OBJECT & OBJECT CONSTRUCTOR

A. Object As a Design Pattern

One of the simplest ways you can begin to organize your code is by grouping things into objects. Take these examples from a 'tic tac toe' game:

```javascript
1   // example one
2   const playerOneName = "tim";
3   const playerTwoName = "jenn";
4   const playerOneMarker = "X";
5   const playerTwoMarker = "O";
6
7   // example two
8   const playerOne = {
9     name: "tim",
10    marker: "X"
11  };
12
13  const playerTwo = {
14    name: "jenn",
15    marker: "O"
```

The benefit of the example two is huge:

1) you dont need to remember var name (like example one)

```javascript
function printName(player) {
    console.log(player.name);
```

example one:

```javascript
console.log(playerOneName);
console.log(playerTwoName);
```

2) you don't need to know which player's name you want to print

```javascript
function gameOver(winningPlayer){
    console.log("Congratulations!");
    console.log(winningPlayer.name + " is the winner!");
```

3) In more complicated (such as an online shopping site with a large inventory) using objects (to keep track of an item's name, price, description and other things) is the only way to go.

Unfortunately, in that (complicated) type of situation, manually typing out the contents of our objects is not feasible either. We need a cleaner way to create our objects, which brings us to OBJECT CONSTRUCTOR.

B. Object Constructor

When you have a specific type of object that you need to duplicate like our player or inventory items, a better way to create them is using an object constructor, which is a function that looks like this:

```javascript
function Player(name, marker) {
    this.name = name;
    this.marker = marker;
```

and which you use by calling the function with the keyword ' new '

```javascript
const player = new Player('steve', 'X');
console.log(player.name); // 'steve'
```

Just like with objects created using the Object Literal method, you can add functions to the object:

```javascript
function Player(name, marker) {
  this.name = name;
  this.marker = marker;
  this.sayName = function() {
    console.log(this.name)
  };
}
```

```
const player1 = new Player('steve', 'X');
const player2 = new Player('also steve', 'O');
player1.sayName(); // logs 'steve'
player2.sayName(); // logs 'also steve'
```

C. The Prototype

All objects in JavaScript have a 'prototype'. The prototype is another object that the original object inherits from, which is to say, the original object has access to all of its prototype's methods and properties.

Semua objeck memiliki prototype. Prototype adalah objek lain yang diwarisi oleh objek asli, artinya objek asli memiliki akses ke semua metode dan properti prototype nya.

Misal: player1 atau player2 adalah objek aslinya, maka player1 atau player 2 bisa mengakses property/function (object)prototypenya. Sebagai contoh jika fungsi .sayHello() ada di (object)prototype nya maka player1 atau player2 bisa mengakses .sayHello().

➢ Accessing an object's prototype

```
1  Object.getPrototypeOf(player1) === Player.prototype; // returns true
2  Object.getPrototypeOf(player2) === Player.prototype; // returns true
```

- You can check the object's `prototype` by using the `Object.getPrototypeOf()` function on the object, like `Object.getPrototypeOf(player1)`.

- The return value (result) of this function refers to the `.prototype` property of the Object Constructor (i.e., `Player(name, marker)`) - `Object.getPrototypeOf(player1) === Player.prototype`.

➢ Benefit of 'prototype'

```
1  Player.prototype.sayHello = function() {
2      console.log("Hello, I'm a player!");
3  };
4
5  player1.sayHello(); // logs "Hello, I'm a player!"
6  player2.sayHello(); // logs "Hello, I'm a player!"
```

Here, we defined the `.sayHello` function 'on' the `Player.prototype` object. It then became available for the `player1` and the `player2` objects to use! Similarly, you can attach other properties or functions you want to use on all `Player` objects by defining them on the objects' prototype (`Player.prototype`).

➢ Object.getPrototypeOf() vs. .__proto__ vs. [[Prototype]]
- .__proto__
  This is a non-standard way of doing so, and **deprecated**.

```
// Don't do this!
player1.__proto__ === Player.prototype; // returns true
player2.__proto__ === Player.prototype; // returns true
```

- [[Prototype]]
  In some places, like **legacy code**, you might also come across [[Prototype]], like player1.[[Prototype]].

➢ The purpose of defining properties and functions on the prototype
1) We can define properties and functions common among all objects on the 'prototype' to save memory.

2) **Prototypal Inheritance**, which (we've referred earlier) is the original object has access to all of its prototype's methods and properties.
 ➢ Perhatikan berikut ini (utk memahami Object.prototype & valueOf())

```
1   // Player.prototype.__proto__
2   Object.getPrototypeOf(Player.prototype) === Object.prototype; // true
3
4   // Output may slightly differ based on the browser
5   player1.valueOf(); // Output: Object { name: "steve", marker: "X", sayName: sayName() }
```

- `Player.prototype` adalah objek prototipe yang diwariskan oleh semua objek `Player`.

- `Object.getPrototypeOf` digunakan untuk mendapatkan prototipe dari `Player.prototype`, yang ternyata adalah `Object.prototype`. Ini menunjukkan bahwa `Player.prototype` mewarisi dari `Object.prototype`.

- Metode `valueOf` berasal dari `Object.prototype`, karena `Player.prototype` mewarisi dari `Object.prototype`.

- Ini menunjukkan bahwa `player1` mewarisi metode `valueOf` dari `Object.prototype`.

 ➢ Perhatikan (utk memahami hasOwnProperty)

```
1   player1.hasOwnProperty('valueOf'); // false
2   Object.prototype.hasOwnProperty('valueOf'); // true
```

- `hasOwnProperty` adalah metode yang digunakan untuk memeriksa apakah properti tersebut dimiliki oleh objek itu sendiri, bukan diwarisi dari prototipe.

- `player1` tidak memiliki properti `valueOf` sendiri, tetapi `Object.prototype` memilikinya.

 ➢ **Prototype Chain (rantai properti)**

- JavaScript menggunakan rantai prototipe untuk mengatur pewarisan properti dan metode.

- Ketika suatu properti atau metode dipanggil, JavaScript akan memeriksa objek itu sendiri. Jika tidak ditemukan, JavaScript akan naik ke rantai prototipe hingga menemukan properti atau metode tersebut atau mencapai `null`.

- `Object.getPrototypeOf(Object.prototype)` mengembalikan `null`, yang menunjukkan akhir dari rantai prototipe.

 ➢ **Note**
  1) Every 'prototype' object inherits from **Object.prototype** by default.
  2) Nilai 'Object.getPrototypeOf()' dari Sebuah Objek hanya satu.

- Setiap objek hanya bisa memiliki satu prototipe. Ini berarti bahwa sebuah objek tidak bisa mewarisi dari beberapa prototipe sekaligus.

- Misalnya, jika kamu memeriksa prototipe dari objek yang dibuat dengan `new Player()`, hasilnya adalah `Player.prototype`. Dan, jika kamu memeriksa prototipe dari `Player.prototype`, hasilnya adalah `Object.prototype`.

D. Recommended method for Prototypal Inheritance
  - `Object.getPrototypeOf()` to 'get' or view the `prototype` of an object,
  - `Object.setPrototypeOf()` to 'set' or mutate it.
  Let's see how it works by adding a 'Person' Object Constructor to the 'Player' example, and making 'Player' inherit from Person:

```javascript
1   function Person(name) {
2       this.name = name;
3   }
4
5   Person.prototype.sayName = function() {
6       console.log(`Hello, I'm ${this.name}!`);
7   };
8
9   function Player(name, marker) {
10      this.name = name;
11      this.marker = marker;
12  }
13
14  Player.prototype.getMarker = function() {
15      console.log(`My marker is '${this.marker}'`);
16  };
17
18  Object.getPrototypeOf(Player.prototype); // returns Object.prototype
19
20  // Now make `Player` objects inherit from `Person`
21  Object.setPrototypeOf(Player.prototype, Person.prototype);
22  Object.getPrototypeOf(Player.prototype); // returns Person.prototype
24  const player1 = new Player('steve', 'X');
25  const player2 = new Player('also steve', 'O');
26
27  player1.sayName(); // Hello, I'm steve!
28  player2.sayName(); // Hello, I'm also steve!
29
30  player1.getMarker(); // My marker is 'X'
31  player2.getMarker(); // My marker is 'O'
```

Note: Using setPrototypeOf() after objects(player1 & player2) have already been created can result in performance issues.