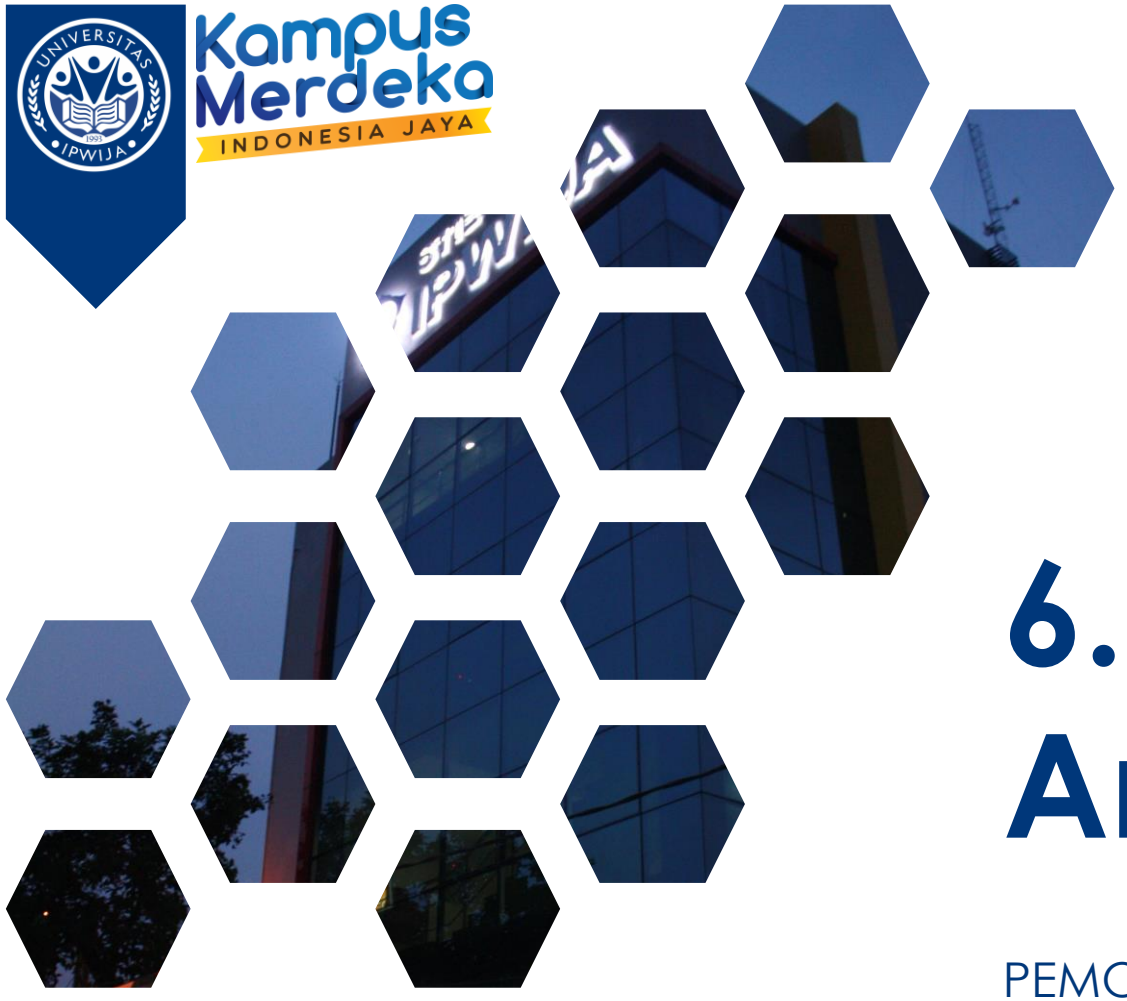
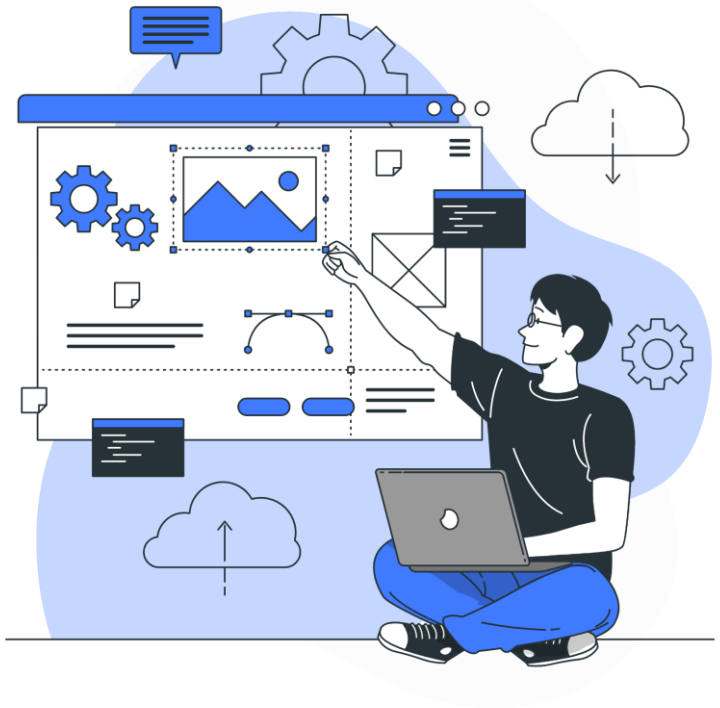


6. Abstraksi dan Antarmuka

PEMOGRAMAN BERORIENTASI OBJEK

1. Pengertian Abstraksi dalam OOP

- Abstraksi dalam OOP adalah proses **menyembunyikan detail** implementasi dan hanya menunjukkan fitur yang relevan kepada pengguna.



Analogi

Bayangkan kalian menggunakan smartphone. Untuk melakukan panggilan, kalian hanya menekan nomor dan menekan tombol "Call", tanpa perlu tahu bagaimana sinyal dikirim ke menara BTS atau bagaimana data suara dikodekan dan didekodekan.

Dalam pemrograman, abstraksi memungkinkan kita fokus pada **apa** yang dilakukan sebuah objek, **bukan bagaimana** cara kerjanya.

Perhatikan bahwa method maju() tidak memiliki implementasi di sini.

```
class Kendaraan {
    constructor(merk) {
        this.merk = merk;
    }

    // Method abstrak (harus diimplementasikan oleh subclass)
    maju() {
        throw new Error("Method 'maju()' harus diimplementasikan!");
    }
}
```

2. Penggunaan Abstract Class dalam JavaScript

Dalam bahasa seperti Java atau C#, ada fitur abstract class, tapi JavaScript tidak memiliki fitur ini secara langsung.

Namun, kita bisa meniru konsep abstraksi dengan membuat kelas induk yang memiliki metode tanpa implementasi, lalu subclass harus mengimplementasikan metode tersebut.

```
class Kendaraan {
    constructor(merk) {
        this.merk = merk;
        if (this.constructor === Kendaraan) {
            throw new Error("Kelas abstrak 'Kendaraan' tidak bisa diinstansiasi langsung.");
        }
    }

    maju() {
        throw new Error("Method 'maju()' harus diimplementasikan!");
    }
}
```

Penjelasan

Kelas Kendaraan bertindak sebagai abstract class.

Jika kita mencoba membuat objek dari Kendaraan langsung, JavaScript akan memberikan error.

Kelas Mobil harus mengimplementasikan method maju() agar bisa digunakan.

```
class Mobil extends Kendaraan {
    maju() {
        return `${this.merk} melaju dengan kecepatan tinggi!`;
    }
}

const avanza = new Mobil("Toyota Avanza");
console.log(avanza.maju()); // Toyota Avanza melaju dengan kecepatan tinggi!

// const kendaraanBaru = new Kendaraan("Generic"); // Error: Kelas abstrak tidak bisa diinstansiasi
```

3. Konsep Interface dan Implementasi dalam JavaScript

Dalam OOP, interface adalah kontrak yang **menentukan metode** apa yang harus dimiliki oleh sebuah kelas, tanpa memberikan implementasi spesifiknya.

```
// Interface sebagai objek blueprint
const kendaraanInterface = {
    maju: function () {
        throw new Error("Method 'maju()' harus diimplementasikan!");
    }
};

class Sepeda {
    constructor(merk) {
        this.merk = merk;
    }

    maju() {
        return `${this.merk} melaju dengan tenaga manusia!`;
    }
}
```

Analogi

Bayangkan colokan listrik. Meskipun bentuknya standar, perangkat yang berbeda bisa menggunakannya sesuai kebutuhannya masing-masing.

JavaScript tidak memiliki fitur bawaan interface, tapi kita bisa mensimulasikannya dengan pendekatan berikut

Dalam JavaScript, kita bisa menggunakan interface dalam bentuk objek blueprint atau class tanpa implementasi metode.

```
// Menguji apakah Sepeda mematuhi interface
const polygon = new Sepeda("Polygon");
console.log(polygon.maju()); // Polygon melaju dengan tenaga manusia!


if (typeof polygon.maju !== "function") {
    throw new Error("Class Sepeda harus mengimplementasikan 'maju()'!");
}
```

4. Studi Kasus Abstraksi dalam Pengembangan Perangkat Lunak

Mari kita lihat bagaimana abstraksi diterapkan dalam aplikasi nyata. Misalkan kita membuat aplikasi e-commerce yang memiliki berbagai jenis pembayaran, seperti Kartu Kredit, PayPal, dan QRIS.

Tanpa abstraksi, kita harus menangani setiap metode pembayaran dalam satu kelas, yang bisa membuat kode kita berantakan.

Dengan abstraksi, kita bisa membuat kelas Pembayaran sebagai abstrak, lalu setiap metode pembayaran mengimplementasikan cara kerjanya sendiri.

```
class Pembayaran {
    constructor(jumlah) {
        this.jumlah = jumlah;
        if (this.constructor === Pembayaran) {
            throw new Error("Kelas abstrak 'Pembayaran' tidak bisa diinstansiasi langsung.");
        }
    }

    prosesPembayaran() {
        throw new Error("Method 'prosesPembayaran()' harus diimplementasikan!");
    }
}
```

Keuntungan

Kode lebih rapi karena setiap metode pembayaran punya kelas sendiri.

Fleksibel jika di masa depan kita ingin menambah metode pembayaran lain, cukup buat subclass baru tanpa mengubah kode yang sudah ada.

```
class KartuKredit extends Pembayaran {
    prosesPembayaran() {
        return `Pembayaran ${this.jumlah} melalui Kartu Kredit berhasil!`;
    } }
class PayPal extends Pembayaran {
    prosesPembayaran() {
        return `Pembayaran ${this.jumlah} melalui PayPal berhasil!`;
    } }
// Implementasi
const bayar1 = new KartuKredit(500000);
console.log(bayar1.prosesPembayaran()); // Pembayaran 500000 melalui Kartu Kredit berhasil!

const bayar2 = new PayPal(250000);
console.log(bayar2.prosesPembayaran()); // Pembayaran 250000 melalui PayPal berhasil!
```

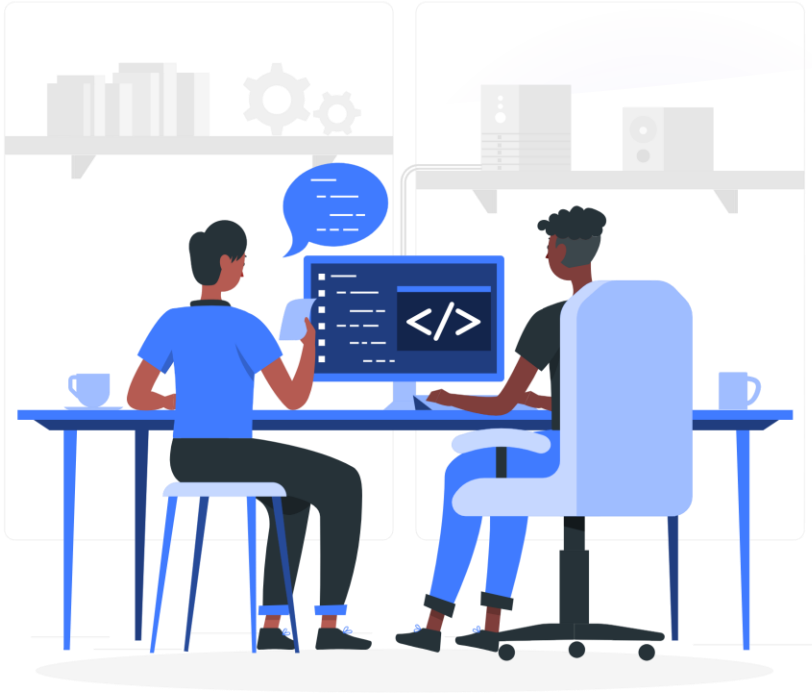
Kesimpulan

Abstraksi **menyembunyikan** detail implementasi dan hanya menunjukkan fitur penting.

Abstract class tidak bisa diinstansiasi langsung dan berfungsi sebagai **blueprint** untuk subclass.

Interface adalah kontrak yang **memastikan** kelas tertentu memiliki metode yang diharapkan.

Dalam pengembangan perangkat lunak, abstraksi membantu membuat kode lebih **terstruktur**, mudah **dipelihara**, dan **scalable**.



Praktikum:

- Gunakan Github untuk menjalankan kode abstraksi