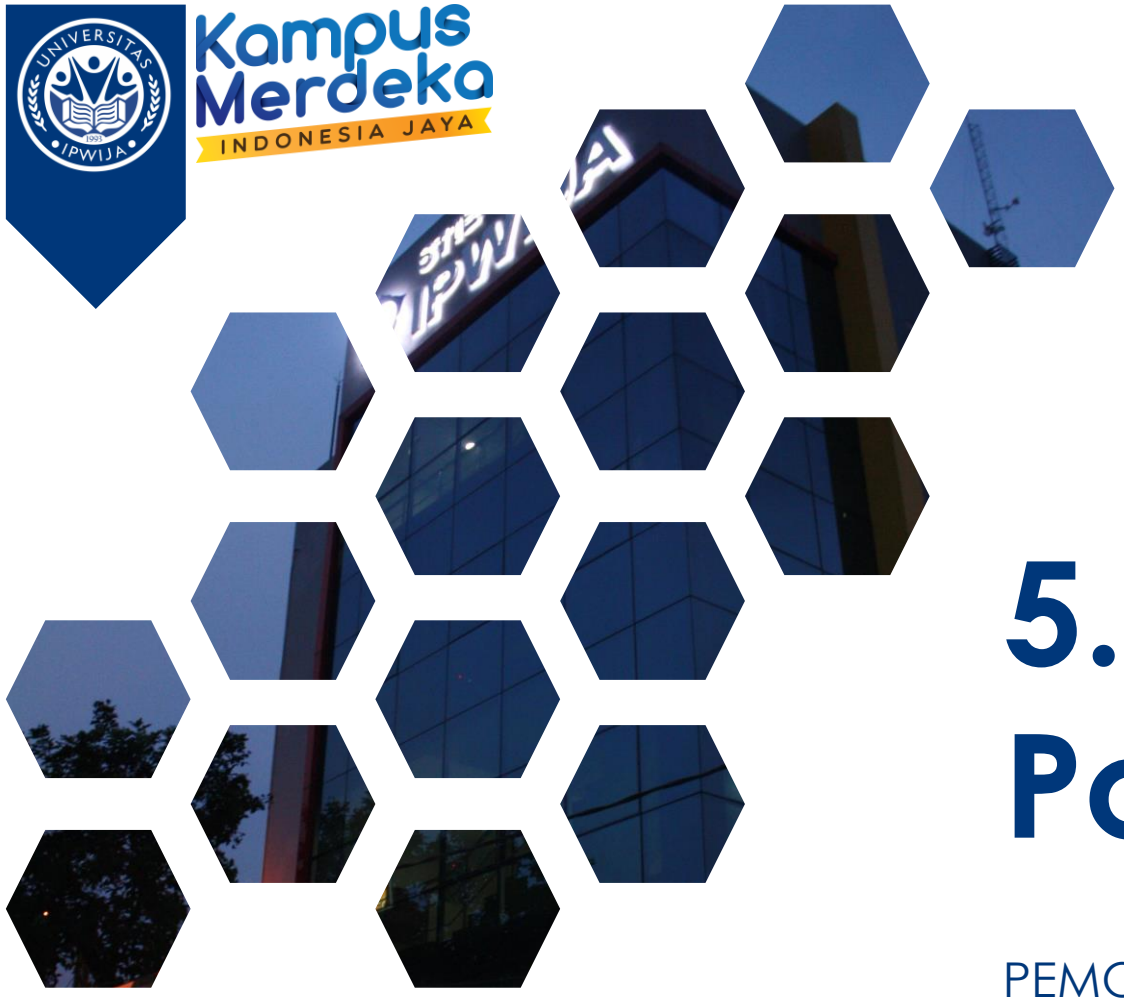
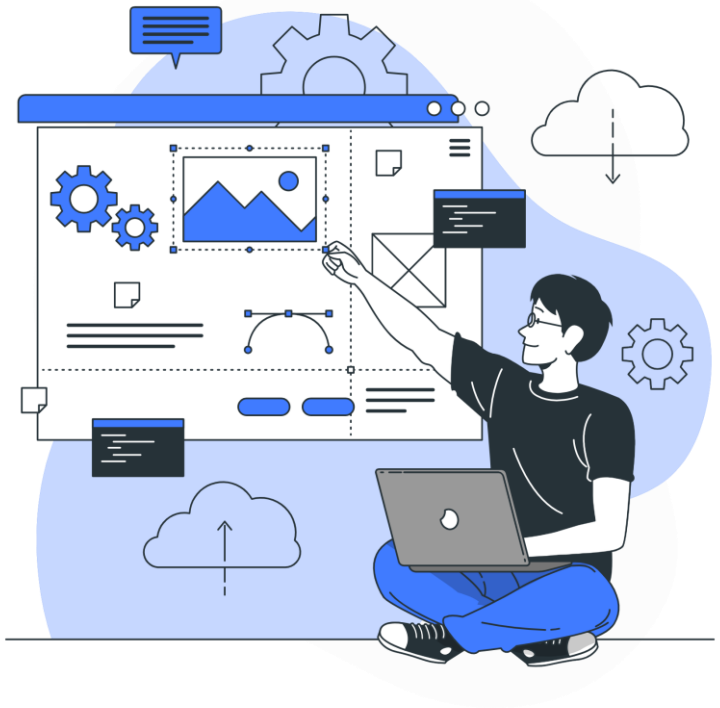


5. Pewarisan dan Polimorfisme

PEMOGRAMAN BERORIENTASI OBJEK

1. Konsep Pewarisan dalam OOP

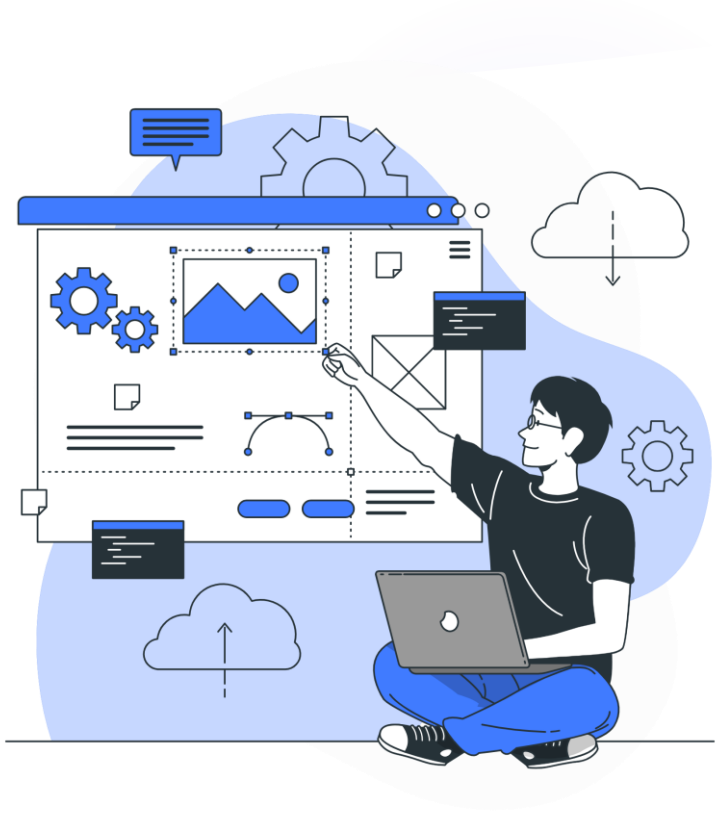
- Pewarisan atau inheritance adalah fitur dalam OOP yang memungkinkan sebuah kelas untuk mewarisi **properti** dan **metode** dari kelas lain. Dengan pewarisan, kita bisa membuat kelas yang lebih **spesifik** tanpa perlu **menulis ulang** kode yang sudah ada.
- Misalnya, kita punya kelas "Hewan", lalu kita ingin membuat kelas "Kucing" dan "Anjing". Daripada menulis ulang properti seperti nama dan jenis, kita cukup mewarisi dari kelas Hewan.



Analogi

Bayangkan kalian membuat karakter dalam sebuah **game**. Semua karakter memiliki atribut dasar seperti nama dan level, tapi beberapa karakter punya skill spesifik, misalnya Mage bisa menggunakan sihir, sementara Warrior memiliki pertahanan lebih kuat.

Daripada mendefinisikan ulang semua atribut, kita cukup membuat kelas dasar "Karakter", lalu kelas "Mage" dan "Warrior" bisa mewarisinya.



2. Pewarisan dalam JavaScript Menggunakan extends

JavaScript mendukung pewarisan dengan kata kunci **extends**.

Saat sebuah kelas menggunakan extends, kelas tersebut akan mewarisi semua properti dan metode dari kelas induknya.

Contoh pewarisan dalam JavaScript:

```
// Kelas induk (superclass)
class Hewan {
    constructor(nama, jenis) {
        this.nama = nama;
        this.jenis = jenis;
    }

    suara() {
        return `${this.nama} bersuara...`;
    }
}
```

Penjelasan:

Kelas Hewan adalah kelas induk (superclass).

Kelas Kucing adalah kelas turunan (subclass) yang mewarisi semua properti dan metode dari Hewan.

super(nama, "Kucing") digunakan untuk memanggil constructor dari kelas induk agar nama dan jenis bisa diinisialisasi dengan benar.

Tanpa super(), JavaScript akan memberikan error karena constructor di subclass harus memanggil constructor superclass terlebih dahulu.

```
// Kelas turunan (subclass)


class Kucing extends Hewan {
    constructor(nama, warna) {
        super(nama, "Kucing"); // Memanggil constructor superclass
        this.warna = warna;
    }
}
const milo = new Kucing("Milo", "Putih");
console.log(milo.nama);  // Milo
console.log(milo.jenis); // Kucing (Didapat dari superclass)
console.log(milo.suara()); // Milo bersuara...
```

3. Overriding Method dan super Keyword

Terkadang, kita ingin mengganti **perilaku** dari **metode** yang diwarisi. Dalam OOP, ini disebut method overriding.

Kita bisa mengganti metode di subclass dengan cara mendefinisikan ulang metode yang sama di dalam subclass.

Jika ingin tetap menggunakan metode dari kelas induk, kita bisa menggunakan super.method().

```
class Anjing extends Hewan {
    constructor(nama, warna) {
        super(nama, "Anjing");
        this.warna = warna;
    }

    // Overriding method
    suara() {
        return `${this.nama} menggonggong: Woof woof!`;
    }
}

const doge = new Anjing("Doge", "Coklat");
console.log(doge.suara()); // Doge menggonggong: Woof woof!
```

Penjelasan

Kelas Anjing mengubah metode suara() dari kelas Hewan.

Jika kita memanggil doge.suara(), hasilnya berbeda dari superclass.

Overriding digunakan saat subclass ingin memiliki perilaku berbeda dari superclass-nya.

Jika kita masih ingin menggunakan metode superclass dalam metode yang telah di-override, kita bisa menggunakan super.suara().

```
class Serigala extends Hewan {
    constructor(nama, warna) {
        super(nama, "Serigala");
        this.warna = warna;
    }

    suara() {
        return super.suara() + " Auuuuu~";
    }
}

const alpha = new Serigala("Alpha", "Abu-abu");
console.log(alpha.suara()); // Alpha bersuara... Auuuuu~
```

4. Konsep Polimorfisme dan Implementasinya

Polimorfisme berasal dari bahasa Yunani yang berarti "**banyak bentuk**".

Dalam OOP, polimorfisme memungkinkan subclass menggunakan **metode yang sama** dengan **perilaku berbeda**.

Misalnya, semua hewan punya metode suara(), tapi cara mereka bersuara berbeda. Dengan polimorfisme, kita bisa menggunakan metode yang sama di semua kelas turunan, tetapi dengan implementasi berbeda.

```
class Burung extends Hewan {
    constructor(nama) {
        super(nama, "Burung");
    }
    suara() {
        return `${this.nama} berkicau: Cip cip!`;
    }
}

const hewanList = [
    new Kucing("Milo", "Putih"),
    new Anjing("Buddy", "Hitam"),
    new Burung("Tweety")
];
hewanList.forEach(hewan =>
console.log(hewan.suara()));
```

Kenapa polimorfisme penting?

Mempermudah pengelolaan kode

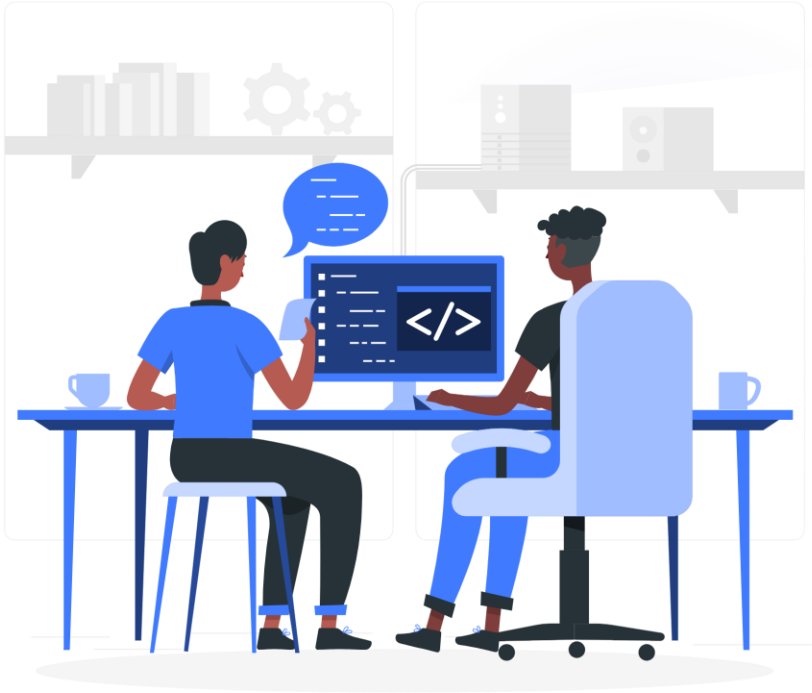
Kita bisa menggunakan metode yang sama tanpa harus mengetahui tipe objeknya.

Kode lebih fleksibel

Jika ada tambahan kelas baru, kita tidak perlu mengubah struktur kode yang sudah ada.

Dukungan untuk prinsip SOLID

Polimorfisme mendukung Open/Closed Principle, yang berarti kita bisa menambahkan fitur baru tanpa mengubah kode lama.



SOLID pada OOP

S – Single Responsibility Principle (SRP)

Setiap kelas hanya boleh punya satu alasan untuk berubah, artinya satu kelas hanya menangani satu tanggung jawab.

O – Open/Closed Principle (OCP)

Sistem harus terbuka untuk ekstensi, tapi tertutup untuk modifikasi. Artinya, kita bisa menambahkan fitur baru tanpa harus mengubah kode lama.

L – Liskov Substitution Principle (LSP)

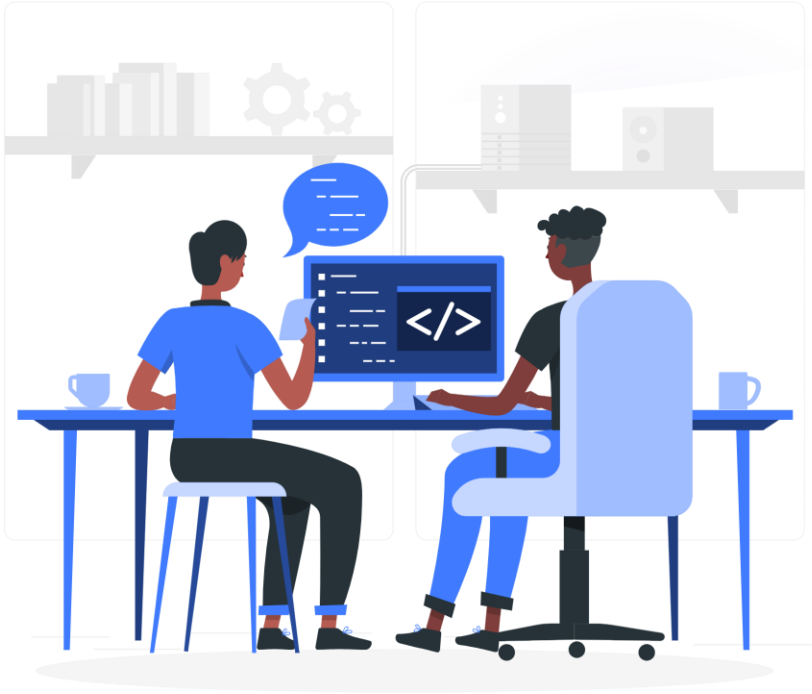
Objek dari subclass harus bisa menggantikan objek superclass-nya tanpa mengubah keakuratan program.

I – Interface Segregation Principle (ISP)

Lebih baik banyak antarmuka khusus daripada satu antarmuka umum yang besar.

D – Dependency Inversion Principle (DIP)

Modul tingkat tinggi tidak boleh bergantung langsung pada modul tingkat rendah. Keduanya harus bergantung pada abstraksi.



Kesimpulan

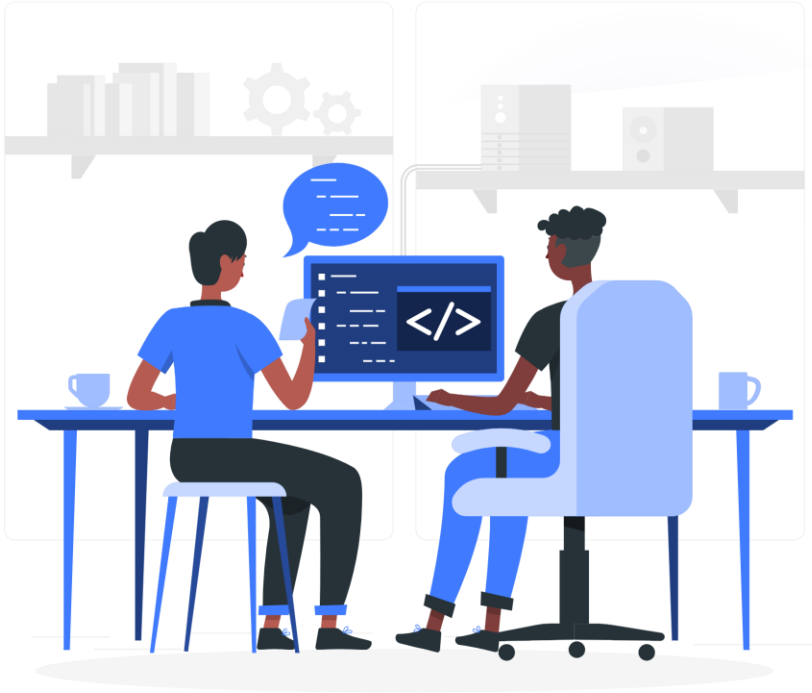
Pewarisan (Inheritance) memungkinkan kita membuat subclass yang **mewarisi properti** dan **metode** dari superclass.

Menggunakan **extends** dalam JavaScript memungkinkan kita untuk membuat hierarki kelas dengan kode yang lebih reusable.

Method **overriding** memungkinkan subclass untuk mengubah perilaku metode yang diwarisi dari superclass.

Polimorfisme memungkinkan metode yang sama memiliki **perilaku berbeda** di setiap subclass, membuat kode lebih fleksibel dan modular.

Konsep-konsep ini sangat berguna dalam pengembangan perangkat lunak modern, karena mempermudah pemeliharaan kode dan memungkinkan kita membangun sistem yang lebih **skalabel**.



Praktikum:

- Gunakan Github untuk menjalankan kode enkapsulasi