Object Oriented Programming in JavaScript 17th Day

JavaScript is not a class-based object-oriented language instead, JavaScript is a prototype-based language. But it still has ways of using object oriented programming (OOP).

According to Mozilla's documentation:
"A prototype-based language has the notion of a prototypical object, an object used as a template from which to get the initial properties for a new object."

There are certain features or mechanisms which makes a Language Object Oriented like:

- Object
- Classes
- Encapsulation
- Inheritance

1). **Object**.
An Object is a **unique** entity which contains **property** and **methods**. For example "car" is a real life Object, which have some characteristics like color, type, model, horsepower and performs certain action like drive. The characteristics of an Object are called as Property, in Object Oriented Programming and the actions are called methods. An Object is an **instance** of a class. Objects are everywhere in JavaScript almost every element is an Object whether it is a function,arrays and string.

Note That: A Method in javascript is a property of an object whose value is a function.

Object can be created in different ways in JavaScript:

a). Using an **Object Literal.**

```
* Defining object  *
let person = {
    first_name:'Liam',
    last_name: 'Summaya',

    /* method  */
    getFunction : function(){
        return (`The name of the person is
          ${person.first_name} ${person.last_name}`)
    },
    /* object within object */
    phone_number : {
        mobile:'12345',
        landline:'6789'
    }
}
console.log(person.getFunction());
console.log(person.phone_number.landline);
```

```
The output will be:

The name of the person is Liam Summaya
6789
```

b). Using an **Object Constructor.**

```
/* using a constructor */
function person(first_name,last_name){
   this.first_name = first_name;
   this.last_name = last_name;
}
/* creating new instances of person object  */
let person1 = new person('Liam','Doe');
let person2 = new person('John','Summaya');

console.log(person1.first_name);
console.log(`${person2.first_name} ${person2.last_name}`);
```

Run the Move code and check the output.

c). Using **Object.create() method**:

The Object.create() method creates a new object, using an existing object as the prototype of the newly created object.

```
/* Object.create() example a simple object with some properties */

const coder = {
    isStudying : false,
    printIntroduction : function(){
        console.log(`My name is ${this.name}. Am I
          studying?: ${this.isStudying}.`)
    }
}
/*  Object.create() method */
const me = Object.create(coder);
/* "name" is a property set on "me", but not on "coder" */
me.name = 'Summaya';
/* Inherited properties can be overwritten */
me.isStudying = 'True';
me.printIntroduction();
```

Output will be:

```
my name is Summaya. Am I studying?: True
```

2). **Classes**.

Classes are blueprint of an Object. A class can have many Object, because class is a template while Object are instances of the class or the concrete implementation.

Before we move further into implementation, we should know unlike other Object Oriented Language there is no classes in JavaScript we have only Object. To be more precise, JavaScript is a prototype based object oriented language, which means it doesn't have classes rather it define behaviors using constructor function and then reuse it using the prototype.

“JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript’s existing prototype-based inheritance. The class syntax is not introducing a new object-oriented inheritance model to JavaScript. JavaScript classes provide a much simpler and clearer syntax to create objects and deal with inheritance.”
–Mozilla Developer Network

Modern way of creating classes:

```
/* Defining class using es6  */
class Vehicle {
  constructor(name, maker, engine) {
    this.name = name;
    this.maker =  maker;
    this.engine = engine;
  }
  getDetails(){
      return (`The name of the bike is ${this.name}.`)
  }
}
/* Making object with the help of the constructor */
let bike1 = new Vehicle('Hayabusa', 'Suzuki', '1340cc');
let bike2 = new Vehicle('Ninja', 'Kawasaki', '998cc');

console.log(bike1.name);     // Hayabusa
console.log(bike2.maker);    // Kawasaki
console.log(bike1.getDetails());
```

The output will be:

The Traditional way of creating classes :

```
/* Defining class in a Traditional Way. */
function Vehicle(name,maker,engine){
    this.name = name,
    this.maker = maker,
    this.engine = engine
};

Vehicle.prototype.getDetails = function(){
    console.log('The name of the bike is ' + this.name);
}

let bike1 = new Vehicle('Hayabusa','Suzuki','1340cc');
let bike2 = new Vehicle('Ninja','Kawasaki','998cc');

console.log(bike1.name);
console.log(bike2.maker);
console.log(bike1.getDetails());
```

The Output will be:

As you may have observed in the two example above it is much simpler to define and reuse object in ES6. Hence, we would be using ES6 implementing encapsulation and inheritance.

3). **Encapsulation**.

– The process of wrapping property and function within a single unit is known as encapsulation.

Example:

```
//encapsulation example

class person{

    constructor(name,id){
        this.name = name;
        this.id = id;
    }
    add_Address(add){
        this.add = add;
    }
    getDetails(){
        console.log(`Name is ${this.name},Address is: ${this.add}`);
    }
}

let person1 = new person('Liam',21);
person1.add_Address('Lagos');
person1.getDetails();
```

Output will be:

Name is Liam, Adress is: Lagos

In the above example we simply create an *person* Object using the constructor and Initialize it property and use it functions we are not bother about the implementation details. We are working with an Objects interface without considering the implementation details.

Sometimes encapsulation refers to hiding of data or data Abstraction which means representing essential features hiding the background detail. Most of the OOP languages provide access modifiers to restrict the scope of a variable, but their are no such access modifiers in JavaScript but their are certain way by which we can restrict the scope of variable within the Class/Object.

```
// Abstraction example
function person(fname,lname){
    let firstname = fname;
    let lastname = lname;

    let getDetails_noaccess = function(){
        return (`First name is: ${firstname} Last
            name is: ${lastname}`);
    }

    this.getDetails_access = function(){
        return (`First name is: ${firstname}, Last
            name is: ${lastname}`);
    }
}
let person1 = new person('Summaya','Liam');
console.log(person1.firstname);
console.log(person1.getDetails_noaccess);
console.log(person1.getDetails_access());
```

Output will be:

First name is: Summaya, Last name is: Liam

In the above example we try to access some property(person1.firstname) and functions(person1.getDetails_noaccess) but it returns u*ndefined* while their is a method which we can access from the *person* object(person1.getDetails_access()), by changing the way to define a function we can restrict its scope.

4). **Inheritance**.

It is a concept in which some property and methods of an Object is being used by another Object. Unlike most of the OOP languages where classes inherit classes, JavaScript Object inherits Object i.e. certain features (property and methods)of one object can be reused by other Objects.

```
/* Inhertiance example  */
class person{
    constructor(name){
        this.name = name;
    }
    /* method to return the string */
    toString(){
        return (`Name of person: ${this.name}`);
    }
}
class student extends person{
    constructor(name,id){
        /* super keyword to for calling above class constructor */
        super(name);
        this.id = id;
    }
    toString(){
        return (`${super.toString()},Student ID: ${this.id}`);
    }
}
let student1 = new student('Summaya',2120);
console.log(student1.toString());
```

Output:

Name of person: Summaya, Student ID: 2120

In the above example we define an *Person* Object with certain property and method and then we *inherit* the *Person* Object in the *Student* Object and use all the property and method of person Object as well define certain property and methods for *Student*.

The Person and Student object both have same method i.e toString(), this is called as **Method Overriding**. Method Overriding allows method in a child class to have the same name and method signature as that of a parent class.

In the above code, super keyword is used to refer immediate parent class instance variable.

“JavaScript is the only language that I’m aware of that people feel they don’t need to learn before they start using it.” Douglas Crockford

Best Wishes Lux Tech Academy.