.: Object Documentation RC2020a :.

♫♫Are you down with OOP?:

Yeah, you know me! ♫♫

Table of Contents

I. What is OOP?

This documentation was developed to explain and explore the concepts and nuances of Object Oriented Javascript (JS) Programming. OOP is a programming paradigm based on the idea of *objects,* that can contain data, in the form of attributes or properties, and code in the form of methods. Object-oriented programming aims to implement real-world concepts like inheritance, abstraction, polymorphism, and encapsulation etc. OOP aims to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

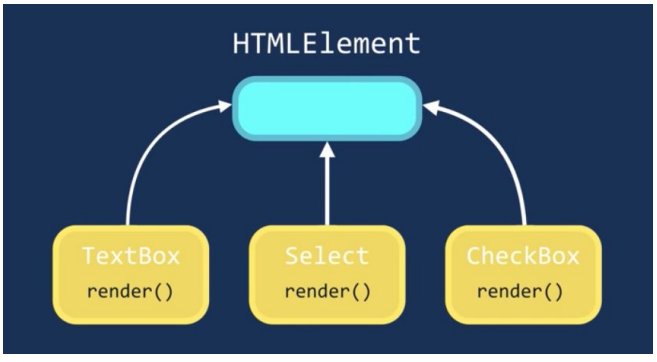
A. Four Pillars of OOP

a. Encapsulation
   i. Encapsulation (in JS) is when you group properties and methods (functions) together under an object.
   ii. It restricts access to certain properties
   iii. The properties and methods made are unique to the object it's under.
   iv. In OOP, we group related variables and functions that operate on them into objects.
   v. Benefits include:
      1. *Reduce Complexity* - We group related variables and functions together and this way we can reduce complexity.
      2. *Increase reusability* - We can reuse these objects in different parts of a program or in different programs.
   vi. Example:

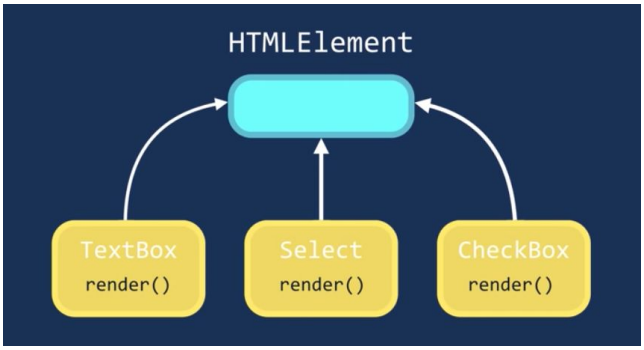
```
class GirlGang{
  constructor (hairType, weaponChoice, intelligenceLevel){
    this.hair = hairType
    this.weapon = weaponChoice
    this.youSmaht = intelligenceLevel
  }
    cutBitches = function (){
      console.log("Slice dat hoe")
    }
    pimpSmack = function (){
      console.log("Smack dat hoe")
    }
    puemPuem = function (){
      console.log("Shoot dat hoe")
    }
}


let beyonce = new GirlGang ("4a", "mic", "89%");
let sam = new GirlGang ("4c", "book", "10000%");
let aiperi = new GirlGang ("3a", "kindness", "10000000%")
```

b. Abstraction

i. We can hide some of the properties and methods from the outside while only showing the essential features
ii. Benefits include:
    1. *Simple Interface* - We'll make the interface of those objects simpler. Understanding an object with fewer properties and methods is easier than on object with several properties and methods.
    2. *Reduce Complexity* - With abstraction, we hide the details and the complexity and show only the essentials. This technique reduces complexity.
    3. *Reduce or isolate the impact of change* - Let's imagine that tomorrow we change inner or private methods of the object. None of these changes will leak to the outside because we don't have any code that touches these methods outside of their content object. We may delete a method or change its parameters but none of these changes will impact the rest of the applications code. So with abstraction, we reduce the impact of change.
iii. Example:
    1. Think of a DVD player as an object. This DVD player has a complex logic board on the inside and a few buttons on the outside that you interact with. You simply press the play button and you don't care what happens on the inside. All that complexity is hidden from you.

c. Inheritance
i. Great code reuses and eliminates redundant code, we call this being dry.
ii. We can create a new class but instead of writing it from scratch, we can base it on an existing class.
iii. In OOP, the Parent (or base class) had properties and methods that can be inherited by the child (or subtype class).
    1. The child class can have additional properties or methods outside of the parent
iv. Reduces complexity isolates the impacts of changes in code
v. Example:
    1. 

d. Polymorphism
   i. Poly means “many” and morph means “form”
   ii. In object-oriented programming, polymorphism is a technique that allows you to get rid of long if-and-else or Switch-case statements.
   iii. Example:



B. Defining an Object Template

JavaScript has a data type called *object.* JavaScript *objects* encapsulate data and functionality in reusable components. Before we start creating objects, it’s important to identify what an object actually is and when/why we use them. An object initializer is an expression that describes the initialization of an Object. Objects consist of *properties*, which are used to describe an object. Values of object properties can either contain primitive data types or other objects. Objects in JavaScript are very similar to arrays, but objects use strings instead of numbers to access the different elements. The strings are called keys or properties, and the elements they point to are called values. Together these pieces of information are called key-value pairs. While arrays are mostly used to represent lists of multiple things, objects are often used to represent single things with multiple characteristics, or attributes.

C. Creating Actual Objects

JavaScript has two ways to create objects:

- By writing an object literal
- By using the object constructor method

Which one you choose depends on the circumstance. We can create **object instances** — objects that contain the data and functionality defined in the class. When an object instance is created from a class, the class's **constructor function** is run to create it. This process of creating an object instance from a class is called **instantiation** — the object instance is **instantiated** from the class. The fancy word for the ability for multiple object types to implement the same functionality is **polymorphism**.

Defining objects with object literals
The object literal method of creating objects starts with a standard variable definition, using the let keyword, followed by the assignment operator:

```
let person =
```

Followed by curly braces with comma-separated name/value pairs:

```
let person = {eyes: 2, feet: 2, hands: 2,  eyeColor:
"blue"};
```

If you don't know the properties that your object will have when you create it or if your program requires that additional properties be added at a later time, you can create the object with few, or even no properties, and then add properties to it later:

```
let person = {};
person.eyes = 2;
person.hair = "brown";
```

document.write and *console.log* both use this method of separating properties with a period, so it may look familiar to you. The dot between the object name and the property indicates that the property belongs to that object.

Another thing to notice about objects is that, like arrays, objects can contain multiple different data types as the values of properties. The secret to really understanding JavaScript is in knowing that arrays and functions are types of objects and that the number, string, and Boolean primitive data types can also be used as objects. What this means is that they have all the properties of objects and can be assigned properties in the same way as objects.

Defining objects with an Object constructor
The second way to define an object is by using an Object constructor. This method declares the object using *new Object* and then give it properties.

An example of an Object constructor is shown here:

```
let person = new Object();
person.feet = 2;
person.name = "Sandy";
person.hair = "black";
```

D. Creating Specialist Classes
In OOP, we can create new classes based on other classes — these new **child classes** can be made to **inherit** the data and code features of their **parent class**, so you can reuse functionality common to all the object types rather than having to duplicate it. Where functionality differs between classes, you can define specialized features directly on them as needed. For example — teachers and students may share

many common features such as name, gender, and age, so it is convenient to only have to define those features once. You can also define the same feature separately in different classes, as each definition of that feature will be in a different namespace.

II. Constructors & Object Instances

1. A Construction Function is a way how JS defines and initializes objects and its respective features.
2. Constructors are a *generalized, efficient* way to create as many objects as you would need.
3. Think of it as a factory for cars. In that factory you won't spend time building a machine for the parts of one car, then build machines for the parts of another individual car, wouldn't you?
    a. Of course not! So, we assemble machines (or in our case *constructors*) that allow us make new cars with the same properties internally set up.
4. Construction Functions attach data and additional functions to the newly created objects as you go.
5. The construction function is JS's way of making a class.
6. In the example, name is the parameter, and *this* is used to refer to the main object for every new property you make as you go down (name … honk … etc)
7. Note: the constructor function name always starts with a capital letter!

Example:

```
function Car(name){
     this.name = name;
     this.honk = function(){
          Alert ('Beep! , said the ' + this.name +
          'car');
     };
}

Let car1 = new Car(bentley)
Let car2 = new Car(honda)
```

III. Objects Basics

There are more than one ways of creating an object than object instances and using a constructor function but we can also use the **object()** constructor and the **create()** method.

A. The Object() Constructor

First of all, you can use the `Object()` constructor to create a new object. Yes, even generic objects have a constructor, which generates an empty object.

- Try entering this into your browser's JavaScript console:

  ```
  let person1 = new Object();
  ```

- This stores an empty object in the `person1` variable. You can then add properties and methods to this object using dot or bracket notation as desired; try these examples in your console:

  ```
  person1.name = 'Chris'; person1['age'] = 38; person1.greeting =
  function() { alert('Hi! I\'m ' + this.name + '.'); };
  ```

- You can also pass an object literal to the `Object()` constructor as a parameter, to prefill it with properties/methods. Try this in your JS console:

  ```
  let person1 = new Object({ name: 'Chris', age: 38, greeting:
  function() { alert('Hi! I\'m ' + this.name + '.'); } });
  ```

B. The Object() Constructor

Constructors can help you give your code order—you can create constructors in one place, then create instances as needed, and it is clear where they came from. However, some people prefer to create object instances without first creating constructors, especially if they are creating only a few instances of an object. JavaScript has a built-in method called `create()` that allows you to do that. With it, you can create a new object based on any existing object.

1. With your finished exercise from the previous sections loaded in the browser, try this in your JavaScript console:

   ```
   let person2 = Object.create(person1);
   ```

2. Now try these:

```
person2.name; person2.greeting();
```

One limitation of `create()` is that IE8 does not support it. So constructors may be more effective if you want to support older browsers.

IV. Objects Basics

An object is a collection of related data and/or functionality (which usually consists of several variables and functions — which are called properties and methods when they are inside objects.) As with many things in JavaScript, creating an object often begins with defining and initializing a variable.

An object is made up of multiple members, each of which has a name (e.g. name and age above), and a value (e.g. `['Ken', 'Alvarez'] and 22`). Each name/value pair must be separated by a comma, and the name and value in each case are separated by a colon. The value of an object member can be pretty much anything — in our person object we've got a string, a number, two arrays, and two functions. The first four items are data items, and are referred to as the object's **properties**. The last two items are functions that allow the object to do something with that data, and are referred to as the object's **methods**.

An object like this is referred to as an **object literal**.. This is in contrast to objects instantiated from classes.

It is very common to create an object using an object literal when you want to transfer a series of structured, related data items in some manner, for example sending a request to the server to be put into a database. Sending a single object is much more efficient than sending several items individually, and it is easier to work with than an array, when you want to identify individual items by name.

V. Dot Notation

What is Dot Notation?

When you want to access a value in the object you can do this using *Dot Notation* (.) Dot Notation is what you use when you know the name of the property you're trying to access ahead of time. It is the most common way for you to access properties in JavaScript with a string identifier. In Dot Notation, you do not need to use quotation marks. You also want to limit spaces. It is used most frequently.

Example:

```
let resilient = {
coders: 'Samantha',
};
let student = resilient.coders;
console.log(student)
// Samantha
```

VI. Bracket Notation

Bracket notation can also be used to create properties. Functionally, bracket notation is the same as dot notation. However, the syntax looks entirely different. Both notations are interchangeable. The syntax for using Bracket Notation is:

```
set = object[property_name]
Object[property_name] = set
```

Now, you may be asking yourself “why are there two different notations for the same procedure?” It has to do with how JS “unboxes” statements. When dot notation is used JS goes till the first dot and then starts to unbox the property after the dot. This difference becomes apparent when you try to set a variable equal to one of the properties!

Example:
The object

```
Var cat = {
     property1: “meow”,
     property2: “hiss”,
     }
```

Attempting to Access A property of the Object

```
cat.property1 “trying to get a meow”
```

It’s a similar process with Bracket Notation

```
cat[“property1”]
```

```
"meow"
```

Example:

```
var x = “ property2”
cat.x
Would become undefined !
```

BUT

```
cat[x]
Would give you “hiss”
```

VII. Setting Object Members

- Setting (updating) the value of an object member is done via dot notation or via bracket notation.
- You can set an entirely new object this way too.
    - Say you set up an object called resilient, and resilient has a bunch of properties: coders, tables, room. Outside of the object literal where you set up the properties, you can “declare” a new property value pair like below:

Example: [dot notation]

```
resilient.coders = "Kenneth"
resilient.location = "oneBroadway"
```

[bracket notation]

```
resilient['coders'] = "Roger"
resilient['location'] = "255Main"
```

The object **resilient** now updated what was the value attached to the property “coders” to **Kenneth**.

We created a new property (outside of the object literal we had before) called location and we set the value of it to **oneBroadway**.

VIII. What is "`this`"?

(When referring to an object you could write out the name of the function and target it that way. Also you could use the .this keyword to target the object. The latter is much simpler and also guarantees that the correct object is being.

Ex:

```
function MathStuff(num1, num2){
 this.num1 = num1

 this.num2 = num2

 this.connect = function(){

 nextFunc(this.num1,this.num2)

  }

 }
```

In the scenario above, using "`this.`"Allows the properties to access the parameters of the function without having to write out the entire name of the function.
)

`This.` is very useful — it always ensures that the correct values are used when a member's context changes (for example, two different person object instances may have different names, but we want to use their own name when saying their greeting). This is equal to the object the code is inside — this isn't hugely useful when you are writing out object literals by hand, but it really comes into its own when you are dynamically generating objects (for example using constructors).
Here are some key takeaways about this:

- Functions that are stored in object properties are called "methods".
- Methods allow objects to "act" like `object.doSomething()`.
- Methods can reference the object as this.
- The value of this is defined at run-time.
- When a function is declared, it may use this, but that this has no value until the function is called.
- A function can be copied between objects.
- When a function is called in the "method" syntax: `object.method()`, the value of this during the call is object.
- Please note that arrow functions are special: they have no this. When this is accessed inside an arrow function, it is taken from outside.

IX. Prototypes

Almost every function in JavaScript basically has a prototype that references an object. This is used as a fallback source for properties. When an object gets a request for a property that it does not have, its prototype will be searched for the property, then the prototype's prototype, and so on. This means that the property will be looking within it's ancestry for the prototype.

1. A prototype is an object from which other objects inherit properties

Example:
```
Human.prototype.eat = function (amount) {
 console.log(`${this.name} is eating.`)
 this.energy += amount
 }

Human.prototype.sleep = function (length) {
console.log(`${this.name} is sleeping.`)
this.energy += length
}

Human.prototype.play = function (length) {
console.log(`${this.name} is playing.`)
this.energy -= length
}
```

The OG Prototype in the ancestry (of objects) is Object.prototype but not every object corresponds with this because they have their own prototype. As such, functions use Function.prototype and arrays use Arrays.prototype.

In order to understand prototypes, understanding that classes are an object-oriented concept might be helpful. A **`class`** defines the shape of a type of object—what methods and properties it has. If all instances of a class share the same value, (like a method) prototypes can be considered useful.

X. What Are Types?

- There are two kinds of types (Primitive and Reference)
- Primitive types are stored as simple data types.
- Reference types are stored as objects, which are really just references to locations in memory

Primitive Types:
Primitive types are simple pieces of stored data, there are five primitive types

1. Boolean: True or False

2. Number: Any integer floating-point numeric value

3. String: A character or sequence of characters delimited by either single or double quotes
4.
5. Null: A primitive type that has only one value, null
6. Undefined: A primitive type that has only one value, undefined (undefined is the value assigned to a variable that is not initialized.

```
var a = 13          // assign `13` to `a`
var b = a           // copy the value of `a` to `b`
b = 37              // assign `37` to `b`
console.log(a)      // => 13
```

Reference Types:

Represent objects in Javascript and are the closest things to classes that you will find in this language.

An object is an unordered list of a properties consisting of a name (always a string) and a value.

When the value of the property is a function, it is called a function it is called a function.

A Reference type can contain other values, Since the contents of a reference type can not fit in the fixed amount of memory available for a variable, the in-memory value of a reference type is the reference itself (a memory address).

```
var a = { c: 13 }  // assign the reference of a new object
to `a`
var b = a          // copy the reference of the object
inside `a` to new variable `b`
b.c = 37           // modify the contents of the object `b`
refers to
console.log(a)     // => { c: 37 }
```

- Array
- Object
- Function

XI. Built-In Reference Types

- Array - An ordered list of numerically indexed values
- Date - A date and time
- Error - A runtime error (there are also several more specific error subtypes)
- Function - A function
- Object - A generic object
- RegExp - A regular expression