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:

```
// example one
const playerOneName = "tim";
const playerTwoName = "jenn";
const playerOneMarker = "X";
const playerTwoMarker = "O";

// example two
const playerOne = {
  name: "tim",
  marker: "X"
};

const playerTwo = {
  name: "jenn",
  marker: "O"
```

The benefit of the example two is huge:

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

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

example one:

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

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

```
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:

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

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

```
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:

```
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

```
Object.getPrototypeOf(player1) === Player.prototype; // returns true
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'

```
Player.prototype.sayHello = function() {
    console.log("Hello, I'm a player!");
};

player1.sayHello(); // logs "Hello, I'm a player!"
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())

```
// Player.prototype.__proto__
Object.getPrototypeOf(Player.prototype) === Object.prototype; // true

// Output may slightly differ based on the browser
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)

```
player1.hasOwnProperty('valueOf'); // false
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. Easiest way to understand (object syntax)

```
let bag = {  //before
    pen: 3
};
bag.pencil = 2;  //process

let bag {  //after-process
    pen: 3,
    pencil: 2
};
```

- Understanding 'this'

```
let animal = {
    eat() {
      this.full = true;
    }
};

let rabbit = {
    __proto__: animal
};
rabbit.eat();
//ketika fungsi eat() dipanggil oleh rabbit
//maka 'this' di dalam eat() adalah rabbit
//sehingga menjadi rabbit.full = true
```

E. This

In general, 'this' references the object that is currently calling the function. Behavior of the this keyword changes between strict and non-strict modes.

```
let counter = {
  count: 0,
  next: function () {
    return ++this.count;
  },
};

counter.next();
```

Means 'return ++counter.count;
Because counter is the object that call next() function

- **Strict Mode (**available since ECMAScript 5.1**)**

To enable the strict mode, you use the directive `"use strict"` at the beginning of the JavaScript file. If you want to apply the strict mode to a specific function only, you place it at the top of the function body.

The 'strict' mode applies to both function and nested functions:

```
"use strict";
function show() {
    console.log(this === undefined);
}
show();
```

```
function show() {
    "use strict";
    console.log(this === undefined); // true

    function display() {
        console.log(this === undefined); // true
    }
    display();
}

show();
```

- Global context

In the global context, the `this` references the global object, which is the `window` object on the web browser or `global` object on Node.js.

This behavior is consistent in both strict and non-strict modes.

```
console.log(this === window); // true
this.color= 'Red';
console.log(window.color); // 'Red'
```

- Function context

In JS, you can call a functin in the following ways:

- Function invocation
- Method invocation
- Constructor invocation
- Indirect invocation

1) Simple function invocation

> In non-strict mode, 'this' references the global object

```
function show() {
   console.log(this === window); // true
}
show();
```

> In strict mode, 'this' is 'undefined'

```
"use strict";
function show() {
    console.log(this === undefined);
}
show();
```

2) Method invocation

When you call method of an object, JavaScript sets 'this' to the object that own the method.

```
let car = {
    brand: 'Honda',
    getBrand: function () {
        return this.brand;
    }
}
console.log(car.getBrand()); // Honda
```

Since a method is a property of an object which is a value, you can store it in a variable.

```
let brand = car.getBrand.bind(car);
console.log(brand()); // Honda
```

We use .bind() due to 'brand' become callback(function as an argument) in console.log(brand())

```
let car = {
    brand: 'Honda',
    getBrand: function () {
        return this.brand;
    }
}
let bike = {
    brand: 'Harley Davidson'
}
let brand = car.getBrand.bind(bike);
console.log(brand());
```

In here we modify so that 'this' keyword is bound to the 'bike' object.

Output:

```
Harley Davidson
```

3) Constructor invocation
When you use the new keyword to create an instance of a function object, you use the function as a constructor.

```
function Car(brand) {
    this.brand = brand;
}
Car.prototype.getBrand = function () {
    return this.brand;
}

let car = new Car('Honda');
console.log(car.getBrand());
```

JavaScript creates a new object(car) and sets ‘this’ to the newly created object(car).

4) Indirect invocation
Function has two methods: call() and apply(). These methods allow you to set the this value when calling a function.

```
function getBrand(prefix) {
    console.log(prefix + this.brand);
}
let honda = {
    brand: 'Honda'
};
let audi = {
    brand: 'Audi'
};

getBrand.call(honda, "It's a ");
getBrand.call(audi, "It's an ");
```

Output:

```
It's a Honda
It's an Audi
```

apply() is similar to the call() except that its second argument is an array of arguments:

```
getBrand.apply(honda, ["It's a "]); // "It's a Honda"
getBrand.apply(audi, ["It's an "]); // "It's a Audi"
```

- Arrow function

- Pada fungsi biasa, `this` mengacu pada objek yang memanggil fungsi tersebut (misalnya, dalam event handler, `this` bisa mengacu pada elemen yang menangani event tersebut).
- Pada fungsi panah, `this`tidak membuat konteks eksekusi sendiri, melainkan mewarisi 'this' dari tempat di mana fungsi tersebut didefinisikan.

```
let getThis = () => this;
console.log(getThis() === window); // true
```

Pada contoh di atas karena 'this' dalam fungsi panah dan fungsi panah tersebut didefinisikan di tingkat global (bukan di dalam objek atau fungsi lain), maka 'this' mengacu pada objek global (dalam browser yaitu window).

F. 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:

```
function Person(name) {
  this.name = name;
}

Person.prototype.sayName = function() {
  console.log(`Hello, I'm ${this.name}!`);
};

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

Player.prototype.getMarker = function() {
  console.log(`My marker is '${this.marker}'`);
};

Object.getPrototypeOf(Player.prototype); // returns Object.prototype

// Now make `Player` objects inherit from `Person`
Object.setPrototypeOf(Player.prototype, Person.prototype);
Object.getPrototypeOf(Player.prototype); // returns Person.prototype
```

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

player1.sayName(); // Hello, I'm steve!
player2.sayName(); // Hello, I'm also steve!

player1.getMarker(); // My marker is 'X'
player2.getMarker(); // My marker is 'O'
```

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

G. call(), apply() & bind() method

- **call() :** to copy properties from one constructor into another constructor

syntax:

```
call(thisArg)
call(thisArg, arg1)
call(thisArg, arg1, arg2)
call(thisArg, arg1, arg2, /* …, */ argN)
```

kode di bawah ini meng-copy property this.name & this.level dari constructor 'Hero' (sebenarnya kode ini meng-copy parameter name & level pada warior, lalu memasukannya ke constructor 'Hero', untuk lebih mudah memahaminya cek slide 6 no 4) indirect invocation)

```
// Initialize constructor functions
function Hero(name, level) {
  this.name = name;
  this.level = level;
}

function Warrior(name, level, weapon) {
  Hero.call(this, name, level);

  this.weapon = weapon;
function Healer(name, level, spell) {
  Hero.call(this, name, level);

  this.spell = spell;
}

// Link prototypes and add prototype methods
Object.setPrototypeOf(Warrior.prototype, Hero.prototype);
Object.setPrototypeOf(Healer.prototype, Hero.prototype);

Hero.prototype.greet = function () {
  return `${this.name} says hello.`;
}

Warrior.prototype.attack = function () {
  return `${this.name} attacks with the ${this.weapon}.`;
}

Healer.prototype.heal = function () {
  return `${this.name} casts ${this.spell}.`;
}

// Initialize individual character instances
const hero1 = new Warrior('Bjorn', 1, 'axe');
const hero2 = new Healer('Kanin', 1, 'cure');
```

Contoh output:

```
hero1.attack();
```

```
Console
"Bjorn attacks with the axe."
```

- **apply() :** apply() is similar to the call() except that its second argument is an array of arguments:

```
getBrand.apply(honda, ["It's a "]); // "It's a Honda"
getBrand.apply(audi, ["It's an "]); // "It's a Audi"
```

- **bind() :** function borrowing & prevent losing 'this'
  - funciton borrowing. In example below 'member' object borrows the fullname method from the 'person' object.

```
const person = {
  firstName:"John",
  lastName: "Doe",
  fullName: function () {
    return this.firstName + " " + this.lastName;
  }
}
const member = {
  firstName:"Hege",
  lastName: "Nilsen",
}

let fullName = person.fullName.bind(member);
```

  - prevent losing 'this'
    When a function is used as a callback, 'this' is lost.
    In example below, it'll display 'undefined' instead of display person name after 3sec

```
const person = {
  firstName:"John",
  lastName: "Doe",
  display: function () {
    let x = document.getElementById("demo");
    x.innerHTML = this.firstName + " " + this.lastName;
  }
}
setTimeout(person.display, 3000);
```

    .bind() method solves this problem

```
let display = person.display.bind(person);
setTimeout(display, 3000);
```

H. for..in loop

The for...in loop iterates its own properties & inherited properties.

```
for(let prop in rabbit) {
  let isOwn = rabbit.hasOwnProperty(prop);

  if (isOwn) {
    alert(`Our: ${prop}`); // Our: jumps
  } else {
    alert(`Inherited: ${prop}`); // Inherited: eats
  }
}

// Object.keys only returns own keys
alert(Object.keys(rabbit)); // jumps

// for..in loops over both own and inherited keys
for(let prop in rabbit) alert(prop); // jumps, then eats
```

Example: (we can also use .hasOwnProperty for obj has its on (not inherited)

I. **Task :** https://javascript.info/prototype-inheritance#tasks