name: last_name,
        getFullName: function() {
            return this.first_name + " " +
        this.last_name
        },
        greet: function(person) {
            alert("Hello, " + person.getFullName() + ".
        I'm " + this.getFullName());
        }
    };
}

const john_doe = createPerson("John", "Doe");
const jane_doe = createPerson("Jane", "Doe");
john_doe.greet(jane_doe);
```

OOP DEFINITIONS

Objects are used to model real world things that need to be represented inside programs, and/or provide an easy way to access functionality

- ➔ Encapsulation
- ➔ Polymorphism
- ➔ Inheritance

CLASSICAL CONSTRUCT/ES6

The constructor function (equivalent of a class in most classical languages) builds the object and defines its methods and properties.

```
    class Person {
            constructor(name) {
                this.name = name;
            }
            greet() {
               return "Hi, my name is " +
this.getName();
            }
            getName () {
                return this.name;
            }
    }

    const me = new Person('bill');
```

OBJECT CONSTRUCTOR

The constructor function (equivalent of a class in most classical languages) builds the object and defines its methods and properties.

The familiar way (Factory approach):

```
const john_doe = {
                first_name : "John",
                last_name : "Doe",

                greet : function() {
                    alert("My name is " +
                this.first_name + " " +
                this.last_name;
                }
            };
```

OBJECT CONSTRUCTOR

```
function Person(first_name, last_name) {
       this.first_name = first_name;
       this.last_name = last_name;
       getFullName = function() {
          return this.first_name + " " +
       this.last_name ;
       }
       greet = function(person) {
          alert("Hello, " +
       this.getFullName() + ". I'm " +
       this.getFullName());
       }
}

const john_doe = new Person("John", "Doe");
const jane_doe = new Person("Jane", "Doe");
jane_doe.greet();
john_doe.greet();
```

OBJECT CONSTRUCTOR

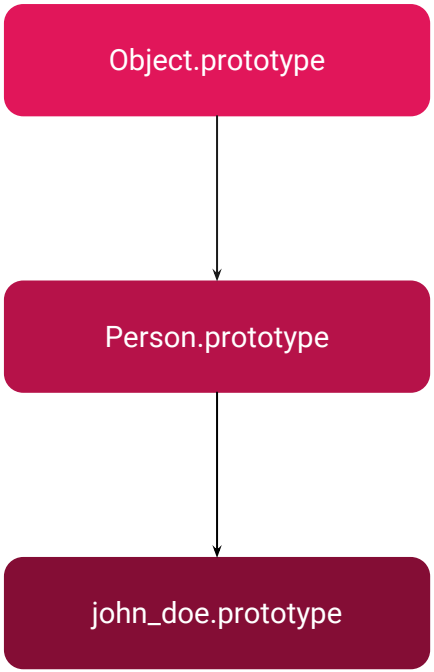
Using the Object approach:

```
const john_doe = new Object();
john_doe.first_name = "John";
…
OR
…
const john_doe = {
          first_name: "John",
          last_name: "Doe",
          getFullName: function() {
              return this.first_name + " " +
          this.last_name ;
          }
       };

john_doe.getFullName();
const jane_doe = Object.create(john_doe);
```

OBJECT PROTOTYPES

- ➔ Prototype object acts as a template object that inherits methods and properties from its creator.
- ➔ An object property of a function containing all its properties and functions.
- ➔ Used to add any new properties and methods after the constructor function has been defined.



PROTOTYPES

```
function Person(first_name, last_name) {
       this.first_name = first_name;
       this.last_name = last_name;
}

Person.prototype.getFullName = function() {
          return this.first_name + " " +
       this.last_name ;
       };
Person.prototype.greet = function(person) {
          alert("Hello, " +
       this.getFullName() + ". I'm " +
       this.getFullName());
       };

const john_doe = new Person("John", "Doe");
const jane_doe = new Person("Jane", "Doe");
```

PROTOTYPES

Get an object's prototype:

```
jane_doe.constructor.prototype;

Object.getPrototypeOf(jane_doe);

Person.prototype.isPrototypeOf(jane_doe)
```

LIVE PROPERTY

The prototype object is live, so if a new property or method is added to the prototype, any instances of it will inherit the new properties and methods automatically, even if that instance has already been created.

```
Person.prototype.sayBye = function(person) {
          alert("Bye, " +
       this.getFullName());
       };

jane_doe.sayBye();
john_doe.sayBye();
```

PROPERTIES AND METHODS

Common pattern to define all properties inside the constructor and methods on its prototype:

```
function Test(a, b, c, d) {
  // property definitions
}

// First method definition
Test.prototype.x = function() { ... };

// Second method definition
Test.prototype.y = function() { ... };

// etc.
```

INHERITANCE

Prototype properties and methods are inherited from the chain - the prototype tree. But not all are inherited. Those defined in the prototype property are inherited. eg. `Object.prototype.sayHi.` **But the ones that are not defined on it are available only to the object scope.**

Create method:

```
Xperson = Object.create(Person);
Xperson.getSong = function() {
  alert ( "woosh " + this.first_name );
};

const storm_doe = Object.create(Xperson);
storm_doe.first_name = "Storm";

storm_doe.getSong();
jane_doe.getSong();
```

INHERITANCE

Constructor method:

```
function Person(first_name, last_name) {
        this.first_name = first_name;
        this.last_name = last_name;
}

Person.prototype.getFullName = function() {
            return this.first_name + " " + this.last_name
        ;
        };
function Xperson(first_name, last_name, power) {
  // Chain constructor with call
  Person.call(this, first_name, last_name);
  this.power = power;
}

Xperson.prototype = Object.create(Person.prototype);
Xperson.prototype.constructor = Xperson;

Xperson.prototype.getSound = function() {
  alert ( "woosh " + this.first_name );
};

const storm_doe = new Xperson("Storm", "Raven", "Wind");
storm_doe.getSound();
storm_doe.getFullName();
```

MODIFYING EXISTING OBJECTS

Add more methods to the prototype of JavaScript's built-in objects—such as Number, String, and Array—to add more functionality (monkey-patching):

```
Number.prototype.isEven = function() {
  return this%2 === 0;
}
Number.prototype.isOdd = function() {
  return this%2 === 1;
}

Array.prototype.first = function() {
   return this[0];
}
Array.prototype.last = function() {
  return this[this.length -1];
}
```

EXAMPLES

Assignment

Next Steps

- ➔ Functional Programming
- ➔ Modules