Purwadhika
Digital Technology School

Full Stack AI Software Development

Object Oriented Programming

Job Connector Program

Outline

Core Principles
Learn the 4 pillars: Encapsulation, Abstraction, Inheritance, Polymorphism – the foundation of structured, reusable, and maintainable code.

Building Blocks
Work with classes, objects, constructors, properties, methods, and access modifiers (public, private, protected) to model real-world entities.

Advanced Features
Go deeper with inheritance, polymorphism, interfaces, abstract classes, and static members, enabling flexibility and enforceable contracts in design.

What is Object Oriented Programming?

Object-Oriented Programming (OOP) is a way of writing programs that focuses on **objects** and **classes**.

- **A class** is like a blueprint or template (for example, a plan for building a car).
- **An object** is something created from that blueprint (for example, an actual car built from the plan).

With OOP, instead of writing the same code again and again, you put related code inside classes and then create objects from them. These objects can then work together and interact, making programs easier to understand, reuse, and maintain.

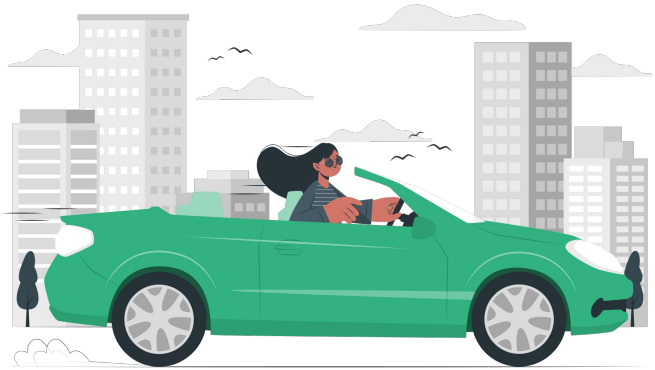
What is Object?

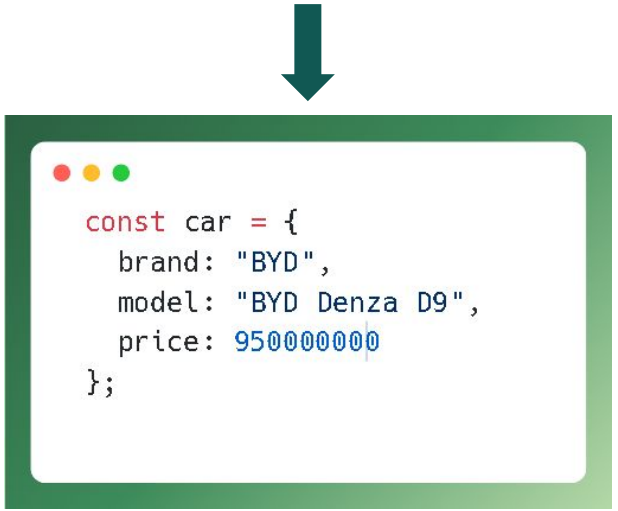
An object is like containers where we can group related information together. An object is something that has two main parts:

- State (properties) ➜ the data it holds.
- Behavior (methods) ➜ the actions it can do.

Why do we use objects?

Because they let us store many pieces of related data in one place (like brand, speed, and actions of a car) instead of keeping them separate. This makes our code organized and easier to work with.

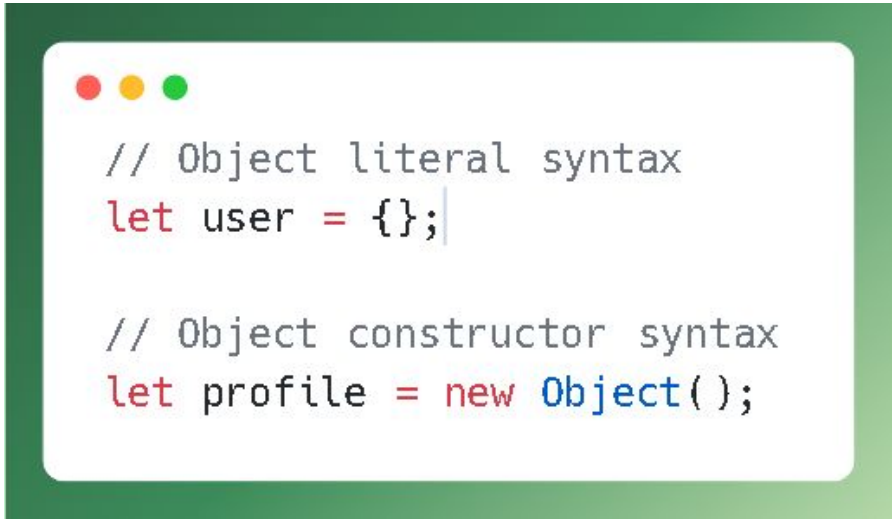




```
const car = {
  brand: "BYD",
  model: "BYD Denza D9",
  price: 950000000
};
```

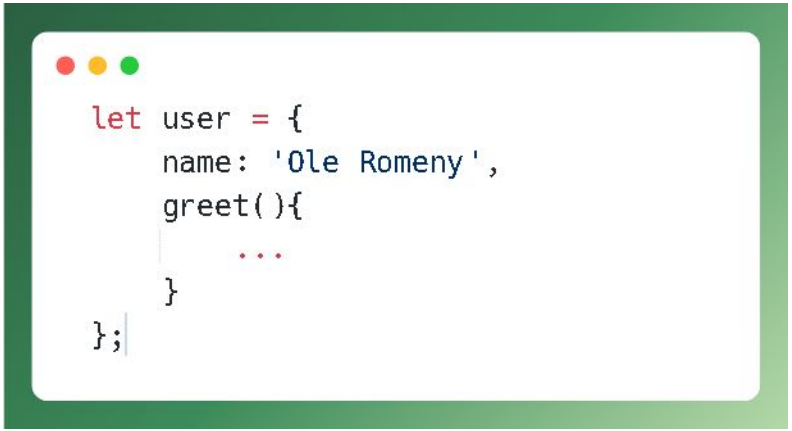
Creating an Object

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

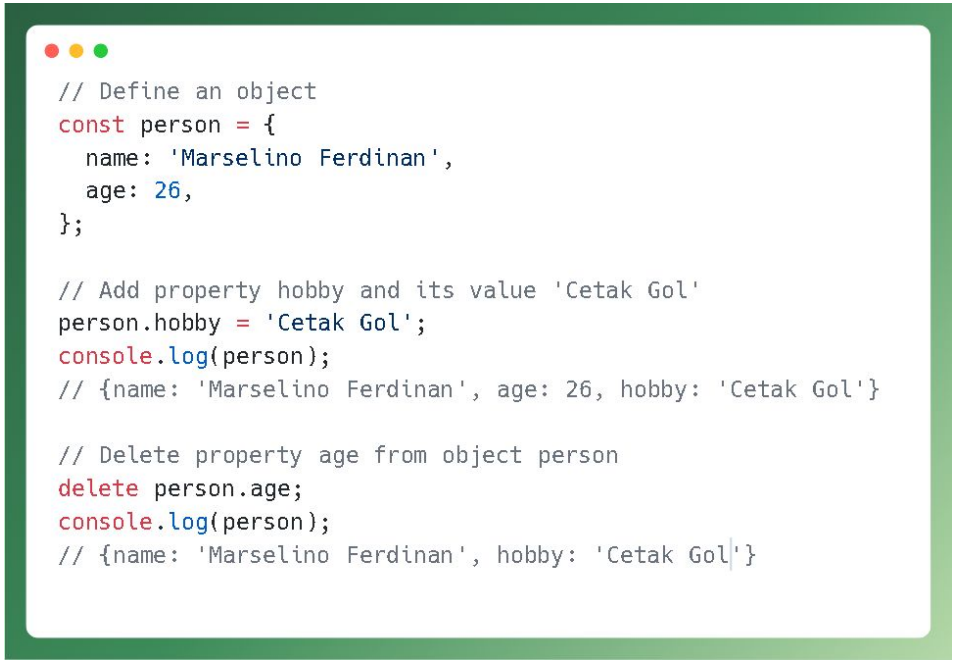


Properties & Methods

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.



Add & Delete Property



```
// Define an object
const person = {
  name: 'Marselino Ferdinan',
  age: 26,
};

// Add property hobby and its value 'Cetak Gol'
person.hobby = 'Cetak Gol';
console.log(person);
// {name: 'Marselino Ferdinan', age: 26, hobby: 'Cetak Gol'}

// Delete property age from object person
delete person.age;
console.log(person);
// {name: 'Marselino Ferdinan', hobby: 'Cetak Gol'}
```

Accessing Value

Access with **dot (.)**

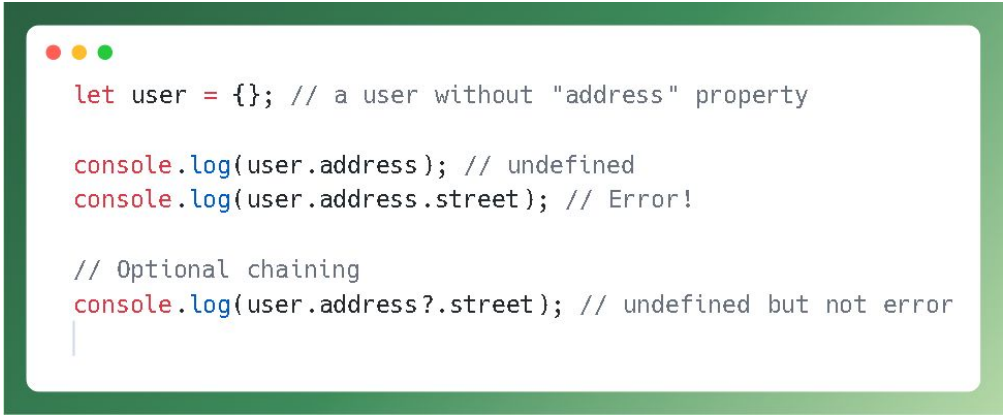
```
let person = {
    name: "Frengky",
    age: 24,
}

// Accessing props
console.log(person.name);
```

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

```
let person = {
    name: "Frengky",
    age: 24,
}

// Accessing props
console.log(person['name']);
```

Optional Chaining '?.'

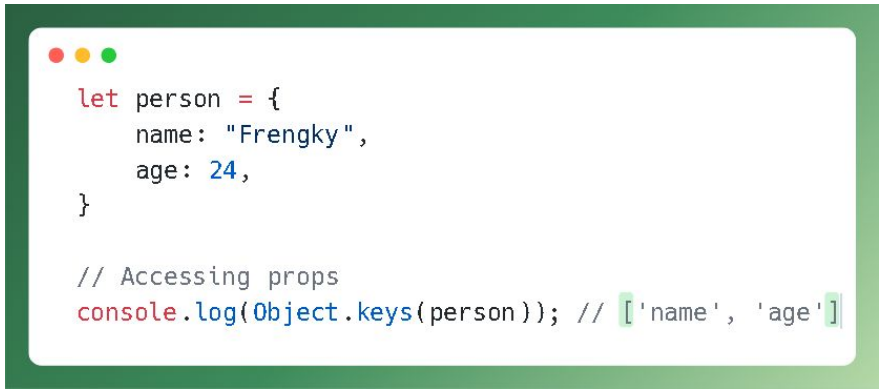
The optional chaining **?.** is a **safe way to access nested object properties**, even if an intermediate property doesn't exist.



```
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']
```

Mutable vs Immutable

- **Mutable** is a type of variable that can be changed. (contains of: non primitive) it is also called as **reference type**
- **Immutable** are the objects whose state cannot be changed once the objects is created. (contains of: primitive) immutable it is also called as **value type**.
- Declaring variable with **const** doesn't make the value immutable. **It depends on data type.**

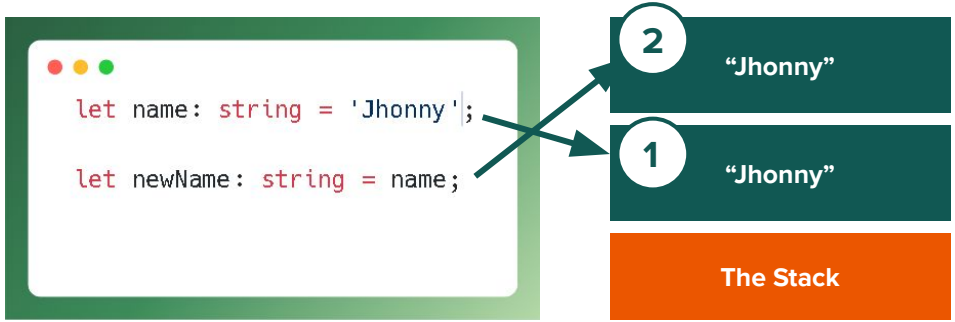
Primitive	
String	Used to represent textual data
Number & BigInt	Used to hold decimal values as well as values without decimals
Boolean	Represents a logical entity and can have two values: true and false
Null	Has exactly one value: null. Represents the intentional absence of any object value
Undefined	A variable that has not been assigned a value has the value undefined

Non Primitive	
Object	Is an entity having properties and methods (keyed collection) → Will be explained in the next session
Array	Used to store more than one element under a single variable → Will be explained in the next session

Mutable vs Immutable

Value types are been stored on the Stack in our memory.

When storing a value type in memory, it adds an element to the top of the stack with the value of the newly created variable. When creating a new variable and assigned the first one to the new one, it adds a new element on top of the stack with the value of the new variable.

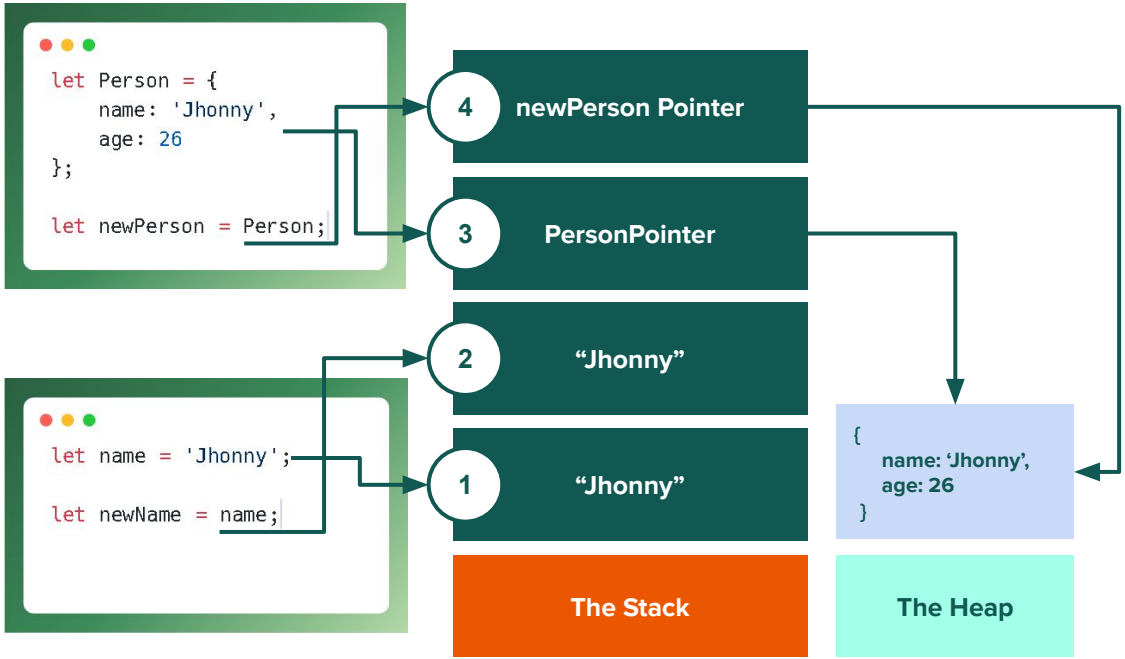


The first variable — **name** gets into the stack with the value of the variable data. Then, the **newName** gets into the stack in a new memory location with the value of the variable data.

Mutable vs Immutable

Reference types are been stored on the Heap. The Heap, indifference from the stack, has no order of how to store the data.

When storing a reference type in memory, it adds a new element to the top of the stack, when its value is a pointer/reference to the address of the object that has been stored on the heap.

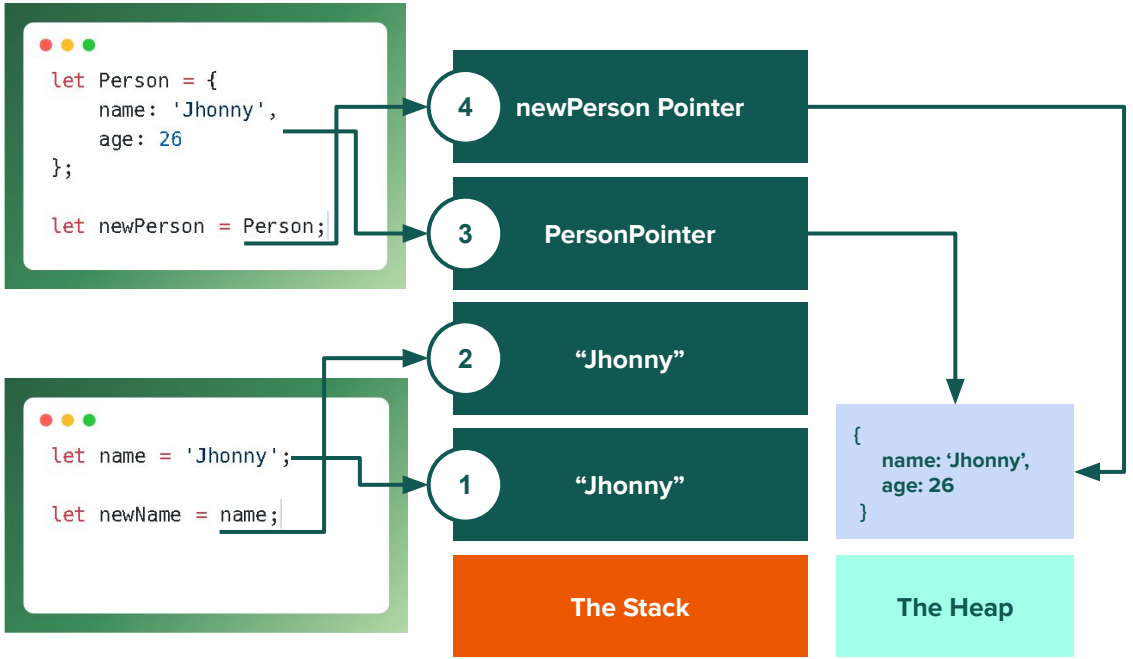


Mutable vs Immutable

Immutable

Try to change value in **newName** variable, check out the result!

Does that change **name** variable value?

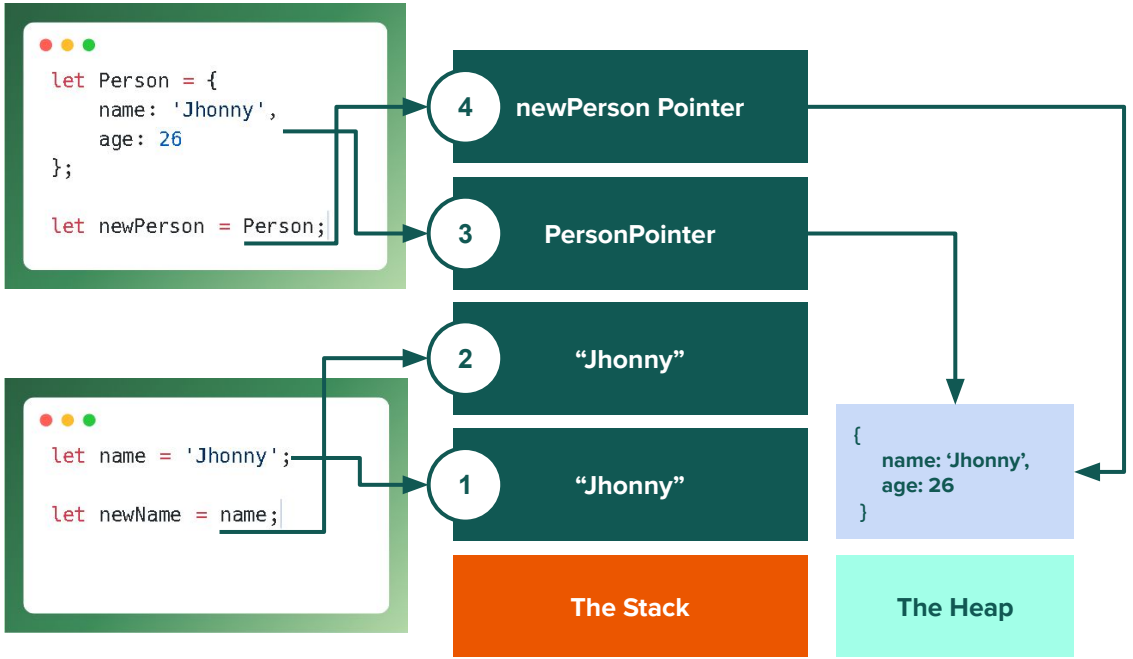


Mutable vs Immutable

Mutable

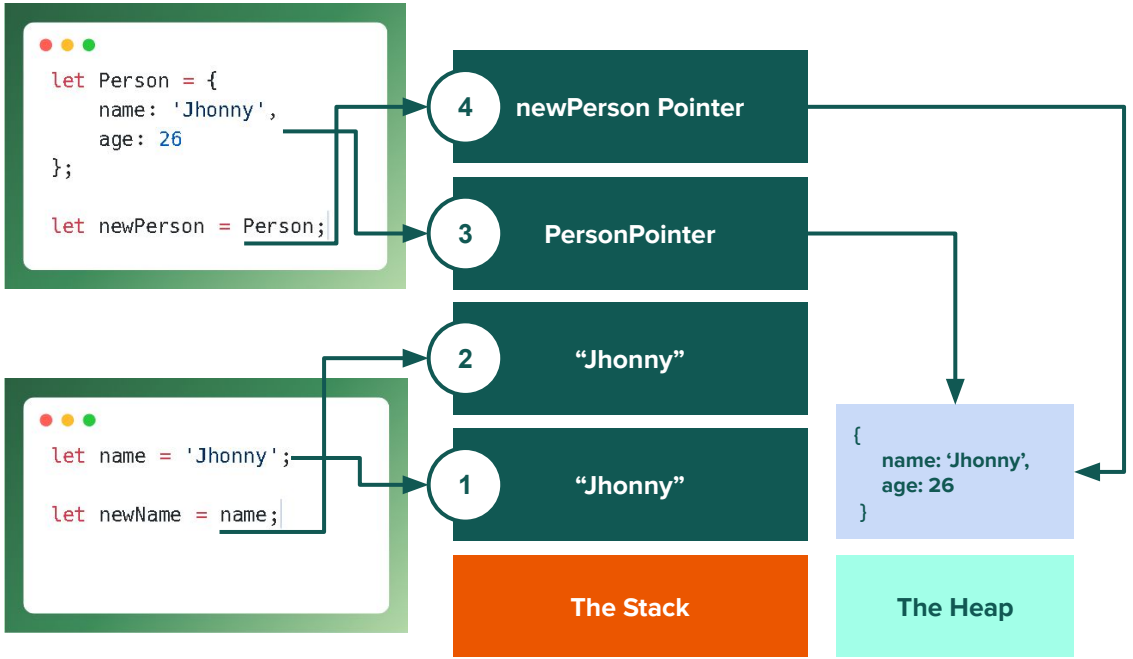
Try to change value in **newPerson** variable, check out the result!

Does that change **Person** variable value?



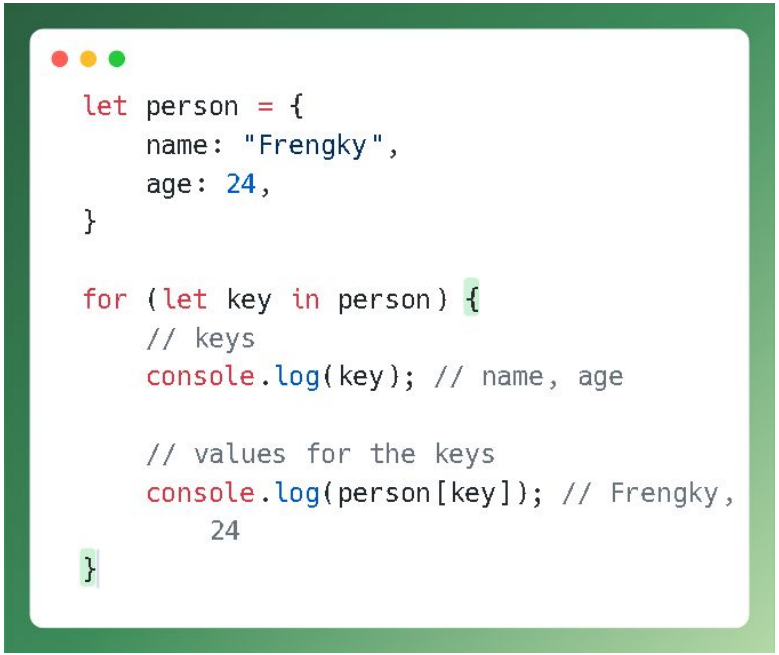
Mutable vs Immutable

With that in mind, we can say that a **value type is immutable** where a **reference type is mutable.**



The "for..in" loop

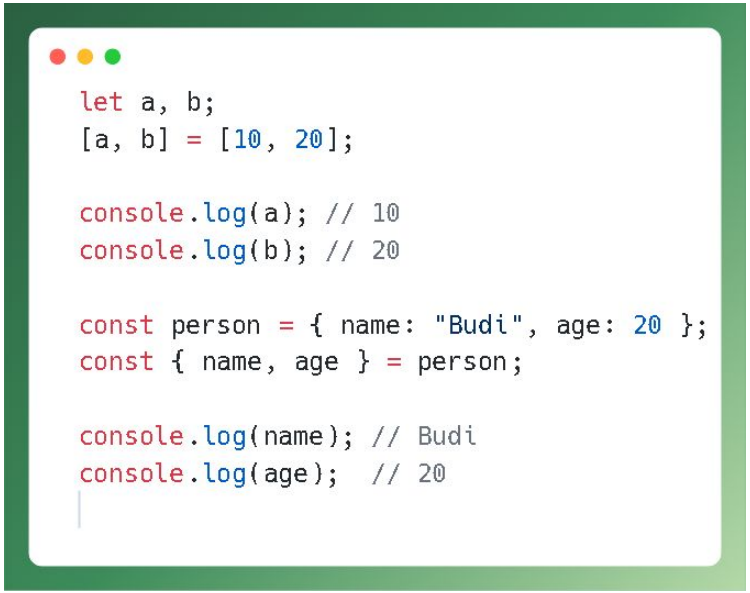
Used to iterate over the keys (property names) of an object.



```
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
}
```

Destructuring Assignment

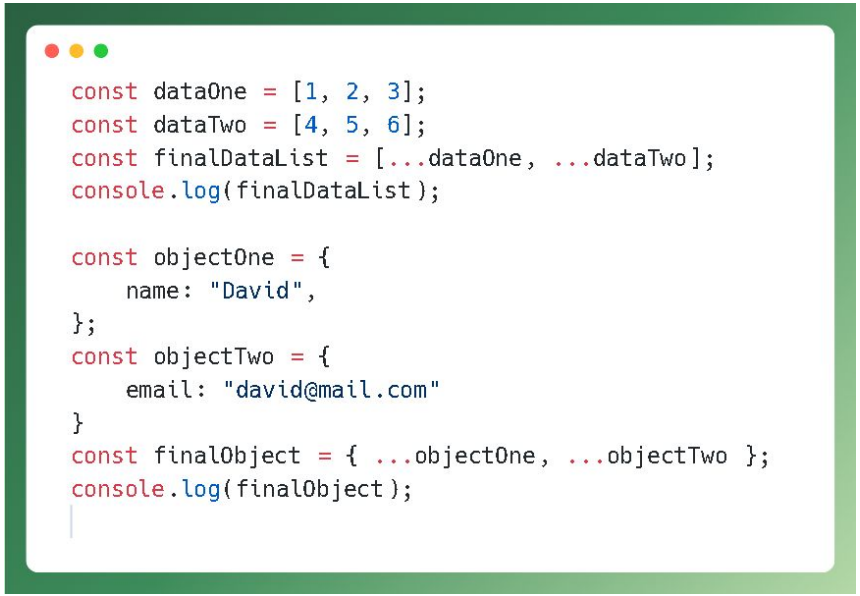
The **destructuring assignment** is a JavaScript expression that makes it possible to **unpack values from arrays, or properties from objects**, into distinct variables.



```
let a, b;
[a, b] = [10, 20];

console.log(a); // 10
console.log(b); // 20

const person = { name: "Budi", age: 20 };
const { name, age } = person;

console.log(name); // Budi
console.log(age);  // 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

this refers to an object, and the object it refers to depends on how it’s called.

Context	this Refers To
In an object method	The object itself
Alone	The global object (window in browsers)
In a regular function	The global object
In strict mode ('use strict')	undefined
In an event handler	The HTML element that received the event
Using call(), apply(), or bind()	Any object you specify

this is ***dynamic***, its value depends on how and where the function is invoked, not where it’s defined.

Using “this” Keyword

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

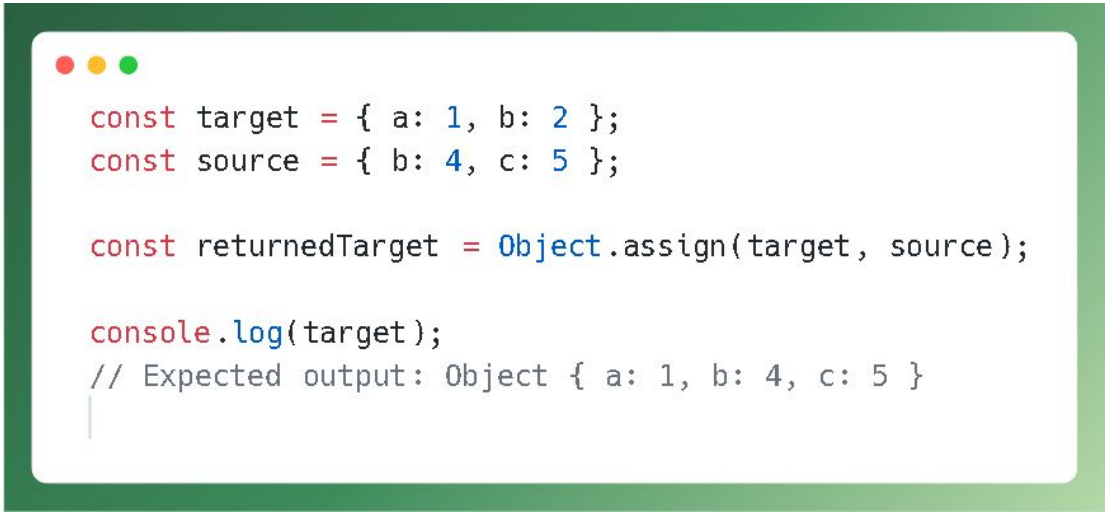
Object Built-In Method

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...

Object Built-In Method Example

The purpose of the code is to demonstrate how to merge the properties of two objects (target and source) into a single object using the Object.assign() method.

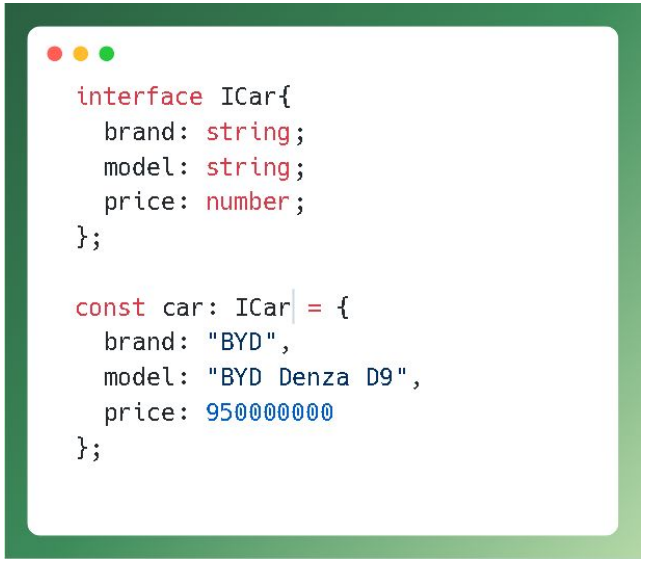


```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);
// Expected output: Object { a: 1, b: 4, c: 5 }
```

TypeScript Interface

An interface in **TypeScript** defines the structure of an object, what properties and methods it should have. It doesn't contain any actual code or functionality.

Interfaces are used for type-checking and make code easier to understand and maintain. They can also be extended by other interfaces.

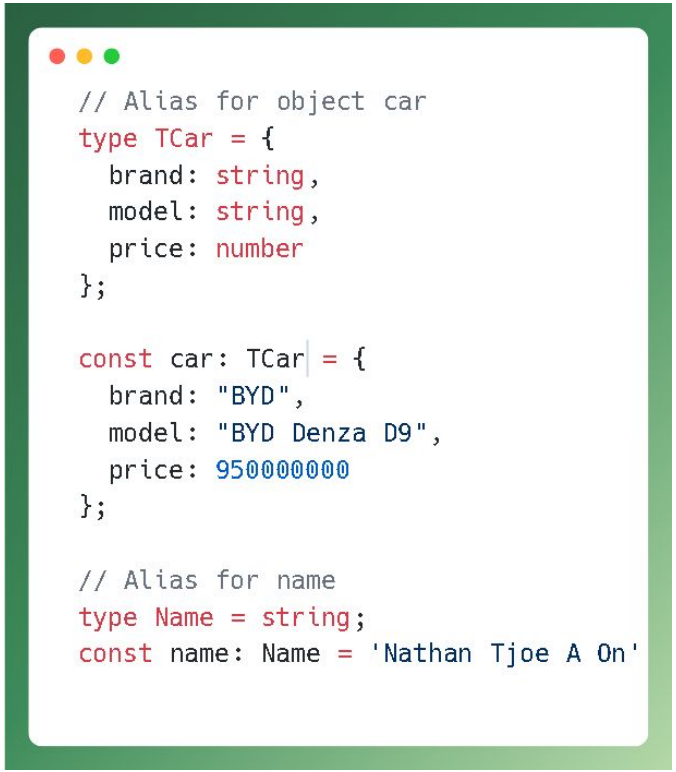


```
interface ICar{
  brand: string;
  model: string;
  price: number;
};

const car: ICar = {
  brand: "BYD",
  model: "BYD Denza D9",
  price: 950000000
};
```

TypeScript Types-Alias

A **type** alias in **TypeScript** lets you create a custom name for an existing type, making your code clearer and easier to maintain.

You can think of **types** like variables for types, they can be reused and defined within different scopes.



```typescript
// Alias for object car
type TCar = {
  brand: string,
  model: string,
  price: number
};

const car: TCar = {
  brand: "BYD",
  model: "BYD Denza D9",
  price: 950000000
};

// Alias for name
type Name = string;
const name: Name = 'Nathan Tjoe A On'
```

TypeScript Types-Alias vs Interface

Interface

- Defines the structure of an object — its properties and methods.
- Used mainly for object shapes and class contracts.
- Can be extended or merged (declaration merging).

Type Alias

- Creates a custom name for any type (primitive, union, tuple, or object).
- More flexible than interfaces.
- Cannot be merged, but can use intersections (&) to combine types.

What is Class ?

A **class** is a template for creating **objects**. It encapsulates data (properties) and the code (methods) that works on that data.

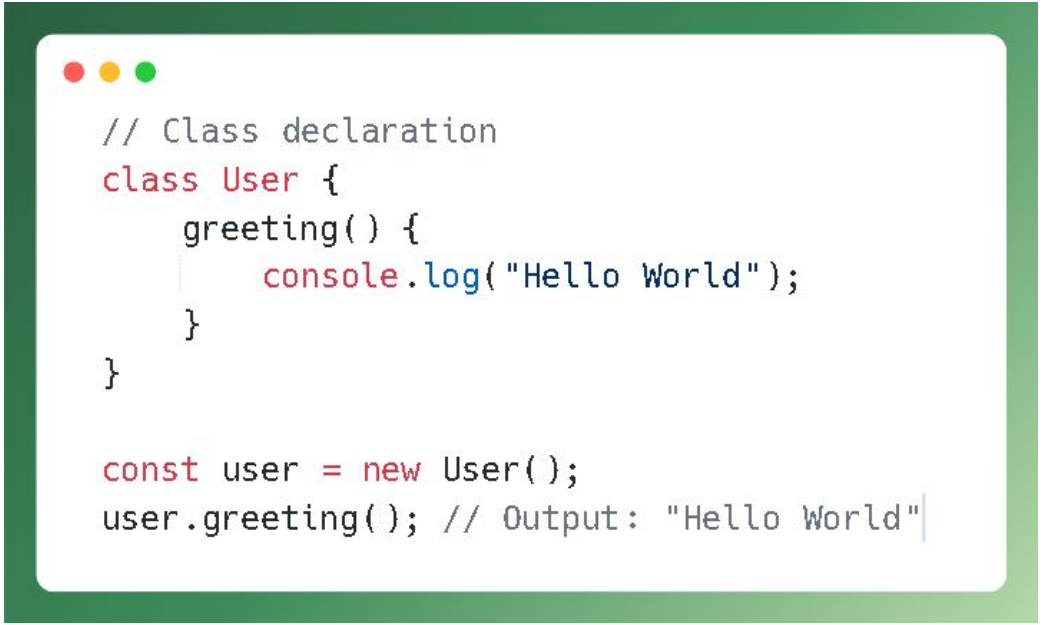
You can define them using class expressions and class declarations.

Classes make it easier to create objects with shared behavior, promoting cleaner, more structured code.

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

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

Create an Object from Class

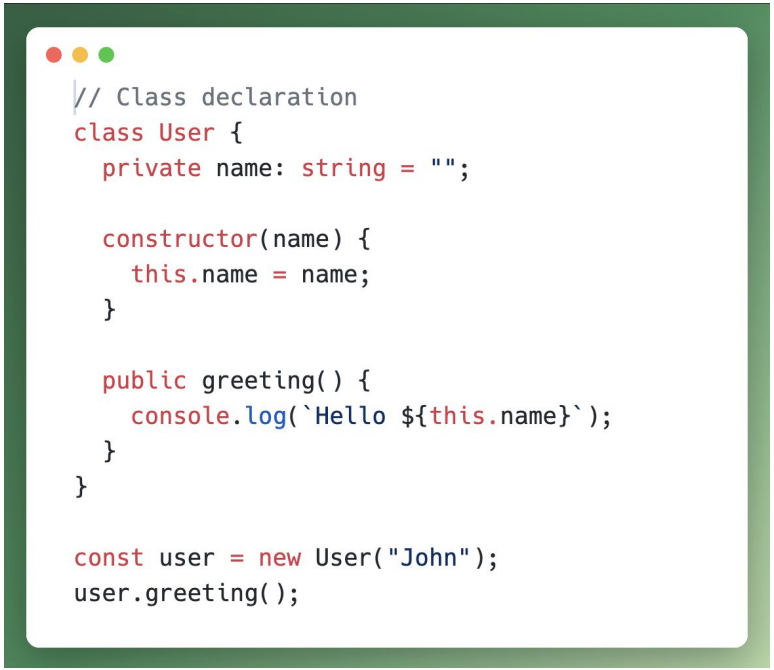


Constructor

The **constructor** is a special method used to create and initialize objects in a class.

Key Points:

- Called automatically when a class is instantiated.
- Must be named constructor exactly.
- If you don't define one, JavaScript adds a default empty constructor.
- A class can have only one constructor.



```
// Class declaration
class User {
  private name: string = "";

  constructor(name) {
    this.name = name;
  }

  public greeting() {
    console.log(`Hello ${this.name}`);
  }
}

const user = new User("John");
user.greeting();
```

Encapsulation

Encapsulation is the concept of hiding internal details and protecting data from unauthorized access.

In **TypeScript**, this is achieved using access modifiers:

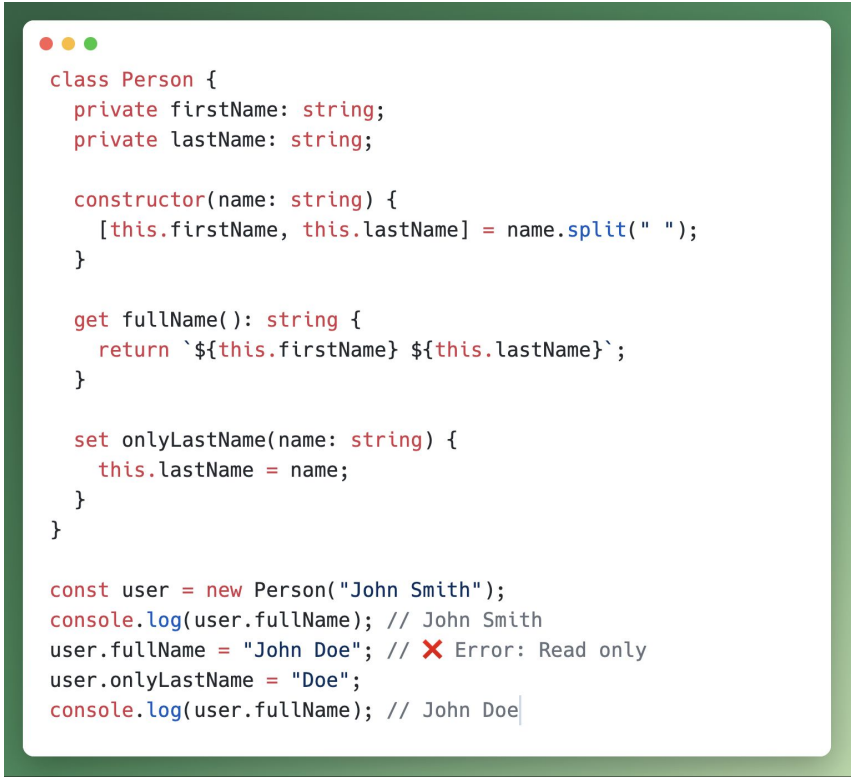
Modifier	Description
Public	Accessible from anywhere
Private	Accessible **only inside** the class
Protected	Accessible **inside the class** and its subclasses

Encapsulation

```
class BankAccount {
  private balance: number;

  constructor(initialBalance: number) {
    this.balance = initialBalance;
  }

  public deposit(amount: number): void {
    this.balance += amount;
  }

  public getBalance(): number {
    return this.balance;
  }
}

const account = new BankAccount(1000);
account.deposit(500);
console.log(account.getBalance()); // ✅ 1500
// console.log(account.balance); ❌ Error: 'balance' is private
```

Getter & Setter

Accessor properties let you **get** or **set** values in an object using special methods called **getters** and **setters**. They make it easier to control how data is read or changed inside an object.



```
class Person {
  private firstName: string;
  private lastName: string;

  constructor(name: string) {
    [this.firstName, this.lastName] = name.split(" ");
  }

  get fullName(): string {
    return `${this.firstName} ${this.lastName}`;
  }

  set onlyLastName(name: string) {
    this.lastName = name;
  }
}

const user = new Person("John Smith");
console.log(user.fullName); // John Smith
user.fullName = "John Doe"; // ❌ Error: Read only
user.onlyLastName = "Doe";
console.log(user.fullName); // John Doe
```

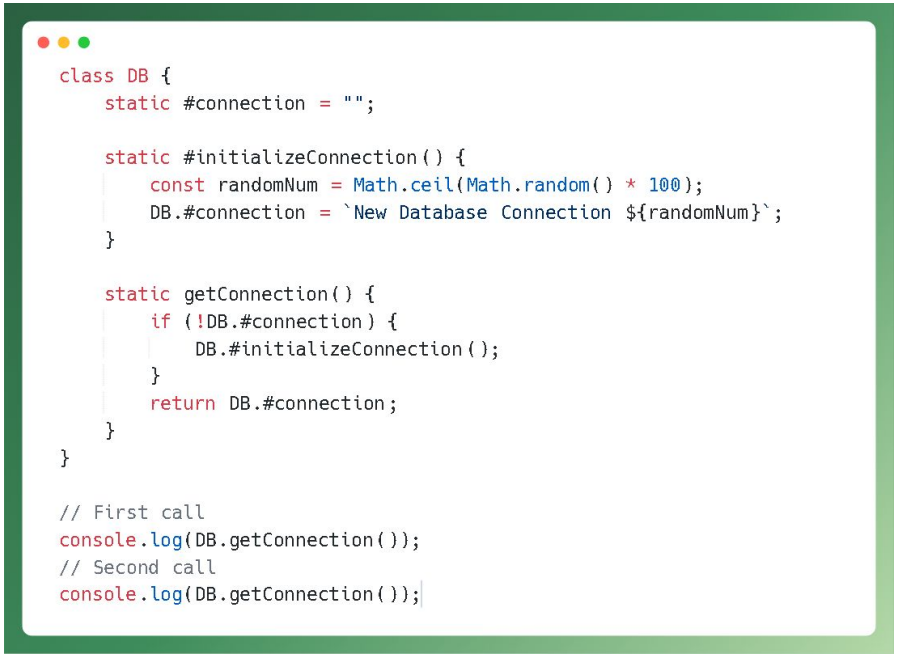
Advantages of Encapsulation

- **Data Hiding:** Hides internal implementation, users interact only through methods (getters/setters).
- **Increased Flexibility:** You can make data read-only (no setters) or write-only (no getters).
- **Reusability:** Encapsulated classes are modular and easy to modify for new requirements.
- **Easy Testing:** Encapsulated code is simpler to test and maintain.

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**.

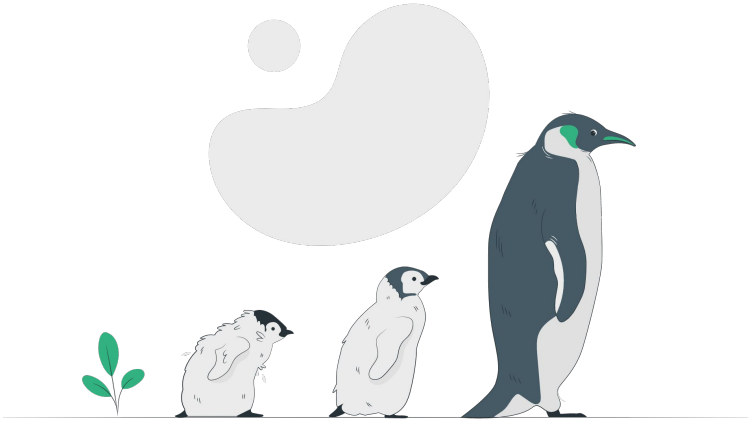


```
class DB {
    static #connection = "";

    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());
```

Inheritance

What is Inheritance?

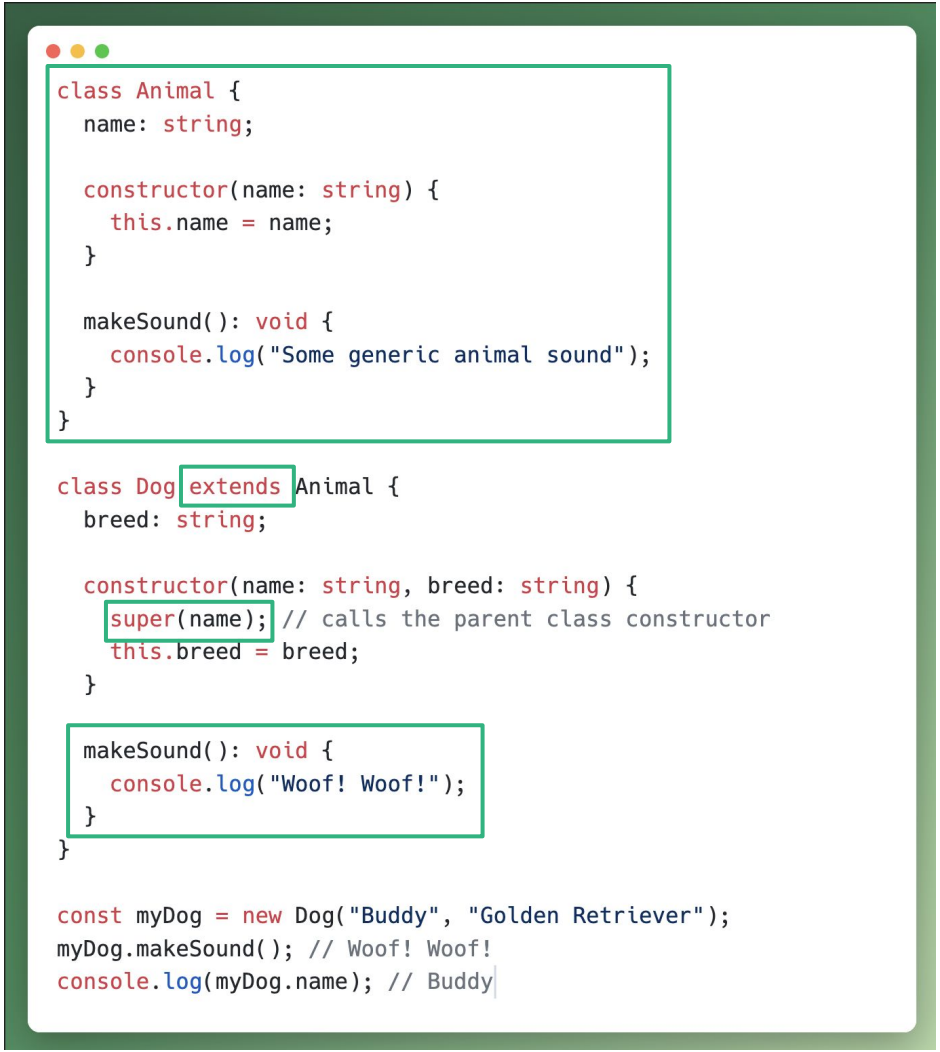
- Inheritance allows one class to use and extend the properties and methods of another class.
- It promotes code reusability and a hierarchical relationship between classes.



Inheritance

How It Works?

- **extends** keyword
  - The **Dog** class inherits all properties and methods from the **Animal** class.
- **super()** function
  - When you use extends, you must call **super()** inside the child's constructor before using this.
  - **super()** runs the parent class's constructor.
- Access to parent members
  - **Dog** automatically gets **name** and **makeSound()** from **Animal**.
- **Method overriding**
  - The **Dog** class redefines the **makeSound()** method with its own implementation.



```
class Animal {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  makeSound(): void {
    console.log("Some generic animal sound");
  }
}

class Dog extends Animal {
  breed: string;

  constructor(name: string, breed: string) {
    super(name); // calls the parent class constructor
    this.breed = breed;
  }

  makeSound(): void {
    console.log("Woof! Woof!");
  }
}

const myDog = new Dog("Buddy", "Golden Retriever");
myDog.makeSound(); // Woof! Woof!
console.log(myDog.name); // Buddy
```

“extends” Keyword

The keyword **extends** is used in classes to create inheritance, it allows one class (the child class or subclass) to inherit properties and methods from another class (the parent class or superclass).

Why Use extends?

- To reuse code from another class.
- To create a hierarchy of related classes (e.g., Animal ➜ Dog ➜ Puppy).
- To make your code more organized and maintainable.

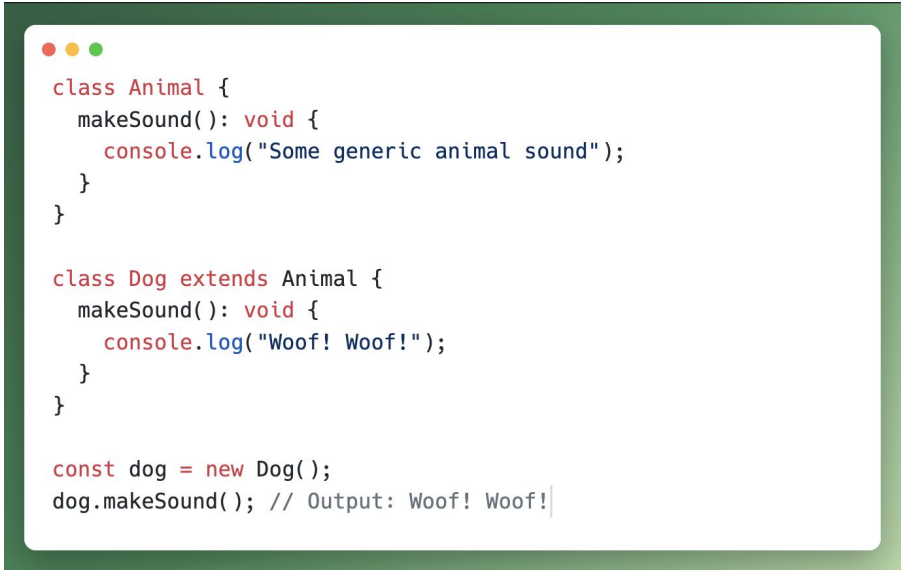
“super” Function

- The **super** Function 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.

Method Overriding

Method overriding happens when a **child class provides its own version of a method** that already exists in the parent class.

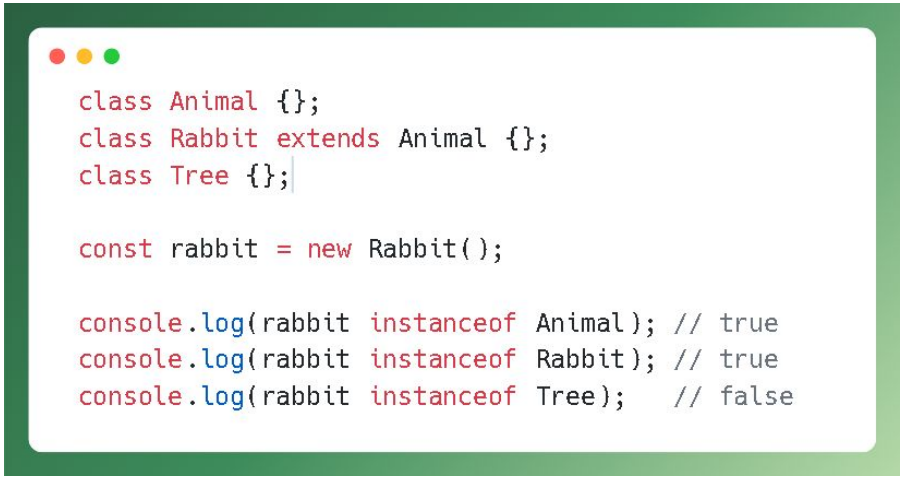
The new method **replaces (or overrides)** the inherited one when called on the child class.



```
class Animal {
  makeSound(): void {
    console.log("Some generic animal sound");
  }
}

class Dog extends Animal {
  makeSound(): void {
    console.log("Woof! Woof!");
  }
}

const dog = new Dog();
dog.makeSound(); // Output: Woof! Woof!
```

“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
```

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