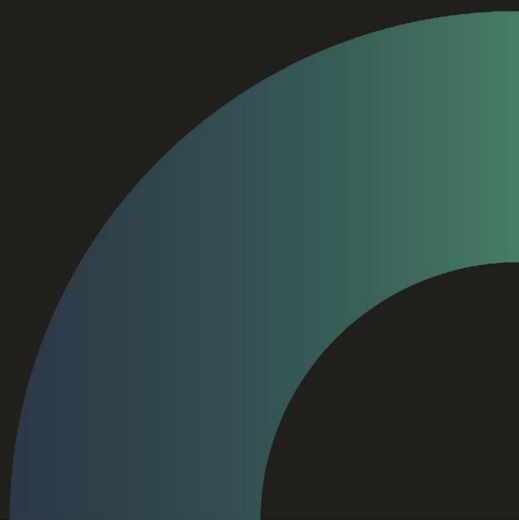


Full Stack Web Development

Object Oriented Programming

-
- Object
 - Class
 - Encapsulation
 - Inheritance
- 

What is Object Oriented Programming ?

Object oriented programming is a programming model that revolves around the concept of **objects and classes**.

The concept is to create reusable pieces of code within classes that create objects, and utilize these objects to interact with each other.

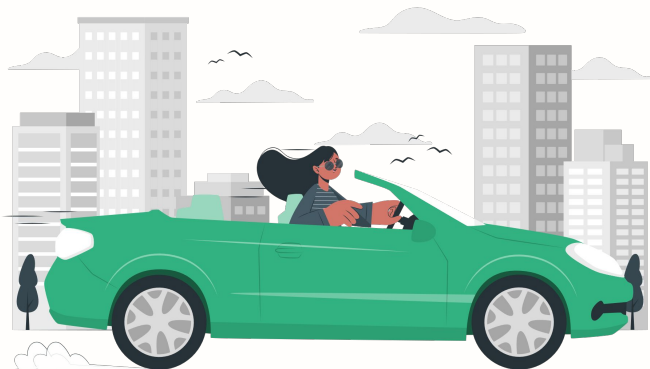


What is Object ?

Object is an entity having state and behavior (properties and method)

The **Object** type represents one of [JavaScript's data types](#). It is used to store various keyed collections and more complex entities.

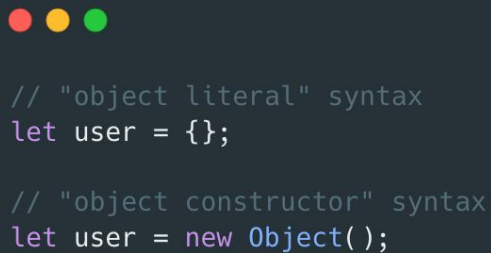
Why we use object ? because objects can store a lot of data along with its properties.



```
const car = {  
  brand: "BMW",  
  model: "M135i xDrive",  
  price: 800000000  
}
```

Creating an Object

An empty object can be created using one of two syntaxes :



```
// "object literal" syntax
let user = {};

// "object constructor" syntax
let user = new Object();
```

Properties & Methods

An object is a collection of properties, and a property is an association between a **name (or key)** and a **value**. A property's value can be a function, in which case the property is known as a method.



```
let user = {  
  name: "David",  
  greet() {  
    console.log("Hello!")  
  }  
};
```

Add & Delete Property



```
let person = {  
  name: "Frengky",  
  age: 24,  
}  
  
// add property "hobby" and it's value into "person"  
person.hobby = "Coding";  
console.log(person);  
// { name: "frengky", age: 24, hobby: "Coding" }  
  
// delete "age" property from "person"  
delete.person.age;  
console.log(person);  
// { name: "frengky", hobby: "Coding" }
```

Accessing Value

Access with **dot (.)**

```
let person = {  
  name: "Frengky",  
  age: 24,  
}  
  
// accessing props  
console.log(person.name);
```

Access with **square bracket ([])**

```
let person = {  
  name: "Frengky",  
  age: 24,  
}  
  
// accessing props  
console.log(person["name"]);  
// Frengky
```


Optional Chaining '?'

The optional chaining **?** is a **safe way to access nested object properties**, even if an intermediate property doesn't exist.

```
let user = {}; // a user without "address" property

console.log(user.address); // undefined
console.log(user.address.street); // Error!

// Optional chaining
console.log(user.address?.street); // undefined but not error
```

Accessing Key

There are several way to access property in an object, one of them is using **Object.keys()**



```
let person = {  
  name: "Frengky",  
  age: 24,  
}  
  
// accessing props  
console.log(Object.keys(person));  
// ['name', 'age']
```

The "for..in" loop

To walk over all keys of an object, there exists a special form of the loop: for..in. This is a completely different thing from the for(;;) construct that we studied before.



```
let person = {  
  name: "Frengky",  
  age: 24,  
}  
  
for (let key in person) {  
  // keys  
  console.log(key); // name, age  
  // values for the keys  
  console.log(person[key]); // Frengky, 24  
}
```

Getter & Setter

Accessor properties are represented by “**getter**” and “**setter**” methods. In an object literal they are denoted by **get** and **set**.

```
let user = {
  firstName: "John",
  lastName: "Smith",

  get fullName() {
    return `${this.name} ${this.surname}`;
  },


  set fullName(value) {
    const splittedValues = value.split(" ");
    this.firstName = splittedValues[0];
    this.lastName = splittedValues[1];
  },
};

// set fullName is executed with the given value.
user.fullName = "Alice Cooper";

console.log(user.firstName); // Alice
console.log(user.lastName); // Cooper
```

Destructuring Assignment

The **destructuring assignment** is a JavaScript expression that makes it possible to **unpack values from arrays, or properties from objects**, into distinct variables.



```
let a, b;  
[a, b] = [10, 20];  
  
console.log(a);  
// expected output: 10  
  
console.log(b);  
// expected output: 20
```

Spread Operator

Spread operator (`...`) allows us to quickly copy all or part of an existing array or object into another array or object.

```
const dataOne = [1, 2, 3];
const dataTwo = [4, 5, 6];
const finalDataList = [...dataOne, ...dataTwo];
console.log(finalDataList);

const objectOne = {
  name: "David",
};
const objectTwo = {
  email: "david@mail.com"
}
const finalObject = {...objectOne, ...objectTwo};
console.log(finalObject);
```

Using “this” Keyword

- In JavaScript, the **this** keyword refers to an **object**. Which object **depends on how this is being invoked** (used or called).
- The **this** keyword refers to different objects depending on how it is used :
 - In an **object method**, this refers to the **object**.
 - **Alone**, this refers to the **global object**.
 - In a **function**, this refers to the **global object**.
 - In a **function, in strict mode**, this is **undefined**.
 - In an **event**, this refers to the **element** that received the event.
 - Methods like **call()**, **apply()**, and **bind()** can refer this to **any object**.



Using "this" Keyword



```
const person = {
  firstName: "Frengky",
  lastName: "Sihombing",
  greet() {
    console.log(`Hello ${this.firstName}`);
  },
};
person.greet(); // Hello Frengky
```


Here are few object built-in method:

- [Object.assign\(\)](#), Copies the values of all enumerable own properties from one or more source objects to a target object.
- [Object.create\(\)](#), Creates a new object with the specified prototype object and properties.
- [Object.entries\(\)](#), Returns an array containing all of the [key, value] pairs of a given object's **own** enumerable string properties.
- [Object.freeze\(\)](#), Freezes an object. Other code cannot delete or change its properties.
- [Object.is\(\)](#), Compares if two values are the same value. Equates all NaN values (which differs from both Abstract Equality Comparison and Strict Equality Comparison).
- etc

What is Class ?

Class are a template for creating objects. They encapsulate data with code to work on that data.

Class are in fact "**special functions**", and just as you can define function expressions and function declarations, the class syntax has two components: [class expressions](#) and [class declarations](#).



Create a Class

```
// Class declaration
class User {
  greeting() {
    console.log("Hello World");
  }
}
```

```
// Class expression
const User = class {
  greeting() {
    console.log("Hello World");
  }
}
```

Create an Object from Class

```
class User {  
  greeting() {  
    console.log("Hello World!");  
  }  
}  
  
const user = new User();  
user.greeting();
```

Constructor

The **constructor** method is a special method for creating and initializing objects created within a class.

The **constructor** method is called automatically when a class is initiated, and it has to have the exact name "constructor", in fact, if you do not have a constructor method, JavaScript will add an invisible and empty constructor method.

A class cannot have more than one **constructor** method.

```
class User {  
  name = "";  
  
  constructor(name) {  
    this.name = name;  
  }  
  
  greeting() {  
    console.log(`Hello ${this.name}!`);  
  }  
}  
  
const user = new User("David");  
user.greeting();
```

Access Modifier

One of the most important principles of object oriented programming – delimiting internal interface from the external one.

The followings are the access modifiers in most of the object-oriented programs :

- **Public** : accessible from anywhere
- **Private** : accessible only from inside the class
- **Protected** : accessible only from inside the class and those extending it

Note : Protected field are not implemented in JavaScript on the language level



Public & Private Properties

On the language level, **#** is a special sign that the field is private. We can't access it from outside or from inheriting classes.

```
class User {  
  // public property  
  name;  
  
  // private property  
  #email;  
  
  constructor(name, email) {  
    this.name = name;  
    this.#email = email;  
  }  
  
  getEmail() {  
    return this.#email;  
  }  
}  
  
const user = new User("David", "david@mail.com");  
console.log(user.name); // David  
console.log(user.#email); // Not accessible
```

Static Properties

Sometimes we want to use methods and properties that **do not necessarily need an instance of that class being created**, but we still want them to be related to that class.

These are called **static properties**.

```
class DB {  
  static #connection = "";  
  
  static #initializeConnection() {  
    const randomNum = Math.ceil(Math.random() * 100);  
    DB.#connection = `New Database Connection ${randomNum}`;  
  }  
  
  static getConnection() {  
    if (!DB.#connection) {  
      DB.#initializeConnection();  
    }  
  
    return DB.#connection;  
  }  
}  
  
// first call  
console.log(DB.getConnection());  
  
// second call  
console.log(DB.getConnection());
```


Encapsulation

What is **Encapsulation** ?

Encapsulation is a concept in OOP used to **bundle data with methods acting upon those data in one unit** such as a class or object in Javascript.

Using encapsulation, we can have more control on how to **control and validate** our data.



- **Data Hiding** : it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.
- **Increased Flexibility** : We can make the variables of the class read-only or write-only depending on our requirement. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program
- **Reusability** : Encapsulation also improves the re-usability and is easy to change with new requirements.
- **Testing code is easy** : Encapsulated code is easy to test for unit testing.

Encapsulation example

Here's a simple implementation of encapsulation using Javascript Classes.

In this example, we've created an `Employee` class that has a property called `employeeName`. Then, we've created getter and setter methods.

At first, this may not seem much since the setter method only changes `employeeName`'s value, and the getter method only returns `employeeName`'s value.

```
class Employee {  
  constructor() {  
    this.employeeName;  
  }  
  
  getEmployeeName() {  
    return this.employeeName;  
  }  
  
  setEmployeeName(newName) {  
    this.employeeName = newName;  
  }  
}
```

Encapsulation example

And you also might ask the question,

“Why don’t we just directly manipulate the properties within that object, rather than having to create separate methods for them?”

Like in this example, we directly manipulate the property and we would also achieve the same results.

```
class Employee {  
    constructor() {  
        this.employeeName;  
    }  
}  
  
const employee = new Employee();  
employee.employeeName = "Jhony"
```

Encapsulation example

However, we can take a step further and add some validation to the setter method.

Of course, when setting a name for an employee, we would want the data type to be a string. Not a number, object, or anything else.

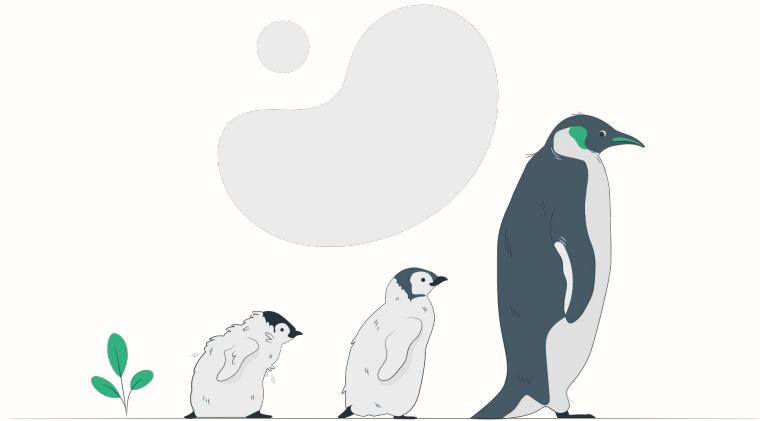
Now, in the setter method we've added a condition to throw an error every time we try to set the employee's name using a data type other than a string.

```
class Employee {  
  constructor() {  
    this.employeeName;  
  }  
  
  getEmployeeName() {  
    return this.employeeName;  
  }  
  
  setEmployeeName(newName) {  
    if(typeof newName !== "string") {  
      throw new Error("Name should be a string")  
    }  
    this.employeeName = newName;  
  }  
}
```

Inheritance

What is **inheritance** ?


Inheritance is another concept in OOP that allows **classes to be created based on other classes**, thus **inheriting the parent's properties and methods**.



Inheritance example

Let's take a look at an example. Say we have a shop application, and inside the application we have a class called *Product* used to store product data.

For the sake of the example, let's assume *Product* already has its own getter and setter methods.



```
class Product {  
    constructor() {  
        this.productName;  
        this.price;  
    }  
    //getters and setters...  
}
```

Inheritance example

Now, as the business owner, you decided to sell books in your shop application. And for that, we need to create a *Book* class.

As you can see, *Book* has a very similar structure to our previous class, *Product*. The only difference it has is that *Book* has a property called *author*.

Instead of repeating the same code, we can **extend** *Book* to have the same properties as *Product*.

```
class Book {  
  constructor() {  
    this.productName;  
    this.price;  
    this.author;  
  }  
  //getters and setters...  
}
```


Inheritance example

Now *Book* will have the same properties and methods as *Product*.

The most important thing to note here is the ***extends*** keyword, it is used to specify from which class do we want our new class to be inherited properties and methods from.

The next thing you need to do is use the ***super*** keyword to access the parent class's properties and methods.

```
class Book extends Product {  
  constructor() {  
    super();  
    this.author;  
  }  
  
  getAuthor() {  
    return this.author;  
  }  
  
  setAuthor(authorName) {  
    this.author = authorName  
  }  
}
```

Inheritance example

And now, when we create an instance of *Book*, it will automatically have the methods and properties we specified for *Book*, and it will also ***inherit*** methods and properties from its parent class which is ***Product***.



```
const book = new Book();  
book.setAuthor("J. R. R. Tolkien");  
book.productName;
```

"instanceof" operator

The **instanceof** operator allows to check whether an object belongs to a certain class. It also takes inheritance into account.

```
class Animal {}  
class Rabbit extends Animal {}  
class Tree {}  
  
const rabbit = new Rabbit();  
console.log(rabbit instanceof Animal); // true  
console.log(rabbit instanceof Rabbit); // true  
console.log(rabbit instanceof Tree); // false
```

Class “super” Keyword

- The **super** keyword is used to call the constructor of its parent class to access the parent's properties and methods.
- By calling the **super** method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

Class "super" Keyword

```
class User {
  name;
  #code = "";

  constructor(codePrefix = "") {
    const randomNumber = Math.round(Math.random() * 1000);
    this.#code = `${codePrefix}${randomNumber.toString().padStart(6, "0")}`;
  }

  getCode() {
    return this.#code;
  }
}

class Student extends User {
  constructor() {
    super("STD");
  }
}

class Employee extends User {
  constructor() {
    super("EMP");
  }
}

const student = new Student();
console.log(student.getCode()); // STDXXXXXX

const employee = new Employee();
console.log(employee.getCode()); // EMPXXXXXX
```

Exercise 1

- Create a function to calculate array of student data
- The object has this following properties :
 - Name → String
 - Email → String
 - Age → Date
 - Score → Number
- Parameters : **array of student**
- Return values :
 - Object with this following properties :
 - Score
 - Highest
 - Lowest
 - Average
 - Age
 - Highest
 - Lowest
 - Average

Exercise 2

- Create a program to create transaction
- Product Class
 - Properties
 - Name
 - Price
- Transaction Class
 - Properties
 - Total
 - Product
 - All product data
 - Qty
 - Add to cart method → Add product to transaction
 - Show total method → Show total current transaction
 - Checkout method → Finalize transaction, return transaction data

Thank You!

