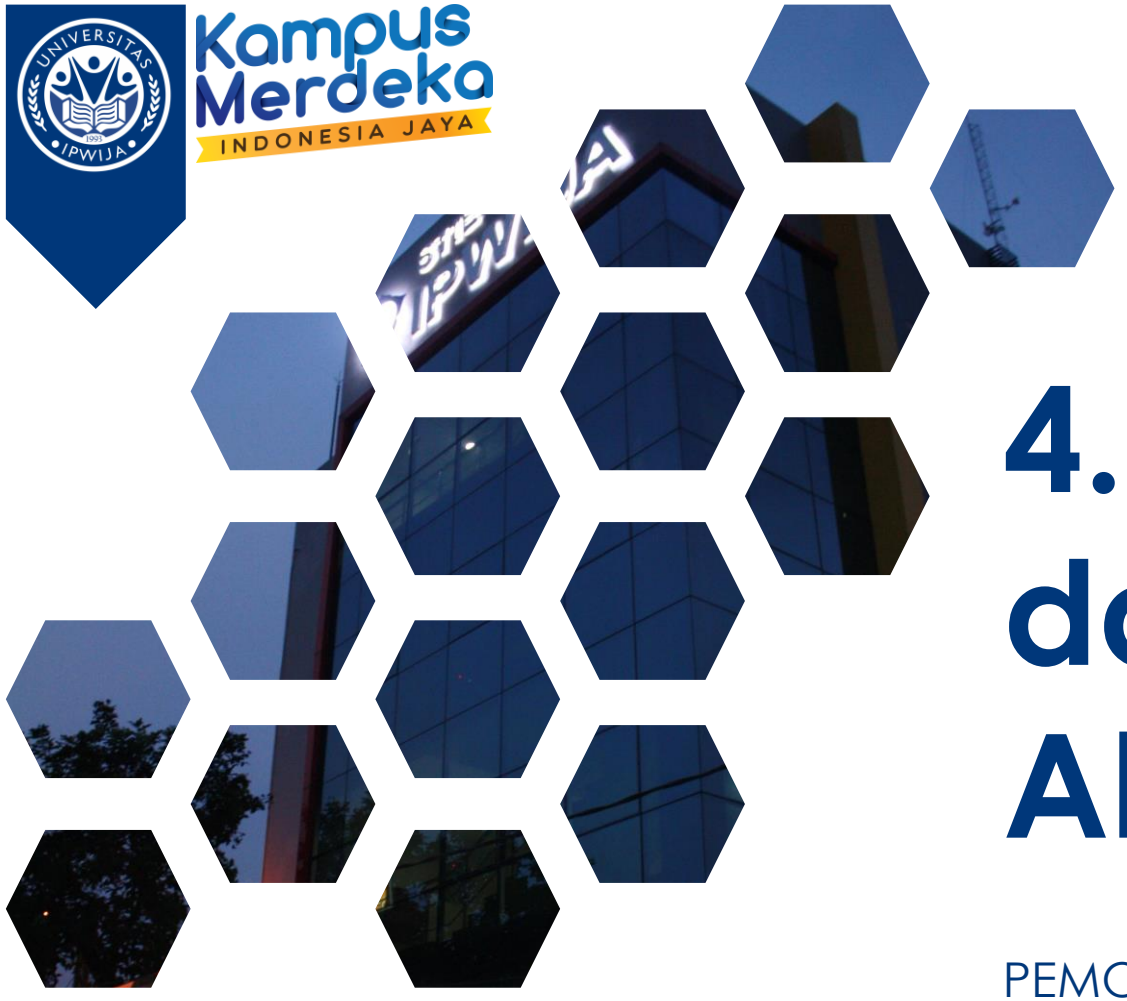


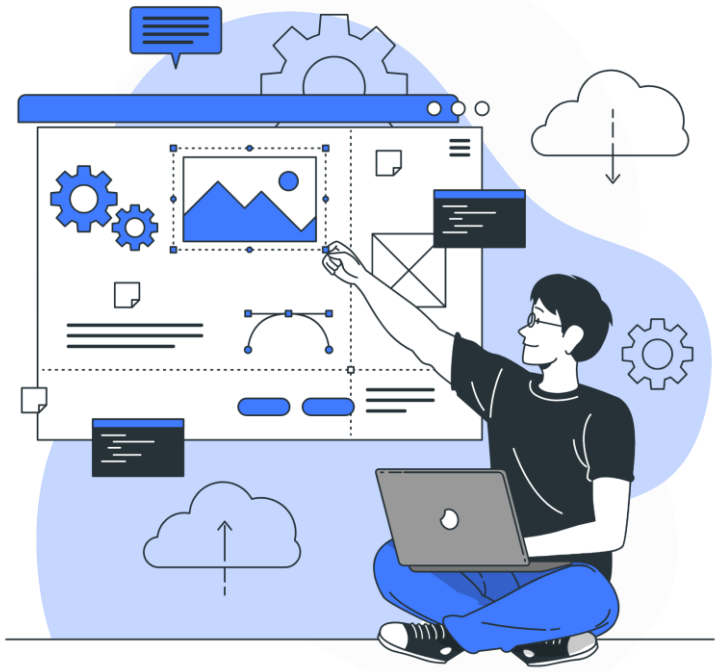
4. Enkapsulasi dan Pengelolaan Akses Data

PEMOGRAMAN BERORIENTASI OBJEK

1. Konsep Enkapsulasi dalam OOP

- Enkapsulasi adalah proses **menyembunyikan** detail implementasi suatu objek dan hanya memperlihatkan bagian yang **diperlukan** kepada pengguna. Dengan kata lain, kita bisa mengontrol data mana yang bisa diakses dari luar kelas dan mana yang tidak.
- Sebagai analogi, bayangkan sebuah ATM. Kita bisa menarik uang dengan memasukkan PIN, tapi kita tidak bisa melihat atau mengubah sistem internalnya secara langsung. Begitu juga dalam OOP, kita **membatasi** akses ke **properti** atau **metode** tertentu untuk mencegah kesalahan atau penyalahgunaan.
- Tanpa enkapsulasi, data bisa diakses dan dimodifikasi secara bebas, yang bisa menyebabkan inkonsistensi dan bug dalam program.

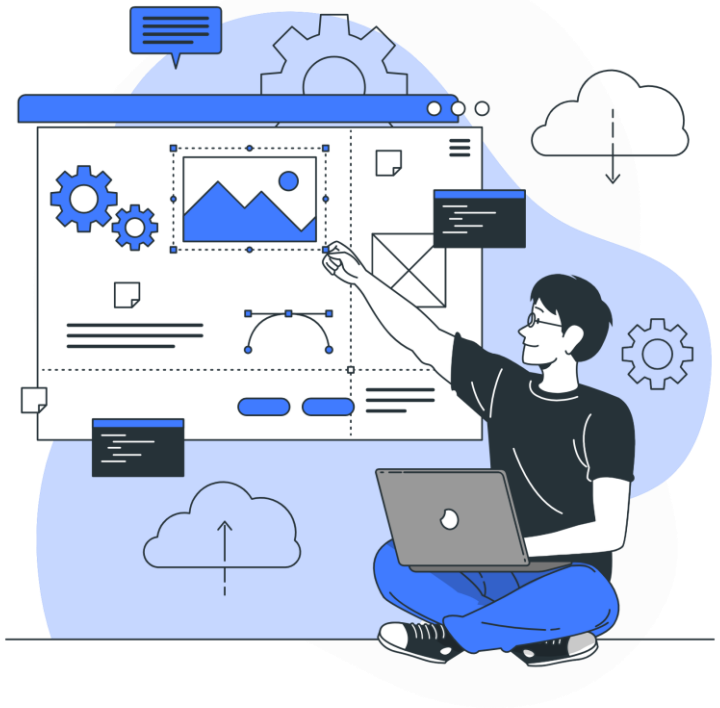
chrome



2. Modifier Akses

- Dalam OOP, kita menggunakan modifier akses untuk **mengontrol siapa** saja yang bisa mengakses **properti** atau **metode** dalam suatu **kelas**.

- Public,
- Private,
- Protected



2. Modifier Akses

a. Public

- Bisa diakses dari mana saja (dari dalam kelas, subclass, atau objek luar).
- Secara default, semua properti dalam JavaScript bersifat public.

```
class Mahasiswa {
    constructor(nama, nim) {
        this.nama = nama; // Public
        this.nim = nim;   // Public
    }
}

const mahasiswa1 = new Mahasiswa("Aldi", "2203123456");
console.log(mahasiswa1.nama); // Bisa diakses dari luar kelas
```

2. Modifier Akses

b. Private (#)

- Hanya bisa diakses dari dalam kelas itu sendiri.
- Ditandai dengan # sebelum nama variabel (sejak ES6).

Kenapa butuh private?

- Mencegah perubahan data secara langsung yang bisa merusak logika program.
- Memastikan keamanan data, terutama untuk aplikasi keuangan, database, atau autentikasi.

```
class BankAccount {
    #saldo; // Private
    constructor(nama, saldoAwal) {
        this.nama = nama;
        this.#saldo = saldoAwal;
    }
    getSaldo() {
        return `Saldo ${this.nama} adalah Rp${this.#saldo}`;
    }
}
const akun = new BankAccount("Alice", 500000);
console.log(akun.getSaldo()); // Bisa diakses melalui method
console.log(akun.#saldo); // ERROR! Tidak bisa diakses langsung
```

2. Modifier Akses

c. Protected (Simulasi dengan _ - Konvensi JavaScript)

- Bisa diakses oleh subclass (kelas turunan), tapi tidak boleh diakses dari luar kelas.
- JavaScript tidak punya keyword protected secara eksplisit, tapi kita bisa menggunakan konvensi dengan _namaProperti untuk menandakan bahwa properti ini "sebaiknya" tidak diakses langsung dari luar kelas.

```
class Hewan {
    constructor(nama) {
        this._nama = nama; // Konvensi protected
    }
    suara() {
        return `${this._nama} mengeluarkan suara...`;
    }
}
class Kucing extends Hewan {
    suara() {
        return `${this._nama} mengeong!`;
    }
}
const milo = new Kucing("Milo");
console.log(milo.suara()); // Milo mengeong!
console.log(milo._nama); // Bisa diakses, tapi SEBAIKNYA tidak
```

2. Modifier Akses

Kenapa tidak ada protected di JavaScript?

JavaScript berbeda dari bahasa seperti Java atau C++ yang memiliki protected.

Biasanya, untuk akses protected, kita menggunakan _namaProperti sebagai konvensi, meskipun tetap bisa diakses dari luar.

3. Getter dan Setter dalam JavaScript

Kadang kita ingin mengontrol bagaimana sebuah properti diakses atau diubah, dan di sinilah peran getter dan setter.

- Getter (get) digunakan untuk **mengambil** nilai dari properti private.
- Setter (set) digunakan untuk **mengubah** nilai properti dengan validasi tertentu.

```
class Mobil {
    #kecepatan = 0; // Private property
    get kecepatan() {
        return `${this.#kecepatan} km/jam`;
    }
    set kecepatan(value) {
        if (value < 0) {
            console.log("Kecepatan tidak bisa negatif!");
        } else {
            this.#kecepatan = value;
        }
    }
}
const avanza = new Mobil();
console.log(avanza.kecepatan); // Output: 0 km/jam
avanza.kecepatan = 100; // Menggunakan setter
console.log(avanza.kecepatan); // Output: 100 km/jam
avanza.kecepatan = -10; // Output: Kecepatan tidak bisa negatif!
```

3. Getter dan Setter dalam JavaScript

Kenapa perlu getter dan setter?

- Memastikan data tetap valid (misalnya kecepatan tidak bisa negatif).
- Mengontrol akses ke data private tanpa mengeksposnya secara langsung.
- Meningkatkan **fleksibilitas**, karena kita bisa menambahkan logika tambahan tanpa mengubah cara akses data.

4. Manfaat Enkapsulasi dalam Pengembangan Perangkat Lunak

Enkapsulasi bukan hanya konsep teoretis, tetapi juga sangat bermanfaat dalam proyek nyata. Beberapa manfaatnya:

- **Keamanan Data**

Dengan menyembunyikan data private, kita bisa mencegah perubahan yang tidak disengaja atau tidak sah.

- **Modularitas & Reusability**

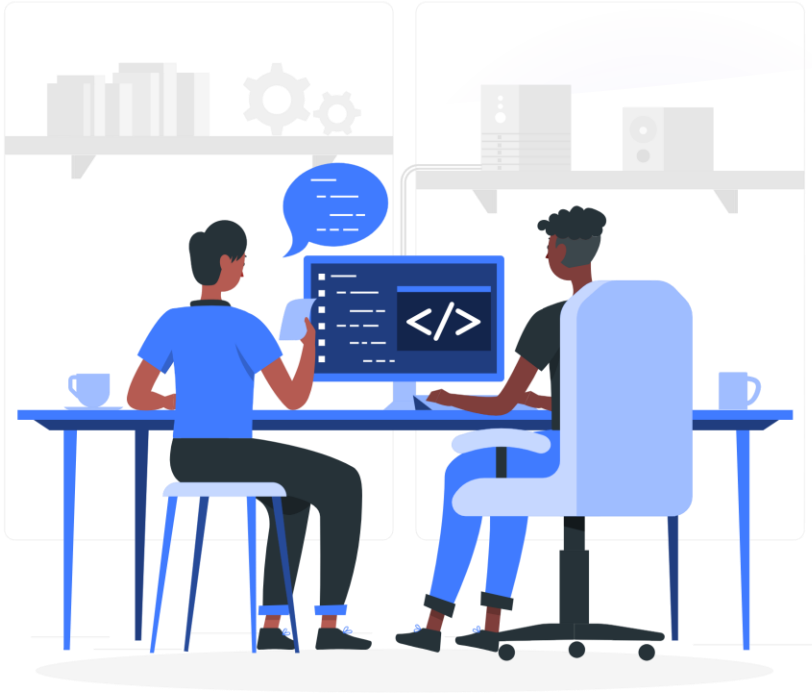
Kode lebih terorganisir, sehingga lebih mudah digunakan ulang tanpa mengubah banyak bagian lain dalam sistem.

- **Mencegah Bug & Error**

Dengan membatasi akses ke data, kita bisa mencegah bug akibat perubahan langsung pada properti internal objek.

- **Meningkatkan Maintainability**

Jika suatu saat kita ingin mengubah cara kerja sebuah kelas, kita cukup mengupdate getter/setter tanpa harus mengubah semua kode yang mengaksesnya.

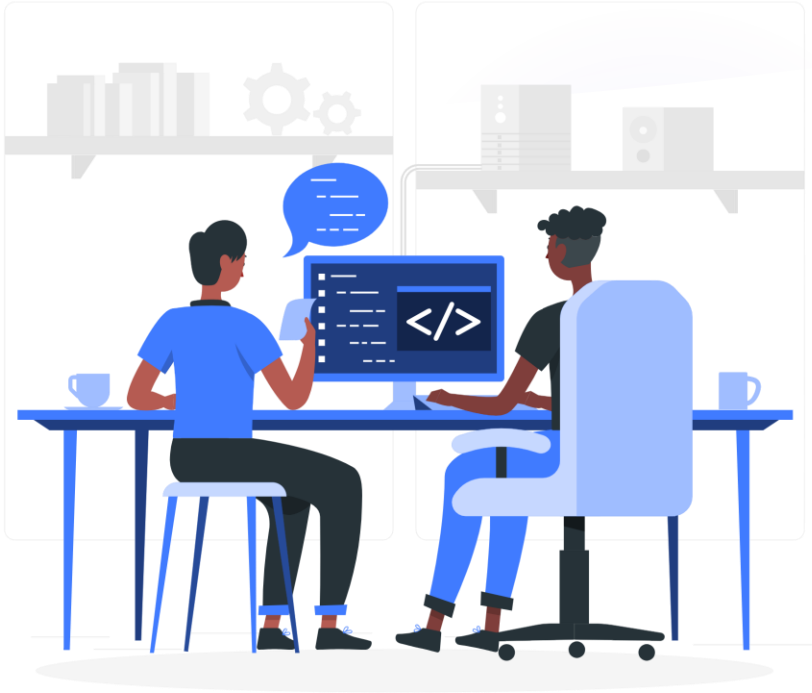


Kesimpulan

Dengan enkapsulasi, kita bisa membuat program yang lebih aman, modular, dan mudah dikelola.

Dalam JavaScript, kita bisa mengimplementasikannya dengan

- private properties (#),
- getter dan setter, serta
- modifier akses seperti **public** dan **protected** (konvensi).



Praktikum:

- Membuat layout web sederhana