OOPs – Object Oriented Programming Using JavaScript (ES6 - standard)

- In **Procedural, a program is divided into set of functions and data is stored in bunch of variables and functions operate on these data**. As program grows we will create more functions and if changes made to function several other functions will break. So much of interdependency.
- In OOPs, it **combines group of relative variables and functions into a unit** called as an **Object**. **Variables are referred as Properties and functions are referred as Methods**.

- ***Properties of OOPs***: *Abstraction, Encapsulation, Inheritance, Polymorphism*.
- **Encapsulation**: Group of relative variables and functions that operate on them into object.
- Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates.
  Ex: make variables of class as private, so no other class of functions use this.
- Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.
  Ex: **In procedural**,

```
let baseSalary = 30_000, overtime = 10, rate = 20;
function getWage(baseSalary, overtime, rate) {
   return baseSalary + (overtime * rate);
}
```

  In OOPs,

```
let employee = {
   baseSalary: 30_000,
   overtime: 10,
   rate: 20,
   getWage: function () { return this.baseSalary + (this.overtime *
                                this.rate); }
};
employee.getWage();
```

- **Abstraction**: Data Abstraction is a property in which essential details are shown to user. Hide the details and complexity, show only essentials which reduce complexity and isolate impact of changes. In java, abstraction is achieved by interfaces and abstract classes.
- **Inheritance**: It is a mechanism that allow to *eliminate redundant code*.
- It is the mechanism by which one class is allow to inherit the features (fields and methods) of another class. Important terminology: Super, Sub class.
- *Reusability*: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.
- **Polymorphism**: *Poly* means many and *morphi* means form. Polymorphism refers to the ability of OOPs programming languages to differentiate between entities with the same name efficiently.
- This is done by Java with the help of the signature and declaration of these entities.

JavaScript Concepts(IMP)

In JavaScript, semicolon(;) is optional.

- **Object literals:**

```
const circle = {} //declaring object literal.
//Internally JavaScript-Engine represents above line this way: const circle = new Object();
```

  Ex:

```
const circle = {
   radius: 1,
   location: {x:1, y=1},
   draw: function (){ console.log("draw"); }
```

```
};
```

- Here **radius, location** and **draw** are called as **members**.
- The member **radius and location are properties** which holds values and **draw is a method** which defines some logic.
- Properties are accessed by "." Operator. Ex: circle.radius = 2

Ways to create object:- Factory function and Constructor.

- **Factory:** It avoid duplicating of data if object as has atleast one function.

Ex:

```
const circle = {
   radius: 1,
   location: {x:1, y=1},
   draw: function() { console.log("draw"); }
};

const circle = {
   radius: 2,
   location: {x:2, y=2},
   draw: function() { console.log("draw"); }
};
circle.draw()
```

Above code can be written as,

```
//this function is called as factory function.
function createCircle(radius) {
   return {
      radius, //In ES6, if both key and value has same names i.e. radius: radius, use
              //only key.
      draw: function() { console.log("draw"); }
   }
}
const circle = createCircle(1)
circle.draw()
```

For more: Factory Functions

- **Constructor:** Function name should Capitalized.

Ex:

```
//this keyword outside function points to global object. In console, window is global
//object.
function Circle(radius) {
   //here this keyword points to current object iff the function is called with new
   //keyword.
   this.radius = radius;
   this.draw = function() { console.log("draw"); }
}
const another = new Circle(1); //new operator creates an empty object.
const other = Circle.call({}, 1); //this is exactly done by new keyword.
const otherAnother = Circle.apply({}, [1]);
//new keyword internally calls call() with parameter {} and radius parameter.
```

- **new** operator functions:-
  - It creates an empty object.
  - It will set this to current object.
  - It returns object from that function and no need to mention the return statement in function.
- **new** keyword internally calls inbuilt function Circle.call({}, 1}. Here {} refers to this and 1 refers to parameter. So we can create an object using Circle.call({}, 1).

- **Value vs Reference Types:-**
  - Value Types:- Number, String, Boolean, Symbol, undefined, null.
  - Reference Types:- Object, Function, Array.

  Primitives are copied by their value whereas Objects are copied by their reference.

- To enumerate all members of an object: `for(let key in circle){ //key, circle[key] }`
  Best to use *[] operator* instead *"." operator*, because if *key=first-key*, we can't use *circle.first-key* to get the value., So use *circle[first-key]*.
- To get all the key of an object use Object.keys(), `const keys = Object.keys(circle);`
- To check for an existence of a property or method use 'in', `if('radius' in circle){} //in operator`

For more: The Difference Between Values and References in JavaScript

Closure: A closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.
Ex:

```
function init() {
  let name = 'Mozilla'; // name is a local variable created by init
  function displayName() { // displayName() is the inner function, a closure
    alert(name); // use variable declared in the parent function
  }
  displayName();
}
init();
```

For more: Closures

Getter and Setter:
Ex:

```
function Circle(){
      let defaultLocation = { x:0, y:0 };
      Object.defineProperty(this, 'defaultLocation', {
            get: function() { return defaultLocation; }
            set: function(value){
                  if(!value.x || !value.y) throw new Error('Invalid location.');
                  defaultLocation = value;
}
```

Instead of using a function for get and set, we can implement above code.

Exercise:

```
function Stopwatch() {
      let running = false, time = 0, duration = 0;
      this.reset = () => ((duration = 0), (running = false));
      this.start = () => {
            if (running) throw new Error("Stopwatch has already started.");
            time = new Date().getTime();
            running = true;
      };
      this.stop = () => {
            if (!running) throw new Error("Stopwatch is not started.");
            duration += (new Date().getTime() - time) / 1000;
            running = false;
      };

      //below one used to access duration variable which has function scope, outside the
      //the function. So it makes duration access outside fun and allow only to read.
      Object.defineProperty(this, "duration", { get: () => duration });
}
```