October 19, 2021

**Object Orientation**

**Objects**

**Objects as Key/Value Pairs**

* **Objects** are a common way to show related key-value pairs
* Allow you to associate a group of labelled data into a single structure
* It’s like a dictionary with values of the word

**Object Syntax**

* Enclose with { }
* Keys (properties) can be strings or not

**const** dog = {

'name': 'Freida',

'color': 'brown/black', <<< strings

'hunger': 40,

'mood': 'feisty',

'age': 9,

}; <<<<if a word is being used in javascript term, you can’t use that word as a term

**const** dog = {

name: 'Freida',

color: 'brown/black', <<<< not strings

hunger: 40,

mood: 'feisty',

age: 9,

}; <<<<values can be any data types: string, object, arrays

**Access values in an object**

Can use **dot notation** or **square brackets**

**const** dog = {

name: 'Freida',

color: 'brown/black',

hunger: 40,

mood: 'feisty',

age: 9,

};

console.log(dog.name) *// prints Freida*

console.log(dog['name']) *// also prints Freida*

*for dot notation you need to know exact key in order it to work, object name and then the key (if something isn’t a key, you won’t be able to access it.)*

*bracket you can use the string or a variable as long as it evaluates to a string*

**Saving values to a variable**

Sometimes it’s useful to save a value to a variable so that you can use it outside of the object easily

**const** dog = {

name: 'Freida',

color: 'brown/black',

hunger: 40,

mood: 'feisty',

age: 9

}

**let** freidaAge = dog.age

**Accessing values using destructuring -any of the keys you want to destructer you put in brackets**

We can also access values using a syntax called destructuring

**const** dog = {

name: 'Freida',

color: 'brown/black',

hunger: 40,

mood: 'feisty',

age: 9

}

**let** {age} = dog

On the left side of the equals sign, you destructure the properties from the object, which goes on the right

We can also desturture multiple properties at a time

**let** {name, color, age} = dog

console.log(`*${*name*}* is a *${*color*}* dog who is *${*age*}* years old.`)

You can also do (name. dog.)

If you need to, you can also rename destructured properties in line

**const** dog = {

name: 'Freida',

color: 'brown/black',

age: 9

}

**const** dog2 = {

name: 'Buddy',

color: 'gray',

age: 1

}

**const** {age: freidaAge} = dog<<<< new way to assign variables

**const** {age: buddyAge} = dog2 <<<

console.log(freidaAge) *// 9*

console.log(buddyAge) *// 1*

console.log(age) *// undefined*

**Loop over properties in an object**

*// use for ... in to loop over object keys*

**for** (**let** attribute **in** dog) { <<<<a for in loop

console.log(`The dog's *${*attribute*}* is *${*dog[attribute]*}*.`);

}

You can even write “ let “key” in object

**Add new key/value pair to existing object**

Assign using **dot notation** or **square brackets**

**const** dog = {

name: 'Freida',

color: 'brown/black',

hunger: 40,

mood: 'feisty',

age: 9,

};

object Key

dog.nickname = 'Puppers'; *// dot notation*

dog['nickname'] = 'Puppers'; *// square brackets*

**Map**

Maps are a semi-recent addition to the JavaScript standard library.

In JavaScript, Maps are *ordered* collections of key-value pairs. They’re used for key-value pairs that need to maintain order, like transaction histories.

**Removing a value from an object**

We can remove values from object using the delete keyword

**let** person = {

name: 'Jake Peralta',

occupation: 'detective',

phrase: 'coolcoolcoolcoolcool',

location: 'Brooklyn'

}

**delete** person.phrase << deletes the key and values

**Classes**

**Benefits of objects**

* Flexible - all ***dog*** objects don’t have to have the same properties
* Can write functions, loops to do useful things with objects

**let** dog1 = {

'name': 'Freida',

'age': 9,

'color': 'brown'

};

**let** dog2 = {

'name': 'Libby',

'age': 3

'nickname': 'Libbers'

};

**function** bark(dog){ <<<<function that

console.log('Arf! I am', dog.name, 'the dog!');

}

**Issues with making your own objects**

* It’s helpful for all dog objects to have the **exact same** properties
  + Prevent bugs in your code
* It would be nice if you could store data about dogs in the same place as dog **behaviors**
  + Things that dog objects know how to do, like ***bark***

**Enter… classes!**

* A **class** is like a little mini factory that knows how to make objects of a single type
* Let’s you define the **blueprint** for future objects
  + For example, all dogs have properties color, name, hunger, and mood
* Also lets you define object behaviors, or **methods**
* **Can assign it to all and use it in a function**

**Example**

**(if something is a different color, it’s a primary that can not be change.)**

**class** Dog {<<<<use this format to create class of something

**constructor**(name, color){ <<<(2 attributes)

**this**.name = name;

**this**.color = color;

}

bark(){

console.log('Arf! I am', **this**.name, 'the dog!');<<<heres what the method does

}

} <<<all of this belongs to the class just created

* ***Dog*** class defines the template or blueprint for **all** dogs
* ***constructor*** allows you to assign data specific to each individual dog
* ***bark*** is the behavior, or method, that each dog knows
  + this.name means “get the name of whatever dog is trying to bark right now”

**Making Objects with Classes**

**class** Dog {

*// ...*

}

**let** dog1 = **new** Dog('Freida', 'brown'); <<make sure the order is right when using class!!!

**let** dog2 = **new** Dog('Sally', 'pink');

console.log(dog1.name) *// Freida*

console.log(dog2.name) *// Sally*

dog1.bark() *// Arf, I'm Freida the dog!<<<<this is a method, not key. Everytime I make the method every dog can use this now.*

dog2.bark() *// Arf, I'm Sally the dog!*

* ***dog1*** and ***dog2*** are **instances** of the class ***Dog***
* ***dog1*** and ***dog*** are also **objects** of the type ***Dog***
* When you make a dog using the ***Dog*** class, called **instantiation**

**The End**

© 2021 Devmountain