**Advanced Objects**

Recall: Objects and JS are containers that store data and functionality.

1. **The this keyword**

Methods and Properties of an object **DO NOT AUTOMATICALLY** have access to different methods and properties of the same object.

Therefore, we have to use the **this** keyword, which references the calling object (left image).

Text

Description automatically generatedText

Description automatically generated

1. **Arrow Functions and this**

When using arrow functions, using this is a bit more complicated (right image)

The reason is that arrow functions naturally bind (or tie) an already defined **this** value in the global scope to the function, which is NOT the calling object. The **this** object in the global scope does not have the property dietType, so it will return undefined.

**🡪 Key take away: avoid using arrow functions for defining methods.**

1. **Privacy**

Sometimes we do not want the value of properties to be manipulated. There is a naming convention in JavaScript that signals programmers they cannot modify the value of that property:

**\_<propertyName>**

This is just a convention, and we can still modify the value of this property regardless. To respect this convention, we’ll need to use methods called getters and setters.

1. **Getters**

- Getters are methods that access the internal properties of an object. It grants access to other properties of the object using **this**, but we do not to directly access these properties.

- When calling getters, we can omit the (). **This is like we are accessing a property.**

const person = {  
  \_firstName: 'John',  
  \_lastName: 'Doe',  
  get fullName() {  
     if (this.\_firstName && this.\_lastName){  
       return `${this.\_firstName} ${this.\_lastName}`;  
   }

else {  
       return 'Missing a first name or a last name.';  
     }  
  }  
}  
// To call the getter method:   
person.fullName;; // 'John Doe'

1. **Setters**

- Setter methods are used to signal that we are reassigning values of existing properties

- Calling setters also do not need the (). **This is like we are reassigning the value of a property.**

const person = {{  
  \_age: 37,  
  set age(newAge){  
     if (typeof newAge === 'number'){  
       this.\_age = newAge;  
     }

else {  
       console.log('You must assign a number to age');  
     }  
  }  
};

// To call the setter method  
person.age = 40; // Logs: 40

|  |
| --- |
| **In summary, both getters and setters methods are only conventions that programmers should follow, to denote the action of accessing (get) or changing (set) properties. We can still do this normally by usign this. in other functions, but it is less intuitive and can prompt logical error.**  The convention is: for a property **\_<propertyA>** we name as **get <propertyA>** and **set <propertyA>** |

1. **Objects and Function Methods ♣♣**

There are some notable Function Methods, that relates to the use of the **this** keyword.

1. .call()

Allowing the use of an object A method on object B. Function arguments are provided individually.

objectA = { fname:”Minh”, lname:”Nguyen”,

printName(num){ console.log(fname,lname,”age = ${num}”); }

objectB = { fname:”Linh”, lname:”Pham”}

We can use the printName() function with objectB by:

objectA.printName(21); // Minh Nguyen age = 21

objectA.printName.call(objectB, 20); // Linh Pham age = 20

However, the method is usually defined outside of the object:

function printName(num){ console.log(**this**.fname, **this**.lname, ”age = ${num}”); }

printName.call(objectA, 21); // Minh Nguyen age = 21

printName.call(objectB, 20); // Linh Pham age = 20

🡪 As we use the .call() method on the function, we are passing the first argument as what we want the **this** keyword to point to. Note that since functions are also object, we can even pass a function, and **this** will refer to the function.

The general prototype of the .call() method is

**<function>.call( <this\_value>, para1, para2,… )**

1. .apply()

Works exactly similar to .call(). However, function arguments are provided as an array. This has an advantage, as the number of parameters do not need to be specified while defining the function.

In the function, there is a special local variable called **argument**. This is an array-like variable that contains all arguments passed in the array. Therefore, we can take advantage of this special variable to work with the values passed.

function calsum(){ sum = 0;

for (let i = 0; i < **arguments**.length; i++) { sum += arguments[i]; }

return sum; }

calsum.call(null, [1, 2, 3, 4]); // 10

calsum.call(null, [1, 2, 3, 4, 5]); // 15

1. .bind()

The .bind() method works exactly the same way of the .call() method. The only difference is that the .bind() will assign the function being used to a new variable. Then, we can call this new variable in order to “execute” the function. While .call() will directly execute the function, .bind() will save it to the new variable, which can later be used when we need.

function printName(num){

console.log(**this**.fname, **this**.lname, ”age = ${num}”);

}

let temp = printName.bind(objectA, 21);

temp(); // Minh Nguyen age = 21

1. **Built-in Object Methods**

We have been creating methods for objects by ourselves. However, Objects also have their built-in methods.

Object instance methods

+) .hasOwnProperty()

+) .valueOf()

Object class methods

+) Object.keys(<objectclass>) 🡪 Returns an array of a given object's **property names**

+) Object.entries(<objectclass>) 🡪 Returns an **array of arrays** of a given object's **property names & values**

+) Object.assign(<target>,<source>)

🡪 copies all properties from one or more *source objects* to a *target object*. It **returns the modified target object**.