



Selamat datang di modul mata kuliah Pemrograman Mobile! Modul ini akan memandu Anda dalam mempelajari bahasa pemrograman Dart secara mendalam, yang merupakan fondasi penting sebelum Anda terjun ke pengembangan aplikasi mobile menggunakan Flutter. Setiap pertemuan akan dilengkapi dengan penjelasan konsep, materi, contoh kode, dan studi kasus yang dirancang agar mudah dimengerti dan dipahami.

## Daftar Isi Modul

- Pertemuan 1: Pengenalan Pemrograman Mobile dan Dart
- Pertemuan 2: Dasar-dasar Sintaks Dart
- Pertemuan 3: Variabel dan Tipe Data
- Pertemuan 4: Operator dan Input/Output Konsol
- Pertemuan 5: Control Flow (Percabangan)
- Pertemuan 6: Control Flow (Perulangan)
- Pertemuan 7: Fungi
- Pertemuan 8: Koleksi (List, Set, Map)
- Pertemuan 9: Pemrograman Berorientasi Objek (OOP) - Kelas dan Objek
- Pertemuan 10: Pemrograman Berorientasi Objek (OOP) - Inheritance dan Polymorphism
- Pertemuan 11: Pemrograman Berorientasi Objek (OOP) - Interfaces dan Mixins
- Pertemuan 12: Asynchronous Programming - Future dan Async/Await
- Pertemuan 13: Asynchronous Programming - Stream dan Error Handling
- Pertemuan 14: Manajemen Paket dengan Pub dan Persiapan Flutter

Berdasarkan Referensi Umum

- Dart Official Website
- Flutter Official Website
- Pub.dev - Dart & Flutter Packages
- DartPad - Online Dart Editor

## Pertemuan 1: Pengenalan Pemrograman Mobile dan Dart

### Tujuan Pembelajaran

Setelah mempelajari materi ini, mahasiswa diharapkan mampu:

- Memahami gambaran umum tentang pemrograman mobile.
- Menjelaskan peran bahasa pemrograman Dart dalam pengembangan aplikasi mobile.
- Menyiapkan lingkungan pengembangan Dart di komputer.
- Menulis dan menjalankan program Dart pertama.

### 1.1 Apa itu Pemrograman Mobile?

Pemrograman mobile adalah proses pembuatan perangkat lunak aplikasi yang berjalan pada perangkat mobile seperti smartphone dan tablet. Aplikasi mobile dapat dikategorikan menjadi beberapa jenis utama:

- **Aplikasi Native:** Dibangun khusus untuk satu platform (misalnya, Android menggunakan Java/Kotlin, iOS menggunakan Swift/Objective-C). Aplikasi native menawarkan performa terbaik dan akses penuh ke fitur perangkat, tetapi memerlukan codebase terpisah untuk setiap platform.
- **Aplikasi Hybrid:** Dibangun menggunakan teknologi web (HTML, CSS, JavaScript) dan dibungkus dalam kontainer native. Contoh framework: Apache Cordova, Ionic. Keuntungannya adalah satu codebase untuk banyak platform, tetapi performa dan akses fitur perangkat mungkin terbatas dibandingkan native.
- **Aplikasi Cross-Platform:** Dibangun menggunakan satu bahasa pemrograman dan framework yang memungkinkan kompilasi ke kode native untuk beberapa platform. Contoh framework: React Native, Xamarin, dan tentu saja, **Flutter** dengan Dart. Ini menawarkan keseimbangan antara performa native dan efisiensi satu codebase.

### 1.2 Pengenalan Dart: Sejarah, Fitur Utama, dan Mengapa Dart untuk Mobile

Dart adalah bahasa pemrograman *client-optimized* yang dikembangkan oleh Google. Awalnya dirancang sebagai alternatif JavaScript untuk pengembangan web, Dart kini menjadi bahasa utama untuk membangun aplikasi multi-platform yang cepat dan indah dengan Flutter.

#### Sejarah Singkat Dart

Dart pertama kali diperkenalkan oleh Google pada tahun 2011. Sejak awal, tujuannya adalah untuk menciptakan bahasa yang mudah dipelajari, memiliki performa tinggi, dan cocok untuk pengembangan aplikasi berskala besar. Dengan munculnya Flutter, popularitas Dart meroket karena menjadi bahasa pilihan untuk framework UI yang inovatif tersebut.

## Fitur Utama Dart

- **Client-Optimized:** Dirancang khusus untuk pengembangan aplikasi client-side, baik itu web, mobile, maupun desktop.
- **Ahead-of-Time (AOT) Compilation:** Dart dapat dikompilasi menjadi kode native, menghasilkan aplikasi dengan performa startup yang sangat cepat dan *frame rate* yang halus.
- **Just-in-Time (JIT) Compilation:** Selama pengembangan, Dart menggunakan kompilasi JIT yang memungkinkan fitur *Hot Reload* dan *Hot Restart* di Flutter, mempercepat proses pengembangan.
- **Type-Safe:** Dart adalah bahasa yang *type-safe* dengan *null safety* bawaan, membantu mencegah banyak bug umum terkait null pointer pada saat kompilasi.
- **Object-Oriented:** Mendukung semua konsep OOP seperti kelas, objek, inheritance, polymorphism, dan mixins.
- **Asynchronous Programming:** Memiliki dukungan bawaan untuk pemrograman asynchronous menggunakan `Future`, `async`, dan `await`, yang penting untuk operasi I/O dan UI yang responsif.
- **Ekosistem yang Kaya:** Memiliki manajer paket `pub` dengan ribuan paket yang siap digunakan, serta alat pengembangan yang kuat.

## Mengapa Dart untuk Mobile (khususnya Flutter)?

Dart adalah pilihan ideal untuk pengembangan mobile dengan Flutter karena beberapa alasan:

1. **Performa Tinggi:** Kompilasi AOT ke kode native memastikan aplikasi Flutter berjalan sangat cepat dan lancar, memberikan pengalaman pengguna yang mirip dengan aplikasi native.
2. **Produktivitas Pengembang:** Fitur seperti *Hot Reload* dan *Hot Restart* di Flutter, yang dimungkinkan oleh kompilasi JIT Dart, memungkinkan pengembang melihat perubahan kode secara instan tanpa harus me-restart aplikasi. Ini sangat mempercepat siklus pengembangan.
3. **Sintaks yang Familiar:** Sintaks Dart mirip dengan bahasa populer lainnya seperti Java, C#, dan JavaScript, membuatnya relatif mudah dipelajari bagi pengembang yang sudah memiliki latar belakang tersebut.
4. **Null Safety:** Fitur *null safety* membantu pengembang menulis kode yang lebih aman dan bebas dari kesalahan runtime yang disebabkan oleh nilai null.
5. **Ekosistem Terintegrasi:** Dart dan Flutter dikembangkan oleh Google, memastikan integrasi yang mulus dan dukungan yang kuat dari satu sumber.

## 1.3 Persiapan Lingkungan Pengembangan

Untuk memulai ngoding dengan Dart, Anda perlu menyiapkan beberapa hal:

### 1.3.1 Instalasi Dart SDK

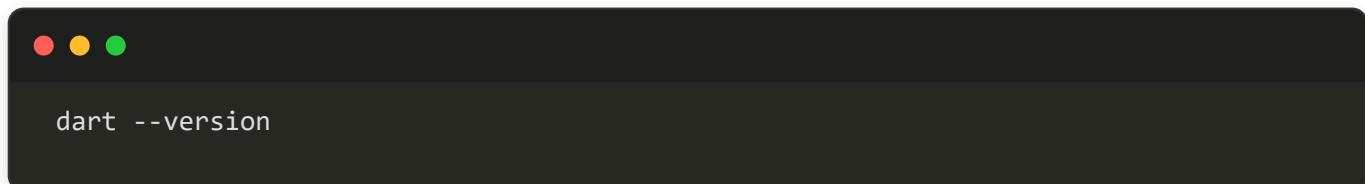
Dart SDK (Software Development Kit) adalah kumpulan alat yang Anda butuhkan untuk mengembangkan aplikasi Dart. Anda bisa mengunduhnya dari situs resmi Dart.

**Langkah-langkah Instalasi (Umum):**

1. Kunjungi situs resmi Dart: <https://dart.dev/get-dart>
2. Pilih sistem operasi Anda (Windows, macOS, Linux).
3. Ikuti instruksi instalasi yang diberikan. Biasanya melibatkan pengunduhan installer atau penggunaan package manager (seperti `brew` untuk macOS, `apt` untuk Linux).

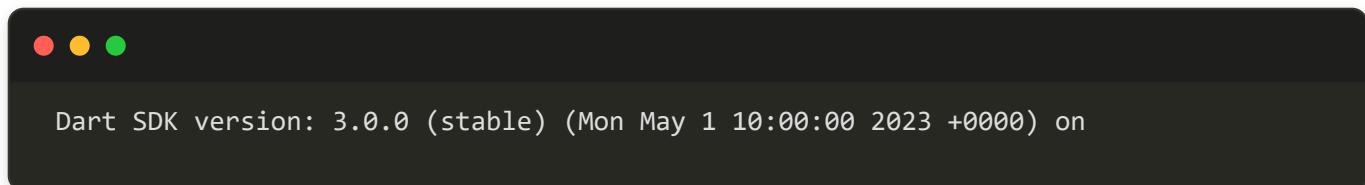
**Verifikasi Instalasi:**

Setelah instalasi selesai, buka **Terminal** (macOS/Linux) atau **Command Prompt/PowerShell** (Windows) dan ketik perintah berikut:



```
dart --version
```

Jika instalasi berhasil, Anda akan melihat output yang menunjukkan versi Dart yang terinstal, contohnya:



```
Dart SDK version: 3.0.0 (stable) (Mon May 1 10:00:00 2023 +0000) on
```

Jika Anda mendapatkan pesan error, periksa kembali langkah-langkah instalasi atau cari solusi di dokumentasi Dart.

**1.3.2 Editor (VS Code)**

Visual Studio Code (VS Code) adalah editor kode yang sangat populer dan direkomendasikan untuk pengembangan Dart dan Flutter. Jika Anda belum memiliki, unduh dari <https://code.visualstudio.com/>.

Setelah VS Code terinstal, Anda perlu menginstal ekstensi Dart dan Flutter untuk mendapatkan fitur-fitur seperti *syntax highlighting*, *code completion*, *debugging*, dan *Hot Reload*.

**Langkah-langkah Instalasi Ekstensi:**

1. Buka VS Code.
2. Pergi ke bagian Extensions (ikon kotak di sidebar kiri atau tekan `Ctrl+Shift+X`).
3. Cari "Dart" dan "Flutter".
4. Instal kedua ekstensi tersebut.

### 1.3.3 Menjalankan Program Dart Pertama (Hello World)

Sekarang, mari kita buat program Dart pertama Anda.

- Buat Folder Proyek:** Buat folder baru di komputer Anda, misalnya `my_dart_project`.
- Buka di VS Code:** Buka folder tersebut di VS Code (`File > Open Folder...`).
- Buat File Dart:** Di dalam VS Code, buat file baru bernama `hello_world.dart` di dalam folder `my_dart_project`.
- Tulis Kode:** Salin kode berikut ke dalam `hello_world.dart`:

```
void main() {  
    print('Halo, Dunia!');  
}
```

*Penjelasan Kode:*

- `void main()`: Ini adalah fungsi utama (entry point) dari setiap program Dart. Kode di dalam fungsi ini akan dieksekusi pertama kali.
- `print('Halo, Dunia!');`: Ini adalah perintah untuk menampilkan teks "Halo, Dunia!" ke konsol. Teks diapit oleh tanda kutip tunggal (`'`) atau ganda (`"`) dan diakhiri dengan titik koma (`;`).

- Jalankan Program:** Ada beberapa cara untuk menjalankan program Dart:

- **Dari VS Code:** Klik kanan pada file `hello_world.dart` di Explorer VS Code, lalu pilih "Run Without Debugging" atau "Run Dart File". Atau, klik tombol "Run" di pojok kanan atas editor.
- **Dari Terminal:** Buka terminal di VS Code (`Terminal > New Terminal`) atau terminal eksternal, navigasikan ke folder `my_dart_project`, lalu jalankan perintah:

```
dart run hello_world.dart
```

Anda akan melihat output `Halo, Dunia!` di konsol.

## Studi Kasus Sederhana: Menampilkan Informasi Diri

Mari kita buat program Dart sederhana yang menampilkan informasi diri Anda.

```
void main() {  
    String nama = "Abdul Yamin";  
    int umur = 30;  
    String pekerjaan = "Dosen";  
  
    print("Nama saya: " + nama);  
    print("Umur saya: " + umur.toString() + " tahun");  
    print("Pekerjaan saya: " + pekerjaan);  
    print("\nSalam kenal!");  
}
```

### Penjelasan:

- Kita mendeklarasikan tiga variabel (`nama`, `umur`, `pekerjaan`) dengan tipe data yang sesuai.
- `umur.toString()` digunakan untuk mengubah nilai integer `umur` menjadi string agar bisa digabungkan dengan string lain menggunakan operator `+`.
- `\n` adalah karakter khusus untuk membuat baris baru (newline).

### Latihan:

1. Ubah informasi diri di kode di atas dengan data Anda sendiri.
2. Tambahkan informasi lain seperti hobi atau alamat.
3. Coba gunakan `print()` dengan interpolasi string (menggunakan `$`) seperti ini:  
`print('Nama saya: $nama');`

### • Referensi:

- [dart.dev/overview](https://dart.dev/overview)
- [dart.dev/get-dart](https://dart.dev/get-dart)

## Pertemuan 2: Dasar-dasar Sintaks Dart

### Tujuan Pembelajaran

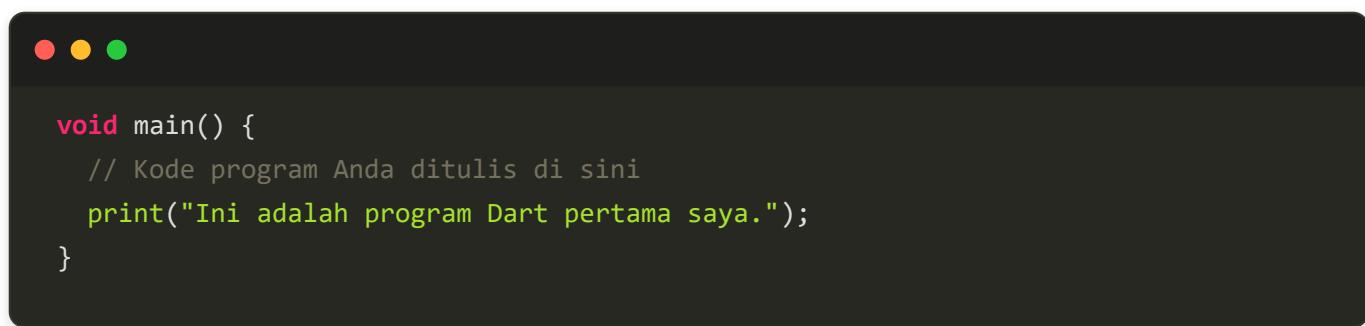
Setelah mempelajari materi ini, mahasiswa diharapkan mampu:

- Memahami struktur dasar sebuah program Dart.
- Menulis berbagai jenis komentar dalam kode Dart.
- Memahami peran dan penggunaan fungsi `main()`.
- Membedakan antara pernyataan (statements) dan ekspresi (expressions) dalam Dart.

### 2.1 Struktur Dasar Program Dart

Setiap program Dart, sekecil apapun, memiliki struktur dasar yang harus diikuti. Program Dart dieksekusi dari atas ke bawah, dimulai dari sebuah fungsi khusus bernama `main()`.

Berikut adalah struktur paling sederhana dari sebuah program Dart:



```
void main() {  
    // Kode program Anda ditulis di sini  
    print("Ini adalah program Dart pertama saya.");  
}
```

- `void`: Kata kunci ini menunjukkan bahwa fungsi `main()` tidak mengembalikan nilai apapun. Jika fungsi mengembalikan nilai, kita akan mengganti `void` dengan tipe data nilai yang dikembalikan (misalnya `int`, `String`, dll).
- `main()`: Ini adalah fungsi utama dan wajib ada di setiap aplikasi Dart. Ketika Anda menjalankan program Dart, eksekusi selalu dimulai dari fungsi `main()`.
- `{}`: Kurung kurawal ini mendefinisikan blok kode. Semua kode yang merupakan bagian dari fungsi `main()` harus berada di dalam kurung kurawal ini.
- `;`: Titik koma digunakan untuk mengakhiri setiap pernyataan (statement) dalam Dart. Ini mirip dengan bahasa pemrograman seperti C++, Java, atau JavaScript.

### 2.2 Komentar

Komentar adalah bagian dari kode yang diabaikan oleh kompiler Dart. Komentar digunakan untuk memberikan penjelasan atau catatan dalam kode, yang sangat berguna untuk:

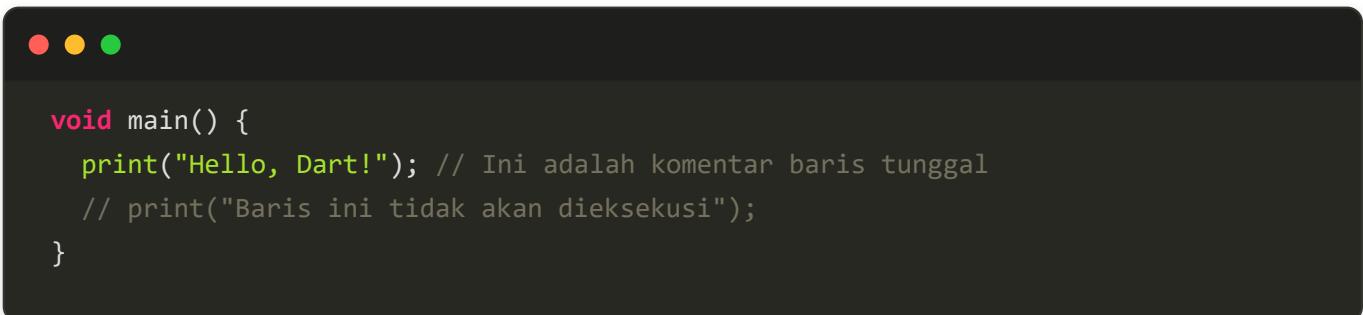
- Menjelaskan tujuan atau logika di balik bagian kode tertentu.

- Membuat kode lebih mudah dibaca dan dipahami oleh orang lain (dan diri Anda sendiri di masa depan).
- Menonaktifkan sementara bagian kode tanpa menghapusnya.

Dart mendukung tiga jenis komentar:

### 2.2.1 Komentar Baris Tunggal (Single-line Comments)

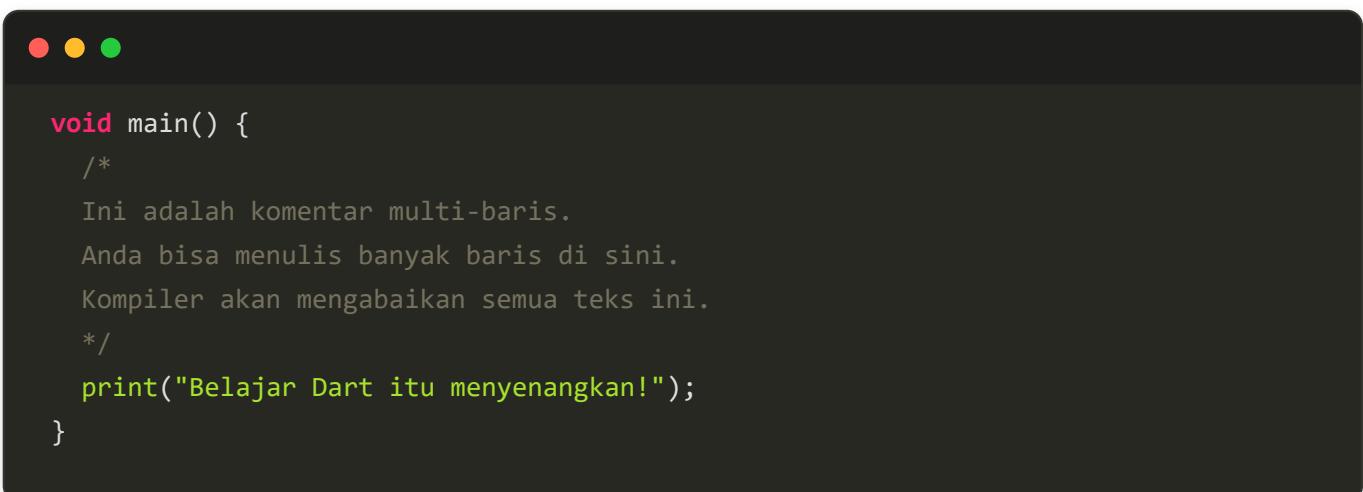
Dimulai dengan dua garis miring (`//`). Semua teks setelah `//` hingga akhir baris akan dianggap sebagai komentar.



```
void main() {
    print("Hello, Dart!"); // Ini adalah komentar baris tunggal
    // print("Baris ini tidak akan dieksekusi");
}
```

### 2.2.2 Komentar Multi-baris (Multi-line Comments)

Dimulai dengan `/*` dan diakhiri dengan `*/`. Semua teks di antara tanda ini, termasuk yang melintasi beberapa baris, akan dianggap sebagai komentar.



```
void main() {
/*
Ini adalah komentar multi-baris.
Anda bisa menulis banyak baris di sini.
Kompiler akan mengabaikan semua teks ini.
*/
print("Belajar Dart itu menyenangkan!");
}
```

### 2.2.3 Komentar Dokumentasi (Documentation Comments)

Dimulai dengan tiga garis miring (`///`) atau `/** ... */`. Komentar ini digunakan untuk menghasilkan dokumentasi API secara otomatis. Alat dokumentasi Dart (seperti `dart doc`) akan memproses komentar ini untuk membuat halaman dokumentasi yang informatif.

```
/// Fungsi ini akan mencetak pesan salam ke konsol.  
/// Pesan salam default adalah "Halo, Dunia!".  
void main() {  
    print("Halo, Dunia!");  
}  
  
/**  
 * Kelas ini merepresentasikan sebuah mobil.  
 * Ini memiliki properti untuk merek, model, dan tahun.  
 */  
class Mobil {  
    String merek;  
    String model;  
    int tahun;  
  
    Mobil(this.merek, this.model, this.tahun);  
}
```

### 2.3 Fungsi `main()`

Seperti yang sudah disinggung di atas, fungsi `main()` adalah jantung dari setiap program Dart. Ini adalah tempat di mana eksekusi program dimulai. Tanpa fungsi `main()`, program Dart tidak akan bisa dijalankan.

Beberapa hal penting tentang `main()` :

- **Wajib Ada:** Setiap aplikasi Dart harus memiliki fungsi `main()`.
- **Titik Masuk:** Kompiler Dart akan mencari fungsi ini sebagai titik awal eksekusi.
- `void` : Umumnya, `main()` dideklarasikan dengan `void` karena tidak mengembalikan nilai. Namun, `main()` juga bisa mengembalikan `int` jika Anda ingin mengindikasikan kode keluar (exit code) dari program (misalnya, `0` untuk sukses, nilai lain untuk error).
- **Parameter Opsional** `List<String> args` : Fungsi `main()` dapat menerima argumen baris perintah (command-line arguments) melalui parameter opsional `List<String> args`. Ini berguna jika Anda ingin program Anda menerima input saat dijalankan dari terminal.

Contoh `main()` dengan argumen:

```
void main(List<String> arguments) {  
    print("Argumen yang diterima: $arguments");
```

```
if (arguments.isNotEmpty) {  
    print("Argumen pertama: ${arguments[0]}");  
}  
}
```

Untuk menjalankan program ini dengan argumen dari terminal:

```
dart run nama_file.dart arg1 arg2 "argumen dengan spasi"
```

Outputnya akan seperti ini:

```
Argumen yang diterima: [arg1, arg2, argumen dengan spasi]  
Argumen pertama: arg1
```

## 2.4 Pernyataan (Statements) dan Ekspresi (Expressions)

Dalam pemrograman, penting untuk memahami perbedaan antara pernyataan dan ekspresi.

### 2.4.1 Pernyataan (Statements)

Pernyataan adalah instruksi yang melakukan suatu tindakan. Mereka tidak selalu menghasilkan nilai. Setiap pernyataan di Dart diakhiri dengan titik koma ( ; ).

Contoh pernyataan:

```
void main() {  
    int angka = 10; // Pernyataan deklarasi dan inisialisasi variabel  
    print(angka); // Pernyataan pemanggilan fungsi  
    if (angka > 5) { // Pernyataan kontrol alur (if statement)  
        print("Angka lebih besar dari 5");  
    }  
}
```

## 2.4.2 Ekspresi (Expressions)

Ekspresi adalah bagian dari kode yang menghasilkan sebuah nilai. Ekspresi dapat digunakan sebagai bagian dari pernyataan.

Contoh ekspresi:

```
void main() {  
    int a = 5;  
    int b = 3;  
  
    int hasilTambah = a + b; // `a + b` adalah ekspresi yang menghasilkan nilai 8  
    print(hasilTambah); // `hasilTambah` adalah ekspresi yang menghasilkan nilai 8  
  
    bool isGenap = (10 % 2 == 0); // `10 % 2 == 0` adalah ekspresi yang menghasilkan nilai true  
    print(isGenap);  
  
    String pesan = "Halo" + " Dunia!"; // `"Halo" + " Dunia!"` adalah ekspresi yang menghasilkan nilai pesan  
    print(pesan);  
}
```

### Perbedaan Kunci:

- **Pernyataan:** Melakukan sesuatu, diakhiri dengan `;`.
- **Ekspresi:** Menghasilkan nilai, dapat menjadi bagian dari pernyataan.

### Studi Kasus Sederhana: Menghitung Luas Persegi Panjang

Mari kita terapkan konsep sintaks dasar, komentar, dan fungsi `main()` untuk menghitung luas persegi panjang.

```
void main() {  
    // Deklarasi variabel untuk panjang dan lebar  
    double panjang = 10.5; // Contoh panjang dalam satuan meter  
    double lebar = 5.0; // Contoh lebar dalam satuan meter  
  
    // Menghitung luas persegi panjang  
    // Rumus: Luas = Panjang * Lebar  
    double luas = panjang * lebar; // Ini adalah ekspresi yang hasilnya disimpan di variabel luas
```

```
// Menampilkan hasil ke konsol
print("Panjang: $panjang meter");
print("Lebar: $lebar meter");
print("Luas persegi panjang adalah: $luas meter persegi");
}
```

**Penjelasan:**

- Kita menggunakan komentar baris tunggal (`//`) untuk menjelaskan setiap bagian kode.
- Fungsi `main()` adalah titik awal eksekusi.
- `panjang * lebar` adalah sebuah ekspresi yang menghasilkan nilai luas.
- `print()` adalah pernyataan yang menampilkan hasil.

**Latihan:**

1. Ubah program di atas untuk menghitung keliling persegi panjang. (Rumus: Keliling =  $2 * (\text{Panjang} + \text{Lebar})$ )
2. Tambahkan komentar multi-baris di awal program untuk menjelaskan tujuan program secara keseluruhan.
3. Coba gunakan argumen baris perintah untuk memasukkan nilai panjang dan lebar, lalu hitung luasnya.

**• Referensi:**

- [dart.dev/language/main](https://dart.dev/language/main)
- [dart.dev/language/comments](https://dart.dev/language/comments)

## Pertemuan 3: Variabel dan Tipe Data

### Tujuan Pembelajaran

Setelah mempelajari materi ini, mahasiswa diharapkan mampu:

- Memahami konsep variabel dan pentingnya tipe data dalam pemrograman.
- Mendeklarasikan variabel menggunakan `var`, `final`, dan `const`.
- Mengenali dan menggunakan tipe data dasar di Dart: `int`, `double`, `String`, `bool`.
- Memahami konsep `dynamic` dan `Object`.
- Mengimplementasikan Null Safety (`?`, `!`, `late`) untuk menulis kode yang lebih aman.

### 3.1 Konsep Variabel dan Tipe Data

Dalam pemrograman, **variabel** adalah nama yang diberikan untuk lokasi memori yang digunakan untuk menyimpan data. Bayangkan variabel sebagai sebuah kotak atau wadah yang bisa Anda beri nama, dan di dalamnya Anda bisa menyimpan berbagai jenis barang (data).

**Tipe data** adalah klasifikasi yang menentukan jenis nilai yang dapat disimpan oleh sebuah variabel. Misalkan, sebuah variabel bisa menyimpan angka bulat, angka desimal, teks, atau nilai benar/salah. Dart adalah bahasa yang *type-safe*, yang berarti setiap variabel memiliki tipe data yang jelas. Ini membantu Dart mendeteksi kesalahan pada saat kompilasi (sebelum program dijalankan), sehingga program menjadi lebih stabil dan andal.

### 3.2 Deklarasi Variabel: `var`, `final`, `const`

Dart menyediakan beberapa cara untuk mendeklarasikan variabel, masing-masing dengan karakteristik yang berbeda:

#### 3.2.1 `var` (Type Inference)

`var` digunakan untuk mendeklarasikan variabel di mana tipe datanya akan secara otomatis disimpulkan (*infer*) oleh Dart berdasarkan nilai awal yang diberikan. Setelah tipe data disimpulkan, variabel tersebut tidak dapat diubah ke tipe data lain, tetapi nilainya bisa diubah.

```
void main() {  
    var nama = 'Alice'; // Dart menginfer `nama` sebagai String  
    print(nama);  
  
    nama = 'Bob'; // Nilai `nama` bisa diubah
```

```
print(nama);

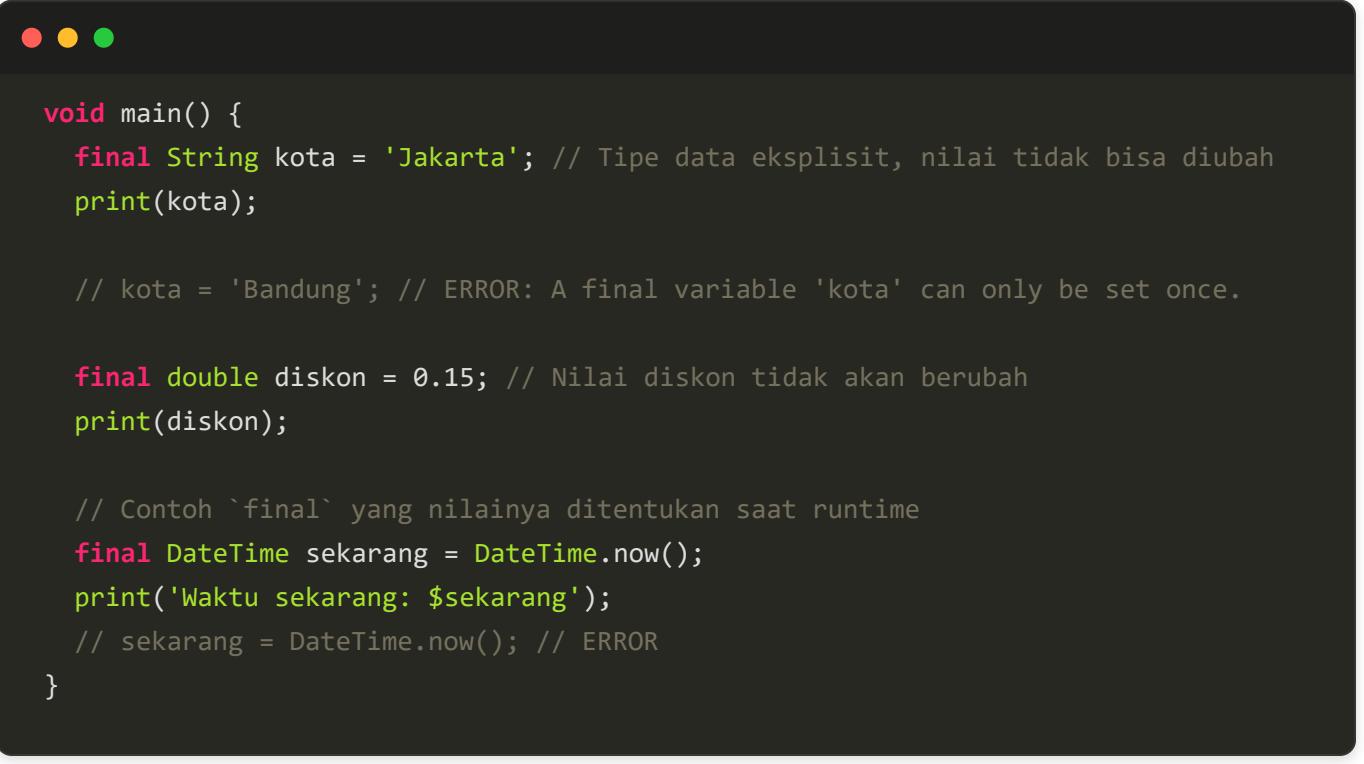
// nama = 123; // ERROR: A value of type 'int' can't be assigned to a variable of type 'String'.

var umur = 25; // Dart menginfer `umur` sebagai int
print(umur);

umur = 26; // Nilai `umur` bisa diubah
print(umur);
}
```

### 3.2.2 `final` (Immutable Once Initialized)

`final` digunakan untuk mendeklarasikan variabel yang nilainya hanya dapat diinisialisasi **sekali**. Setelah diinisialisasi, nilainya tidak dapat diubah lagi. Nilai `final` dapat ditentukan pada saat runtime (saat program berjalan).



```
void main() {
    final String kota = 'Jakarta'; // Tipe data eksplisit, nilai tidak bisa diubah
    print(kota);

    // kota = 'Bandung'; // ERROR: A final variable 'kota' can only be set once.

    final double diskon = 0.15; // Nilai diskon tidak akan berubah
    print(diskon);

    // Contoh `final` yang nilainya ditentukan saat runtime
    final DateTime sekarang = DateTime.now();
    print('Waktu sekarang: $sekarang');
    // sekarang = DateTime.now(); // ERROR
}
```

### 3.2.3 `const` (Compile-time Constant)

`const` digunakan untuk mendeklarasikan variabel yang nilainya harus diketahui **pada waktu kompilasi**. Ini berarti nilai tersebut harus konstan dan tidak dapat berubah sama sekali, bahkan tidak bisa ditentukan saat runtime. `const` lebih ketat daripada `final`.

```
void main() {  
    const double PI = 3.14159; // Nilai PI adalah konstan pada waktu kompilasi  
    print(PI);  
  
    // PI = 3.14; // ERROR: Constant variables can't be assigned a value.  
  
    const String APP_NAME = 'My Awesome App';  
    print(APP_NAME);  
  
    // const DateTime waktuMulai = DateTime.now(); // ERROR: const variables must be initialized  
}
```

### Kapan menggunakan yang mana?

- Gunakan `var` jika Anda tahu variabel akan berubah nilainya dan Anda ingin Dart menginfer tipenya.
- Gunakan `final` jika variabel akan diinisialisasi sekali dan tidak akan berubah setelahnya, dan nilainya bisa ditentukan saat runtime.
- Gunakan `const` jika variabel adalah konstanta sejati yang nilainya sudah diketahui pada waktu kompilasi.

## 3.3 Tipe Data Dasar

Dart memiliki beberapa tipe data bawaan (built-in types) yang paling sering digunakan:

### 3.3.1 `int` (Integer)

Digunakan untuk menyimpan bilangan bulat (tanpa desimal). Contoh: `1`, `100`, `-50`.

```
void main() {  
    int jumlahBarang = 150;  
    int suhu = -10;  
    print('Jumlah barang: $jumlahBarang');  
    print('Suhu: $suhu derajat Celsius');  
}
```

### 3.3.2 `double` (Floating-point Number)

Digunakan untuk menyimpan bilangan desimal (pecahan). Contoh: `3.14`, `0.5`, `-12.75`.

```
void main() {  
    double hargaProduk = 25.99;  
    double beratBadan = 65.5;  
    print('Harga produk: $$hargaProduk'); // $ adalah simbol dolar  
    print('Berat badan: $beratBadan kg');  
}
```

### 3.3.3 String (Text)

Digunakan untuk menyimpan urutan karakter (teks). String diapit oleh tanda kutip tunggal (' ') atau ganda (""). Anda juga bisa menggunakan tiga tanda kutip ('''') atau (''''') untuk string multi-baris.

```
void main() {  
    String namaDepan = 'Budi';  
    String namaBelakang = "Santoso";  
    String namaLengkap = namaDepan + ' ' + namaBelakang;  
    print('Nama lengkap: $namaLengkap');  
  
    String pesanMultiBaris = '''  
    Ini adalah pesan  
    yang ditulis dalam  
    beberapa baris.  
''';  
    print(pesanMultiBaris);  
  
    // Interpolasi String (lebih disarankan)  
    int tahunLahir = 1990;  
    print('Halo, $namaDepan! Anda lahir tahun $tahunLahir.');// Bisa juga ekspresi  
}
```

### 3.3.4 bool (Boolean)

Digunakan untuk menyimpan nilai kebenaran: `true` atau `false`. Sangat penting untuk logika kondisional.

```
void main() {  
    bool isMahasiswa = true;  
    bool isLulus = false;  
    print('Apakah dia mahasiswa? $isMahasiswa');  
    print('Apakah dia sudah lulus? $isLulus');  
  
    if (isMahasiswa) {  
        print('Dia adalah seorang mahasiswa.');//  
    }  
}
```

### 3.4 `dynamic` dan `Object`

#### 3.4.1 `dynamic`

Variabel yang dideklarasikan dengan `dynamic` dapat menyimpan nilai dari tipe data apa pun, dan tipe datanya dapat berubah selama runtime. Ini memberikan fleksibilitas, tetapi mengurangi keamanan tipe (type-safety) yang merupakan salah satu keunggulan Dart. Gunakan `dynamic` dengan hati-hati.

```
void main() {  
    dynamic x = 'Halo'; // x adalah String  
    print(x); // Output: Halo  
  
    x = 123; // x sekarang adalah int  
    print(x); // Output: 123  
  
    x = true; // x sekarang adalah bool  
    print(x); // Output: true  
}
```

#### 3.4.2 `Object`

`Object` adalah superclass dari semua tipe data di Dart. Ini berarti setiap nilai di Dart (angka, string, boolean, dll.) secara implisit adalah sebuah `Object`. Variabel bertipe `Object` dapat menyimpan nilai dari tipe data apa pun, mirip dengan `dynamic`, tetapi dengan perbedaan penting: Anda tidak dapat memanggil metode atau mengakses properti yang spesifik untuk tipe data tertentu tanpa melakukan *type casting* atau *type checking* terlebih dahulu.

```
void main() {  
    Object nilai = 'Dart Programming'; // nilai adalah String  
    print(nilai); // Output: Dart Programming  
  
    nilai = 42; // nilai sekarang adalah int  
    print(nilai); // Output: 42  
  
    // print(nilai.length); // ERROR: The getter 'length' isn't defined for the type '(  
  
    // Untuk mengakses properti spesifik, harus di-cast atau di-check  
    if (nilai is String) {  
        print((nilai as String).length); // Output: 16 (jika nilai masih 'Dart Programming')  
    }  
}
```

### Perbedaan `dynamic` vs `Object`:

- `dynamic` memungkinkan Anda memanggil metode apa pun pada variabel tanpa pemeriksaan tipe saat kompilasi (pemeriksaan dilakukan saat runtime). Ini bisa menyebabkan error runtime jika metode tidak ada.
- `Object` memerlukan pemeriksaan tipe eksplisit atau *casting* sebelum memanggil metode yang spesifik untuk tipe data tertentu. Ini lebih aman karena error akan terdeteksi saat kompilasi.

## 3.5 Null Safety

Null Safety adalah fitur penting di Dart yang membantu Anda menghindari error *null reference* (sering disebut *null pointer exception* di bahasa lain). Dengan Null Safety, variabel tidak dapat memiliki nilai `null` secara default, kecuali Anda secara eksplisit mengizinkannya.

### 3.5.1 Variabel Non-nullable (Default)

Secara default, semua variabel di Dart adalah *non-nullable*. Artinya, mereka harus memiliki nilai dan tidak boleh `null`.

```
void main() {  
    int angka;  
    // print(angka); // ERROR: Non-nullable variable 'angka' must be assigned before it is used.  
  
    int angka2 = 10;
```

```
print(angka2);

String nama = 'John Doe';
// nama = null; // ERROR: A value of type 'Null' can't be assigned to a variable o-
}
```

### 3.5.2 Variabel Nullable ( ? )

Untuk mengizinkan variabel memiliki nilai `null`, Anda dapat menambahkan tanda tanya (`?`) setelah tipe datanya.

```
void main() {
    int? angkaNullable; // Variabel ini bisa null
    print(angkaNullable); // Output: null

    angkaNullable = 20;
    print(angkaNullable); // Output: 20

    String? pesan = null; // Pesan bisa null
    print(pesan);

    pesan = 'Halo dunia!';
    print(pesan);
}
```

### 3.5.3 Operator ! (Null Assertion Operator)

Operator `!` digunakan untuk memberi tahu Dart bahwa Anda yakin sebuah ekspresi non-nullable tidak akan `null` pada saat itu. Gunakan ini dengan hati-hati, karena jika ekspresi tersebut ternyata `null`, program akan *crash* (runtime error).

```
void main() {
    String? nama = 'Alice';
    print(nama!.length); // Kita yakin nama tidak null, jadi kita bisa akses length

    String? alamat;
```

```
// print(alamat!.length); // ERROR: Jika alamat null, ini akan menyebabkan runtime  
}
```

### 3.5.4 late Keyword

Kata kunci `late` digunakan untuk mendeklarasikan variabel non-nullable yang akan diinisialisasi nanti, tetapi Anda berjanji bahwa variabel tersebut akan memiliki nilai sebelum digunakan pertama kali. Ini berguna untuk variabel yang nilainya bergantung pada sesuatu yang belum tersedia saat deklarasi, atau untuk inisialisasi yang mahal.

```
● ● ●  
  
class Kucing {  
    late String nama;  
  
    Kucing(String namaAwal) {  
        nama = namaAwal; // Inisialisasi 'nama' di constructor  
    }  
  
    void sapa() {  
        print('Meong! Nama saya $nama.');//  
    }  
}  
  
void main() {  
    late String deskripsi;  
    deskripsi = 'Ini adalah deskripsi yang diinisialisasi nanti.';  
    print(deskripsi);  
  
    Kucing myCat = Kucing('Kitty');  
    myCat.sapa();  
  
    // Jika kita tidak menginisialisasi 'deskripsi' sebelum digunakan, akan ada error !  
    // late String belumDiinisialisasi;  
    // print(belumDiinisialisasi); // ERROR: LateInitializationError  
}
```

## Studi Kasus: Sistem Pendaftaran Pengguna Sederhana

Mari kita buat program sederhana yang mensimulasikan pendaftaran pengguna, menggunakan variabel, tipe data, dan null safety.

```
import 'dart:io';

void main() {
    print('--- Sistem Pendaftaran Pengguna ---');

    // Menggunakan late untuk variabel yang akan diinisialisasi dari input pengguna
    late String namaLengkap;
    late int umur;
    late String email;
    late bool isSetujuSyarat;

    stdout.write('Masukkan nama lengkap Anda: ');
    namaLengkap = stdin.readLineSync()!; // Menggunakan ! karena kita yakin input tidak kosong

    stdout.write('Masukkan umur Anda: ');
    String? inputUmur = stdin.readLineSync();
    if (inputUmur != null && int.tryParse(inputUmur) != null) {
        umur = int.parse(inputUmur);
    } else {
        print('Umur tidak valid. Menggunakan umur default 0.');
        umur = 0;
    }

    stdout.write('Masukkan email Anda (opsional): ');
    email = stdin.readLineSync() ?? 'Tidak Ada Email'; // Jika null, gunakan nilai default

    stdout.write('Apakah Anda setuju dengan syarat dan ketentuan? (ya/tidak): ');
    String? inputSetuju = stdin.readLineSync()?.toLowerCase();
    isSetujuSyarat = (inputSetuju == 'ya');

    print('\n--- Ringkasan Pendaftaran ---');
    print('Nama Lengkap: $namaLengkap');
    print('Umur: $umur tahun');
    print('Email: $email');
    print('Setuju Syarat & Ketentuan: ${isSetujuSyarat ? "Ya" : "Tidak"}');

    if (umur < 17) {
        print('Peringatan: Pengguna di bawah umur.');
    }
}
```

**Penjelasan:**

- Kita menggunakan `late` untuk `namaLengkap`, `umur`, `email`, dan `isSetujuSyarat` karena nilainya akan diisi setelah mendapatkan input dari pengguna.
- `stdin.readLineSync()!` digunakan untuk membaca input. Operator `!` (null assertion) digunakan karena kita berasumsi pengguna akan selalu memberikan input (meskipun dalam aplikasi nyata, Anda harus menangani kasus null dengan lebih hati-hati).
- Untuk `umur`, kita menggunakan `int.tryParse()` untuk mencoba mengkonversi string ke integer. Jika gagal (misalnya, pengguna memasukkan teks), kita memberikan nilai default.
- Untuk `email`, kita menggunakan operator `??` (null-aware operator) yang berarti "jika `stdin.readLineSync()` mengembalikan `null`, gunakan string 'Tidak Ada Email' sebagai gantinya".
- `isSetujuSyarat` adalah `bool` yang ditentukan berdasarkan input pengguna.
- Ada percabangan sederhana untuk memberikan peringatan jika umur di bawah 17 tahun.

**Latihan:**

1. Modifikasi program di atas untuk menambahkan input nomor telepon (boleh null).
2. Tambahkan validasi untuk format email sederhana (misalnya, harus mengandung '@').
3. Coba gunakan `final` atau `const` untuk variabel yang sesuai dalam program ini.

**• Referensi:**

- [dart.dev/language/variables](https://dart.dev/language/variables)
- [dart.dev/language/built-in-types](https://dart.dev/language/built-in-types)
- [dart.dev/null-safety](https://dart.dev/null-safety)

## Pertemuan 4: Operator dan Input/Output Konsol

### Tujuan Pembelajaran

Setelah mempelajari materi ini, mahasiswa diharapkan mampu:

- Mengidentifikasi dan menggunakan berbagai jenis operator di Dart.
- Melakukan operasi matematika, perbandingan, logika, dan penugasan.
- Melakukan input data dari pengguna melalui konsol.
- Menampilkan output informasi ke konsol dengan berbagai cara.

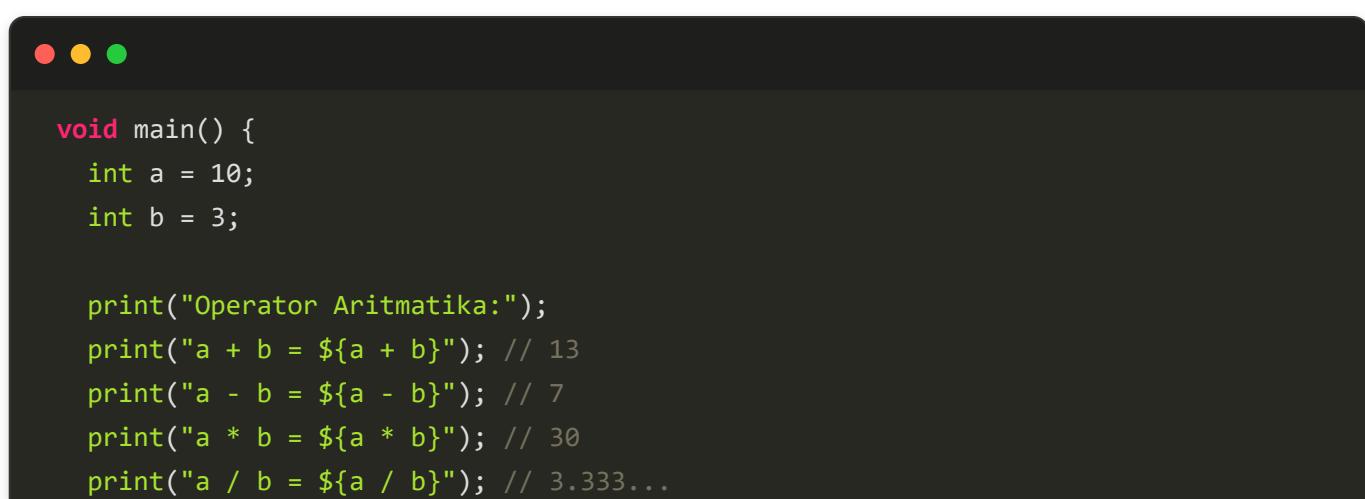
### 4.1 Operator di Dart

Operator adalah simbol yang memberi tahu kompiler untuk melakukan operasi matematika atau logis tertentu. Dart menyediakan berbagai jenis operator yang mirip dengan bahasa pemrograman lain.

#### 4.1.1 Operator Aritmatika

Digunakan untuk melakukan operasi matematika dasar.

Operator	Deskripsi	Contoh	Hasil
+	Penjumlahan	5 + 3	8
-	Pengurangan	5 - 3	2
*	Perkalian	5 * 3	15
/	Pembagian	5 / 3	1.66
~/	Pembagian Integer	5 ~/ 3	1
%	Modulo (Sisa Bagi)	5 % 3	2



```
void main() {
    int a = 10;
    int b = 3;

    print("Operator Aritmatika:");
    print("a + b = ${a + b}"); // 13
    print("a - b = ${a - b}"); // 7
    print("a * b = ${a * b}"); // 30
    print("a / b = ${a / b}"); // 3.333...
```

```

print("a ~/ b = ${a ~/ b}"); // 3 (hasil pembagian bulat)
print("a % b = ${a % b}"); // 1 (sisa bagi)

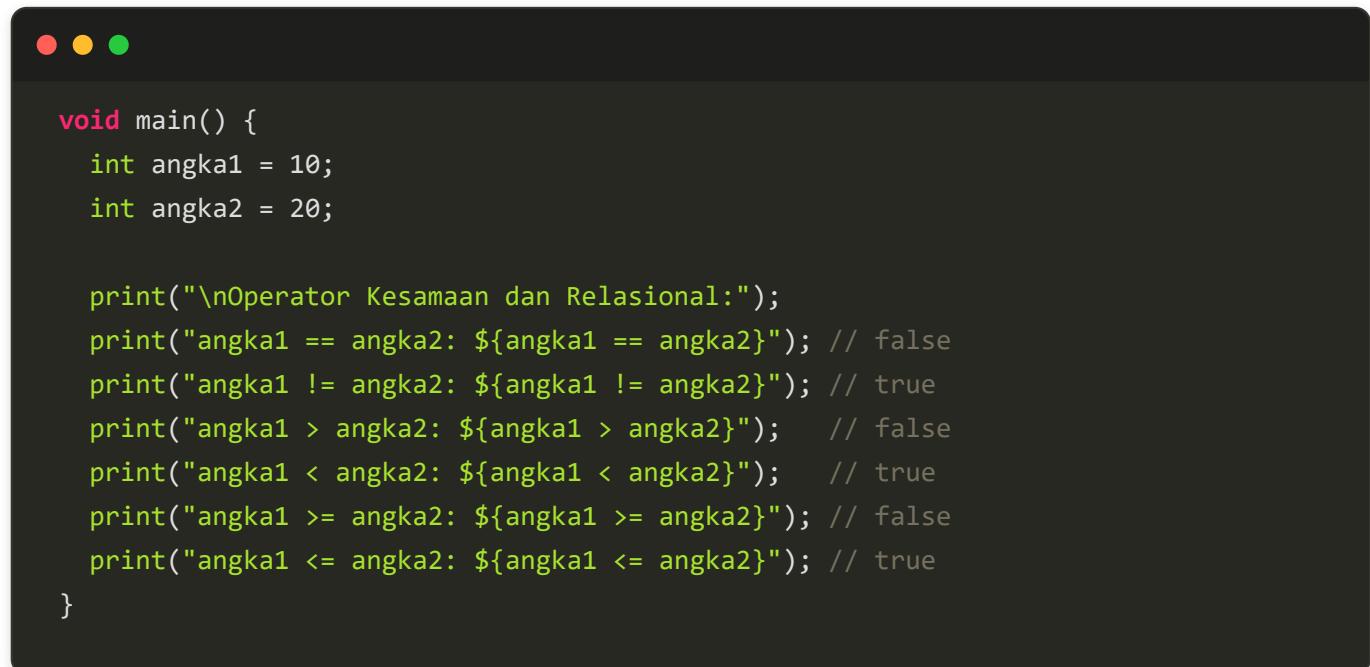
double x = 10.0;
double y = 3.0;
print("x / y = ${x / y}"); // 3.333...
}

```

#### 4.1.2 Operator Kesamaan dan Relasional

Digunakan untuk membandingkan dua nilai dan menghasilkan nilai boolean (`true` atau `false`).

Operator	Deskripsi	Contoh	Hasil
<code>==</code>	Sama dengan	<code>5 == 3</code>	<code>false</code>
<code>!=</code>	Tidak sama dengan	<code>5 != 3</code>	<code>true</code>
<code>&gt;</code>	Lebih besar dari	<code>5 &gt; 3</code>	<code>true</code>
<code>&lt;</code>	Lebih kecil dari	<code>5 &lt; 3</code>	<code>false</code>
<code>&gt;=</code>	Lebih besar atau sama dengan	<code>5 &gt;= 3</code>	<code>true</code>
<code>&lt;=</code>	Lebih kecil atau sama dengan	<code>5 &lt;= 3</code>	<code>false</code>



```

void main() {
    int angka1 = 10;
    int angka2 = 20;

    print("\nOperator Kesamaan dan Relasional:");
    print("angka1 == angka2: ${angka1 == angka2}"); // false
    print("angka1 != angka2: ${angka1 != angka2}"); // true
    print("angka1 > angka2: ${angka1 > angka2}"); // false
    print("angka1 < angka2: ${angka1 < angka2}"); // true
    print("angka1 >= angka2: ${angka1 >= angka2}"); // false
    print("angka1 <= angka2: ${angka1 <= angka2}"); // true
}

```

#### 4.1.3 Operator Penugasan

Digunakan untuk memberikan nilai ke variabel. Ada juga operator penugasan gabungan yang melakukan operasi dan penugasan secara bersamaan.

Operator	Deskripsi	Contoh	Setara dengan
=	Penugasan	x = 10	x = 10
+=	Tambah dan Tugaskan	x += 5	x = x + 5
-=	Kurang dan Tugaskan	x -= 5	x = x - 5
*=	Kali dan Tugaskan	x *= 5	x = x * 5
/=	Bagi dan Tugaskan	x /= 5	x = x / 5
%=	Modulo dan Tugaskan	x %= 5	x = x % 5

```

void main() {
    int nilai = 10;
    print("\nOperator Penugasan:");
    print("Nilai awal: $nilai"); // 10

    nilai += 5; // nilai = 10 + 5 = 15
    print("Nilai setelah += 5: $nilai"); // 15

    nilai -= 3; // nilai = 15 - 3 = 12
    print("Nilai setelah -= 3: $nilai"); // 12

    nilai *= 2; // nilai = 12 * 2 = 24
    print("Nilai setelah *= 2: $nilai"); // 24

    nilai ~/= 4; // nilai = 24 ~/ 4 = 6
    print("Nilai setelah ~/= 4: $nilai"); // 6

    nilai %= 5; // nilai = 6 % 5 = 1
    print("Nilai setelah %= 5: $nilai"); // 1
}

```

#### 4.1.4 Operator Logika

Digunakan untuk menggabungkan atau memanipulasi nilai boolean.

Operator	Deskripsi	Contoh	Hasil
----------	-----------	--------	-------

Operator	Deskripsi	Contoh	Hasil
&&	AND (Logika DAN)	true && false	false
	OR (Logika ATAU)	true    false	true
!	NOT (Logika TIDAK)	!true	false



```

void main() {
  bool kondisi1 = true;
  bool kondisi2 = false;

  print("\nOperator Logika:");
  print("kondisi1 && kondisi2: ${kondisi1 && kondisi2}"); // false
  print("kondisi1 || kondisi2: ${kondisi1 || kondisi2}"); // true
  print("!kondisi1: ${!kondisi1}"); // false

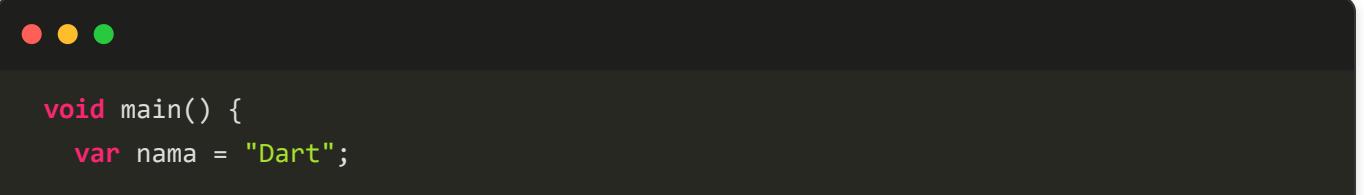
  int umur = 20;
  bool punyaSIM = true;
  if (umur >= 17 && punyaSIM) {
    print("Boleh mengemudi.");
  } else {
    print("Tidak boleh mengemudi.");
  }
}

```

#### 4.1.5 Operator Tipe

Digunakan untuk memeriksa atau mengkonversi tipe data.

Operator	Deskripsi	Contoh
is	Memeriksa tipe	obj is String
is!	Memeriksa bukan tipe	obj is! int
as	Typecast (Konversi Tipe)	obj as String



```

void main() {
  var nama = "Dart";

```

```
var angka = 123;

print("\nOperator Tipe:");
print("nama is String: ${nama is String}"); // true
print("angka is! double: ${angka is! double}"); // true

Object nilaiCampuran = "Halo Dunia";
if (nilaiCampuran is String) {
    // Melakukan typecast secara aman
    String pesan = nilaiCampuran as String;
    print("Panjang pesan: ${pesan.length}");
}

// Contoh typecast yang tidak aman (akan error jika tipe tidak cocok)
// int x = nilaiCampuran as int; // Ini akan menyebabkan error jika nilaiCampuran !=
```

## 4.2 Input/Output Konsol

Interaksi dengan pengguna melalui konsol adalah dasar dari banyak aplikasi. Dart menyediakan cara sederhana untuk membaca input dari keyboard dan menampilkan output ke layar konsol.

### 4.2.1 Output ke Konsol (`print()` dan `stdout.write()`)

- `print()` : Fungsi yang paling umum digunakan untuk menampilkan teks ke konsol. Secara otomatis menambahkan karakter baris baru (`\n`) di akhir output.
- `stdout.write()` : Digunakan untuk menampilkan teks ke konsol tanpa menambahkan karakter baris baru. Berguna ketika Anda ingin prompt input muncul di baris yang sama dengan kursor.



```
import 'dart:io'; // Perlu diimpor untuk stdout

void main() {
    print("Ini adalah baris pertama.");
    print("Ini adalah baris kedua.");

    stdout.write("Masukkan nama Anda: ");
    // Input akan dibaca di baris yang sama
}
```

#### 4.2.2 Input dari Konsol (`stdin.readLineSync()`)

Untuk membaca input dari pengguna melalui keyboard, kita menggunakan `stdin.readLineSync()`. Fungsi ini membaca satu baris teks dari input standar (keyboard) hingga pengguna menekan Enter. Hasilnya adalah `String?` (`String nullable`), karena pengguna mungkin tidak memasukkan apa pun (mengembalikan `null`).



```
import 'dart:io'; // Perlu diimpor untuk stdin

void main() {
    stdout.write("Siapa nama Anda? ");
    String? nama = stdin.readLineSync(); // Membaca input dari pengguna

    if (nama != null && nama.isNotEmpty) {
        print("Halo, $nama! Selamat datang di Dart.");
    } else {
        print("Nama tidak boleh kosong.");
    }

    stdout.write("Masukkan umur Anda: ");
    String? inputUmur = stdin.readLineSync();
    try {
        int umur = int.parse(inputUmur!); // Mengkonversi string ke integer
        print("Umur Anda adalah $umur tahun.");
    } catch (e) {
        print("Input umur tidak valid.");
    }
}
```

#### Penting:

- Anda perlu mengimpor library `dart:io` untuk menggunakan `stdin` dan `stdout`.
- `stdin.readLineSync()` mengembalikan `String?` (`nullable`). Anda harus menangani kemungkinan nilai `null` untuk menghindari error. Operator `!` (null assertion) digunakan di `inputUmur!` untuk memberi tahu Dart bahwa kita yakin `inputUmur` tidak akan `null` saat itu, tetapi ini berisiko jika input benar-benar `null`.
- Untuk mengkonversi input string ke tipe data lain (misalkan `int` atau `double`), Anda bisa menggunakan metode `parse()` (misalkan `int.parse()`, `double.parse()`). Selalu gunakan `try-catch` atau `tryParse()` untuk menangani error jika input tidak sesuai format yang diharapkan.

## Studi Kasus: Kalkulator Sederhana

Mari kita buat program kalkulator sederhana yang menerima dua angka dan satu operator dari pengguna, lalu menampilkan hasilnya.

```
import 'dart:io';

void main() {
    print('--- Kalkulator Sederhana ---');

    stdout.write('Masukkan angka pertama: ');
    String? inputAngka1 = stdin.readLineSync();
    double? angka1 = double.tryParse(inputAngka1 ?? '');

    stdout.write('Masukkan operator (+, -, *, /, %): ');
    String? operator = stdin.readLineSync();

    stdout.write('Masukkan angka kedua: ');
    String? inputAngka2 = stdin.readLineSync();
    double? angka2 = double.tryParse(inputAngka2 ?? '');

    if (angka1 == null || angka2 == null || operator == null) {
        print('Input tidak valid. Pastikan Anda memasukkan angka dan operator yang benar');
        return;
    }

    double hasil;
    switch (operator) {
        case '+':
            hasil = angka1 + angka2;
            print('Hasil: $angka1 + $angka2 = $hasil');
            break;
        case '-':
            hasil = angka1 - angka2;
            print('Hasil: $angka1 - $angka2 = $hasil');
            break;
        case '*':
            hasil = angka1 * angka2;
            print('Hasil: $angka1 * $angka2 = $hasil');
            break;
        case '/':
            if (angka2 != 0) {
```

```
    hasil = angka1 / angka2;
    print('Hasil: $angka1 / $angka2 = $hasil');
} else {
    print('Error: Pembagian dengan nol tidak diizinkan.');
}
break;
case '%':
    hasil = angka1 % angka2;
    print('Hasil: $angka1 % $angka2 = $hasil');
    break;
default:
    print('Operator tidak dikenal.');
}
}
```

### Penjelasan:

- Program meminta dua angka dan satu operator dari pengguna.
- `double.tryParse()` digunakan untuk mengkonversi input string ke `double`. Jika konversi gagal, ia mengembalikan `null`, yang kemudian ditangani dengan operator `?? ''` untuk memastikan `tryParse` menerima string kosong jika input `null`.
- Validasi awal dilakukan untuk memastikan semua input adalah angka yang valid dan operator tidak `null`.
- Pernyataan `switch` digunakan untuk memilih operasi berdasarkan operator yang dimasukkan pengguna.
- Penanganan kasus pembagian dengan nol ditambahkan untuk mencegah error.

### Latihan:

1. Modifikasi kalkulator agar bisa menghitung pangkat (misalnya `2^3`). Anda mungkin perlu mencari cara untuk menghitung pangkat di Dart (hint: `pow` dari `dart:math`).
2. Tambahkan opsi untuk menghitung akar kuadrat dari satu angka.
3. Buat agar kalkulator bisa terus beroperasi sampai pengguna mengetik 'exit'.

### • Referensi:

- [dart.dev/language/operators](https://dart.dev/language/operators)
- [api.dart.dev/stable/dart-io/dart-io-library.html](https://api.dart.dev/stable/dart-io/dart-io-library.html)

## Pertemuan 5: Control Flow (Percabangan)

### Tujuan Pembelajaran

Setelah mempelajari materi ini, mahasiswa diharapkan mampu:

- Memahami konsep *control flow* atau alur kontrol dalam pemrograman.
- Mengimplementasikan logika percabangan menggunakan pernyataan `if`, `else if`, dan `else`.
- Menggunakan operator ternary untuk percabangan sederhana.
- Menggunakan pernyataan `switch` untuk percabangan berdasarkan nilai ekspresi.

### 5.1 Pengenalan Control Flow

Dalam program komputer, *control flow* (alur kontrol) adalah urutan di mana instruksi individu atau pernyataan dieksekusi. Secara default, program dieksekusi secara sekuensial (berurutan dari atas ke bawah). Namun, dalam banyak kasus, kita perlu mengubah alur ini berdasarkan kondisi tertentu atau untuk mengulang blok kode. Inilah peran *control flow statements*.

Ada dua jenis utama *control flow*:

1. **Percabangan (Conditional Statements):** Memungkinkan program untuk memilih jalur eksekusi yang berbeda berdasarkan kondisi tertentu.
2. **Perulangan (Looping Statements):** Memungkinkan program untuk mengulang blok kode berkali-kali.

Pada pertemuan ini, kita akan fokus pada **percabangan**.

### 5.2 Pernyataan `if`, `else if`, `else`

Pernyataan `if` adalah struktur percabangan paling dasar. Ini memungkinkan Anda untuk mengeksekusi blok kode hanya jika suatu kondisi bernilai `true`.

#### 5.2.1 `if` Statement

Sintaks dasar `if`:

```
if (kondisi) {  
    // Kode yang akan dieksekusi jika kondisi true  
}
```

- **kondisi**: Sebuah ekspresi boolean yang dievaluasi menjadi `true` atau `false`.

Contoh:

```
void main() {  
    int nilai = 85;  
  
    if (nilai >= 70) {  
        print("Anda lulus ujian!");  
    }  
}
```

### 5.2.2 if-else Statement

Pernyataan `if-else` memungkinkan Anda untuk mengeksekusi satu blok kode jika kondisi `true`, dan blok kode lain jika kondisi `false`.

Sintaks dasar `if-else` :

```
if (kondisi) {  
    // Kode jika kondisi true  
} else {  
    // Kode jika kondisi false  
}
```

Contoh:

```
void main() {  
    int nilai = 60;  
  
    if (nilai >= 70) {  
        print("Anda lulus ujian!");  
    } else {  
        print("Anda tidak lulus ujian. Coba lagi!");  
    }  
}
```

### 5.2.3 if-else if-else Statement

Ketika Anda memiliki lebih dari dua kemungkinan jalur eksekusi, Anda dapat menggunakan serangkaian **else if** untuk menguji beberapa kondisi secara berurutan.

Sintaks dasar **if-else if-else** :

```
if (kondisi1) {  
    // Kode jika kondisi1 true  
} else if (kondisi2) {  
    // Kode jika kondisi1 false DAN kondisi2 true  
} else if (kondisi3) {  
    // Kode jika kondisi1 false, kondisi2 false DAN kondisi3 true  
} else {  
    // Kode jika semua kondisi di atas false  
}
```

Contoh:

```
void main() {  
    int nilai = 75;  
  
    if (nilai >= 90) {  
        print("Nilai Anda A");  
    } else if (nilai >= 80) {  
        print("Nilai Anda B");  
    } else if (nilai >= 70) {  
        print("Nilai Anda C");  
    } else if (nilai >= 60) {  
        print("Nilai Anda D");  
    } else {  
        print("Nilai Anda E");  
    }  
}
```

**Penting:** Urutan **else if** sangat penting. Dart akan mengevaluasi kondisi dari atas ke bawah, dan blok kode pertama yang kondisinya **true** akan dieksekusi. Kondisi selanjutnya tidak akan dievaluasi.

### 5.3 Operator Ternary (Conditional Expression)

Operator ternary adalah cara singkat untuk menulis pernyataan `if-else` sederhana yang mengembalikan sebuah nilai. Ini sangat berguna untuk penugasan variabel berdasarkan kondisi.

Sintaks dasar operator ternary:

```
variabel = (kondisi) ? nilai_jika_true : nilai_jika_false;
```

Contoh:

```
void main() {
    int suhu = 28;
    String statusCuaca;

    statusCuaca = (suhu > 25) ? "Panas" : "Normal";
    print("Status cuaca: $statusCuaca"); // Output: Panas

    suhu = 20;
    statusCuaca = (suhu > 25) ? "Panas" : "Normal";
    print("Status cuaca: $statusCuaca"); // Output: Normal

    // Contoh lain
    String pesan = (suhu > 25) ? "Suhu tinggi" : "Suhu sedang";
    print(pesan);
}
```

### 5.4 Pernyataan `switch`

Pernyataan `switch` digunakan ketika Anda memiliki banyak kemungkinan nilai untuk satu variabel atau ekspresi, dan Anda ingin mengeksekusi blok kode yang berbeda untuk setiap nilai tersebut. Ini seringkali lebih rapi daripada serangkaian `if-else if` yang panjang.

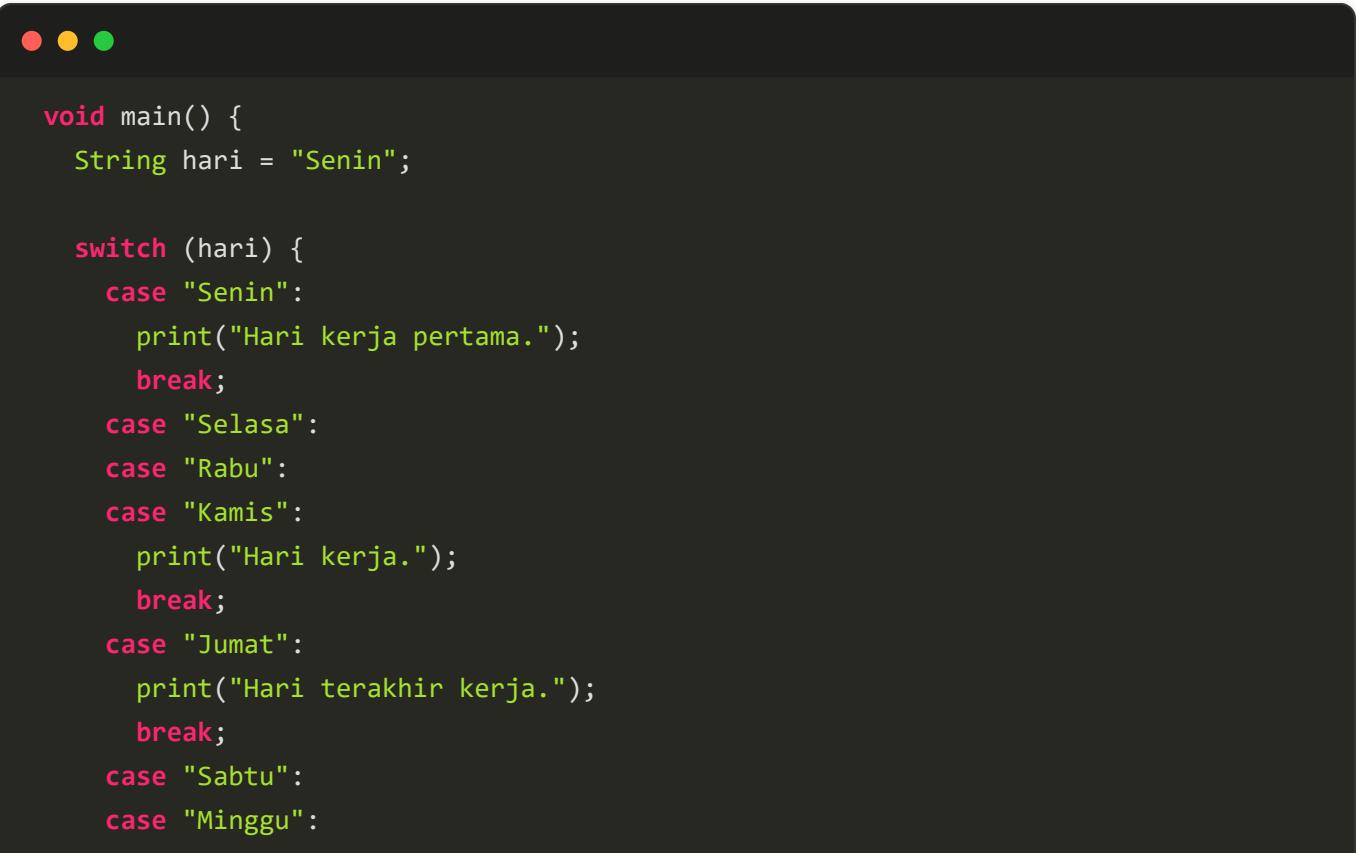
Sintaks dasar `switch` :

```
switch (ekspresi) {
```

```
case nilai1:  
    // Kode jika ekspresi == nilai1  
    break;  
case nilai2:  
    // Kode jika ekspresi == nilai2  
    break;  
default:  
    // Kode jika tidak ada case yang cocok  
    break;  
}
```

- `ekspresi` : Nilai yang akan dievaluasi. Tipe data ekspresi harus berupa `int`, `String`, atau tipe yang dapat dikonversi ke `int` atau `String` (misalnya `enum`).
- `case nilaiX` : Setiap `case` membandingkan `ekspresi` dengan `nilaiX`. Jika cocok, blok kode di bawah `case` tersebut akan dieksekusi.
- `break;` : Penting! Pernyataan `break` digunakan untuk keluar dari blok `switch` setelah sebuah `case` dieksekusi. Jika `break` dihilangkan, eksekusi akan "jatuh" ke `case` berikutnya (fall-through), yang biasanya bukan perilaku yang diinginkan.
- `default:` : Opsional. Blok `default` akan dieksekusi jika tidak ada `case` yang cocok dengan nilai `ekspresi`.

Contoh:



```
void main() {  
    String hari = "Senin";  
  
    switch (hari) {  
        case "Senin":  
            print("Hari kerja pertama.");  
            break;  
        case "Selasa":  
        case "Rabu":  
        case "Kamis":  
            print("Hari kerja.");  
            break;  
        case "Jumat":  
            print("Hari terakhir kerja.");  
            break;  
        case "Sabtu":  
        case "Minggu":  
    }  
}
```

```
    print("Hari libur!");
    break;
default:
    print("Nama hari tidak valid.");
}

int kodeError = 404;
switch (kodeError) {
    case 200:
        print("OK - Permintaan berhasil.");
        break;
    case 400:
        print("Bad Request - Permintaan tidak valid.");
        break;
    case 401:
        print("Unauthorized - Tidak memiliki otorisasi.");
        break;
    case 404:
        print("Not Found - Sumber daya tidak ditemukan.");
        break;
    case 500:
        print("Internal Server Error - Terjadi kesalahan di server.");
        break;
default:
    print("Kode error tidak dikenal.");
}
}
```

### Studi Kasus: Penentuan Kategori Usia

Buatlah program yang meminta pengguna memasukkan umurnya, lalu program akan menentukan kategori usia berdasarkan aturan berikut:

- 0-12 tahun: Anak-anak
- 13-19 tahun: Remaja
- 20-59 tahun: Dewasa
- 60 tahun ke atas: Lansia

```
import 'dart:io';

void main() {
```

```
stdout.write('Masukkan umur Anda: ');
String? inputUmur = stdin.readLineSync();

if (inputUmur == null || inputUmur.isEmpty) {
    print('Input umur tidak boleh kosong.');
    return;
}

try {
    int umur = int.parse(inputUmur);
    String kategoriUsia;

    if (umur >= 0 && umur <= 12) {
        kategoriUsia = 'Anak-anak';
    } else if (umur >= 13 && umur <= 19) {
        kategoriUsia = 'Remaja';
    } else if (umur >= 20 && umur <= 59) {
        kategoriUsia = 'Dewasa';
    } else if (umur >= 60) {
        kategoriUsia = 'Lansia';
    } else {
        kategoriUsia = 'Umur tidak valid';
    }

    print('Kategori usia Anda: $kategoriUsia');

} catch (e) {
    print('Input tidak valid. Harap masukkan angka.');
}
}
```

### Penjelasan:

- Program membaca input umur dari pengguna.
- Melakukan validasi input untuk memastikan tidak kosong dan bisa diubah menjadi angka.
- Menggunakan serangkaian `if-else if-else` untuk menentukan kategori usia berdasarkan rentang umur.
- Menampilkan kategori usia yang sesuai.

### Latihan:

1. Modifikasi studi kasus di atas untuk menggunakan pernyataan `switch` jika memungkinkan (pikirkan bagaimana Anda bisa mengkonversi rentang umur menjadi nilai diskrit yang bisa digunakan `switch`).

2. Buat program yang meminta pengguna memasukkan nilai ujian (0-100), lalu tampilkan grade (A, B, C, D, E) menggunakan `if-else if-else`.
3. Buat program sederhana yang menentukan apakah sebuah angka genap atau ganjil menggunakan operator ternary.

- **Referensi:**

- [dart.dev/language/control-flow#if-and-else](https://dart.dev/language/control-flow#if-and-else)
- [dart.dev/language/control-flow#switch-statements](https://dart.dev/language/control-flow#switch-statements)

## Pertemuan 6: Control Flow (Perulangan)

### Tujuan Pembelajaran

Setelah mempelajari materi ini, mahasiswa diharapkan mampu:

- Memahami konsep perulangan (looping) dalam pemrograman.
- Mengimplementasikan perulangan menggunakan `for` (tradisional dan `for-in` ).
- Menggunakan perulangan `while` dan `do-while` .
- Menggunakan pernyataan `break` dan `continue` untuk mengontrol alur perulangan.

### 6.1 Pengenalan Perulangan

Perulangan adalah struktur kontrol yang memungkinkan kita untuk mengeksekusi blok kode berulang kali. Ini sangat berguna ketika kita perlu melakukan tugas yang sama berkali-kali, seperti memproses setiap item dalam daftar, menghitung sesuatu sampai kondisi tertentu terpenuhi, atau mengulang input pengguna sampai valid.

Dart menyediakan beberapa jenis perulangan:

1. `for` loop: Digunakan ketika kita tahu berapa kali perulangan harus dilakukan.
2. `for-in` loop: Digunakan untuk mengiterasi elemen dalam koleksi (seperti List, Set, Map).
3. `while` loop: Digunakan ketika kita tidak tahu pasti berapa kali perulangan akan dilakukan, tetapi kita tahu kondisi untuk berhenti.
4. `do-while` loop: Mirip dengan `while` loop, tetapi blok kode dieksekusi setidaknya sekali sebelum kondisi diperiksa.

### 6.2 Perulangan `for`

Perulangan `for` adalah perulangan yang paling umum digunakan ketika jumlah iterasi (pengulangan) sudah diketahui sebelumnya.

Sintaks dasar `for` loop:

```
for (inisialisasi; kondisi; increment/decrement) {  
    // Kode yang akan diulang  
}
```

- **inisialisasi** : Dieksekusi sekali di awal perulangan. Biasanya untuk mendeklarasikan dan menginisialisasi variabel kontrol perulangan.
- **kondisi** : Dievaluasi sebelum setiap iterasi. Jika `true`, perulangan berlanjut; jika `false`, perulangan berhenti.
- **increment/decrement** : Dieksekusi setelah setiap iterasi. Biasanya untuk mengubah nilai variabel kontrol perulangan.

Contoh:

```
void main() {  
    print("\n--- Contoh for loop ---");  
    for (int i = 0; i < 5; i++) {  
        print("Iterasi ke-$i");  
    }  
    // Output:  
    // Iterasi ke-0  
    // Iterasi ke-1  
    // Iterasi ke-2  
    // Iterasi ke-3  
    // Iterasi ke-4  
}
```

### 6.2.1 `for-in` Loop

`for-in` loop (juga dikenal sebagai *enhanced for loop* atau *for-each loop* di bahasa lain) digunakan untuk mengiterasi elemen-elemen dalam koleksi (seperti List, Set, atau Map).

Sintaks dasar `for-in` loop:

```
for (var elemen in koleksi) {  
    // Kode yang akan dieksekusi untuk setiap elemen  
}
```

Contoh:

```
void main() {
```

```
print("\n--- Contoh for-in loop ---");
List<String> buah = ["Apel", "Jeruk", "Mangga", "Pisang"];
for (var itemBuah in buah) {
    print("Saya suka $itemBuah");
}
// Output:
// Saya suka Apel
// Saya suka Jeruk
// Saya suka Mangga
// Saya suka Pisang
```

### 6.3 Perulangan while

Perulangan `while` digunakan ketika Anda ingin mengulang blok kode selama kondisi tertentu masih `true`. Jumlah iterasi tidak harus diketahui sebelumnya.

Sintaks dasar `while` loop:

```
while (kondisi) {
    // Kode yang akan diulang
    // Pastikan ada sesuatu yang mengubah kondisi agar perulangan berhenti
}
```

- `kondisi`: Dievaluasi sebelum setiap iterasi. Jika `true`, blok kode dieksekusi; jika `false`, perulangan berhenti.

Contoh:

```
void main() {
    print("\n--- Contoh while loop ---");
    int hitung = 0;
    while (hitung < 3) {
        print("Hitungan: $hitung");
        hitung++; // Penting: mengubah nilai hitung agar perulangan tidak tak terbatas
    }
    // Output:
    // Hitungan: 0
```

```
// Hitungan: 1  
// Hitungan: 2  
}
```

## 6.4 Perulangan do-while

Perulangan **do-while** mirip dengan **while** loop, tetapi perbedaannya adalah blok kode di dalam **do-while** akan dieksekusi **setidaknya sekali**, bahkan jika kondisi awalnya **false**. Kondisi diperiksa setelah blok kode dieksekusi.

Sintaks dasar **do-while** loop:

```
do {  
    // Kode yang akan diulang  
} while (kondisi);
```

Contoh:

```
void main() {  
    print("\n--- Contoh do-while loop ---");  
    int angka = 5;  
    do {  
        print("Angka: $angka");  
        angka++;  
    } while (angka < 5); // Kondisi false, tapi blok dieksekusi sekali  
    // Output:  
    // Angka: 5  
  
    int input;  
    do {  
        stdout.write("Masukkan angka antara 1-10: ");  
        input = int.tryParse(stdin.readLineSync() ?? "") ?? 0;  
    } while (input < 1 || input > 10);  
    print("Anda memasukkan angka: $input");  
}
```

## 6.5 Pernyataan `break` dan `continue`

Kita bisa mengontrol alur perulangan lebih lanjut menggunakan pernyataan `break` dan `continue`.

### 6.5.1 `break`

Pernyataan `break` digunakan untuk **menghentikan perulangan secara paksa** dan keluar dari perulangan tersebut. Eksekusi program akan dilanjutkan pada pernyataan setelah perulangan.

Contoh:

```
void main() {
    print("\n--- Contoh break ---");
    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            print("Angka 5 ditemukan, menghentikan perulangan.");
            break; // Keluar dari for loop
        }
        print("Iterasi: $i");
    }
    print("Perulangan selesai.");
    // Output:
    // Iterasi: 0
    // Iterasi: 1
    // Iterasi: 2
    // Iterasi: 3
    // Iterasi: 4
    // Angka 5 ditemukan, menghentikan perulangan.
    // Perulangan selesai.
}
```

### 6.5.2 `continue`

Pernyataan `continue` digunakan untuk **melewatkannya sisanya dalam iterasi saat ini** dan langsung melanjutkan ke iterasi berikutnya dari perulangan.

Contoh:

```
void main() {
```

```
print("\n--- Contoh continue ---");
for (int i = 0; i < 5; i++) {
    if (i == 2) {
        print("Melewatkkan iterasi ke-2.");
        continue; // Melewatkkan sisa kode di iterasi ini, lanjut ke iterasi berikutnya
    }
    print("Memproses iterasi: $i");
}
// Output:
// Memproses iterasi: 0
// Memproses iterasi: 1
// Melewatkkan iterasi ke-2.
// Memproses iterasi: 3
// Memproses iterasi: 4
}
```

### Studi Kasus: Menghitung Jumlah Angka Genap dan Ganjil

Buatlah program yang meminta pengguna memasukkan 10 angka, lalu menghitung berapa banyak angka genap dan ganjil yang dimasukkan.

```
import 'dart:io';

void main() {
    int jumlahGenap = 0;
    int jumlahGanjil = 0;

    print('--- Penghitung Angka Genap dan Ganjil ---');
    print('Masukkan 10 angka.');

    for (int i = 1; i <= 10; i++) {
        stdout.write('Masukkan angka ke-$i: ');
        String? input = stdin.readLineSync();
        int? angka = int.tryParse(input ?? '');

        if (angka == null) {
            print('Input tidak valid. Silakan masukkan angka.');
            i--; // Kurangi i agar iterasi ini diulang
            continue; // Lanjut ke iterasi berikutnya
        }
    }
}
```

```
if (angka % 2 == 0) {  
    jumlahGenap++;  
} else {  
    jumlahGanjil++;  
}  
  
print('\n--- Hasil ---');  
print('Jumlah angka genap: $jumlahGenap');  
print('Jumlah angka ganjil: $jumlahGanjil');  
}
```

**Penjelasan:**

- Program menggunakan `for` loop untuk mengulang permintaan input sebanyak 10 kali.
- Di setiap iterasi, program meminta angka dari pengguna.
- Validasi input dilakukan menggunakan `int.tryParse()`. Jika input tidak valid, pesan error ditampilkan, `i` dikurangi (agar iterasi saat ini diulang), dan `continue` digunakan untuk langsung ke iterasi berikutnya.
- Operator modulo (`%`) digunakan untuk menentukan apakah angka genap atau ganjil.
- Counter `jumlahGenap` dan `jumlahGanjil` diperbarui sesuai.

**Latihan:**

1. Modifikasi program di atas agar pengguna bisa menentukan berapa banyak angka yang ingin dimasukkan (bukan selalu 10).
2. Buat program yang menampilkan deret Fibonacci hingga N suku. (Deret Fibonacci: 0, 1, 1, 2, 3, 5, 8, ...)
3. Buat program yang meminta pengguna memasukkan password. Program akan terus meminta password sampai password yang dimasukkan benar (misalnya "rahasia123"). Gunakan `do-while` loop.

**• Referensi:**

- [dart.dev/language/control-flow#for-loops](https://dart.dev/language/control-flow#for-loops)
- [dart.dev/language/control-flow#while-and-do-while](https://dart.dev/language/control-flow#while-and-do-while)

## Pertemuan 7: Fungsi

### Tujuan Pembelajaran

Setelah mempelajari materi ini, mahasiswa diharapkan mampu:

- Memahami konsep fungsi dan pentingnya dalam modularisasi kode.
- Mendeklarasikan dan memanggil fungsi dengan atau tanpa nilai kembalian.
- Menggunakan berbagai jenis parameter fungsi: wajib, opsional posisi, dan opsional bernama.
- Menerapkan parameter dengan nilai default.
- Menulis fungsi ekspresi (`=>`) untuk kode yang ringkas.
- Memahami dan menggunakan fungsi anonim (lambda).

### 7.1 Konsep Fungsi

**Fungsi** adalah blok kode yang terorganisir dan dapat digunakan kembali untuk melakukan tugas tertentu. Bayangkan fungsi sebagai sebuah "mesin" kecil yang menerima input (parameter), melakukan serangkaian operasi, dan mungkin menghasilkan output (nilai kembalian).

#### Mengapa Fungsi Penting?

1. **Modularitas:** Memecah program besar menjadi bagian-bagian yang lebih kecil dan mudah dikelola.
2. **Reusabilitas Kode:** Kode yang ditulis dalam fungsi dapat dipanggil berkali-kali dari berbagai bagian program tanpa perlu menulis ulang.
3. **Keterbacaan:** Kode menjadi lebih mudah dibaca dan dipahami karena setiap fungsi memiliki tujuan yang jelas.
4. **Debugging:** Lebih mudah menemukan dan memperbaiki kesalahan karena masalah dapat diisolasi ke fungsi tertentu.

### 7.2 Deklarasi dan Pemanggilan Fungsi

Fungsi di Dart dideklarasikan dengan menentukan tipe data kembalian (atau `void` jika tidak ada), nama fungsi, dan daftar parameter dalam tanda kurung.

#### 7.2.1 Fungsi Tanpa Nilai Kembalian (`void`)

Fungsi ini hanya melakukan suatu tindakan dan tidak mengembalikan nilai apa pun. Digunakan dengan kata kunci `void`.



```
void sapaPengguna() {  
    print("Halo, selamat datang!");
```

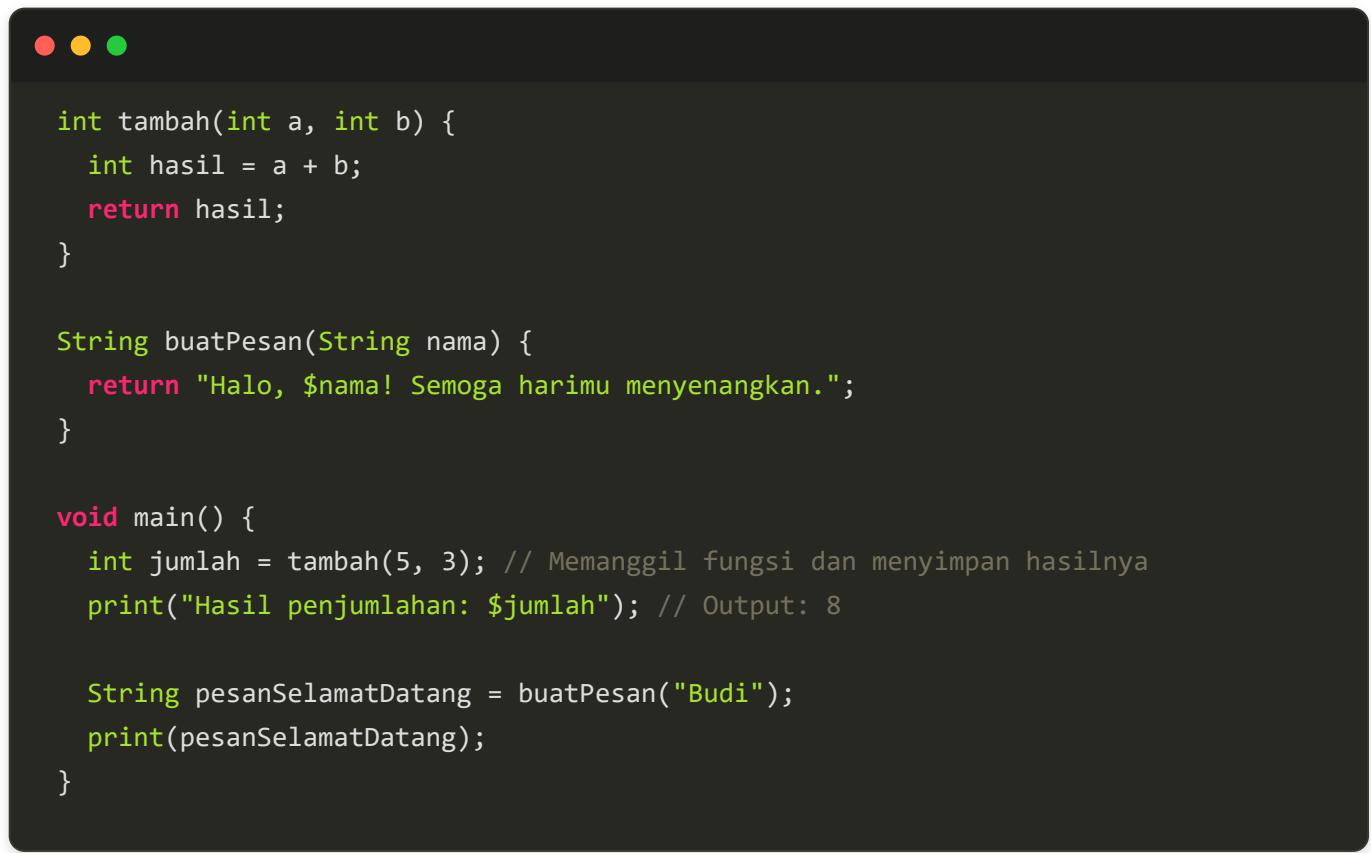
```
}
```

```
void main() {
    sapaPengguna(); // Memanggil fungsi
    sapaPengguna(); // Bisa dipanggil berkali-kali
}
```

## 7.2.2 Fungsi dengan Nilai Kembalian

Fungsi ini melakukan tindakan dan mengembalikan sebuah nilai. Tipe data nilai yang dikembalikan harus ditentukan sebelum nama fungsi. Gunakan kata kunci `return` untuk mengembalikan nilai.



```
int tambah(int a, int b) {
    int hasil = a + b;
    return hasil;
}

String buatPesan(String nama) {
    return "Halo, $nama! Semoga harimu menyenangkan.";
}

void main() {
    int jumlah = tambah(5, 3); // Memanggil fungsi dan menyimpan hasilnya
    print("Hasil penjumlahan: $jumlah"); // Output: 8

    String pesanSelamatDatang = buatPesan("Budi");
    print(pesanSelamatDatang);
}
```

## 7.3 Parameter Fungsi

Parameter adalah variabel yang digunakan untuk menerima input ke dalam fungsi. Dart mendukung beberapa jenis parameter:

### 7.3.1 Parameter Wajib (Required Parameters)

Parameter yang harus selalu disediakan saat memanggil fungsi. Jika tidak disediakan, akan terjadi error kompilasi.

```
void cetakInfoProduk(String nama, double harga) {  
    print("Produk: $nama, Harga: $$${harga.toStringAsFixed(2)}");  
}  
  
void main() {  
    cetakInfoProduk("Laptop", 1200.00);  
    // cetakInfoProduk("Keyboard"); // ERROR: The argument type 'String' can't be assigned to the parameter type 'double'.  
}
```

### 7.3.2 Parameter Opsional Posisi (Optional Positional Parameters)

Parameter ini bersifat opsional dan ditentukan berdasarkan posisinya. Mereka ditempatkan dalam kurung siku `[]` setelah parameter wajib. Jika tidak disediakan, nilainya akan `null` (jika nullable) atau menggunakan nilai default jika ada.

```
void cetakAlamat(String jalan, [String? kota, String? kodePos]) {  
    String alamatLengkap = "$jalan";  
    if (kota != null) {  
        alamatLengkap += ", $kota";  
    }  
    if (kodePos != null) {  
        alamatLengkap += ", $kodePos";  
    }  
    print(alamatLengkap);  
}  
  
void main() {  
    cetakAlamat("Jl. Merdeka No. 10");  
    cetakAlamat("Jl. Sudirman No. 5", "Jakarta");  
    cetakAlamat("Jl. Gatot Subroto", "Bandung", "40123");  
}
```

### 7.3.3 Parameter Opsional Bernama (Optional Named Parameters)

Parameter ini juga opsional, tetapi ditentukan berdasarkan namanya saat memanggil fungsi. Mereka ditempatkan dalam kurung kurawal `{}`. Ini membuat panggilan fungsi lebih mudah dibaca, terutama jika ada banyak parameter.

```
void cetakDetailPengguna({String? nama, int? umur, String? email}) {  
    print("Detail Pengguna:");  
    if (nama != null) print("Nama: $nama");  
    if (umur != null) print("Umur: $umur");  
    if (email != null) print("Email: $email");  
}  
  
void main() {  
    cetakDetailPengguna(nama: "Siti");  
    cetakDetailPengguna(umur: 28, email: "siti@example.com");  
    cetakDetailPengguna(nama: "Budi", umur: 35, email: "budi@example.com");  
}
```

### 7.3.4 Parameter dengan Nilai Default

Anda dapat memberikan nilai default untuk parameter opsional (baik posisi maupun bernama). Jika parameter tidak disediakan saat pemanggilan fungsi, nilai default akan digunakan.

```
void sapa(String nama, {String pesan = "Halo", String bahasa = "Indonesia"}) {  
    print("$pesan, $nama! (${bahasa})");  
}  
  
void main() {  
    sapa("Andi"); // Output: Halo, Andi! (Indonesia)  
    sapa("Dewi", pesan: "Selamat pagi"); // Output: Selamat pagi, Dewi! (Indonesia)  
    sapa("Peter", bahasa: "English"); // Output: Halo, Peter! (English)  
    sapa("Maria", pesan: "Hola", bahasa: "Spanish"); // Output: Hola, Maria! (Spanish)  
}
```

## 7.4 Fungsi Ekspresi (`=>`)

Untuk fungsi yang hanya berisi satu ekspresi (satu baris kode), Anda dapat menggunakan sintaks fungsi ekspresi yang lebih ringkas, juga dikenal sebagai *arrow function* atau *fat arrow syntax*.

```
// Fungsi biasa  
int tambah(int a, int b) {
```

```
return a + b;  
}  
  
// Fungsi ekspresi  
int kurang(int a, int b) => a - b;  
  
void main() {  
    print("Hasil tambah: ${tambah(10, 5)}"); // Output: 15  
    print("Hasil kurang: ${kurang(10, 5)}"); // Output: 5  
}
```

## 7.5 Fungsi Anonim (Lambda)

Fungsi anonim adalah fungsi tanpa nama. Mereka sering digunakan sebagai argumen untuk fungsi lain atau ketika Anda membutuhkan fungsi sekali pakai. Fungsi anonim juga dikenal sebagai *lambda* atau *closure*.

Sintaks dasar fungsi anonim:

```
(parameter_list) { // body fungsi }
```

Atau untuk fungsi ekspresi anonim:

```
(parameter_list) => ekspresi;
```

Contoh:

```
void main() {  
    // Fungsi anonim sebagai argumen untuk forEach  
    List<String> nama = ["Alice", "Bob", "Charlie"];  
    nama.forEach((element) {  
        print("Nama: $element");  
    });  
  
    // Fungsi anonim dengan parameter dan nilai kembalian  
    var kaliDua = (int angka) => angka * 2;
```

```
print("5 dikali dua: ${kaliDua(5)}"); // Output: 10

// Fungsi anonim yang disimpan dalam variabel
Function cetakPesan = (String msg) {
    print("Pesan: $msg");
};

cetakPesan("Ini adalah pesan dari fungsi anonim.");
}
```

## Studi Kasus: Konversi Suhu

Buatlah program yang dapat mengkonversi suhu dari Celsius ke Fahrenheit dan sebaliknya. Gunakan fungsi dengan parameter opsional untuk menentukan arah konversi.

### Rumus Konversi:

- Celsius ke Fahrenheit:  $F = (C * 9/5) + 32$
- Fahrenheit ke Celsius:  $C = (F - 32) * 5/9$



```
import 'dart:io';

void main() {
    print('--- Konverter Suhu ---');

    stdout.write('Masukkan suhu: ');
    double? suhu = double.tryParse(stdin.readLineSync() ?? '');

    if (suhu == null) {
        print('Input suhu tidak valid.');
        return;
    }

    stdout.write('Konversi ke (Fahrenheit/Celsius)? [F/C]: ');
    String? arahKonversi = stdin.readLineSync()?.toUpperCase();

    double hasilKonversi;
    String satuanHasil;

    if (arahKonversi == 'F') {
        hasilKonversi = konversiSuhu(suhu, toFahrenheit: true);
        satuanHasil = 'Fahrenheit';
    } else {
        hasilKonversi = konversiSuhu(suhu, toFahrenheit: false);
        satuanHasil = 'Celsius';
    }

    print('Hasil konversi: $hasilKonversi $satuanHasil');
}
```

```
    } else if (arahKonversi == 'C') {
        hasilKonversi = konversiSuhu(suhu, toCelsius: true);
        satuanHasil = 'Celsius';
    } else {
        print('Arah konversi tidak valid. Menggunakan default: Celsius ke Fahrenheit.');
        hasilKonversi = konversiSuhu(suhu, toFahrenheit: true);
        satuanHasil = 'Fahrenheit';
    }

    print('Hasil konversi: ${hasilKonversi.toStringAsFixed(2)} $satuanHasil');
}

// Fungsi konversi suhu dengan parameter bernama opsional
double konversiSuhu(double suhuAwal, {bool toFahrenheit = false, bool toCelsius = false}) {
    if (toFahrenheit) {
        // Konversi Celsius ke Fahrenheit
        return (suhuAwal * 9 / 5) + 32;
    } else if (toCelsius) {
        // Konversi Fahrenheit ke Celsius
        return (suhuAwal - 32) * 5 / 9;
    } else {
        // Jika tidak ada arah konversi yang ditentukan, kembalikan suhu awal
        print('Peringatan: Arah konversi tidak ditentukan. Mengembalikan suhu awal.');
        return suhuAwal;
    }
}
```

### Penjelasan:

- Fungsi `konversiSuhu` menerima `suhuAwal` sebagai parameter wajib.
- Dua parameter bernama opsional, `toFahrenheit` dan `toCelsius`, digunakan untuk menentukan arah konversi. Keduanya memiliki nilai default `false`.
- Di dalam fungsi, kita memeriksa nilai `toFahrenheit` atau `toCelsius` untuk melakukan perhitungan yang sesuai.
- Di `main()`, kita meminta input suhu dan arah konversi dari pengguna, lalu memanggil fungsi `konversiSuhu` dengan parameter yang sesuai.

### Latihan:

1. Tambahkan validasi pada input `arahKonversi` agar hanya menerima 'F' atau 'C'. Jika input tidak valid, minta pengguna untuk memasukkan ulang.
2. Buat fungsi yang menghitung faktorial dari sebuah angka. (Faktorial  $5! = 5 * 4 * 3 * 2 * 1$ ).

3. Buat fungsi yang menerima sebuah List angka dan mengembalikan rata-ratanya. Gunakan fungsi anonim untuk memproses List jika diperlukan.

- **Referensi:**

- [dart.dev/language/functions](https://dart.dev/language/functions)

## Pertemuan 8: Koleksi (List, Set, Map)

### Tujuan Pembelajaran

Setelah mempelajari materi ini, mahasiswa diharapkan mampu:

- Memahami konsep dan kegunaan struktur data koleksi di Dart.
- Menggunakan **List** untuk menyimpan data berurutan.
- Menggunakan **Set** untuk menyimpan data unik tanpa urutan.
- Menggunakan **Map** untuk menyimpan pasangan kunci-nilai.
- Melakukan operasi dasar (tambah, hapus, akses, modifikasi) pada setiap jenis koleksi.

### 8.1 Pengenalan Koleksi

Dalam pemrograman, **koleksi** (atau *collections*) adalah cara untuk mengelompokkan dan mengelola banyak data dalam satu struktur. Daripada membuat banyak variabel terpisah untuk setiap item data, kita bisa menggunakan koleksi untuk menyimpan dan memanipulasi data secara efisien. Dart menyediakan beberapa jenis koleksi bawaan yang sangat berguna:

- **List:** Koleksi terurut yang memungkinkan duplikasi elemen.
- **Set:** Koleksi tidak terurut yang hanya menyimpan elemen unik (tidak ada duplikasi).
- **Map:** Koleksi yang menyimpan data dalam pasangan kunci-nilai (key-value pairs), di mana setiap kunci harus unik.

### 8.2 List

**List** adalah koleksi yang paling umum digunakan. Ini adalah kumpulan objek yang terurut, di mana setiap objek memiliki indeks numerik (dimulai dari 0) yang menunjukkan posisinya. **List** di Dart mirip dengan array di bahasa pemrograman lain.

#### 8.2.1 Membuat List

Anda bisa membuat **List** kosong atau menginisialisasinya dengan elemen awal.

```
void main() {  
    // List kosong dengan tipe data eksplisit  
    List<String> namaBuah = [];  
    print("List buah kosong: $namaBuah");  
  
    // List dengan elemen awal dan tipe data eksplisit
```

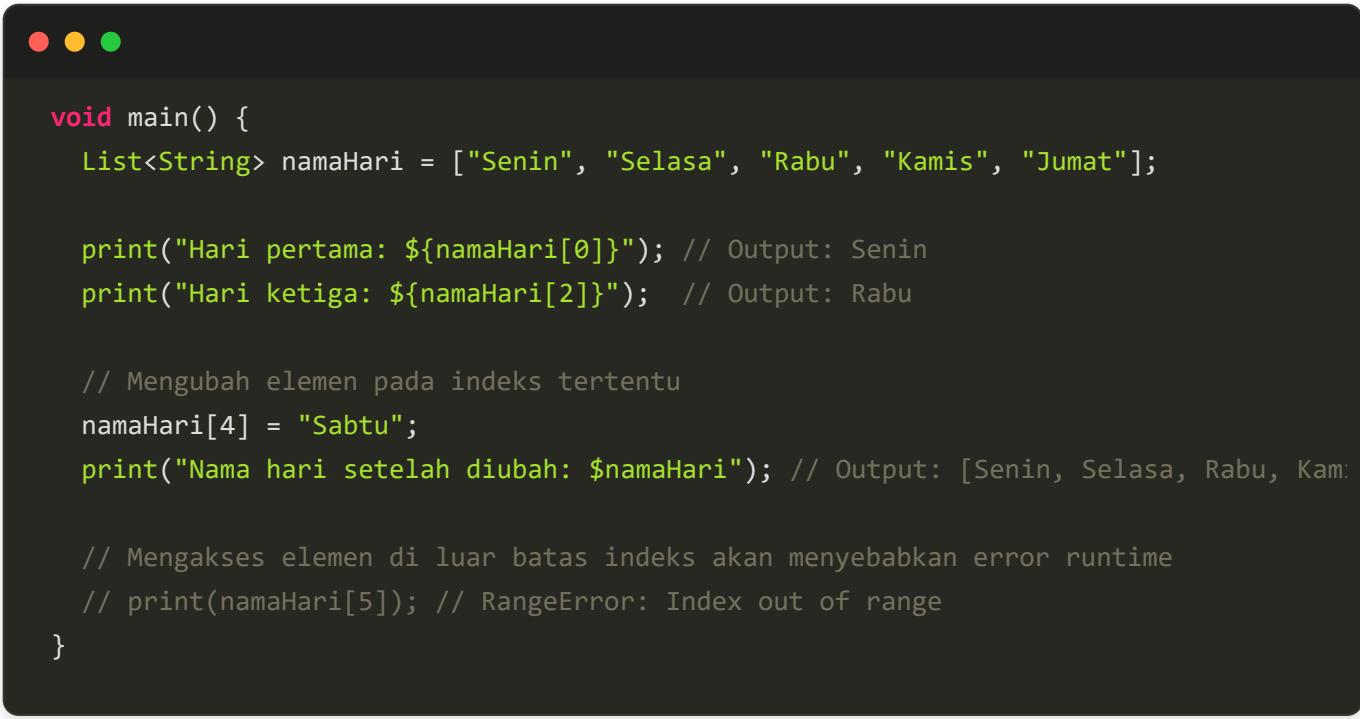
```
List<int> angkaGenap = [2, 4, 6, 8, 10];
print("Angka genap: $angkaGenap");

// List dengan inferensi tipe data (Dart akan menyimpulkan List<String>)
var namaHewan = ["Kucing", "Anjing", "Burung"];
print("Nama hewan: $namaHewan");

// List dengan tipe data campuran (tidak disarankan untuk kode produksi karena mengandung risiko runtime error)
List<dynamic> campuran = [1, "Hello", true, 3.14];
print("List campuran: $campuran");
}
```

## 8.2.2 Mengakses Elemen List

Elemen `List` diakses menggunakan indeks berbasis nol (indeks pertama adalah 0, kedua adalah 1, dan seterusnya).



```
void main() {
  List<String> namaHari = ["Senin", "Selasa", "Rabu", "Kamis", "Jumat"];

  print("Hari pertama: ${namaHari[0]}"); // Output: Senin
  print("Hari ketiga: ${namaHari[2]}"); // Output: Rabu

  // Mengubah elemen pada indeks tertentu
  namaHari[4] = "Sabtu";
  print("Nama hari setelah diubah: $namaHari"); // Output: [Senin, Selasa, Rabu, Kamis, Sabtu]

  // Mengakses elemen di luar batas indeks akan menyebabkan error runtime
  // print(namaHari[5]); // RangeError: Index out of range
}
```

## 8.2.3 Menambahkan Elemen ke List

- `add(element)` : Menambahkan satu elemen ke akhir `List`.
- `addAll(iterable)` : Menambahkan semua elemen dari koleksi lain ke akhir `List`.
- `insert(index, element)` : Menambahkan elemen pada indeks tertentu.

```
void main() {  
    List<String> daftarBelanja = ["Apel", "Susu"];  
    print("Daftar belanja awal: $daftarBelanja");  
  
    daftarBelanja.add("Roti"); // Menambahkan satu item  
    print("Setelah add Roti: $daftarBelanja"); // [Apel, Susu, Roti]  
  
    daftarBelanja.addAll(["Telur", "Gula"]); // Menambahkan beberapa item  
    print("Setelah addAll: $daftarBelanja"); // [Apel, Susu, Roti, Telur, Gula]  
  
    daftarBelanja.insert(1, "Keju"); // Menambahkan Keju di indeks 1  
    print("Setelah insert Keju: $daftarBelanja"); // [Apel, Keju, Susu, Roti, Telur, Gula]  
}
```

#### 8.2.4 Menghapus Elemen dari List

- `remove(element)` : Menghapus kemunculan pertama dari elemen yang cocok.
- `removeAt(index)` : Menghapus elemen pada indeks tertentu.
- `clear()` : Menghapus semua elemen dari `List`.

```
void main() {  
    List<String> tugas = ["Belajar Dart", "Kerjakan PR", "Olahraga", "Belajar Dart"];  
    print("Daftar tugas awal: $tugas");  
  
    tugas.remove("Belajar Dart"); // Menghapus kemunculan pertama "Belajar Dart"  
    print("Setelah remove \"Belajar Dart\": $tugas"); // [Kerjakan PR, Olahraga, Belajar Dart]  
  
    tugas.removeAt(0); // Menghapus elemen di indeks 0 ("Kerjakan PR")  
    print("Setelah removeAt(0): $tugas"); // [Olahraga, Belajar Dart]  
  
    tugas.clear(); // Menghapus semua elemen  
    print("Setelah clear: $tugas"); // []  
}
```

#### 8.2.5 Properti dan Metode List Lainnya

- `length` : Mengembalikan jumlah elemen dalam `List`.

- `isEmpty`, `isNotEmpty` : Memeriksa apakah `List` kosong atau tidak.
- `first`, `last` : Mengembalikan elemen pertama dan terakhir.
- `indexOf(element)` : Mengembalikan indeks dari elemen pertama yang cocok.
- `contains(element)` : Memeriksa apakah `List` mengandung elemen tertentu.
- `sort()` : Mengurutkan elemen dalam `List`.

```
void main() {
    List<int> angka = [5, 2, 8, 1, 9];

    print("Panjang List: ${angka.length}"); // 5
    print("List kosong? ${angka.isEmpty}"); // false
    print("Elemen pertama: ${angka.first}"); // 5
    print("Elemen terakhir: ${angka.last}"); // 9

    print("Indeks dari 8: ${angka.indexOf(8)}"); // 2
    print("Mengandung 7? ${angka.contains(7)}"); // false

    angka.sort();
    print("Setelah diurutkan: $angka"); // [1, 2, 5, 8, 9]
}
```

## 8.3 Set

`Set` adalah koleksi objek yang tidak terurut dan hanya menyimpan elemen **unik**. Ini berarti Anda tidak bisa memiliki elemen duplikat dalam sebuah `Set`. `Set` berguna ketika Anda hanya peduli tentang keberadaan suatu elemen, bukan urutannya atau berapa kali muncul.

### 8.3.1 Membuat Set

```
void main() {
    // Membuat Set kosong dengan tipe data eksplisit
    Set<String> namaUnik = {};
    print("Set nama unik kosong: $namaUnik");

    // Membuat Set dengan elemen awal (duplikat akan diabaikan)
    Set<int> angkaUnik = {1, 2, 3, 2, 1, 4};
    print("Angka unik: $angkaUnik"); // Output: {1, 2, 3, 4}
}
```

```
// Menggunakan from List untuk membuat Set dari List yang ada
List<String> daftarKata = ["apel", "jeruk", "apel", "mangga"];
Set<String> kataUnik = Set.from(daftarKata);
print("Kata unik dari daftar: $kataUnik"); // Output: {apel, jeruk, mangga}
}
```

### 8.3.2 Menambahkan dan Menghapus Elemen

- `add(element)` : Menambahkan elemen ke `Set`. Jika elemen sudah ada, tidak ada yang terjadi.
- `remove(element)` : Menghapus elemen dari `Set`.



```
void main() {
    Set<String> warna = {"Merah", "Hijau"};
    print("Warna awal: $warna");

    warna.add("Biru");
    print("Setelah add Biru: $warna"); // {Merah, Hijau, Biru}

    warna.add("Merah"); // Tidak akan ditambahkan karena sudah ada
    print("Setelah add Merah (lagi): $warna"); // {Merah, Hijau, Biru}

    warna.remove("Hijau");
    print("Setelah remove Hijau: $warna"); // {Merah, Biru}

    print("Mengandung Kuning? ${warna.contains("Kuning")}");
}
```

### 8.3.3 Operasi Set

`Set` mendukung operasi matematika seperti union (gabungan), intersection (irisan), dan difference (selisih).

- `union(other)` : Mengembalikan `Set` baru yang berisi semua elemen dari kedua `Set`.
- `intersection(other)` : Mengembalikan `Set` baru yang berisi elemen yang ada di kedua `Set`.
- `difference(other)` : Mengembalikan `Set` baru yang berisi elemen yang ada di `Set` ini tetapi tidak ada di `other`.



```
void main() {
```

```
Set<int> setA = {1, 2, 3, 4};  
Set<int> setB = {3, 4, 5, 6};  
  
print("Set A: $setA");  
print("Set B: $setB");  
  
// Union (gabungan)  
print("Union (A U B): ${setA.union(setB)}"); // {1, 2, 3, 4, 5, 6}  
  
// Intersection (irisan)  
print("Intersection (A ^ B): ${setA.intersection(setB)}"); // {3, 4}  
  
// Difference (selisih)  
print("Difference (A - B): ${setA.difference(setB)}"); // {1, 2}  
print("Difference (B - A): ${setB.difference(setA)}"); // {5, 6}  
}
```

## 8.4 Map

Map adalah koleksi yang menyimpan data dalam pasangan **kunci-nilai** (key-value pairs). Setiap kunci dalam Map harus unik, dan kunci digunakan untuk mengakses nilai yang terkait. Map mirip dengan kamus (dictionary) di Python atau objek di JavaScript.

### 8.4.1 Membuat Map

```
void main() {  
    // Membuat Map kosong dengan tipe kunci dan nilai eksplisit  
    Map<String, int> nilaiSiswa = {};  
    print("Map nilai siswa kosong: $nilaiSiswa");  
  
    // Membuat Map dengan elemen awal  
    Map<String, String> kamus = {  
        "apple": "apel",  
        "banana": "pisang",  
        "cat": "kucing",  
    };  
    print("Kamus: $kamus");  
  
    // Map dengan inferensi tipe data  
    var inventori = {  
        "Laptop": 5,  
    };
```

```
    "Mouse": 10,  
    "Keyboard": 7,  
};  
print("Inventori: $inventori");  
}
```

#### 8.4.2 Mengakses Elemen Map

Elemen `Map` diakses menggunakan kuncinya dalam kurung siku `[]`.

```
void main() {  
  Map<String, String> kamus = {  
    "apple": "apel",  
    "banana": "pisang",  
    "cat": "kucing",  
  };  
  
  print("Arti apple: ${kamus["apple"]}"); // Output: apel  
  print("Arti dog: ${kamus["dog"]}"); // Output: null (karena kunci tidak ada)  
  
  // Mengubah nilai yang terkait dengan kunci tertentu  
  kamus["cat"] = "kucing domestik";  
  print("Kamus setelah diubah: $kamus"); // {apple: apel, banana: pisang, cat: kucing}  
  
  // Menambahkan pasangan kunci-nilai baru  
  kamus["dog"] = "anjing";  
  print("Kamus setelah ditambah: $kamus"); // {apple: apel, banana: pisang, cat: kucing, dog: anjing}  
}
```

#### 8.4.3 Menghapus Elemen dari Map

- `remove(key)` : Menghapus pasangan kunci-nilai berdasarkan kunci.

```
void main() {  
  Map<String, int> stokBarang = {  
    "pensil": 10,  
    "buku": 25,  
    "penghapus": 5,  
  };  
  
  print("Stok barang sebelum dihapus: $stokBarang"); // {pensil: 10, buku: 25, penghapus: 5}  
  
  // Menghapus kunci "pensil"  
  stokBarang.remove("pensil");  
  
  print("Stok barang setelah dihapus: $stokBarang"); // {buku: 25, penghapus: 5}
```

```
};

print("Stok awal: $stokBarang");

stokBarang.remove("pensil");
print("Setelah remove pensil: $stokBarang"); // {buku: 25, penghapus: 5}

// Properti dan metode Map lainnya
print("Ukuran Map: ${stokBarang.length}"); // 2
print("Map kosong? ${stokBarang.isEmpty}"); // false
print("Mengandung kunci buku? ${stokBarang.containsKey("buku")}");
print("Mengandung nilai 25? ${stokBarang.containsValue(25)}"); // true

print("Semua kunci: ${stokBarang.keys}"); // (buku, penghapus)
print("Semua nilai: ${stokBarang.values}"); // (25, 5)

stokBarang.clear();
print("Setelah clear: $stokBarang"); // {}

}
```

## Studi Kasus: Sistem Manajemen Inventori Sederhana

Buatlah program konsol sederhana untuk mengelola inventori barang menggunakan **Map**. Program harus bisa menambah barang baru, melihat daftar barang, dan memperbarui stok barang.

```
import 'dart:io';

Map<String, int> inventori = {};

void main() {
    bool running = true;
    while (running) {
        print('\n--- Sistem Manajemen Inventori ---');
        print('1. Tambah Barang Baru');
        print('2. Lihat Inventori');
        print('3. Perbarui Stok Barang');
        print('4. Hapus Barang');
        print('5. Keluar');
        stdout.write('Pilih opsi: ');
        String? pilihan = stdin.readLineSync();

        switch (pilihan) {
```

```
case '1':
    tambahBarang();
    break;
case '2':
    lihatInventori();
    break;
case '3':
    perbaruiStok();
    break;
case '4':
    hapusBarang();
    break;
case '5':
    running = false;
    print('Terima kasih telah menggunakan sistem inventori.');
    break;
default:
    print('Opsi tidak valid. Silakan coba lagi.');
}

}
}

void tambahBarang() {
    stdout.write('Masukkan nama barang: ');
    String? nama = stdin.readLineSync();
    if (nama == null || nama.isEmpty) {
        print('Nama barang tidak boleh kosong.');
        return;
    }
    nama = nama.toLowerCase(); // Standardisasi nama barang

    if (inventori.containsKey(nama)) {
        print('Barang \'$nama\' sudah ada dalam inventori. Gunakan opsi 3 untuk memperbaikinya.');
        return;
    }

    stdout.write('Masukkan jumlah stok: ');
    String? inputStok = stdin.readLineSync();
    int? stok = int.tryParse(inputStok ?? '');

    if (stok == null || stok < 0) {
        print('Jumlah stok tidak valid. Harus angka positif.');
        return;
    }
}
```

```
inventori[nama] = stok;
print('Barang \'$nama\' dengan stok $stok berhasil ditambahkan.');
}

void lihatInventori() {
    if (inventori.isEmpty) {
        print('Inventori kosong.');
        return;
    }
    print('\n--- Daftar Inventori ---');
    inventori.forEach((nama, stok) {
        print('${nama[0].toUpperCase()}${nama.substring(1)}: $stok');
    });
}

void perbaruiStok() {
    stdout.write('Masukkan nama barang yang stoknya ingin diperbarui: ');
    String? nama = stdin.readLineSync();
    if (nama == null || nama.isEmpty) {
        print('Nama barang tidak boleh kosong.');
        return;
    }
    nama = nama.toLowerCase();

    if (!inventori.containsKey(nama)) {
        print('Barang \'$nama\' tidak ditemukan dalam inventori.');
        return;
    }

    stdout.write('Masukkan jumlah stok baru: ');
    String? inputStok = stdin.readLineSync();
    int? stokBaru = int.tryParse(inputStok ?? '');

    if (stokBaru == null || stokBaru < 0) {
        print('Jumlah stok tidak valid. Harus angka positif.');
        return;
    }

    inventori[nama] = stokBaru;
    print('Stok barang \'$nama\' berhasil diperbarui menjadi $stokBaru.');
}

void hapusBarang() {
```

```
stdout.write('Masukkan nama barang yang ingin dihapus: ');
String? nama = stdin.readLineSync();
if (nama == null || nama.isEmpty) {
    print('Nama barang tidak boleh kosong.');
    return;
}
nama = nama.toLowerCase();

if (inventori.containsKey(nama)) {
    inventori.remove(nama);
    print('Barang \'$nama\' berhasil dihapus dari inventori.');
} else {
    print('Barang \'$nama\' tidak ditemukan dalam inventori.');
}
}
```

### Penjelasan:

- Program menggunakan `Map<String, int>` bernama `inventori` untuk menyimpan nama barang (String) dan jumlah stoknya (int).
- Fungsi `main` berisi loop utama yang menampilkan menu dan memproses pilihan pengguna menggunakan `switch`.
- Fungsi-fungsi terpisah (`tambahBarang`, `lihatInventori`, `perbaruiStok`, `hapusBarang`) mengelola operasi pada `inventori`.
- `nama.toLowerCase()` digunakan untuk memastikan nama barang tidak case-sensitive.

### Latihan:

1. Modifikasi program agar saat melihat inventori, barang diurutkan berdasarkan nama.
2. Tambahkan fitur untuk mencari barang berdasarkan nama dan menampilkan detailnya.
3. Implementasikan fitur untuk menambah stok barang yang sudah ada (bukan mengganti total stok, tapi menambahkannya).

- **Referensi:**

- [dart.dev/language/collections](https://dart.dev/language/collections)

## Pertemuan 9: Pemrograman Berorientasi Objek (OOP) - Kelas dan Objek

### Tujuan Pembelajaran

Setelah mempelajari materi ini, mahasiswa diharapkan mampu:

- Memahami konsep dasar Pemrograman Berorientasi Objek (OOP).
- Mendefinisikan kelas (class) sebagai blueprint untuk objek.
- Membuat objek (instance) dari sebuah kelas.
- Mendeklarasikan properti (fields) dan metode (methods) dalam sebuah kelas.
- Memahami dan menggunakan constructor untuk menginisialisasi objek.

### 9.1 Pengenalan Pemrograman Berorientasi Objek (OOP)

**Pemrograman Berorientasi Objek (OOP)** adalah paradigma pemrograman yang didasarkan pada konsep "objek". Dalam OOP, program dirancang dengan mengorganisir data dan perilaku ke dalam entitas yang disebut objek. Objek ini merupakan representasi dari entitas di dunia nyata (misalnya, mobil, siswa, buku, akun bank).

#### Konsep Utama OOP:

1. **Enkapsulasi (Encapsulation):** Membungkus data (properti) dan kode (metode) yang beroperasi pada data tersebut menjadi satu unit (objek). Ini juga melibatkan penyembunyian detail implementasi internal dari dunia luar.
2. **Abstraksi (Abstraction):** Menyajikan hanya informasi yang relevan kepada pengguna dan menyembunyikan detail yang tidak perlu. Fokus pada "apa" yang dilakukan objek, bukan "bagaimana" objek melakukannya.
3. **Pewarisan (Inheritance):** Mekanisme di mana sebuah kelas baru (subclass) dapat mewarisi properti dan metode dari kelas yang sudah ada (superclass). Ini mempromosikan penggunaan kembali kode.
4. **Polimorfisme (Polymorphism):** Kemampuan objek untuk mengambil banyak bentuk. Ini memungkinkan objek dari kelas yang berbeda untuk diperlakukan sebagai objek dari kelas yang sama melalui interface atau inheritance.

Dart adalah bahasa pemrograman berorientasi objek, yang berarti semua yang Anda gunakan di Dart (angka, string, fungsi, bahkan `null`) adalah objek.

### 9.2 Kelas (Class) dan Objek (Object)

#### 9.2.1 Kelas (Class)

**Kelas** adalah cetak biru (blueprint) atau template untuk membuat objek. Kelas mendefinisikan struktur dan perilaku yang akan dimiliki oleh semua objek yang dibuat dari kelas tersebut. Kelas tidak mengalokasikan memori; ia hanya mendefinisikan bagaimana objek akan terlihat dan bertindak.

Contoh analogi: Kelas `Mobil` adalah cetak biru yang mendefinisikan bahwa setiap mobil memiliki merek, model, tahun, dan kemampuan untuk menyalakan mesin atau bergerak maju.

### 9.2.2 Objek (Object)

**Objek** adalah instansi (instance) dari sebuah kelas. Ketika Anda membuat objek dari sebuah kelas, Anda sebenarnya menciptakan sebuah entitas konkret yang memiliki properti dan perilaku yang didefinisikan oleh kelas tersebut. Setiap objek memiliki identitas unik dan menyimpan datanya sendiri.

Contoh analogi: `Mobil saya` (Toyota Camry 2020) dan `Mobil teman` (Honda Civic 2022) adalah objek-objek yang berbeda dari kelas `Mobil`. Keduanya memiliki merek, model, dan tahun, tetapi nilai-nilai tersebut berbeda untuk setiap objek.

## 9.3 Properti (Fields) dan Metode (Methods)

Di dalam sebuah kelas, kita mendefinisikan:

- **Properti (Fields/Attributes):** Variabel yang menyimpan data atau karakteristik dari objek. Mereka mendefinisikan "apa" yang dimiliki objek.
- **Metode (Methods/Behaviors):** Fungsi yang mendefinisikan tindakan atau perilaku yang dapat dilakukan oleh objek. Mereka mendefinisikan "apa" yang dapat dilakukan objek.

```
void main() {  
    // Membuat objek (instance) dari kelas Mobil  
    Mobil mobilSaya = Mobil("Toyota", "Camry", 2020); // Memanggil constructor  
  
    // Mengakses properti objek  
    print("Merek mobil saya: ${mobilSaya.merek}");  
    print("Model mobil saya: ${mobilSaya.model}");  
    print("Tahun mobil saya: ${mobilSaya.tahun}");  
  
    // Memanggil metode objek  
    mobilSaya.nyalakanMesin();  
    mobilSaya.maju();  
  
    // Membuat objek lain  
    Mobil mobilTeman = Mobil("Honda", "Civic", 2022);  
    mobilTeman.nyalakanMesin();  
    mobilTeman.berhenti();  
}  
  
// Definisi Kelas Mobil  
class Mobil {
```

```
// Properti (Fields/Attributes)
String merek; // Contoh: "Toyota"
String model; // Contoh: "Camry"
int tahun; // Contoh: 2020

// Constructor (akan dibahas lebih lanjut di bagian selanjutnya)
// Ini adalah cara untuk menginisialisasi properti saat objek dibuat
Mobil(this.merek, this.model, this.tahun);

// Metode (Methods/Behaviors)
void nyalakanMesin() {
    print("Mesin ${merek} ${model} dinyalakan. Vroom!");
}

void maju() {
    print("${merek} ${model} sedang bergerak maju.");
}

void berhenti() {
    print("${merek} ${model} berhenti.");
}
```

## 9.4 Constructor

**Constructor** adalah metode khusus dalam sebuah kelas yang dipanggil secara otomatis saat sebuah objek baru dari kelas tersebut dibuat. Tujuan utama constructor adalah untuk menginisialisasi properti objek dengan nilai awal.

### 9.4.1 Default Constructor

Jika Anda tidak mendeklarasikan constructor secara eksplisit dalam sebuah kelas, Dart akan secara otomatis menyediakan *default constructor* tanpa argumen. Constructor default ini akan memanggil constructor tanpa argumen dari superclass.



```
void meong() {
    print("$nama meong-meong!");
}

void main() {
    Kucing myCat = Kucing(); // Memanggil default constructor
    myCat.nama = "Kitty";
    myCat.meong();
}
```

#### 9.4.2 Constructor Parameterized

Anda dapat membuat constructor yang menerima parameter untuk menginisialisasi properti objek saat pembuatan. Dart memiliki sintaks yang ringkas untuk ini.

```
● ● ●

class Buku {
    String judul;
    String penulis;
    int tahunTerbit;

    // Constructor parameterized
    // Sintaks singkat: this.property akan menginisialisasi properti dengan nama yang :
    Buku(this.judul, this.penulis, this.tahunTerbit);

    void tampilkanInfoBuku() {
        print("Judul: $judul");
        print("Penulis: $penulis");
        print("Tahun Terbit: $tahunTerbit");
    }
}

void main() {
    Buku buku1 = Buku("Filosofi Teras", "Henry Manampiring", 2018);
    buku1.tampilkanInfoBuku();

    Buku buku2 = Buku("Atomic Habits", "James Clear", 2018);
    buku2.tampilkanInfoBuku();
}
```

#### 9.4.3 Named Constructor

Sebuah kelas dapat memiliki lebih dari satu constructor dengan menggunakan *named constructor*. Ini sangat berguna ketika Anda ingin menyediakan beberapa cara yang berbeda untuk membuat objek dari kelas yang sama, masing-masing dengan tujuan yang jelas.

```
● ● ●

class Point {
    double x, y;

    // Constructor utama
    Point(this.x, this.y);

    // Named constructor untuk membuat titik di origin (0,0)
    Point.origin() : x = 0.0, y = 0.0;

    // Named constructor untuk membuat titik dari koordinat 3D (mengabaikan z)
    Point.from3D({required double x, required double y, double z = 0.0}) :
        this.x = x,
        this.y = y;

    void display() {
        print("Point: ($x, $y)");
    }
}

void main() {
    Point p1 = Point(10.0, 20.0);
    p1.display(); // Output: Point: (10.0, 20.0)

    Point p2 = Point.origin();
    p2.display(); // Output: Point: (0.0, 0.0)

    Point p3 = Point.from3D(x: 5.0, y: 7.0, z: 1.0);
    p3.display(); // Output: Point: (5.0, 7.0)
}
```

#### 9.4.4 Factory Constructor

*Factory constructor* adalah constructor yang tidak selalu membuat instance baru dari kelasnya. Ia bisa mengembalikan instance yang sudah ada, atau instance dari subclass. Factory constructor ditandai dengan kata kunci `factory`.

```
class Logger {  
    final String name;  
    static final Map<String, Logger> _cache = <String, Logger>{};  
  
    factory Logger(String name) {  
        if (_cache.containsKey(name)) {  
            return _cache[name]!;  
        } else {  
            final logger = Logger._internal(name);  
            _cache[name] = logger;  
            return logger;  
        }  
    }  
  
    Logger._internal(this.name); // Private constructor  
  
    void log(String msg) {  
        print("[ " + name + " ] " + msg);  
    }  
}  
  
void main() {  
    var logger1 = Logger("UI");  
    logger1.log("User clicked button.");  
  
    var logger2 = Logger("UI"); // Mengembalikan instance yang sama dengan logger1  
    logger2.log("User navigated to new screen.");  
  
    print(identical(logger1, logger2)); // Output: true  
  
    var logger3 = Logger("Database");  
    logger3.log("Data fetched successfully.");  
    print(identical(logger1, logger3)); // Output: false  
}
```

### Studi Kasus: Sistem Manajemen Mahasiswa Sederhana

Buatlah kelas `Mahasiswa` yang memiliki properti `nama`, `nim`, dan `jurusan`. Tambahkan metode untuk menampilkan informasi mahasiswa dan constructor untuk menginisialisasi objek `Mahasiswa`.

```
class Mahasiswa {  
    String nama;  
    String nim;  
    String jurusan;  
  
    // Constructor untuk menginisialisasi properti  
    Mahasiswa(this.nama, this.nim, this.jurusan);  
  
    // Metode untuk menampilkan informasi mahasiswa  
    void tampilanInfo() {  
        print("\n--- Informasi Mahasiswa ---");  
        print("Nama: $nama");  
        print("NIM: $nim");  
        print("Jurusan: $jurusan");  
    }  
  
    // Named constructor untuk mahasiswa baru tanpa jurusan awal  
    Mahasiswa.mahasiswaBaru(this.nama, this.nim) : jurusan = "Belum Ditentukan";  
}  
  
void main() {  
    // Membuat objek mahasiswa menggunakan constructor utama  
    Mahasiswa mhs1 = Mahasiswa("Budi Santoso", "12345678", "Teknik Informatika");  
    mhs1.tampilanInfo();  
  
    // Membuat objek mahasiswa menggunakan named constructor  
    Mahasiswa mhs2 = Mahasiswa.mahasiswaBaru("Siti Aminah", "87654321");  
    mhs2.tampilanInfo();  
  
    // Mengubah jurusan mhs2  
    mhs2.jurusan = "Sistem Informasi";  
    mhs2.tampilanInfo();  
}
```

### Penjelasan:

- Kelas `Mahasiswa` memiliki tiga properti: `nama`, `nim`, dan `jurusan`.
- Constructor `Mahasiswa(this.nama, this.nim, this.jurusan)` digunakan untuk membuat objek dengan semua properti diinisialisasi.
- Metode `tampilanInfo()` mencetak detail mahasiswa ke konsol.

- Named constructor `Mahasiswa.mahasiswaBaru` memungkinkan pembuatan objek `Mahasiswa` hanya dengan nama dan NIM, dengan jurusan default "Belum Ditentukan".

**Latihan:**

1. Tambahkan properti `ipk` (indeks prestasi kumulatif) ke kelas `Mahasiswa` (tipe `double`).
2. Buat metode di kelas `Mahasiswa` untuk memperbarui `ipk`.
3. Buat named constructor `Mahasiswa.dariDataString(String data)` yang menerima string seperti "Nama:Budi,NIM:123,Jurusan:TI" dan mengurai data tersebut untuk membuat objek `Mahasiswa`.

**• Referensi:**

- [dart.dev/language/classes](https://dart.dev/language/classes)

## Pertemuan 10: Pemrograman Berorientasi Objek (OOP) - Inheritance dan Polymorphism

### Tujuan Pembelajaran

Setelah mempelajari materi ini, mahasiswa diharapkan mampu:

- Memahami konsep pewarisan (inheritance) dalam OOP Dart.
- Menggunakan kata kunci `extends` untuk membuat subclass.
- Memahami dan menggunakan kata kunci `super`.
- Melakukan *method overriding*.
- Memahami konsep polimorfisme dan implementasinya di Dart.
- Mendefinisikan dan menggunakan *abstract classes* dan *abstract methods*.

#### 10.1 Pewarisan (Inheritance)

**Pewarisan (Inheritance)** adalah salah satu pilar utama OOP yang memungkinkan sebuah kelas baru (disebut **subclass**, **child class**, atau **derived class**) untuk mewarisi properti (fields) dan metode (methods) dari kelas yang sudah ada (disebut **superclass**, **parent class**, atau **base class**). Ini adalah mekanisme yang kuat untuk mencapai *reusabilitas kode* dan membangun hierarki kelas yang logis.

#### Konsep Dasar:

- `extends` **keyword**: Digunakan untuk menunjukkan bahwa sebuah kelas mewarisi dari kelas lain.
- `super` **keyword**: Digunakan untuk merujuk ke anggota (properti atau metode) dari superclass.

#### Contoh Sederhana Inheritance

Misalkan kita memiliki kelas `Hewan` dengan properti `nama` dan metode `makan()`. Kita bisa membuat kelas `Kucing` dan `Anjing` yang mewarisi dari `Hewan`, sehingga mereka secara otomatis memiliki properti `nama` dan metode `makan()`, dan kita bisa menambahkan perilaku spesifik mereka (misalnya, `bersuara()`).

```
class Hewan {  
    String nama;  
  
    Hewan(this.nama);  
  
    void makan() {  
        print("$nama sedang makan.");  
    }  
}
```

```
    }
}

class Kucing extends Hewan {
    // Kucing mewarisi properti 'nama' dan metode 'makan()' dari Hewan
    Kucing(String nama) : super(nama); // Memanggil constructor superclass

    void bersuara() {
        print("$nama meong-meong!");
    }
}

class Anjing extends Hewan {
    Anjing(String nama) : super(nama);

    void bersuara() {
        print("$nama guk-guk!");
    }
}

void main() {
    Kucing kucingSaya = Kucing("Kitty");
    kucingSaya.makan();      // Metode dari superclass Hewan
    kucingSaya.bersuara();   // Metode spesifik Kucing

    Anjing anjingSaya = Anjing("Buddy");
    anjingSaya.makan();     // Metode dari superclass Hewan
    anjingSaya.bersuara();  // Metode spesifik Anjing
}
```

#### Penjelasan `super` :

Dalam contoh di atas, `Kucing(String nama) : super(nama);` adalah cara untuk memanggil constructor dari superclass (`Hewan`). Ketika sebuah objek `Kucing` dibuat, pertama-tama constructor `Hewan` akan dipanggil untuk menginisialisasi properti `nama` yang diwarisi, baru kemudian constructor `Kucing` menyelesaikan inisialisasinya.

## 10.2 Method Overriding

**Method Overriding** adalah kemampuan subclass untuk menyediakan implementasi yang berbeda untuk metode yang sudah didefinisikan di superclass-nya. Ini memungkinkan subclass untuk mengubah atau memperluas perilaku metode yang diwarisi.

Untuk melakukan *override* metode, Anda harus menggunakan anotasi `@override` di Dart. Ini adalah praktik yang baik karena membantu kompiler mendeteksi kesalahan jika Anda salah mengeja nama metode atau tanda tangan (signature) metode tidak cocok.

```
● ● ●

class Hewan {
    String nama;

    Hewan(this.nama);

    void makan() {
        print("$nama sedang makan makanan hewan.");
    }
}

class Kucing extends Hewan {
    Kucing(String nama) : super(nama);

    @override // Menandakan bahwa metode ini meng-override metode di superclass
    void makan() {
        print("$nama sedang makan ikan."); // Implementasi yang berbeda
    }

    void bersuara() {
        print("$nama meong-meong!");
    }
}

void main() {
    Hewan hewanUmum = Hewan("Binatang");
    hewanUmum.makan(); // Output: Binatang sedang makan makanan hewan.

    Kucing kucingSaya = Kucing("Kitty");
    kucingSaya.makan(); // Output: Kitty sedang makan ikan. (Metode yang di-override d:
}
```

Anda juga bisa memanggil implementasi metode dari superclass di dalam metode yang di-override menggunakan `super.methodName()`.

```
class Kucing extends Hewan {  
    Kucing(String nama) : super(nama);  
  
    @override  
    void makan() {  
        super.makan(); // Memanggil metode makan() dari kelas Hewan  
        print("Tapi $nama lebih suka makan ikan.");  
    }  
}  
  
void main() {  
    Kucing kucingSaya = Kucing("Kitty");  
    kucingSaya.makan();  
    // Output:  
    // Kitty sedang makan makanan hewan.  
    // Tapi Kitty lebih suka makan ikan.  
}
```

### 10.3 Polimorfisme (Polymorphism)

**Polimorfisme** berarti "banyak bentuk". Dalam OOP, polimorfisme memungkinkan objek dari kelas yang berbeda untuk diperlakukan sebagai objek dari kelas yang sama melalui superclass atau interface yang umum. Ini berarti Anda dapat menulis kode yang lebih fleksibel dan umum yang dapat bekerja dengan berbagai jenis objek.

#### Konsep Dasar:

- Sebuah variabel bertipe superclass dapat menampung objek dari subclass-nya.
- Ketika metode dipanggil pada variabel tersebut, metode yang diimplementasikan oleh objek sebenarnya (subclass) yang akan dieksekusi (ini disebut *dynamic dispatch*).

```
class Hewan {  
    void bersuara() {  
        print("Hewan bersuara.");  
    }  
}  
  
class Kucing extends Hewan {  
    @override  
    void bersuara() {
```

```
        print("Meong!");
    }
}

class Anjing extends Hewan {
    @override
    void bersuara() {
        print("Guk guk!");
    }
}

void main() {
    // Variabel bertipe superclass (Hewan) menampung objek subclass
    Hewan hewan1 = Kucing();
    Hewan hewan2 = Anjing();
    Hewan hewan3 = Hewan();

    // Memanggil metode bersuara() pada variabel bertipe Hewan
    // Metode yang dieksekusi adalah implementasi dari objek sebenarnya
    hewan1.bersuara(); // Output: Meong!
    hewan2.bersuara(); // Output: Guk guk!
    hewan3.bersuara(); // Output: Hewan bersuara.

    // Contoh lain dengan List
    List<Hewan> daftarHewan = [];
    daftarHewan.add(Kucing());
    daftarHewan.add(Anjing());
    daftarHewan.add(Hewan());

    print("\nSuara dari daftar hewan:");
    for (var hewan in daftarHewan) {
        hewan.bersuara(); // Setiap objek memanggil implementasi bersuara() miliknya sendiri
    }
    // Output:
    // Meong!
    // Guk guk!
    // Hewan bersuara.
}
```

Polimorfisme sangat penting untuk desain perangkat lunak yang fleksibel dan mudah diperluas. Anda dapat menulis fungsi yang menerima tipe superclass, dan fungsi tersebut akan bekerja dengan benar untuk semua subclass-nya.

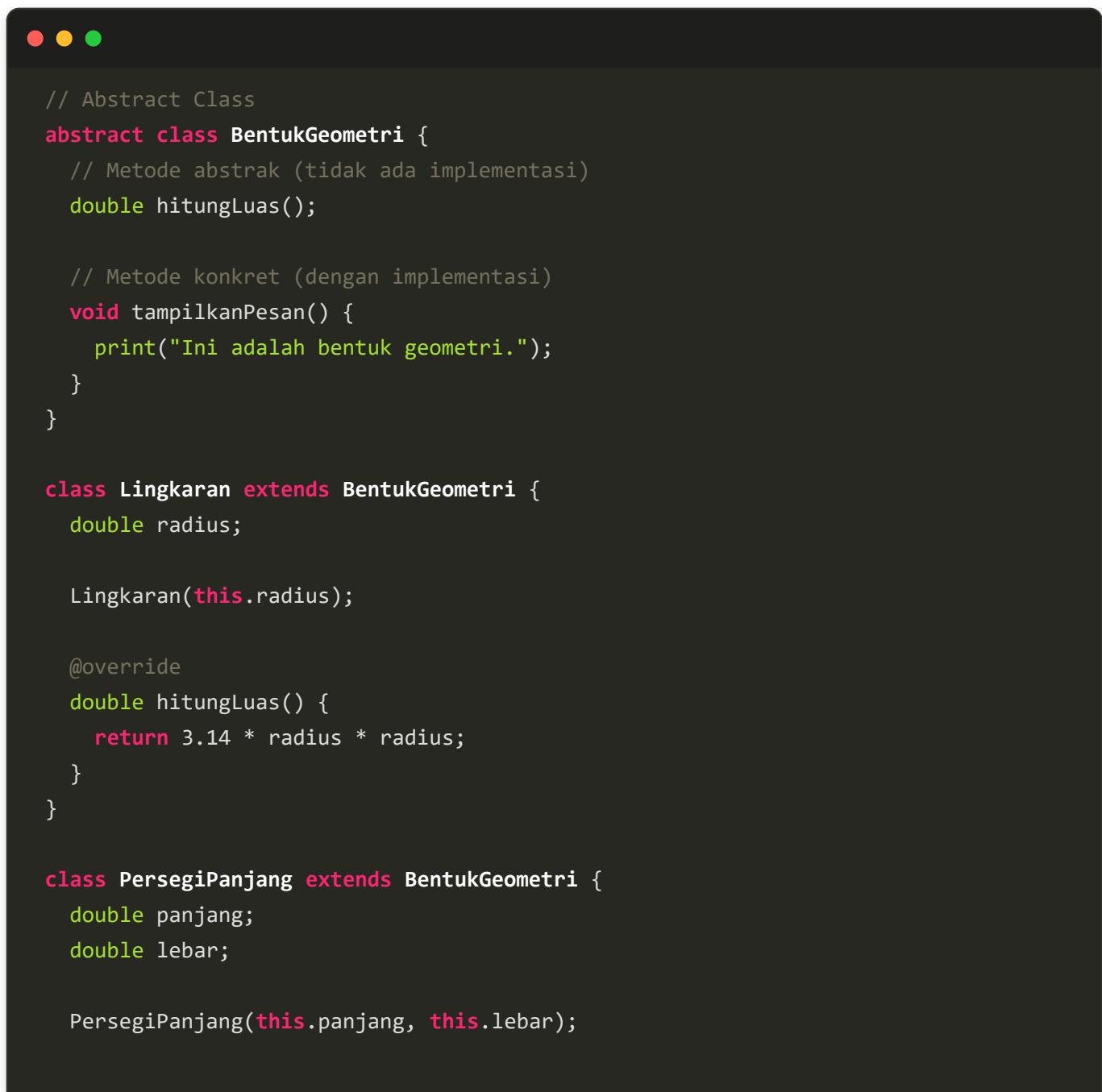
## 10.4 Abstract Classes dan Abstract Methods

**Abstract Class** adalah kelas yang tidak dapat diinstansiasi secara langsung (Anda tidak bisa membuat objek dari abstract class). Abstract class dirancang untuk menjadi blueprint atau kerangka dasar bagi subclass-nya. Mereka dapat memiliki metode konkret (dengan implementasi) dan metode abstrak (tanpa implementasi).

**Abstract Method** adalah metode yang dideklarasikan dalam abstract class tetapi tidak memiliki implementasi. Subclass yang mewarisi abstract class **wajib** mengimplementasikan semua metode abstraknya.

### Kapan Menggunakan Abstract Class?

- Ketika Anda ingin mendefinisikan sebuah antarmuka umum untuk sekelompok kelas yang terkait, tetapi Anda tidak ingin kelas dasar itu sendiri diinstansiasi.
- Ketika Anda ingin memaksa subclass untuk menyediakan implementasi spesifik untuk metode tertentu.



```
// Abstract Class
abstract class BentukGeometri {
    // Metode abstrak (tidak ada implementasi)
    double hitungLuas();

    // Metode konkret (dengan implementasi)
    void tampilkanPesan() {
        print("Ini adalah bentuk geometri.");
    }
}

class Lingkaran extends BentukGeometri {
    double radius;

    Lingkaran(this.radius);

    @override
    double hitungLuas() {
        return 3.14 * radius * radius;
    }
}

class PersegiPanjang extends BentukGeometri {
    double panjang;
    double lebar;

    PersegiPanjang(this.panjang, this.lebar);
}
```

```

@Override
double hitungLuas() {
    return panjang * lebar;
}

void main() {
    // BentukGeometri bentuk = BentukGeometri(); // ERROR: Abstract classes can't be instantiated

    Lingkaran lingkaran = Lingkaran(5.0);
    print("Luas Lingkaran: ${lingkaran.hitungLuas()}");
    lingkaran.tampilkanPesan();

    PersegiPanjang persegi = PersegiPanjang(4.0, 6.0);
    print("Luas Persegi Panjang: ${persegi.hitungLuas()}");
    persegi.tampilkanPesan();

    // Contoh polimorfisme dengan abstract class
    List<BentukGeometri> daftarBentuk = [];
    daftarBentuk.add(Lingkaran(7.0));
    daftarBentuk.add(PersegiPanjang(3.0, 8.0));

    print("\nMenghitung luas dari daftar bentuk:");
    for (var bentuk in daftarBentuk) {
        print("Luas: ${bentuk.hitungLuas()}");
    }
}

```

**Penjelasan:**

- `BentukGeometri` adalah `abstract class` karena memiliki metode abstrak `hitungLuas()`. Ini berarti setiap subclass dari `BentukGeometri` harus menyediakan implementasi untuk `hitungLuas()`.
- `Lingkaran` dan `PersegiPanjang` adalah subclass yang mengimplementasikan `hitungLuas()` sesuai dengan perhitungan luas masing-masing.
- Kita tidak bisa membuat objek langsung dari `BentukGeometri`, tetapi kita bisa menggunakan tipe `BentukGeometri` untuk menampung objek dari subclass-nya (polimorfisme).

**Studi Kasus: Sistem Penggajian Karyawan**

Buatlah sistem penggajian sederhana dengan konsep inheritance dan polymorphism. Ada kelas dasar `Karyawan` dan subclass `KaryawanTetap` serta `KaryawanKontrak`. Setiap jenis karyawan memiliki cara perhitungan gaji yang berbeda.

```
// Kelas dasar abstract
abstract class Karyawan {
    String nama;
    String id;

    Karyawan(this.nama, this.id);

    // Metode abstrak untuk menghitung gaji
    double hitungGaji();

    void tampilanInfo() {
        print("Nama: $nama, ID: $id");
    }
}

// Subclass Karyawan Tetap
class KaryawanTetap extends Karyawan {
    double gajiPokok;
    double tunjangan;

    KaryawanTetap(String nama, String id, this.gajiPokok, this.tunjangan)
        : super(nama, id);

    @override
    double hitungGaji() {
        return gajiPokok + tunjangan;
    }

    @override
    void tampilanInfo() {
        super.tampilanInfo();
        print("Tipe: Karyawan Tetap");
        print("Gaji Pokok: ${gajiPokok.toStringAsFixed(2)}");
        print("Tunjangan: ${tunjangan.toStringAsFixed(2)}");
        print("Total Gaji: ${hitungGaji().toStringAsFixed(2)}");
    }
}

// Subclass Karyawan Kontrak
class KaryawanKontrak extends Karyawan {
    double upahPerJam;
    int jamKerja;
```

```
KaryawanKontrak(String nama, String id, this.upahPerJam, this.jamKerja)
    : super(nama, id);

@Override
double hitungGaji() {
    return upahPerJam * jamKerja;
}

@Override
void tampilkanInfo() {
    super.tampilkanInfo();
    print("Tipe: Karyawan Kontrak");
    print("Upah per Jam: ${upahPerJam.toStringAsFixed(2)}");
    print("Jam Kerja: $jamKerja");
    print("Total Gaji: ${hitungGaji().toStringAsFixed(2)}");
}
}

void main() {
    List<Karyawan> daftarKaryawan = [];

    daftarKaryawan.add(KaryawanTetap("Budi", "KT001", 5000000, 1000000));
    daftarKaryawan.add(KaryawanKontrak("Siti", "KK001", 50000, 160));
    daftarKaryawan.add(KaryawanTetap("Andi", "KT002", 4500000, 800000));

    print("\n--- Laporan Gaji Karyawan ---");
    for (var karyawan in daftarKaryawan) {
        karyawan.tampilkanInfo();
        print("-----");
    }

    // Contoh penggunaan polimorfisme untuk menghitung total gaji
    double totalGajiKeseluruhan = 0;
    for (var karyawan in daftarKaryawan) {
        totalGajiKeseluruhan += karyawan.hitungGaji();
    }
    print("Total Gaji Keseluruhan Perusahaan: ${totalGajiKeseluruhan.toStringAsFixed(2)}
}
```

**Penjelasan:**

- Karyawan adalah abstract class dengan metode abstrak hitungGaji(). Ini memastikan setiap jenis karyawan harus memiliki cara perhitungan gajinya sendiri.
- KaryawanTetap dan KaryawanKontrak adalah subclass yang mengimplementasikan hitungGaji() sesuai dengan model gaji masing-masing.
- Metode tampilkanInfo() di-override di subclass untuk menampilkan detail spesifik karyawan, sambil tetap memanggil super.tampilkanInfo() untuk informasi umum.
- List<Karyawan> digunakan untuk menyimpan objek dari kedua subclass, menunjukkan polimorfisme. Kita bisa mengiterasi List ini dan memanggil hitungGaji() atau tampilkanInfo() pada setiap objek, dan Dart akan secara otomatis memanggil implementasi yang benar dari subclass masing-masing.

**Latihan:**

1. Tambahkan subclass baru, misalnya KaryawanParuhWaktu, dengan perhitungan gaji yang berbeda (misalnya, berdasarkan jam kerja dan upah per jam, tetapi dengan tunjangan tetap).
2. Modifikasi program agar bisa mencari karyawan berdasarkan ID dan menampilkan detailnya.
3. Buat sebuah fungsi yang menerima List<Karyawan> dan mengembalikan karyawan dengan gaji tertinggi.

**• Referensi:**

- [dart.dev/language/inheritance](https://dart.dev/language/inheritance)
- [dart.dev/language/abstract-classes](https://dart.dev/language/abstract-classes)

## Pertemuan 11: Pemrograman Berorientasi Objek (OOP) - Interfaces dan Mixins

### Tujuan Pembelajaran

Setelah mempelajari materi ini, mahasiswa diharapkan mampu:

- Memahami konsep *implicit interfaces* di Dart.
- Menggunakan kata kunci `implements` untuk mengimplementasikan interface.
- Memahami konsep *mixins* dan kapan menggunakan.
- Menggunakan kata kunci `with` untuk menerapkan mixin ke sebuah kelas.
- Membedakan antara *abstract class*, *interface*, dan *mixin*.

#### 11.1 Interfaces (Implicit Interfaces)

Di Dart, tidak ada kata kunci `interface` secara eksplisit seperti di Java atau C#. Sebaliknya, **setiap kelas secara implisit mendefinisikan sebuah interface**. Ini berarti Anda dapat mengimplementasikan interface dari kelas mana pun. Ketika sebuah kelas mengimplementasikan interface dari kelas lain, ia harus menyediakan implementasi untuk semua metode dan properti yang didefinisikan dalam interface tersebut.

#### Kapan Menggunakan `implements` ?

- Ketika Anda ingin sebuah kelas mematuhi kontrak tertentu, yaitu memiliki metode dan properti yang sama dengan kelas lain, tetapi tanpa mewarisi implementasinya.
- Untuk mencapai *multiple inheritance of types* (sebuah kelas dapat berperilaku seperti beberapa tipe yang berbeda).

Sintaks dasar:

```
● ● ●  
class NamaKelas implements NamaInterface {  
    // Implementasi semua metode dan properti dari NamaInterface  
}
```

Contoh:

```
● ● ●  
// Kelas ini secara implisit mendefinisikan interface Kendaraan  
class Kendaraan {  
    void jalankan() {
```

```
        print("Kendaraan sedang berjalan.");
    }

    void berhenti() {
        print("Kendaraan berhenti.");
    }
}

// Kelas Mobil mengimplementasikan interface Kendaraan
class Mobil implements Kendaraan {
    @override
    void jalankan() {
        print("Mobil melaju di jalan.");
    }

    @override
    void berhenti() {
        print("Mobil mengerem.");
    }

    void klakson() {
        print("Beep beep!");
    }
}

void main() {
    Mobil myCar = Mobil();
    myCar.jalankan(); // Output: Mobil melaju di jalan.
    myCar.berhenti(); // Output: Mobil mengerem.
    myCar.klakson(); // Output: Beep beep!

    // Polimorfisme dengan interface
    Kendaraan kendaraanUmum = Mobil();
    kendaraanUmum.jalankan(); // Output: Mobil melaju di jalan.
    // kendaraanUmum.klakson(); // ERROR: Metode klakson tidak ada di interface Kendaraan
}
```

Perbedaan dengan `extends` :

- `extends` : Mewarisi implementasi dan interface. Hanya bisa mewarisi dari satu kelas.
- `implements` : Hanya mewarisi interface (kontrak). Harus menyediakan implementasi sendiri untuk semua anggota interface. Bisa mengimplementasikan banyak interface.

## 11.2 Mixins

**Mixins** adalah cara untuk menggunakan kembali kode kelas dalam hierarki kelas. Mereka memungkinkan Anda untuk menambahkan fungsionalitas ke sebuah kelas tanpa menggunakan inheritance tradisional. Mixins sangat berguna ketika Anda memiliki fungsionalitas yang ingin Anda bagikan di antara beberapa kelas yang tidak memiliki hubungan inheritance yang kuat.

### Kapan Menggunakan Mixins?

- Ketika Anda ingin menambahkan perilaku tertentu ke beberapa kelas tanpa harus membuat hierarki inheritance yang kompleks.
- Untuk menghindari masalah *diamond problem* yang mungkin muncul dengan multiple inheritance.

Sintaks dasar:

```
● ● ●

 mixin  NamaMixin {
    // Properti dan metode yang akan ditambahkan
}

 class  NamaKelas  with  NamaMixin {
    // ...
}
```

Contoh:

```
● ● ●

 mixin  Terbang {
     void  terbang() {
        print("Sedang terbang di udara.");
    }
}

 mixin  Berenang {
     void  berenang() {
        print("Sedang berenang di air.");
    }
}

 class  Burung  with  Terbang {
     void  bersuara() {
        print("Chirp chirp!");
    }
}
```

```

    }
}

class Ikan with Berenang {
    void bersuara() {
        print("Blub blub!");
    }
}

class Bebek with Terbang, Berenang {
    void bersuara() {
        print("Quack quack!");
    }
}

void main() {
    Burung elang = Burung();
    elang.terbang();
    elang.bersuara();

    Ikan hiu = Ikan();
    hiu.berenang();
    hiu.bersuara();

    Bebek donald = Bebek();
    donald.terbang();
    donald.berenang();
    donald.bersuara();
}

```

**Catatan:** Mixin tidak dapat diinstansiasi secara langsung. Mereka hanya bisa digunakan dengan kata kunci `with` pada kelas lain.

### 11.3 Perbedaan antara Abstract Class, Interface, dan Mixin

Penting untuk memahami perbedaan dan kapan menggunakan masing-masing konsep ini:

Fitur	Abstract Class	Interface (Implicit)	Mixin
Tujuan	Menyediakan blueprint dasar, bisa punya implementasi sebagian.	Mendefinisikan kontrak yang harus dipatuhi kelas.	Menambahkan fungsionalitas (perilaku) ke kelas.

Fitur	Abstract Class	Interface (Implicit)	Mixin
<b>Instansiasi</b>	Tidak bisa diinstansiasi langsung.	Tidak bisa diinstansiasi langsung (karena kelas).	Tidak bisa diinstansiasi langsung.
<b>Inheritance</b>	Digunakan dengan <code>extends</code> . Hanya satu superclass.	Digunakan dengan <code>implements</code> . Bisa banyak interface.	Digunakan dengan <code>with</code> . Bisa banyak mixin.
<b>Implementasi</b>	Bisa punya metode abstrak (tanpa implementasi) dan konkret (dengan implementasi).	Semua metode harus diimplementasikan oleh kelas pengimplementasi.	Bisa punya implementasi penuh.
<b>Reusabilitas</b>	Reusabilitas kode melalui pewarisan hierarkis.	Reusabilitas tipe/kontrak.	Reusabilitas perilaku (code reuse).

### Ringkasan Penggunaan:

- Gunakan **Abstract Class** ketika Anda memiliki hierarki kelas yang kuat dan ingin menyediakan implementasi dasar serta memaksa subclass untuk mengimplementasikan metode tertentu.
- Gunakan **Interface** ketika Anda ingin mendefinisikan kontrak yang harus dipatuhi oleh kelas, tanpa memaksakan implementasi tertentu.
- Gunakan **Mixin** ketika Anda ingin menambahkan fungsionalitas tertentu ke beberapa kelas yang mungkin tidak memiliki hubungan inheritance yang sama, untuk menghindari duplikasi kode.

### Studi Kasus: Sistem Notifikasi Fleksibel

Buatlah sistem notifikasi yang dapat mengirim pesan melalui berbagai media (Email, SMS, Push Notification). Gunakan interface dan mixin untuk mencapai fleksibilitas.

```
// Interface untuk layanan notifikasi
abstract class NotifikasiService {
    void kirimNotifikasi(String penerima, String pesan);
}

// Implementasi layanan Email
class EmailService implements NotifikasiService {
    @override
    void kirimNotifikasi(String penerima, String pesan) {
        print("Mengirim email ke $penerima: \"$pesan\"");
    }
}
```

```
// Implementasi layanan SMS
class SmsService implements NotifikasiService {
    @override
    void kirimNotifikasi(String penerima, String pesan) {
        print("Mengirim SMS ke $penerima: \"$pesan\"");
    }
}

// Mixin untuk fungsionalitas logging notifikasi
 mixin NotifikasiLogger {
    void logNotifikasi(String penerima, String pesan) {
        print("[LOG] Notifikasi terkirim ke $penerima: \"$pesan\"");
    }
}

// Kelas yang menggunakan layanan notifikasi dan mixin logger
class SistemPeringatan with NotifikasiLogger {
    final NotifikasiService service;

    SistemPeringatan(this.service);

    void peringatkan(String penerima, String pesan) {
        service.kirimNotifikasi(penerima, pesan);
        logNotifikasi(penerima, pesan); // Menggunakan metode dari mixin
    }
}

void main() {
    // Menggunakan Email Service
    SistemPeringatan peringatanEmail = SistemPeringatan(EmailService());
    peringatanEmail.peringatkan("user@example.com", "Akun Anda telah diaktifkan.");

    print("\n");

    // Menggunakan SMS Service
    SistemPeringatan peringatanSms = SistemPeringatan(SmsService());
    peringatanSms.peringatkan("+628123456789", "Kode OTP Anda adalah 12345.");

    // Contoh lain: kelas yang mengimplementasikan NotifikasiService dan menggunakan NotifikasiLogger
    class PushNotificationService implements NotifikasiService, NotifikasiLogger {
        @override
        void kirimNotifikasi(String deviceToken, String pesan) {
            print("Mengirim Push Notification ke $deviceToken: \"$pesan\"");
        }
    }
}
```

```

        logNotifikasi(deviceToken, pesan); // Menggunakan metode dari mixin
    }

    @override
    void logNotifikasi(String penerima, String pesan) {
        // Implementasi khusus untuk push notification logging jika diperlukan
        print("[PUSH LOG] Notifikasi push terkirim ke $penerima: \"$pesan\"");
    }
}

print("\n");
PushNotificationService pushService = PushNotificationService();
pushService.kirimNotifikasi("device_token_xyz", "Ada pembaruan aplikasi!");
}

```

**Penjelasan:**

- `NotifikasiService` adalah `abstract class` yang berfungsi sebagai interface, mendefinisikan kontrak `kirimNotifikasi`.
- `EmailService` dan `SmsService` mengimplementasikan `NotifikasiService`, menyediakan cara spesifik untuk mengirim notifikasi.
- `NotifikasiLogger` adalah mixin yang menyediakan fungsionalitas logging yang dapat digunakan kembali oleh kelas mana pun yang membutuhkannya.
- `SistemPeringatan` menggunakan `NotifikasiLogger` dengan kata kunci `with`, dan menerima `NotifikasiService` melalui constructor, menunjukkan bagaimana kita bisa menggabungkan interface dan mixin.
- Kelas `PushNotificationService` menunjukkan bagaimana sebuah kelas bisa mengimplementasikan interface dan juga menggunakan mixin secara bersamaan.

**Latihan:**

1. Tambahkan layanan notifikasi baru, misalnya `WhatsAppService`, yang mengimplementasikan `NotifikasiService`.
2. Buat mixin baru, misalnya `AnalyticsTracker`, yang memiliki metode `trackEvent(String eventName, Map<String, dynamic> data)`. Terapkan mixin ini ke `SistemPeringatan` dan panggil `trackEvent` setelah setiap notifikasi terkirim.
3. Jelaskan dalam komentar kode, mengapa `NotifikasiService` dibuat sebagai `abstract class` dan bukan `mixin` atau kelas biasa.

- **Referensi:**

- [dart.dev/language/mixins](https://dart.dev/language/mixins)
- [dart.dev/language/class-modifiers#abstract-class](https://dart.dev/language/class-modifiers#abstract-class)

## Pertemuan 12: Asynchronous Programming - Future dan Async/Await

### Tujuan Pembelajaran

Setelah mempelajari materi ini, mahasiswa diharapkan mampu:

- Memahami konsep dasar *asynchronous programming* dan mengapa itu penting.
- Mengenal dan menggunakan objek `Future` untuk merepresentasikan hasil operasi asynchronous.
- Menggunakan kata kunci `async` untuk menandai fungsi asynchronous.
- Menggunakan kata kunci `await` untuk menunggu hasil dari sebuah `Future`.
- Menulis kode asynchronous yang bersih dan mudah dibaca.

### 12.1 Pengenalan Asynchronous Programming

Sebagian besar aplikasi modern perlu melakukan operasi yang memakan waktu, seperti mengambil data dari internet (API call), membaca file dari disk, atau melakukan komputasi yang kompleks. Jika operasi-operasi ini dilakukan secara *synchronous* (berurutan), aplikasi akan "hang" atau tidak responsif sampai operasi tersebut selesai. Ini akan memberikan pengalaman pengguna yang buruk.

**Asynchronous programming** adalah teknik yang memungkinkan program untuk memulai operasi yang memakan waktu tanpa harus menunggu operasi tersebut selesai. Program dapat terus menjalankan tugas lain, dan ketika operasi yang memakan waktu itu selesai, program akan diberitahu dan dapat memproses hasilnya. Ini membuat aplikasi tetap responsif dan lancar.

Dart, meskipun *single-threaded* (menjalankan satu operasi pada satu waktu), mendukung asynchronous programming dengan sangat baik melalui konsep `Future`, `async`, dan `await`.

### 12.2 `Future`

Sebuah `Future` adalah objek yang merepresentasikan hasil dari operasi asynchronous. Bayangkan `Future` sebagai janji bahwa Anda akan mendapatkan sebuah nilai (atau error) di masa depan. `Future` dapat berada dalam salah satu dari tiga state:

1. **Incomplete:** Operasi asynchronous belum selesai.
2. **Completed with a value:** Operasi asynchronous telah selesai dengan sukses dan menghasilkan sebuah nilai.
3. **Completed with an error:** Operasi asynchronous telah selesai tetapi mengalami error.

Ketika Anda memanggil fungsi yang mengembalikan `Future`, Anda tidak langsung mendapatkan nilainya. Sebaliknya, Anda mendapatkan objek `Future` yang akan "menyelesaikan" dirinya sendiri dengan nilai atau error di kemudian hari.

```
void main() {  
    print("Memulai program.");  
  
    // Memanggil fungsi yang mengembalikan Future  
    Future<String> hasilOperasi = ambilData();  
  
    // Melakukan sesuatu yang lain sementara data diambil  
    print("Melanjutkan tugas lain...");  
  
    // Ketika Future selesai, kita bisa memproses hasilnya  
    hasilOperasi.then((data) {  
        print("Data diterima: $data");  
    }).catchError((error) {  
        print("Terjadi error: $error");  
    }).whenComplete(() {  
        print("Operasi pengambilan data selesai (baik sukses maupun gagal).");  
    });  
  
    print("Program selesai (tapi operasi asynchronous masih berjalan).");  
}  
  
Future<String> ambilData() {  
    // Simulasi operasi yang memakan waktu (misal: HTTP request ke server)  
    return Future.delayed(Duration(seconds: 3), () {  
        // Setelah 3 detik, Future akan selesai dengan nilai ini  
        return "Data dari server";  
    });  
}
```

### Penjelasan:

- `Future.delayed()` adalah cara untuk membuat `Future` yang selesai setelah durasi tertentu. Ini sering digunakan untuk mensimulasikan operasi jaringan atau I/O.
- `.then()` : Dipanggil ketika `Future` selesai dengan sukses. Argumennya adalah fungsi yang menerima nilai hasil.
- `.catchError()` : Dipanggil jika `Future` selesai dengan error. Argumennya adalah fungsi yang menerima objek error.
- `.whenComplete()` : Dipanggil ketika `Future` selesai, baik dengan nilai maupun dengan error. Berguna untuk membersihkan sumber daya.

### 12.3 `async` dan `await`

Meskipun `.then()`, `.catchError()`, dan `.whenComplete()` bekerja dengan baik, Dart menyediakan sintaks yang lebih bersih dan mudah dibaca untuk menangani `Future` menggunakan kata kunci `async` dan `await`.

- `async`: Digunakan untuk menandai sebuah fungsi sebagai asynchronous. Fungsi yang ditandai dengan `async` akan selalu mengembalikan sebuah `Future`.
- `await`: Digunakan di dalam fungsi `async` untuk "menunggu" `Future` selesai dieksekusi. Ketika `await` digunakan, eksekusi fungsi `async` akan dijeda sampai `Future` yang ditunggu selesai. Namun, ini tidak memblokir *thread* utama aplikasi; Dart akan menjalankan tugas lain sementara menunggu.

#### Contoh Penggunaan `async` dan `await`

```
void main() async { // Tandai main() sebagai async
    print("Memulai program.");

    // Memanggil fungsi asynchronous dan menunggu hasilnya
    // Eksekusi akan dijeda di sini sampai ambilData() selesai
    String data = await ambilData();
    print("Data diterima: $data");

    print("Program selesai.");
}

Future<String> ambilData() {
    return Future.delayed(Duration(seconds: 3), () {
        return "Data dari server";
    });
}
```

#### Perbandingan dengan `.then()`:

Kode dengan `async` / `await` terlihat lebih linear dan mirip dengan kode synchronous, membuatnya lebih mudah dipahami, terutama untuk operasi asynchronous yang berantai.

#### Menangani Error dengan `async` / `await`

Untuk menangani error dalam fungsi `async` / `await`, Anda bisa menggunakan blok `try-catch` seperti yang biasa Anda lakukan untuk kode synchronous.

```
void main() async {
    print("Memulai program dengan error handling.");

    try {
        String data = await ambilDataYangMungkinGagal(true);
        print("Data berhasil diambil: $data");
    } catch (e) {
        print("Terjadi error: $e");
    }

    print("\nCoba lagi dengan sukses...");
    try {
        String data = await ambilDataYangMungkinGagal(false);
        print("Data berhasil diambil: $data");
    } catch (e) {
        print("Terjadi error: $e");
    }

    print("Program selesai.");
}

Future<String> ambilDataYangMungkinGagal(bool isGagal) {
    return Future.delayed(Duration(seconds: 2), () {
        if (isGagal) {
            throw Exception("Gagal mengambil data dari server!");
        } else {
            return "Data sukses dari server";
        }
    });
}
```

### Studi Kasus: Mengambil Data Pengguna dari API Simulasi

Mari kita simulasikan pengambilan data pengguna dari sebuah API. Kita akan membuat fungsi asynchronous yang mengembalikan data pengguna setelah beberapa detik.

```
import 'dart:convert'; // Untuk mengkonversi JSON

void main() async {
```

```
print("Aplikasi dimulai: Mengambil data pengguna...");  
  
try {  
    // Panggil fungsi untuk mengambil data pengguna  
    Map<String, dynamic> userData = await fetchUserData(1);  
    print("\nData Pengguna Berhasil Diambil:");  
    print("ID: ${userData['id']}");  
    print("Nama: ${userData['name']}");  
    print("Email: ${userData['email']}");  
    print("Username: ${userData['username']}");  
  
    // Coba ambil data pengguna yang tidak ada  
    print("\nMencoba mengambil data pengguna dengan ID 99...");  
    Map<String, dynamic> userDataNotFound = await fetchUserData(99);  
    print("Data Pengguna Berhasil Diambil: $userDataNotFound"); // Ini tidak akan terjadi  
  
} catch (e) {  
    print("\nTerjadi Kesalahan: $e");  
}  
  
print("\nAplikasi selesai.");  
}  
  
// Fungsi asynchronous untuk mensimulasikan pengambilan data pengguna dari API  
Future<Map<String, dynamic>> fetchUserData(int userId) async {  
    print("Mengambil data untuk pengguna ID: $userId...");  
    // Simulasi penundaan jaringan  
    await Future.delayed(Duration(seconds: 2));  
  
    // Data simulasi  
    final Map<int, Map<String, dynamic>> mockApiData = {  
        1: {  
            'id': 1,  
            'name': 'Leanne Graham',  
            'username': 'Bret',  
            'email': 'Sincere@april.biz'  
        },  
        2: {  
            'id': 2,  
            'name': 'Ervin Howell',  
            'username': 'Antonette',  
            'email': 'Shanna@melissa.tv'  
        },  
    };  
}
```

```
if (mockApiData.containsKey(userId)) {  
    print("Data ditemukan untuk ID: $userId");  
    return mockApiData[userId]!;  
} else {  
    throw Exception('Pengguna dengan ID $userId tidak ditemukan.');//  
}  
}
```

### Penjelasan:

- Fungsi `fetchUserData` adalah `async` dan mengembalikan `Future<Map<String, dynamic>>`.
- Di dalamnya, `await Future.delayed` mensimulasikan waktu yang dibutuhkan untuk melakukan permintaan jaringan.
- Jika `userId` ditemukan dalam `mockApiData`, data dikembalikan. Jika tidak, sebuah `Exception` dilemparkan.
- Di fungsi `main`, kita menggunakan `try-catch` untuk memanggil `fetchUserData`. Jika `fetchUserData` berhasil, data dicetak. Jika `Exception` dilemparkan, blok `catch` akan menangkapnya dan mencetak pesan error.

### Latihan:

1. Modifikasi `fetchUserData` agar juga mensimulasikan kegagalan jaringan (misalnya, melempar error jika `userId` adalah 0).
2. Buat fungsi `uploadFile()` yang mengembalikan `Future<bool>`. Fungsi ini mensimulasikan proses upload file yang memakan waktu 5 detik dan mengembalikan `true` jika sukses, `false` jika gagal. Panggil fungsi ini di `main()` dan tangani hasilnya.
3. Pelajari tentang `Future.wait()` untuk menjalankan beberapa `Future` secara paralel dan menunggu semuanya selesai.

### • Referensi:

- [dart.dev/codelabs/async-await](https://dart.dev/codelabs/async-await)
- [dart.dev/language/futures](https://dart.dev/language/futures)

## Pertemuan 13: Asynchronous Programming - Stream dan Error Handling

### Tujuan Pembelajaran

Setelah mempelajari materi ini, mahasiswa diharapkan mampu:

- Memahami konsep `Stream` sebagai urutan peristiwa asynchronous.
- Membedakan antara `Future` dan `Stream`.
- Menggunakan `Stream` untuk menangani data yang berkelanjutan.
- Menerapkan strategi penanganan error yang efektif menggunakan `try-catch-finally` untuk operasi synchronous dan asynchronous.

### 13.1 Stream

Pada pertemuan sebelumnya, kita belajar tentang `Future`, yang merepresentasikan satu nilai yang akan tersedia di masa depan. Namun, bagaimana jika kita perlu menangani serangkaian nilai yang datang dari waktu ke waktu? Di sinilah `Stream` berperan.

`Stream` adalah urutan peristiwa asynchronous. Bayangkan `Stream` sebagai pipa air yang terus-menerus mengalirkan air (data) dari waktu ke waktu. `Stream` dapat menghasilkan nol atau lebih peristiwa (data atau error) dari waktu ke waktu. Ini sangat cocok untuk menangani data yang terus-menerus datang, seperti:

- Event UI (klik tombol, input teks).
- Data dari sensor (misalnya, lokasi GPS).
- Data dari jaringan (misalnya, WebSocket).
- Membaca file besar secara bertahap.

#### Perbedaan `Future` vs `Stream`

Fitur	<code>Future</code>	<code>Stream</code>
<b>Jumlah Nilai</b>	Satu nilai tunggal	Nol atau lebih nilai
<b>Kapan Selesai</b>	Selesai setelah satu nilai tersedia	Dapat terus menghasilkan nilai dari waktu ke waktu
<b>Contoh</b>	Hasil HTTP request, membaca satu file	Event klik, data sensor, WebSocket

### Membuat dan Menggunakan Stream

Ada beberapa cara untuk membuat `Stream`. Salah satu cara yang umum adalah menggunakan fungsi `async*` (async generator) dengan kata kunci `yield`.



```
void main() async {
    print("Memulai mendengarkan stream angka...");
    // Mendengarkan event dari stream
    // await for digunakan untuk mengiterasi event dari stream secara asynchronous
    await for (int angka in hitungMundur(5)) {
        print("Angka yang diterima: $angka");
    }

    print("Stream angka selesai.");
    print("\nMemulai mendengarkan stream pesan...");
    // Contoh lain dengan listen()
    pesanStream().listen(
        (pesan) => print("Pesanan diterima: $pesan"),
        onError: (error) => print("Error pada stream pesan: $error"),
        onDone: () => print("Stream pesan selesai."),
        cancelOnError: true, // Jika terjadi error, stream akan di-cancel
    );
}

// Fungsi async generator yang mengembalikan Stream<int>
Stream<int> hitungMundur(int dariAngka) async* {
    for (int i = dariAngka; i >= 0; i--) {
        await Future.delayed(Duration(seconds: 1)); // Simulasi penundaan
        yield i; // Mengirimkan nilai ke stream
    }
}

// Fungsi yang mengembalikan Stream<String>
Stream<String> pesanStream() {
    return Stream.fromIterable([
        "Halo",
        "Apa kabar?",
        "Semoga sehat selalu!",
    ]);
}
```

**Penjelasan:**

- `async*` : Menandai fungsi sebagai *async generator*, yang berarti ia akan mengembalikan `Stream`.
- `yield` : Digunakan di dalam fungsi `async*` untuk mengirimkan nilai ke `Stream`.
- `await for` : Cara yang nyaman untuk mengonsumsi `Stream` dengan mengiterasi setiap event yang datang.
- `.listen()` : Metode lain untuk mengonsumsi `Stream`. Ia menerima fungsi callback untuk setiap data, error, dan ketika `Stream` selesai.

## 13.2 Error Handling

Penanganan error adalah bagian krusial dari setiap aplikasi yang robust. Dart menyediakan mekanisme yang kuat untuk menangani exception (kesalahan) yang terjadi selama eksekusi program. Exception adalah peristiwa yang mengganggu alur normal program.

### 13.2.1 `try-catch-finally` untuk Synchronous Code

Blok `try-catch-finally` digunakan untuk menangani exception. Ini memungkinkan Anda untuk mencoba menjalankan kode yang mungkin menimbulkan error, menangkap error jika terjadi, dan menjalankan kode pembersihan terlepas dari hasilnya.

- `try` : Blok kode yang mungkin menimbulkan exception.
- `catch` : Blok kode yang dieksekusi jika exception terjadi di blok `try`. Anda dapat menangkap exception tertentu atau semua exception.
- `on` : Digunakan dengan `catch` untuk menangkap exception dengan tipe tertentu (misalnya `FormatException`, `StateError`).
- `finally` : Blok kode yang selalu dieksekusi, terlepas dari apakah exception terjadi atau tidak. Berguna untuk membersihkan sumber daya (misalnya, menutup file, koneksi database).

```
● ○ ●
void main() {
  print("Memulai operasi synchronous dengan error handling...");

  // Contoh 1: Pembagian dengan nol
  try {
    int hasil = 10 ~/ 0; // Ini akan melempar IntegerDivisionByZeroException
    print("Hasil pembagian: $hasil");
  } on IntegerDivisionByZeroException {
    print("Terjadi kesalahan: Tidak bisa membagi dengan nol!");
  } catch (e) {
    print("Terjadi exception lain: $e");
  } finally {
    print("Blok finally untuk contoh 1 selalu dieksekusi.");
}
```

```
}

print("\n");

// Contoh 2: Konversi string ke int yang gagal
try {
    String teksAngka = "abc";
    int angka = int.parse(teksAngka); // Ini akan melempar FormatException
    print("Angka hasil konversi: $angka");
} on FormatException catch (e) {
    print("Terjadi FormatException: ${e.message}");
} catch (e, s) { // Menangkap semua exception lain dan stack trace
    print("Terjadi exception umum: $e");
    print("Stack trace: $s");
} finally {
    print("Blok finally untuk contoh 2 selalu dieksekusi.");
}

print("\nProgram selesai.");
}
```

### 13.2.2 `throw` Keyword

Anda dapat secara eksplisit melempar (throw) exception menggunakan kata kunci `throw`. Ini berguna ketika Anda mendeteksi kondisi error yang tidak dapat ditangani oleh fungsi saat ini.

```
● ● ●

void cekUmur(int umur) {
    if (umur < 0) {
        throw ArgumentError("Umur tidak boleh negatif!");
    } else if (umur < 18) {
        throw Exception("Anda harus berusia minimal 18 tahun.");
    }
    print("Umur valid: $umur");
}

void main() {
    try {
        cekUmur(20);
        cekUmur(15);
    } on ArgumentError catch (e) {
        print("Error Argumen: ${e.message}");
    }
}
```

```

    } catch (e) {
        print("Error Umum: $e");
    }
}

```

### 13.2.3 Error Handling untuk Asynchronous Code (`Future` dan `Stream`)

Untuk `Future`, Anda bisa menggunakan `try-catch` dengan `async` / `await` seperti yang sudah dibahas di Pertemuan 12. Ini adalah cara yang paling direkomendasikan.

Untuk `Stream`, Anda bisa menangani error di dalam metode `.listen()` menggunakan parameter `onError`, atau menggunakan `try-catch` dengan `await for`.



```

void main() async {
    print("\nMenangani error pada Stream dengan await for...");
    try {
        await for (String data in dataStreamDenganError()) {
            print("Data dari stream: $data");
        }
    } catch (e) {
        print("Terjadi error saat mengonsumsi stream: $e");
    }

    print("\nMenangani error pada Stream dengan .listen()...");
    dataStreamDenganError().listen(
        (data) => print("Data diterima: $data"),
        onError: (error) => print("Error di .listen(): $error"),
        onDone: () => print("Stream selesai di .listen()."),
    );
}

Stream<String> dataStreamDenganError() async* {
    yield "Data 1";
    await Future.delayed(Duration(seconds: 1));
    yield "Data 2";
    await Future.delayed(Duration(seconds: 1));
    throw Exception("Koneksi terputus!"); // Melempar error
    // yield "Data 3"; // Kode ini tidak akan tercapai
}

```

## Studi Kasus: Pembacaan Data Log File

Simulasikan pembacaan data dari sebuah log file yang mungkin berisi baris yang tidak valid atau error. Program akan membaca baris demi baris dan memprosesnya, sambil menangani error yang mungkin terjadi.

```
import 'dart:async';

void main() async {
    print("\n--- Memulai Pembacaan Log File --- ");

    Stream<String> logStream = getLogEntries();

    int barisValid = 0;
    int barisError = 0;

    try {
        await for (String entry in logStream) {
            try {
                // Simulasi pemrosesan setiap baris log
                if (entry.contains("ERROR")) {
                    throw FormatException("Baris log mengandung ERROR: $entry");
                }
                print("Memproses: $entry");
                barisValid++;
            } on FormatException catch (e) {
                print("SKIP (Format Error): ${e.message}");
                barisError++;
            } catch (e) {
                print("SKIP (Unknown Error): $e");
                barisError++;
            }
        }
    } catch (e) {
        print("\nFatal Error saat membaca stream log: $e");
    } finally {
        print("\n--- Ringkasan Pembacaan Log ---");
        print("Total baris valid: $barisValid");
        print("Total baris error: $barisError");
        print("Pembacaan log selesai.");
    }
}
```

```
// Simulasi Stream dari entri log file
Stream<String> getLogEntries() async* {
  List<String> logData = [
    "[INFO] User logged in.",
    "[DEBUG] Data fetched.",
    "[ERROR] Database connection failed.",
    "[INFO] Item added to cart.",
    "[WARNING] Disk space low.",
    "[CRITICAL] Invalid data format received.",
    "[INFO] Report generated.",
  ];

  for (String entry in logData) {
    await Future.delayed(Duration(milliseconds: 500)); // Simulasi baca per baris
    yield entry;
  }
}
```

**Penjelasan:**

- Fungsi `getLogEntries()` adalah `async*` yang mensimulasikan pembacaan baris-baris log dari sebuah sumber (misalkan file). Setiap baris `yield` ke stream.
- Di `main()`, kita menggunakan `await for` untuk mengonsumsi stream log.
- Di dalam loop `await for`, setiap `entry` log diproses dalam blok `try-catch` tersendiri.
- Jika sebuah baris mengandung "ERROR", kita secara eksplisit melempar `FormatException` untuk mensimulasikan baris log yang tidak valid.
- Blok `catch` menangani `FormatException` dan exception lainnya, mencatatnya sebagai baris error dan melanjutkan ke baris berikutnya (`continue` tidak diperlukan di sini karena `try-catch` sudah mengelola alur).
- Blok `finally` di `main()` akan selalu dieksekusi setelah stream selesai atau jika ada error fatal pada stream itu sendiri, memberikan ringkasan.

**Latihan:**

1. Modifikasi `getLogEntries()` untuk secara acak melempar `Exception` (bukan `FormatException`) pada beberapa baris untuk menguji penanganan error umum.
2. Tambahkan fitur ke studi kasus agar program dapat menyimpan baris-baris log yang error ke dalam sebuah `List<String>` terpisah, lalu tampilkan `List` tersebut di akhir.
3. Buat sebuah `Stream` yang menghasilkan angka acak setiap detik, dan gunakan `listen()` untuk mencetak angka tersebut. Hentikan stream setelah 10 angka atau jika angka yang dihasilkan lebih dari 0.9 (gunakan `StreamSubscription.cancel()` ).

- **Referensi:**

- [dart.dev/languagestreams](https://dart.dev/languagestreams)
- [dart.dev/language/error-handling](https://dart.dev/language/error-handling)

## Pertemuan 14: Manajemen Paket dengan Pub dan Persiapan Flutter

### Tujuan Pembelajaran

Setelah mempelajari materi ini, mahasiswa diharapkan mampu:

- Memahami peran dan fungsi Pub sebagai manajer paket Dart.
- Mengelola dependensi proyek Dart menggunakan file `pubspec.yaml`.
- Menambah, memperbarui, dan menggunakan paket eksternal dari `pub.dev`.
- Mendapatkan gambaran umum tentang Flutter dan keterkaitannya dengan Dart.
- Memahami langkah-langkah awal untuk memulai pengembangan dengan Flutter.

### 14.1 Manajemen Paket dengan Pub

**Pub** adalah sistem manajemen paket resmi untuk bahasa pemrograman Dart. Ini mirip dengan `npm` di Node.js, `pip` di Python, atau `Maven` / `Gradle` di Java. Pub memungkinkan Anda untuk dengan mudah menemukan, menginstal, dan mengelola *paket* (libraries atau modul kode yang ditulis oleh orang lain) yang dapat Anda gunakan dalam proyek Dart Anda.

#### 14.1.1 `pubspec.yaml`

Setiap proyek Dart (termasuk proyek Flutter) memiliki file konfigurasi bernama `pubspec.yaml` di direktori root-nya. File ini adalah jantung dari manajemen paket proyek Anda. Ini berisi metadata tentang proyek Anda dan, yang paling penting, daftar dependensi (paket eksternal) yang dibutuhkan proyek Anda.

Contoh struktur `pubspec.yaml`:

```
description: A sample Dart project. # Deskripsi singkat
version: 1.0.0+1 # Versi aplikasi (major.minor.patch+build_number)

environment:
  sdk: ">=3.0.0 <4.0.0" # Versi SDK Dart yang kompatibel

dependencies:
  # Daftar paket yang dibutuhkan aplikasi Anda untuk berjalan
  http: ^1.1.0 # Contoh: untuk melakukan HTTP requests
  path: ^1.8.3 # Contoh: untuk manipulasi path file

dev_dependencies:
  # Daftar paket yang hanya dibutuhkan selama pengembangan atau pengujian
```

```

lints: ^2.1.1 # Contoh: untuk analisis kode statis
test: ^1.24.0 # Contoh: untuk unit testing

# flutter:
# uses-material-design: true
# assets:
#   - images/a_dot_burr.jpeg
#   - images/a_dot_ham.jpeg

```

### Penjelasan Bagian Penting:

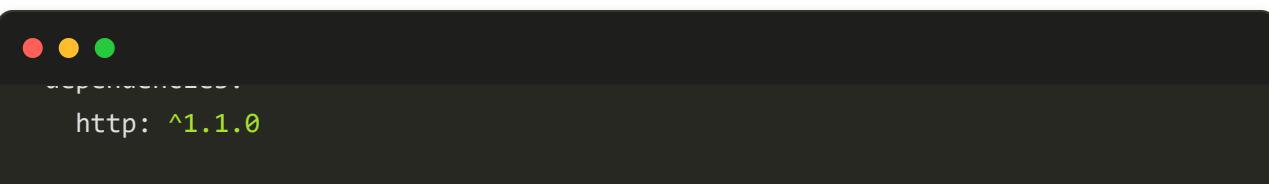
- `name` : Nama proyek Anda. Harus berupa identifier Dart yang valid.
- `description` : Deskripsi singkat proyek.
- `version` : Versi proyek Anda. Format `major.minor.patch+build_number` adalah standar.
- `environment` : Menentukan rentang versi Dart SDK yang kompatibel dengan proyek Anda.
- `dependencies` : Bagian ini berisi daftar paket yang dibutuhkan oleh aplikasi Anda untuk berfungsi. Ketika Anda menyebarluaskan aplikasi Anda, paket-paket ini akan disertakan.
- `dev_dependencies` : Bagian ini berisi daftar paket yang hanya dibutuhkan selama pengembangan atau pengujian, dan tidak akan disertakan dalam aplikasi yang sudah jadi (misalnya, alat testing, linter).

#### 14.1.2 Menambah dan Menggunakan Paket Eksternal

Untuk menggunakan paket eksternal dalam proyek Dart Anda, ikuti langkah-langkah berikut:

1. **Cari Paket:** Kunjungi [pub.dev](https://pub.dev), repositori pusat untuk paket-paket Dart dan Flutter. Cari paket yang Anda butuhkan (misalnya, `http` untuk HTTP requests, `shared_preferences` untuk menyimpan data lokal).
2. **Tambahkan ke `pubspec.yaml`:** Setelah menemukan paket, di halaman paket tersebut akan ada instruksi instalasi. Biasanya, Anda hanya perlu menyalin baris dependensi dan menempelkannya di bawah bagian `dependencies` atau `dev_dependencies` di file `pubspec.yaml` proyek Anda.

Misalnya, untuk menambahkan paket `http` :



3. **Jalankan `pub get`:** Setelah memodifikasi `pubspec.yaml`, buka terminal di direktori root proyek Anda dan jalankan perintah:

```
pub get
```

Perintah ini akan mengunduh semua dependensi yang terdaftar di `pubspec.yaml` dan menyimpannya di cache lokal Anda. Pub juga akan membuat file `pubspec.lock` yang mengunci versi spesifik dari setiap dependensi untuk memastikan build yang konsisten.

1. **Gunakan dalam Kode:** Setelah paket diunduh, Anda dapat mengimpornya ke dalam file Dart Anda menggunakan pernyataan `import`.

```
// main.dart
import 'package:http/http.dart' as http; // Mengimpor paket http

void main() async {
    var url = Uri.https('jsonplaceholder.typicode.com', '/todos/1');
    try {
        var response = await http.get(url);
        if (response.statusCode == 200) {
            print('Response body: ${response.body}');
        } else {
            print('Request failed with status: ${response.statusCode}.' );
        }
    } catch (e) {
        print('Error fetching data: $e');
    }
}
```

Dalam contoh di atas, kita mengimpor paket `http` dan menggunakannya untuk membuat permintaan HTTP GET ke sebuah API publik. Ini menunjukkan bagaimana mudahnya mengintegrasikan fungsionalitas eksternal menggunakan Pub.

## 14.2 Pengantar Flutter

Setelah Anda memiliki pemahaman yang kuat tentang Dart, Anda siap untuk melangkah ke **Flutter**.

**Flutter** adalah UI toolkit dari Google untuk membangun aplikasi yang indah, dikompilasi secara native, dan multi-platform dari satu codebase. Dengan Flutter, Anda dapat membangun aplikasi untuk:

- Mobile (Android & iOS)
- Web
- Desktop (Windows, macOS, Linux)
- Embedded devices

#### 14.2.1 Mengapa Flutter?

- **Pengembangan Cepat:** Fitur *Hot Reload* dan *Hot Restart* memungkinkan Anda melihat perubahan kode secara instan.
- **UI yang Ekspresif dan Fleksibel:** Flutter menggunakan widget sebagai blok bangunan UI. Semuanya adalah widget, dari teks hingga tata letak. Ini memungkinkan desain UI yang sangat kustom dan indah.
- **Performa Native:** Aplikasi Flutter dikompilasi langsung ke kode ARM native untuk mobile, atau JavaScript untuk web, atau kode mesin untuk desktop, memastikan performa yang cepat dan lancar.
- **Satu Codebase:** Tulis kode sekali, jalankan di mana saja.
- **Didukung oleh Google:** Dukungan kuat dari Google dan komunitas yang berkembang pesat.

#### 14.2.2 Keterkaitan Dart dan Flutter

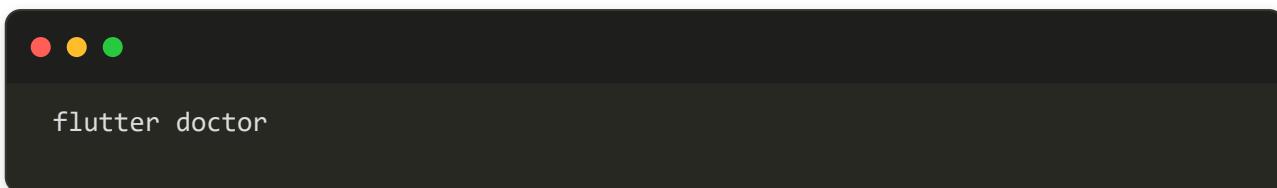
Dart adalah bahasa pemrograman yang digunakan oleh Flutter. Flutter dibangun di atas Dart. Semua kode UI, logika bisnis, dan interaksi di aplikasi Flutter Anda akan ditulis dalam Dart. Oleh karena itu, pemahaman yang kuat tentang Dart adalah prasyarat mutlak untuk menjadi pengembang Flutter yang efektif.

Dart menyediakan fitur-fitur yang sangat cocok untuk Flutter:

- **Kompilasi AOT:** Untuk performa produksi yang cepat.
- **Kompilasi JIT:** Untuk pengembangan yang cepat dengan Hot Reload.
- **Null Safety:** Untuk kode yang lebih stabil dan bebas bug.
- **Asynchronous Programming:** Penting untuk operasi jaringan dan I/O di aplikasi mobile.
- **Ekosistem Pub:** Memungkinkan Flutter untuk dengan mudah mengintegrasikan berbagai library dan alat.

#### 14.2.3 Langkah-langkah Awal untuk Memulai dengan Flutter

1. **Instal Flutter SDK:** Kunjungi [flutter.dev/docs/get-started/install](https://flutter.dev/docs/get-started/install) dan ikuti instruksi instalasi untuk sistem operasi Anda. Ini akan menginstal Flutter CLI (Command Line Interface) dan Dart SDK yang sudah terbundle.
2. **Verifikasi Instalasi:** Jalankan `flutter doctor` di terminal. Perintah ini akan memeriksa lingkungan Anda dan melaporkan komponen apa pun yang perlu diinstal atau dikonfigurasi.



3. **Buat Proyek Flutter Baru:**

```
flutter create my_first_flutter_app  
cd my_first_flutter_app
```

#### 4. Jalankan Aplikasi:

```
flutter run
```

Ini akan menjalankan aplikasi Flutter default di emulator/simulator atau perangkat fisik yang terhubung.

#### Studi Kasus: Membuat Aplikasi Konsol Sederhana dengan Dependensi Eksternal

Mari kita buat aplikasi konsol sederhana yang mengambil daftar tugas (todos) dari sebuah API publik menggunakan paket `http`.

```
import 'dart:convert'; // Untuk mengurai JSON  
import 'package:http/http.dart' as http; // Impor paket http  
  
void main() async {  
    print('--- Aplikasi Pengambil Daftar Tugas ---');  
  
    try {  
        List<Todo> todos = await fetchTodos();  
  
        if (todos.isNotEmpty) {  
            print('\nDaftar Tugas:');  
            for (var todo in todos) {  
                print('- ${todo.title} [Selesai: ${todo.completed ? "Ya" : "Tidak"}]');  
            }  
        } else {  
            print('Tidak ada tugas yang ditemukan.');        }  
    } catch (e) {  
        print('Terjadi kesalahan: $e');  
    }  
  
    print('\nAplikasi selesai.');
```

```
}

// Kelas model untuk Todo
class Todo {
    final int userId;
    final int id;
    final String title;
    final bool completed;

    Todo({
        required this.userId,
        required this.id,
        required this.title,
        required this.completed,
    });

    // Factory constructor untuk membuat objek Todo dari JSON Map
    factory Todo.fromJson(Map<String, dynamic> json) {
        return Todo(
            userId: json['userId'] as int,
            id: json['id'] as int,
            title: json['title'] as String,
            completed: json['completed'] as bool,
        );
    }
}

// Fungsi asynchronous untuk mengambil daftar tugas dari API
Future<List<Todo>> fetchTodos() async {
    print('Mengambil tugas dari API...');
    final response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/todos'));

    if (response.statusCode == 200) {
        // Jika server mengembalikan respons OK (status code 200),
        // maka parse JSON.
        List<dynamic> jsonList = jsonDecode(response.body);
        return jsonList.map((json) => Todo.fromJson(json)).toList();
    } else {
        // Jika respons tidak OK, lempar exception.
        throw Exception('Gagal memuat tugas. Status code: ${response.statusCode}');
    }
}
```

**Langkah-langkah Menjalankan Studi Kasus:****1. Buat Proyek Dart Baru:**

```
mkdir todo_app
cd todo_app
dart create console
```

**2. Tambahkan Dependensi http :** Buka file `pubspec.yaml` di dalam folder `todo_app` dan tambahkan `http` di bawah `dependencies:` :

```
http: ^1.1.0 # Pastikan versi sesuai dengan yang terbaru di pub.dev
```

**3. Dapatkan Dependensi:** Jalankan perintah ini di terminal di dalam folder `todo_app` :

```
dart pub get
```

**4. Salin Kode:** Ganti isi file `bin/todo_app.dart` (atau `bin/<nama_proyek_anda>.dart`) dengan kode di atas.**5. Jalankan Aplikasi:**

```
dart run
```

**Penjelasan Kode:**

- Kita mengimpor `dart:convert` untuk mengurai data JSON dan `package:http/http.dart` untuk melakukan permintaan HTTP.
- Kelas `Todo` adalah model data yang merepresentasikan sebuah tugas, dengan `factory Todo.fromJson` untuk memudahkan konversi dari JSON ke objek Dart.
- Fungsi `fetchTodos()` adalah `async` yang menggunakan `http.get()` untuk mengambil data dari API. Ia menangani respons dan melempar error jika status code bukan 200.

- Di `main()`, kita memanggil `fetchTodos()` dan mencetak daftar tugas yang berhasil diambil, atau menangani error jika terjadi.

**Latihan:**

1. Modifikasi studi kasus di atas untuk hanya menampilkan tugas yang `completed` bernilai `false` (tugas yang belum selesai).
2. Tambahkan fitur untuk mengambil detail satu tugas berdasarkan ID yang dimasukkan pengguna. (API endpoint: [https://jsonplaceholder.typicode.com/todos/ID\\_TUGAS](https://jsonplaceholder.typicode.com/todos/ID_TUGAS) ).
3. Coba gunakan paket lain dari [pub.dev](#) (misalnya `logger`) untuk logging yang lebih canggih) dan integrasikan ke dalam proyek ini.

**• Referensi:**

- [dart.dev/tools/pub/cmd](#)
- [flutter.dev/docs/get-started/install](#)